utbm

université de technologie
Belfort-Montbéliard

Compilation and Language Theory

14th revision

Stéphane GALLAND

Chapter 0
Overview of the module DA53

Stéphane GALLAND

1. Study models, techniques and algorithms that permit to analyze a text-based language
2. Study models, techniques and algorithms that permit to generate and execute code
3. Study the techniques for the optimization of executable codes (available soon)

Languages

- Java — tutorials and projects
- C/C++/C# — projects

Integrated Development Environment

- Eclipse — tutorials and projects
- NetBean, IntelliJ, Visual Studio — projects

Compilation Tools

- javacc — tutorials, projects
- Xtext — projects
- jlex, lex, flex, yacc, bison — projects

Lectures

Supervised tutorials [ ]

Laboratory works [ ]

Exams

Mid-Term Exam 30%

Final Exam 30%

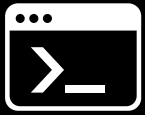
Laboratory works 40%

Project — see DA50

Object Oriented Programming

Java, C#, C++, Python

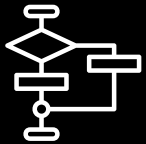
1



CLI Compilation

Java, g++, Maven, Make

2

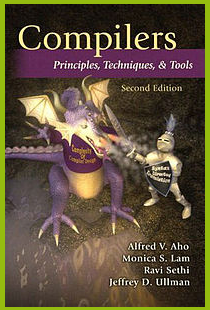


Algorithms

Statements, Data structures

3

1. Download the PDF files of the slides before the lecture
2. Do not read each word of the slides during the lectures
3. Listen carefully the teachers and takes notes on the side of the slides
4. Ask questions . . . Ask questions . . . Ask questions
5. You must read the slides at home as soon as possible, not few hours before the exams



Compilers — Principles, Techniques and Tools Second Edition
2nd edition

Alfred V. AHO, Monica S. LAM, Ravi SETHI and Jeffrey D. ULLMAN

Pearson & Addison Wesley, 2007

ISBN 0-321-48681-1

Parsing Techniques — A Practical Guide

Dick Grune and Ceriel J.H. Jacobs

Springer Verlag New York, 2007

ISBN 0-387-20248-X

Calculabilité, Complexité et Approximation

Jean-François REY

Vuibert, France, 2004

ISBN 2-7117-4808-1

Chapter 1
Overview of the Compilation Theory

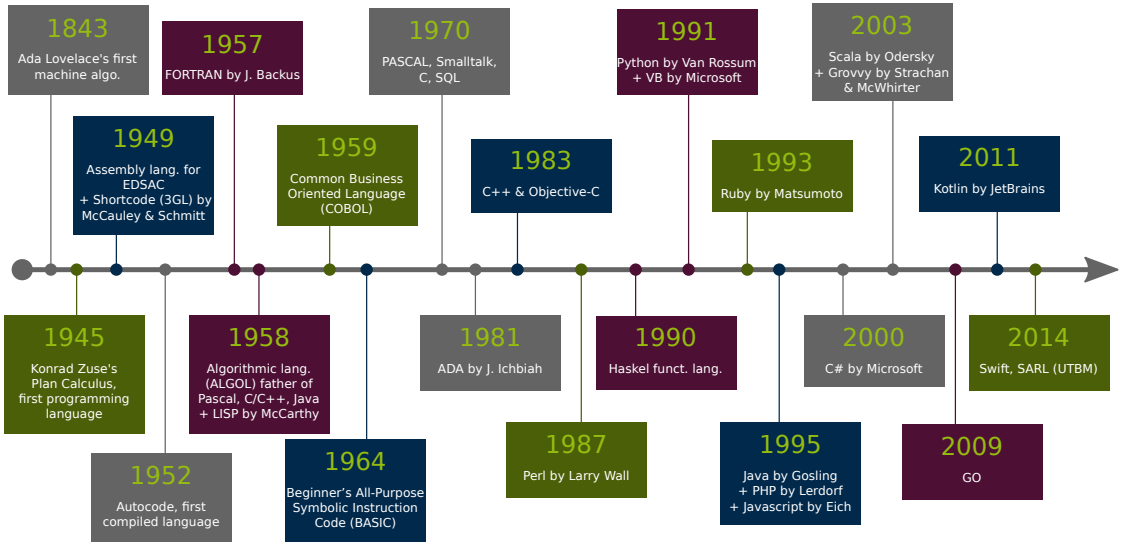
Stéphane GALLAND

- Programming languages are notations for describing computations to people and to machines
- All the software running on all the computers was written in some programming language
- Before a program can be run, it first must be translated into a form in which it can be executed by a computer
- The software systems that do this translation are called a compiler

Overview of the principles, architecture and implementation of a simple compiler

With this chapter, you may understand the key points of language theory

BRIEF HISTORY OF MAIN PROGRAMMING LANGUAGES



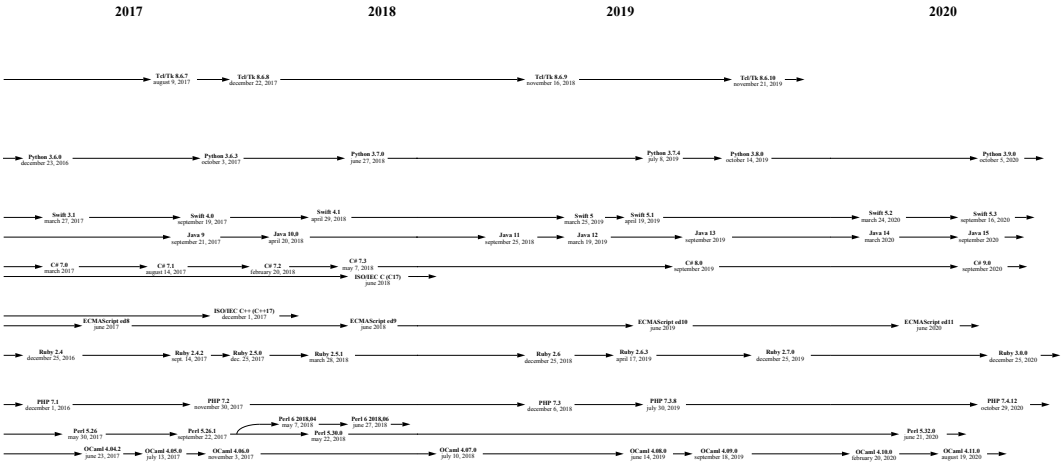
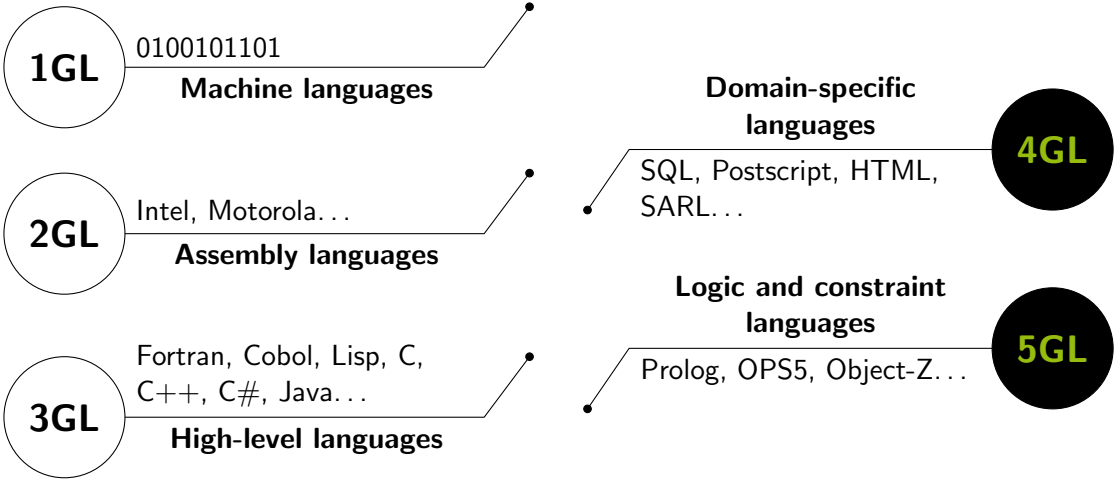
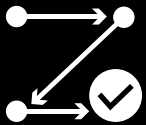


Figure by Éric Lévénez — `www.levenez.com/lang`


1GL
0100101101
Machine languages
2GL
Intel, Motorola...
Assembly languages
3GL
Fortran, Cobol, Lisp, C,
C++, C#, Java...
High-level languages
Domain-specific
languages
SQL, Postscript, HTML,
SARL...
4GL
Logic and constraint
languages
Prolog, OPS5, Object-Z...
5GL

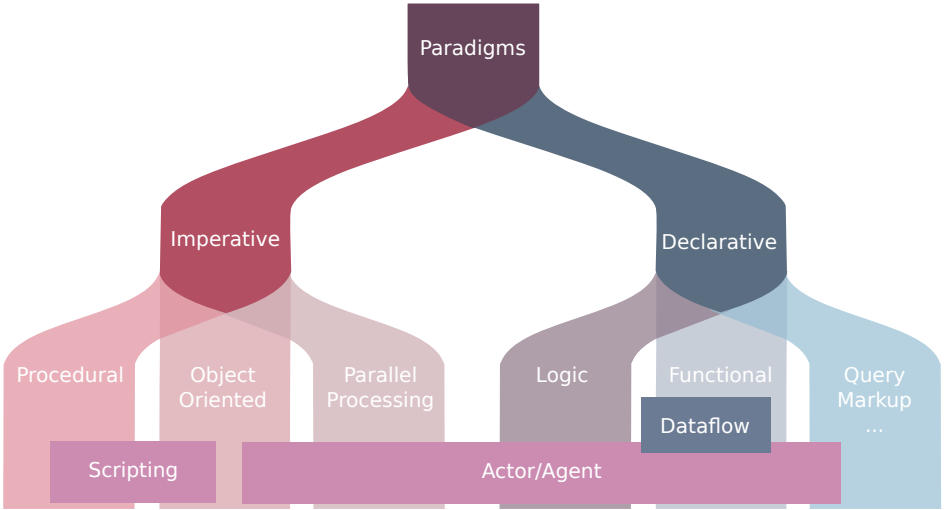


Imperative

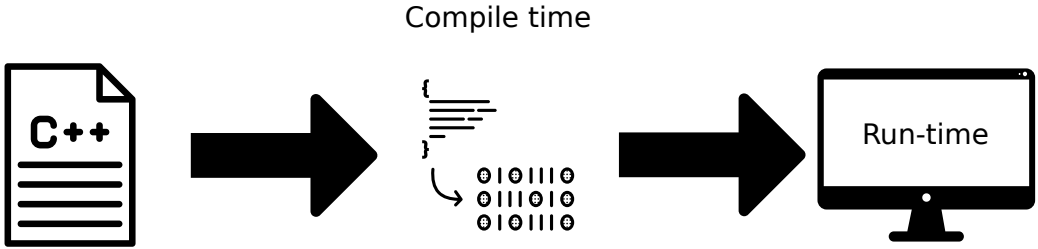
HOW a computation is to be done
Notion of program state, statements and control flow
(C, C++, C#, Java)

Declarative

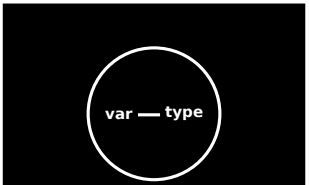
WHAT computation is to be done
Description of the logic of computation but not its control flow
(ML, Haskell, Prolog, SQL, HTML)



[Firme et al., 2013] - www.vimustar.com


Compile time
C++
Run-time
Check type here
Static typing
Check type here
Dynamic typing

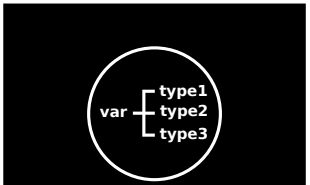
A variable may be associated to a type of values, i.e. the definition of a set of values



Strongly Typed

Single type for each variable
Type defined at compile time

⊕reliable
⊖tedious to write type annotations



Weakly Typed

Variable type depends on the run-time context
Values are converted at run-time

⊕flexible
⊖unexpected run-time errors

Name

A string of characters that refers to a thing in the program

Identifier

A string of characters that refers to an entity (data object, procedure, class, type)

- All identifiers are names; but not all names are identifiers
- `x.y` is a name but not an identifier, and `x` and `y` are identifiers.

Variable

A particular location of the store of the values at run-time.
A variable is denoted by a name. Each declaration of an identifier introduces a new variable.

Keyword

An identifier that has a particular meaning to the programming language

[Sethi, 1996, Scott, 2006]

Program or Subprogram

A sequence of instructions and statements

Procedure

A subprogram with a name and formal parameters that may be called

Function

A procedure that may return a value of some type (the "return type")

Method

A procedure or a function inside a class in object-oriented languages

Caution

In the C-family languages, all the subprograms are functions; and a function is enabled to return nothing (**void**)

[Sethi, 1996, Scott, 2006]

Declaration

Tells us about the type of an element. Example: **int** i;

Definition

Tells us about the value of an element. Example: i = 1;

Signature of a procedure/function

The declaration of the procedure/function
Composed of: a return type, an identifier, and a collection of parameter declarations

Example

In C++:

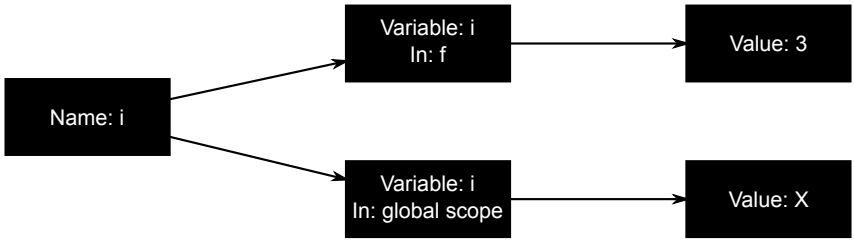
- a method is declared in a `.hpp` file
- a method is defined in a `.cpp` file

[Sethi, 1996, Scott, 2006]

Association of names with locations in memory (the store) and then with values is described by two mappings:

- Environment: mapping from names to locations in the store.
- State: mapping from locations in store to their values.

```
int i;          /* global i */
...
void f(...) {
  int i;        /* local i */
  ...
  i = 3;        /* use of local i */
  ...
}
...
x = i + 1;      /* use of global i */
```



One of the most important issues when designing a compiler is related to the decisions the compiler make about the program

Static Policy

A program uses a policy that enables the compiler to decide an issue; the decision could be decided at compile time.

Dynamic Policy

The decision can be made when we execute the program; the decision is required at run time.

Static Scope

A language uses a static scope if it is possible to determine the scope of a declaration by looking only at the program (C, Java...)

Dynamic Scope

With dynamic scope, as the program runs, the same use of a variable x could refer to any of several different declarations of x (Perl, PHP...)

[Church, , Frege, 1967, Wexelblat, 1981]

public static int x = 1;

- Here **static** refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory
- If **static** is omitted each object has this variable and the compiler cannot determine where it is in advance

Environment and state mappings are often dynamic

Static or Dynamic Environment Mapping?

- Most of binding names to locations are dynamic
- Some declarations (e.g., global `i`) are determine at compile time; they are static

Static or Dynamic State Mapping?

- Most of binding locations to values are dynamic because it is impossible to determine the location until we run the program
- Declared constants are an exception

All programming languages have the notion of procedur; but they can differ in how these procedures get their arguments

How are the actual parameters (the parameters used in the call of a procedure) associated with the formal parameters (those used in the procedure definition)?

① Call-by-value

② Call-by-reference

③ Call-by-name

Principle

- Actual parameter is evaluated or copied
- Value is in the location of the formal parameter

Constraints

- Changes to formal parameter is local to the procedure
- Actual parameters themselves cannot be changed

Example

Used in C and Java; and the default option in C++

Caution

In Java, all the object variables are references (or pointers) to the objects. Parameters are passed with the call-by-value policy, not the call-by-reference

Principle

- Address of the actual parameter is passed to the callee as the value of the corresponding formal parameter
- Uses of the formal parameter are implemented by following the pointer to the location indicated by the caller

Constraints

- Changes to the formal parameter thus appear as changes to the actual parameter

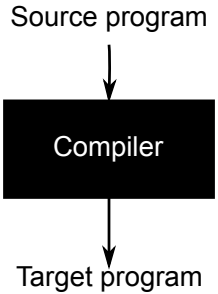
Principle

- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee
- Uses of the formal parameter are implemented by following the pointer to the location indicated by the caller

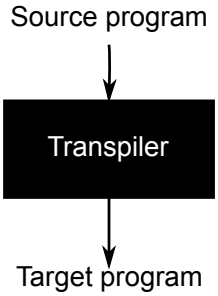
Example

Macro-functions in the C-family languages use this parameter passing mechanism

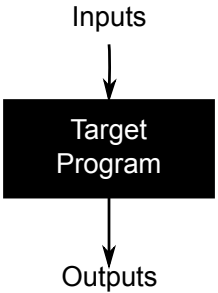
- Read a program in one language — the source language
- Translate it into an equivalent program in a **low-level language** — the target language
- Report any errors in the source program that are detected during the translation process



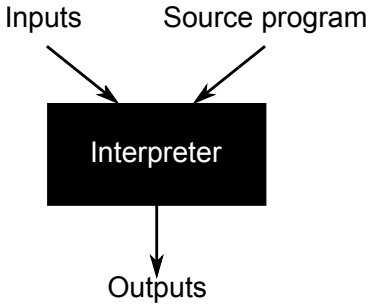
- Read a program in one language — the source language
- Translate it into an equivalent program in **another language that is not low-level** — the target language
- Report any errors in the source program that are detected during the translation process



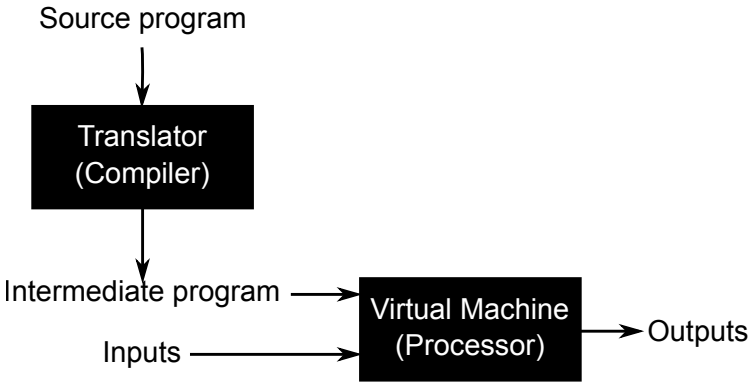
If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs



- A kind of language processor
- Does not produce a target program
- Directly execute the operations specified in the source program on inputs supplied by the user



- Combine compilation and interpretation
- Generate intermediate program in a platform-independent language
- Execute the intermediate program in a platform-dependent virtual machine



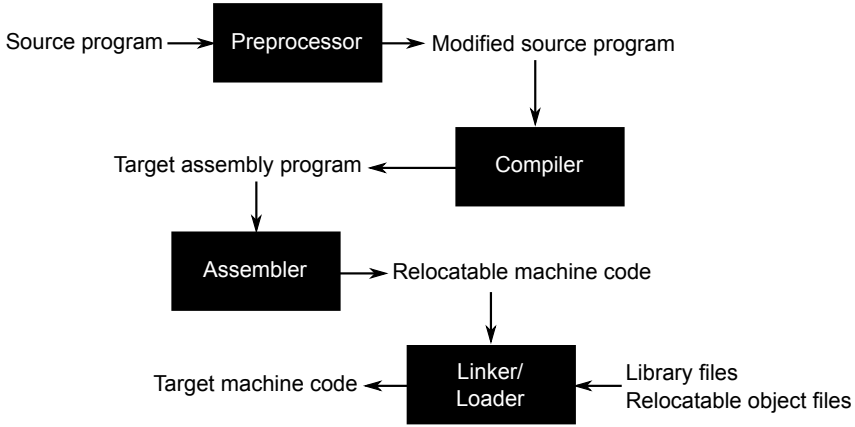
Compiler v.s. Transpiler v.s. Interpreter

- Compiler and transpiler is faster than interpreter at mapping inputs to outputs
- Interpreter gives better error diagnostics than compiler, because it executes the source program statement by statement (no code optimization)

Hybrid Compiler

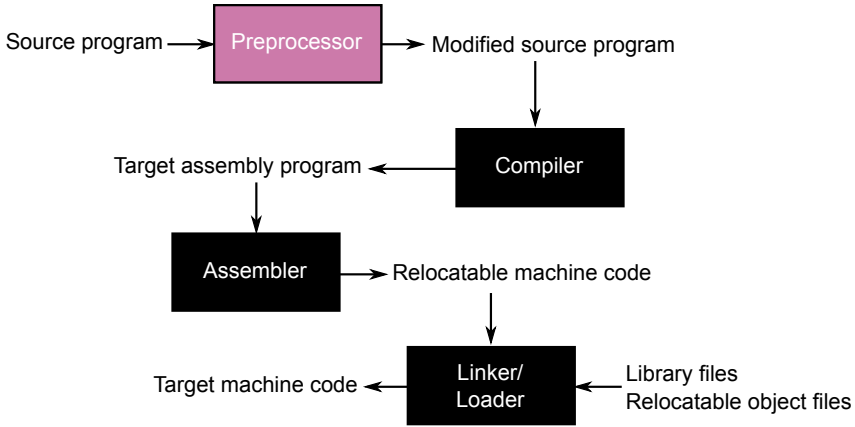
- Compile on one machine/architecture, execute the generated program on another machine/architecture
- To be faster, use just-in-time compilers to translate intermediate programs into machine language and avoid the interpretation, e.g., the Oracle's Java Runtime Environment

- Several programs may be required to create an executable target program
- They compose the toolchain of the compiler



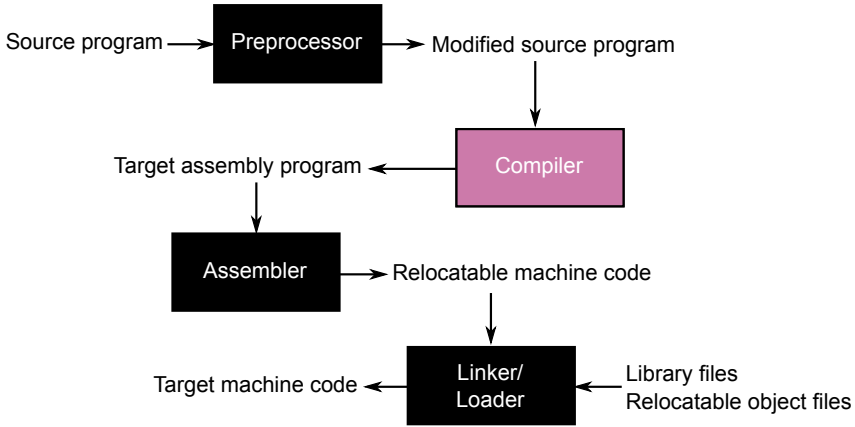
Goals

- To collect the different files of the program's modules to compile
- To expand shorthands, macros into statements



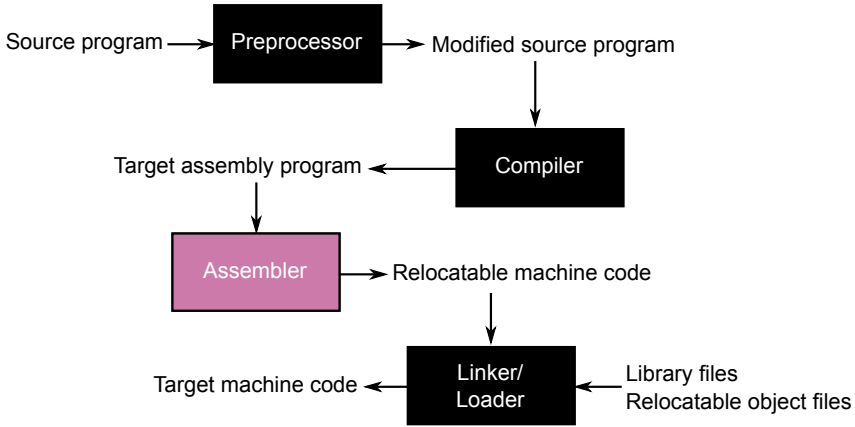
Goals

- To produce an assembly-language program from the modified source program
- Assembly-language is easier to produce and debug



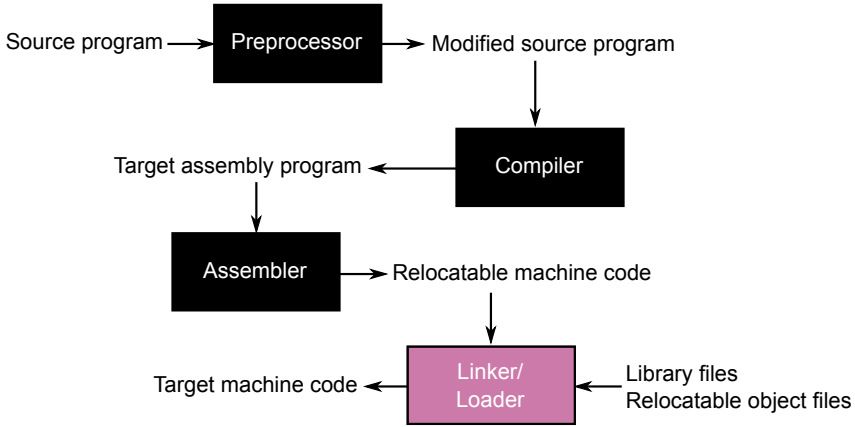
Goals

- To translate to a machine code that could be relocated in the code segment of the program
- Code segment: the part of the memory where machine code is store



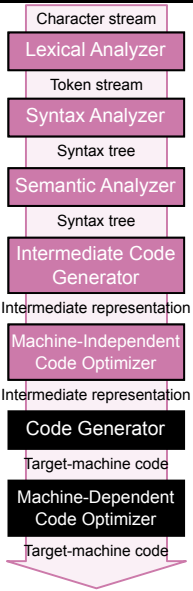
Goals

- To resolve external memory addresses, where the code in one file (library or object) may refer to a location in another file (library or object)



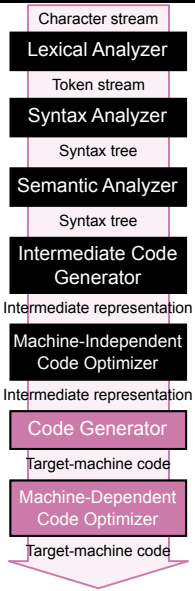
Analysis

- The analysis breaks up the source program into constituent pieces and imposes a grammatical structure to them.
- It detects if the source program is ill formed or semantically unsound.
- It collects informations about the source program and stores it in a data structure called symbol table.
- This part is often called the front end of the compiler.



Synthesis

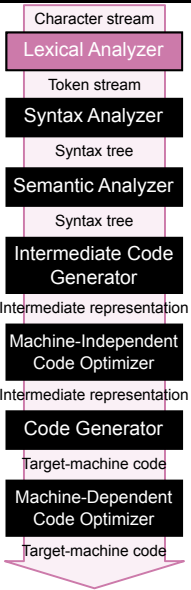
- The synthesis constructs the desired target program from the intermediate representation and the information in the symbol table.
- This part is often called the back end of the compiler.



- Reads the stream of characters making up the source program
- Groups the characters into meaningful sequences called lexemes
- Output for each lexeme:

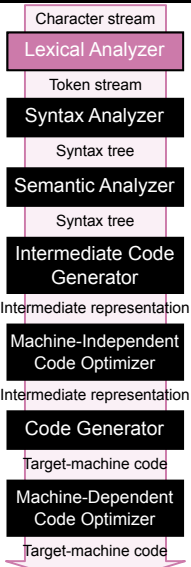
  `token=<token-name, attribute-value>`

  - token−name: the identifier of the token
  - attribute −value: entry in the symbol table for this token



position = initial + rate * 60

Lexeme	Token
position	<id,1> ■ id: abstract symbol standing for "identifier" ■ "1": points to the symbol-table entry for position
=	<=>
initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<number,60>



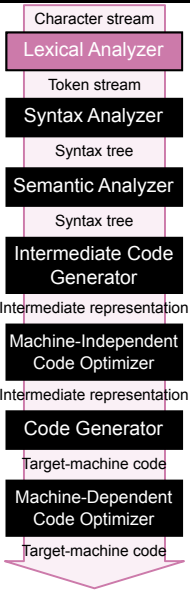
position = initial + rate * 60

↓

<id,1><=><id,2><+><id,3><*><number,60>

Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...



Symbol Table

Central data structure containing a record for each variable name, with record fields for the attributes associated to the name

Record Attribute

Information about the storage allocated for a name: type, scope, number and types of the formal parameters, the method of passing each argument, and the type of the returned value

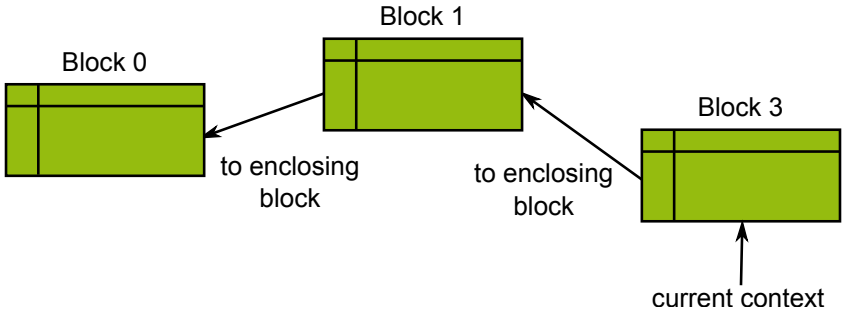
Caution

Should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

Scopes are implemented by setting up a separate symbol table for each scope

Principle

The most-closely nested rule for blocks permits to define a data structure, which is based on chained symbol tables.



```java
/** Define the properties of a single symbol. */
public class Symbol {
 public final String lexeme;
 public Type type;
 public Address storagePosition;
 public Symbol(String lexeme) { this.lexeme = lexeme; }
}

/** Define a symbol table. */
public class SymbolTable {

 /** Collection of the symbol in the current context. */
 private final Map<String,Symbol> table = new TreeMap<String,Symbol>();

 /** Reference to the symbol table that is associated to the enclosing scope.
    */
 private final SymbolTable enclosingEnvironment;

 /** Constructor. */
 private SymbolTable(SymbolTable enclosingEnvironment) {
     this.enclosingEnvironment = enclosingEnvironment;
 }
```

```java
/** Declare a symbol in the current context. */
public void declare(String identifier, Symbol symbol) {
    this.table.put(identifier, symbol);
}

/** Get the definition of a symbol in the current context,
    or in an enclosing scope. */
public Symbol get(String identifier) {
    SymbolTable e = this;
    Symbol symbol;
    while (e!=null) {
        symbol = e.table.get(identifier);
        if (symbol != null) {
            return symbol;
        }
        e = e.enclosingEnvironment;
    }
    return null;
}
```

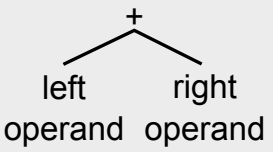
```
    /** Reference to the current symbol table.
        The reference is initialized with the
        root context (or the global context). */
    private static SymbolTable current = new SymbolTable(null);

    /** Replies the symbol table of the current context. */
    public static SymbolTable getCurrent() {
        return current;
    }

    /** Open a new context and create the corresponding
        symbol table. */
    public static void openContext() {
        current = new SymbolTable(current);
    }
    /** Close the current context. */
    public static void closeContext() {
        if (current.enclosingEnvironment!=null) {
            current = current.enclosingEnvironment;
        }
    }
}
```

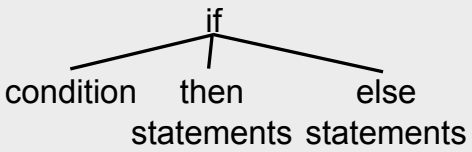
Uses the tokens produced by the lexical analyzer to create an intermediate representation

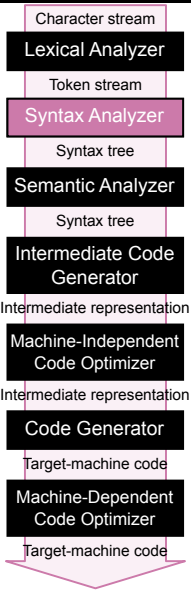
Syntax Tree

A typical representation is a syntax tree:

- node: operation in the program
- children: parameters of the operation

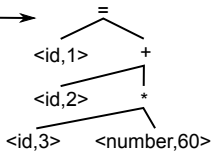






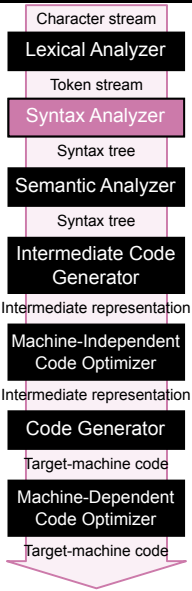
position = initial + rate * 60

<id,1><=><id,2><+><id,3><*><number,60>



Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...



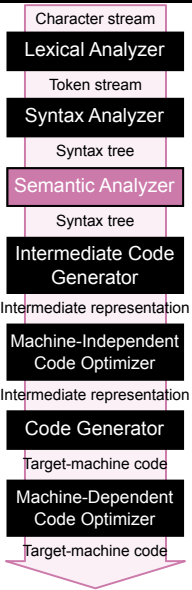
Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition

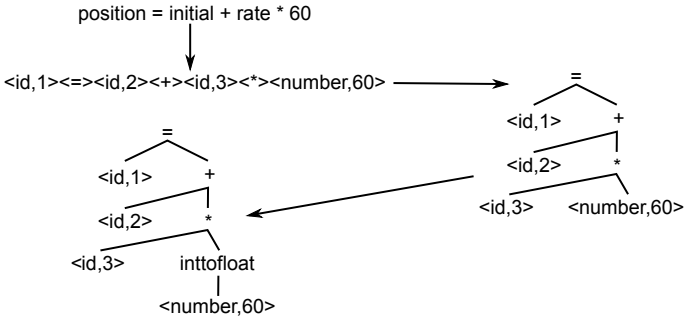
Actions

- Gathers type information and saves it in either the syntax tree and the symbol table
- Applies coercions, or type conversions

Type Checking

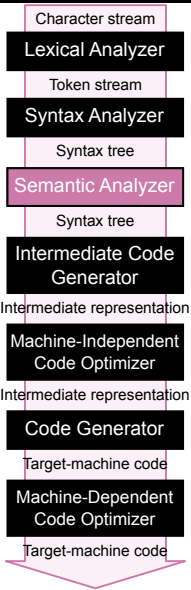
Important part of the semantic analyzer: the compiler checks that each operator has matching operands





Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...

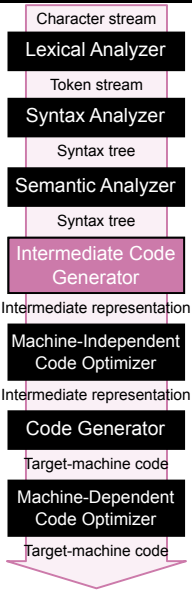


Many compilers generate an explicit low-level or machine-like intermediate representation, which is a program for an abstract machine

Intermediate code

Two representations are generally used:

- Syntax tree
- Three-address code, that is easy to produce, and translate into the target machine



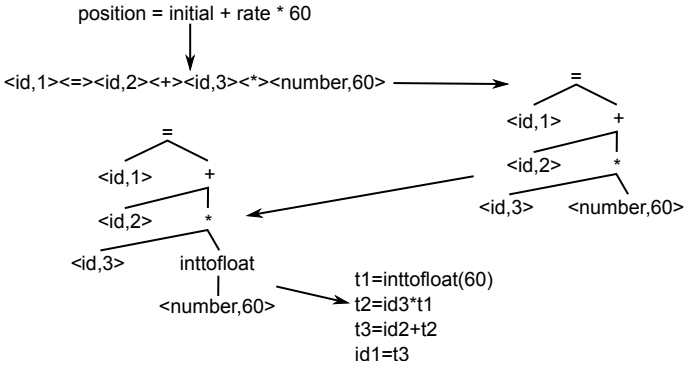
A sequence of assembly-like instructions with, at most, three operands per instruction.

`<variable> = <operand1> <operator> <operand2>`

- Each operand can act like a register
- The affectation operator is implicit and always present

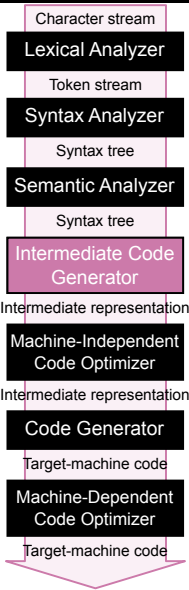
Constraints

1. At most one operator on the right side
2. Temporary names are generated to hold the value computed by the three-address instruction
3. Some instructions have fewer then three operands



Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...

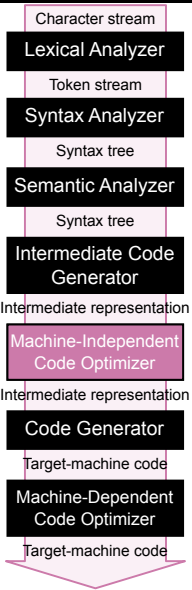


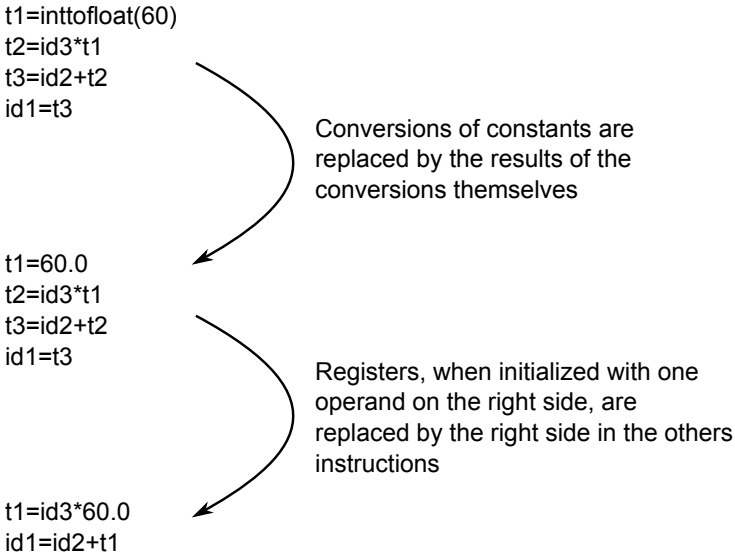
Improves the intermediate code for better target code
(faster, shorter, less power consumer. . .)

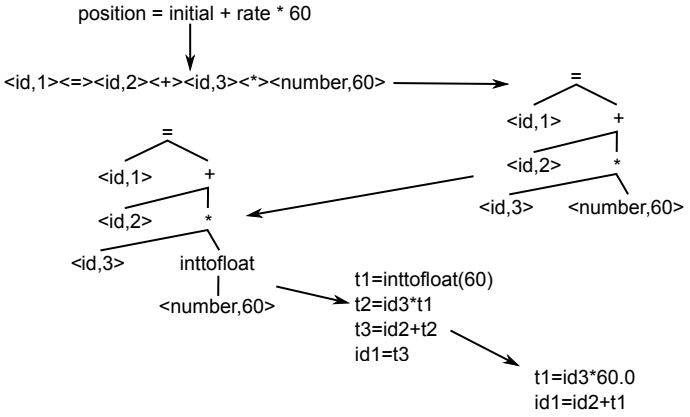
- All the compilers include a machine-independent code optimizer
- Those that spent a large amount of time on this phase are named “optimizing compilers”

Note

Many of the simple optimizations permit to significantly improve the running time of the target program without too much time spent on this phase

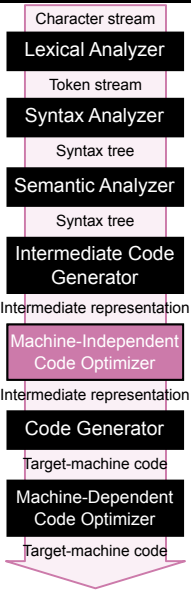



t1=inttofloat(60)
t2=id3*t1
t3=id2+t2
id1=t3
Conversions of constants are replaced by the results of the conversions themselves
t1=60.0
t2=id3*t1
t3=id2+t2
id1=t3
Registers, when initialized with one operand on the right side, are replaced by the right side in the others instructions
t1=id3*60.0
id1=id2+t1



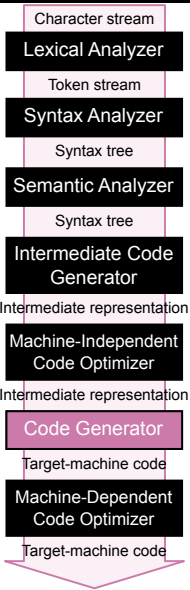
Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...

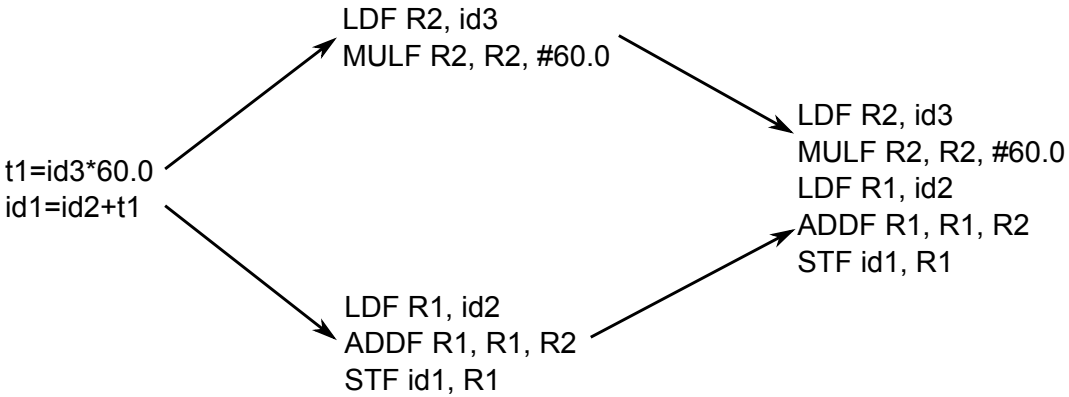


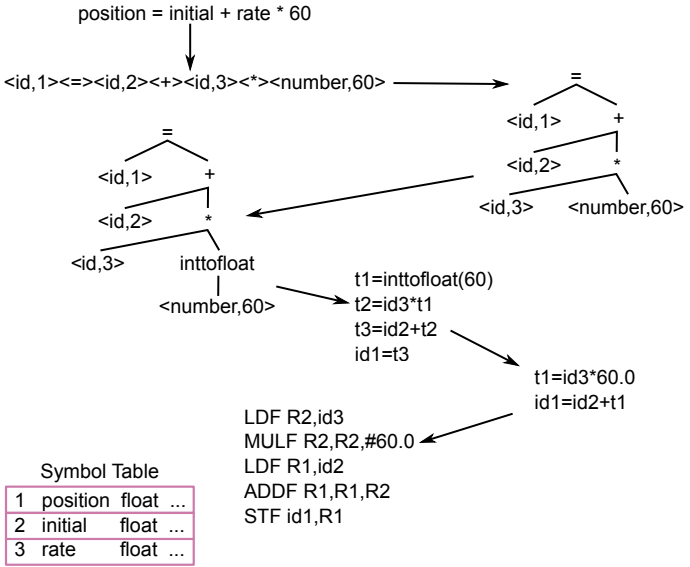
Maps an intermediate representation to the target language

- If the target language is machine code, registers or memory locations are selected for each variables used by the program
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same tasks
- A crucial aspect is the judicious assignment of registers to hold variables



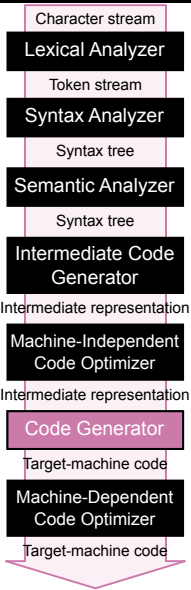
- Assumes that R1 and R2 are registers
- Variables are mapped to registers so that they can be easily used for the generation of the next instructions





Symbol Table

1	position	float	...
2	initial	float	...
3	rate	float	...

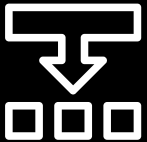


Several tools are available to help the compiler writer to build his/her compiler

Scanner generators

Lexical analyzers from a reg-ex description tokens (Flex, JFlex...)

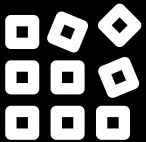
1



Parser generators

Syntax analyzers from grammar (Yacc, JavaCC, Bison...)

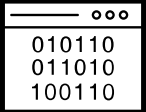
2



Syntax-directed translation engines

Parse-tree walkthrough routines for intermediate code generation

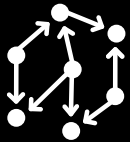
3



Code-generator generators

Procedures for generating target machine code from intermediate code

4



Data-flow analysis engines

Help for management of value exchange between compiler components

5

Construction toolkits

Include the other tools and IDE integration (Xtext. . .)

6

- Language Processors: An integrated software development environment: compilers, interpreters, linkers, loaders, debuggers, profilers.
- Compiler Phases: Sequence of phases, each of which transforms the source program from one intermediate representation to another.
- Machine and Assembly Languages: Machine languages were the first-generation programming languages, followed by assembly languages.
- Code Optimization: the science of improving the efficiency of code in both complex and very important. It is a major portion of the study of compilation.
- Higher-Level Languages: Programming languages take on progressively more of the tasks that formerly were left to the programmer: memory management, type-consistency...
- Environments: The association of names with locations in memory and then with values can be described in terms of environments.
- Parameter Passing: Parameters are passed from a calling procedure to the callee either by value or by reference.

- Aliasing: When parameters are (effectively) passed by reference, two formal parameters can refer to the same object.
- Compiler Front End: The part of the compiler that is dedicated to the analysis phases. The compiler front end takes the source program, breaks it to token, analyzes the grammar, detects errors and inconsistencies, and generate an intermediate representation.
- Compiler Back End: The part of the compiler that is dedicated to the synthesis phases. The compiler back end takes the intermediate representation, generates assembly and machine code.
- Lexical Analyzer: The lexical analyzer reads the input one character at a time and produces as output a stream of tokens. A token consists of a terminal symbol and attribute values.
- Parsing: Parsing is the problem of figuring out how a string of terminals can be derived from the start symbol of the grammar by repeatedly replacing a nonterminal by the body of one of its productions.

- Parse Tree: A graphical tree representation of the productions that are matching a sequence of input tokens.
- Intermediate Code: The result of the syntax analysis is a representation of the source program, called intermediate code. Two primary forms of intermediate code are illustrated: abstract syntax tree (similar to parse tree), and three-address code.
- Symbol Table: A data structure that holds information about identifiers.

Church, A.
The Calculs of Lambda Conversion.
Princeton University Press, Princeton, N.J.

Firme, J., Valera, N., Canemre, Y., Burchill, S., and Khurshid, B. (2013).
Programming language families.

Frege, G. (1967).
Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought, chapter From Frege to Gödel.
Havard Univ. Press, Cambridge, MA.

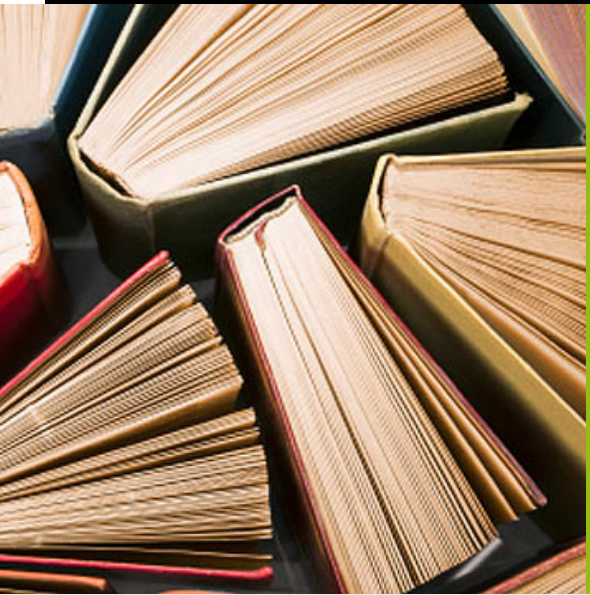
Scott, M. (2006).
Programming Language Pragmatics.
Morgan-Kaufmann, Sans Francisco, CA, 2nd edition edition.

Sethi, R. (1996).
Programming Languages: Concepts and Constructs.
Addison-Wesley.

Wexelblat, R. L. (1981).
History of Programming Languages, volume 1.
Academic Press.

Chapter 2
Lexical Analysis

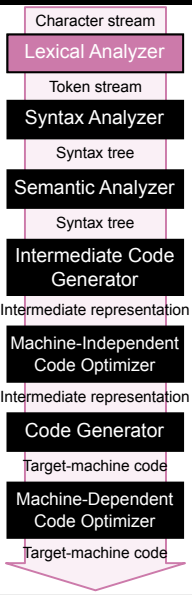
Stéphane GALLAND

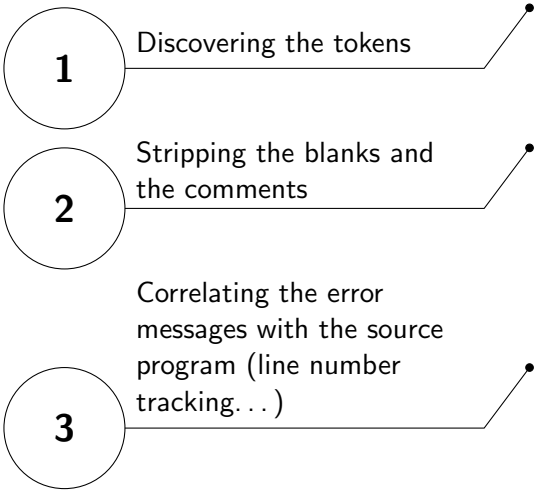


The lexical analyzer reads source program and extract lexemes

Each lexeme is associated to a token:
`<token-name, attribute-value>`

Outputs the tokens





Cascading Process (most of the time)

1. Scanning: processes that do not require tokenization of the input, e.g. deletion of comments and compaction of consecutive white spaces
2. Lexical analysing: produces tokens from the output of the scanner

Definition

A sequence of characters in the source program that is identified by the lexical analyzer as a lexical unit (element of the language)

Example

- Let the statement: printf ("Total = %dn", score);
- Both printf and score are lexemes
- String of characters is a lexeme
- Parenthesis, coma and semicolumn characters are also lexemes

Definition

A pair consisting of a token name and an optional attribute value

- name: abstract symbol representing a kind of lexical unit
- token names are the input symbols that the parser processes
- token name is written in **bold-face**

Example

- Let the statement: printf ("Total = %dn", score);
- both printf and score are lexemes matching the pattern for token **id**

Definition

A description of the form that the lexemes of a token may take

- For keyword: the pattern is a sequence of characters that form the keyword
- For identifier and some other token: the pattern is a more complex structure that is matched by many strings

Example

- Let the statement: printf ("Total = %dn", score);
- both `printf` and `score` are described by the pattern **[a-zA-Z][a-zA-Z]*** (regex)

In many programming languages, the following classes cover most or all of the tokens:

Class	Description
keyword	Pattern is the name of the token itself
operator	Individually or in classes, e.g., class **comparison**
identifier	One token per identifier
constant	One token per type of constant, e.g. number or string literal
punctuation	One token per punctuation symbol, e.g. left and right parentheses, comma, and semicolon

Definition

Additional information associated to a token, when more than one lexeme can match a pattern

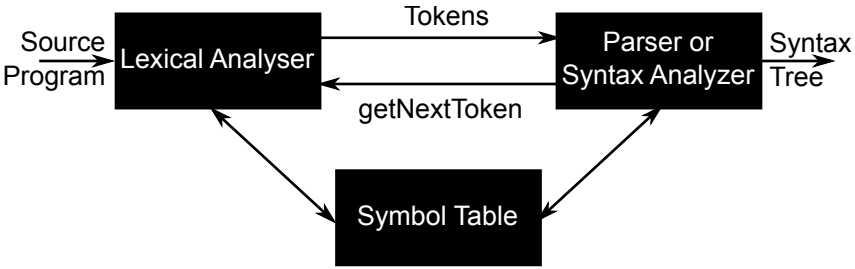
Examples

- value of the parsed number (lexeme) for token **number**
- position of the identifier into the symbol table for token **id**

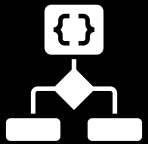
Assumption

Usually, token has at most one associated attribute; but it could be a data structure

Lexical analyzer generally does not control the execution flow of the compiler



- Lexical analyzer is invoked by the parser through a call to getNextToken function
- Then, lexical analyzer tries to discover and to reply a token



Design Simplicity

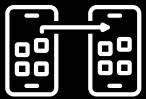
Enable simplification of at least one of these tasks

1

Compiler Efficiency

Enable application of specialized techniques

2



Easier Portability

Specific input devices supported by lexical analyzer

3

It is hard for a lexical analyzer to tell that there is a source-code error

Example

fi (a == f(x)) ...

Problems

- Cannot tell whether fi is a misspelling of the keyword **if** or an undeclared function identifier
- Fails when none of the patterns for tokens matches any prefix of the remaining input

If such an error is detected, lexical analyzer must output an error message and try to recover a stable state

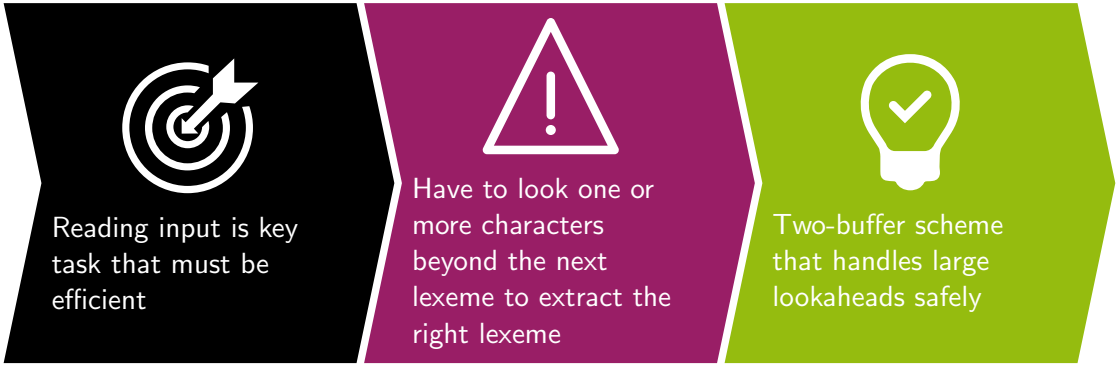
Successive characters are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of input

Delete character from input

Insert character into input

Replace character in input

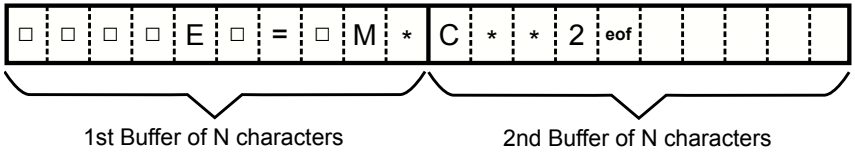
Transpose adjacent characters



Example

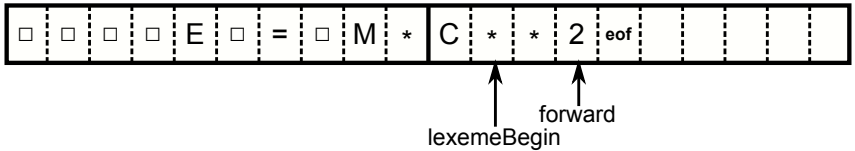
To be sure that a character is the last of an identifier, the next character must be read, and it is not part of the lexeme for **id**

Two buffers are read and used alternatively



- Assumption: the larger lexeme has a size lower or equals to N
- N is usually the size of the disk block
- **eof** character is put in the buffer when there is not enough characters in the input

How to be efficient? By invoking the read system function for N characters rather than a call per character



- `lexemeBegin`: the beginning of the current lexeme
- `forward`: the current character

Algorithmic Principle

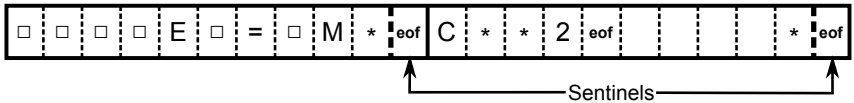
1. If `forward` is outside a buffer, the other buffer is reloaded from the input, and move `forward` to the beginning of the newly loaded buffer
2. If character pointed by `forward` does not match a lexeme from `lexemeBegin`:
   - If is is a valid lexeme, output the lexeme, move `lexemeBegin` to `foward`
   - Else generate an error
3. Else move `forward` to the right

Problem of efficiency

For each character read, two tests:

1. one for the end of the buffer
2. one to determine what character is read (usually with a multiway branch)

⇒ To improve the speed of the treatment, we can combine the two tests by extending each buffer with a sentinel character (usually **eof**)



In most modern languages, lexemes are short $N \geq 1000$ is ample

Character strings can be very long (more than N)

Add a dynamic buffer scheme for large lexeme

Reply a sequence of **str** tokens, one for each of the shorter strings (see example)

Example

Compile-time string concatenation in C: "ABC" "DEF"

Alphabet

Any finite set of symbols; Example: $A = \{a, b, c, \delta\}$

String

A finite sequence s of symbols drawn from an alphabet A.
Example: $s \in S/S = \mathcal{S}(\mathcal{P}(A)) \setminus \{\emptyset\} = \{a, b, c, \delta, ab, ac, a\delta, \dots\}$
$|s|$ is the size of s

Language

Any countable set of strings over some fixed alphabet
Example: $L \subseteq S = \{abc, \delta, b, bc\}$

The following operations are used for defining the pattern matchings for languages

Union of the languages L and M

$L \cup M = \{s | s \in L \vee s \in M\}$

Example: Let $L = \{a, b, c\}$ and $M = \{d, e\}$
then $L \cup M = \{a, b, c, d, e\}$

Concatenation of the languages L and M

$LM = \{st | s \in L, t \in M\}$

Example: Let $L = \{a, b, c\}$ and $M = \{d, e\}$
then $LM = \{ad, ae, bd, be, cd, ce\}$

[Kleene, 1956]

Self-concatenation of the language L

$$L^i = \begin{cases} \{\epsilon\} & \text{if } i = 0 \\ L^{i-1}L & \text{if } i > 0 \end{cases}$$

Example: Let $M = \{d, e\}$

$$\text{then } M^4 = \left\{ \begin{array}{llll} dddd, & ddde, & dded, & ddee, \\ dedd, & dede, & deed, & deee, \\ eddd, & edde, & eded, & edee, \\ eedd, & eede, & eeed, & eeee \end{array} \right\}$$

[Kleene, 1956]

Kleene's Closure of the language L

$L^* = \bigcup_{i=0}^{\infty} L^i$

Example: Let $M = \{d, e\}$

then $M^* = \left\{ \begin{array}{l} \epsilon, d, e, dd, de, ed, ee, ddd, dde, ded, dee, edd, ede, eed, \\ eee, dddd, ddde, dded, ddee, dedd, dede, deed, deee, ... \end{array} \right\}$

Positive Closure of the language L

$L^+ = \bigcup_{i=1}^{\infty} L^i$

Example: Let $M = \{d, e\}$

then $M^+ = \left\{ \begin{array}{l} d, e, dd, de, ed, ee, ddd, dde, ded, dee, edd, ede, eed, \\ eee, dddd, ddde, dded, ddee, dedd, dede, deed, ... \end{array} \right\}$

[Kleene, 1956]

Regular Expressions (shortened as regex, regexp or rational expression)

A sequence of characters that specifies a search pattern.

Usually used for describing all the languages that can be built from the operators previously defined

Built recursively out of smaller regular expressions (see following slides).

Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's expressions.

Aho, 1990, Shannon and McCarthy, 1956, [Kleene, 1956, Aho et al., 1986]

The rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote are:

Definition (BASIS)

There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
2. If a is a symbol in Σ, then **a** is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with *a* in its position.

Remark

By convention, we use *italics* for symbols, and **boldface** for their corresponding regular expressions.

Definition (INDUCTION)

There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$
3. $(r)*$ is a regular expression denoting the language $(L(r))^*$
4. (r) is a regular expression denoting $L(r)$

This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Regular expressions often contain unnecessary pairs of parentheses

Simplification Rules

a) The unary operator "$*$" has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) "$|$" has lowest precedence and is left associative.

Regular Set

A language that can be defined by a regular expression
If two regular expressions r and s denote the same regular set, they are equivalent $r = s$

Law	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = st\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r* = (r\|\epsilon)*$	ϵ is guaranteed in a closure
$r** = r*$	$*$ is idempotent

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols

Regular definition

If Σ is an alphabet of basic symbols, a regular definition is a sequence of definitions of the form:

$$\begin{aligned} \mathbf{d_1} &\rightarrow r_1 \\ \mathbf{d_2} &\rightarrow r2 \\ &\dots \\ \mathbf{d_n} &\rightarrow r_n \end{aligned}$$

- $\mathbf{d_i}$ is a new symbol, not in and not the same as any other of the d's
- r_i is a regular expression over the alphabet $\Sigma \cup \{\mathbf{d_1}, \mathbf{d_2}, \dots, \mathbf{d_{i-1}}\}$.

$$
\begin{aligned}
\textbf{letter} &\rightarrow A|B|\ldots|Z|a|b|\ldots|z \\
\textbf{letter\_} &\rightarrow \textbf{letter}|\_ \\
\textbf{digit} &\rightarrow 0|1|\ldots|9 \\
\textbf{letters} &\rightarrow \textbf{letter letter}* \\
\textbf{digits} &\rightarrow \textbf{digit digit}* \\
\textbf{id} &\rightarrow \textbf{letter\_}(\textbf{letter\_}|\textbf{digit})* \\
\textbf{optFrac} &\rightarrow . \ \textbf{digits}|\epsilon \\
\textbf{optExp} &\rightarrow ((E|e)(+|-|\epsilon)\textbf{digits})|\epsilon \\
\textbf{number} &\rightarrow \textbf{digits optFrac optExp}
\end{aligned}
$$

Since [Kleene, 1956] introduced regular expressions with the basic operators in 1950s, many extensions have been added to enhance their ability to specify string patterns

$r+$

One or more instances

$L(r+) = (L(r))^+$
Same precedence and associativity as "$*$"

1

$r?$

Zero or one instance

$L(r?) = L(r) \cup \{\epsilon\}$
Same precedence and associativity as "$*$"

2

[a]

Character classes

$[ab...z] = a|b|...|z$
Consecutive symbols: $[a-z]$

3

[Kleene, 1956]

$$
\begin{aligned}
\textbf{letter} &\rightarrow [A-Za-z] \\
\textbf{letter\_} &\rightarrow [A-Za-z\_] \\
\textbf{digit} &\rightarrow [0-9] \\
\textbf{letters} &\rightarrow \textbf{letter}+ \\
\textbf{digits} &\rightarrow \textbf{digit}+ \\
\textbf{id} &\rightarrow \textbf{letter\_}(\textbf{letter\_}|\textbf{digit})* \\
\textbf{number} &\rightarrow \textbf{digits}(.\textbf{digits})?([Ee][+-]?\textbf{digits})?
\end{aligned}
$$

We are able to express patterns using regular expressions
How to regex patterns for all the tokens of our language?

term	$\rightarrow$	**number**
	$\rightarrow$	**id**
expr	$\rightarrow$	**term = term**
	$\rightarrow$	**term <> term**
	$\rightarrow$	**term < term**
	$\rightarrow$	**term > term**
	$\rightarrow$	**term <= term**
	$\rightarrow$	**term >= term**
statement	$\rightarrow$	`if` **expr** `then` **statement** `else` **statement**
	$\rightarrow$	**term**

Lexeme	Regular Expression	Token	Token Attributes
ws	$[\ \backslash n \backslash t \backslash r]+$	-	-
if	*if*	**if**	-
then	*then*	**then**	-
else	*else*	**else**	-
id	**letter**_ (**letter**_|**digit**)$*$	**id**	pointer to symbol table's entry
number	**digits**(. **digits**)?	**number**	pointer to symbol table's entry
=	=	**relop**	EQ
<>	<>	**relop**	NE
<	<	**relop**	LT
>	>	**relop**	GT
<=	<=	**relop**	LE
>=	>=	**relop**	GE

As an intermediate step in the construction of a lexical analyzer, patterns are converted to flowcharts, called transition diagrams.

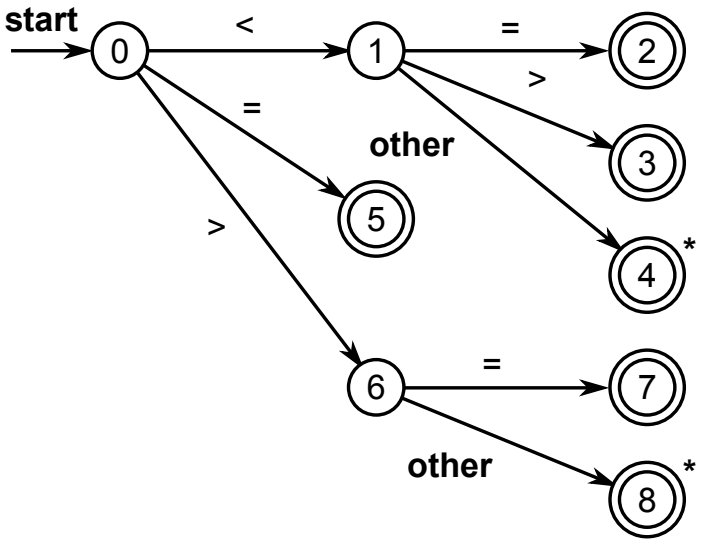
Definition (Transition Diagram)

- Diagram: composed of states and edges
- State: a step in the scanning of a string, that also indicates if the input stream is validating the regular expression, or not
- Edge: directed from one state to another. Each edge is labeled by a symbol or a set of symbols

Assumption

All transition diagrams are deterministic: never more than one edge out of a given state with a given symbol among its labels.

Notation	Explanation
	The transition diagram always begins in the start state before any input symbols have been read.
	The transition labelled with **other** is traversable when no other transition is traversable.
	The accepting state (or final) indicates that a lexeme has been found (between pointers lexemeBegin and forward).
	If the lexeme does not include the symbol that got us to the accepting state, it is necessary to retract the forward pointer by one position.



<relop,LE>

<relop,NE>

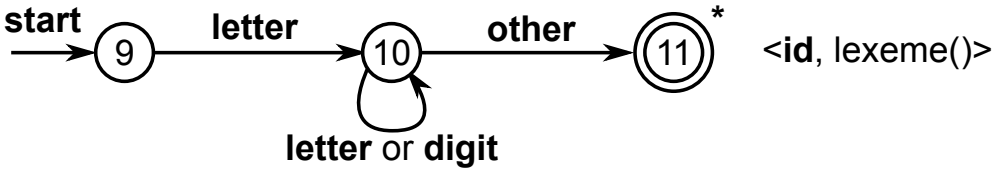
<relop,LT>

<relop,EQ>

<relop,GE>

<relop,GT>

- The transition diagram that recognizes the identifiers is:

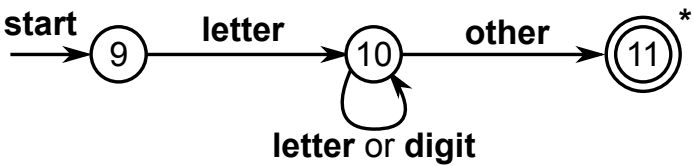


- lexeme() replies the current lexeme (between lexemeBegin and forward pointers).

Recognizing keywords and identifiers presents a specific problem: keywords are not identifiers even though they look like identifiers.

① Install all the keywords in the symbol table initially
A field of the symbol-table entry indicates that the string are never ordinary identifier

- installID () places the identifier in the symbol table if it is not already there and returns a pointer to the symbol-table entry.
- getToken() replies the token that is corresponding to the lexeme, or **id** otherwise.



<getToken(lexeme()),
installID(lexeme())>

② Create a separate transition diagram for each keyword

- tokens must be prioritized so that the reserved-word tokens are recognized in preference to **id**
- Approach less used than the previous approach when the lexical analyzer is written by hand

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer

- A variable state is holding the number of the current state for a transition diagram
- Each transition diagram is simulated by a piece of code inside a function
- The code of a state is itself a switch statement or a multiway branch that determines the next state by reading and examining the next input character.

```
Token getRelop() { /* return null on failure */
  char c;
  int state = 0;
  Token token = new Token(Tag.RELOP);
  while (true) { /* repeat until a return or failure */
    switch(state) {
    case 0:
      c = nextChar();
      if (c=='<') state = 1;
      else if (c=='=') state = 5;
      else if (c=='>') state = 6;
      else return null; /* lexeme is not a relop */
      break;
    case 1: ...
    case 8:
      retract(); // move back the "forward" and "lexemeBegin" pointers
      token.attribute = "GT";
      return token;
    default: return null;
    }
  }
}
```

To build the entire lexical analyzer, the codes for simulating the transition diagrams may be arranged in different ways

1. Arrange for the transition diagrams for each token to be tried sequentially
   - When the function is replying **null** (failure), the pointer forward is reset and the next transition diagram is started
   - This approach allows us to use the transition diagrams for the individual keywords
   - We have only to use them before we use the diagram for **id**, in order the keywords to be reserved words

② Run the various transition diagrams "in parallel"

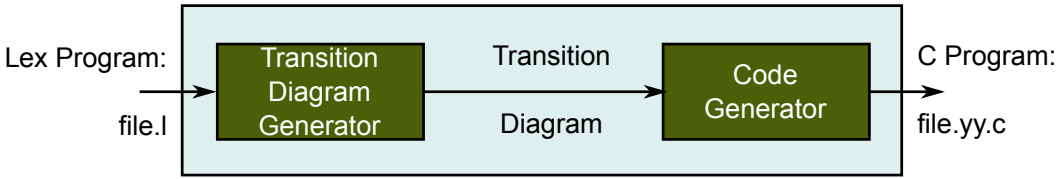
- Caution: be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input
- Strategy: take the longest prefix of the input that matches any pattern

③ Preferred approach: combine all the transition diagrams in one
- Transition diagram reads input until there is no possible next state
- Then, longest lexeme that matched any pattern is replied

The problem of combining transition diagrams for several tokens is complex. The easiest way to solve this problem is to study how lexical-analyzer generators, such as Lex or Flex, are working

Several tools allow to generate a lexical analyzer by specifying regular expressions to describe the patterns for the tokens

- This section introduces the tool:
  - Lex, and its more recent implementation Flex (dedicated to compilers written in C or C++)
  - JavaCC (dedicated to compilers written in Java)
- The input notation is the Lex language



[Lesk, 1975]


Lex Program:
file.l
Lex
Compiler
C Program:
file.yy.c


C Program: file.yy.c
C Program: main.c
C
Compiler
Executable:
a.out


Input Stream
a.out
Sequence
of tokens

- The lexical analyzer is a C function that returns an integer, which is a code for one of the possible token names
- This subroutine is called by the parser
- Attribute value (another numeric code) is a pointer to the symbol table, or nothing
- This value is placed in a global variable yylval

A Lex program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Declarations

- Declarations of variables (in C)
- Manifest constants: identifiers declared to stand for a constant, eg. the name of a token
- Regular definitions

A Lex program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Translation rules

- Have the form:

  `Pattern { Actions }`

- Pattern is a regex
- Actions are fragments of C code
- Evaluation order: first matching rule, first used

A Lex program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

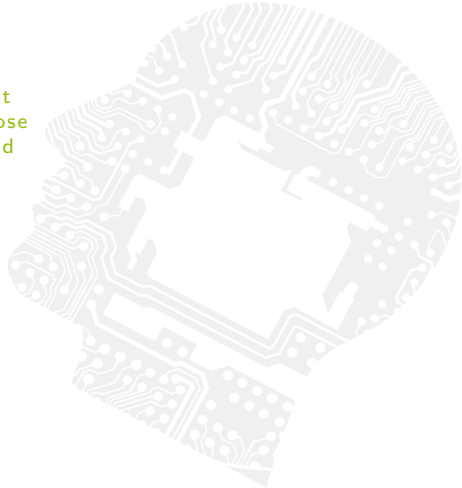
Auxiliary functions

- Holds whatever additional functions are used in the actions

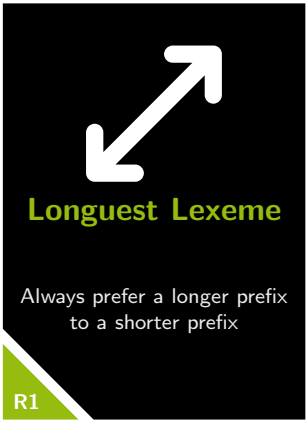
```
%{
   /* definitions of manifest constants, if not already
      declared in the parser files (yacc) */
   enum { LT, LE, EQ, NE, GT, GE } RelopId;
   enum { IF, THEN, ELSE, ID, NUMBER, RELOP } TokenName;
%}


/* regular expressions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?([Ee][+-]?{digit}+)?

%%
```

```
{ws}        { /* no action and no return */ }
if          { return IF; }
then        { return THEN; }
else        { return ELSE; }
{id}        { yylval = (int)installID(); return ID; }
{number}    { yylval = (int)installNumber(); return NUMBER; }
"<"         { yylval = LT; return RELOP; }
"<="        { yylval = LE; return RELOP; }
"="         { yylval = EQ; return RELOP; }
"<>"        { yylval = NE; return RELOP; }
">"         { yylval = GT; return RELOP; }
">="        { yylval = GE; return RELOP; }

%%
```

```
int installID() {
  /* function to install the lexeme, whose first
     character is pointed to by yytext, and whose
     length is yyleng, into the symbol table and
     return a pointer thereto */
}

int installNumber() {
  /* similar to installID, but puts numerical.
     Constants into a separate table */
}
```



Two rules are used by Lex to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns



- Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input
- Problem: Sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters
- Solution: use the character "/" in the pattern to indicate the end of the part of the pattern that matches the lexeme
  - a / b means "a followed by b" (a and b are regular expressions)
  - The additional pattern (b) is not consumed from the input in the lexical analyzer point-of-view

- Several implementations of lexical-analyzer generators provides Java source code
- JLex is a lexical analyzer generator, written for Java, in Java
- JLex is based upon the Lex lexical analyzer generator model ⇒ the input file is the similar as the one for Lex, but not the same

```
User code
%%
JLex directives
%%
Translation rules
```

http://www.cs.princeton.edu/~appel/modern/java/JLex/

- User code: copied verbatim into the lexical analyzer source file
- JLex directives: explained in the online documentation
- Translation rules: series of rules for breaking the input stream into tokens
  Each rule has three distinct parts: the optional state list, the regular expression, and the associated action:

```
[<states>] <expression> { <action> }
```

```
User code
%%
JLex directives
%%
Translation rules
```

- JFLex is a lexical analyzer generator, written for Java, in Java
- It is a rewrite of JLex with extended features (as for Flex/Lex implementations)

`http://www.jflex.de`

- Java Compiler Compiler (JavaCC) is one of the most popular parser generators for Java applications
- Even if JavaCC is a parser, it includes a lexical analyzer in a transparent way
- Regex, strings, and the grammar specifications (the BNF) are both written together in the same file
- JavaCC is detailed in Chapter 3

`http://javacc.java.net`

To go deeper in how a program like Lex turns its input program into a lexical analyzer, the formalism called "**finite automata**" is at the heart of this transition

[Aho, 1990, Hopcroft et al., 2006]

Finite Automaton

Finite automaton is recognizer: it says “yes” or “no” about each possible input string

Deterministic finite automaton - DFA

DFA has, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state

Nondeterministic finite automaton - NFA

NFA have no restrictions on the labels of their edges

- DFA and NFA are represented by transition graphes
- Similar to transition diagram, except the same label can be on edges from one state, and an edge may be labeled by ϵ

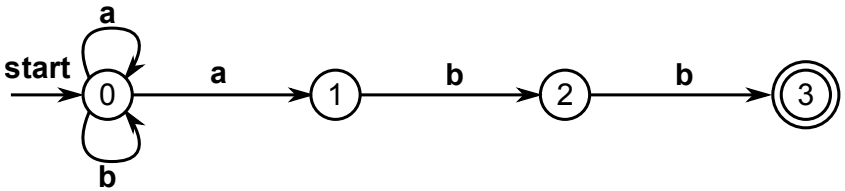
[McCullough and Pitts, 1943]

A nondeterministic finite automaton (NFA) is defined by:

$$\langle S, \Sigma, move, s_0, F \rangle$$

1. Finite set of states S
2. Set of input symbols Σ, the input alphabet, $\epsilon \notin \Sigma$ and $\Sigma_+ = \Sigma \cup \{\epsilon\}$
3. Transition function *move*: $S \times \Sigma_+ \to \mathcal{P}S$, gives from a state and symbol pair the next states
4. Initial state $s_0 \in S$ that is the start state or initial state
5. Set of states $F \subseteq S$ that are the accepting states or final states

The regular expression "$(a|b)*abb$" is described by the following NFA:



- $S = \{0, 1, 2, 3\}$
- $\Sigma = \{a, b\}$
- $s_0 = 0$
- $F = \{3\}$
- $move =$

S	Σ_+	S'
0	a	0 or 1
0	b	0
1	b	2
2	b	3

Inputs : An input string x terminated by **eof** character. A NFA N with start state s_0, accepting states F, and transition function `move`

Output : Answer "yes" if N accepts x; "no" otherwise

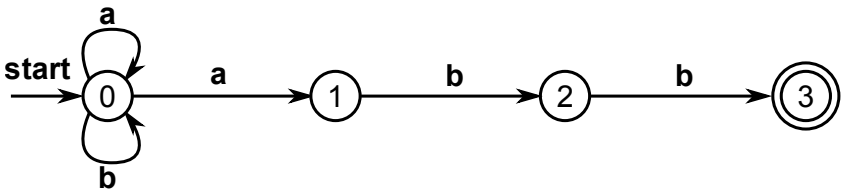
Behavior : The algorithm keeps a set of current states S, those that are reached from s_0 following a path labeled by the inputs read so far. If c is the next input character, read by the function `nextChar`, then we first compute `move` (S,c) and then close that set using ϵ-`closure`

begin
  $S \leftarrow \epsilon$-`closure` (s_0);
  $c \leftarrow$ `nextChar`;
  while $c \neq$ **eof do**
    $S \leftarrow \epsilon$-`closure` (`move` (S,c));
    $c \leftarrow$ `nextChar`;
  end
  return $S \cap F \neq \emptyset$;
end

Operation	Description
ϵ-closure(s)	States reachable from state s on ϵ-transitions
ϵ-closure(T)	States reachable from $\forall s \in T$ on ϵ-transitions

[Huffman, 1954, Moore, 1956, Thompson, 1968]

Let the input: "abababb"

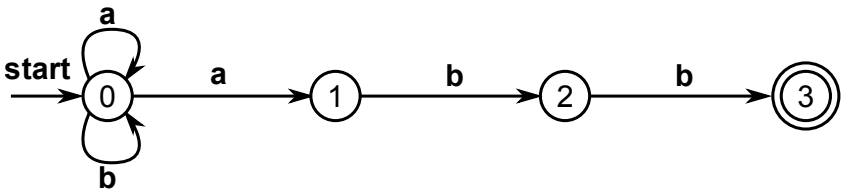



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
foward: abababb

Let the input: "abababb"

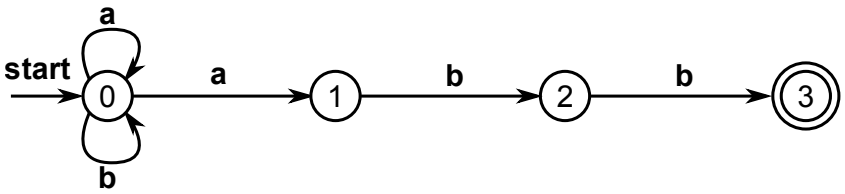



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
$c =$ a
foward: bababb

Let the input: "abababb"

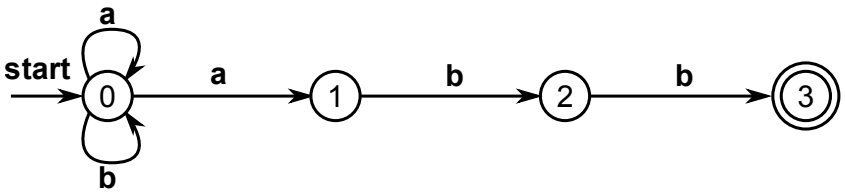



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
$c = \mathtt{a}$
$\text{move}(\{0\}, \mathtt{a}) = \{0, 1\}$
$\epsilon\text{-closure}(\{0, 1\}) = \{0, 1\}$
$S' = \{0, 1\}$

Let the input: "abababb"

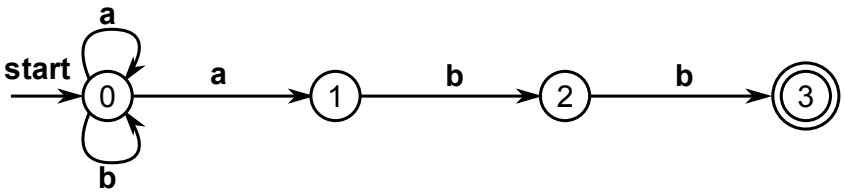



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 1\}$
$c =$ b
foward: ababb

Let the input: "abababb"

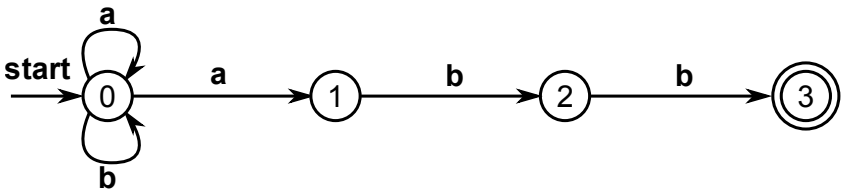



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 1\}$
$c = \mathrm{b}$
$\mathrm{move}(\{0, 1\}, \mathrm{b}) = \{0, 2\}$
$\epsilon\text{-closure}(\{0, 2\}) = \{0, 2\}$
$S' = \{0, 2\}$

Let the input: "abababb"

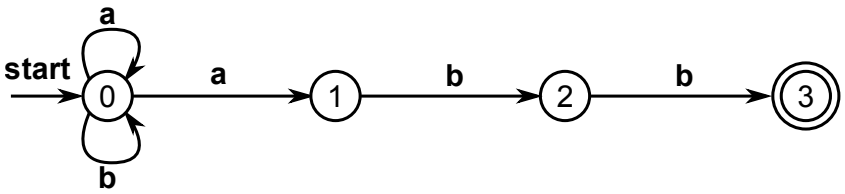



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 2\}$
$c =$ a
foward: babb

Let the input: "abababb"

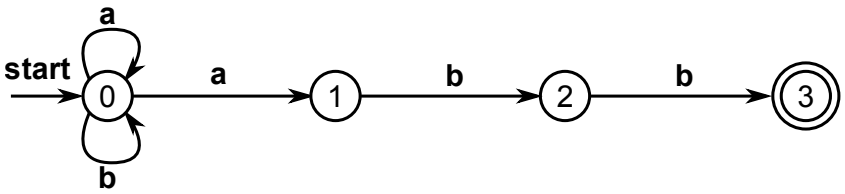



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 2\}$
$c = \mathrm{a}$
$\mathrm{move}(\{0, 2\}, \mathrm{a}) = \{0\}$
$\epsilon\text{-closure}(\{0\}) = \{0\}$
$S' = \{0\}$

Let the input: "abababb"

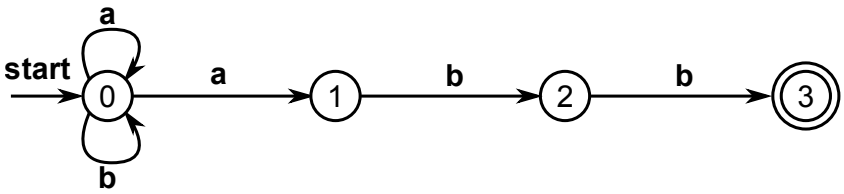


begin
  $S \leftarrow \epsilon$-closure (s_0);
  $c \leftarrow$ nextChar;
  while $c \neq$ **eof do**
    $S \leftarrow \epsilon$-closure (move (S,c));
    $c \leftarrow$ nextChar;
  end
  return $S \cap F \neq \emptyset$;
end

Example

$S = \{0\}$
$c =$ b
foward: abb

Let the input: "abababb"

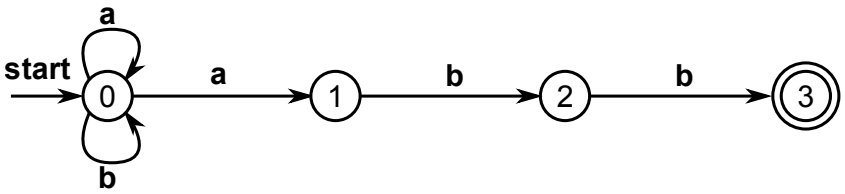



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
$c = \mathrm{b}$
$\mathrm{move}(\{0\}, \mathrm{b}) = \{0\}$
$\epsilon\text{-closure}(\{0\}) = \{0\}$
$S' = \{0\}$

Let the input: "abababb"

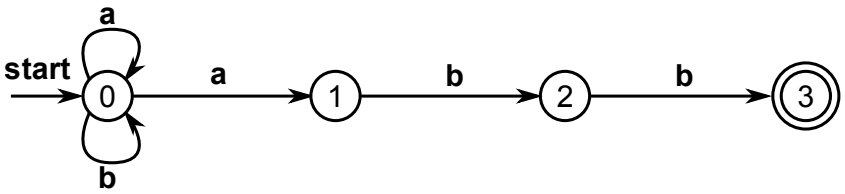



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
$c =$ a
foward: bb

Let the input: "abababb"

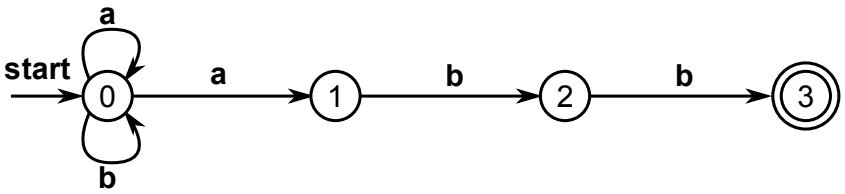



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0\}$
$c = \mathtt{a}$
$\text{move}(\{0\}, \mathtt{a}) = \{0, 1\}$
$\epsilon\text{-closure}(\{0, 1\}) = \{0, 1\}$
$S' = \{0, 1\}$

Let the input: "abababb"

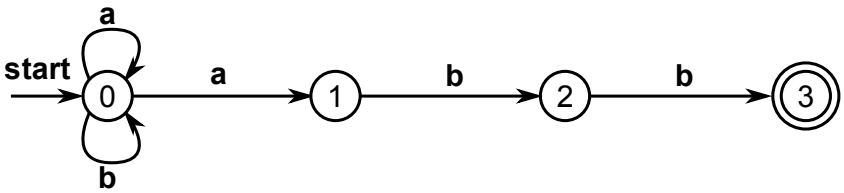



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 1\}$
$c =$ b
foward: b

Let the input: "abababb"

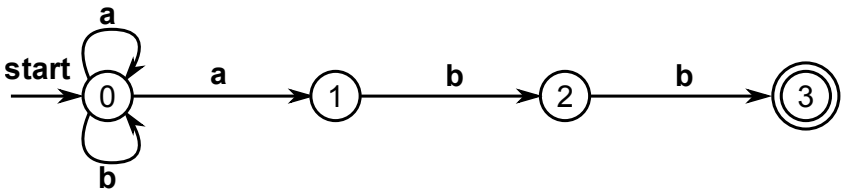



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 1\}$
$c = \mathrm{b}$
$\mathrm{move}(\{0, 1\}, \mathrm{b}) = \{0, 2\}$
$\epsilon\text{-closure}(\{0, 2\}) = \{0, 2\}$
$S' = \{0, 2\}$

Let the input: "abababb"

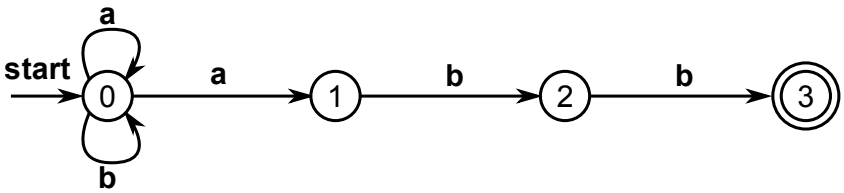



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 2\}$
$c =$ b
foward: **eof**

Let the input: "abababb"

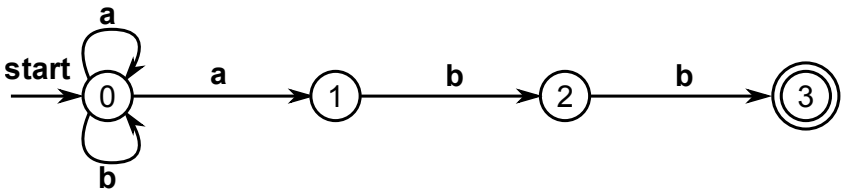



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 2\}$
$c = \mathrm{b}$
$\mathrm{move}(\{0, 2\}, \mathrm{b}) = \{0, 3\}$
$\epsilon\text{-closure}(\{0, 3\}) = \{0, 3\}$
$S' = \{0, 3\}$

Let the input: "abababb"

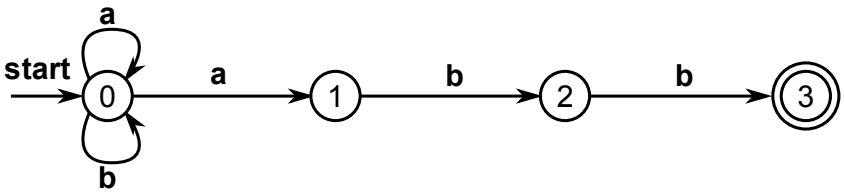



```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 3\}$
$c =$ **eof**

Let the input: "abababb"




```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 3\}$
$F = \{3\}$
$S \cap F = \{0, 3\} \cap \{3\} = \{3\}$
Return "true"

Deterministic Finite Automaton

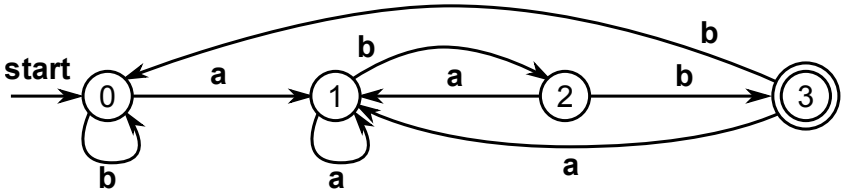
A special case of an NFA where:

1. There are no moves on input ϵ
2. For each state s and input symbol a, there is exactly one edge out of s labeled with a

- While the NFA is used to recognize the strings of a language, the DFA is a simple and concrete algorithm for recognizing strings
- Every regular expression and every NFA can be converted to a DFA accepting the same language

Lexical analyzers are built upon DFA

The regular expression "$(a|b) * abb$" is described by the following DFA:



- $S = \{0, 1, 2, 3\}$
- $\Sigma = \{a, b\}$
- $s_0 = 0$
- $F = \{3\}$
- $move =$

S	Σ_+	S'
0	a	1
0	b	0
1	a	1
1	b	2
2	a	1
2	b	3
3	a	1
3	b	0

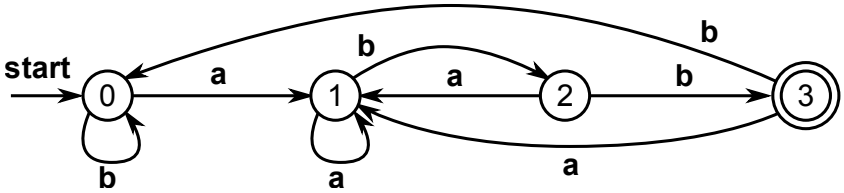
Inputs : An input string `x` terminated by **eof** character. A DFA D with start state s_0, accepting states F, and transition function move.

Output : Answer "yes" if D accepts `x`; "no" otherwise.

Behavior: Apply algorithm on `x`. The function `move` (s,c) gives the state to which there is an edge from state s on input c. The function `nextChar` returns the next character of the input string `x`

begin
    $s \leftarrow s_0$;
    $c \leftarrow$ `nextChar`;
    while $c \neq$ **eof do**
        $s \leftarrow$ `move` (s,c);
        $c \leftarrow$ `nextChar`;
    end
    return $s \in F$;
end

Let the input: "abababb"

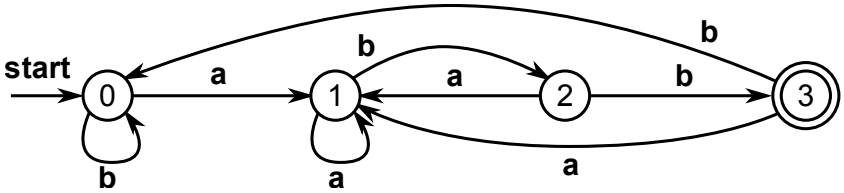



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 0$
foward: abababb

Let the input: "abababb"

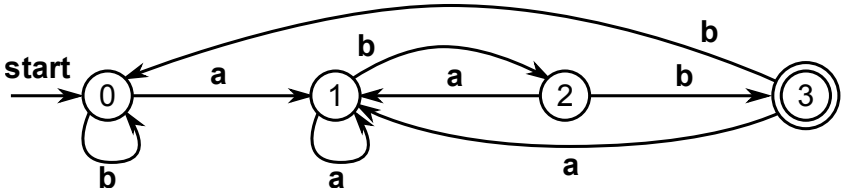



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 0$
$c =$ a
foward: bababb

Let the input: "abababb"

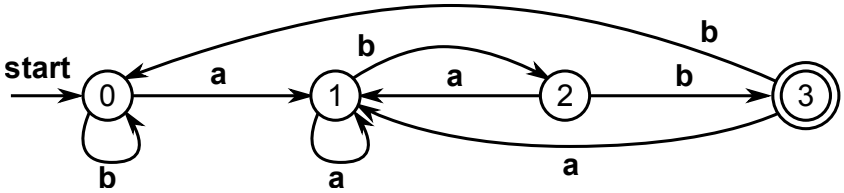



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 0$
$c = \mathrm{a}$
$\mathrm{move}(0, \mathrm{a}) = 1$
$s' = 1$

Let the input: "abababb"

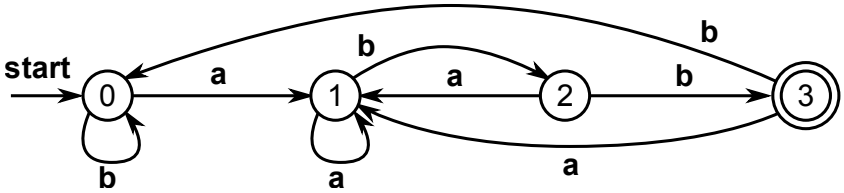



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c =$ b
foward: ababb

Let the input: "abababb"

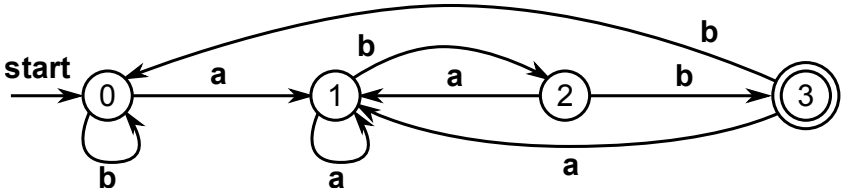



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c = \mathrm{b}$
$\mathrm{move}(1, \mathrm{b}) = 2$
$s' = 2$

Let the input: "abababb"

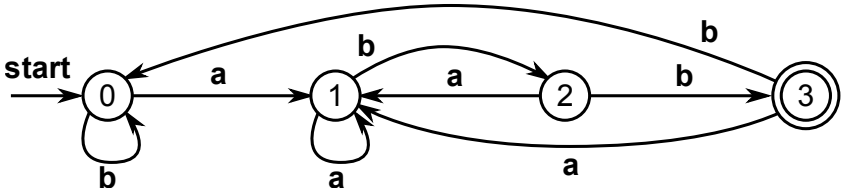



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c =$ a
foward: babb

Let the input: "abababb"

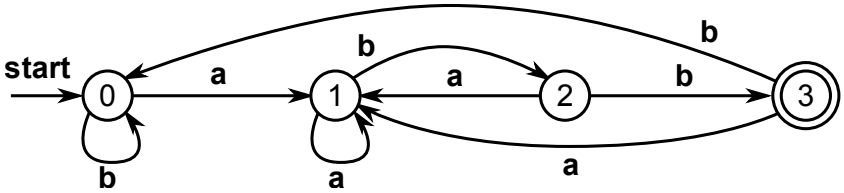



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c = \mathrm{a}$
$\mathrm{move}(2, \mathrm{a}) = 1$
$s' = 1$

Let the input: "abababb"

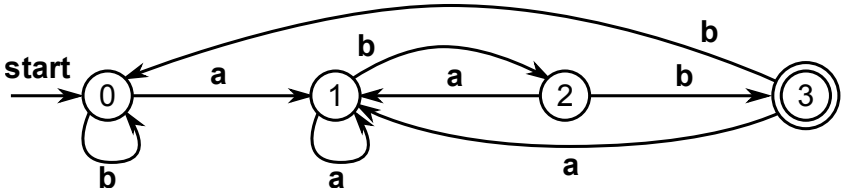



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c =$ b
foward: abb

Let the input: "abababb"

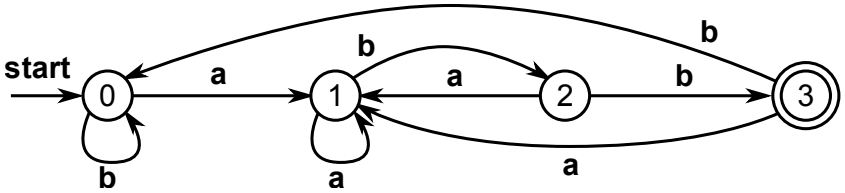



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c = \mathrm{b}$
$\mathrm{move}(1, \mathrm{b}) = 2$
$s' = 2$

Let the input: "abababb"

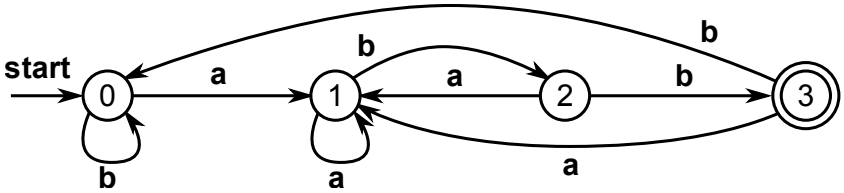



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c =$ a
foward: bb

Let the input: "abababb"

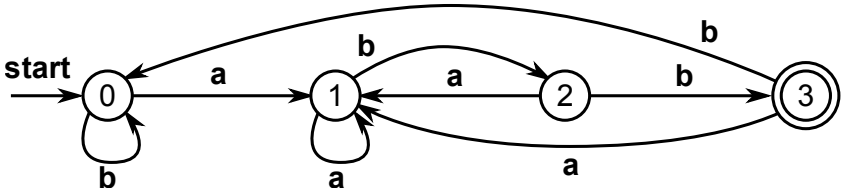



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c = \mathrm{a}$
$\mathrm{move}(2, \mathrm{a}) = 1$
$s' = 1$

Let the input: "abababb"

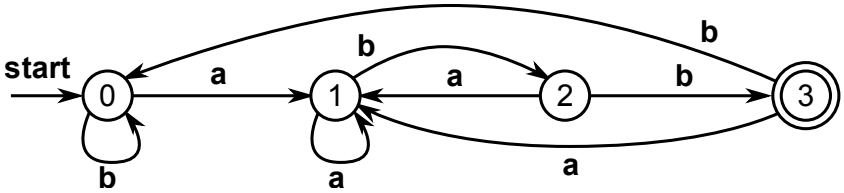



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c =$ b
foward: b

Let the input: "abababb"

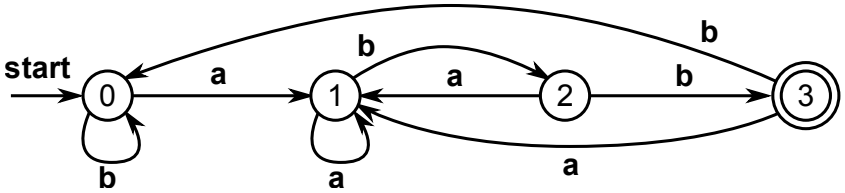



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 1$
$c = \mathrm{b}$
$\mathrm{move}(1, \mathrm{b}) = 2$
$s' = 2$

Let the input: "abababb"

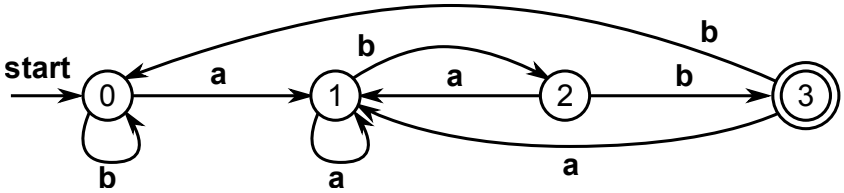



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c =$ b
foward: **eof**

Let the input: "abababb"

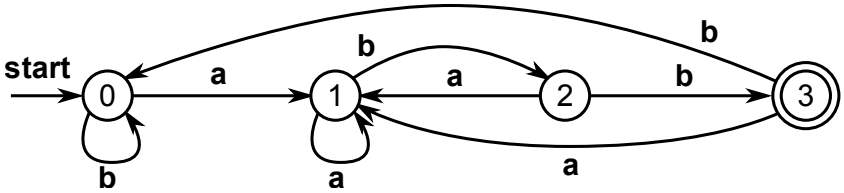



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 2$
$c = \mathrm{b}$
$\mathrm{move}(2, \mathrm{b}) = 3$
$s' = 3$

Let the input: "abababb"

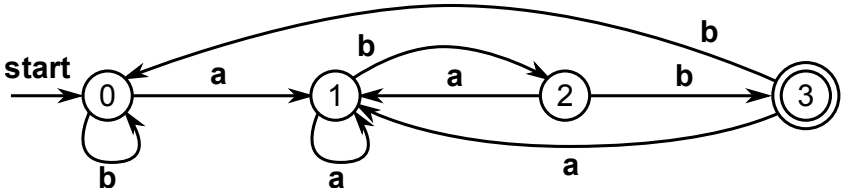



```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

$s = 3$
$c =$ **eof**

Let the input: "abababb"




```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

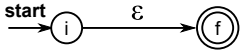
$s = 3$
$F = \{3\}$
Return "true"

Input : Regex r over alphabet S

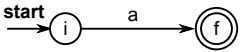
Output : NFA N accepting $L(r)$

Behavior: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of *basis rules* for handling subexpressions with no operators, and inductive rules for a constructing larger NFA from the NFAs for the immediate subexpressions of a given expression

Basis 1 : For each ϵ in r, construct the following NFA:



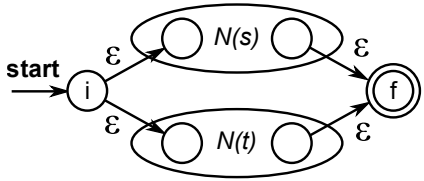
Basis 2 : For any subexpression a in Σ, construct the following NFA:



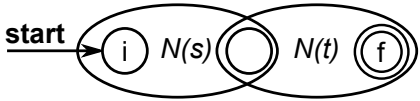
Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of ϵ or some a as a subexpression of r

[McNaughton and Yamada, 1960, Aho et al., 1988, Thompson, 1968]

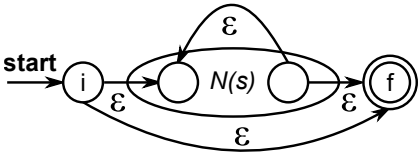
Induction 1: Suppose $r = s|t$. Then $N(r)$ is:



Induction 2: Suppose $r = st$. Then $N(r)$ is:

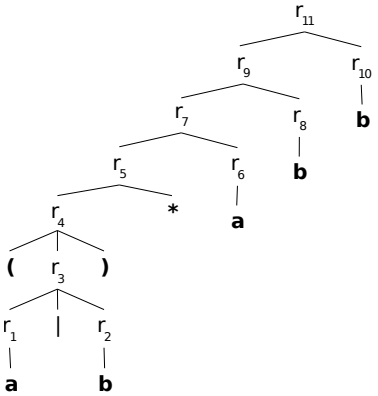


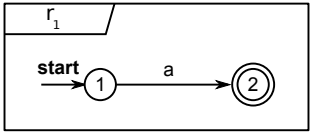
Induction 3: Suppose $r = s*$. Then $N(r)$ is:

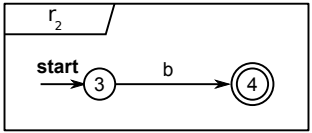


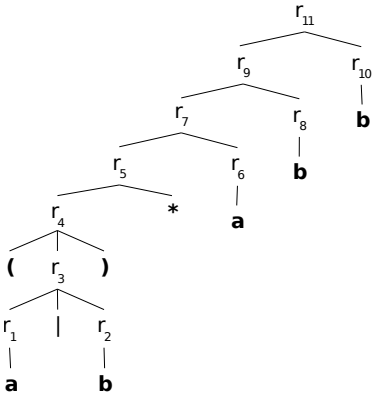
Induction 4: Suppose $r = (s)$. Then $N(r) = N(s)$

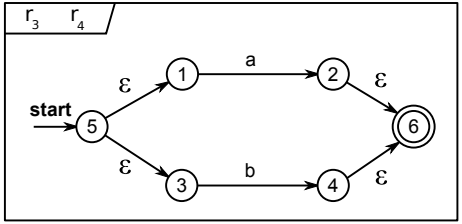
[McNaughton and Yamada, 1960, Aho et al., 1988, Thompson, 1968]

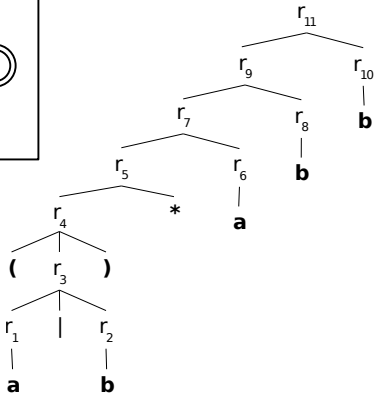

r11
r9
r10
r7
r8
b
r5
r6
b
r4
*
a
(
r3
)
r1
|
r2
a
b

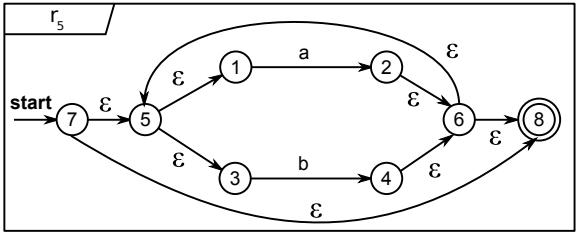

r1
start
1
a
2

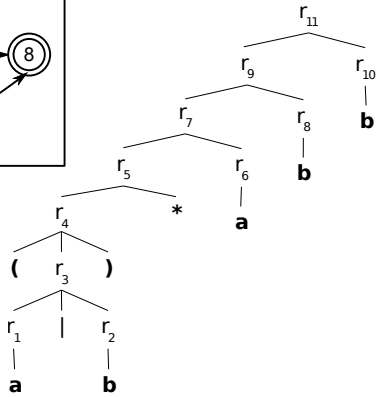

r2
start
3
b
4


r11
r9
r10
r7
r8
b
r5
r6
b
r4
*
a
(
r3
)
r1
|
r2
a
b


r3 r4
start
5
ε
1
a
2
ε
6
ε
3
b
4
ε


r11
r9
r10
r7
r8
b
r5
r6
b
r4
*
a
(
r3
)
r1
|
r2
a
b


r5
start
7
ε
5
ε
1
a
2
ε
ε
6
ε
8
ε
3
b
4
ε
ε


r11
r9
r10
r7
r8
b
r5
r6
b
r4
*
a
(
r3
)
r1
|
r2
a
b

etc.

Each state of the constructed DFA corresponds to a set of NFA states

Number of DFA states may be exponential
$\Rightarrow$ Difficulties to implement the DFA

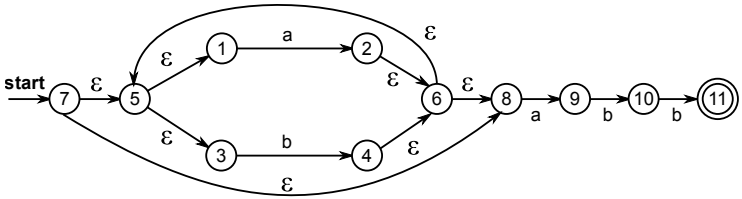
The conversion algorithm is described on the following slides

Input : NFA N
Output : DFA D accepting the same language as N
Behavior:

1. Algorithm constructs a transition table *Dtran* from D. Each state of D is a set of NFA states, and we construct *Dtran* so that D will simulate "in parallel" all the possible moves N can make on a given input string
2. NDA may be built from the table *Dtran*

```
begin
    T ← ε-closure (s0);
    Dstates ← {T};
    Unmarked ← {T};
    while ∃T ∈ Umarked do
        Unmarked ← Unmarked \ {T};
        foreach input symbol a do
            U ← ε-closure (move (T,a));
            if U ∉ DStates then
                Dstates ← Dstates ∪ {U};
                Unmarked ← Umarked ∪ {U};
            end
            Dtran[T, a] ← U;
        end
    end
end
```

Let consider the NFA for the regular expression $(a|b) * abb$.

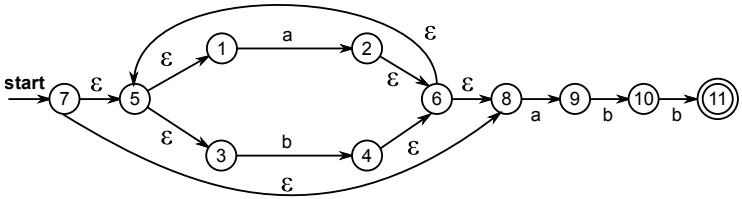


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$	$\times$		

Notes

$T = \epsilon\text{-closure}(s_0) = \epsilon\text{-closure}(7) = \{1, 3, 5, 7, 8\}$

Let consider the NFA for the regular expression $(a|b) * abb$.

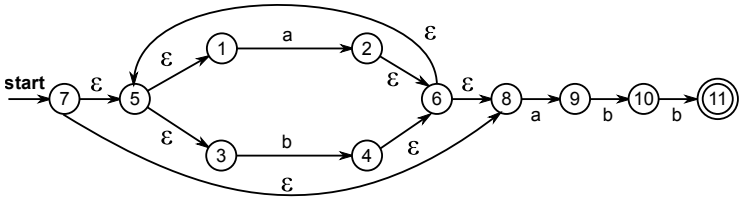


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1,3,5,7,8\}$		B	
B	$\{1,2,3,5,6,8,9\}$	$\times$		

Notes

$T = \{1,3,5,7,8\}$ and unmark T
$a =$ "a"
$U = \epsilon$-closure(move$(T,a)) = \epsilon$-closure$(\{2,9\}) = \{1,2,3,5,6,8,9\}$
U is a new state (B), and $Dtran[T,a]$ =B

Let consider the NFA for the regular expression $(a|b) * abb$.

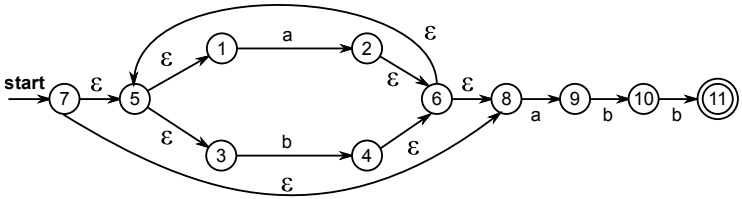


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1,3,5,7,8\}$		B	C
B	$\{1,2,3,5,6,8,9\}$	$\times$		
C	$\{1,3,4,5,6,8\}$	$\times$		

Notes

$T = \{1,3,5,7,8\}$
$a =$ "b"
$U = \epsilon$-closure(move(T, a)) = ϵ-closure($\{4\}$) = $\{1,3,4,5,6,8\}$
U is a new state (C), and $Dtran[T, a]$ =C

Let consider the NFA for the regular expression $(a|b) * abb$.

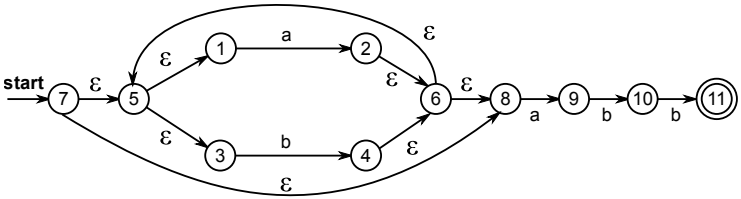


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1,3,5,7,8\}$		B	C
B	$\{1,2,3,5,6,8,9\}$		B	
C	$\{1,3,4,5,6,8\}$	$\times$		
		$\times$		

Notes

$T = \{1,2,3,5,6,8,9\}$ and unmark T
$a =$ "a"
$U = \epsilon$-closure(move(T, a)) $= \epsilon$-closure($\{2,9\}$) $= \{1,2,3,5,6,8,9\}$
U is B, $Dtran[T, a]$ =B

Let consider the NFA for the regular expression $(a|b) * abb$.

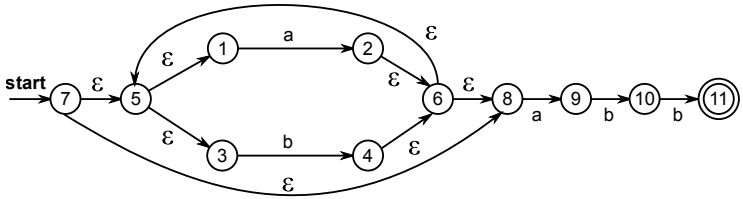


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$	$\times$		
D	$\{1, 3, 4, 5, 6, 8, 10\}$	$\times$		

Notes

$T = \{1, 2, 3, 5, 6, 8, 9\}$
$a =$ "b"
$U = \epsilon$-closure(move$(T, a)) = \epsilon$-closure$(\{4, 10\}) = \{1, 3, 4, 5, 6, 8, 10\}$
U is a new state (D), $Dtran[T, a] =$D

Let consider the NFA for the regular expression $(a|b) * abb$.

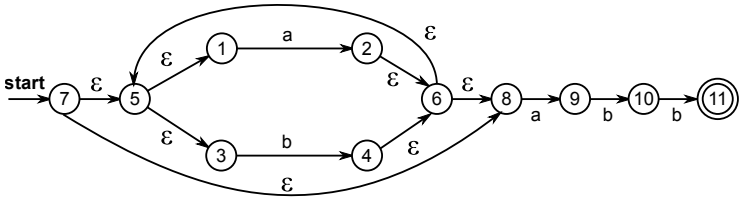


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$		B	
D	$\{1, 3, 4, 5, 6, 8, 10\}$	$\times$		

Notes

$T = \{1, 3, 4, 5, 6, 8\}$ and unmark T
$a =$ "a"
$U = \epsilon$-closure(move(T, a)) $= \epsilon$-closure($\{2, 9\}$) $= \{1, 2, 3, 5, 6, 8, 9\}$
U is B, $Dtran[T, a]$ =B

Let consider the NFA for the regular expression $(a|b) * abb$.

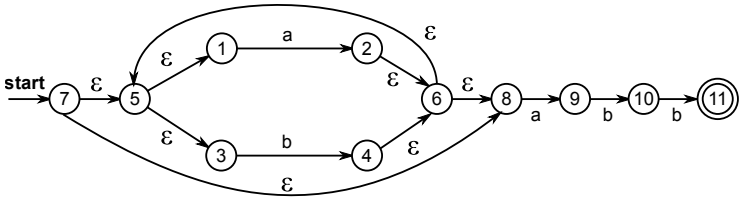


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$		B	C
D	$\{1, 3, 4, 5, 6, 8, 10\}$	$\times$		

Notes

$T = \{1, 3, 4, 5, 6, 8\}$
$a =$ "b"
$U = \epsilon$-closure(move(T, a)) $= \epsilon$-closure($\{4\}$) $= \{1, 3, 4, 5, 6, 8\}$
U is C, $Dtran[T, a]$ =C

Let consider the NFA for the regular expression $(a|b) * abb$.

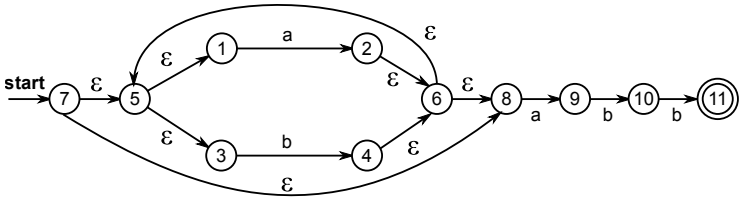


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$		B	C
D	$\{1, 3, 4, 5, 6, 8, 10\}$	$\times$	B	

Notes

$T = \{1, 3, 4, 5, 6, 8, 10\}$ and unmark T
$a =$ "a"
$U = \epsilon$-closure(move(T, a)) $= \epsilon$-closure($\{2, 9\}$) $= \{1, 2, 3, 5, 6, 8, 9\}$
U is B, $Dtran[T, a]$ =B

Let consider the NFA for the regular expression $(a|b) * abb$.

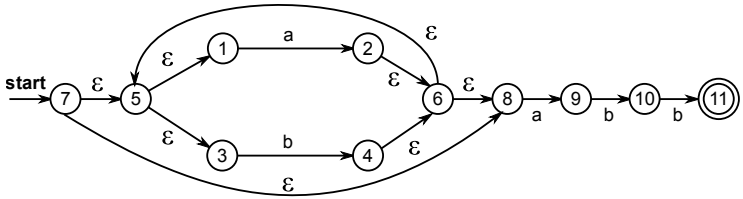


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$		B	C
D	$\{1, 3, 4, 5, 6, 8, 10\}$		B	E
E	$\{1, 3, 4, 5, 6, 8, 11\}$	$\times$		

Notes

$T = \{1, 3, 4, 5, 6, 8, 10\}$
$a =$ "b"
$U = \epsilon$-closure(move(T, a)) $= \epsilon$-closure($\{4, 11\}$) $= \{1, 3, 4, 5, 6, 8, 11\}$
U is a new state (E), $Dtran[T, a] =$E

Let consider the NFA for the regular expression $(a|b) * abb$.

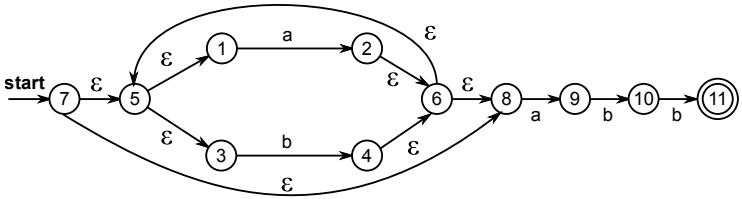


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1,3,5,7,8\}$		B	C
B	$\{1,2,3,5,6,8,9\}$		B	D
C	$\{1,3,4,5,6,8\}$		B	C
D	$\{1,3,4,5,6,8,10\}$		B	E
E	$\{1,3,4,5,6,8,11\}$		B	

Notes

$T = \{1,3,4,5,6,8,11\}$ and unmark T
$a =$ "a"
$U = \epsilon$-closure(move$(T,a)) = \epsilon$-closure$(\{2,9\}) = \{1,2,3,5,6,8,9\}$
U is B, $Dtran[T,a]$ =B

Let consider the NFA for the regular expression $(a|b) * abb$.

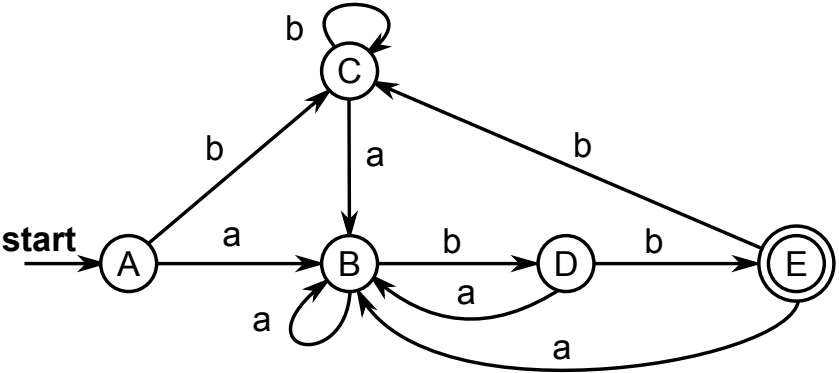


Label	*Dstates*	$\in$ *Unmarked*	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$		B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$		B	D
C	$\{1, 3, 4, 5, 6, 8\}$		B	C
D	$\{1, 3, 4, 5, 6, 8, 10\}$		B	E
E	$\{1, 3, 4, 5, 6, 8, 11\}$		B	C

Notes

$T = \{1, 3, 4, 5, 6, 8, 11\}$
$a =$ "b"
$U = \epsilon$-closure(move(T, a)) = ϵ-closure($\{4\}$) = $\{1, 3, 4, 5, 6, 8\}$
U is C, $Dtran[T, a]$ =C

Label	*Dstates*	Init.	Final	"a"	"b"
A	$\{1, 3, 5, 7, 8\}$	yes	$\emptyset$	B	C
B	$\{1, 2, 3, 5, 6, 8, 9\}$	no	$\emptyset$	B	D
C	$\{1, 3, 4, 5, 6, 8\}$	no	$\emptyset$	B	C
D	$\{1, 3, 4, 5, 6, 8, 10\}$	no	$\emptyset$	B	E
E	$\{1, 3, 4, 5, 6, 8, 11\}$	no	$\{11\}$	B	C

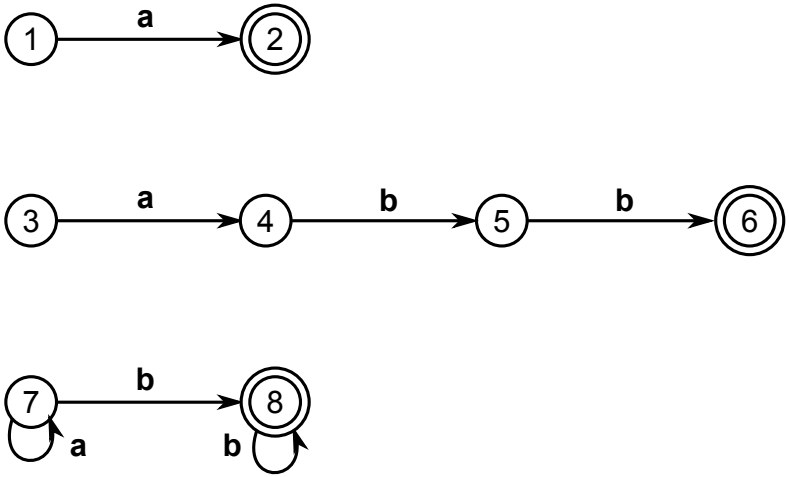


Automaton based on NFA

- Each regular-expression pattern is converted to an NFA
- Single global automaton combines all the NFA's in one

Example

```
a       { do_Action1(); }
abb     { do_Action2(); }
a*b+    { do_Action3(); }
```


1
a
2
3
a
4
b
5
b
6
7
b
8
a
b

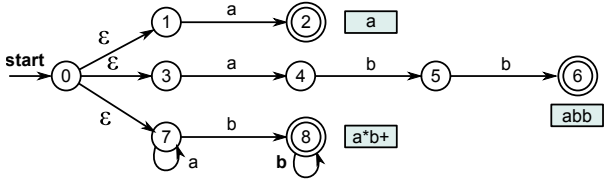
Lexical analyzer reads the input from `lexemeBegin`

NFA is evaluated according to the input pointed by the `forward` pointer

When the NFA simulation does not find any more state, we could find the longest validated lexeme:

- Look backwards in the sequence of sets of states, until accepting states were found
- If found accepting states, replies the associated lexeme
- Otherwise, there is a syntax error

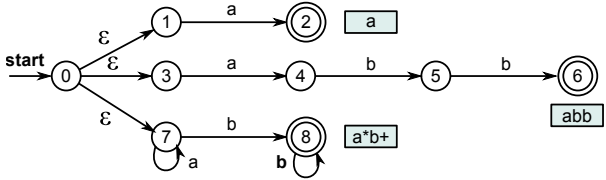
Let consider the input: aaba

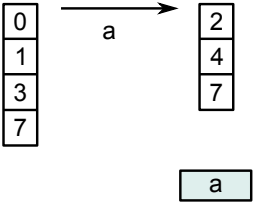


0
1
3
7

Initially, the set of states contains the ϵ-closure of the state 0.

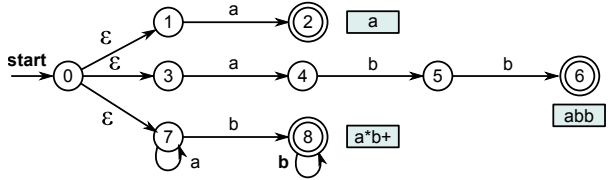
Let consider the input: aaba

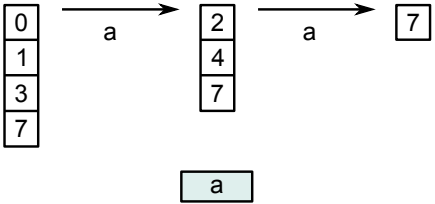




Read "a"
States: ϵ-closure(move($\{0, 1, 3, 7\}$, "a"))= $\{2, 4, 7\}$
State 2 is a final state $\Rightarrow$ lexeme detected for pattern *a*

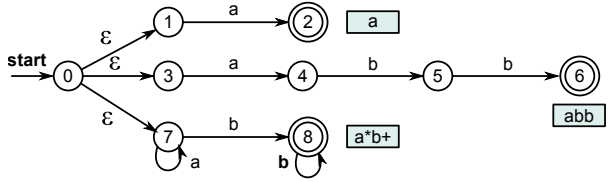
Let consider the input: aaba

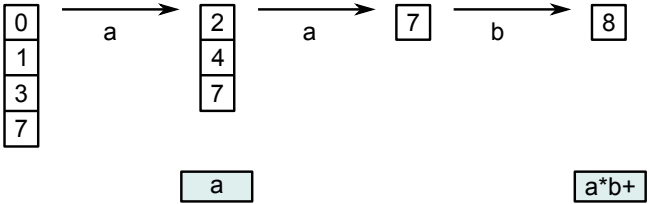




Read "a"
States: ϵ-closure(move($\{2, 4, 7\}$, "a"))$= \{7\}$

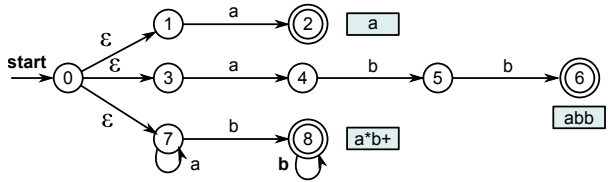
Let consider the input: aaba

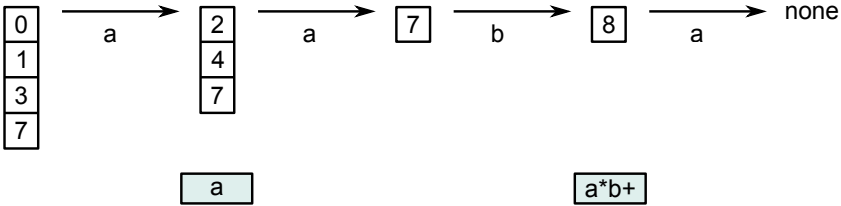




Read "b"
States: ϵ-closure(move({7}, "b"))= {8}
State 8 is a final state $\Rightarrow$ lexeme detected for pattern $a * b+$

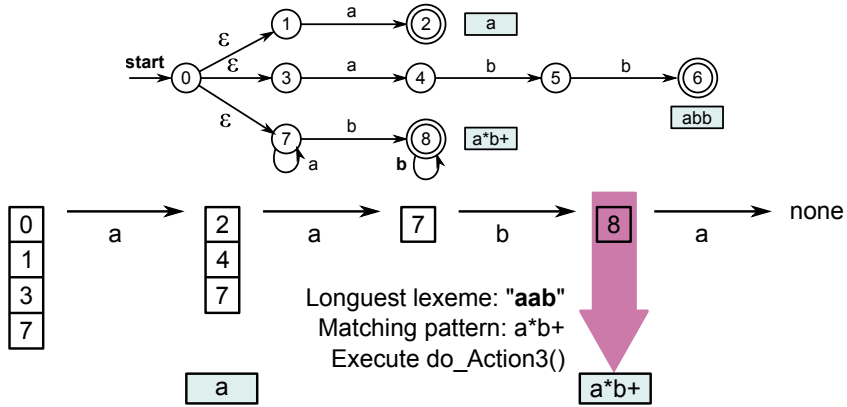
Let consider the input: aaba





Read "a"
States: ϵ-closure(move($\{8\}$, "a"))$= \emptyset$
Simulation is done. Look backward.

Let consider the input: aaba

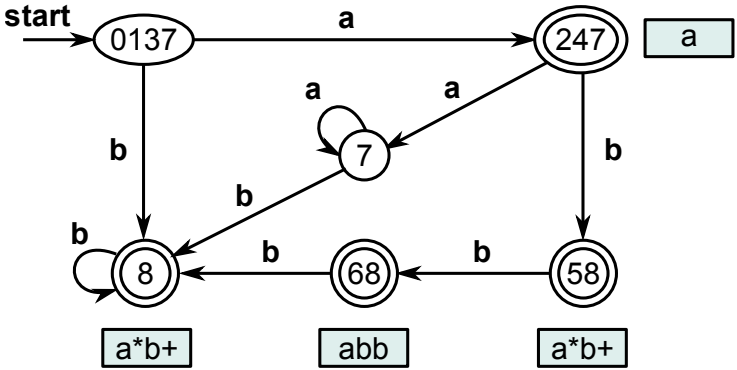


Automaton based on DFA

- Each regular-expression pattern is converted to an DFA (directly or via a NFA)
- For each DFA state, if there is one accepting NFA state, use the first pattern in the Lex program associated to the NFA states

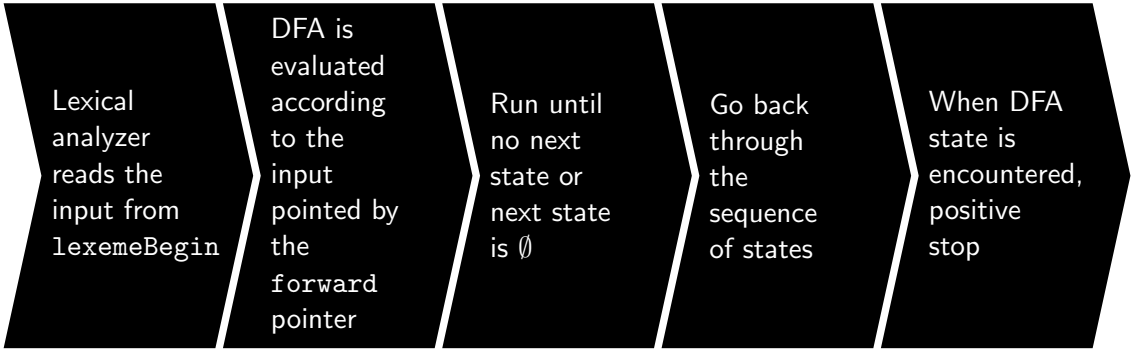
Example

```
a       { do_Action1(); }
abb     { do_Action2(); }
a*b+    { do_Action3(); }
```

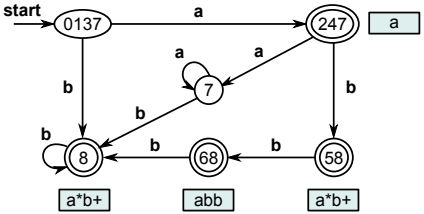


Note

Both states 6 and 8 are final states for patterns "abb" and "$a*b+$", resp
Only the first in the Lex program is considered by the NDA


Lexical analyzer reads the input from lexemeBegin
DFA is evaluated according to the input pointed by the forward pointer
Run until no next state or next state is ∅
Go back through the sequence of states
When DFA state is encountered, positive stop

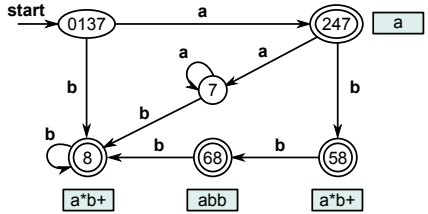
Let consider the input: aaba

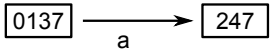


0137

Initially, the selected state is (0137)

Let consider the input: aaba

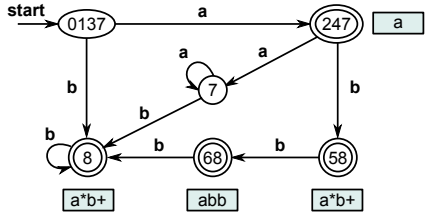


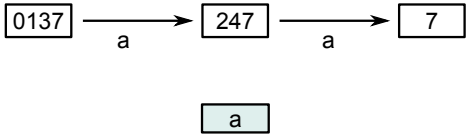


a

Read "a"
Pass the edge of the DFA, and update the current state
Because the state 2 is a final state in the NFA, the state (247) is marked

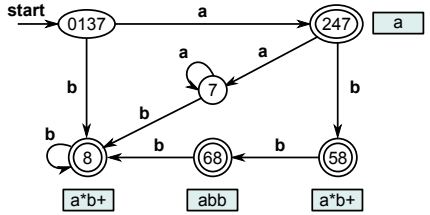
Let consider the input: aaba

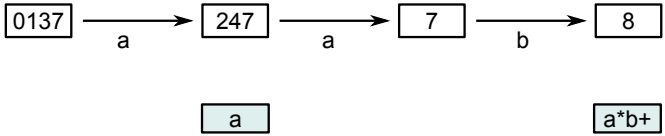




Read "a", and pass the edge

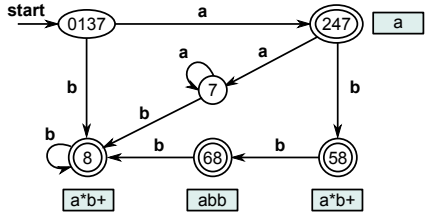
Let consider the input: aaba

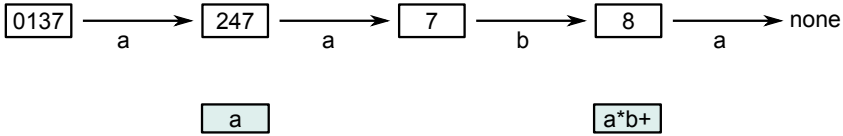




Read "b"
Pass the edge of the DFA, and update the current state
Because the state 8 is a final state in the NFA, the state (8) is marked

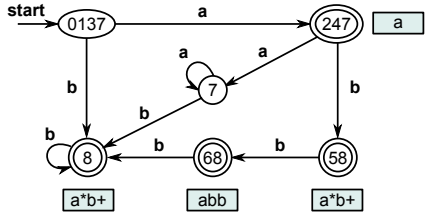
Let consider the input: aaba

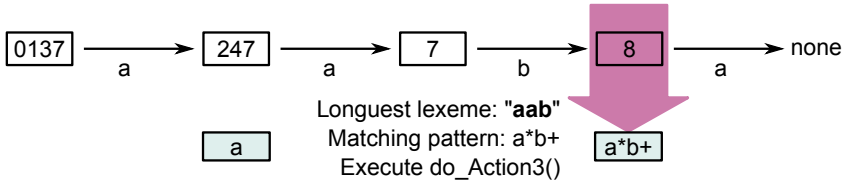




Read "a"
No state is accessible
Simulation is done. Look backward

Let consider the input: aaba





- Tokens: The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser.
- Lexemes: Each time the lexical analyzer returns a token to the parser, is has an associated lexeme: the sequence of characters that the token represents.
- Buffering: Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends, it is necessary for the lexical analyzer to buffer the input.
- Patterns: Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token.
- Regular Expressions: These expressions are commonly used to describe patterns. Regular expressions are built from single characters, using union, concatenation, and the Kleene closure.

- Transition Diagram: The behavior of a lexical analyzer can be described with a transition diagram. The states of that diagram represent the history of the characters seen during the analysis. The edges between the states indicate the possible next characters.
- Finite Automata: These are a formalization of transition diagrams. Accepting states indicates that a lexeme for a token has been found. Unlike transition diagrams, finite automata can make transitions on empty input as well as on input characters.
- Deterministic Finite Automata: A DFA is a special kind of finite automata that has exactly one transition out from each state for each input symbol.
- Nondeterministic Finite Automata: Automata that are not DFA are called nondeterministic.

- Conversion Among Pattern Representations: It is possible to convert any regular expression to NFA, and to convert any NFA to DFA.
- Lex: Family of software systems that are able to generate lexical analyzers from input specifications.

Aho, A. (1990).
Algorithms for finding patterns in strings.
Handbook of Theoretical Computer Science, A.

Aho, A., Kernighan, W., and Weinberger, P. (1988).
The AWK Programming Language.
Addison-Wesley, Boston, MA.

Hopcroft, J., Motwani, R., and Ullman, J. (2006).
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Boston, MA.

Huffman, D. (1954).
The synthesis of sequential machines.
J. Franklin Inst., 257:3–4, 161, 190, 375–303.

Kleene, S. (1956).
Representation of events in nerve nets, pages 3–40.
In [Shannon and McCarthy, 1956].

Lesk, M. (1975).
Lex - a lexical analyzer generator.
Computing Science Tech. Report 39, Bell Laboratories, Murray Hill, NJ.

McCullough, W. and Pitts, W. (1943).
A logical calculus of the ideas immanent in nervous activity.
Bull. Math. Biophysics, 5:115–133.

McNaughton, R. and Yamada, H. (1960).
Regular expressions and state graphs for automata.
IRETrans. on Electronic Computers, 1(1):38–47.

Moore, E. (1956).
Gedanken Experiments on Sequential Machines, pages 129–153.
In [Shannon and McCarthy, 1956].

Shannon, C. and McCarthy, J., editors (1956).
Automata Studies.
Princeton University Press.

Thompson, K. (1968).
Regular expression search algorithm.
Comm. ACM, 11(6):419–422.

Chapter 3
Syntax Analysis

Stéphane GALLAND

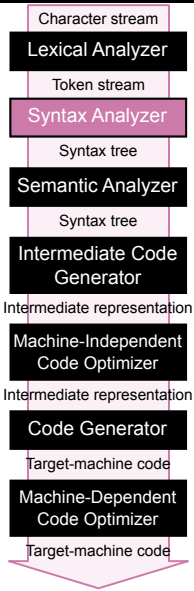
Every programming language has precise rules that prescribe the syntactic structure of well-formed programs

Language syntax is specified by context-free grammars or Backus-Naur Form (BNF)

Reads a stream of tokens

Do a semantic analysis

Outputs a syntax tree



[Chomsky, 1956, Backus, 1959, Naur, 1963]

Syntax

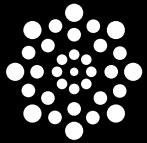
Set of rules that defines the combinations of symbols that are considered to be correctly structured statements or expressions in that language

Grammar

For text-based computer languages, a grammar gives a precise, easy-to-understand, syntactic specification of a programming language

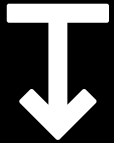
Semantics

Syntax therefore refers to the form of the code, and is contrasted with semantics: the meaning



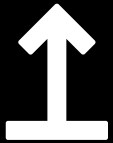
Universal Parsers

Build syntax tree as a whole



Top-down Parsers

Build the syntax tree from the root rule to tokens



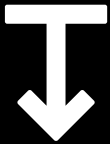
Bottom-up Parsers

Build the syntax tree from the tokens to the root rule

[Younger, 1967, Kasami, 1965, Earley, 1970]

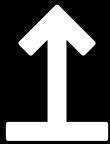
Universal Parsers

Algorithms, e.g., Cocke-Younger-Kasami and Earley, are too inefficient



Top-down Parsers

Efficient with Left-Left (LL) and Left-Right (LR) grammars



Bottom-up Parsers

Efficient with Left-Left (LL) and Left-Right (LR) grammars

$LL \rightarrow$ writing a parser by hands — $LR \rightarrow$ automatic generation of parser

[Younger, 1967, Kasami, 1965, Earley, 1970]

Compiler assists programmers in locating and tracking errors

Most programming language specifications do not describe how a compiler should respond to errors

Error handling is left to the compiler designer

[Dain, 1991]

1. *LL* and *LR* methods permits to detect errors efficiently and as soon as possible

2. Many errors appear syntactic, whatever they cause, and avoid the code generation

[Dain, 1991]

Report errors

Report the presence of errors clearly and accurately

01

Recover consistent state

Recover from each error quickly to detect subsequent errors

02

Efficient

Add minimal overhead to the processing of correct programs

03

[Dain, 1991]

Lexical Errors

Invalid lexemes, e.g., misspellings of identifiers, keywords, or operators; and missing quotes around text intended as a string

Syntactic errors

Invalid tokens that are broking the grammar, e.g., misplaced semicolons or extra or missing braces. Another example is a **case** outside an enclosing **switch** block

Semantic errors

Incorrect usage of the language elements in the syntax tree, e.g., type mismatches between operators and operands

Logical errors

Incorrect reasoning of the programmer, e.g., the use of the operator "=" in place of the operator "=="; or unreachable code

Once an error is detected, how should the parser recover?

Simplest approach: parser quits with an informative error message when it detects the first error; additional errors are uncovered

If errors are piled up, compiler stops after exceeding some limit

Two error recovery strategies are usually used:

Parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found

Synchronizing tokens are usually delimiters (semicolons or closing braces)

Simple to implement and ensure not to go into an infinite loop

Local correction on the remaining input

Replace the prefix of the remaining input by some string that allows the parser to continue

Examples

Replacing a coma by a semicolon, remove extraneous semicolon, or insert a missed semicolon

The major drawback of the phrase-level recovery is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection

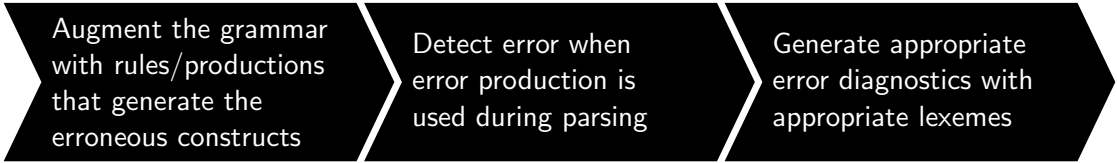
Principle of Global Corrections

- Given an incorrect input string x and grammar G
- Find a syntax tree for a related string y
- Condition: number of insertions, deletions, and changes of tokens required to transform x to y is as small as possible

This method is usually too costly in time and space

Global corrections has been used to

- Evaluate error-recovery algorithms
- Find optimal replacement strings for phrase-level recovery


Augment the grammar with rules/productions that generate the erroneous constructs
Detect error when error production is used during parsing
Generate appropriate error diagnostics with appropriate lexemes

A formal grammar consists of productions, that consist of terminals, nonterminals; and a start symbol

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$$
$$S \rightarrow \mathbf{b}\ \mathbf{a}$$

[Chomsky, 1956, Backus, 1959, Naur, 1963, Ingerman, 1967, Haggard et al., 2006]

A formal grammar consists of productions, that consist of terminals, nonterminals; and a start symbol

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$$
$$S \rightarrow \mathbf{b}\ \mathbf{a}$$

Terminal — Token Name

The basic symbols from which strings are formed
It could be assimilated to a token, replied by the lexical analyzer (see Chapter 2)

A formal grammar consists of productions, that consist of terminals, nonterminals; and a start symbol

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$$
$$S \rightarrow \mathbf{b}\ \mathbf{a}$$

Nonterminals

Syntactic variables that denote sets of strings that generate the language
Nonterminals impose a hierarchical structure on the language
Nonterminals must be defined in the grammar itself

A formal grammar consists of productions, that consist of terminals, nonterminals; and a start symbol

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$$
$$S \rightarrow \mathbf{b}\ \mathbf{a}$$

Production

Productions specify the manner in which the terminals and nonterminals can be combined to form strings

Each production consists of:

1. A nonterminal called the head or left side
2. The symbol "$\rightarrow$" (or "::=", or "$\models$")
3. A body, or right side, consisting of zero or more terminals and nonterminals, describing a replacement for the head

A formal grammar consists of productions, that consist of terminals, nonterminals; and a start symbol

$$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$$
$$S \rightarrow \mathbf{b}\ \mathbf{a}$$

Start Symbol

Nonterminal from which all the language's strings could be revided
Conventionally: the head of the first production is the start symbol

$$G = \langle N, \Sigma, P, S \rangle$$

- Finite set N of nonterminal symbols, that is disjoint with the strings formed from G
- Finite set Σ of terminal symbols that is disjoint from N
- Finite set P of production rules, each rule of the form
  $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
  where $*$ is the Kleene star operator
- Symbol $S \in N$ that is the start symbol, also called the sentence symbol
- Empty string is represented by ϵ (or λ, or $\perp$)

The language of G, denoted as $L(G)$, is defined as the set of sentences built by G

[Chomsky, 1956]

Terminals

- Lowercase letters early in the alphabet, such as a, b, c, ...
- Operator symbols, such as $+$, *, ...
- Punctuation symbols, such as parentheses, commas, ...
- Digits 0, ..., 9
- Boldface strings, such as **id** or **number**
- Underlined strings, such as id or number

Nonterminals

- Uppercase letters, such as A, B, C, ...
- The letter S which, when it appears, is usually the start symbol
- Lowercase, italic names, such as *expression*, *factor*, ...
- Enclosed names, e.g. $\langle \text{expression} \rangle$

Productions

A set of productions $A \to a_1, A \to a_2, \ldots, A \to a_k$ with a common head A (call them A-productions), may be written $A \to a_1 \mid a_2 \mid \ldots \mid a_k$

Others Notations

- Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols that is, either nonterminals or terminals
- Lowercase letters late in the alphabet, chiefly u, v, ..., z, represent (possibly empty) strings of terminals
- Lowercase Greek letters α, β, ..., represent (possibly empty) strings of grammar symbols

Parsing

Parsing is the process of taking a string of terminals and figuring out how to **derive** it from the start symbol
If the string cannot be derived, the parser reports a syntax error

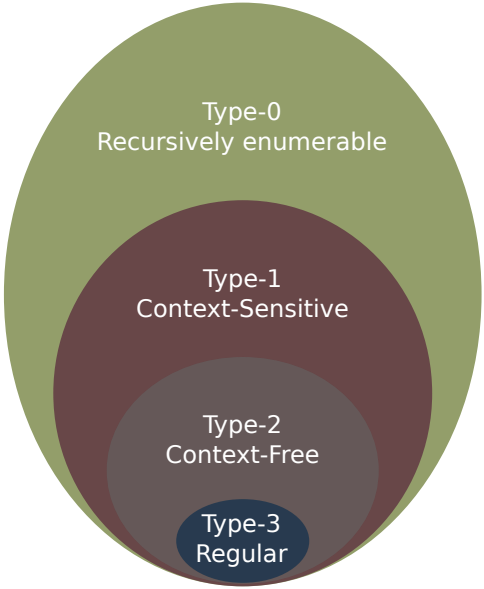
Grammar derives strings by beginning with the start symbol

Repeated replacement of a nonterminal by the body of a production for that nonterminal

Terminal strings, that can be derived, form the language defined by the grammar, namely, $L(G)$

- Arithmetic expressions are defined by the following grammar.
- The terminals are:
  - Operators: +, -, *, /, (,)
  - Numbers: **number** stands for any number
  - Identifier: **id** stands for any variable's name
- The grammar is:

$$
\begin{aligned}
\langle \text{expression} \rangle &::= \langle \text{expression} \rangle \mathbf{+} \langle \text{term} \rangle \\
&::= \langle \text{expression} \rangle \text{-} \langle \text{term} \rangle \\
\langle \text{term} \rangle &::= \langle \text{term} \rangle \mathbf{*} \langle \text{factor} \rangle \\
&::= \langle \text{term} \rangle \mathbf{/} \langle \text{factor} \rangle \\
&::= \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &::= \mathbf{(} \langle \text{expression} \rangle \mathbf{)} \\
&::= \textbf{number} \\
&::= \textbf{id}
\end{aligned}
$$



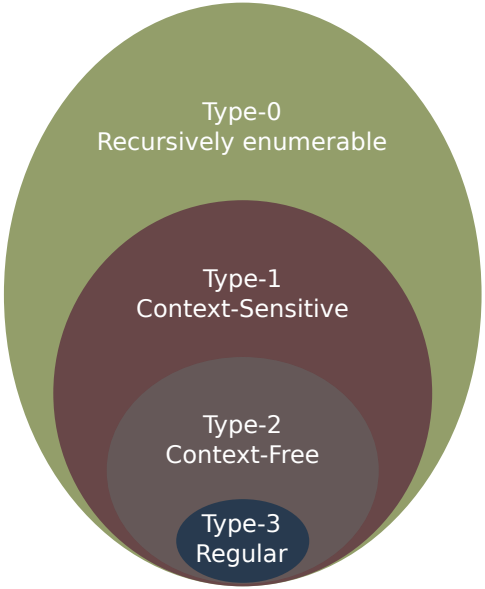
Type-0 grammars generates languages that can be recognized by a Turing machine

$\gamma \rightarrow \alpha$

Example: $L = \{w|w\}$ describes a terminating Turing machine

Recognition Complexity: NP-hard

[Chomsky, 1956]



Type-1 grammars generate context-sensitive languages

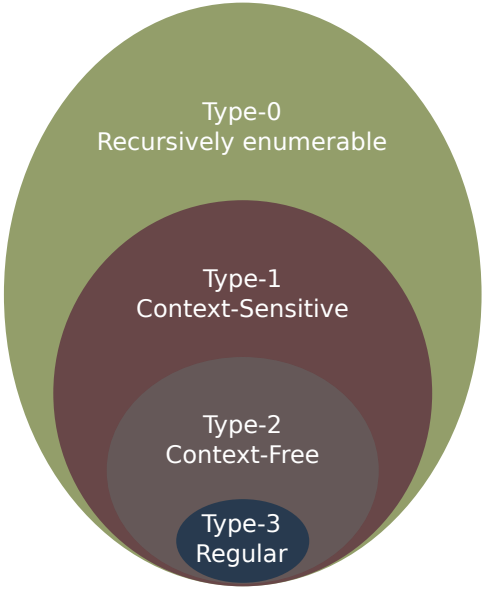
$\alpha A \beta \rightarrow \alpha \gamma \beta$
with $\gamma \neq \epsilon$

Example:
$L = \{a^n b^n c^n | n > 0\}$

Example: natural languages

Recognition Complexity:
NP-hard

[Chomsky, 1956]



Type-2 grammars generate the context-free languages

$A \rightarrow \alpha$

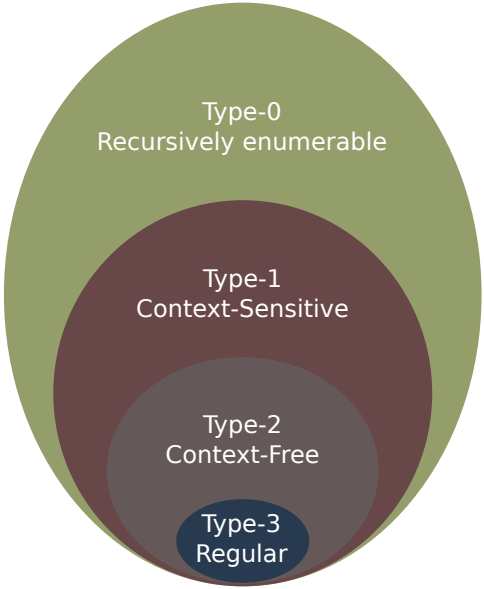
Example: $L = \{a^n b^n | n > 0\}$

Example: Most of programming languages

Recognition Complexity: $O(n^3)$

The rest of this lecture focuses on context-free grammars

[Chomsky, 1956]



Type-3 grammars generate the regular languages

$A \rightarrow \mathbf{a}$
$A \rightarrow \mathbf{a}B$

Example: $L = \{a^n | n \geq 0\}$

Example: Regex pattern specification

Recognition Complexity: $O(n)$

[Chomsky, 1956]

Production Rule Application

Rule application replaces the production's nonterminal by its body

Formally: string v is a result of applying the rule $\alpha \to \beta$ to string u if $\exists \alpha \to \beta \in P$ and $\exists u_1, u_2 \in (N \cup \Sigma)^*$, such that $u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$

Notation: $\alpha \Rightarrow \beta$

Example

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \text{-}\langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \textbf{id}
\end{aligned}
$$

- Input is: `-variable`
- The replacement of $\langle \mathrm{E} \rangle$ by -$\langle \mathrm{E} \rangle$ will be described by writing:

$$\langle \mathrm{E} \rangle \Rightarrow \text{-}\langle \mathrm{E} \rangle$$

- It means: $\langle \mathrm{E} \rangle$ derives in one step to -$\langle \mathrm{E} \rangle$

Repetitive Rule Application

Sequence of derivations $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k$ rewrites u_1 to u_n

Formally: string u derives to string v if $\exists k \in \mathbb{N}+$ and $\exists u_1, \cdots, u_k \in (N \cup \Sigma)^*$ such that $u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k$, $u = u_1$ and $v = u_k$

Notation 1: $u_1 \overset{*}{\Rightarrow} u_k$ (reflexive transitive closure)

Notation 2: if $k \geq 2$, $u_1 \overset{+}{\Rightarrow} u_k$ (transitive closure)

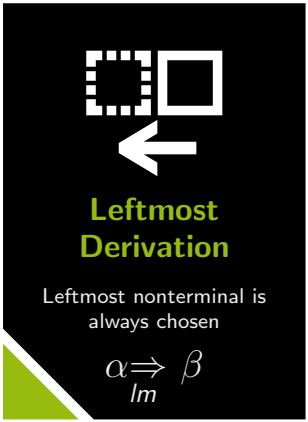
Properties

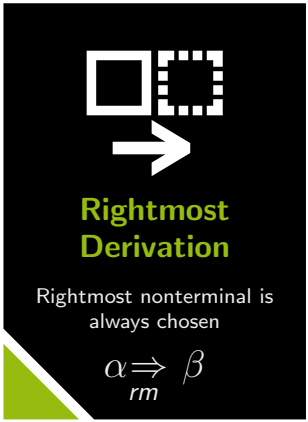
- Identity: $\alpha \overset{*}{\Rightarrow} \alpha$, for any string α
- Transitivity: If $\alpha \overset{*}{\Rightarrow} \beta$, and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$

- If $S \stackrel{*}{\Rightarrow} a$, where $\langle \mathrm{S} \rangle$ is the start symbol of a grammar G, we say that a is a sentential form of G
- A sentence of G is a sentential form, which is nonterminal
- The language generated by G is its set of sentences
- A string of terminals **w** is in $L(G)$ iff $S \stackrel{*}{\Rightarrow} w$
  Thus $L(G)$ is said to be a context-free language

At each step in a derivation, there are two choices to be made:

1. Choose which nonterminal to replace
2. Pick a production with that nonterminal as head





- **Grammar:**

$$
\begin{aligned}
\langle \mathrm{E} \rangle \quad &::= \quad \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \quad \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \quad (\langle \mathrm{E} \rangle) \\
&::= \quad \mathbf{id}
\end{aligned}
$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$\langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle$$

Right-most Derivations:

$$\langle \mathrm{E} \rangle \underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle$$

- **Grammar:**

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id} + \langle \mathrm{E} \rangle
\end{aligned}
$$

Right-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle
\end{aligned}
$$

- **Grammar:**

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id} + \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id} + \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle
\end{aligned}
$$

Right-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle * \mathbf{id}
\end{aligned}
$$

- **Grammar:**

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\
&::= \mathbf{(} \langle \mathrm{E} \rangle \mathbf{)} \\
&::= \mathbf{id}
\end{aligned}
$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id+} \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id+} \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\
&\underset{lm}{\Rightarrow} \mathbf{id+id*} \langle \mathrm{E} \rangle
\end{aligned}
$$

Right-most Derivations:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \mathbf{*id} \\
&\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+id*id}
\end{aligned}
$$

- **Grammar:**

$$\begin{aligned}\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\ &::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\ &::= (\langle \mathrm{E} \rangle) \\ &::= \mathbf{id}\end{aligned}$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$\begin{aligned}\langle \mathrm{E} \rangle &\underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id} + \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id} + \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id} + \mathbf{id} * \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}\end{aligned}$$

Right-most Derivations:

$$\begin{aligned}\langle \mathrm{E} \rangle &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle * \mathbf{id} \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle + \mathbf{id} * \mathbf{id} \\ &\underset{rm}{\Rightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}\end{aligned}$$

- **Grammar:**

$$\begin{aligned}\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\ &::= \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\ &::= \mathbf{(} \langle \mathrm{E} \rangle \mathbf{)} \\ &::= \mathbf{id}\end{aligned}$$

- Input string:
  `2 + 4 * 6`
- List of tokens:
  id+id*id

Left-most Derivations:

$$\begin{aligned}\langle \mathrm{E} \rangle &\underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id+}\langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id+}\langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id+id*}\langle \mathrm{E} \rangle \\ &\underset{lm}{\Rightarrow} \mathbf{id+id*id}\end{aligned}$$

Right-most Derivations:

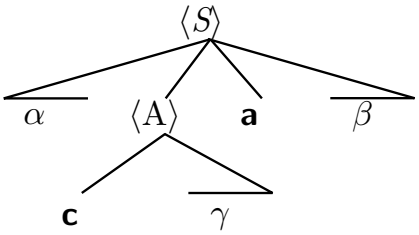
$$\begin{aligned}\langle \mathrm{E} \rangle &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \mathbf{*id} \\ &\underset{rm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+id*id} \\ &\underset{rm}{\Rightarrow} \mathbf{id+id*id}\end{aligned}$$

Grammar is ambiguous because the following derivation is possible on the input:

$$\langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \mathbf{id+}\langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \mathbf{id+id*}\langle \mathrm{E} \rangle \underset{lm}{\Rightarrow} \mathbf{id+id*id}$$

Parse Tree

A parse tree shows how the start symbol of a grammar derives a string
It is a graphical representation of the productions on an input string of tokens



- The root is labeled by the start symbol
- Each leaf is labeled by a terminal or by ϵ
- Each interior node is labeled by a nonterminal
- If A is the nonterminal of some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \ldots X_n$

Parsing is the process of building a parse tree from a string of tokens

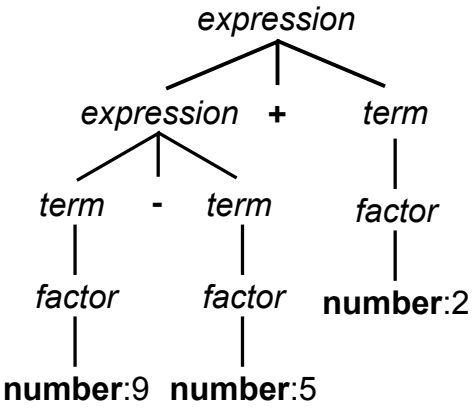
Let the string to parse:

```
9 - 5 + 2
```

Let the grammar:

⟨expression⟩	::=	⟨expression⟩**+**⟨term⟩
	::=	⟨expression⟩**-**⟨term⟩
	::=	⟨term⟩
⟨term⟩	::=	⟨term⟩*****⟨factor⟩
	::=	⟨term⟩**/**⟨factor⟩
	::=	⟨factor⟩
⟨factor⟩	::=	**(**⟨expression⟩**)**
	::=	**number**
	::=	**id**

The parse tree is:



Inputs : A sequence of tokens T. A grammar G with the start symbol s_0.
Output : A parse tree that corresponds to T and G.

```
begin
    r ← node (s0, T) ; L ← [r] ; input[r] ← T ;
    while L = [n].L′ do                                   /* Leftmost derivation */
        L ← L′ ;
        if ∃(label(n) → b) ∈ G|input[n] matches b then
            foreach αsβ = b do
                m ← ω ∈ T|(input[α] ω input[β]) = input[n] ;
                c ← node (s) ;
                addChild (n,c) ;
                if s is nonterminal then
                    L ← L.[c] ;
                    input[c] ← m ;
                end
            end
        end
    end
    return r
end
```

Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle \mathbf{+} \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle \mathbf{*} \langle \mathrm{E} \rangle \\
&::= \mathbf{-}\langle \mathrm{E} \rangle \\
&::= \mathbf{(}\langle \mathrm{E} \rangle \mathbf{)} \\
&::= \mathbf{id}
\end{aligned}
$$

Parse tree is:

E

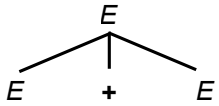
Tokens: **id+id*id**

$L = [E]$

Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \text{-} \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

Parse tree is:



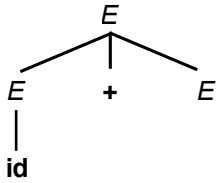
Tokens: **id+id*id**

$L = [n].L' = [E]$
$input = \mathbf{id+id*id}$
$b = E_0 + E_1$
$input_{E_0} = \mathbf{id}$
$input_{E_1} = \mathbf{id*id}$
$L = [E_0, E_1]$

Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= -\langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

Parse tree is:



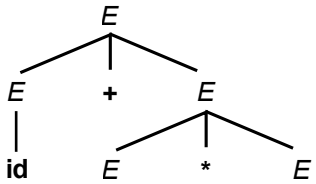
Tokens: **id+id*id**

$L = [n].L' = [E, E]$
$input =$ **id**
$b =$ **id**
$L = [E]$

Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \text{-} \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

Parse tree is:



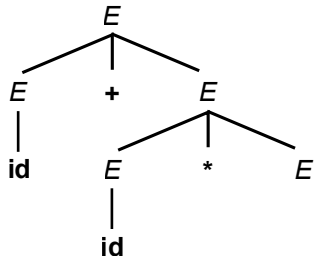
Tokens: **id+id*id**

$L = [n].L' = [E]$
$input =$ **id*id**
$b = E_0$***** E_1
$input_{E_0} =$ **id**
$input_{E_1} =$ **id**
$L = [E_0, E_1]$

Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \text{-} \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

Parse tree is:



Tokens: **id+id*id**

$L = [n].L' = [E, E]$
$input =$ **id**
$b =$ **id**
$L = [E]$

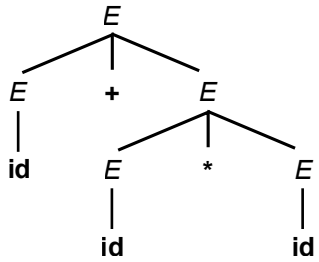
Let the grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= \text{-} \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \mathbf{id}
\end{aligned}
$$

Tokens: **id+id*id**

$L = [n].L' = [E]$
$input =$ **id**
$b =$ **id**
$L = []$

Parse tree is:



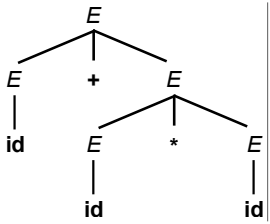
A grammar that produces more than one parse tree for some sentence is said to be ambiguous

- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence

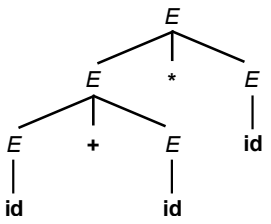
Example

Leftmost derivations for the arithmetic expression **id+id*id**

$$
\begin{aligned}
\langle E\rangle &\Rightarrow \langle E\rangle + \langle E\rangle \\
&\Rightarrow \mathbf{id} + \langle E\rangle \\
&\Rightarrow \mathbf{id} + \langle E\rangle * \langle E\rangle \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \langle E\rangle \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



$$
\begin{aligned}
\langle E\rangle &\Rightarrow \langle E\rangle * \langle E\rangle \\
&\Rightarrow \langle E\rangle + \langle E\rangle * \langle E\rangle \\
&\Rightarrow \mathbf{id} + \langle E\rangle * \langle E\rangle \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \langle E\rangle \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



[Cantor, 1962, Floyd, 1962]

- For parsers, it is desirable that the grammar be made unambiguous. Otherwise we cannot determine which parse tree to select for a sentence

- Another way is to use carefully chosen ambiguous grammars, together with disambiguating rules that discard undesirable parse trees, leaving only one tree for each sentence

Even if compiler designers rarely do this task, it is useful to be able to verify if a language can be generated from a grammar

Proof that a grammar G generates a language L has two parts:

1. Show up that every string generated by G is in L
2. Show up that every string in L can be generated by G

Example

Considerer the following grammar:

$$\langle \mathrm{S} \rangle \rightarrow \mathbf{(}\langle \mathrm{S} \rangle\mathbf{)}\langle \mathrm{S} \rangle \mid \epsilon$$

It may not be apparent, but this grammar generates all the strings of balanced parentheses, and only such strings. That why, we need to proceed the two steps of the proof

Basis : The basis is $n = 1$. The only string of terminals derivable from $\langle \mathrm{S} \rangle$ in one step is the empty string, which is balanced

Assumption: Assume that all derivations of fewer than n steps produce balanced sentences, and consider a leftmost derivation of exactly n steps

Induction : Such derivations must be of the form:

$$\langle \mathrm{S} \rangle \underset{lm}{\Rightarrow} \mathbf{(}\langle \mathrm{S} \rangle\mathbf{)}\langle \mathrm{S} \rangle \underset{lm}{\Rightarrow} \mathbf{(}\alpha\mathbf{)}\langle \mathrm{S} \rangle \underset{lm}{\Rightarrow} \mathbf{(}\alpha\mathbf{)}\beta$$

Derivations of α and β from $\langle \mathrm{S} \rangle$ take fewer than n steps, so by the inductive hypothesis α and β are balanced
Therefore, the string "$\mathbf{(}\alpha\beta$" must be balanced

Basis : If the string has length 0, it must be ϵ, which is balanced

Induction:

- Observe that every balanced string has even length
- Assume that every balanced string of length less than $2n$ is derivable from $\langle \mathrm{S} \rangle$, and consider a balanced string w of length $2n$, $n \geq 1$. Surely w begins with a left parenthesis
- Let (α) be the shortest nonempty prefix of w having an equal number of left and right parentheses
- Then w can be written $w = (\alpha)\beta$ where both α and β are balanced. Since α and β are of length less than $2n$, they are derivable from $\langle \mathrm{S} \rangle$ by the inductive hypothesis. Thus, we can find a derivation of the form:

$$\langle \mathrm{S} \rangle \Rightarrow (\langle \mathrm{S} \rangle)\langle \mathrm{S} \rangle \overset{*}{\Rightarrow} (\alpha)\langle \mathrm{S} \rangle \overset{*}{\Rightarrow} (\alpha)\beta$$

- Proving that $w = (\alpha)\beta$ is also derivable from $\langle \mathrm{S} \rangle$

Grammars are more powerful notation than regular expressions
Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa

Example (Grammar, and Regex)

Regex **(a|b)*abb** and the following grammar describe the same language:

$$
\begin{aligned}
\langle \mathrm{A} \rangle &::= \mathbf{a}\langle \mathrm{A} \rangle \\
&::= \mathbf{b}\langle \mathrm{A} \rangle \\
&::= \mathbf{a}\langle \mathrm{B} \rangle \\
\langle \mathrm{B} \rangle &::= \mathbf{b}\langle \mathrm{C} \rangle \\
\langle \mathrm{C} \rangle &::= \mathbf{b}\langle \mathrm{D} \rangle \\
\langle \mathrm{D} \rangle &::= \epsilon
\end{aligned}
$$

Example (Grammar, no Regex)

Language $L = a^n b^n | n \geq 1$ can be described by a grammar but not by a regular expression (except with Posix extension)

```
begin
    foreach state i of the NFA do
        A_i ← createNonterminal (i) ;
        foreach transition t from i to j do
            if t on token a then
                P ← P ∪ (⟨A_i⟩ → a⟨A_j⟩) ;
            end
            if t on ϵ then
                P ← P ∪ (⟨A_i⟩ → ⟨A_j⟩) ;
            end
        end
        if i is accepting state then
            P ← P ∪ (⟨A_i⟩ → ϵ) ;
        end
        if i is starting state then
            makeFirstProduction(A_i) ;
        end
    end
end
```

Why use regular expressions to define the lexical syntax of a language?

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a better modularity
2. Lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as complex as the grammars
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars

Regular expressions are useful to describe constructs such as identifiers, numbers...

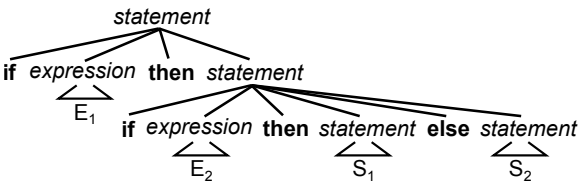
Grammars are most useful for describing nested structures such as balanced parentheses, corresponding if-then-else...

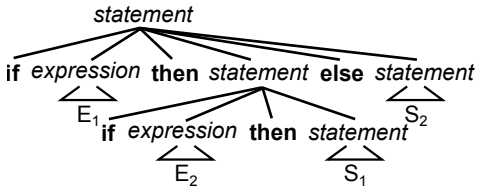
An ambiguous grammar can be rewritten to eliminate the ambiguity

- Grammar:

$$\begin{aligned} \langle \text{statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \\ &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \textbf{else} \langle \text{statement} \rangle \\ &::= \textbf{other} \end{aligned}$$

- Input: **if** E_1 **then if** E_2 **then** S_1 **else** S_2





- The first tree is preferred according to "Match each **else** with the closest unmatched **then**." This rule is rarely built into productions

The disambiguation of this "if-then-else" problem may be included into a new grammar

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \\
&::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \textbf{else} \langle \text{statement} \rangle \\
&::= \textbf{id}
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \langle \text{matched\_statement} \rangle \\
&::= \langle \text{open\_statement} \rangle \\
\langle \text{matched\_statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{matched\_statement} \rangle \\
&\quad\ \textbf{else} \langle \text{matched\_statement} \rangle \\
&::= \textbf{id} \\
\langle \text{open\_statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \\
&::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{matched\_statement} \rangle \textbf{else} \langle \text{open\_statement} \rangle
\end{aligned}
$$

The disambiguation of this "if-then-else" problem may be included into a new grammar

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \\
&::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \textbf{else} \langle \text{statement} \rangle \\
&::= \textbf{id}
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement} \rangle \langle \text{else\_statement} \rangle \\
&::= \textbf{id} \\
\langle \text{else\_statement} \rangle &::= \textbf{else} \langle \text{statement} \rangle \\
&::= \epsilon
\end{aligned}
$$

Left-Recursive Grammar

Grammar is left recursive if it has a nonterminal $\langle \mathrm{A} \rangle$ such that for some string α there is a derivation

$$\langle \mathrm{A} \rangle \overset{+}{\Rightarrow} \langle \mathrm{A} \rangle \alpha$$

Top-down parsing methods cannot handle left-recursive grammars

A transformation is needed to eliminate left recursion

$$
\begin{array}{rcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
& ::= & \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
& ::= & - \langle \mathrm{E} \rangle \\
& ::= & (\langle \mathrm{E} \rangle) \\
& ::= & \textbf{id}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathrm{E} \rangle & ::= & - \langle \mathrm{E} \rangle \langle R_E \rangle \\
& ::= & (\langle \mathrm{E} \rangle) \langle R_E \rangle \\
& ::= & \textbf{id} \langle R_E \rangle \\
\langle R_E \rangle & ::= & + \langle \mathrm{E} \rangle \langle R_E \rangle \\
& ::= & * \langle \mathrm{E} \rangle \langle R_E \rangle \\
& ::= & \epsilon
\end{array}
$$

Input : Grammar $G = \langle N, \Sigma, P, S \rangle$
Output : An equivalent grammar without left recursion

begin
  while $\exists A | (\langle A \rangle \rightarrow \langle A \rangle\ \gamma) \in P$ **do**
    foreach $p = (\langle A \rangle \rightarrow b\ \delta) \in P \wedge b \neq A$ **do**
      $P \leftarrow P \setminus \{p\}$;
      $P \leftarrow P \cup \{(\langle R_A \rangle \rightarrow b\ \delta\ \langle R_A \rangle)\}$;
    end
    $P = P \cup \{(\langle R_A \rangle \rightarrow \epsilon)\}$;
    foreach $p = (\langle A \rangle \rightarrow \langle A \rangle\ \omega) \in P$ **do**
      $P \leftarrow P \setminus \{p\}$;
      $P \leftarrow P \cup \{(\langle \mathrm{A} \rangle \rightarrow \omega\ \langle R_A \rangle)\}$;
    end
  end
end

When the choice between two alternatives A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice

$$
\begin{aligned}
\langle \text{statement} \rangle \quad &::= \quad \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{expression} \rangle \textbf{ else } \langle \text{expression} \rangle \\
&::= \quad \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{expression} \rangle \\
&::= \quad \textbf{id}
\end{aligned}
$$

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{statement} \rangle \textbf{ else } \langle \text{statement} \rangle \\
&::= \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{statement} \rangle \\
&::= \textbf{id}
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{statement} \rangle &::= \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{statement} \rangle \ \langle \text{else\_statement} \rangle \\
&::= \textbf{id} \\
\langle \text{else\_statement} \rangle &::= \textbf{else } \langle \text{statement} \rangle \\
&::= \epsilon
\end{aligned}
$$

Input : Grammar $G = \langle N, \Sigma, P, S \rangle$
Output : An equivalent left-factored grammar

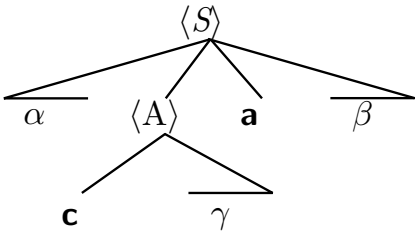
begin
  while $\exists A \in P | (\langle A \rangle \rightarrow \alpha\ \gamma), (\langle A \rangle \rightarrow \alpha\ \delta)$ **do**
    foreach $p = (\langle A \rangle \rightarrow \alpha\ \omega) \in P$ **do**
      $P \leftarrow P \setminus \{p\}$;
      if $\omega \neq \epsilon$ **then**
        $P \leftarrow P \cup \{(\langle R_A \rangle \rightarrow \omega)\}$;
      end
    end
    $P \leftarrow P \cup \{(\langle \mathrm{A} \rangle \rightarrow \alpha\ \langle R_A \rangle)\}$;
    $P \leftarrow P \cup \{(\langle R_A \rangle \rightarrow \epsilon)\}$;
  end
end

- The construction of both top-down and bottom-up parsers is aided by two functions associated with a grammar G:
  1. FIRST
  2. FOLLOW
- These functions allow us to choose which production to apply, based on the next input symbol
- During panic-mode error recovery the set of tokens replied by FOLLOW can be used as synchronizing tokens

- Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α
- If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α)

Example

$A \stackrel{*}{\Rightarrow} c\ \gamma$
FIRST(A) $= \{c\}$

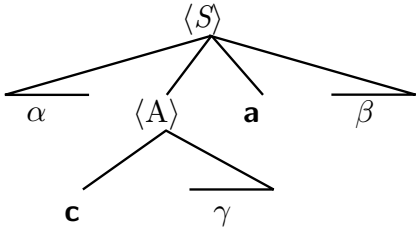


- To compute FIRST(X), apply the following rules until no more terminals or ϵ can be added to any FIRST set
  1. FIRST(X) $= \{X\}$ if X is a terminal
  2. If X is a nonterminal and $\langle \mathrm{X} \rangle \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then
     - Add a in FIRST(X) if $\exists i$ such that $a \in$ FIRST(Y_i)
     - Add ϵ in FIRST(X) if $Y_1\ Y_2 \ldots Y_k \stackrel{*}{\Rightarrow} \epsilon$
     - Add ϵ to FIRST(X) if $\forall j \in \{1, 2, \ldots, k\}$, $\epsilon \in$ FIRST(Y_j)
  3. Add ϵ to FIRST(X) if $\langle \mathrm{X} \rangle \rightarrow \epsilon$ is a production
- Add all non-ϵ symbols of FIRST(X_i) for $i \in \{1 \ldots n\}$ to FIRST($X_1\ X_2 \ldots X_n$)

- Define FOLLOW(A), where A is a nonterminal, to be the set of terminals a that can appear immediately to the right of A in some sentential form
- The set of terminals a such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha\ A\ a\ \beta$, for some α and β
- Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derive ϵ and disappeared
- If A can be the rightmost symbol, then **eof** (or usually **$**) is in FOLLOW($A$)

Example

$A \overset{*}{\Rightarrow} c\ \gamma$
$a \in \text{FOLLOW}(A)$



- To compute FOLLOW(A) for a nonterminal A, apply the following rules until nothing can be added to any FOLLOW set

1. Place **eof** in FOLLOW(S), where S is the start symbol, and **eof** is the input right end-marker
2. If there is a production $\langle \mathrm{A} \rangle \rightarrow \alpha \langle \mathrm{B} \rangle \beta$, then everything in FIRST(β), except ϵ is added in FOLLOW(B)
3. If there is a production $\langle \mathrm{A} \rangle \rightarrow \alpha \langle \mathrm{B} \rangle$, or a production $\langle \mathrm{A} \rangle \rightarrow \alpha \langle \mathrm{B} \rangle \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is added to FOLLOW(B)

Top-Down Parsing

Constructing a parse tree for the input string, starting from the root of the grammar, and creating the nodes of the parse tree in preorder

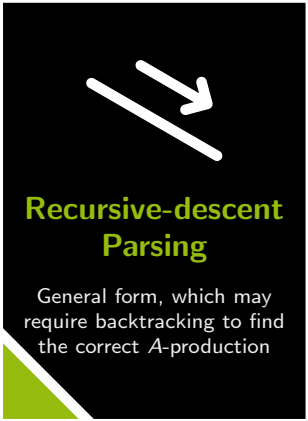
Top-down parsing can be viewed as finding a leftmost derivation for an input string

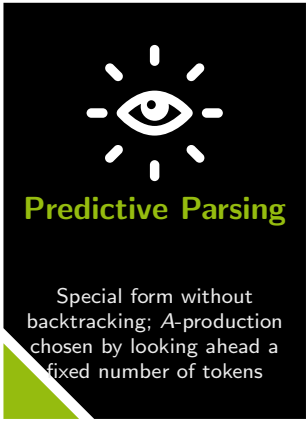
Illustrative Input String and Grammar

Input string: **id+id*id**

Grammar:

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{T} \rangle \, \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle &::= \mathbf{+} \, \langle \mathrm{T} \rangle \, \langle \mathrm{E'} \rangle \mid \epsilon \\
\langle \mathrm{T} \rangle &::= \langle \mathrm{F} \rangle \, \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle &::= \mathbf{*} \, \langle \mathrm{F} \rangle \, \langle \mathrm{T'} \rangle \mid \epsilon \\
\langle \mathrm{F} \rangle &::= \mathbf{(} \, \langle \mathrm{E} \rangle \, \mathbf{)} \mid \mathbf{id}
\end{aligned}
$$





Class of grammars dedicated to the predictive parsers looking k symbols ahead in the input is called $LL(k)$ class

Recursive-descent parsing program consists of a set of procedures, one for each nonterminal

Procedure *A*

Input : Production $\langle \mathrm{A} \rangle \rightarrow \alpha_1 \ldots \alpha_k$

begin

  for $i \leftarrow 1$ **to** k **do**

    if α_i *is nonterminal* **then**

      `call` $\alpha_i()$;

    else if $\alpha_i =$*current input symbol a* **then**

      forward $\leftarrow$ forward $+ 1$ `// Move input pointer;`

    else

      Report an error ;

    end

  end

end

[Hoare, 1962, Schorre, 1964, McClure, 1965]

Left-recursive grammar can cause a recursive-descent parser to go into an infinite loop

- **When:** expand a nonterminal A, the same nonterminal is found again and expanded without consuming input

$$
\begin{aligned}
\langle \mathrm{E} \rangle &::= \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle \\
&::= \langle \mathrm{E} \rangle * \langle \mathrm{E} \rangle \\
&::= - \langle \mathrm{E} \rangle \\
&::= (\langle \mathrm{E} \rangle) \\
&::= \textbf{id}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{E} &\Rightarrow \mathrm{E} + \mathrm{E} \\
&\Rightarrow \mathrm{E} + \mathrm{E} + \mathrm{E} \\
&\Rightarrow \mathrm{E} + \mathrm{E} + \mathrm{E} + \mathrm{E} \\
&\Rightarrow \mathrm{E} + \mathrm{E} + \mathrm{E} + \mathrm{E} + \mathrm{E} \\
&\Rightarrow \ldots
\end{aligned}
$$

LL(1) Grammar

Class of grammars that have the following properties:

- **L**eft-to-right input scanning
- **L**eftmost derivation
- **1** input symbol is used for lookahead to make parsing action decisions

- Predictive parser can be constructed for *LL*(1) grammars, because no backtracking is needed
- *LL*(1) grammars are rich enough to cover most programming constructs
- But, they must be neither left-recursive nor ambiguous

[Lewis and Stearns, 1968, Birman and Ullman, 1973]

LL(1) Grammar (refined)

Grammar G is *LL*(1) iff whenever $\langle \mathrm{A} \rangle \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

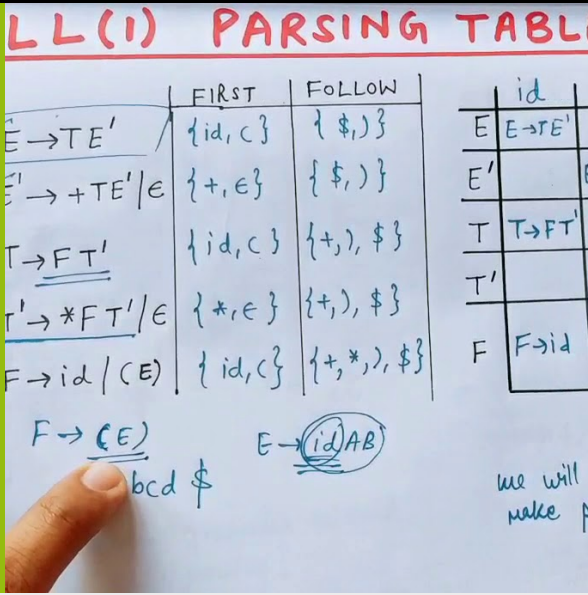
1. For nonterminal **a**, both α and β derive strings beginning with **a**
2. At most one of α and β can derive the empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)
4. Likewise, if $\alpha \overset{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A)

$$
\begin{aligned}
\langle \text{statement\_list} \rangle &::= \langle \text{statement} \rangle \ \langle \text{statement\_list} \rangle \\
&::= \epsilon \\
\langle \text{statement} \rangle &::= \textbf{if(} \langle \text{expression} \rangle \textbf{)} \langle \text{statement} \rangle \textbf{else} \langle \text{statement} \rangle \\
&::= \textbf{while(} \langle \text{expression} \rangle \textbf{)} \langle \text{statement} \rangle \\
&::= \{ \langle \text{statement\_list} \rangle \}
\end{aligned}
$$

To parse an input string, a **table** should be build

It determines the production to use from a given production and the input symbol

Table $M[A, a]$ is built from FIRST and FOLLOW sets, where A is a nonterminal, and a is a terminal or **eof**



Input : Grammar $G = \langle N, \Sigma, P, S \rangle$
Output : Parsing table M

begin
  foreach $(\langle A \rangle \to \alpha) \in P$ **do**
    foreach *terminal* $a \in$ *FIRST*(α) **do**
      $M[A, a] \leftarrow M[A, a] \cup (\langle \mathrm{A} \rangle \to \alpha)$
    end
    if $\epsilon \in$ *FIRST*(α) **then**
      foreach $b \in$ *FOLLOW*(A) **do**
        $M[A, b] \leftarrow M[A, b] \cup (\langle \mathrm{A} \rangle \to \alpha)$
      end
      if **eof** $\in$ *FOLLOW*(A) **then**
        $M[A, \textbf{eof}] \leftarrow M[A, \textbf{eof}] \cup (\langle \mathrm{A} \rangle \to \alpha)$
      end
    end
  end
  if $\forall \alpha,\ M[A, \alpha] = \emptyset$ **then**
    $M[A, \alpha] \leftarrow$ error
  end
end

$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & \mathbf{+} \ \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & \mathbf{*} \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & \mathbf{(} \ \langle \mathrm{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

	id	+	*	(	)	eof
E						
E'						
T						
T'						
F						

$\langle \text{E} \rangle$ ::= $\langle \text{T} \rangle \langle \text{E'} \rangle$
$\langle \text{E'} \rangle$ ::= **+** $\langle \text{T} \rangle \langle \text{E'} \rangle$
::= ϵ
$\langle \text{T} \rangle$ ::= $\langle \text{F} \rangle \langle \text{T'} \rangle$
$\langle \text{T'} \rangle$ ::= ***** $\langle \text{F} \rangle \langle \text{T'} \rangle$
::= ϵ
$\langle \text{F} \rangle$ ::= **(** $\langle \text{E} \rangle$ **)**
::= **id**

For production: (1) $\langle \text{E} \rangle \rightarrow \langle \text{T} \rangle \langle \text{E'} \rangle$
$\text{FIRST}(TE') = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$

	id	+	*	(	)	eof
E						
E'						
T						
T'						
F						

$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & \mathbf{+} \ \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & \mathbf{*} \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & \mathbf{(} \ \langle \mathrm{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

For production: (1) $\langle \mathrm{E} \rangle \rightarrow \langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle$
$\mathrm{FIRST}(TE') = \mathrm{FIRST}(T) = \mathrm{FIRST}(F) = \{\mathbf{(}, \mathbf{id}\}$
Then put the production in $M[E, \mathbf{(}]$ and $M[E, \mathbf{id}]$; the rest of the line is error

	id	+	*	(	)	eof
E	(1)			(1)		
E'						
T						
T'						
F						

$\langle \text{E} \rangle \quad ::= \quad \langle \text{T} \rangle \, \langle \text{E'} \rangle$
$\langle \text{E'} \rangle \quad ::= \quad + \, \langle \text{T} \rangle \, \langle \text{E'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{T} \rangle \quad ::= \quad \langle \text{F} \rangle \, \langle \text{T'} \rangle$
$\langle \text{T'} \rangle \quad ::= \quad * \, \langle \text{F} \rangle \, \langle \text{T'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{F} \rangle \quad ::= \quad (\, \langle \text{E} \rangle \,)$
$\quad ::= \quad \textbf{id}$

For production: (2) $\langle \text{E'} \rangle \rightarrow + \langle \text{T} \rangle \langle \text{E'} \rangle$
FIRST$(+TE') = \{+\}$
Then put the production in $M[E', +]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)				
T						
T'						
F						

$\langle \text{E} \rangle \quad ::= \quad \langle \text{T} \rangle \ \langle \text{E'} \rangle$
$\langle \text{E'} \rangle \quad ::= \quad \mathbf{+} \ \langle \text{T} \rangle \ \langle \text{E'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{T} \rangle \quad ::= \quad \langle \text{F} \rangle \ \langle \text{T'} \rangle$
$\langle \text{T'} \rangle \quad ::= \quad \mathbf{*} \ \langle \text{F} \rangle \ \langle \text{T'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{F} \rangle \quad ::= \quad \mathbf{(} \ \langle \text{E} \rangle \ \mathbf{)}$
$\quad ::= \quad \mathbf{id}$

For production: (3) $\langle \text{E'} \rangle \rightarrow \epsilon$
FIRST(ϵ) = $\{\epsilon\}$
FOLLOW(E') = $\{\mathbf{)}, \mathbf{eof}\}$
Put the production in $M[E', \mathbf{)}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	
T						
T'						
F						

$\langle \text{E} \rangle \quad ::= \quad \langle \text{T} \rangle \ \langle \text{E'} \rangle$
$\langle \text{E'} \rangle \quad ::= \quad \textbf{+} \ \langle \text{T} \rangle \ \langle \text{E'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{T} \rangle \quad ::= \quad \langle \text{F} \rangle \ \langle \text{T'} \rangle$
$\langle \text{T'} \rangle \quad ::= \quad \textbf{*} \ \langle \text{F} \rangle \ \langle \text{T'} \rangle$
$\quad ::= \quad \epsilon$
$\langle \text{F} \rangle \quad ::= \quad \textbf{(} \ \langle \text{E} \rangle \ \textbf{)}$
$\quad ::= \quad \textbf{id}$

For production: (3) $\langle \text{E'} \rangle \rightarrow \epsilon$
FIRST(ϵ) = $\{\epsilon\}$
FOLLOW(E') = $\{\textbf{)}, \textbf{eof}\}$
Put the production in $M[E', \textbf{)}]$
Put the production in $M[E', \textbf{eof}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T						
T'						
F						

$\langle \text{E} \rangle ::= \langle \text{T} \rangle \langle \text{E'} \rangle$
$\langle \text{E'} \rangle ::= + \langle \text{T} \rangle \langle \text{E'} \rangle$
$::= \epsilon$
$\langle \text{T} \rangle ::= \langle \text{F} \rangle \langle \text{T'} \rangle$
$\langle \text{T'} \rangle ::= * \langle \text{F} \rangle \langle \text{T'} \rangle$
$::= \epsilon$
$\langle \text{F} \rangle ::= (\langle \text{E} \rangle)$
$::= \textbf{id}$

For production: (4) $\langle \text{T} \rangle \rightarrow \langle \text{F} \rangle \langle \text{T'} \rangle$
$\text{FIRST}(F\ T') = \text{FIRST}(F) = \{(, \textbf{id}\}$
Put the production in $M[T, (]$ and $M[T, \textbf{id}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'						
F						

$\langle \text{E} \rangle ::= \langle \text{T} \rangle \langle \text{E'} \rangle$
$\langle \text{E'} \rangle ::= + \langle \text{T} \rangle \langle \text{E'} \rangle$
$::= \epsilon$
$\langle \text{T} \rangle ::= \langle \text{F} \rangle \langle \text{T'} \rangle$
$\langle \text{T'} \rangle ::= * \langle \text{F} \rangle \langle \text{T'} \rangle$
$::= \epsilon$
$\langle \text{F} \rangle ::= (\langle \text{E} \rangle)$
$::= \textbf{id}$

For production: (5) $\langle \text{T'} \rangle \rightarrow *\langle \text{F} \rangle \langle \text{T'} \rangle$
FIRST(* F T') = {*}
Put the production in $M[T', *]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'			(5)			
F						

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::= \textbf{id}$

For production: (6) $\langle T' \rangle \rightarrow \epsilon$
FIRST(ϵ) = $\{\epsilon\}$
FOLLOW(T') = $\{+,), \textbf{eof}\}$
Put the production in $M[T', +]$ and $M[T,)]$
Put the production in $M[T', \textbf{eof}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F						

$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & \mathbf{+} \ \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & \mathbf{*} \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & \mathbf{(} \ \langle \mathrm{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

For production: (7) $\langle \mathrm{F} \rangle \rightarrow \mathbf{(}\langle \mathrm{E} \rangle \mathbf{)}$
FIRST($\mathbf{(}\ E\mathbf{)}$) = $\{\mathbf{(}\}$
Put the production in $M[F, \mathbf{(}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F				(7)		

$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & + \langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & * \langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & (\langle \mathrm{E} \rangle) \\
 & ::= & \mathbf{id}
\end{array}
$$

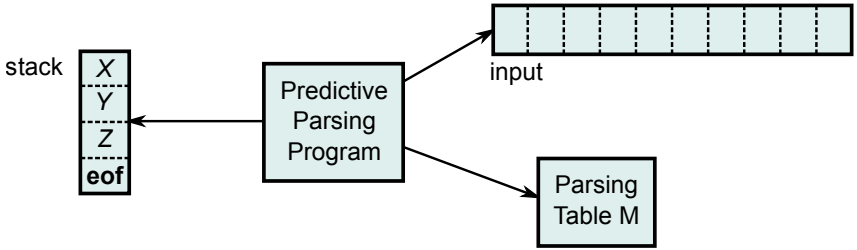
For production: (8) $\langle \mathrm{F} \rangle \rightarrow \mathbf{id}$
FIRST(**id**) = {**id**}
Put the production in $M[F, \mathbf{id}]$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

- Nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls
- Parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that:

$$S \overset{*}{\Rightarrow} w\ \alpha$$

- Table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table, and an output stream

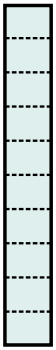


Input : A string w, a parsing table M for grammar G, and a start symbol s_0
Output : If w is in $L(G)$, a lef-most derivation of w; otherwise, an error indication

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 ... Yn) then
            print (⟨X⟩ → Y1 ... Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error                          // M[X, a] is empty;
        end
    end
end
```

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

stack:



input:

a

output:

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$

$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$

$::= \epsilon$

$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$

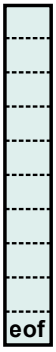
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$

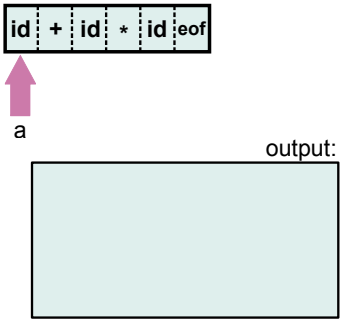
$::= \epsilon$

$\langle F \rangle ::= (\langle E \rangle)$

$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$

$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$

$::= \epsilon$

$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$

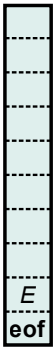
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$

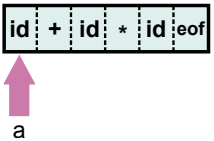
$::= \epsilon$

$\langle F \rangle ::= (\langle E \rangle)$

$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$

$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$

$::= \epsilon$

$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$

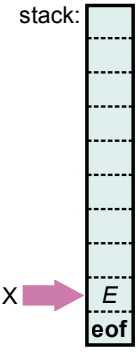
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$

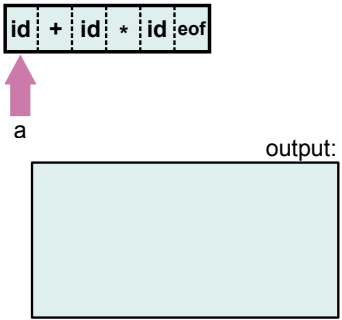
$::= \epsilon$

$\langle F \rangle ::= (\langle E \rangle)$

$::= \mathbf{id}$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

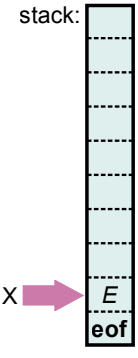


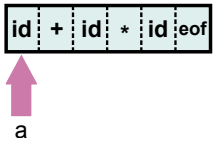


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

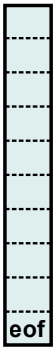


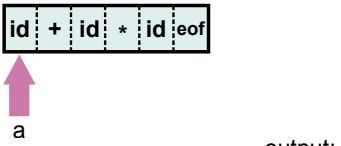


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	(⟨E⟩)
	::=	id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

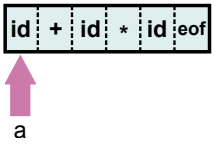




	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	(⟨E⟩)
	::=	id

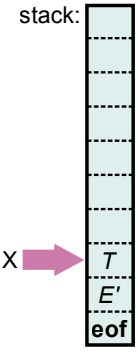
```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

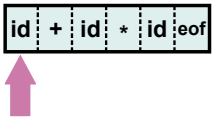


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	**(** ⟨E⟩ **)**
	::=	**id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





output:

$E \rightarrow T\ E'$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$

$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$

$::= \epsilon$

$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$

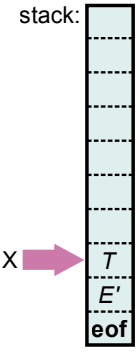
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$

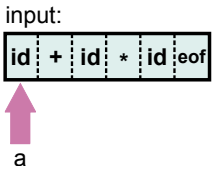
$::= \epsilon$

$\langle F \rangle ::= (\langle E \rangle)$

$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$

$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$

$::= \epsilon$

$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$

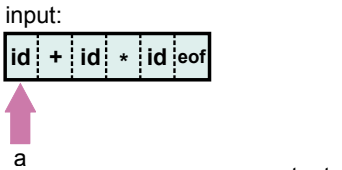
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$

$::= \epsilon$

$\langle F \rangle ::= (\langle E \rangle)$

$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

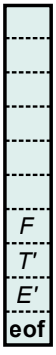


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	(⟨E⟩)
	::=	id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

stack:



input:

a

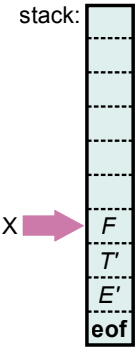
output:

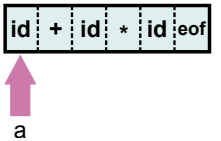
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





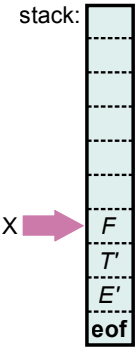
output:

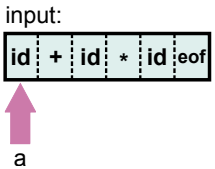
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```



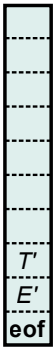


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

stack:



input:

a

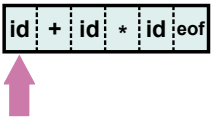
output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	**(** ⟨E⟩ **)**
	::=	**id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```



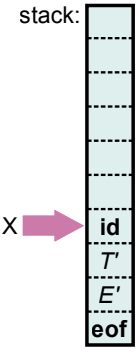
output:

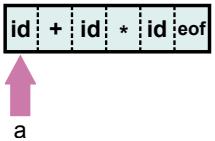
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\ \langle E \rangle\ )$
$::=$ **id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





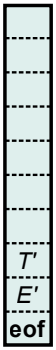
output:

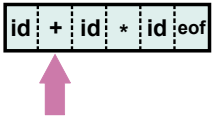
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\ \langle E \rangle\ )$
$::=$ **id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





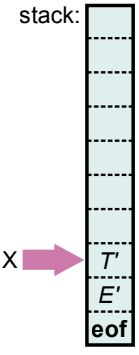
output:

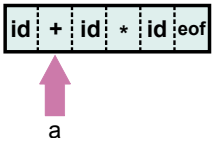
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::=$ **id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





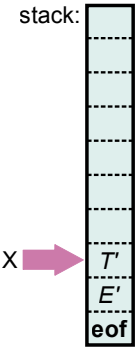
output:

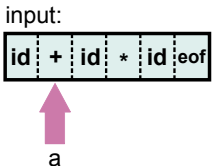
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::=$ **id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

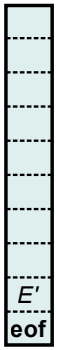


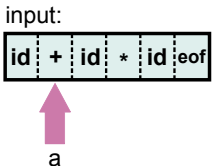


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩	::=	⟨T⟩ ⟨E'⟩
⟨E'⟩	::=	+ ⟨T⟩ ⟨E'⟩
	::=	ϵ
⟨T⟩	::=	⟨F⟩ ⟨T'⟩
⟨T'⟩	::=	* ⟨F⟩ ⟨T'⟩
	::=	ϵ
⟨F⟩	::=	**(** ⟨E⟩ **)**
	::=	**id**

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

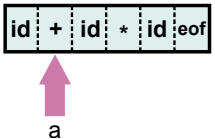




	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

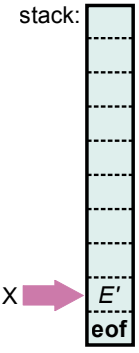
```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

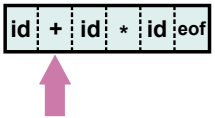


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





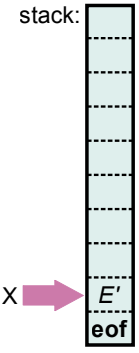
output:

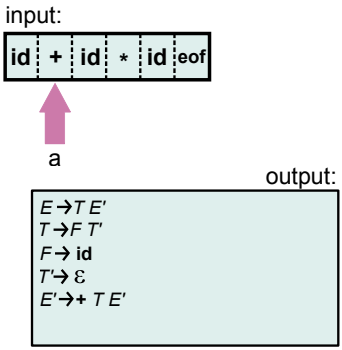
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**
$T' \rightarrow \varepsilon$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle ::= \langle T \rangle \langle E' \rangle$
$\langle E' \rangle ::= + \langle T \rangle \langle E' \rangle$
$::= \epsilon$
$\langle T \rangle ::= \langle F \rangle \langle T' \rangle$
$\langle T' \rangle ::= * \langle F \rangle \langle T' \rangle$
$::= \epsilon$
$\langle F \rangle ::= (\langle E \rangle)$
$::= id$

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```

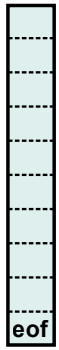


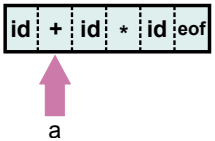


	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```





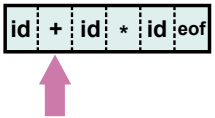
output:

E → *T E'*
T → *F T'*
F → **id**
T' → ε
E' → **+** *T E'*

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

⟨E⟩ ::= ⟨T⟩ ⟨E'⟩
⟨E'⟩ ::= + ⟨T⟩ ⟨E'⟩
::= ϵ
⟨T⟩ ::= ⟨F⟩ ⟨T'⟩
⟨T'⟩ ::= * ⟨F⟩ ⟨T'⟩
::= ϵ
⟨F⟩ ::= (⟨E⟩)
::= id

```
begin
    a ← nextInputSymbol;
    push (S, eof) ;
    push (S, s0) ;
    while topOf(S) ≠ eof do
        X ← topOf (S) ;
        if X = a then
            pop (S); a ← nextInputSymbol;
        else if X is a terminal then
            Report an error;
        else if M[X, a] = (⟨X⟩ → Y1 . . . Yn) then
            print (⟨X⟩ → Y1 . . . Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi);
        else
            Report an error;
        end
    end
end
```



output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow$ **id**
$T' \rightarrow \varepsilon$
$E' \rightarrow +\ T\ E'$

	id	+	*	(	)	eof
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

$\langle E \rangle$ ::= $\langle T \rangle\ \langle E' \rangle$
$\langle E' \rangle$ ::= + $\langle T \rangle\ \langle E' \rangle$
::= ϵ
$\langle T \rangle$ ::= $\langle F \rangle\ \langle T' \rangle$
$\langle T' \rangle$ ::= * $\langle F \rangle\ \langle T' \rangle$
::= ϵ
$\langle F \rangle$ ::= ($\langle E \rangle$)
::= **id**

- Panic-mode error recovery is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears

- Its effectiveness depends on the choice of synchronizing set
- The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice

- Some heuristics are explained in the following slides

As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue

If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input

It is not enough to use FOLLOW(A) as the synchronizing set for A. We can add to the set of a lower-level construct the symbols that begin higher-level constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions

If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens

If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery

Bottom-Up Parsing

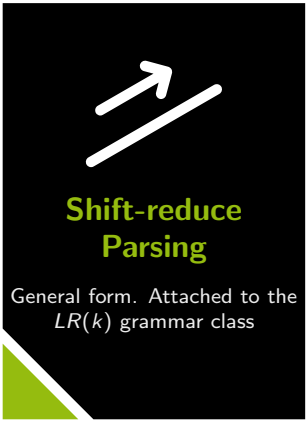
Constructing a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root

General Principle

Bottom-up parsing is the process of "reducing" a string w to the start symbol of the grammar

By definition, reduction is the reverse of derivation
The goal of the bottom-up parsing is therefore to construct a derivation in reverse

[Knuth, 1965, Korenjak, 1969]



- **L**eft-to-right input scanning
- **R**ightmost derivation
- **k** input symbol is used for lookahead to make parsing action decisions

$LR(k)$ parser is too difficult to be written by hand
We prefer to use automatic parser generators

[Knuth, 1965, Korenjak, 1969]

General Reduction Algorithm

At each reduction step:

- Select a specific substring matching the body of a production
- Replace the selected substring by the nonterminal at the head of the production

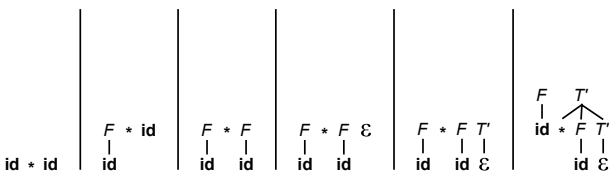
Key Decisions

- when to reduce
- what production to apply

- Let the grammar:

$$
\begin{array}{rcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & \mathbf{+} \ \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & \mathbf{*} \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & \mathbf{(} \ \langle \mathrm{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

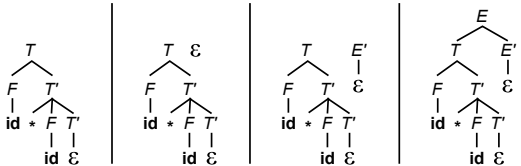
- A possible sequence of reductions is:
  $\mathbf{id} * \mathbf{id} \Leftarrow F * \mathbf{id} \Leftarrow F * F \Leftarrow F * F \, \epsilon \Leftarrow F * F \ T' \Leftarrow F \ T'$



- Let the grammar:

$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & \mathbf{+} \ \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & \mathbf{*} \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & \mathbf{(} \ \langle \mathrm{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

- A possible sequence of reductions is:
  $\mathbf{id} * \mathbf{id} \Leftarrow F * \mathbf{id} \Leftarrow F * F \Leftarrow F * F \ \epsilon \Leftarrow F * F \ T' \Leftarrow F \ T' \Leftarrow T \Leftarrow T \ \epsilon \Leftarrow T \ E' \Leftarrow E$



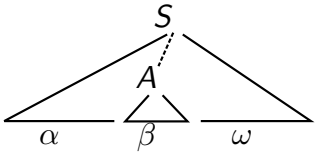
What is the best sequence of reductions to build the parse tree?

- One method is to use the shift-reduce parsing method
- Shift-reduce method is based on the handle pruning

Handle

Substring matching the body of a production, and whose reduction represents one step along the reverse of a right-most derivation

If $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha\, A\, \omega \underset{rm}{\Rightarrow} \alpha\, \beta\, \omega$ then production $\langle \mathrm{A} \rangle \to \beta$ in the position following α is a handle of $\alpha\beta\omega$



- A handle of a right-sentential form γ is a production $\langle \mathrm{A} \rangle \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ
- Note that ω must contain only terminal symbols

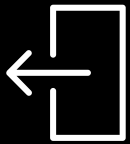
Inputs : A string of terminals ω. A grammar $G = \langle N, \Sigma, P, S \rangle$
Output : A sequence of reductions of ω, or an error if no sequence was found
Assumption: $w = \gamma_n$, where γ_n is the nth right-sentential form of some, **yet unknown**, rightmost derivation:

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \overset{*}{\underset{rm}{\Rightarrow}} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

```
begin
    d←[]; f←ω;
    while f ≠ S do
        if ∃h ∈ f|f = α h β ; β contains only terminals then
            if ∃p ∈ G|p = (⟨A⟩ → h) then
                f←α p β;
                d←[(α h β)].d;
            else
                throw("Cannot find a production for reduction")
            end
        else
            throw("Cannot find a handle")
        end
    end
    return d;
end
```

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed

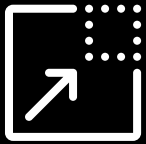
- The handle always appears at the top of the stack just before it is identified as the handle



Shift

Shift the next input symbol onto the top of the stack

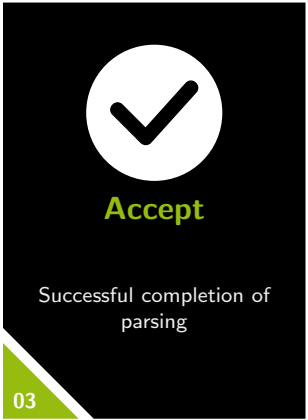
01



Reduce

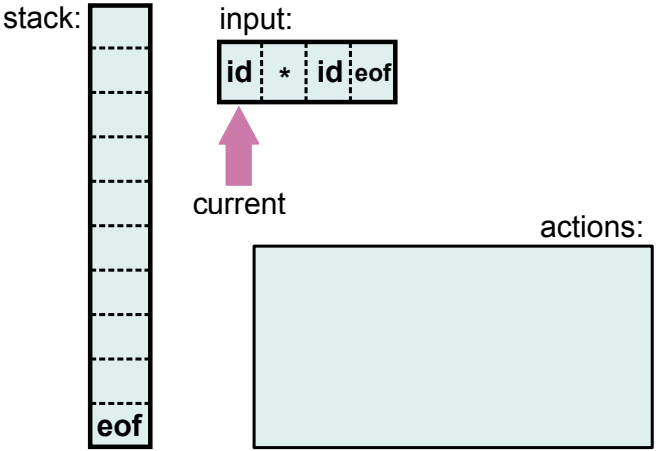
Get string from the stack, and detect the production to replace it

02

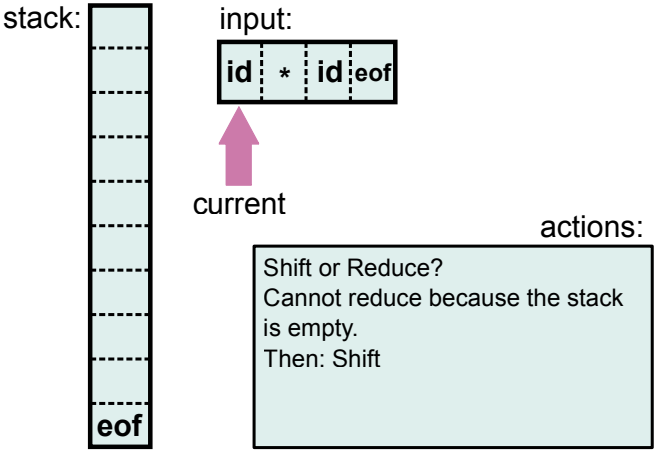

Accept
Successful completion of parsing
03


Error
Syntax error and call an error recovery routine
04

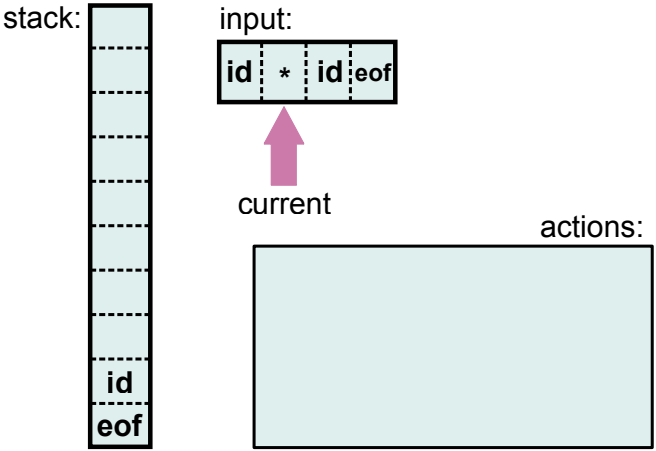
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



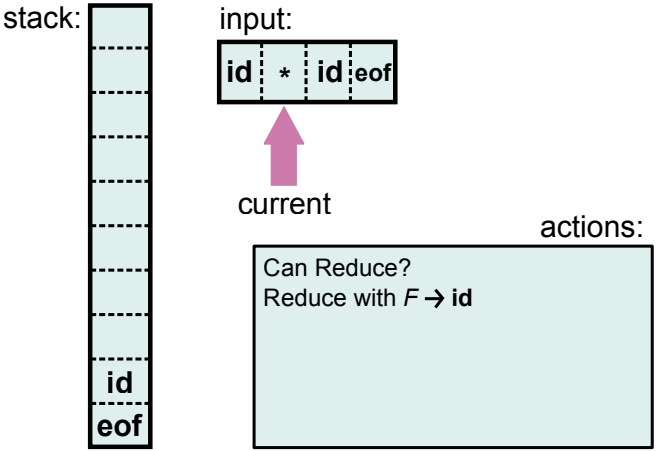
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



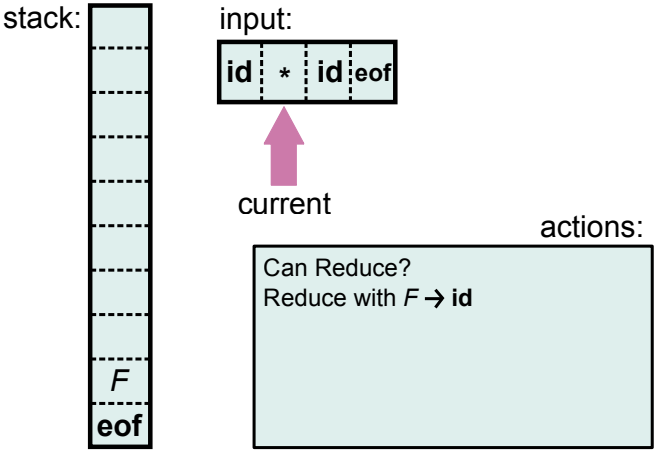
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



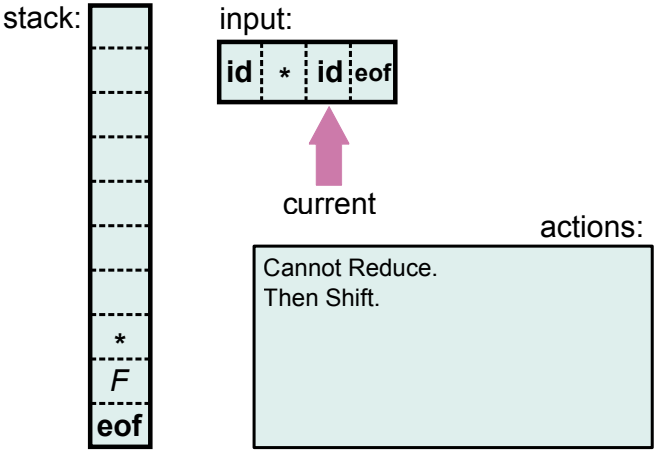
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



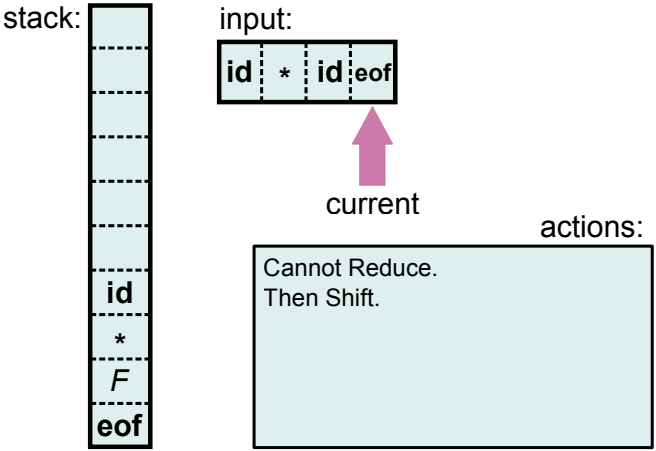
$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & + \ \langle \mathrm{T} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & * \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & (\ \langle \mathrm{E} \rangle \ ) \\
 & ::= & \mathbf{id}
\end{array}
$$



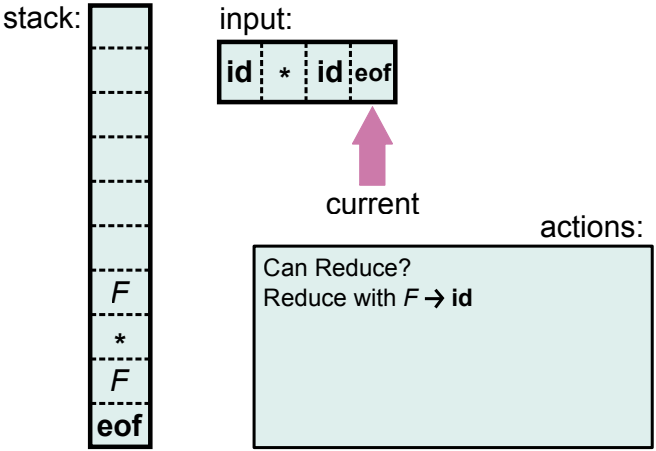
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



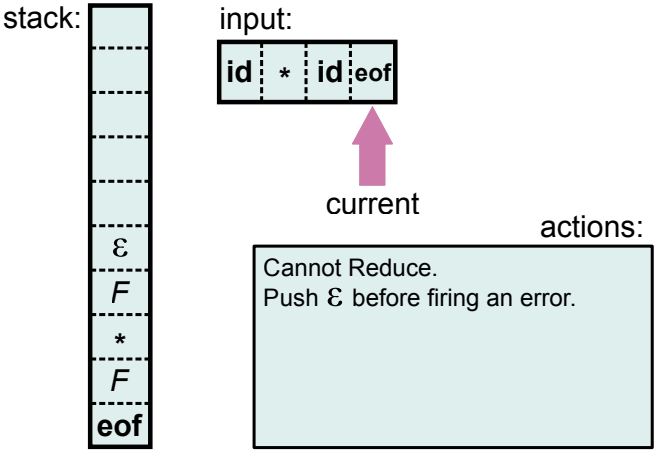
$$
\begin{array}{rcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



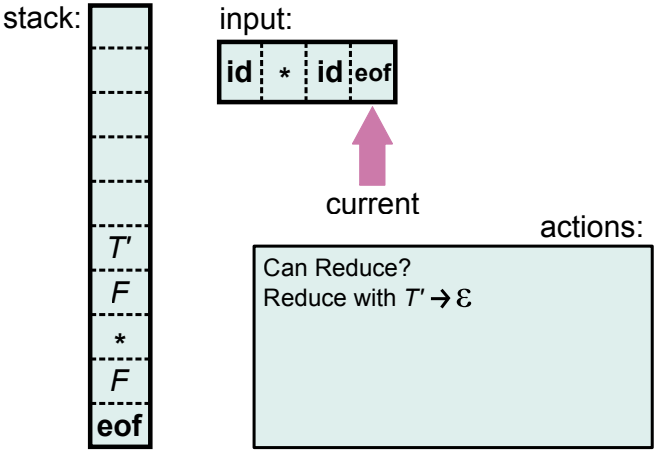
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



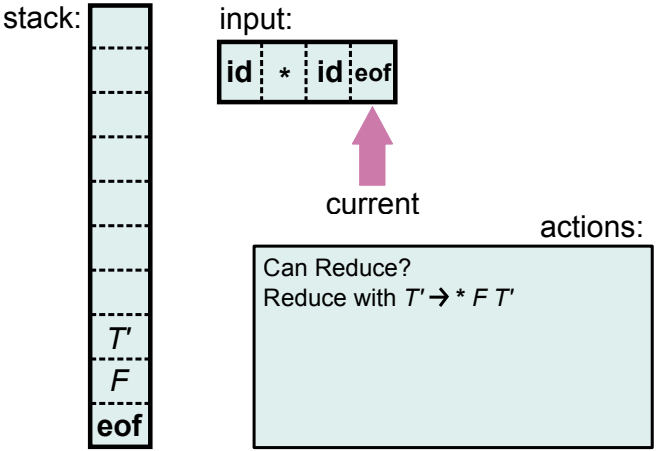
$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & + \ \langle \mathrm{T} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & * \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & (\ \langle \mathrm{E} \rangle\ ) \\
 & ::= & \mathbf{id}
\end{array}
$$



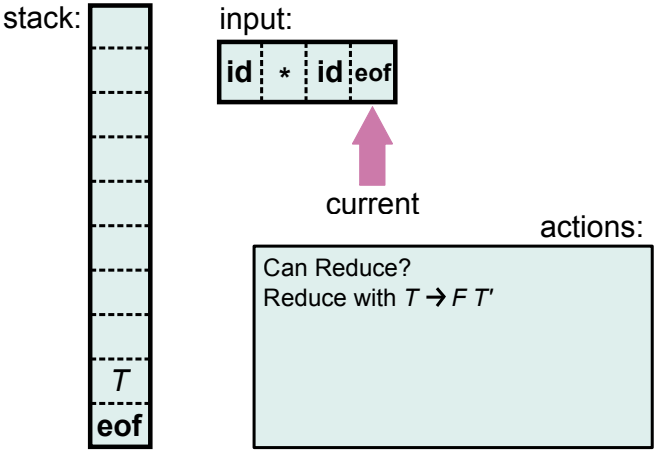
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



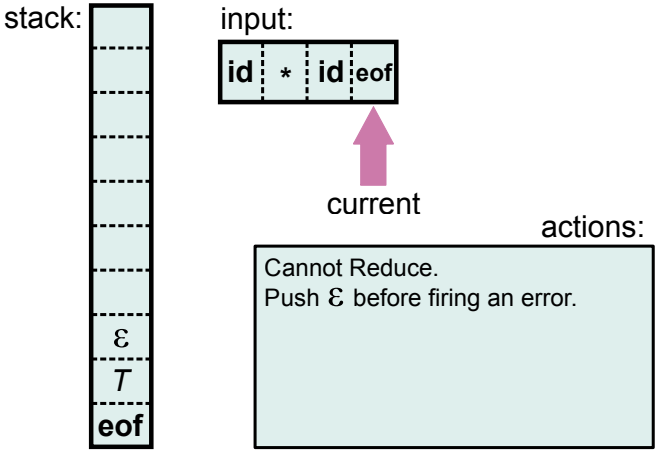
$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & + \ \langle \mathrm{T} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & * \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & (\ \langle \mathrm{E} \rangle\ ) \\
 & ::= & \mathbf{id}
\end{array}
$$



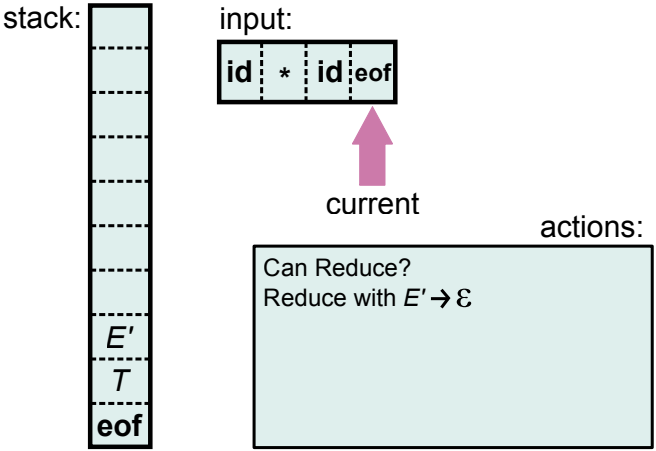
$$
\begin{array}{lcl}
\langle \mathrm{E} \rangle & ::= & \langle \mathrm{T} \rangle \ \langle \mathrm{E'} \rangle \\
\langle \mathrm{E'} \rangle & ::= & + \ \langle \mathrm{T} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{T} \rangle & ::= & \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
\langle \mathrm{T'} \rangle & ::= & * \ \langle \mathrm{F} \rangle \ \langle \mathrm{T'} \rangle \\
 & ::= & \epsilon \\
\langle \mathrm{F} \rangle & ::= & (\ \langle \mathrm{E} \rangle \ ) \\
 & ::= & \mathbf{id}
\end{array}
$$



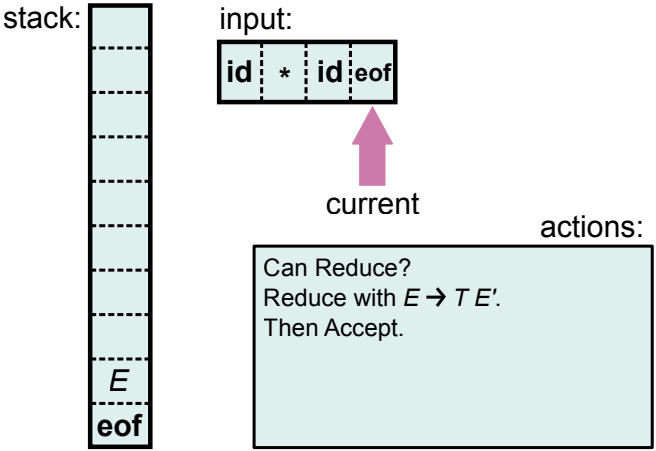
$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



$$
\begin{array}{lcl}
\langle \text{E} \rangle & ::= & \langle \text{T} \rangle \ \langle \text{E'} \rangle \\
\langle \text{E'} \rangle & ::= & + \ \langle \text{T} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{T} \rangle & ::= & \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
\langle \text{T'} \rangle & ::= & * \ \langle \text{F} \rangle \ \langle \text{T'} \rangle \\
 & ::= & \epsilon \\
\langle \text{F} \rangle & ::= & (\ \langle \text{E} \rangle \ ) \\
 & ::= & \textbf{id}
\end{array}
$$



There are context-free grammars for which shift-reduce parsing cannot be used

Every shift-reduce parser can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols

Shift/reduce Conflict
Cannot decide whether to shift or to reduce

Reduce/reduce Conflict
Cannot decide which of several reductions to make

Grammars causing these conflicts are not in the $LR(k)$ class

- Consider the grammar:

$$\begin{aligned} \langle \text{statement} \rangle &::= \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{statement} \rangle \\ &::= \textbf{if } \langle \text{expression} \rangle \textbf{ then } \langle \text{statement} \rangle \textbf{ else } \langle \text{statement} \rangle \\ &::= \textbf{id} \end{aligned}$$

- Consider the stack : **eof** . . . **if**⟨expression⟩**then**⟨statement⟩
- Consider the input : **else** . . . **eof**
- We cannot tell whether **if**⟨expression⟩**then**⟨statement⟩ is the handle, no matter what appears below it on the stack. There is a shift/reduce conflict
- Depending on what follows the **else** on the input, it might be correct to reduce if-then to ⟨statement⟩, or it might be correct to shift **else** and then to look for another ⟨statement⟩ to complete the if-then-else

- Consider the grammar with array indexes between parenthesis:

$$
\begin{array}{rcl}
\langle \text{statement} \rangle & ::= & \textbf{id(}\langle \text{parameters} \rangle\textbf{)} \\
& ::= & \textbf{id:=}\langle \text{expression} \rangle \\
\langle \text{parameters} \rangle & ::= & \langle \text{parameters} \rangle\textbf{,id} \\
& ::= & \textbf{id} \\
\langle \text{expression} \rangle & ::= & \textbf{id(}\langle \text{expressions} \rangle\textbf{)} \\
& ::= & \textbf{number} \\
\langle \text{expressions} \rangle & ::= & \langle \text{expressions} \rangle\textbf{,}\langle \text{expression} \rangle \\
& ::= & \langle \text{expression} \rangle
\end{array}
$$

- Consider the stack : **eof** . . . **id(id)**
- Consider the input : **,id)** . . . **eof**
- It is evident that the **id** on top of the stack should be reduced, but by which production?
  1. $\langle \text{parameters} \rangle \rightarrow$ **id** if p is a procedure
  2. $\langle \text{expressions} \rangle \rightarrow$ **id** if p is an array

This section introduces a simple $LR(k)$ (or $SLR(k)$) parsing based on the concepts previously presented

$LR(k)$ Grammar

Class of grammars that have the following properties:

- **L**eft-to-right input scanning
- **R**ightmost derivation
- **k** input symbol is used for lookahead to make parsing action decisions

$LR(k)$ parser is table-driven, as the nonrecursive $LL(k)$ parser

In this chapter, only cases $k = 0$ and $k = 1$ are considered

[DeRemer, 1971]

1. Can be constructed to recognize all programming language constructs
   Non-LR context-free grammars exist, but not used for typical programming-language constructs
2. LR-parsing method is the most general nonbacktracking shift-reduce parsing method
3. LR parser can detect a syntactic error early
4. For a grammar to be $LR(k)$, we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead
   This requirement is far less stringent than that for $LL(k)$ grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives
   Thus, it should not be surprising that LR grammars can describe more languages than LL grammars

- **$LR(0)$ automaton helps with shift-reduce decisions**
- Suppose that the string γ of symbols takes the $LR(0)$ automaton from the start state I_0 to some state I_j
- Then, shift on the next input symbol **a** if state I_j has a transition on **a**
- Otherwise, we choose to reduce
  Items in state I_j indicate which production to use

- $LR(0)$ parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse
- States represent sets of "items"

Item

$LR(0)$ item (or item) of a grammar G is a production of G with a dot (noted $\bullet$) at some position of the body

- Production $\langle \mathrm{A} \rangle \rightarrow XYZ$ generates the four items:

$$\begin{aligned} \langle \mathrm{A} \rangle &::= \bullet\, X\ Y\ Z \\ &::= X \bullet\ Y\ Z \\ &::= X\ Y \bullet\ Z \\ &::= X\ Y\ Z \bullet \end{aligned}$$

- Production $\langle \mathrm{A} \rangle \rightarrow \epsilon$ generates only one item: $\langle \mathrm{A} \rangle ::= \bullet$

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process

Examples

1. $\langle \mathrm{A} \rangle ::= \bullet\ X\ Y\ Z$
   we hope to see a string derivable from $X\ Y\ Z$ next on the input
2. $\langle \mathrm{A} \rangle ::= X\ \bullet\ Y\ Z$
   we have just seen on the input a string derivable from X and that we hope next to see a string derivable from $Y\ Z$
3. $\langle \mathrm{A} \rangle ::= X\ Y\ Z\ \bullet$
   we have seen the body $X\ Y\ Z$ and that it may be time to reduce $X\ Y\ Z$ to A

Kernel Item

Initial item, $\langle \mathrm{S'} \rangle \rightarrow \langle \mathrm{S} \rangle$, and all items whose dots are not at the left end

Nonkernel Item

All items with their dots at the left end, except for $\langle \mathrm{S'} \rangle \rightarrow \langle \mathrm{S} \rangle$

Set of Items in DFA

Each state of the $LR(0)$ automaton represents a set of items in the canonical $LR(0)$ collection

Canonical $LR(0)$ Collection

Collection of sets of $LR(0)$ items, used as the basis for constructing a deterministic finite automaton

To construct the canonical $LR(0)$ collection, we define:

1. an augmented grammar
2. the functions CLOSURE and GOTO

Augmented Grammar

Let $G = \langle N, \Sigma, P, S \rangle$
Augmented grammar G' of G is defined by $\langle N, \Sigma, P \cup (\langle \mathrm{S'} \rangle \rightarrow \langle \mathrm{S} \rangle), S' \rangle$

- The purpose of the new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input
- Acceptance occurs when and only when the parser is about to reduce by $\langle \mathrm{S'} \rangle \rightarrow \langle \mathrm{S} \rangle$

Input : Set I of items for grammar $G = \langle N, \Sigma, P, S \rangle$
Output : CLOSURE(I)

begin
  CLOSURE(I) $\leftarrow$ I ;
  while *CLOSURE(I) has been changed* **do**
    foreach $(\langle A \rangle \rightarrow \alpha \bullet B\ \beta) \in$ *CLOSURE(I)* **do**
      if $(\langle B \rangle \rightarrow \gamma) \in P$ **then**
        CLOSURE(I) $\leftarrow$ CLOSURE(I) $\cup (\langle \mathrm{B} \rangle \rightarrow \bullet\, \gamma)$;
      end
    end
  end
end

$$
\begin{array}{lcl}
\langle \text{E'} \rangle & ::= & \langle \text{E} \rangle \\
\langle \text{E} \rangle & ::= & \langle \text{E} \rangle \ \mathbf{+} \ \langle \text{T} \rangle \\
 & ::= & \langle \text{T} \rangle \\
\langle \text{T} \rangle & ::= & \langle \text{T} \rangle \ \mathbf{*} \ \langle \text{F} \rangle \\
 & ::= & \langle \text{F} \rangle \\
\langle \text{F} \rangle & ::= & \mathbf{(} \ \langle \text{E} \rangle \ \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

I_0
$\langle \text{E'} \rangle \rightarrow \bullet \ \langle \text{E} \rangle$

$I = \{(E' \rightarrow \bullet \ E)\}$ and CLOSURE(I) = I_0

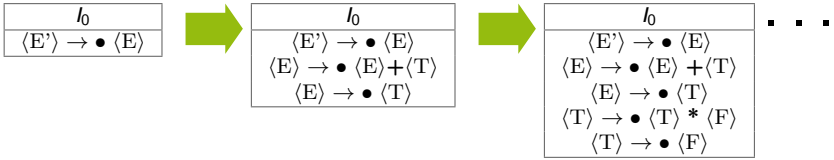
$$
\begin{array}{lcl}
\langle \text{E'} \rangle & ::= & \langle \text{E} \rangle \\
\langle \text{E} \rangle & ::= & \langle \text{E} \rangle + \langle \text{T} \rangle \\
 & ::= & \langle \text{T} \rangle \\
\langle \text{T} \rangle & ::= & \langle \text{T} \rangle * \langle \text{F} \rangle \\
 & ::= & \langle \text{F} \rangle \\
\langle \text{F} \rangle & ::= & (\langle \text{E} \rangle) \\
 & ::= & \textbf{id}
\end{array}
$$

I_0
$\langle \text{E'} \rangle \rightarrow \bullet \langle \text{E} \rangle$

I_0
$\langle \text{E'} \rangle \rightarrow \bullet \langle \text{E} \rangle$
$\langle \text{E} \rangle \rightarrow \bullet \langle \text{E} \rangle + \langle \text{T} \rangle$
$\langle \text{E} \rangle \rightarrow \bullet \langle \text{T} \rangle$

Consider E-productions because E is on the right of the dot. Add $\langle \text{E} \rangle \rightarrow \bullet \langle \text{E} \rangle + \langle \text{T} \rangle$ and $\langle \text{E} \rangle \rightarrow \bullet \langle \text{T} \rangle$ to I_0

$$
\begin{array}{rcl}
\langle \text{E'} \rangle & ::= & \langle \text{E} \rangle \\
\langle \text{E} \rangle & ::= & \langle \text{E} \rangle + \langle \text{T} \rangle \\
& ::= & \langle \text{T} \rangle \\
\langle \text{T} \rangle & ::= & \langle \text{T} \rangle * \langle \text{F} \rangle \\
& ::= & \langle \text{F} \rangle \\
\langle \text{F} \rangle & ::= & (\langle \text{E} \rangle) \\
& ::= & \textbf{id}
\end{array}
$$



Consider E-productions and T-productions because they are both on the right of the dot. Items for E-productions are already inside I_0, but not items for T-productions.

GOTO Function

Closure of the set of all items $\langle \mathrm{A} \rangle \rightarrow \alpha \ \langle \mathrm{X} \rangle \ \bullet \ \beta$
Such that $(\langle \mathrm{A} \rangle \rightarrow \alpha \ \bullet \ \langle \mathrm{X} \rangle \ \beta) \in I$
Where I is a set of items, and X is a grammar symbol

- Intuitively, the GOTO function is used to define the transitions in the $LR(0)$ automaton for a grammar
- States of the automaton corresponds to sets of items, and GOTO(I, X) specifies the transition from the state I under input X

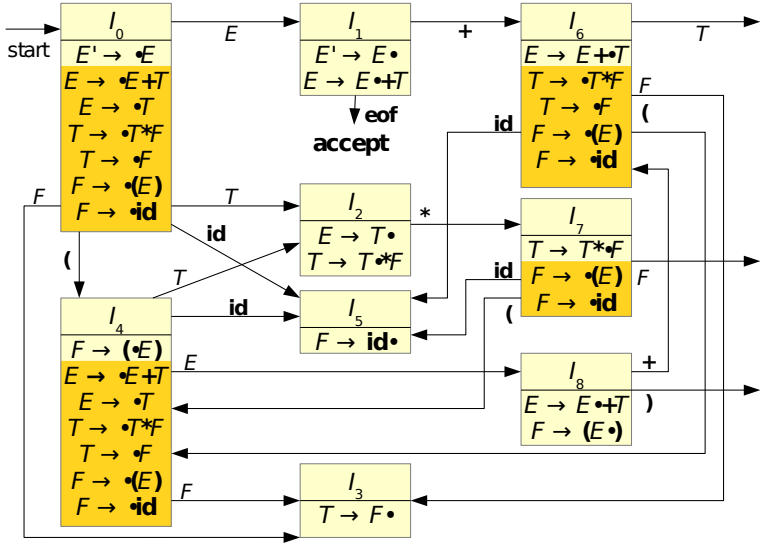
$$
\begin{array}{lcl}
\langle \text{E'} \rangle & ::= & \langle \text{E} \rangle \\
\langle \text{E} \rangle & ::= & \langle \text{E} \rangle \; \mathbf{+} \; \langle \text{T} \rangle \\
 & ::= & \langle \text{T} \rangle \\
\langle \text{T} \rangle & ::= & \langle \text{T} \rangle \; \mathbf{*} \; \langle \text{F} \rangle \\
 & ::= & \langle \text{F} \rangle \\
\langle \text{F} \rangle & ::= & \mathbf{(} \; \langle \text{E} \rangle \; \mathbf{)} \\
 & ::= & \mathbf{id}
\end{array}
$$

- If I is the set of two items $\{ (\langle \text{E'} \rangle \rightarrow \langle \text{E} \rangle \bullet), (\langle \text{E} \rangle \rightarrow \langle \text{E} \rangle \bullet \mathbf{+} \langle \text{T} \rangle) \}$
- Then, GOTO($I, +$) = CLOSURE($\{ (\langle \text{E} \rangle \rightarrow \langle \text{E} \rangle \mathbf{+} \bullet \langle \text{T} \rangle) \}$ is

$$
\begin{array}{c}
\langle \text{E} \rangle \rightarrow \langle \text{E} \rangle \mathbf{+} \bullet \langle \text{T} \rangle \\
\langle \text{T} \rangle \rightarrow \bullet \langle \text{T} \rangle \mathbf{*} \langle \text{F} \rangle \\
\langle \text{T} \rangle \rightarrow \bullet \langle \text{F} \rangle \\
\langle \text{F} \rangle \rightarrow \bullet \mathbf{(} \langle \text{E} \rangle \mathbf{)} \\
\langle \text{T} \rangle \rightarrow \bullet \langle \text{id} \rangle
\end{array}
$$

- Simple LR (or SLR) parsing constructs $LR(0)$ automaton from the grammar
- States of this automaton are the sets of items from the canonical $LR(0)$ collection, and the transitions are given by the GOTO function

- In the following slide, there is an example of $LR(0)$ automaton
  - Kernel items are in the light-yellow part of the box
  - Nonkernel items are in the dark-yellow part of the box
  - Egde represents the transitions given by the function GOTO, where the label is the token name

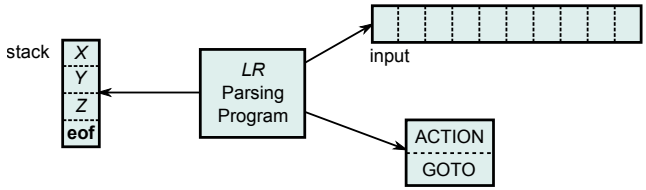

start
I_0
E' → •E
E → •E+T
E → •T
T → •T*F
T → •F
F → •(E)
F → •id
E
I_1
E' → E•
E → E•+T
eof
accept
+
I_6
E → E+•T
T → •T*F
T → •F
F → •(E)
F → •id
T
F
(
id
F
T
id
(
I_2
E → T•
T → T•*F
*
I_7
T → T*•F
F → •(E)
F → •id
id
F
(
I_4
F → (•E)
E → •E+T
E → •T
T → •T*F
T → •F
F → •(E)
F → •id
id
I_5
F → id•
E
I_8
E → E+•T
F → (E•)
+
)
F
I_3
T → F•

Input : Augmented grammar G'
Output : Canonical $LR(0)$ Collection, namely C

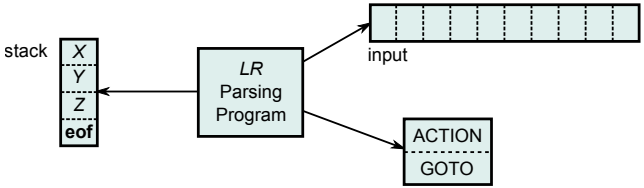
begin
  $C \leftarrow$ CLOSURE({ ($\langle$S'$\rangle \rightarrow \bullet \langle$S$\rangle$) });
  repeat
    foreach *set of items* $I \in C$ **do**
      foreach *grammar symbol in* X **do**
        if *GOTO*(I, X) *is not empty and not in* C **then**
          $C \leftarrow C \cup$ GOTO(I, X);
        end
      end
    end
  until *no new sets of items are added to* C *on a round*;
  return C;
end

- *LR* parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: ACTION and GOTO
- **Only the parsing table change from one parser to another**

- Parsing program reads characters from an input buffer one at a time
- It **shifts a state; not a symbol**. This is a major difference between a *LR* parser and a shift-reduce parser



- Stack holds a sequence of states, $s_0\ s_1 \ldots s_m$, where s_m is on top [1]
- All the transition that are entering in a state are labeled with the same symbol
  State may be associated to one symbol and only one, except for the start state



[1] In *SLR* method, the stack holds the states from the $LR(0)$ automaton; the canonical *LR* and *LALR* methods are similar

- This function takes as arguments a state s_i and a terminal **a** (or **eof**).
- The value of ACTION$[s_i, a]$ can have one of the four forms:
  1. **Shift** j, where s_j is a state
     Action taken by the parser effectively shifts input a to the stack, but uses state s_j to represent a
  2. **Reduce** $\langle \mathrm{A} \rangle \rightarrow \beta$
     Action of the parser effectively reduces β on the top of the stack to head A
  3. **Accept**
     Parser accepts the input and terminates
  4. **Error**
     Parser discovers an error and takes some corrective action

- The GOTO function, defined on sets of items, is extended to states.
- If GOTO$[I_i, A] = I_j$, then GOTO also maps a state I_i and a nonterminal A to state I_j.

Input : An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G
Output : If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication

```
begin
    S ← [s0]; a ← inputSymbol () ;
    stopParser ← false ;
    while ¬stopParser do
        s ← topOf (S) ;
        if ACTION[s, a] = Shift (t) then
            push (S, t) ;
            a ← inputSymbol () ;
        if ACTION[s, a] = Reduce (⟨A⟩ → β) then
            pop (S, β) ;
            t ← topOf (S) ;
            push (S, GOTO(t, A));
            print ( ⟨A⟩ → β) ;
        if ACTION[s, a] = Accept then
            stopParser ← true ;
        else
            throw("No production found")
        end
    end
end
```

- *SLR* method refers to the parsing table, the SLR table

- *SLR* method begins with $LR(0)$ items and $LR(0)$ automata:
  1. Given a grammar G, we augment G to produce G', with a new start symbol S'
  2. From G', we construct C, the canonical collection of sets of items for G' together with the GOTO function
  3. ACTION and GOTO entries in the parsing table are then constructed using the algorithm in the following slides
  4. This algorithm requires us to know FOLLOW(A) for each nonterminal A of the grammar

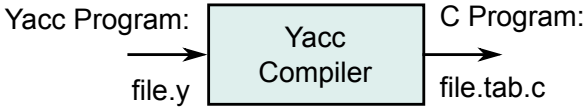
Input : Augmented grammar G'
Output : *SLR*-parsing table functions ACTION and GOTO for G'
begin
  $C \leftarrow \{I_0, I_1, \ldots, I_n\}$ // Sets of $LR(0)$ items for G' ;
  for $i \leftarrow 1$ **to** n **do**
    if $(\langle A \rangle \rightarrow \alpha \bullet \alpha\ \beta) \in I_i$ *and* $GOTO(I_i, \mathbf{a}) = I_j$ **then**
      ACTION$[i, a] \leftarrow$ "Shift j"
    else if $(\langle A \rangle \rightarrow \alpha \bullet \alpha) \in I_i$ **then**
      foreach a $\in$ *FOLLOW(A) and* $A \neq S'$ **do**
        ACTION$[i, \mathbf{a}] \leftarrow$ "Reduce $\langle \mathrm{A} \rangle \rightarrow \alpha$"
      end
    else if $(\langle S' \rangle \rightarrow \langle S \rangle) \in I_i$ **then**
      ACTION$[i, \mathbf{a}] \leftarrow$ "Accept"
    else
      `throw("G is not SLR(1)")`
    end
    foreach *nonterminal A* **do**
      if $GOTO(I_i, A) = I_j$ **then**
        GOTO$(s_i, A) \leftarrow s_j$
      end
    end
  end
  $S_0 \leftarrow$ state that corresponds to item $\langle \mathrm{S}' \rangle \rightarrow \langle \mathrm{S} \rangle \bullet$;
end

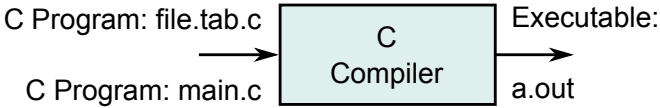
- *LALR* parsing (LookAhead *LR* parsing) if often used in practice (details in the reference books)
- Tables obtained by *LALR* methods are significantly smaller than tables obtained by canonical *LR* methods
- *LALR* parsers offer many of the advantages of *SLR* and canonical-*LR* parsers
- They combine the states that have the same kernels (sets of items, ignoring the associated lookahead sets)
- Thus, the number of states is the same as that of the *SLR* parser, but some parsing-action conflicts present in the *SLR* parser may be removed in the *LALR* parser
- *LALR* parsers have become the method of choice in practice

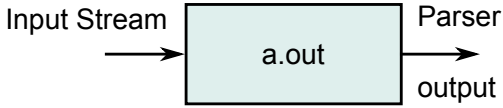
[DeRemer, 1969]

- Parser generators such as Yacc and its more recent implementation Bison, are generally LALR parser generators
- They permit to facilitate the creation of the front-end of a compiler by generating the source code from a grammar and a lexical analyzer specification
- This section describes two families of parser generators:
  - Yacc, or Bison, that generates C and C++ parsers
  - JavaCC, that generates Java parsers

[Johnson, 1975]


Yacc Program:
file.y
Yacc
Compiler
C Program:
file.tab.c


C Program: file.tab.c
C Program: main.c
C
Compiler
Executable:
a.out


Input Stream
a.out
Parser
output

- A Yacc program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Declarations

- C ordinary declarations, between %{ and %}
- Declarations of tokens with the command %token

- A Yacc program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Translation rules

Each rule consists of a grammar production and the associated action (note the final semicolon)

```
<head>  :  <body_1> { <action_1> }
        |  <body_2> { <action_2> }
           ...
        |  <body_n> { <action_n> }
        ;
```

- A Yacc program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

Auxiliary functions

- Auxiliary functions are the section where additional C routines should be put
- Note that you must provide the function yylex(), which is invoking the lexical analyzer (explained later)

```
%{
    #include <ctype.h>
%}
%token DIGIT
%%
line    :   expr '\n'         { printf("%d\n", $1); }
        ;
expr    :   expr '+' term     { $$ = $1 + $3; }
        | term
        ;
term    :   term '*' factor   { $$ = $1 * $3; }
        | factor
        ;
factor  :   '(' expr ')'      { $$ = $2; }
        | DIGIT
        ;
%%
int yylex() {
   int c;
   c = getchar();
   if (isdigit(c)) {
      yylval = c - '0'; /* convert char to int */
      return DIGIT;
   }
   return c;
}
```

- Yacc provides a set of declarations that may be used to remove grammar ambiguity.

- Associativity and Precedence:
  - Left associativity: %left <op1> <op2>...
  - Right associativity: %right <op3> <op4>...
  - No associativity: %nonassoc <op5> <op6>...
  - The tokens are given precedences in the order in which they appear in the declaration part, lower first
  - Tokens in the same declaration have the same precedence

- Lex was designed to produce lexical analyzers that could be used with Yacc
- Lex library provides a driver program named `yylex()`
- To use Lex in Yacc, you must remove any definition of `yylex()` in the Yacc specification; and replace this definition by:

```
#include "lex.yy.c"
```

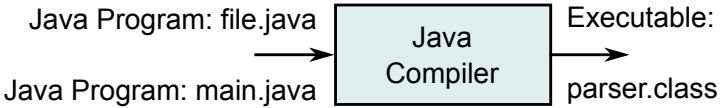
- All the tokens defined in the Yacc declarations are directly available in the Lex program

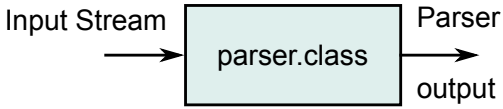
- In Yacc, error recovery uses a form of error productions
- First, you must decides what “major” nonterminals will have error recovery associated to them
- Typical choices are some subset of the nonterminals generating expressions, statements, blocks, and functions

- `yyerror()` reports an error
- `yyerrok()` resets the parser to its normal mode of operation
- Here, the error production causes the program to suspend normal parsing when a syntax error is found on an input line
- On encountering the error, the parser in the program starts popping symbols from its stack until it encounters a statethat as a shift action on the token error. Then the input is read until the new-line character is read. Then the parser reduces error '\n' to lines, and emits the diagnotic message "error message"

```
line    :   lines expr '\n'  { printf("%g\n", $2); }
        |   lines '\n'
        |   /* empty or epsilon */
        |   error '\n'       { yyerror("error message");
                               yyerrok(); }
```


JavaCC Program:
file.jj
JavaCC
Compiler
Java Program:
file.java


Java Program: file.java
Java Program: main.java
Java
Compiler
Executable:
parser.class


Input Stream
parser.class
Parser
output

- A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Parser Definition

- The name that follows “PARSER_BEGIN” and “PARSER_END” must be the same and this identifies the name of the generated parser

- A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Options

- JavaCC options permits to control the behavior of the parser

- A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Java compilation unit

The Java compilation unit is a Java code that must contain at least the declaration of the class of the parser:

```
...
public class <parser_name> {
...
}
...
```

- A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Predefined functions in the `parser_name` class

Two functions are automatically generated inside the parser class:

- Token getNextToken(): returns the next available token
- Token getToken(**int** index): returns the ith token ahead

- A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Translation rules

- Java code production (see error recovery for an example)
- Regular expression definitions for tokens
- Grammar (BNF-like) productions

Definition

```
[<state_list>] <kind> [IGNORE_CASE] :
{ <regexpr> | <regexpr> | ...}
```

- <state_list> specifies the lexer states in which the rule is enabled (default is DEFAULT)
- **IGNORE_CASE** specifies, by its presence, that if the regular expression is case sensitive or case insensitive
- The regular definitions are defined and used as follows, respectively (The "#" before the id indicates that this definition exists solely for the purpose of defining other tokens):
  < [#]id : regexpr >
  <id>

1. TOKEN: describes tokens in the grammar. The token manager creates a Token object for each match of such a regular expression and returns it to the parser
2. SPECIAL_TOKEN: like tokens, except that they do not have significance during parsing, ie. the BNF productions ignore them
3. SKIP: simply skipped (ignored) by the token manager
4. MORE: Sometimes it is useful to gradually build up a token to be passed on to the parser. Matches to this kind of regular expression are stored in a buffer until the next **TOKEN** or **SPECIAL_TOKEN** match

- **int** kind
  This is the index for this kind of token in the internal representation scheme of JavaCC. It may be replaced by a constant

- **int** beginLine, beginColumn, endLine, endColumn
  The beginning and ending positions of the token as it appeared in the input stream

- String image
  The lexeme of the token as it appeared in the input stream

- Token next
  A reference to the next regular (non-special) token from the input stream

- Object getValue()
  An optional attribute value of the Token. Tokens which are not used as syntactic sugar will often contain meaningful values that will be used later on by the compiler or interpreter. This attribute value is often different from the image. Any subclass of Token that actually wants to return a non-null value can override this method as appropriate

- **static final** Token newToken(**int** ofKind)
  static final Token newToken(**int** ofKind, String image)
  Returns a new token object as its default behavior

Definition

```
<access_modifier> <return_type> <identifier> ( <parameters> ) :
<java_block>
{ <expansion_choices> }
```

- Name of the non-terminal ⇒ name of the method
  - parameters and return value depends on the goal of the compiler
- Calls to non-terminals ⇒ function calls
- Default access modifier: **public**

Definition

```
<access_modifier> <return_type> <identifier> ( <parameters> ) :
<java_block>
{ <expansion_choices> }
```

- Java block: arbitrary Java declarations and code put at the beginning of the method generated for the Java non-terminal
- Expansion choices: a sequence of expansion units. Each nonterminal is written as a function call. Semantic actions are Java blocks inside this part of the BNF production

```
PARSER_BEGIN(CalculatorParser)
    public class CalculatorParser {
    }
PARSER_END(CalculatorParser)

SKIP: {
        " "
    |   "\t"
    |   "\n"
    |   "\r"
}

TOKEN :
{
     <DIGIT : [0-9]>
}
```

⟨line⟩	::=	⟨expr⟩
⟨expr⟩	::=	⟨expr⟩**+**⟨term⟩
	::=	⟨term⟩
⟨term⟩	::=	⟨term⟩*****⟨factor⟩
	::=	⟨factor⟩
⟨factor⟩	::=	**(**⟨expr⟩**)**
	::=	**digit**

```
private void line() :
{
  int e;
}
{ e = expr()        { System.out.println(e); }
}

private int expr() :
{
  int e, t;
}
{ e = expr() "+" t = term()    { return e+t; }
  | t = term()                 { return v; }
}
```

```
private int term() :
{
    int t, f;
}
{ t = term() "*" f = factor()    { return t*f; }
  | f = factor()                 { return f; }
}

private int factor() :
{
    int e, d;
}
{ "(" e = expr() ")"             { return e; }
  | d = <DIGIT>                  { return Integer.parseInt(d.image); }
}
```

- Modify file `ParseException.java` : e.g. changing `getMessage` method for customizing error reporting
- See Javadoc in `ParseException.java` and `TokenMgrError.java` for details

- Override or call `generateParseException()` : (in the generated parser) it generates an object of type `ParseException`

JavaCC offers two kinds of error recovery:

1. Shallow recovery: recovers if none of the current choices have succeeded in being selected

2. Deep recovery: is when a choice is selected, but then an error happens sometime during the parsing of this choice

When no token found, we want to skip until the next given symbol (semi-column)

```
TOKEN : { <SEMICOLON: ";"> }
private void stm() :
{}
{ ifStm()
  | whileStm()
}
```

```
TOKEN : { <SEMICOLON: ";"> }
private void stm() :
{}
{ ifStm()
  | whileStm()
  | error_skipto(SEMICOLON)
}
```

- `error_skipto()` is a nonterminal that must be define prior to its first usage
- To do so, we must use the following code:

```
//JAVACODE
void error_skipto(int kind) {
    ParseException e = generateParseException()
    System.err.println(e);
    Token t;
    do {
        t = getNextToken();
    }
    while (t.kind != kind);
}
```

```
TOKEN : { <SEMICOLON: ";"> }
private void stm() :
{}
{ ifStm()
  | whileStm()
}
```

When error occurs (even deeper in the parse tree), we want to recover

```
TOKEN : { <SEMICOLON: ";"> }
private void stm() :
{}
{ try {
      ifStm()
      | whileStm()
  } catch(ParseException e) {
      error_skipto(e, SEMICOLON);
  }
}
```

```
//JAVACODE
void error_skipto(ParseException e, int kind) {
    System.out.println(e);
    Token t;
    do {
        t = getNextToken();
    }
    while (t.kind != kind);
}
```

- Parsers: A parser takes as input tokens from the lexical analyzer and treats the token names as terminal symbols of a context-free grammar. The parser then constructs a parse tree for its input sequence of tokens; the parse tree may be constructed figuratively or literally
- Context-Free Grammars: A grammar specifies a set of terminal symbols (inputs), another set of nonterminals (symbols representing syntactic constructs), and a set of productions, each of which gives a way in which strings represented by one nonterminal can be constructed from terminal symbols and strings represented by certain other nonterminals. A production consists of a head (the nonterminal to be replaced) and a body (the replacing string of grammar symbols)
- Derivations: The process of starting with the start-nonterminal of a grammar and successively replacing it by the body of one its productions is called derivation. If the leftmost (or rightmost) nonterminal is always replaced, then the derivation is called leftmost (resp. rightmost)

- Parse Trees: A parse tree is a picture of a derivation, in which there is a node for each nonterminal that appears in the derivation. The children of a node are the symbols by which that nonterminal is replaced in the derivation. There is a one-to-one correspondence between parse trees, leftmost derivation, and rightmost derivations of the same terminal string
- Ambiguity: A grammar for which some terminal string has two or more different parse tree is said to be ambiguous
- Top-Down and Bottom-Up Parsing: Parsers are generally distinguished by whether they work top-down or bottom-up. Top-down parsers include recursive-descent and LL parsers, while the most common forms of bottom-up parsers are LR parsers
- Design of Grammars: Grammars suitable for top-down parsing often are harder to design than those used by bottom-up parsers. It is necessary to eliminate left-recursion. We also must left-factor/group productions for the same nonterminal that have a common prefix in the body

- Recursive-Descent Parsers: These parsers use a procedure for each nonterminal
- $LL(1)$ Parsers: A grammar such that it is possible to choose the correct production with which to expand a given nonterminal, looking only at the next input symbol, is called $LL(1)$. These grammars allow us to construct a predictive parsing table that gives, for each nonterminal and each lookahead symbol, the correct choice of production
- Shift-Reduce Parsing: Bottom-up parsers generally operate by choosing on the basis of the next input symbol and the contents of the stack, whether to shift the next input onto the stack, or to reduce some symbols at the top of the stack. A reduce takes a production body at the top of the stack and replaces it by the head of the production
- Viable Prefixes: In shift-reduce parsing, the stack contents are always a viable prefix, ie. a prefix of some right-sentential form that ends no further right than the end of the handle of that right-sentential form. The handler is the substring that was introduced in the last step of the rightmost derivation of that sentential form

- Valid Items: An item is a production with a dot somewhere in the body. An item is valid for a viable prefix if the production of that item is used to generate the handler, and the viable prefix includes all those symbols to the left of the dot
- *LR* Parsers: Each of the several kinds of LR parsers operate by first constructing the sets of valid items (called LR states) for all possible viable prefixes, and keeping track of the state for each prefix on the stack. The set of valid items guide the shift-reduce parsing decision
- Simple *LR* Parsers: In an *SLR* parser, we perform a reduction implied by a valid item with a dot at the right end, provided the lookahead symbol can follow the head of that production in some sentential form
- Canonical-*LR* Parsers: This more complex form of *LR* parser uses items that are augmented by the set of lookahead symbols that can follow the use of the underlying production. A canonical-*LR* parser can avoid some of the parsing-action conflicts that are present in *SLR* parsers, but often has many more states than the *SLR* parser for the same grammar

Backus, J. (1959).
The syntax and semantics of the proposed international algebraic language of the zurich-ACM-GAMM conference.
In *Intl. Conf. Information Processing*, pages 125–132, Paris. UNESCO.

Birman, A. and Ullman, J. (1973).
Parsing algorithms with backtrack.
Information and Control, 23:1–34.

Cantor, D. (1962).
On the ambiguity problem of backus systems.
In *J. ACM*, volume 9, pages 477–479. ACM.

Chomsky, N. (1956).
Three models for the description of language.
IRE Trans. on Information Theory, 2(3):113–124.

Dain, J. (1991).
Bibliography on syntax error handling in language translation systems.
Available from the `comp.compilers` newsgroup.

DeRemer, F. (1969).
Practical Translators for LR(k) Languages.
PhD thesis, MIT, Cambridge, MA.

DeRemer, F. (1971).
Simple *LR*(*k*) grammars.
Comm. ACM, 14(7):453–460.

Earley, J. (1970).
An efficient context-free parsing algorithm.
Comm. ACM, 13(2):94–102.

Floyd, R. (1962).
On the ambiguity in phase-structure languages.
In *Comm. ACM*, volume 5, pages 526–534. ACM.

Hoare, C. (1962).
Report on the elliott ALGOLtranslator.
Computer J., 5(2):127–129.

Hopcroft, J., Motwani, R., and Ullman, J. (2006).
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Boston, MA.

Ingerman, P. (1967).
Panini-backus form suggested.
In *Comm. ACM*, volume 10, page 137. ACM.

Johnson, S. (1975).
Yacc - yet another compiler compiler.
Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ.

Kasami, T. (1965).
An efficient recognition and syntax analysis algorithm for context-free languages.
Technical Report AFCRL-65-768, Air Force Cambridge Research Laboratory, Bedford, MA.

Knuth, D. (1965).
On the translation of languages from left to right.
Information and Control, 8(6):607–639.

Korenjak, A. (1969).
A practical method for constructing $LR(k)$ processors.
Comm. ACM, 8(11):613–623.

Lewis, P. and Stearns, R. (1968).
Syntax-directed transduction.
J. ACM, 15(3):465–488.

McClure, R. (1965).
TMG: a syntax-directed compiler.
In *20th ACM Natl. Conf.*, pages 262–274.

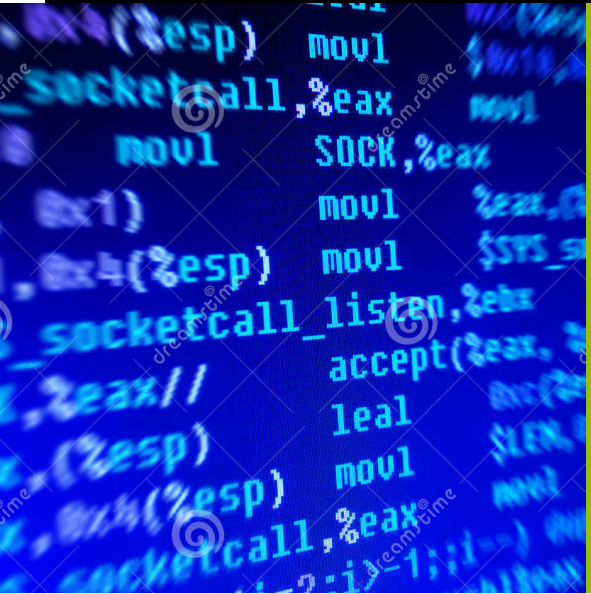
Naur, P. e. a. (1963).
Report on the algorithmic language ALGOL 60.
In *Comm. ACM*, volume 3, pages 299–314. ACM.

Schorre, D. (1964).
Meta-II: a syntax-oriented compiler writing language.
In *19th ACM Natl. Conf.*, pages D1.3–1–D1.3–11.

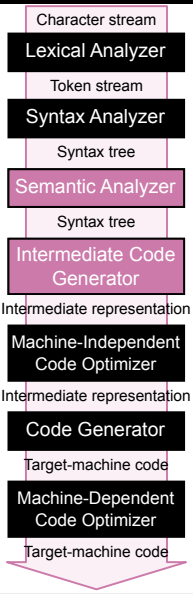
Younger, D. (1967).
Recognition and parsing of context-free languages in time n^3.
Information, 10:189–208.

Chapter 4
Semantic Analysis and Intermediate Code Generation

Stéphane GALLAND



- Translation of languages guided by context-free grammars
- Type checking for source language
- Generation of parse tree or intermediate code
- Parse tree is an abstract representation of the program
- Intermediate code is assembly language independent of any platform



- Syntax-directed translation specifies the values of attributes, attached to the grammar symbols, by associating semantic rules with the grammar productions

Grammar Productions			Semantic Rules
expr	::=	⟨expr⟩ + ⟨term⟩	head.t = expr.t | term.t | '+'
	|	⟨expr⟩-⟨term⟩	head.t = expr.t | term.t | '-'
	|	⟨term⟩	head.t = term.t
term	::=	**0**	head.t = '0'
	|	**1**	head.t = '1'
...			...
	|	**9**	head.t = '9'

- Semantic rules are program fragments (or semantic actions, between braces)

[Samelson and Bauer, 1960, Brooker and Morris, 1962, Paakki, 1995]

Construct a parse tree → Compute the attributes' values of the tree nodes

In many cases, translation can be done during parsing, without building an explicit tree in memory

L-attributed Translations
L=left

S-attributed Translations
S=synthesized

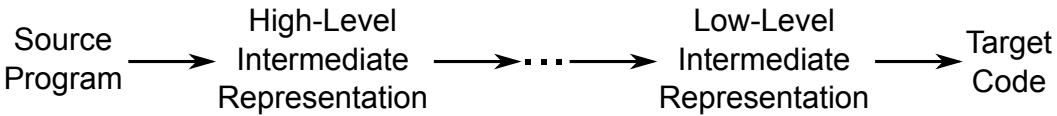
Type Checking

Ensures that operators are applied to compatible operands

Syntactic Checking

Apply additional syntactic check that are not done by the syntax analyzer

In translating a program to target machine code, a compiler may construct a sequence of intermediate representations



High-level representation is close to the source language, e.g., syntax tree

Low-level representation are close to the target machine, e.g., three-address code

Syntax-Directed Definition — SDD

Context-free grammar together with attributes and semantic rules

- Attribute: is associated with a nonterminal or terminal
- Semantic Rule: is associated with production, and a production could be associated to $[0; n]$ rules

Productions			Semantic Rules
expr	::=	⟨expr⟩ **+** ⟨term⟩	head.t = expr.t | term.t | '+'
	|	⟨expr⟩ - ⟨term⟩	head.t = expr.t | term.t | '-'
	|	⟨term⟩	head.t = term.t
term	::=	**0**	head.t = '0'
	|	**1**	head.t = '1'
...			...
	|	**9**	head.t = '9'

- Notation during the lectures:

Productions			Semantic Rules
A	$::=$	$\langle \mathrm{B} \rangle \langle \mathrm{C} \rangle$	First part of Rule 1
		$\langle \mathrm{D} \rangle \langle \mathrm{E} \rangle$	Second part of Rule 1
	$\mid$	$\langle \mathrm{F} \rangle$	Rule 2
G	$::=$	$\langle \mathrm{H} \rangle \langle \mathrm{I} \rangle \langle \mathrm{J} \rangle$	Rule 3

- Notation during the tutorial sessions and labworks:

$\langle \mathrm{A} \rangle \rightarrow \langle \mathrm{B} \rangle \langle \mathrm{C} \rangle$ `{ First part of Rule 1 }` $\langle \mathrm{D} \rangle \langle \mathrm{E} \rangle$ `{ Second part of Rule 1 }`

$\mid \langle \mathrm{F} \rangle$ `{ Rule 2 }`

$\langle \mathrm{G} \rangle \rightarrow \langle \mathrm{H} \rangle \langle \mathrm{I} \rangle \langle \mathrm{J} \rangle$ `{ Rule 3 }`

Attribute

Any quantity associated with a programming construct (nonterminal or terminal)

Examples

Data types, number of instructions, line of the first occurrence of an identifier...

- In this lecture, attributes are written following one of the notations:

```
<terminal>.<attribute name>
<nonterminal>.<attribute name>
```

- Keyword "head" represents the nonterminal in the production's head
- If the same nonterminal is present many times in the body, the attribute's prefix is indexed by the position of the nonterminal in this body (see green example below)

expr	::=	⟨expr⟩ + ⟨expr⟩	head.t = expr_1.t + expr_2.t
	|	⟨expr⟩ - ⟨expr⟩	head.t = expr_1.t - expr_2.t
	|	⟨term⟩	head.t = term.t
term	::=	**0**	head.t = 0
	|	**1**	head.t = 1
...			...
	|	**9**	head.t = 9

Synthesized Attribute

Attribute for a nonterminal A head at a parse-tree node N, defined in a semantic rule associated with the production at N

Synthetized attribute is defined by a semantic rule

Inherited Attribute

Attribute for a nonterminal B in the body at a parse-tree node N, defined in a semantic rule associated with the production at the parent of N

Inherited attribute is read in a semantic rule

[Knuth, 1968]

To visualize the translation specified by an SDD, it helps to work with parse trees

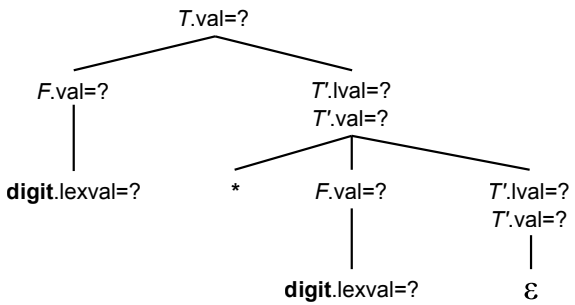
- Parse tree: representation of the grammar derivations
- Annotated Parse Tree: parse tree with attributes

Parse tree $\neq$ Syntax tree (source program representation, see slides later)

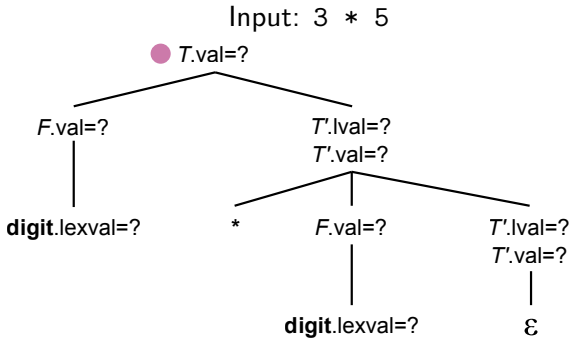
Depth-first iteration on the parse tree enables to execute the semantic rules in the right order, i.e., based on attributes' dependencies (see later)

T	::=	$\langle \text{F} \rangle \langle \text{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \text{F} \rangle \langle \text{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \text{digit} \rangle$	head.val = **digit**.lexval

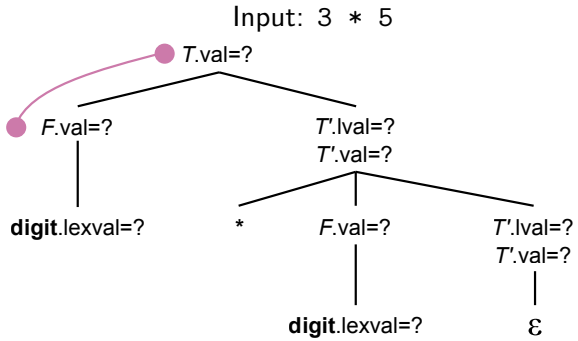
Input: 3 * 5



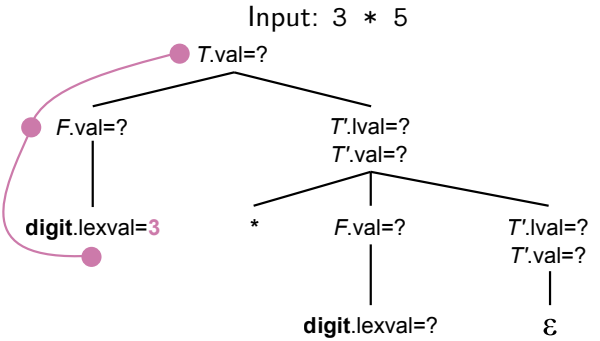
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



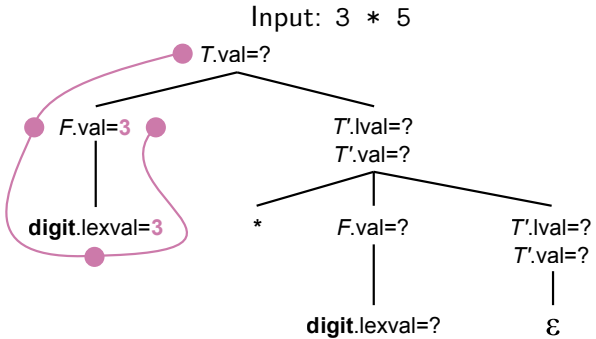
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



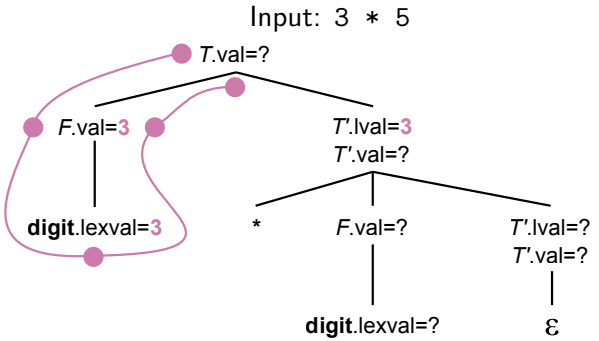
T	::=	⟨F⟩ ⟨T'⟩	T'.lval = F.val
			head.val = T'.val
T'	::=	* ⟨F⟩ ⟨T'⟩	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	⟨digit⟩	head.val = **digit**.lexval



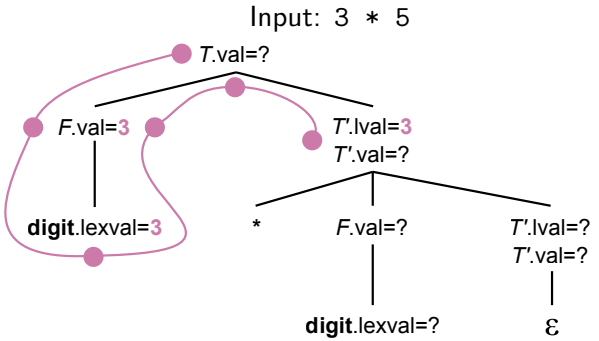
T	::=	⟨F⟩ ⟨T'⟩	T'.lval = F.val
			head.val = T'.val
T'	::=	* ⟨F⟩ ⟨T'⟩	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	⟨digit⟩	head.val = **digit**.lexval



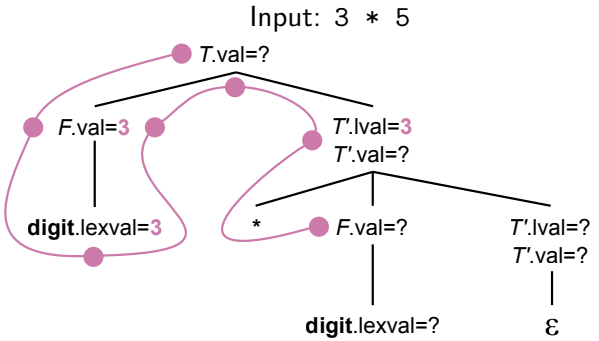
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val
			head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



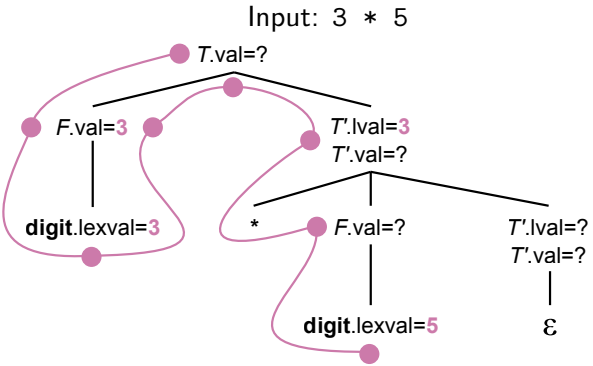
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



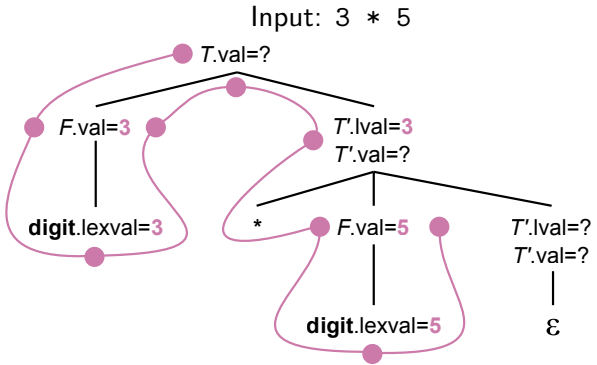
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val
			head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



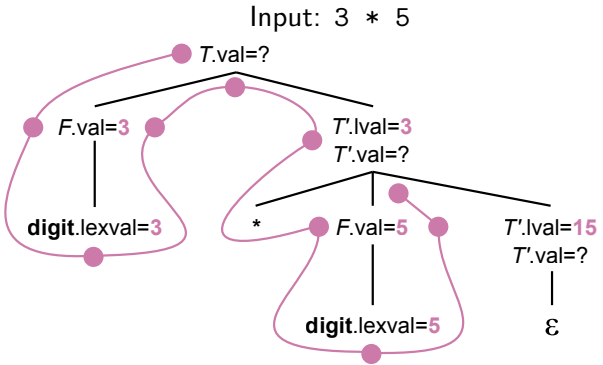
T	::=	⟨F⟩ ⟨T'⟩	T'.lval = F.val
			head.val = T'.val
T'	::=	* ⟨F⟩ ⟨T'⟩	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	⟨digit⟩	head.val = **digit**.lexval



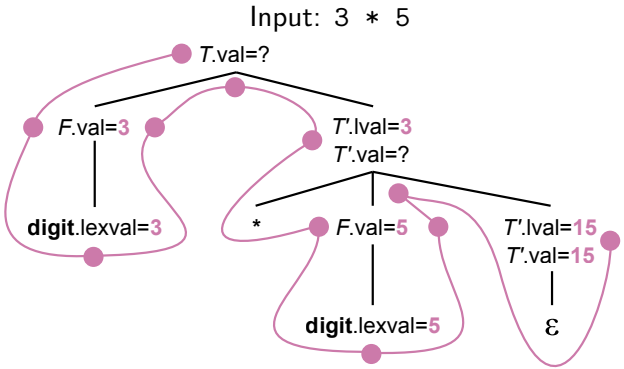
T	::=	⟨F⟩ ⟨T'⟩	T'.lval = F.val
			head.val = T'.val
T'	::=	* ⟨F⟩ ⟨T'⟩	T'.lval = head.lval * F.val
			head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	⟨digit⟩	head.val = **digit**.lexval



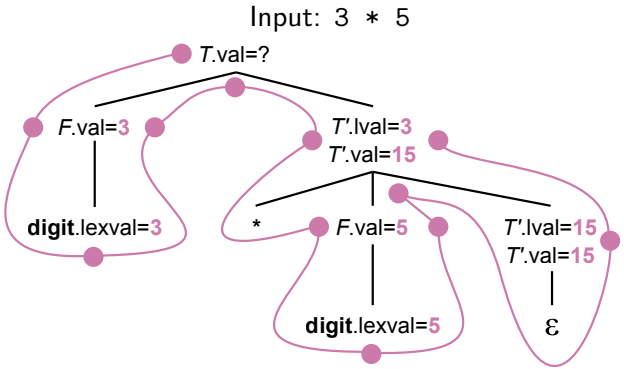
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



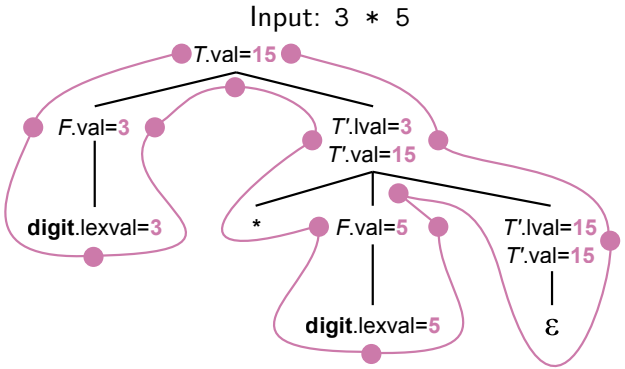
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \mathrm{digit} \rangle$	head.val = **digit**.lexval



T	::=	$\langle \text{F} \rangle \langle \text{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \text{F} \rangle \langle \text{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	$\langle \text{digit} \rangle$	head.val = **digit**.lexval



How to determine the correct sequence of evaluations of the semantic rules' lines?

Introduction of a graph of dependencies between the attributes

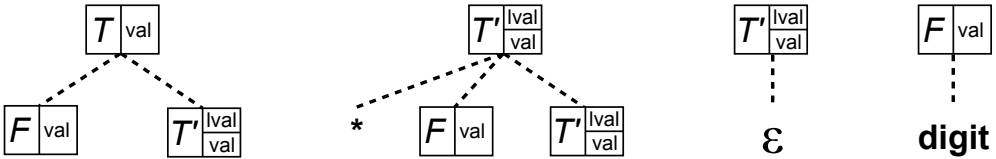
Dependency Graph

Flow of information among the attribute instances in a particular parse tree

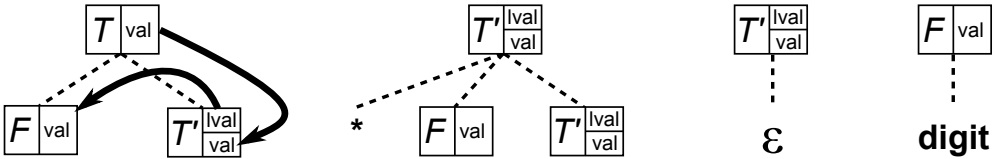
- Node: For each parse-tree node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X
- Edge: Between two attribute instances: the value of the first is needed to compute the value of the second

[Knuth, 1968]

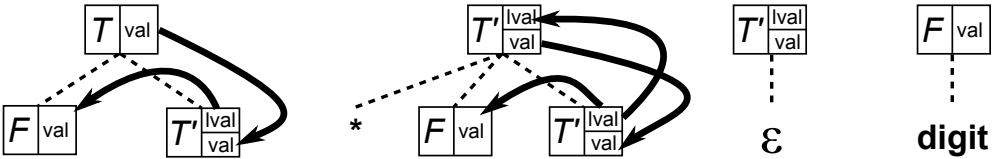
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



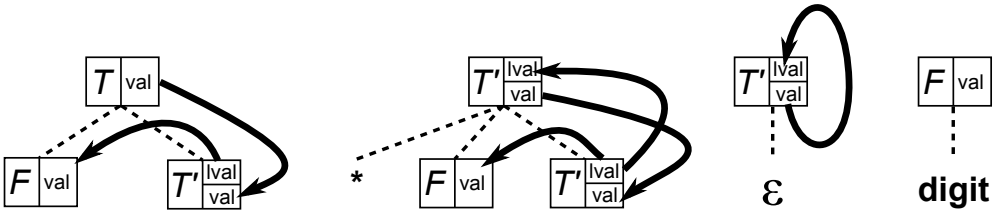
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



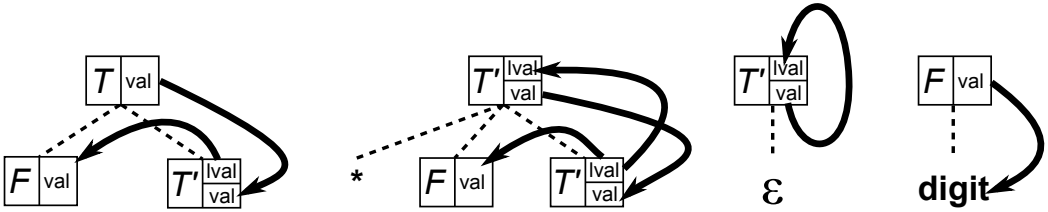
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



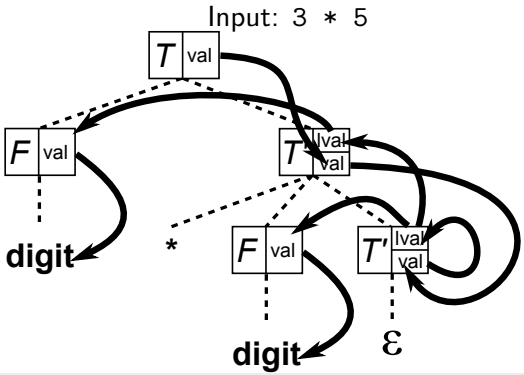
T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



T	::=	$\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = F.val head.val = T'.val
T'	::=	* $\langle \mathrm{F} \rangle \langle \mathrm{T'} \rangle$	T'.lval = head.lval * F.val head.val = T'.val
	|	ϵ	head.val = head.lval
F	::=	**digit**	head.val = **digit**.lexval



Order of the attributes

Let $N_1, N_2, \ldots, N_k$ the evaluation sequence of the dependency graph's nodes
Such that $i < j \implies \exists$ an edge from N_i to N_j

Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph

[Knuth, 1968]

If there is any cycle in the graph, then there are no topological sorts, i.e., there is no way to evaluate the SDD on the parse tree

Given a SDD, it is very hard to cycle

Translations can be implemented using classes of SDD that guarantee an evaluation order, i.e., without cycle

Two classes could be used:

1. S-attributed definition (bottom-up)
2. L-attributed definition (top-down)

[Knuth, 1968, Jazayeri et al., 1975]

S-Attributed Definition

SDD in which all the attributes are synthesized

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal of the parse tree

[Irons, 1961]

L-Attributed Definition

SDD in which, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left

Each attribute must be:

- Synthesized, or
- Inherited, with the rules limited as follows.
  Suppose a production $\langle \mathrm{A} \rangle \rightarrow X_1 X_2 \dots X_n$, and an inherited attribute $X_i.a$. The rule may uses only:
  a) Inherited attributes associated with the head A.
  b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1 X_2 \dots X_{i-1}$ located to the left of X_i
  c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycle in a graph dependency formed by the attributes of this X

[Lewis et al., 1974]

- Since some compilers use syntax tree as an intermediate representation, a common form of SDD turns its input string into a tree

- To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules than the parse tree

Syntax Tree

Tree defined as $\langle N, C \rangle$, where:

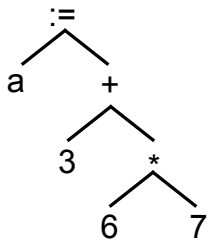
- N is the set of nodes;
  Each node $n \in N$ represents a language construct
- $C : N \to \mathcal{P}N$ is the function that maps a node to its child nodes;
  Each child node $c \in C(n)$ is one of the meaningful components of n

Implementation Notes

- Each object representing n has a field that is the label of the node, and the following additional fields:
  - If the node is a leaf, the lexical value for the leaf
  - If the node is not a leaf, all the children are stored in individual fields

This is the syntax tree for the statement:

```
a := 3 + ( 6 * 7 )
```



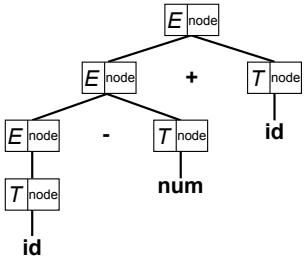
E	::=	$\langle \mathrm{E} \rangle$ + $\langle \mathrm{T} \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle \mathrm{E} \rangle$ - $\langle \mathrm{T} \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle \mathrm{T} \rangle$	head.node = T.node
T	::=	**(** $\langle \mathrm{E} \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

- Semantic rules contain the creation of the syntax tree nodes
- Root of the syntax tree becomes `E.node`

Annotated parse tree is implicitly defined by the grammar rules and not directly built by the compiler

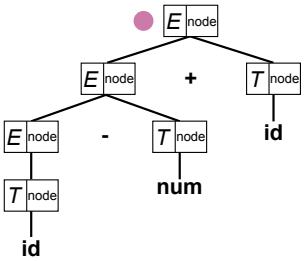
E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: `a - 4 + c`



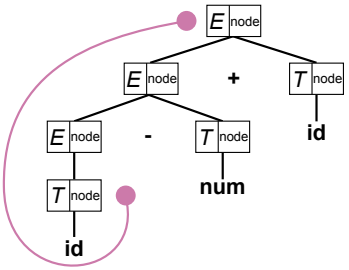
E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: `a - 4 + c`



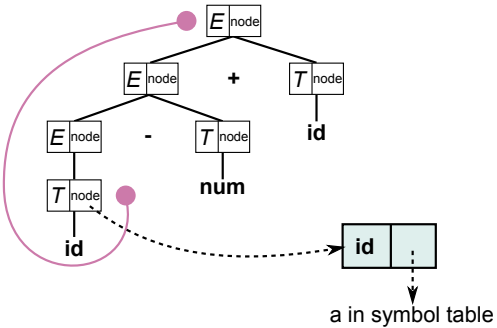
E	::=	⟨E⟩ + ⟨T⟩	head.node = **new** Node("+", E.node, T.node)
	|	⟨E⟩ - ⟨T⟩	head.node = **new** Node("−", E.node, T.node)
	|	⟨T⟩	head.node = T.node
T	::=	**(** ⟨E⟩ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: `a - 4 + c`



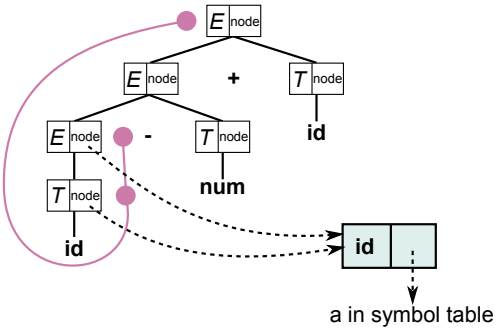
E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: `a - 4 + c`



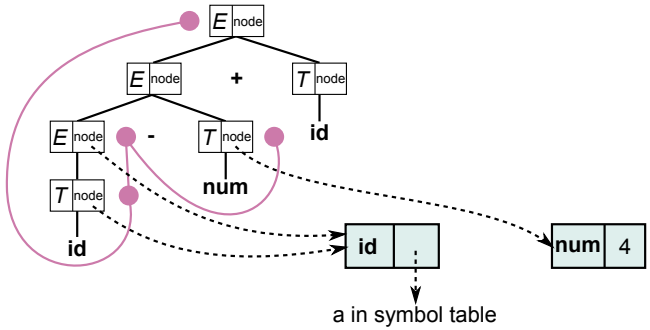
E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: a − 4 + c



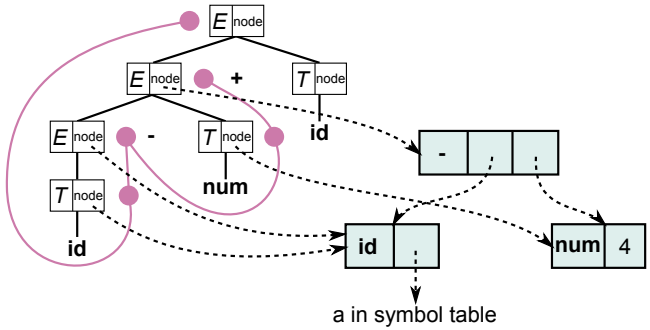
E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: a − 4 + c



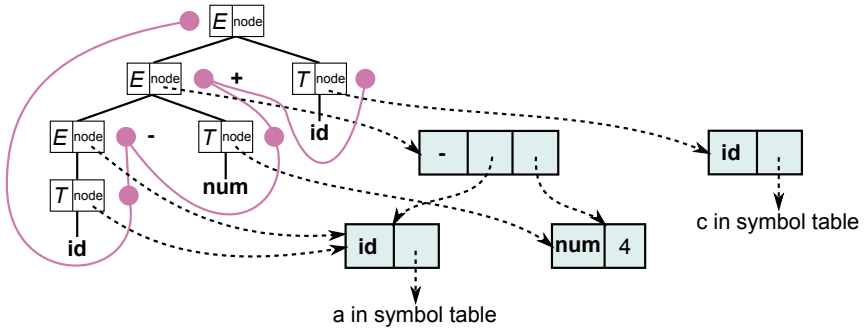
E	::=	⟨E⟩ + ⟨T⟩	head.node = **new** Node("+", E.node, T.node)
	|	⟨E⟩ - ⟨T⟩	head.node = **new** Node("−", E.node, T.node)
	|	⟨T⟩	head.node = T.node
T	::=	**(** ⟨E⟩ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: a - 4 + c

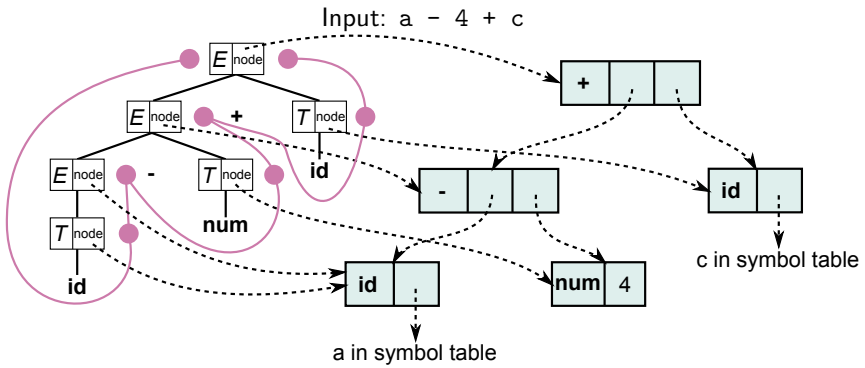


E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

Input: a − 4 + c



E	::=	$\langle E \rangle$ + $\langle T \rangle$	head.node = **new** Node("+", E.node, T.node)
	|	$\langle E \rangle$ - $\langle T \rangle$	head.node = **new** Node("−", E.node, T.node)
	|	$\langle T \rangle$	head.node = T.node
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf(**id**, **id**.lexeme)
	|	**num**	head.node = **new** Leaf(**num**, **num**.value)

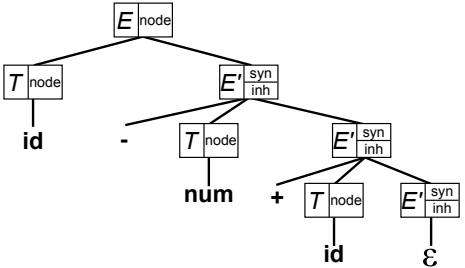


E	::=	$\langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle \mathrm{T} \rangle \langle \mathrm{E'} \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle \mathrm{E} \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Production	Attribute	Description
E	node	Global syntax tree node
E'	inherited	Node computed by production in the parent's node
	synthesized	Intermediate node computed by the current production
T	node	Node of an atomic expression

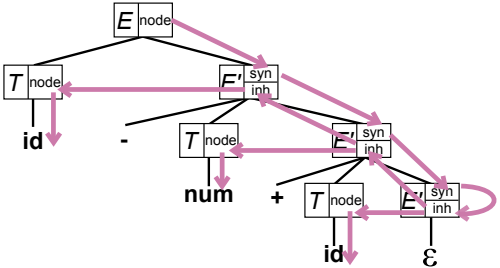
E	::=	$\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle$E$\rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



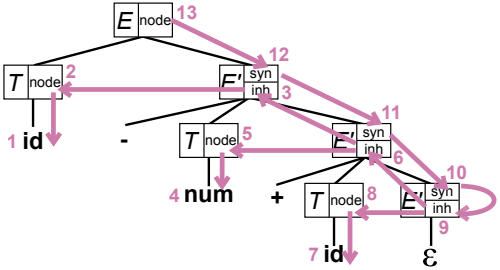
E	::=	$\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = T.node
			head.node = E'.synthesized
E'	::=	+ $\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = **new** Node("+", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	- $\langle$T$\rangle$ $\langle$E'$\rangle$	E'.inherited = **new** Node("-", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle$E$\rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



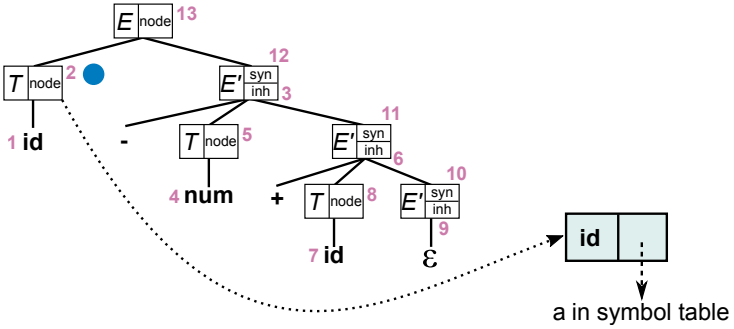
E	::=	⟨T⟩ ⟨E'⟩	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ ⟨T⟩ ⟨E'⟩	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- ⟨T⟩ ⟨E'⟩	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** ⟨E⟩ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



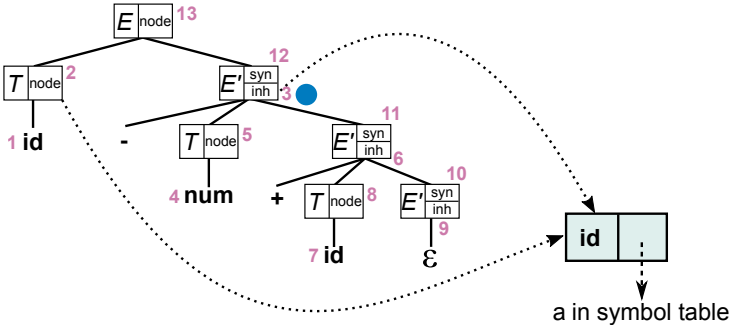
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



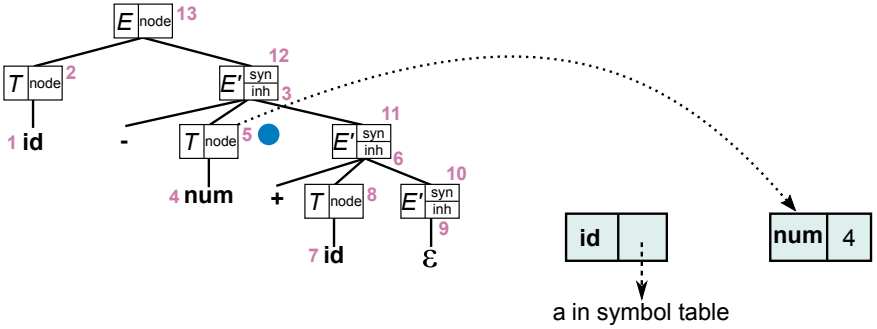
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



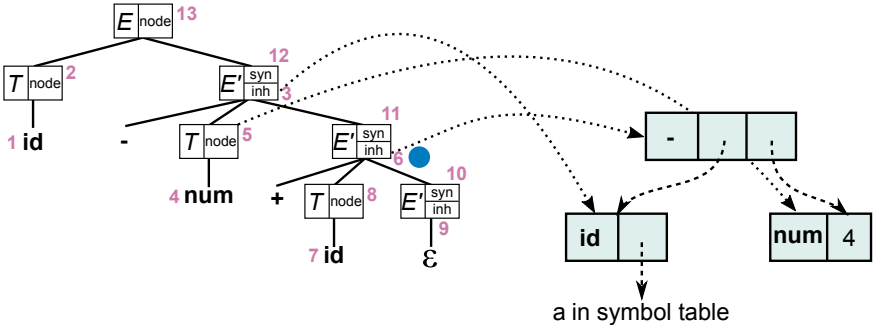
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a - 4 + c



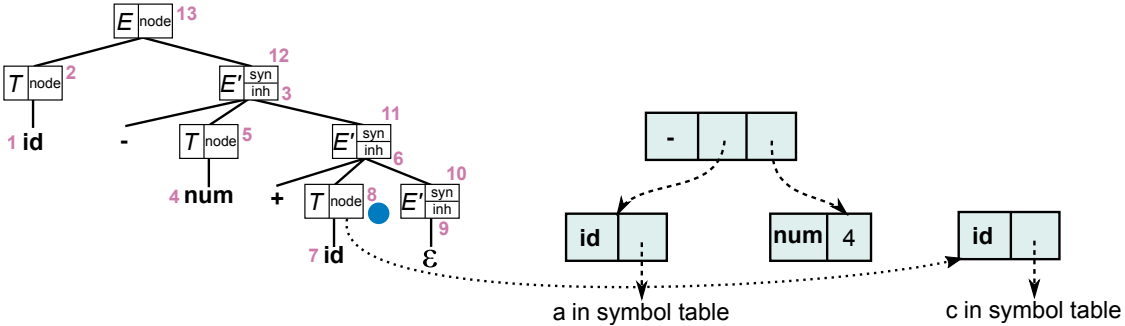
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node
			head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

nput: a − 4 + c



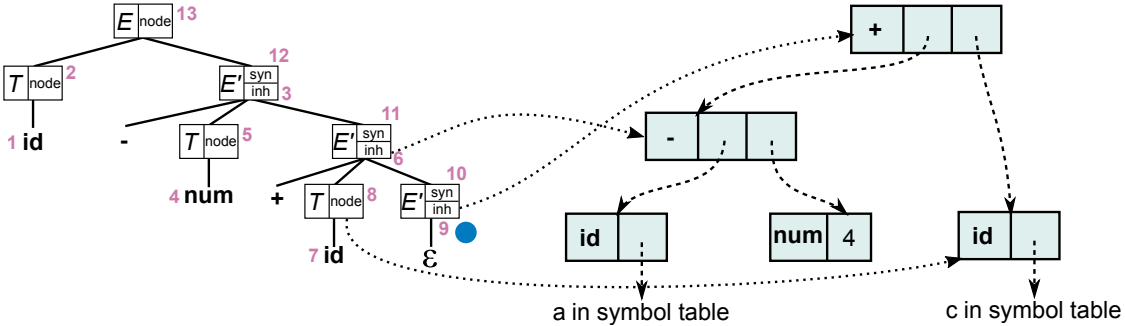
E	::=	⟨T⟩ ⟨E'⟩	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ ⟨T⟩ ⟨E'⟩	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- ⟨T⟩ ⟨E'⟩	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** ⟨E⟩ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: `a - 4 + c`



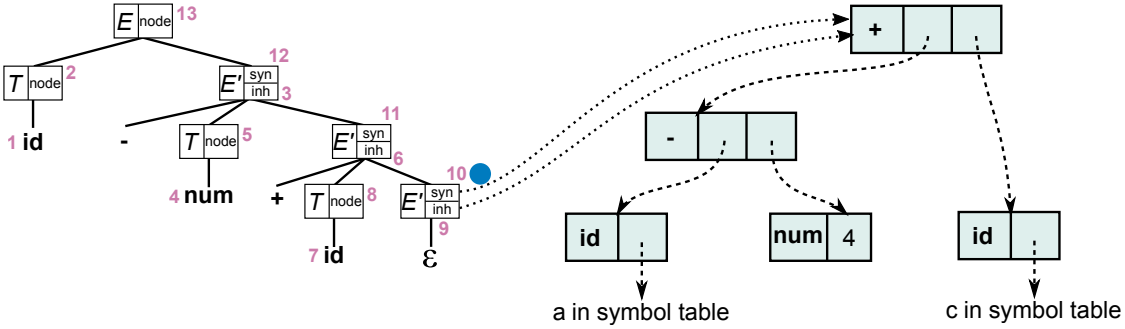
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a - 4 + c



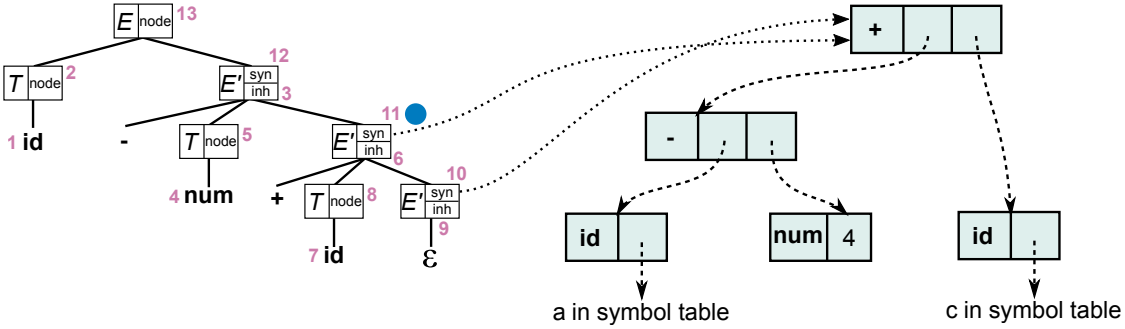
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node
			head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a − 4 + c



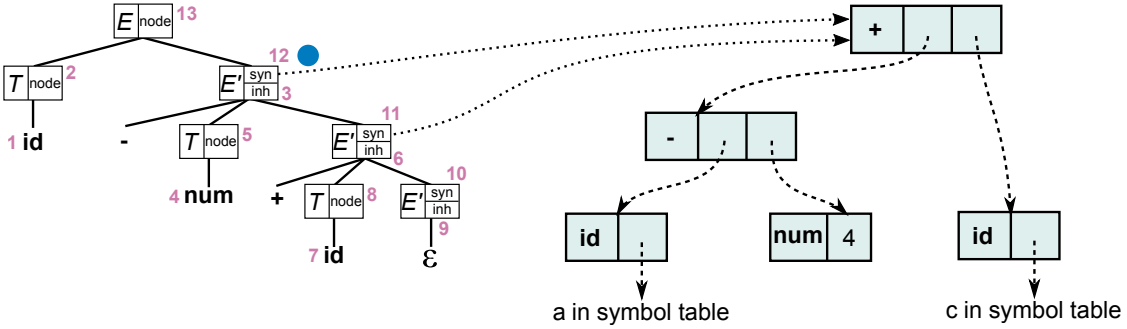
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a - 4 + c



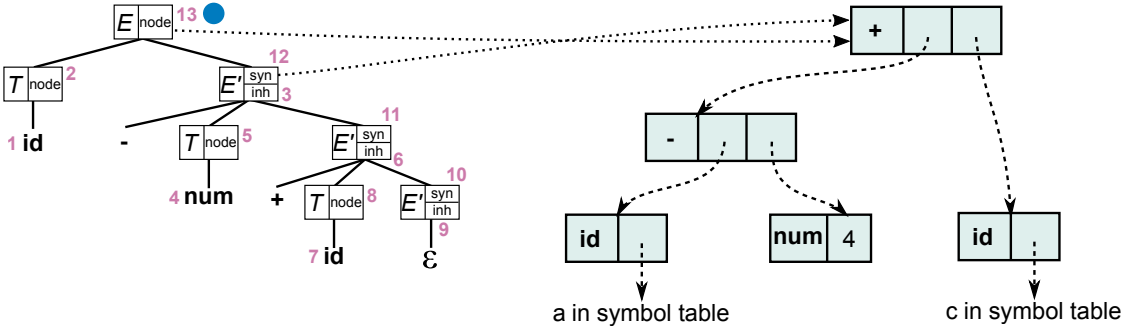
E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node
			head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node)
			head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a − 4 + c



E	::=	$\langle T \rangle \langle E' \rangle$	E'.inherited = T.node head.node = E'.synthesized
E'	::=	+ $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("+", head.inherited, T.node) head.synthesized = E'.synthesized
	|	- $\langle T \rangle \langle E' \rangle$	E'.inherited = **new** Node("-", head.inherited, T.node) head.synthesized = E'.synthesized
	|	ϵ	head.synthesized = head.inherited
T	::=	**(** $\langle E \rangle$ **)**	head.node = E.node
	|	**id**	head.node = **new** Leaf("**id**", **id**.lexeme)
	|	**num**	head.node = **new** Leaf("**num**", **num**.lexeme)

Input: a - 4 + c



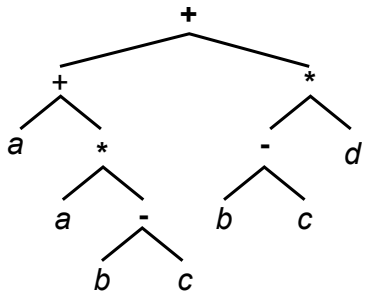
- Nodes in a syntax tree represent language constructs in the source program
- Children of a node represent the meaningful components of a construct

Directed Acyclic Graph (DAG)

DAG represents the language constructs in the source program
Ensures that a construct is present only one time in the DAG

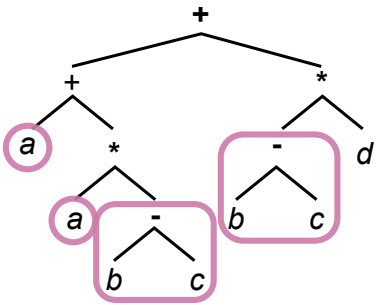
- The difference between syntax tree and DAG is that the DAG node may have more than one parent
- Consequently, a subexpression is repeated in a tree; and shared in a DAG

$$a + a * (b - c) + (b - c) * d$$



Syntax Tree

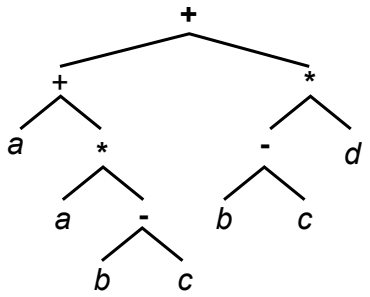
$$a + a * (b - c) + (b - c) * d$$



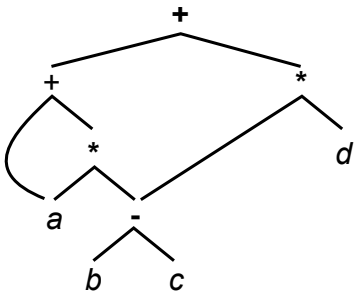
Syntax Tree

Same subexpressions
are duplicated in the tree

$$a + a * (b - c) + (b - c) * d$$

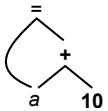


Syntax Tree



DAG

Often, the nodes of a DAG are stored in an array of records (also true for a syntax tree)



1	**id**	to symbol *a*	
2	**num**	10	
3	+	1	2
4	=	1	3

```
struct {
  int token_id;
  union {
    unsigned int symbol_index;
    double fvalue;
    long lvalue;
    struct {
      unsigned int left;
      unsigned int right;
    } operands;
  } attr;
} Record;
```

- Each node of the DAG is referred by its index in the table; its **value number**
- Let the signature of an interior node be the triple $\langle op, l, r \rangle$, where op is the label, l its left child's value number, and r its right child's value number. l and r are set to 0 when there is no child

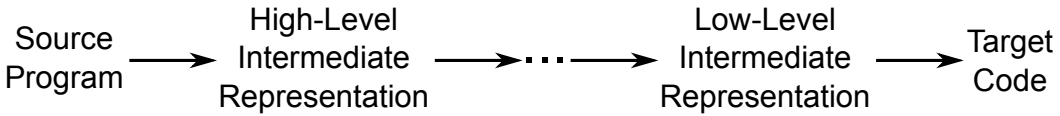
Inputs : Label op, node l, and node r, DAG D
Output : The value number of a node in the array with signature $\langle op, l, r \rangle$

```
begin
    M ← 0 ;
    for i ← 1 to |D| do
        c ← D_i ;
        if c = ⟨op, l, r⟩ then
            M ← i
        end
    end
    if M ≠ 0 then
        return M
    else
        i ← |D| ;
        D ← D ∪ ⟨op, l, r⟩ ;
        return i
    end
end
```

Inefficient approach for searching
Use an hash table in place of the array for D

High-level representation is close to the source language, e.g., syntax tree

Low-level representation are close to the target machine



The **three-address code** is a form of low-level intermediate representation that is close to the assembler languages

Rest of this lecture focuses on the generation of code with three-address code.
Syntax tree may also be used as a basis of the generation

[Strong et al., 1958, Wirth, 1971, Johnson, 1979, Ritchie, 1979]

Three-Address Code (TAC)

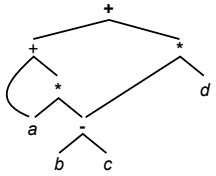
Sequence of three-address instructions

Three-Address Instruction

Has the form: `r = l op r`

- `op` is operation to apply
- `l` and `r` are the addresses of the operands if needed
- `r` is the address storing the result of the operation

Example



t_1 = b - c
t_2 = a * t_1
t_3 = a + t_2
t_4 = t_1 * d
t_5 = t_3 + t_4

where t_i is a name automatically generated

[Strong et al., 1958, Gosling, 1995]

Address can be one of:

Constant
Compiler must deal with many different types of constants and variables

Name
Source-program name. In implementation, it is replaced by a pointer to a symbol-table entry

Temporary Name
Created by the compiler.
Usefull for creating a distinct name each time a temporary is needed.
Usual syntax: t_i

Symbolic Label

Index of a three-address instruction in the sequence of instructions

- Numeric positions can be substituted for the labels, either by making a separate computing pass or by “backpatching”

```
L:t1 = i + 1
  i = t1
  t2 = i * 8
  t3 = a [ t2 ]
  if t3 < v then goto L
```

Symbolic Label

```
103:t1 = i + 1
104:i = t1
105:t2 = i * 8
106:t3 = a [ t2 ]
107:if t3 < v then goto 103
```

Numeric Position

Copy

`x = y`

The value of `y` is copied at the address of `x`

Assignment after Binary Operation

`x = y <op> z`

`<op>` is a binary arithmetic or logical operation, `y` and `z` are operands, and `x` receives the result

Assignment after Unary Operation

`x = <op> y`

`<op>` is an unary operation, e.g., unary minus, logical negation, conversion operators, and `y` is operand, and `x` receives the result

Unconditional Jump

`goto L`

The three-address instruction with label `L` is the next to be executed

Relational Condition Jump

`if x <relop> y goto L`

The three-address instruction with label `L` is the next to be executed if, and only if, the relational operator `<relop>`, e.g., $<$, $<=$, $>$..., applied to `x` and `y` is evaluated to true

Otherwise, the instruction following the conditional jump instruction in sequence is executed next

Boolean Condition Jump

`if x goto L`, or: `ifFalse x goto L`

Equivalent to: `if x = true goto L`, and: `if x = false goto L`, respectively

Procedure calls and returns are implemented using the following instructions:

- `param x`: for passing the value `x` as parameters
- `call p,n`: for the procedure call
- `y = call p,n`: for the function call
- `return y`: for returning a value

where `x` and `y` are addresses, `p` is the name of the subroutine, `n` is the number of parameters to pass to the subroutine.

```
param x1
param x2
...
param xn
call p,n
```

Procedures and their implementation are detailed in Chapter 5

Reading

`x = y[i]`

Copy the value of the i^{th} memory unit beyond `y` at the address of `x`

Writing

`x[i] = y`

Copy the value `y` at the i^{th} memory unit beyond `x`

Refencing of a value

`x = &y`

Copy the location of `y` in memory at the address of `x`

Derefencing of an address

`x = *y`

Copy the value at the address `y` in memory at the address of `x`

Indirect Copy

`*x = y`

Copy the value of `y` at the address stored into `x`

```
byte v;
int i;
byte[] a;

do {
   i = i + 1;
} while (a[i] < v)
```

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v then goto L
```

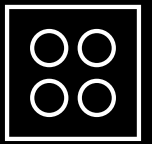
Available operators is an important issue in the design of an intermediate form

Operator set must be rich enough to implement the operations from the source language

Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine

If long sequences of instructions for source-language must be generated, then the optimizer and code generator may have to work harder to generate good code

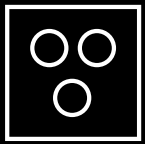
Three-address instructions could be implemented in a compiler as objects or as records following one of:



Quadruple

Representation with four fields

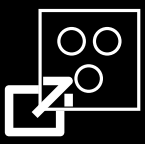
01



Triple

Representation with three fields

02



Indirect Triple

References to triples

03

Quadruple Form for Three-Address Instruction

A quadruple has four fields:

$$\texttt{<op>}\ \texttt{<arg}_1\texttt{>}\ \texttt{<arg}_2\texttt{>}\ \texttt{<result>}$$

- `<op>`: this field contains an internal code for the operator
- $\texttt{<arg}_1\texttt{>}$ and $\texttt{<arg}_2\texttt{>}$: they are the arguments of the operator
- `<result>`: it contains the result value computed by the operator

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

op	arg_1	arg_2	result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

```c
/* Operations supported by the three-address code */
typedef enum { MULTIPLY, ADD, MINUS, ...} Operator;

/* Definition of a parameter or a return value */
typedef union {
   unsigned long address; /* address of a variable */
   long integer_value;    /* integer constant */
   double float_value;    /* floating-point constant */
} Value;

/* Definition of a single quadruple */
typedef struct {
   Operator operator;
   Value arg1;
   Value arg2;
   Value result;
} Quadruple;

/* Instruction sequence */
Quadruple[] code;
```

Result in quadruples is primarily used for temporary names

Triple Form for Three-Address Instruction

A triple has only three fields:

$$\texttt{<op>}\ \texttt{<arg}_1\texttt{>}\ \texttt{<arg}_2\texttt{>}$$

Result of an operation `x <op> y` is referred by its position, e.g., `(0)`, rather than by an explicit temporary name

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

```c
/* Operations supported by the three-address code */
typedef enum { MULTIPLY, ADD, MINUS, ...} Operator;

/* Definition of a parameter or a return value */
typedef union {
   unsigned long address; /* address of a variable */
   long integer_value;    /* integer constant */
   double float_value;    /* floating-point constant */
} Value;

/* Definition of a single triple */
typedef struct {
   Operator operator;
   Value arg1;
   Value arg2;
} Triple;

/* Instruction sequence */
Triple[] code;
```

QUADRUPLE

Relevant for optimizing compiler, where instructions are often moved around
When moving an instruction, then instructions that use the result require no change

TRIPLE

Result of an operation is referred by its position, so moving may require to change all references to that result

This last problem does not occur with indirect triples

Indirect triples consist of a listing of references to triples, rather than a listing of triples themselves

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Optimizing compiler can reorder the instruction list, without affecting the triples themselves

In Java, an array of instructions is similar to an indirect triple representation, since Java treats the array elements as references to objects

Static Single-Assignment form (SSA)

Evolution of three-address form.
Intermediate representation that facilitates certain code optimizations

Two differences between SSA and the standard form of the three-address code:

1. All assignments in SSA are to variables with distinct names
2. Introduction of the ϕ-function

All assignments in SSA are to variables with distinct names

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

Standard Form

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

SSA Form

ϕ-Function

Notation convention to combine two definitions of the same variable in parallel control-flow paths

Example

if *flag* **then** x = -1;
else x = 1;
y = x * a

if *flag* **then** x_1 = -1;
else x_2 = 1;
y = $\phi(x_1, x_2)$ * a

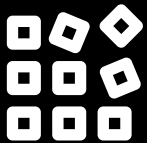
Impossible to determine which x value is used for x * a
$\Rightarrow$ harder to optimize the target code

ϕ-function replies the "defined" value in its arguments

Application of types can be grouped as follows:

Type Checking

Ensures operand types match expected ones



Translation Application

Storage size and location at run-time
Implicit type conversion...

Types have structure represented by the type expressions

Type Expression

Type expression is one of:

1. Basic type: **boolean**, **char**, **integer**, **float**, **void**
2. Type name
3. Expression built with the array type constructor
4. Record: data structure with named fields
5. Function prototype: by using the function prototype constructor $inputType \rightarrow outputType$
6. Cartesian product of two type expressions: if s and t are type expressions, then $s \times t$ is a type expression

We must define how to convert a value from one type to others
Many type-checking rules have the form:
"if two type expressions are equivalent then return a certain type else error"

Type Equivalence

Two types are structurally equivalent when:

1. They are of the same basic type
2. They are formed by applying the same constructor to structurally equivalent types
3. One is a type name that denotes the other

Points 1 and 2 are used to defined the equivalence between two type names, i.e., the name equivalence

- Declaration of types is handled by a grammar like:

$$
\begin{aligned}
\langle \text{decls} \rangle &::= \langle \text{type} \rangle \ \textbf{id} \ \textbf{;} \ \langle \text{decls} \rangle \\
&::= \epsilon \\
\langle \text{type} \rangle &::= \langle \text{base\_type} \rangle \ \langle \text{array\_decls} \rangle \\
&::= \textbf{record} \ \{ \ \langle \text{decls} \rangle \ \} \\
\langle \text{base\_type} \rangle &::= \textbf{int} \ \mid \ \textbf{float} \\
\langle \text{array\_decls} \rangle &::= \textbf{[num]} \ \langle \text{array\_decls} \rangle \\
&::= \epsilon
\end{aligned}
$$

- This grammar supports basic types, arrays and records:
  - **float** name0
  - **int** [3][4] name0
  - **record** { **float** name1; **record** { **int** name2; } } name0

From the type of a name, amount of storage needed at run-time could be determined at compile time

Size Determination

- Width of a type (and not of a variable) is the number of storage units (usually bytes) needed for values of that type
- Basic type requires an integral number of storage units
- Data of varying length (string, dynamic array...) is handled by reserving a known fixed amount of storage units for a pointer to the data (usually 64 or 128 bits)

Address (Location) Determination for Run-time

- Relative address of each name could be determined based on their size
- For easy access, aggregated data (array, class...) is allocated in contiguous block

Both type (size) and relative address are saved in the symbol table

Storage layout for data objects is strongly influenced by the addressing constraints of the target machine

Examples

- Instructions to add integers may expect integers to be aligned, i.e., placed at certain positions in memory such as an address divisible by 4
- Array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes (the next multiple of 4)

Padding

Space left unused due to alignment considerations

A compiler may generate instructions for limiting padding

SDD below computes types and their widths for basic and array types
(records will be discussed later)

Type	::=	⟨Base⟩	t = Base.type w = Base.width
		⟨Arrays⟩	head.type = Arrays.type head.width = Arrays.width
Base	::=	**int**	Base.type = integer Base.width = 4
	|	**float**	Base.type = float Base.width = 8
Arrays	::=	**[num]** ⟨Arrays⟩	head.type = **array** (num.value, Arrays.type) head.width = num.value * Arrays.width
	|	ϵ	head.type = t head.width = w

Modern languages allow all the declarations in a single procedure to be processed as a group

Declarations may be distributed in a procedure, e.g., in Java, but they can still be processed when the procedure is analyzed

Variable named "`offset`" to keep track of the next available relative address

Program	::=		offset = 0
		⟨Decls⟩	
Decls	::=	⟨Type⟩ **id**	s = **new** Symbol(**id**.lexeme) s.offset = offset s.type = Type.type SymbolTable.current.declare(**id**.lexeme, s) offset = offset + Type.width
		⟨Decls⟩	
	|	ϵ	

Semantic action of ⟨Decls⟩ creates a symbol table's entry

Symbol table takes:
- name of the variable (its lexeme)
- type of the variable (implicit size)
- storage position of the variable

Extension of the previous grammar with T-production

Type	::=	⟨Base⟩	t = Base.type; w = Base.width
		⟨Arrays⟩	head.type = Arrays.type; head.width = Arrays.width
	|	**record** { ⟨Types⟩ }	head.type = record; head.width = Types.width
Types	::=	⟨Type⟩⟨Types⟩	head.width = Type.width + Types.width
	|	ϵ	head.width = 0
Base	::=	**int**	head.type = integer; head.width = 4
	|	**float**	head.type = float; head.width = 8
Arrays	::=	**[num]** ⟨Arrays⟩	head.type = **array** (num.value,C.type) head.width = num.value * C.width
	|	ϵ	head.type = t; head.width = w

Field names in a record must be distinct

Offset or relative address for a field name is relative to the data area for that record

For convenience, record is defined with a specific symbol table, or environment

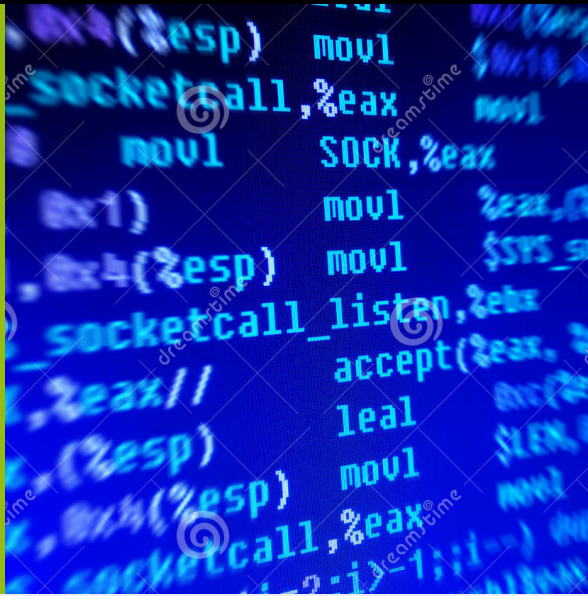
Type	::=	**record** {	SymbolTable.current.offset = offset SymbolTable.current = SymbolTable.openContext() offset = 0
		⟨Types⟩ }	head.type = **record** (SymbolTable.current) head.width = offset SymbolTable.current = SymbolTable.closeContext() offset = SymbolTable.current.offset

Classes are stored as records, since no storage is reserved for methods

How to translate source expressions to three-address code?

Translation function may be placed in two locations:

1. inside the semantic actions themselves
2. inside a dedicated method, usually called `generate()`, of the syntax tree



Each operation in the source expression are translated to its equivalent three-address code, e.g., assignment and arithmetic operators

S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Each operation in the source expression are translated to its equivalent three-address code, e.g., assignment and arithmetic operators

S	::=	**id** = ⟨E⟩ **;**	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

①: | is the operator for string concatenation

②: **quadruple** creates a quadruple form of three-address code

③: Attribute `code` represents the generated three-address code for each nonterminal

Each operation in the source expression are translated to its equivalent three-address code, e.g., assignment and arithmetic operators

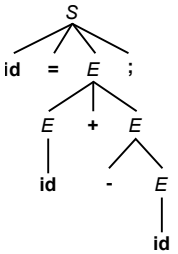
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

④: `TemporaryVariable` creates a temporary variable with specific index

⑤: Attribute address is address of the expression symbol's value

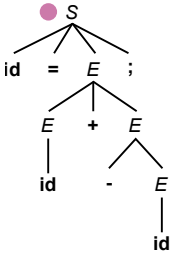
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



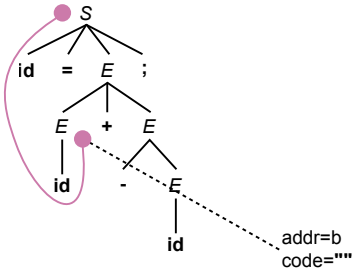
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



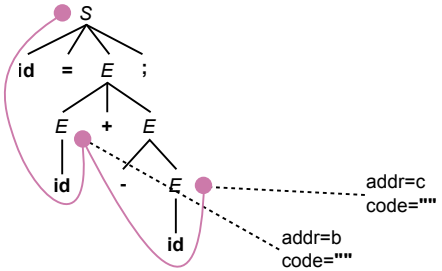
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



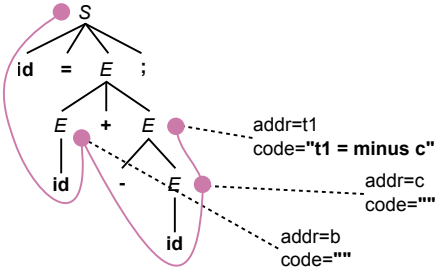
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



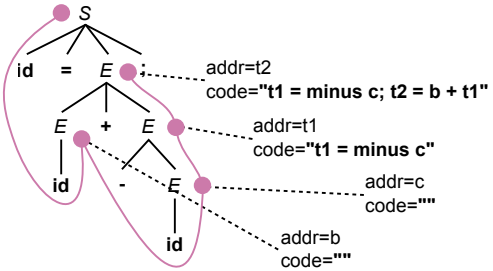
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



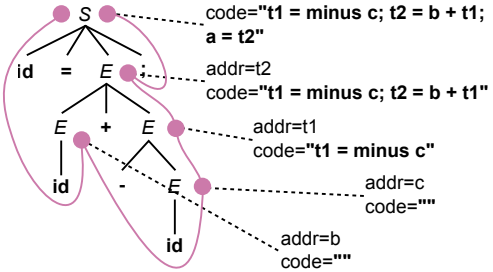
S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



S	::=	**id** = ⟨E⟩ ;	head.code = E.code | **quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E_1.code | E_2.code | **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() head.code = E.code | **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr; head.code = E.code
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme) head.code = ""

Input: `a = b + - c`



Code attributes can be long string, so they are usually generated incrementally

- Instead of building up E.code as previously, we can modify **quadruple** to output the new three-address instructions in a external data structure
- The code attribute is removed

S	::=	**id** = ⟨E⟩ ;	**quadruple** ("=", E.addr, ∅, SymbolTable.current.get(**id**.lexeme))
E	::=	⟨E⟩ + ⟨E⟩	head.addr = **new** TemporaryVariable() **quadruple** ("+", E_1.addr, E_2.addr, head.addr)
E	::=	- ⟨E⟩	head.addr = **new** TemporaryVariable() **quadruple** ("minus", E.addr, ∅, head.addr)
	|	**(** ⟨E⟩ **)**	head.addr = E.addr
	|	**id**	head.addr = SymbolTable.current.get(**id**.lexeme)

Array elements can be accessed quickly if they are stored in a block of consecutive locations

Position of the element at index i in 1-dimension array (zero-base indexing)

$$base + w \times i$$

where w is the width of each array element, $base$ is the relative address of the storage allocated for the array

Position of the element at index $(i_0, \ldots, i_{n-1})$ in n-dimension array (zero-base indexing, row major)

$$base + w \times \left[\sum_{j \in [0..n)} \left(\left(\prod_{d \in [j..n)} s_d \right) \times i_j \right) + i_{n-1} \right]$$

where s_d is the number of cells for dimension d.

The major problem in generating code for array references is to relate the address-calculation formulas to a grammar for array references

- Let the nonterminal L generates an array name followed by a sequence of index expressions

$$\langle \mathrm{L} \rangle \rightarrow \langle \mathrm{L} \rangle \ \mathbf{[}\ \langle \mathrm{E} \rangle\ \mathbf{]} \quad | \quad \mathbf{id}\ \mathbf{[}\ \langle \mathrm{E} \rangle\ \mathbf{]}$$

- Assume all arrays are zero-based indexing

$L \quad ::= \quad$ **id [** $\langle E \rangle$ **]**	head.base = SymbolTable.current.get(**id**.lexeme) head.type = head.base.elementType head.addr = **new** TemporaryVariable() **quadruple** ("*", E.addr, head.type.width, head.addr)

- Attribute "base": the symbol of the array
- Attribute "type": the type of the elements of the array (given by the symbol table entry)
- Attribute "addr": the address of the element in the storage from the beginning of the array

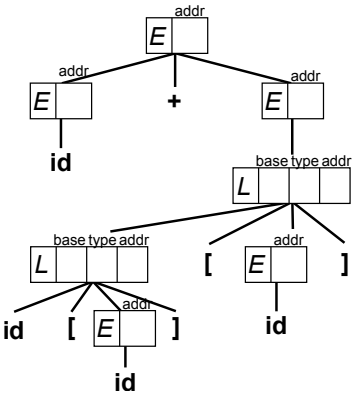
$L \quad ::= \quad \langle \mathrm{L} \rangle \; \mathbf{[} \; \langle \mathrm{E} \rangle \; \mathbf{]}$	head.base = L.base head.type = L.type.elementType t = **new** TemporaryVariable() head.addr = **new** TemporaryVariable() **quadruple** ("*", E.addr, head.type.width, t) **quadruple** ("+", L.addr, t, head.addr)

- Attribute "base": the symbol of the array
- Attribute "type": the type of the elements of the array (given by the symbol table entry)
- Attribute "addr": the address of the element in the storage from the beginning of the array

$S ::= \langle L \rangle = \langle E \rangle ;$		**quadruple** ("[]=", L.addr, E.addr, L.base)
$E ::= \langle L \rangle$		head.addr = **new** TemporaryVariable() **quadruple** ("=[]", L.base, L.addr, head.addr)

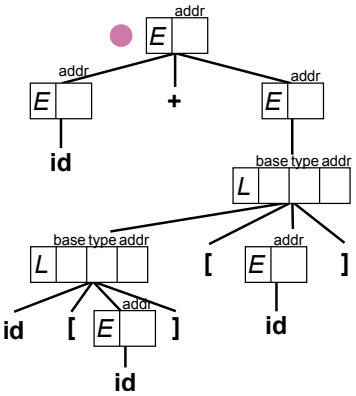
- Attribute "base": the symbol of the array
- Attribute "addr": the address of the element in the storage from the beginning of the array; or the address of a temporary variable

Input: `c + a[i][j]`



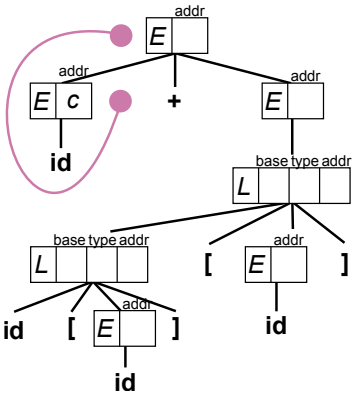
Output:

Input: `c + a[i][j]`



Output:

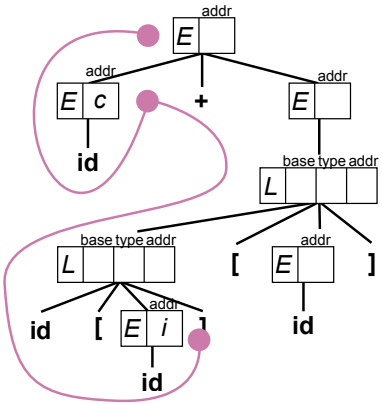
Input: `c + a[i][j]`



Output:

E	::=	**id**	head.addr = SymbolTable.current.get(**id**.lexeme)

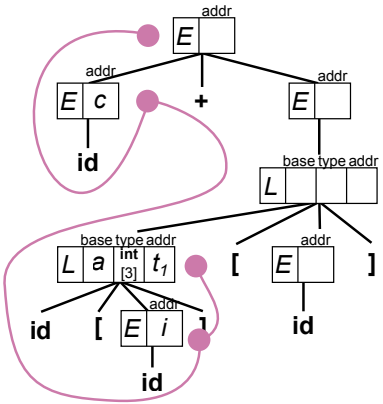
Input: `c + a[i][j]`



Output:

E	::=	**id**	head.addr = SymbolTable.current.get(**id**.lexeme)

Input: `c + a[i][j]`



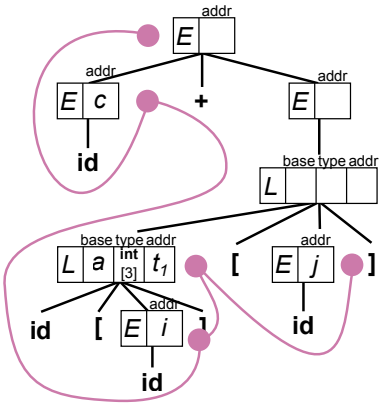
Output:

$t_1 = i * 12$

Assume that:
a) *a* was declared as **int**[2][3]
b) An integer takes 4 bytes

L	::=	**id [** ⟨E⟩ **]**	head.base = SymbolTable.current.get(**id**.lexeme) head.type = head.base.elementType head.addr = **new** TemporaryVariable() **quadruple** ("*", E.addr, head.type.width, head.addr)

Input: `c + a[i][j]`

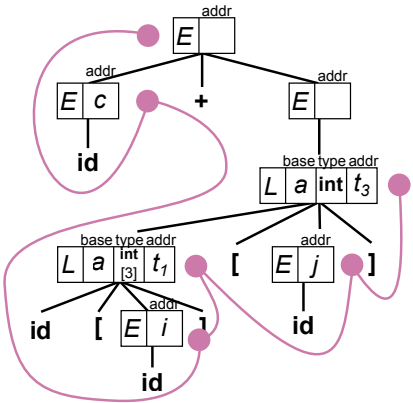


Output:

$t_1 = i * 12$

E	::=	**id**	head.addr = SymbolTable.current.get(**id**.lexeme)

Input: `c + a[i][j]`

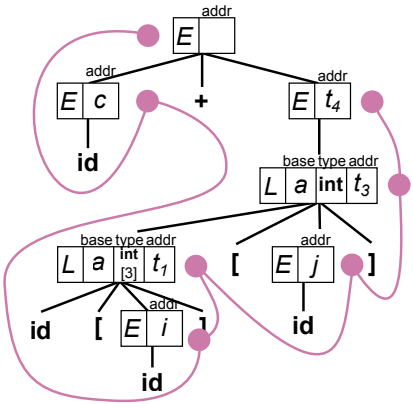


Output:

$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$

L	::=	⟨L⟩ **[** ⟨E⟩ **]**	head.base = L.base head.type = L.type.elementType t = **new** TemporaryVariable() head.addr = **new** TemporaryVariable() **quadruple** ("*", E.addr, head.type.width, t) **quadruple** ("+", L.addr, t, head.addr)

Input: `c + a[i][j]`

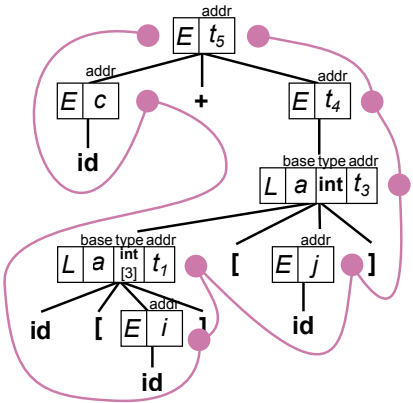


Output:

$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$
$t_4 = a\,[\,t_3\,]$

E	::=	$\langle \mathrm{L} \rangle$	head.addr = **new** TemporaryVariable() **quadruple** ("=[]", L.base, L.addr, head.addr)

Input: `c + a[i][j]`



Output:

$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$
$t_4 = a\,[\,t_3\,]$
$t_5 = c + t_4$

E	::=	$\langle E \rangle + \langle E \rangle$	head.addr = **new** TemporaryVariable(); **quadruple** ("+", E_1.addr, E_2.addr, head.addr)

Compiler determines if the types are consistent according to a collection of logical rules that is called the type system

Assign a type expression to each component of the source program

Verify type compliance and catch errors

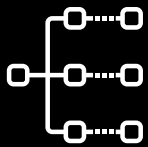
Synthesis Type Checking

Inference Type Checking

[Pierce, 2002, Milner, 1978]

Dynamic Type Checking

Run-time
Target code carries types



Static Type Checking

Compile time
Eliminates the need for dynamic checking

[Pierce, 2002, Milner, 1978]

Type Synthesis

Builds up the type of an expression from the types of its subexpressions

- It requires names and their types to be declared before they are used
- Relationship between types must be defined, e.g., $T_1 \subset T_2$ for value set of T_1 are included in value set of T_2

Example

- Type of $E_1 + E_2$ is defined according to the types of E_1 and E_2
- If E_1 is **int** and E_2 is **float** then E_1+E_2 is **float** (**int** $\subset$ **float**)

Type Inference

Determines the type of a language construct from the way it is used

- Type inference is needed for languages like ML or Python, which check types, but do not require names to be declared

Example

- Let the PHP code: { `$a = 14; $b = "a" . $a; $c = 1 + $a;` }
- `$a` is used as a string for `$b`'s expression
- `$a` is used as an integer for `$c`'s expression

Consider:

Expression, e.g. `f + i`
Where `f` is a **float** and `i` is an **integer**

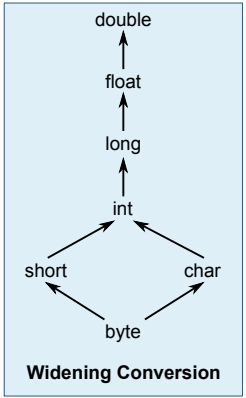
- Representations in computer memory of floating-point numbers and integers are different
- Different machine instructions are used for operations on integers and floats
- Compiler may need to convert one of the operands to ensure that both operands are of the same type when the operator is applied:

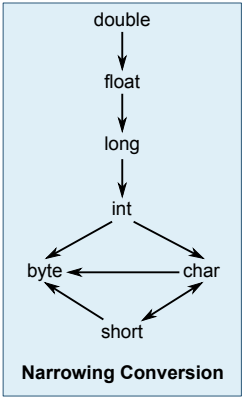
```
t1 = (float) 2
t2 = t1 * 3.14
```

Type conversion rules vary from language to language

- Widening conversion: preserve information between the value before the conversion and the value after the conversion
- Narrowing conversion: can lose information
- $a \rightarrow b$ means: value of type a could be converted to value of type b

Java Conventions





Implicit Conversion, or Coercion

Automatically done by the compiler, with a possible warning message in the case of narrowing conversion

- Many languages limit the implicit conversions to widening conversions

Explicit Conversion, or Cast

Conversions written in the source code by the programmer

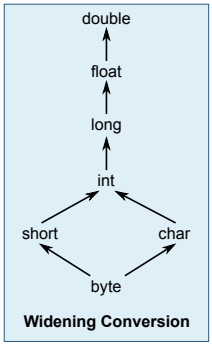
Definition

$maxType : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$

$(t_1, t_2) \mapsto$ maximum or least upper bounds of t_1 and t_2 in the widening hierarchy; Otherwise error

Example

`maxType(short, char)` $\rightarrow$ **int**



Definition

$$widenVar : \mathbb{A} \times \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{A}$$

$(a, t_{out}, t_{in}) \mapsto$ Generates the code that widens the value pointed by a to the type t_{out}

Assuming that a is of type t_{in}

Conversion is done only if it is required

Returns the address were the result of the is available.

Assume a language with only the two types **int** and **float**.

Function widenVar(a : $\mathbb{A}$, t_{out} : $\mathbb{T}$, t_{in} : $\mathbb{T}$) : $\mathbb{A}$
begin
  if $t_{out} = t_{in}$ **then**
    return a;
  else if t_{in} = **int** *and* t_{out} = **float then**
    t ←**new** TemporaryVariable();
    quadruple ("(float)", a, ∅, t);
    return t
  else
    `Throw("Cannot widen the variable")`
  end
end

Definition

$narrowVar : \mathbb{A} \times \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{A}$

$(a, t_{out}, t_{in}) \mapsto$ Generates the code that narrows the value pointed by a to the type t_{out}

Assuming that a is of type t_{in}

Conversion is done only if it is required

Returns the address were the result of the is available

Assume a language with only the two types **int** and **float**.

Function narrowVar(a : $\mathbb{A}$, t_{out} : $\mathbb{T}$, t_{in} : $\mathbb{T}$) : $\mathbb{A}$
begin
  if $t_{out} \neq$ **int** *or* $t_{in} \neq$ **float then**
    Throw(*"Cannot narrow the variable"*)
  else if $t_{out} =$ **int** *and* $t_{in} =$ **float then**
    t ←**new** TemporaryVariable();
    quadruple ("(int)", a, $\emptyset$, t);
    return t
  else
    return a
  end
end

Attribute "type " is added to store the type of an expression
SDD is updated to check the types:

$E \quad ::= \quad \langle \mathrm{E} \rangle + \langle \mathrm{E} \rangle$	head.type = maxType(E_1.type, E_2.type) o1 = widenVar(E_1.addr, E_1.type, head.type) o2 = widenVar(E_2.addr, E_2.type, head.type) head.addr = **new** TemporaryVariable() **quadruple** ("+", o1, o2, head.addr)

$E \quad ::= \quad \textbf{id} = \langle \mathrm{E} \rangle$	head.addr = SymbolTable.current.get(**id**.lexeme)
	head.type = v.type
	w = narrowVar(E.addr, E.type, head.type)
	if (w $\neq$ E.addr)
	  warning "May loose information"
	else
	  w = widenVar(E.addr, E.type, head.type)
	quadruple ("=", w, $\emptyset$, head.addr)

Definition

Allow the creation of several functions with the same name, which differ from each other in the type of the input(s) and the output(s) of the function

- Depending on the context, the overloading may be for a function, a procedure, a method, or an operator
- **Symbol table must contains all the signatures of the functions (in the context, which is using the symbol table)**
- A signature consists of:
  1. the function name
  2. the list of the types of the formal parameters of the function
  3. the return type (optional)

Definition

The term “polymorphic” refers to any code fragment that can be executed with arguments of different types

Only parametric polymorphism is considered in this section
where the polymorphism is characterized by parameters or type variables

- Consider the following definition in ML language:

```
fun length(x) = if null(x) then 0 else length( tl(x) ) + 1;
```

- Consider the following statement in ML language:

```
length( ["sun", "mon", "tue"] ) + length( [10, 9, 8, 7] )
```

- The same function length() is invoked on an array of strings and on an array of integers.
- The result of the ML statement is: $3 + 4 = 7$

- Using the symbol $\forall$ and the type constructor **list**, the type/signature of the function `length` is:

$$\forall a.\textbf{list}(a) \rightarrow \textbf{int}$$

- $\forall$ symbol is the universal quantifier, and the type variable to which it is applied is said to be bound by it
- Type expression with a $\forall$ symbol is referred as a "polymorphic type"

- Each time a polymorphic function is applied, its bound type variables (a...) can denote a different type

How to determine the types in the signature of a polymorphic function?

We must infer the types by exploring the syntax tree of the function and applying the substitution and unification operations

Substitution

Mapping from type variables to type expressions

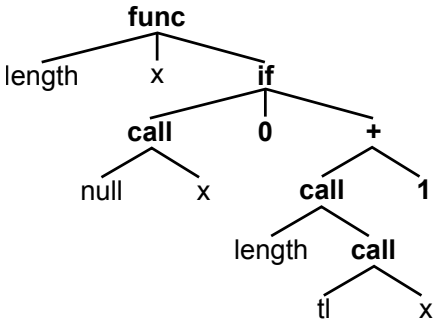
Example: **list** (**int**) is an instance of **list** (α), since it is the result of substituting **int** for α in **list** (α)

Unification

Determine whether type variables s and t are structurally equivalent by substituting the type variables in s and t by type expressions

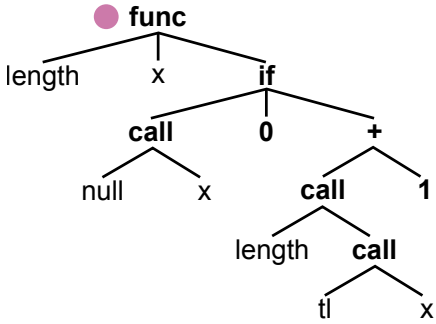
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification



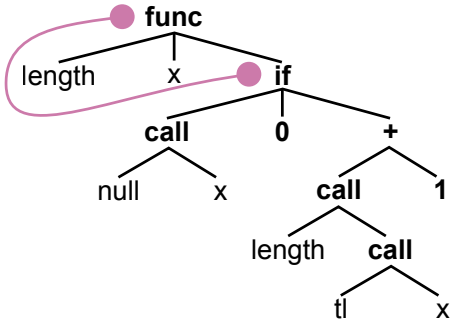
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \rightarrow \gamma$	
x	β	



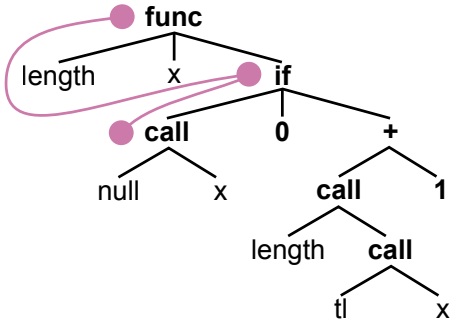
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \to \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \to \alpha$	$\alpha = \gamma$



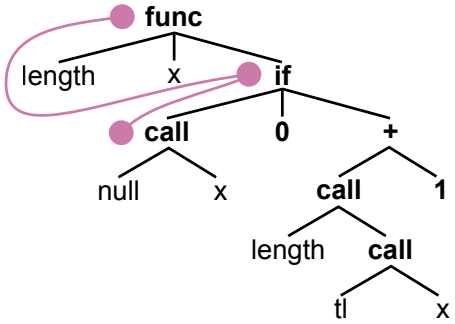
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \rightarrow \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \rightarrow \alpha$	$\alpha = \gamma$
null	**list**$(\omega_n) \rightarrow \mathbb{B}$	



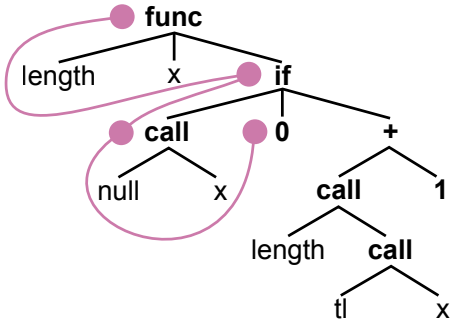
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \to \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \to \alpha$	$\alpha = \gamma$
null	$\textbf{list}(\omega_n) \to \mathbb{B}$	
null(x)	$\mathbb{B}$	$\textbf{list}(\omega_n) = \beta$



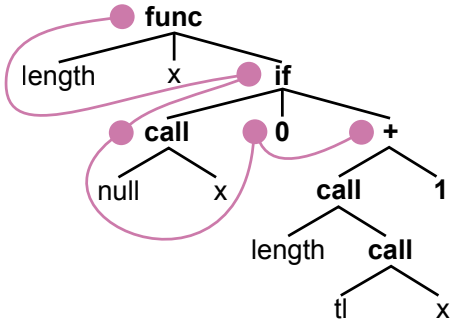
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \to \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \to \alpha$	$\alpha = \gamma$
null	**list**$(\omega_n) \to \mathbb{B}$	
null(x)	$\mathbb{B}$	**list**$(\omega_n) = \beta$
0	**int**	$\alpha =$ **int**



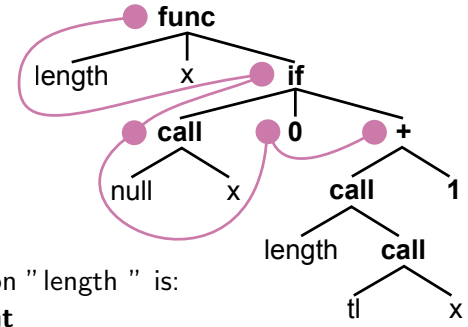
```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \to \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \to \alpha$	$\alpha = \gamma$
null	**list**$(\omega_n) \to \mathbb{B}$	
null(x)	$\mathbb{B}$	**list**$(\omega_n) = \beta$
0	**int**	$\alpha =$ **int**
+	$\phi \times \phi \to \phi$	$\phi = \alpha$



```
fun length(x) =
  if null(x) then 0
  else length( tl(x) ) + 1;
```

Expression	Type	Unification
length	$\beta \rightarrow \gamma$	
x	β	
if	$\mathbb{B} \times \alpha \times \alpha \rightarrow \alpha$	$\alpha = \gamma$
null	**list**$(\omega_n) \rightarrow \mathbb{B}$	
null(x)	$\mathbb{B}$	**list**$(\omega_n) = \beta$
0	**int**	$\alpha =$ **int**
+	$\phi \times \phi \rightarrow \phi$	$\phi = \alpha$



The type of the function " length " is:
length : **list** $(\omega_n) \rightarrow$ **int**

Translation of statements (if-else-statements, while-statements...) needs translation of boolean expressions

Boolean expressions are used for:

- Altering the flow of control
- Computing logical values

- To support this distinction, we may:
  1. Use two different nonterminals
  2. Use inherited attributes
  3. Use a set of flags during the parsing
  4. Build a syntax tree and invoke different procedures for the two different uses

Definition

Boolean operators are translated into jumps
These operators themselves do not appear in the three-address code

Value of a boolean expression is represented by a position in the code sequence

Example

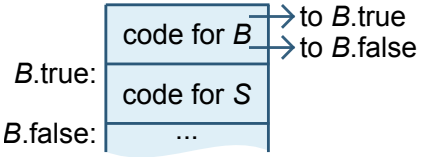
```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
    if x < 100 then goto L1
    ifFalse x > 200 then goto L2
    ifFalse x ≠ y then goto L2
L1: x = 0
L2: ...
```

- Consider the grammar:

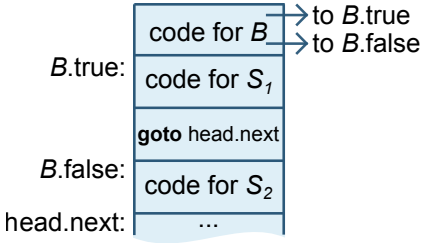
$$
\begin{aligned}
\langle \mathrm{S} \rangle \quad &::= \quad \textbf{if (} \langle \mathrm{B} \rangle \textbf{)} \langle \mathrm{S} \rangle \\
&::= \quad \textbf{if (} \langle \mathrm{B} \rangle \textbf{)} \langle \mathrm{S} \rangle \textbf{ else } \langle \mathrm{S} \rangle \\
&::= \quad \textbf{while (} \langle \mathrm{B} \rangle \textbf{)} \langle \mathrm{S} \rangle
\end{aligned}
$$

- We introduce the attributes:
  - `B.code` and `S.code`: synthesized attributes; three-address code of the nonterminals
  - `B.true`: inherited attribute; the label of the code associated to the then-statements
  - `B.false`: inherited attribute; the label of the code associated to the else-statements
  - `B.next`: inherited attribute; the label of the code just after the current if-then-else statements

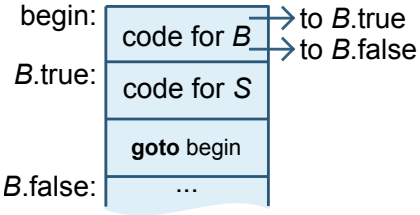


S	::=	**if (**	B.true = **newlabel** ()
		⟨B⟩ **)**	B.false = head.next
			S.next = head.next
		⟨S⟩	**label** (B.true)

- Nonterminal for the condition is no more E (nonterminal for expressions), but B (specific nonterminal for boolean expressions in control flow)
- `newlabel()` creates a new label each time it is called
- `label(`α`)` attaches label α to the next three-address instruction to be generated



S	::=	**if (**	B.true = **newlabel** ()
			B.false = **newlabel** ()
		⟨B⟩ **)**	S_1.next = head.next
			label (B.true)
		⟨S⟩ **else**	**quadruple** ("goto", head.next, ∅, ∅)
			S_2.next = head.next
			label (B.false)
		⟨S⟩	

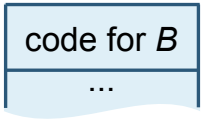


S	::=	**while (**	begin = **newlabel** ()
			B.true = **newlabel** ()
			B.false = head.next
		⟨B⟩ **)**	S.next = begin
			label (B.true)
		⟨S⟩	**quadruple** ("goto", begin, ∅, ∅)

Boolean expressions for control flow need dedicated semantic rules

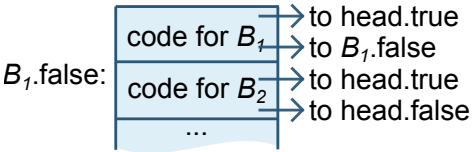
Boolean expressions used in control-flow statements must be translated into jumping three-address code

B	::=	**true**	**quadruple** ("goto", head.true, $\emptyset$, $\emptyset$)
B	::=	**false**	**quadruple** ("goto", head.false, $\emptyset$, $\emptyset$)



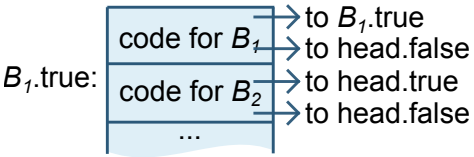
B ::= ! ⟨B⟩		B.true = head.false B.false = head.true

- No code is needed for an expression of the form `!B`
- Just interchange the `true` and `false` attributes of the head to set the `true` and `false` attributes of B



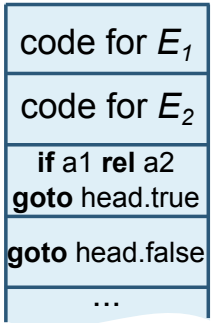
B ::= $\langle B \rangle$ || $\langle B \rangle$	B_1.true = head.true B_1.false = **newlabel** () B_2.true = head.true B_2.false = head.false

- If B_1 is true, the head is true
- If B_1 is false, evaluate B_2
- So `B1.false` is the label of the first instruction of `B2`
- The value of the head becomes the same as the value of B_2



B ::=	B_1.true = **newlabel** () B_1.false = head.false B_2.true = head.true B_2.false = head.false
$\langle B \rangle$	
&& $\langle B \rangle$	

- If B_1 is false, the head is false
- If B_1 is true, evaluate B_2
- So `B1.true` is the label of the first instruction of B_2
- The value of the head becomes the same as the value of B_2



B	$::=$ $\langle \mathrm{E} \rangle$ **rel** $\langle \mathrm{E} \rangle$	t = **new** TemporaryVariable() **quadruple** (**rel**.operator, E_1.addr, E_2.addr, t) **quadruple** ("if", t, head.true, $\emptyset$) **quadruple** ("goto", head.false, $\emptyset$, $\emptyset$)

- Form $a < b$ is translated to:

```
t = (a < b)
if t then goto B.true
goto B.false
```

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
    t1 = x < 100
    if t1 then goto L2
    goto L3
L3: t1 = x > 200
    if x > 200 then goto L4
    goto L1
L4: t1 = x ≠ y
    if t1 then goto L2
    goto L1
L2: x = 0
L1: ...
```

The semantic rules described in the previous slides
may generate more goto instructions than strictly necessary

Example

```
L3:t1 = x > 200
   if t1 then goto L4
   goto L1
L4:...
L1:...
```

Best Practice

```
L3:t1 = x > 200
   ifFalse t1 then goto L1
   ...
L1:...
```

Avoiding redundant gotos is done by introducing a constant for the value of the labels: `fall`
It means “don’t generate any jump” or “fall in the next available instruction”

We can adapt the semantic rules of the boolean expressions. $\langle S \rangle \rightarrow$ **if (** $\langle B \rangle$ **)** $\langle S \rangle$

S	::=	**if (** $\langle B \rangle$ **)** $\langle S \rangle$	B.true = **newlabel** () B.false = head.next S.next = head.next **label** (B.true)

S	::=	**if (** $\langle B \rangle$ **)** $\langle S \rangle$	B.true = **fall** B.false = head.next S.next = head.next

B ::= $\langle B \rangle$ || $\langle B \rangle$	B_1.true = head.true B_1.false = **newlabel** () B_2.true = head.true B_2.false = head.false

B ::= $\langle B \rangle$ || $\langle B \rangle$	**if** head.true = **fall**
	  B_1.true = newlabel()
	else
	  B_1.true = head.true
	B_1.false = **fall**
	B_2.true = head.true
	B_2.false = head.false
	if head.true = **fall**
	  label(B_1.true)

A key problem is the matching of a jump instruction
with the target address of the jump

Example

- Consider the statement **if (** $\langle \mathrm{B} \rangle$ **)** $\langle \mathrm{S} \rangle$
- In a one-pass translation, B must be translated before S is examined
- What is the address of the label that permits to go over the code for S?

Solution 1

- In the previous slides, we solve this problem by using inherited attribute "`next`"
- But a separate (additionnal) pass is then needed to bind labels to addresses

Solution 2

- Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass
- This approach is detailled in the following slides

When the jump target is after the current instruction

Address of the current instruction is added into a list

When the address of the target instruction is known

Instructions in the list are updated

New synthesized attributes in B:

- B.bptruelist: list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true.
- B.bpfalselist: list of instructions that eventually get the label to which control goes when B is false.

- makebplist(adr): creates a new list containing only *adr*, an index into the array of instructions
- mergebplists(lst1,lst2): concatenates the lists pointed by *lst*1 and *lst*2, and returns a pointer to the result
- backpatch(lst,adr): inserts *adr* as the target label for each of the instructions on the list pointed to by *lst*
- instadr(): replies the address of the instruction that will be generated by the **next** call to `quadruple ()`
- Unknown address: Keyword **?** represents an unkwown address

B	::=	**true**	head.bptruelist = makebplist(instadr()) **quadruple** ("goto", **?**, $\emptyset$, $\emptyset$)
B	::=	**false**	head.bpfalselist = makebplist(instadr()) **quadruple** ("goto", **?**, $\emptyset$, $\emptyset$)
B	::=	$\langle$E$\rangle$ **rel** $\langle$E$\rangle$	t = **new** TemporaryVariable() **quadruple** (**rel**.operator, E_1.addr, E_2.addr, t) head.bptruelist = makebplist(instadr()) **quadruple** ("if", t, **?**, $\emptyset$) head.bpfalselist = makebplist(instadr()) **quadruple** ("goto", **?**, $\emptyset$, $\emptyset$)
B	::=	$\langle$B$\rangle$ || $\langle$B$\rangle$	**backpatch** (B_1.bpfalselist, instadr()) head.bptruelist = mergbplists(B_1.bptruelist, B_2.bptruelist) head.bpfalselist = B_2.bpfalselist

S	::=	**if (** ⟨B⟩ **)** ⟨S⟩	backpatch(B.bptruelist, instadr()) head.bpnextlist = mergebplists(B.bpfalselist, S.bpnextlist)
S	::=	**if (** ⟨B⟩ **)** ⟨S⟩ **else** ⟨S⟩	backpatch(B.bptruelist, instadr()) backpatch(B.bpfalselist, instadr()) head.bpnextlist = mergebplists(S_1.bpnextlist, S_2.bpnextlist)
S	::=	**while** **(** ⟨B⟩ **)** ⟨S⟩	a = instadr() backpatch(B.bptruelist, instadr()) backpatch(S.bpnextlist, a) **quadruple** ("goto", a, ∅, ∅) head.bpnextlist = B.bpfalselist

The attribute `bpnextlist` is the list of the addresses of the instructions that are refering the "next instruction"

Procedure backpatch(list, address)
Input : $\mathbb{Q}$ is the global list of the generated quadruples
begin
  foreach $a \in list$ **do**
    q $\leftarrow \mathbb{Q}[a]$;
    if $q.op = "goto"$ **then**
      if $q.arg_1 \neq$ **?** **then** `Throw("Cannot backpatch")`;
      q.$arg_1 \leftarrow$ address;
    else if $q.op = "if"$ **then**
      if $q.arg_2 \neq$ **?** **then** `Throw("Cannot backpatch")`;
      q.$arg_2 \leftarrow$ address;
    else if $q.op = "ifFalse"$ **then**
      if $q.arg_2 \neq$ **?** **then** `Throw("Cannot backpatch")`;
      q.$arg_2 \leftarrow$ address;
    else
      `Throw("Instruction to backpatch not found")`
    end
  end
end

- Inherited and synthesized attributes: Syntax-directed definitions may use two kinds of attributes. A synthesized attribute at a parse-tree node is computed from attributes at its children. An inherited attribute at a node is computed from attributes at its parent and/or siblings
- Dependency graphs: Given a parse tree and an SDD, we draw edges among the attribute instances associated with each parse-tree node to denote that the value of the attribute at the head of the edge is computed in terms of the value of the attribute at the tail of the edge
- S-Attributed definitions: In a S-attributed SDD, all attributes are synthesized
- L-Attributed definitions: In a L-attributed SDD, attributes may be inherited or synthesized. However, inherited attributes at a parse-tree node may depend only on inherited attributes of its parent and on (any) attributes of siblings to its left
- Syntax trees: Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct

- Intermediate representation: An intermediate representation is typically some combination of a graphical notation and three-address code. As in syntax, a node in a graphical notation represents a construct; the children of a node represent its subconstructs. Three address code takes its name from instructions of the form `x = y op z`, with at most one operator per instruction. There are additional instructions for control flow
- Translate expressions: Expressions with built-up operations can be unwound into a sequence of individual operations by attaching actions to each production of the form $E \rightarrow E_1 \mathbf{op} E_2$. The action either creates a node for E with the nodes for E_1 and E_2 as children, or it generates a three-address instruction that applies op to the addresses for E_1 and E_2 and puts the result into a new temporary name, which becomes the address of E

- Check types: The type of an expression $\mathtt{E}_1$ op $\mathtt{E}_1$ is determined by the operator **op** and the types of $\mathtt{E}_1$ and $\mathtt{E}_2$. A coercion is an implicit type conversion. Intermediate code contains explicit type conversions to ensure an exact match between operand types and the types expected by an operator
- Generate jumping code for boolean expression: In short-circuit or jumping code, the value of a boolean expression is implicit in the position reached in the code. Jumping code is useful because a boolean expression B is typically used to `t=true` or `t=false`, as appropriate, where `t` is a temporary name. Using labels for jumps, a boolean expression can be translated by inheriting labels corresponding to its true and false exits attributes. The constants **true** and **false** translate into a jump to the true and false attributes, respectively

- Implement statements using control flow: Statements can be translated by inheriting a label next, where next marks the first instruction after the code for this statement. The conditional $\langle \mathrm{S} \rangle \rightarrow \textbf{if(}\langle \mathrm{B} \rangle\textbf{)}\langle \mathrm{S} \rangle$ can be translated by attaching a new label marking the beginning of the code for S and passing the new label and `S.next` for the true and false attributes, respectively, of B
- Alternatively, use backpatching: Backpatching is a technique for generating code for boolean expressions and statements in one pass. The idea is to maintain lists of incomplete jumps, where all the jump instructions on a list have the same target. When the target becomes known, all the instructions on its list are completed by filling in the target
- Implement records: Field names in a record or class can be treated as a sequence of declarations. A record type encodes the types and relative addresses of the fields. A symbol table object can be used for this purpose

Brooker, R. and Morris, D. (1962).
A general translation program for phrase structure languages.
J. ACM, 9(1):1–10.

Gosling, J. (1995).
Java intermediate bytecodes.
In *ACM SIGPLAN Workshop on Intermediate Representations*.

Irons, E. (1961).
A syntax-directed compiler for ALGOL 60.
Comm. ACM, 4(1):51–55.

Jazayeri, M., Ogden, W., and Rounds, W. (1975).
The intrinsic exponential complexity of the circularity problem for attribute grammars.
Comm. ACM, 18(12):697–706.

Johnson, S. (1979).
A tour through the portable C compiler.
Technical report, Bell Telephone Laboratories Inc., Murray Hill, N.J.

Knuth, D. (1968).
Semantics of context-free languages.
Mathematical Systems Theory, 2(2):127–145.

Lewis, P., Rosenkrantz, D., and Stearns, R. (1974).
Attributed translations.
J. Computer and System Sciences, 9(3):279–307.

Milner, R. (1978).
A theory of type polymorphism in programming.
J. Computer and System Sciences, 17(3):348–375.

Paakki, J. (1995).
Attribute grammar paradigms – a high-level methodology in language implementation.
Computing Surveys, 27(2):196–255.

Pierce, B. (2002).
Types and Programming Languages.
MIT Press, Cambridge.

Ritchie, D. (1979).
A tour through the portable UNIX C compiler.
Technical report, Bell Telephone Laboratories Inc., Murray Hill, N.J.

Samelson, K. and Bauer, F. (1960).
Sequential formula translation.
Comm. ACM, 3(2):76–83.

Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T. (1958).
The problem of programming communication with changing machines: a proposed solution.
Comm. ACM, 1(8):12–18.

Wirth, N. (1971).
The design of a pascal compiler.
Software - Practice and Experience, 1(1):309–333.

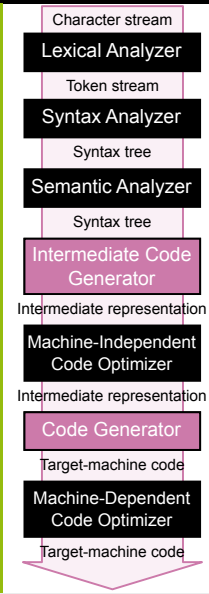
Chapter 5
Run-time Environments

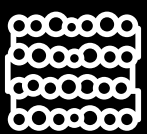
Stéphane GALLAND

Compiler implements abstractions from source language

Compiler cooperates with operating system and other systems software to support these abstracts on the target machine

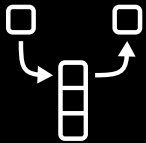
Compiler creates and manages a **run-time environment** in which it assumes its target programs are being executed





Heap

Management of dynamic memory allocation



Stack

Management of static memory allocation

Dynamic Memory Deallocation

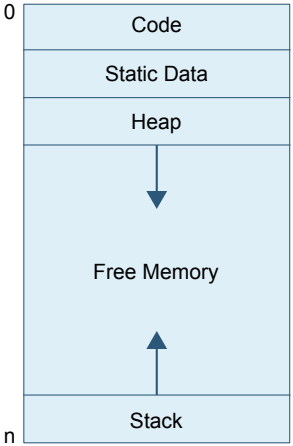
Management of deallocation

Target program runs in its own logical address space in which each value has a location

Operating system maps the logical addresses into physical addresses

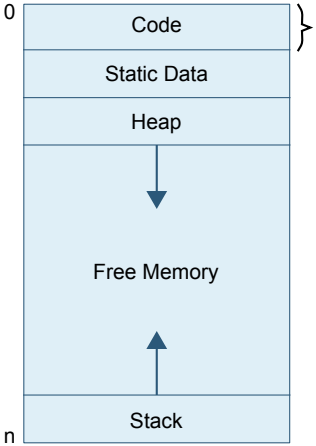
Virtual machine represents the operating system and the target machine into an abstract and platform-independent machine

In the run-time environment, logical address space has a structure; typically:



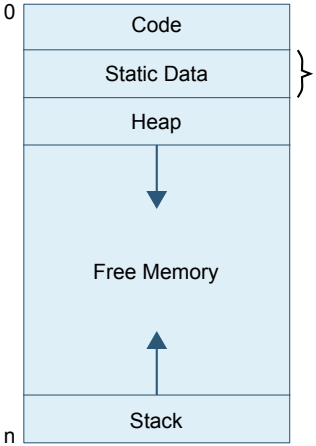
[Randell and Russel, 1964]

In the run-time environment, logical address space has a structure; typically:



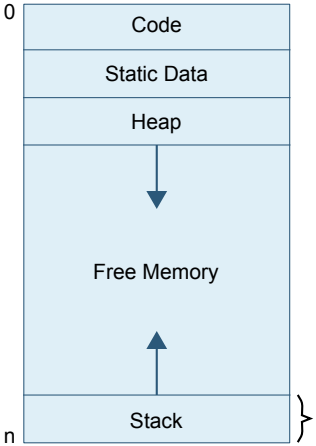
- Compiler places executable target code in a statically determined area, named Code
- Contains binary representations of the instructions to execute
- Format of the code depends on the target machine: Intel binary assembler, byte code...

In the run-time environment, logical address space has a structure; typically:



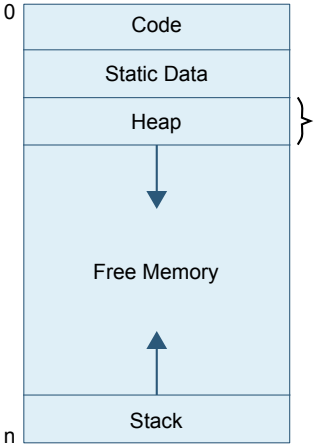
- Size of some program data may be known at compile time
- Area where these data are stored in named Static Data, usually put just after the Code area
- Examples: string literals, global constants and variables, information related to garbage collection...
- Address of static data is directly put in the code

In the run-time environment, logical address space has a structure; typically:



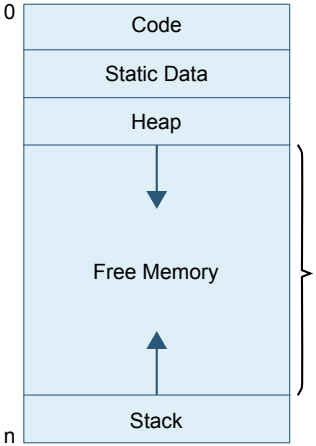
- Stack is used to store data structures, named activation records
- Activation records are generated during the procedure/function calls (explained later)
- Each record contains the status of the machine: ordinal counter, machine registers, and data whose lifetimes are the same as the activation time (usually local variables)

In the run-time environment, logical address space has a structure; typically:



- Many languages allow the programmer to allocates and deallocates data under program control (malloc, new...)
- Heap is used to manage this kind of long-lived data

In the run-time environment, logical address space has a structure; typically:



- Heap and the stack are growing up and consume the free memory space between them
- When the stack cannot grow up, the classical “stack overflow” error is fired
- When the heap cannot grow up, the classical “out of memory” error is fired

Static Storage Allocation

Made by the compiler looking only at the text of the program, not at what the program does when it executes
Allocation is usually done on the static data area

Dynamic Storage Allocation

Made only while the program is running
Allocation may be on the stack or the heap (see next slide)

Stack Storage

- Local-scope names are allocated space on a stack
- Stack serves the normal call/return policy for procedures

Heap Storage

- Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.
- The heap is an area of virtual memory that allows data to obtain and release storage.
- Garbage collection: the run-time environment detects useless data in heap and releases them.

[Randell and Russel, 1964]

Compilers that use procedures, functions, or methods as units manage a part of their run-time memory as a stack

Procedure will be used as a generic term for procedure, function and method

When procedure is called
Space for its local variables is pushed on stack

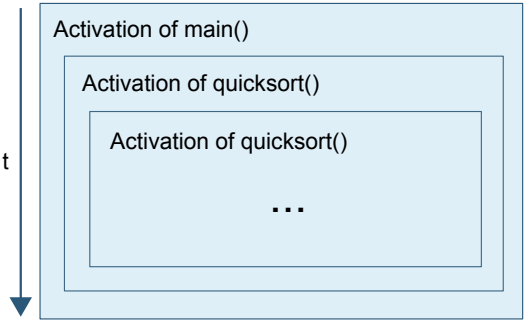
When procedure terminates
Space is popped from stack

Procedure activation
≡
Procedure call

[Randell and Russel, 1964]

Stack allocation would not be feasible if procedure calls did not nest in time

If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end




```
int a[11];
void readArray() {
  int i; // read and fill a
}
int partition(int m, int n) {
  // let v, a[m..p-1] < v, a[p]=v, a[p+1..n] >= v
  // return p
}
void quicksort(int m, int n) {
  int i;
  if (n>m) {
    i = partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
  }
}
main() {
  readArray();
  a[0] = -9999;
  a[11] = 9999;
  quicksort(1,9);
}
```

Three common cases when p calls q

Normal: activation of q terminates normally
Then in many languages, control resumes just after the point of p at which the call to q was made

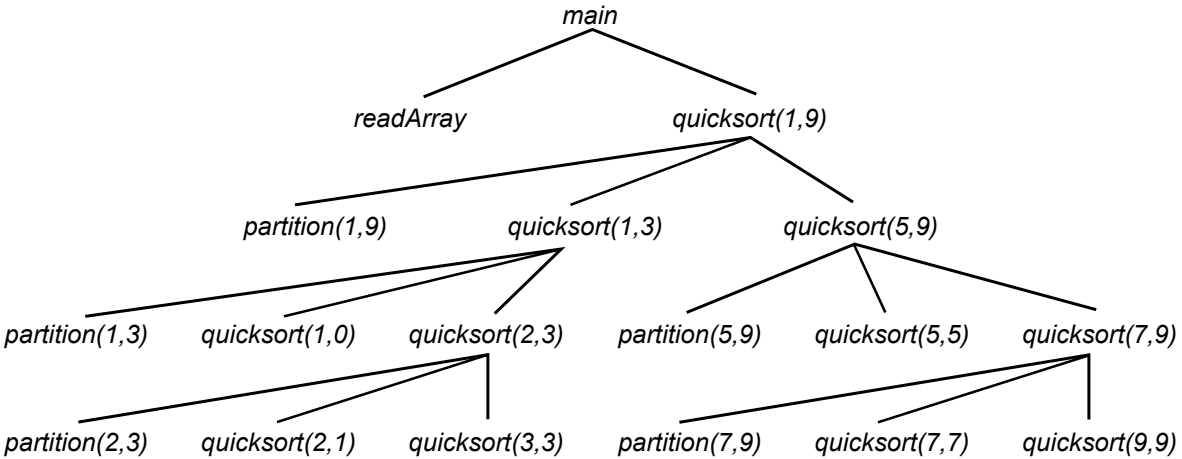
Abort: activation of q, or some procedure called by q, either directly or indirectly, aborts
p ends simultaneously with q

Exception: activation of q terminates because of an exception that q cannot handle
Procedure p may handle the exception: activation of q has terminated but p continues (not necessary where q was called)
If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and exception will be handled by some other open activation of a procedure

Definition

Activations of procedures during the running is represented by a tree, where:

- Node: an activation
- Root Node: activation of the “main” procedure
- Child Node: activations of the procedures called by the activation represented by the parent node
  The order of the children (from left to right) is the order of the activations


main
readArray
quicksort(1,9)
partition(1,9)
quicksort(1,3)
quicksort(5,9)
partition(1,3)
quicksort(1,0)
quicksort(2,3)
partition(5,9)
quicksort(5,5)
quicksort(7,9)
partition(2,3)
quicksort(2,1)
quicksort(3,3)
partition(7,9)
quicksort(7,7)
quicksort(9,9)

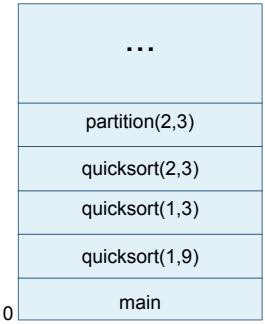
Activation Record

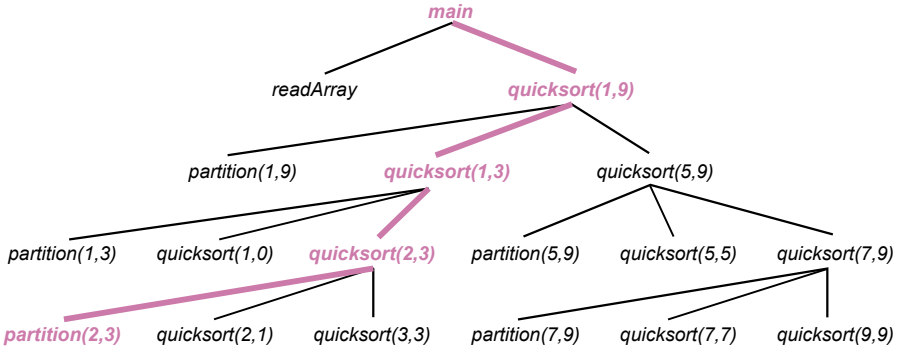
List of information that are describing a procedure activation

Control Stack

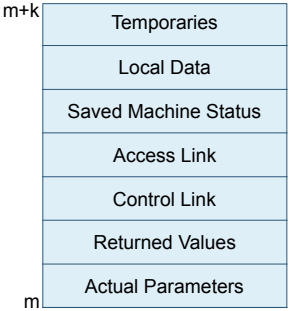
Stack used for managning the procedure calls and returns
Each live activation has an activation record (or frame) on the control stack

- Entire sequence of activation records on the stack is the path in the activation tree to the activation where control currently resides
- Latter activation has its record at the top of the stack

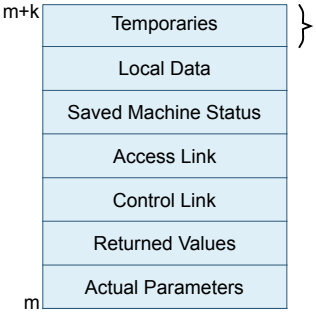

...
partition(2,3)
quicksort(2,3)
quicksort(1,3)
quicksort(1,9)
main
0


main
readArray
quicksort(1,9)
partition(1,9)
quicksort(1,3)
quicksort(5,9)
partition(1,3)
quicksort(1,0)
quicksort(2,3)
partition(5,9)
quicksort(5,5)
quicksort(7,9)
partition(2,3)
quicksort(2,1)
quicksort(3,3)
partition(7,9)
quicksort(7,7)
quicksort(9,9)

Contents of activation records vary with the language being implemented; typically:

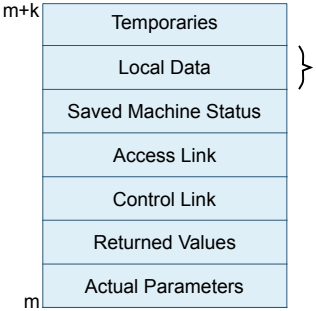


Contents of activation records vary with the language being implemented; typically:



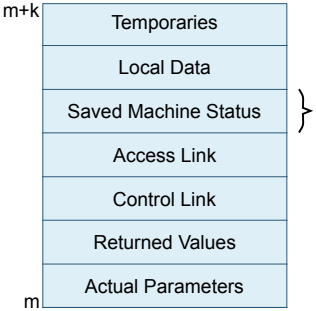
- Temporary values t_k
- Arising from the evaluation of expressions
- Only in case where the temporaries cannot be held in processor registers

Contents of activation records vary with the language being implemented; typically:



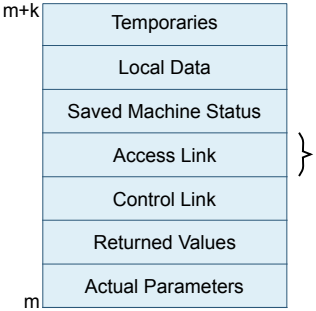
- Local data declared in activated procedure

Contents of activation records vary with the language being implemented; typically:



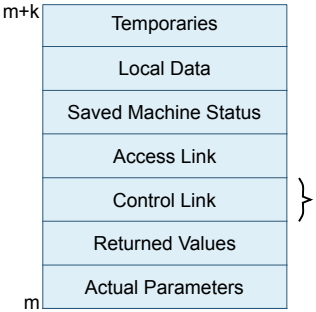
- Information about the state of the machine just before the call to the procedure
- It typically includes:
  - Return address: value of the ordinal counter to which the called procedure must return
  - Registers: Contents of registers that were used by the calling procedure and that must be restored when the return occurs

Contents of activation records vary with the language being implemented; typically:



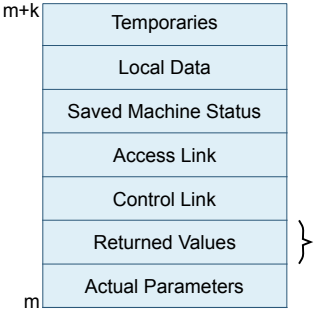
- "Access link" to locate data needed by the called procedure found elsewhere (in another activation record...)

Contents of activation records vary with the language being implemented; typically:



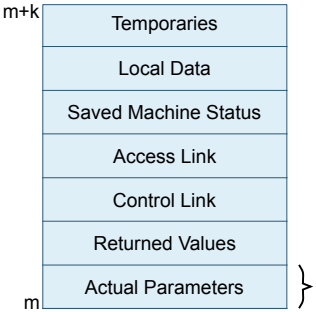
- "Control link" is pointing to the activation record of the caller

Contents of activation records vary with the language being implemented; typically:



- Space for the return value of the called function, if any
- Not all called procedures return a value
- We may prefer to place that value in a register for efficiency

Contents of activation records vary with the language being implemented; typically:



- Actual parameters are given by the caller and used by the callee procedure
- Commonly, these values are not placed in the activation record but rather in registers, when possible

Calling Sequence

A code that allocates an activation record on the stack and enters information into its fields

Return Sequence

A code that deallocates an activation record from the stack and restores the state of the machine

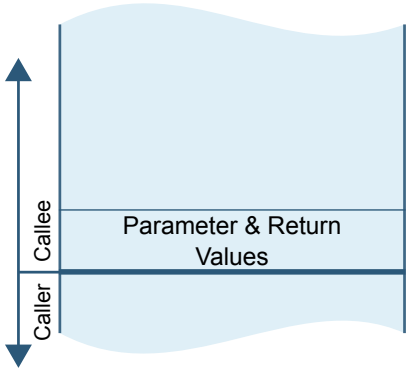
Calling sequences and the layout of activation records may differ greatly, even among implementations of the same language

[Johnson and Ritchie, 1981]

1. Values communicated between caller and callee procedures are generally placed at the beginning of the callee activation record

Caller can compute the actual parameters and put them at the top of the stack, without the necessity to create the entire record of the callee, and knowing how the callee's record layout is

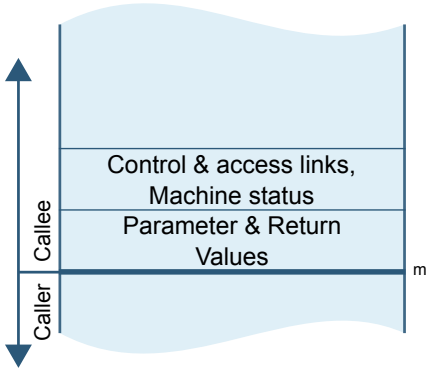
Caller knows where to put the return value, relative to its own record



[Randell and Russel, 1964]

2 Fixed-length items are placed in the middle of the record

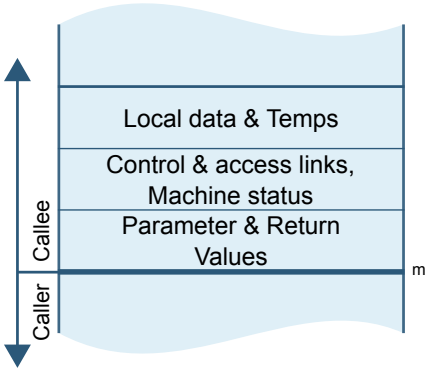
If machine status are standardized, then programs such as debuggers will have an easier time deciphering the stack contents if an error occurs



3 Items those size may not be known early enough are placed at the end of the activation record

Most of the variables have a size that can be determined by the compiler. But some cannot (dynamic arrays...)

Amount of space needed for temporaries is not known during the first phase of the intermediate code generation

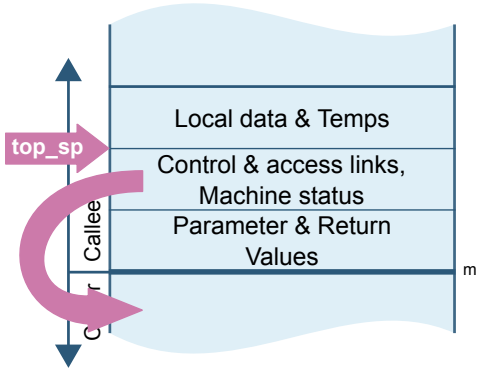


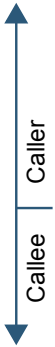
4 "Top-of-stack" pointer must be located judiciously

Commonly, it points to the end of the fixed-length fields in the activation record

Control link points to the "top-of-stack" of the previous record

Fixed-length data can then be accessed by a fixed negative offset, and variable-length with a run-time positive offset





1. Caller evaluates and stores the actual parameters
2. Caller stores a return address,
   stores the old value of `top_sp` into the callee's, activation record
   increments `top_sp` to point to the callee activation record
3. Callee saves the register values and other status information
4. Callee initializes its local data and begins execution

1. Callee places the return value next to the parameters
2. Using information in the machine-status fields, callee restores top_sp and other registers,
   branches to the return address that the caller placed in the status field
3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp
   Caller may use that value

Local Variable-length Data

Programs contain a lot of data whose **sizes are known at run-time**; but which are local to a procedure

Because **they are local to the procedure**, they may be allocated on the stack

- In most of the modern languages, these objects are allocated in the heap
- However, it is also possible to allocate objects, arrays, or other data structures of unknown size on the stack

Why on the stack?

Avoiding the expense of garbage collecting the space allocated for the variable-length data.

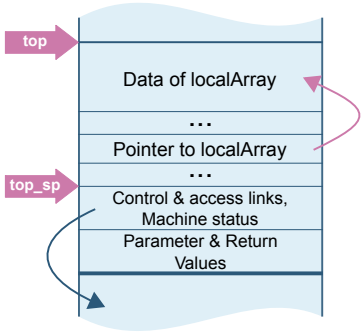
Below, example of programs in C99 and C# in which a local array is declared
Its size depends on the value of the procedure parameter

```c
/* C99 */
void myFunction(int n) {
  float localArray[n];
  /* Do something */
}
```

```c
/* C# */
unsafe void myFunction(int n) {
  int* localArray = stackalloc int[n];
  /* Do something */
}
```

The common strategy is to:

1. Allocate the arrays at the end of the record
2. Put pointers to the allocated regions in the local data

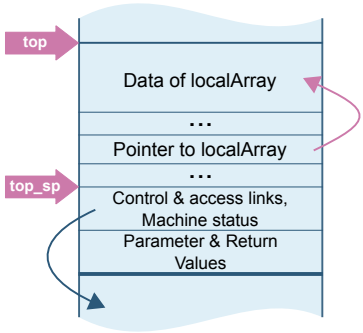


top

Marks the actual top of the stack
It points to the position at which the next activation record will begin

top_sp

Used to find local, fixed-length fields of the top activation record



$$\texttt{top} \leftarrow \texttt{top\_sp} - \texttt{length(fixed\_record\_part)}$$

How could procedure p access to data defined outside p?

Programs without nested procedures

Programs with nested procedures

Many languages (C...) disallow nested procedures

Storage Allocation

- Global variables are allocated in the static storage: locations remain fixed and are known at compile time
- Any other name must be local to the activation at the top of the stack: locations are relative to the `top_sp` pointer in stack

Many languages enable to declare procedures inside the scope of another procedure (Algol60, Pascal, ML, LISP)

Factorial Function, non-tail-recursion algorithm

```
(deffun factorial (n)
  (if (<= n 1)
    1
    (* n factorial (- n 1))))
```

Factorial Function, tail-recursion algorithm

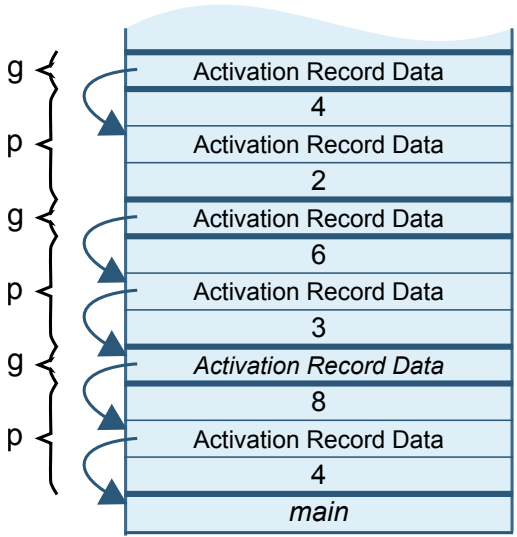
```
(deffun factorial (n)
  (let ((deffun fact (n,acc)
    (if (<= n 1) acc
      (fact (- n 1) (* n acc))))
      (fact n 1)))
```

With nested procedure declaration, it is more complicated to determine the addresses of the names used in the procedure

Example

- Let the procedure g declared inside the scope of the procedure p
- g is accessing to the variable a, locally declared in p
- It is difficult to determine at compile time where is the variable a in the stack, because of the recursive calls
- The address of a in the stack can be determined only at run-time

```
Procedure p(n)
begin
    Declare a ← n/2;
    Procedure g()
    begin
        if n > 1 then p(n-1);
        else if n = 1 then p(a/2);
    end
    g() ;
end
```




```
Procedure p(n)
begin
    Declare a = n*2;
    Procedure g()
    begin
        if (n>1) then p(n-1);
        else if (n=1) then
        p(a/2);
    end
    g() ;
end

p(4)
```

Nesting Depth

- 1: p is declared outside another procedure
- n: p is declared inside a procedure with nesting depth $n-1$

```
Procedure p(n)
begin
    Declare a ← n/2;
    Procedure g()
    begin
        if n > 1 then
          p(n-1);
        ;
        else if n = 1 then
          p(a/2);
        ;
    end
    g() ;
end
```

p has nesting depth 1

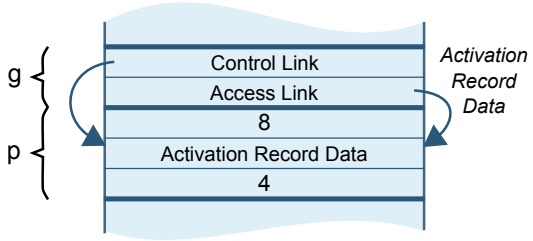
g has nesting depth 2

Definition

Access link provides a mean for the implementation of the static scope rule for nested function

If procedure g is immediately nested in procedure p; then the access link in any activation of g points to the most recent activation of p

Nesting depth of p must be exactly one less than the nesting depth of g

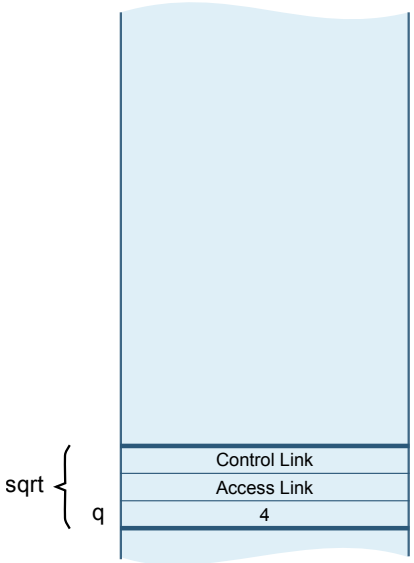



```
Procedure p(n)
begin
    Declare a ← n/2;
    Procedure g()
    begin
        if n > 1 then p(n-1);
        ;
        else if n = 1 then p(a/2);
        ;
    end
    g() ;
end
```

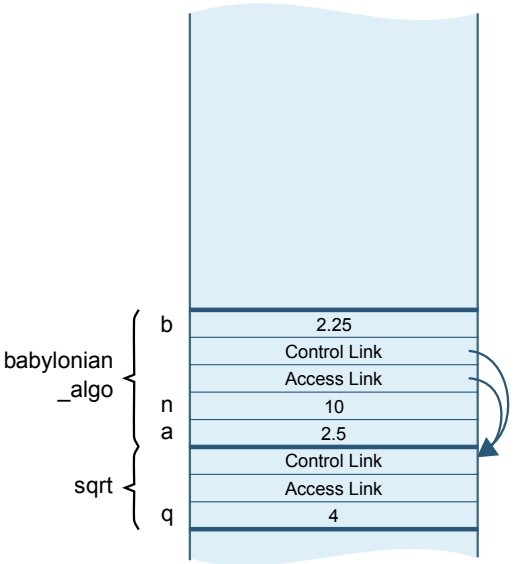
Definition

Access links form a chain from the activation record at the top of the stack to activations at lower nesting depths

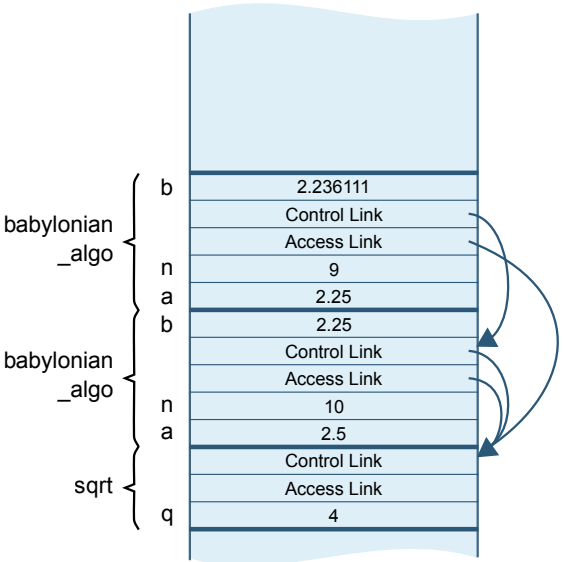
Along this chain, all data declared in the procedures are accessible to the currently executing procedure



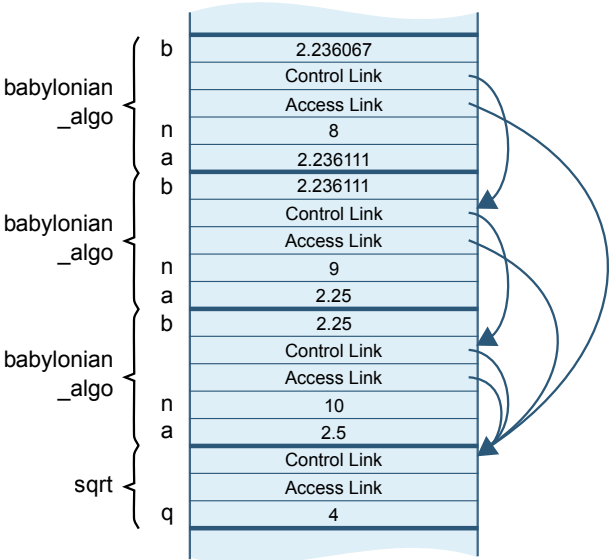

```
Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(5);
```




```
Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(5);
```




```
Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(5);
```




```
Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(5);
```

To access to the value of q, we know at compile time, that it is reachable after one dereferencing in the access link pointer chain

- Let the procedure q calling p.
- Let N_α the nesting depth of α.
- Let D_β the set of the nesting procedures in which β is defined.

First Case

$$(N_p > N_q) \Rightarrow (q \in D_p \wedge N_p = N_q + 1)$$

Then the access link from p leads to q.

- Let the procedure q calling p.
- Let N_α the nesting depth of α.
- Let D_β the set of the nesting procedures in which β is defined.

Second Case

$$(N_p \leq N_q) \Rightarrow \left(\exists r \;\middle|\; \begin{pmatrix} r \in D_p \wedge N_p = N_r + 1 \wedge \\ r \in D_q \wedge N_r > N_q \end{pmatrix}\right)$$

Then

- The access link from p leads to r.
- There is $N_q - N_p + 1$ access links from q to r.
- Include recursive calls, where p = q.

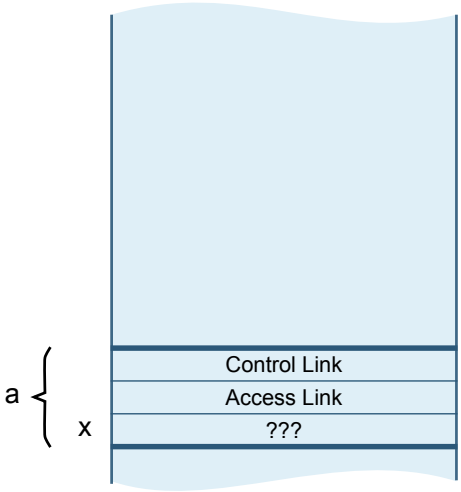
A procedure p is passed to another procedure q as a parameter; q calls its parameter

Problem

- If q does not know the context in which p appears in the program;
- it is impossible for q to know how to set the access link for p

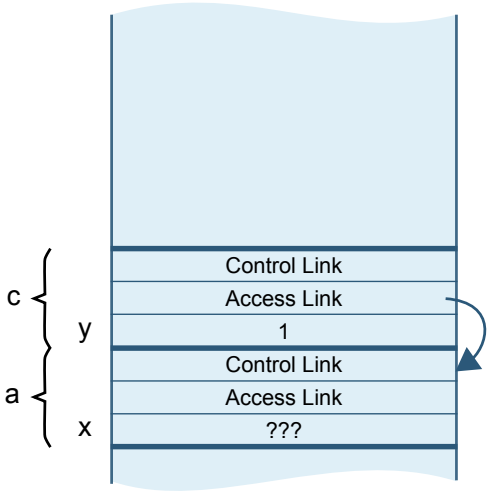
Solution

- Caller of a procedure with a procedure as parameter must also pass the proper access link to the parameter
- i.e. caller must pass the name and the access link as parameters



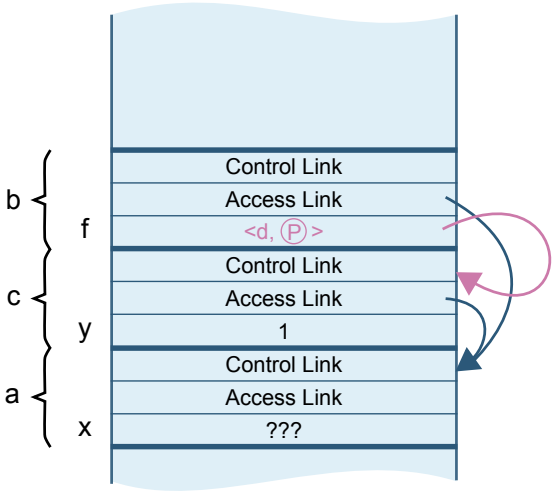
Function a is called

```
(defun a(x)
    (let (defun b(f)
            (... f ...)
        )
        (defun c(y)
            (let (defun d(z) (...)) )
                (... (b d) ...)
            )
        )
        )
        (... (c 1) ...)
    )
)
```



Function c is called
According to the first case, access link leads to a

```
(defun a(x)
    (let (defun b(f)
            (... f ...)
        )
        (defun c(y)
            (let (defun d(z) (...)) )
                (... (b d) ...)
            )
        )
        )
        (... (c 1) ...)
    )
)
```

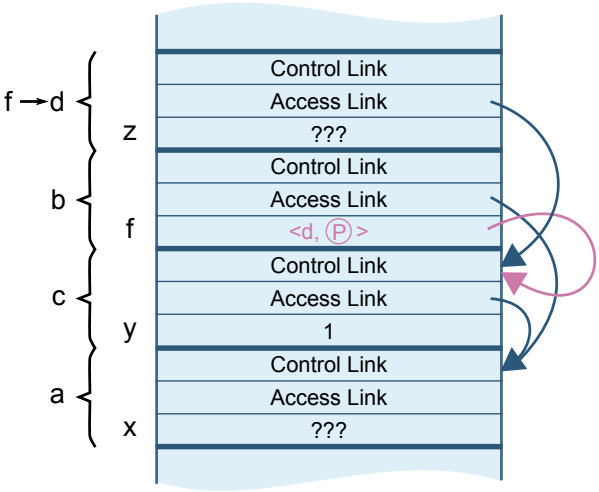


Function b is called with the procedure d as parameter

According to the second case, access link leads to a

Context of d is also passed as parameter

```
(defun a(x)
    (let (defun b(f)
            (... f ...)
        )
        (defun c(y)
            (let (defun d(z) (...)) )
                (... (b d) ...)
            )
        )
        )
        (... (c 1) ...)
    )
)
```



Function `d` is called through the parameter `f`. The access link is directly taken from the context Ⓟ

```
(defun a(x)
    (let (defun b(f)
            (... f ...)
        )
        (defun c(y)
            (let (defun d(z) (...)) )
                (... (b d) ...)
            )
        )
        )
        (... (c 1) ...)
    )
)
```

If the nesting depth gets large, we may have to follow long chains of links to reach the data we need

Use of an auxiliary array d, called the display

Display *d*

Collection (e.g., array) of pointers, one for each nesting depth

$d[i]$ is a pointer to the highest activation record on the stack for any procedure at nesting depth i

[Dijkstra, 1960]

Direct access to a context at compile time

Compiler knows what `i` is, so it can generate code to access `x` using `d[i]` and the offset of `x` from the top of the activation record for `q`

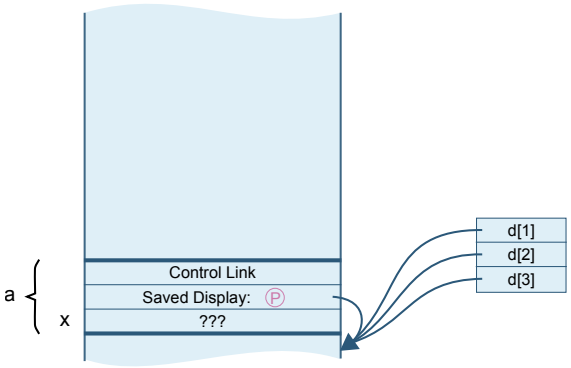
Direct access to a context at run-time

If procedure `p` is executing, and it needs to access element `x` belonging to some procedure `q`, we need to look only in `d[i]`, where `i` is the nesting depth of `q`

Limited Chain

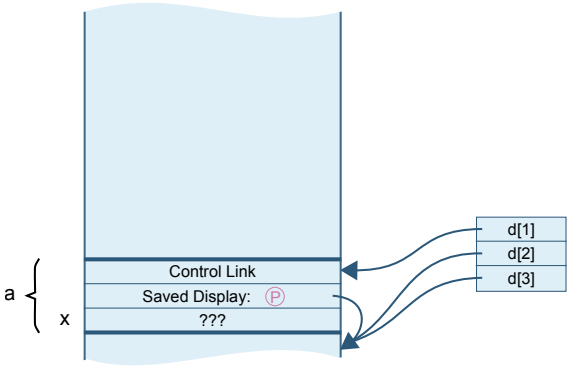
Code never needs to follow a long chain of access links

Inputs : Stack s; called procedure p; nesting depth of p is N_p
begin
  Ⓟ of $p \leftarrow d[N_p]$;
  if $d[N_p] \neq$ *any activation record of* p **then**
    $d[N_p] \leftarrow$ activation record of p
  end
end



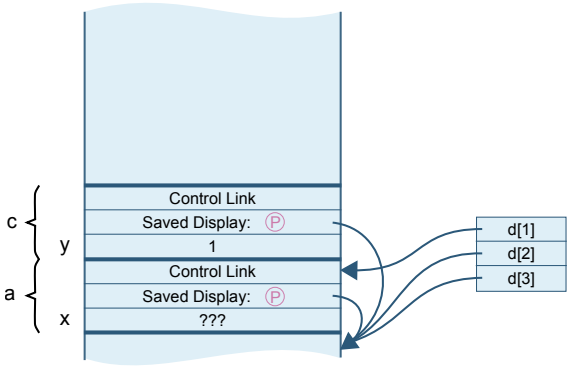
The displays are pointing somewhere in the stack
Function a is called
Create the record
Save `d[1]`, which is pointing on a lower activation record

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
            )
            (defun c(y)
                  (let ( (defun d(z) (...)) )
                        (... (b d) ...)
                  )
            )
            (... (c 1) ...)
      )
)
```



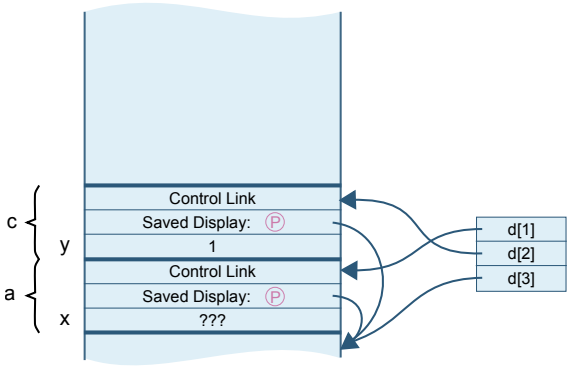
Because d[1] is not pointing to the record of a, change d[1]

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
             )
             (defun c(y)
                  (let ( (defun d(z) (...)) )
                       (... (b d) ...)
                  )
             )
             (... (c 1) ...)
      )
)
```



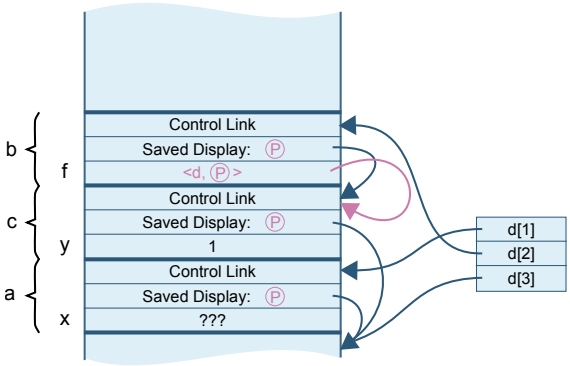
Function c is called
Its record is created
The previous value of d[2] is saved

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
             )
             (defun c(y)
                  (let ( (defun d(z) (...)) )
                        (... (b d) ...)
                  )
             )
             (... (c 1) ...)
      )
)
```



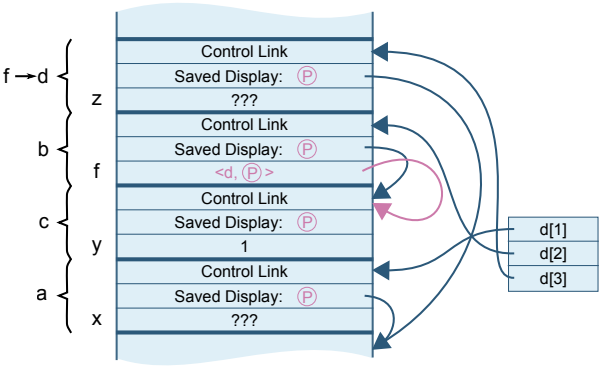
Because the record of c is not the one pointed by d[2], set d[2] to leads to c

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
            )
            (defun c(y)
                  (let ( (defun d(z) (...)) )
                        (... (b d) ...)
                  )
            )
            (... (c 1) ...)
      )
)
```



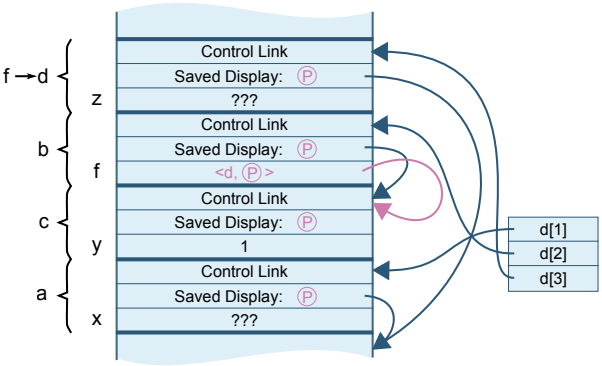
Function `b` is called
Parameter *f* is set to value of `d[2]`
Save the `d[2]`, and set its value to
leads to `b`

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
            )
            (defun c(y)
                  (let ( (defun d(z) (...)) )
                        (... (b d) ...)
                  )
            )
            (... (c 1) ...)
      )
)
```



Function d is called through the parameter f
Displays are updated

```
(defun a(x)
      (let ( (defun b(f)
                  (... f ...)
            )
            (defun c(y)
                  (let ( (defun d(z) (...)) )
                        (... (b d) ...)
                  )
            )
            (... (c 1) ...)
      )
)
```



To obtain the value of x:

- Because x is at nesting depth 1, follows d[1] to reach the right record
- Read the value of x in the record

```
(defun a(x)
    (let ( (defun b(f)
                (... f ...)
           )
           (defun c(y)
                (let ( (defun d(z) (...)) )
                     (... (b d) ...)
                )
           )
           (... (c 1) ...)
    )
)
```

Inputs : Stack s; called procedure p; nesting depth of p is N_p
begin
| $d[N_p] \leftarrow$ Ⓟ of of p
end

Memory Heap

Portion of the memory store that is used for data that lives indefinitely, or until the program explicitly deletes it

- Modern languages provides dedicated operators for the allocation and deallocation in the heap

Example

new and **delete** in C++

Memory Manager

Subsystem that allocates and deallocates space within the heap

Interface between the application program and the operating system

Garbage Collection

Process of finding spaces within the heap that are no longer used by the program and can be reallocated

Garbage collector is an important subcomponent of the memory manager

Produces a chunk of contiguous memory for each variable or object associated to the allocation request

If not enough contiguous space is available for a chunk, it seeks to increase the heap storage space by requesting memory to the operating system

Defragmentation of the heap is generally not implemented

Returns deallocated space to the pool of free space
Deallocation space may be reused for future allocations

Typically, memory manager does not return memory to the operating system, even if the program's heap usage drops

Space Efficiency Property

Memory manager should minimize the total heap space need by a program

Space efficiency is achieved by minimizing the “fragmentation” (discussed later)

Program Efficiency Property

Memory manager should make good use of the memory subsystem to allow programs to run faster

- The time taken to execute an instruction can vary widely depending on where objects are placed in memory
- Programs tends to exhibit "locality" (discussed later), which refers to the nonrandom clustered way in which typical programs access memory
- By attention to the placement of objects in memory, the memory manager can make better use of space, and make the program run faster

Because memory allocations and deallocations are frequent operations in many programs (such as ones written in Java), it is important that these operations be as efficient as possible

Low Overhead Property

Minimize the overhead, the fraction of execution time spent performing allocation and deallocation

Overhead of allocation is dominated by a large amount of small requests; the overhead of managing large objects is less important

Program Efficiency Property

Efficiency of a program is determined by:

1. the number of instructions executed
2. the time taken to execute each of these instructions

- Data-intensive programs can therefore benefit significantly from optimizations that make good use of the memory subsystem
- Run-time environment should prefer to use the memory storages close to the processor, e.g. registers
- Concept of "locality" will help us to improve the use of the memory subsystem

Hypothesis / Conventional Wisdom

Programs spend 90% of their time executing 10% of the code

Temporal Locality

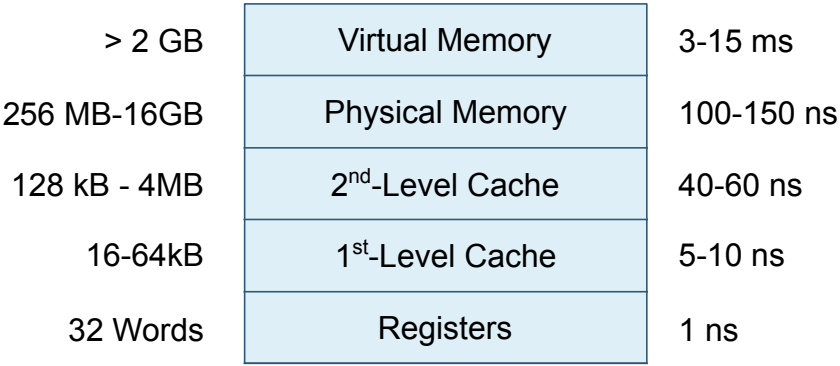
Memory locations are likely to be accessed again within a short period of time

Spatial Locality

Memory locations close to the accessed location are to be accessed within a short period of time

1. Programs often contains many instructions that are never executed
2. After evolution, legacy systems contain many instructions that are no longer used
3. Only a small fraction of the code that could be invoked is actually executed in a typical run of the program
4. Typical program spends most of its time executing innermost loops and tight recursive cycles in a program

- Memory manager (and compiler optimizer) must be aware of how the operating system is managing its memory
- In modern systems, the memory is composed of several layers:



Locality permits to take advantage of the memory hierarchy

Placing the most common instructions and data in the fast-but-small storage

Leaving the rest in the slow-but-large storage

Minimize the average memory-access time

Put the most-recent-used instruction in the fastest memory

Put together in the same memory page/block the instructions that may be always executed together

Locality of data can be improved by changing:
1) the data layout
2) the order of the computations

Example

- Visiting a large amount of data and performing small operations on it is not a good approach
- Preferably, smaller data should be pushed down into a faster memory level, and perform the computations on them

Memory Chunk

A fragment of memory that is allocated and deallocated as a whole

Size of a chunk depends on the type of object to be allocated

General Principle

- At the beginning of the program, the heap is one contiguous unit of free space
- As the program allocates and deallocates memory, this space is broken up into free and used chunks

Memory Hole

Free chunks are named hole

Memory Fragmentation

Alternating chunks and holes is named the fragmentation of the heap

Fragmentation is reduced by controlling how the memory manager places new objects in the heap

Best-Fit Object Placement
Allocate in the smallest available hole that is large enough
Not good for spatial locality

First-Fit Object Placement
Allocate in the first hole, which is able to contains the requested chunk
Less efficient than the previous one

Next-Fit Object Placement
When no hole of the exact size was found, allocate in the lastly split hole
Good for spatial locality and efficient

Bin

Free space chunks are grouped into bins according to their sizes

Many bins for the smaller sizes, because there are usually many more small objects in programs

Lea Memory Manager (GNU C compiler)

- Bins of every multiple of 8 bytes until 512 bytes
- Larger-sized bins are logarithmically spaced
- Within the bins, the chunks are ordered by their sizes
- Wilderness chunk: largest bin because its size may be extended after requesting more memory to OS

Inputs : Size s of the chunk to be allocated; sorted list $\mathbb{S}$ of available sizes of chunks
Output : Allocated chunk c; or error

```
begin
    if ∃c ∈ bin_s then                                  // Search of a bin of size s
        c ← allocate(bin_s) ;
        return c
    end
    foreach α ∈ 𝕊 ∪ {wilderness chunk} do               // Search for smallest chunk
        if ∃c ∈ bin_α then
            ⟨c, r⟩ ← allocate(bin_α, s) ;               // First-fit or best-fit strategy
            β ← α − s ;
            bin_β ← bin_β ∪ r ;
            return c
        end
    end
    throw("Out of memory")
end
```

Dealocation of a Chunk

When an object is deallocated, memory manager makes its chunk free

Coalescing of Free Space

It may also be possible to combine (coalesce) the just freed chunk with adjacent chunks

Boundary Tags

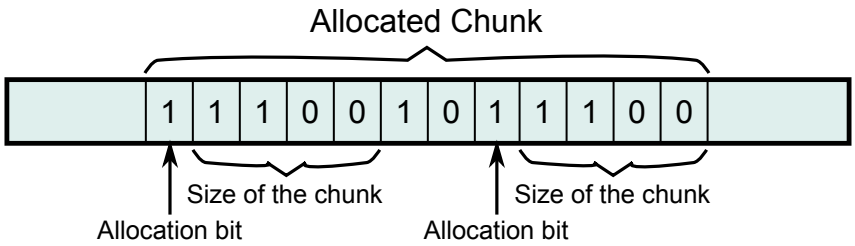
Double-Linked Free List

Chunk with Boundary Tags

Bits of the chunk is composed by:

- **Boundary tags**
- Chunk data
- **Boundary tags** (again)

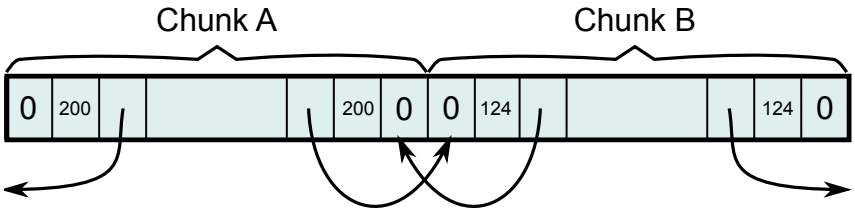
Order of tags depends on run-time environment



[Knuth, 1968]

Free Chunks in a Double-Linked List

- Free chunks (but not the allocated ones) are linked in a double-linked list
- Boundary tags include pointers to the previous and next free chunks
- Does not need to allocate more space for these pointers: the pointers takes the unused bytes of the free chunks
- For the smaller chunks, they are expanded to allow to contain the pointers



- Any storage that will no longer be accessed should be deleted
- Any storage that may be referenced must not be deleted
- It is hard to enforce these properties

Two common errors may occurs in manual memory management:

1. Memory Leak: failing ever to delete data that cannot be referenced
2. Dangling-pointer reference: referencing deleted data

Observation

- Hard for a developer to tell if a program will never refer to some storage in the future
- Common mistake is not deleting storage that will no more referenced

Problem

May slow down the execution of the program due to increased memory usage

Remarks

- Correctness of the program is not changed
- Many programs may tolerate leaks but not the long-time and critical ones (operating systems, server code...)

1. Automatic garbage collection gets rid of memory leaks by deallocating all the garbage
   Even with a garbage collector, programs may still use more memory than necessary

2. A programmer may know that an object will never be referenced
   He must deliberately remove the references to objects that will never be referenced, so the objects can be deallocated automatically

Observation

- Deletion of a storage, and then referencing the deleted storage
- These pointers are named “dangling pointers”

Problem

- When the storage has been reallocated, it produces random effects on the program
- Writing through a dangling pointer changes another variable than the expecting one by the dangling pointer

Remarks

Read, write or deallocate a pointer is named “dereferencing the pointer”

Unfortunately, there is no “magic” solution

1. Programmer must be aware and may pay attention to his uses of the pointers
2. Dangling-pointer-dereference error does not occurs in run-time environments that have an **automatic garbage collector**

Definition

Occurs when the address to dereference is `null` or outside the bounds of any allocated memory (including the bounds of the memory space of the process)

- Related to the dangling-pointer-dereference error
- At the origin of many security violations from hackers

Solution

- Compiler inserts checks with every access, to make sure it is within the bounds
- Compiler optimizer may remove several of these checks when they are detected as not necessary

Object Ownership

- Associate an **owner** with each object at all times:
  - Usually a function
  - Responsible for either deleting the object or for passing the object to another owner
- Non-owning pointers may reference the object, but the object must never be deallocated through them

- Eliminates memory leaks
- Eliminates deletion of the same object twice

Does not solve the dangling-point-reference problem

Reference Counting

- Associate a **counter** with each dynamically allocated object:
  - counter is incremented when object use is added
  - counter is decremented when object use is deleted
- Object is released when the counter is zero

- Eliminates memory leaks
- Eliminates deletion of the same object twice

- Expensive operation
- Do not work with inaccessible circular data structures

Region-Based Allocation

- When objects are created to be used only within some step of a computation, we can allocate all such objects in the same region
- Entire region is deleted once the computation step is completed

Very efficient

Limited applicability

Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection

Garbage Collection (GC)

Process to deallocate no-more referenced storages from the heap

- First garbage collection dates from the initial implementation of LISP in 1958
- Several languages provide natively a GC: Java, Perl, ML, Modula-3, Prolog, Smalltalk, C#, Ruby, Python...

[Wilson, 1994]

1. Garbage collector must know the type of the objects at run-time. This type permits to determine:
   - the size of the object in bytes
   - its components that are references to other objects
2. References to the objects are always to the address of the beginning of these objects
3. All the references to the same object have the same value and may be identified easily.

Mutator

Mutator, i.e., the user program, modifies the collection of objects in the heap

- Creates objects by acquiring space from the memory manager
- Introduce and drop references to existing objects

Relationship with GC

- Objects become garbage when the mutator program cannot “reach” them
- GC finds the garbage and reclaims their space to the memory manager

Source language must be type safe

Type of the data must be known and determined at compile or run-time

GC must be able to determine if a data is a pointer to a chunk

Statically typed languages
Types are determined at compile time (ML...)

Dynamically typed languages
Types are determined at run-time (Java...)

Unsafe languages (C, C++...) are bad candidate for GC

In unsafe languages, memory addresses can be manipulated arbitrarily (pointer arithmetic...)
Programs can refer to any location in memory at any time

Consequently, no memory location can be considered to be inaccessible
No storage can ever be reclaimed safely

Overall Execution Time

GC should not significantly increase the total run time of an application

Pause Time

Simple GC causes the mutator to pause suddenly for an extremely long time.
Maximum pause time must be minimized

Program Locality

- GC speed cannot be evaluated solely by its running time
- GC controls the placement of data and thus influences the data locality of the mutator program
- GC can improve the mutator's temporal locality by freeing the space and reusing it
- GC can improve the mutator's spatial locality by relocating data used together in the same cache or pages

Root Set

It refer to all data that can be accessed directly by a program, without having to dereference any pointer

Example

In Java, the root set is composed of all the static fields and all the variables in the stack

- Program can reach any member of its root set at any time
- Recursively, any object with a reference that is stored in the field members or array elements of any reachable object is itself reachable

When an object becomes unreachable, it will never be reachable again

Object Allocation

- Performed by the memory manager
- Memory manager returns a reference to each newly allocated chunk of memory
- This operation adds members to the set of reachable objects.

Parameter Passing and Return Values

- References to objects are passed from the actual input parameter to the corresponding formal parameters; and from the returned result back to the caller
- Objects pointed by these references remain reachable

Reference Assignments

- Assignments x = y (x and y are references) have two effects:
  1. x is a now a reference to the object referred by y
     Referenced object by x and y is reachable while x or y is reachable
  2. Original reference of x is lost
     If this lost reference is the last on the object, the object becomes unreachable
- When an object becomes unreachable, all the reachable objects inside becomes unreachable also

Procedure returns

- As a procedure exists, the activation record holding its local variables is popped off from the stack
- If the activation record holds the only reachable reference to any object, that object becomes unreachable
- If the now unreachable objects holds the only references to other objects, they become unreachable too, and so on

Approach #1

Transitions from reachability to unreachability are catched, or
reachable objects are periodically located;
assuming that all the other objects are not reachable

Reference counting is an approximation of the first approach:

- Counter of the reference to an object is maintained
- When the counter goes to zero, the object becomes unreachable (discussed in the next section)

Approach #2

Reachability is computed by tracing all the references transitively

- Trace-based garbage collector starts by labeling/marking all objects in the root set as “reachable”
- Examine iteratively all the references in reachable objects to find more reachable objects
- Mark the discovered objects as “reachable”
- Once the reachable set is computed, GC may find the unreachable objects
- All the unreachable objects could be deallocated at the same time

Simple and imperfect garbage collector based on reference counting

Every object must have a field for the reference counting
Additional field is maintained as described in the following slides

Object Allocation

Reference counter of the new object is set to 1

Parameter Passing

Reference counter of each object passed into a procedure is incremented

Procedure Returns

As a procedure exists, objects referred by the local variables in its activation record have their counters decremented

If several local variables hold references to the same object, that object's counter must be decremented once for each such reference

Reference Assignment

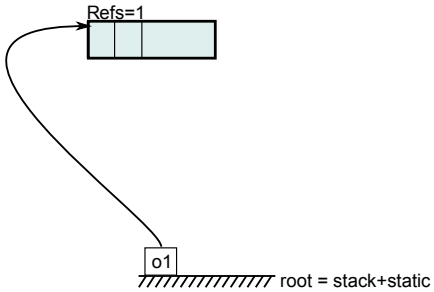
For statement `u=v` (`u` and `v` are references):
counter of the object referred by `v` is incremented
counter of the old object referred by `u` is decremented

Transitive Loss of Reachability

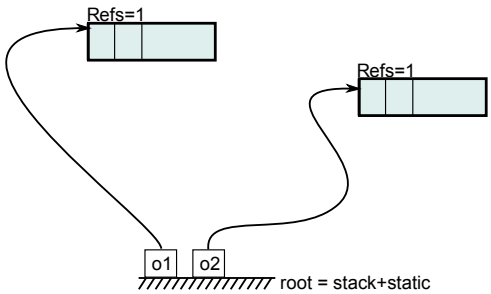
When the reference counter of an object becomes zero, the counter of each object pointed by a reference within the object is decremented

```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```

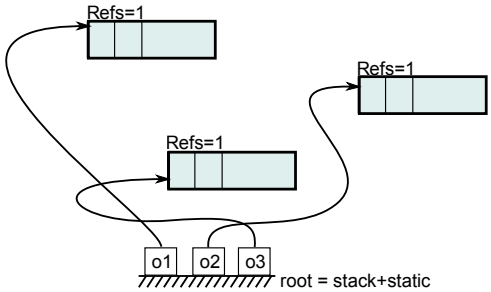
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



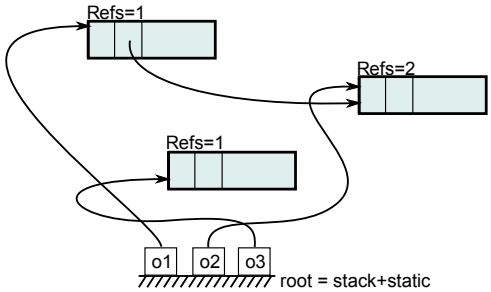
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



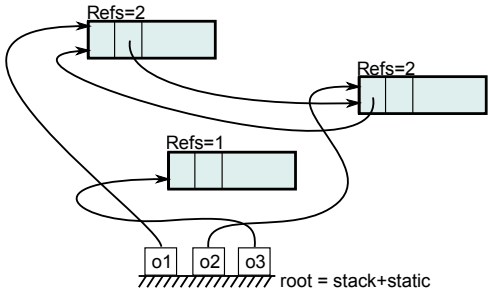
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



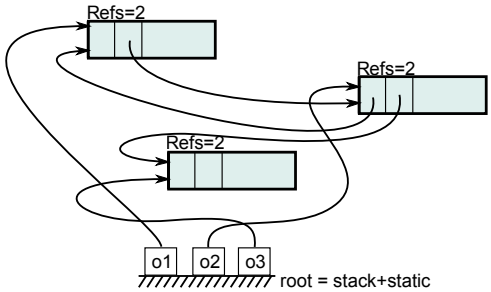
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



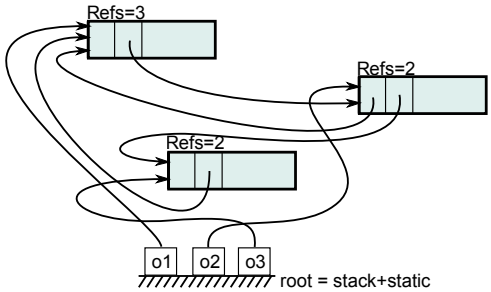
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



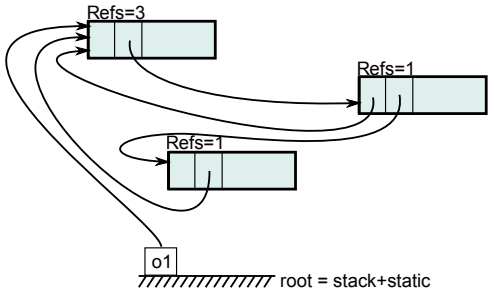
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



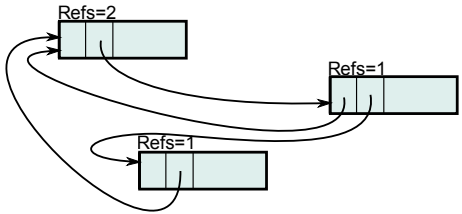
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```

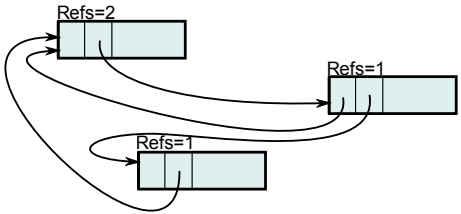


```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```



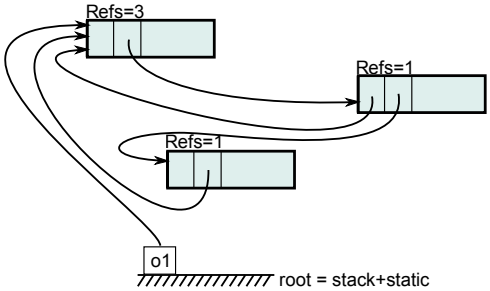
```
class Obj {
  public Obj a = null;
  public Obj b = null;
}

class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
```

This set of objects should be garbage collected. But their counters are greater than 0
Such a situation is tantamount to a memory leak, since this set of objects will never be deallocated

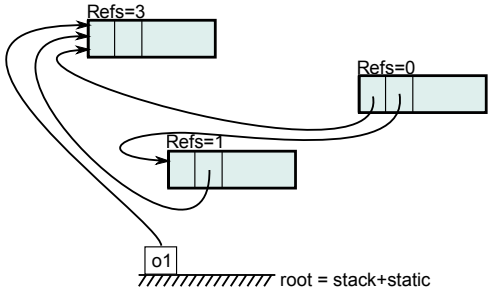


```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```

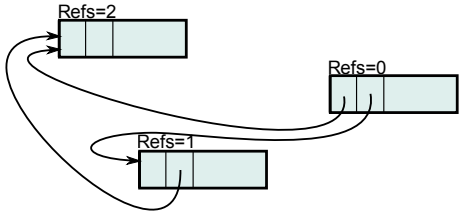
A line is added to reset the reference from o1 to o2



```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```

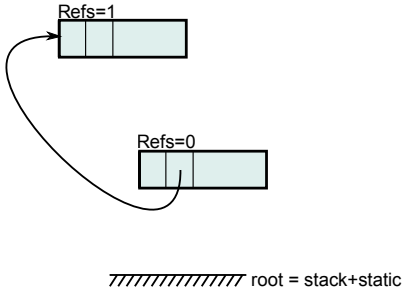


```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```



```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```

Chunk, previously referred by o2, is no more referenced
It is garbage collected



```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```

Chunk previously referred by o3 is garbage collected

root = stack+static

```
class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(
    String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
    o1.b = null;
  }
}
```

Chunk previously referred by o1 is garbage collected
There is **no memory leak**

Weak references in several languages (e.g., Java) may be a good replacement for the added line

Definition

Mean to eliminate the overhead associated with updating the reference counters due to stack accesses

- Reference counts do not include references from the root set of the program
- An object is not considered to be garbage until the entire root set is scanned and no reference to the object is found

1. Garbage Collection is performed in an incremental fashion.
   - The operations are made through the mutator's operations.
   - Removing one reference may render a large number of objects unreachable, the operation of recursively modifying reference counts can easily be deferred and performed piecemeal across time.
   - Reference counting is particularly attractive when timing deadlines must be met.

2. Garbage are collected immediately, keeping space usage low.

1. Reference counting cannot collect unreachable, cyclic data structures.
   - Cyclic data structures are quite plausible.
   - Data structures often point back to their parent nodes, or point to each other as cross references.

2. Overhead of reference counting is high:
   - Additional operations were introduced with each reference assignment.
   - Additional operations were introduced with each procedure call and exit.
   - The overhead is proportional to the amount of computation in the program, and not just to the number of objects in the system.

Instead of collecting garbage as it is created, trace-based collectors run periodically to find unreachable object

When free space is exhausted or its amount drops below a threshold

All trace-based algorithms:

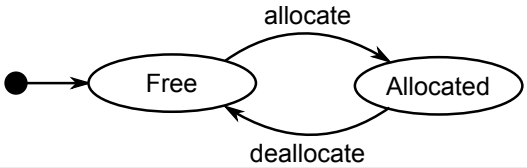
1. compute the set of reachable objects
2. take the complement of this list

Free

- Chunk is in the Free state if it is ready to be allocated
- Free chunk must not hold a reachable object

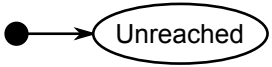
Allocated

- Chunk is in the Allocated state if it was used to store data
- Allocated chunk must be in one of the three substates:
  1. Unreached
  2. Unscanned
  3. Scanned



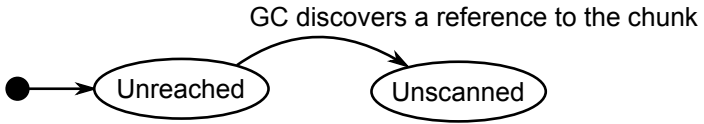
Unreached

- Chunks are presumed unreachable, unless proven reachable by tracing
- Chunk is in the Unreached state at any point during garbage collection if its reachability has not yet been established
- After a round of garbage collection, the state of a reachable object is reset to Unreachable to get ready for the next round (see the next states)



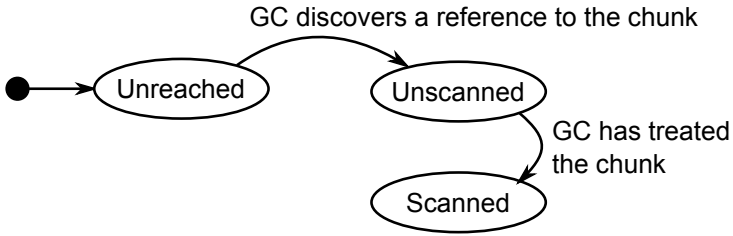
Unscanned

- Chunk is in the Unscanned state if it is known as reachable, but its pointers have not yet been scanned
- Transition to Unscanned from Unreached occurs when we discover that chunk is reachable.

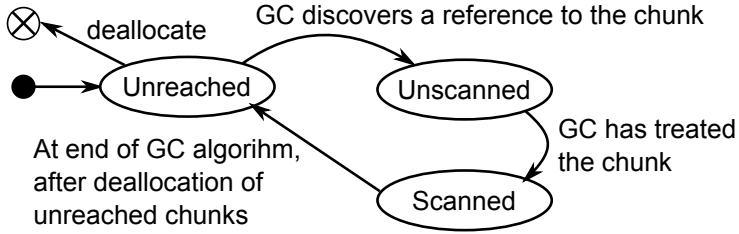


Scanned

- Every Unscanned object will eventually be scanned and move to the Scanned state
- To scan an object, each pointer within it and follow this pointer to the target object
- Scanned object can only contain references to other scanned or unscanned objects, never to unreached objects
  $\Rightarrow$ accessible chunks are moved to the Unscanned state if they are unreachable



- At the end of its algorithm, GC deallocates the unreached chunks
- Chunk states are set to “Unreached” for the next GC execution



Definition

- Mark-and-sweep GC is **“stop-the-world”** algorithm
- Find all the unreachable objects, and put them on the list of free space

- Visits and “marks” all the reachable objects in the first tracing step
- “Sweeps” the entire heap to free up unreachable objects

[McCarthy, 1960]

Inputs : Root set of objects, a heap, and a free list (named Free), with all the unallocated chunks of the heap

Output : Modified `Free` list after garbage has been removed

begin

```
/* MARKING PHASE                                                    */
Unscanned ← copy-of(root);
foreach o ∈ Unscanned do
    reached_bit[o] ← false
end
while ∃ o ∈ Unscanned do
    Unscanned ← Unscanned \{o};
    foreach r ∈ references_in(o) do
        if neg reached_bit[r] then
            reached_bit[r] ← true;
            Unscanned ← Unscanned ∪{r};
        end
    end
end
```

```
    /* SWEEPING PHASE                                                    */
    Free ← ∅;
    foreach c ∈ chunks do
        if neg reached_bit[c] then
            Free ← Free ∪{c}
        else
            reached_bit[c] ← false
        end
    end
end
```

Problem

- Final step in the mark-and-sweep algorithm is expensive
- Not easy way to find unreachable objects without examining the entire heap

Improved Algorithm

- **Baker's Mark-and-sweep Algorithm**
- Keeps a list of all allocated objects
- Computes the difference between allocated objects and reached objects

Inputs : Root set of objects, heap, free list (named `Free`), list of allocated objects `Unreached`
Output : Modified `Free` and `Unreached` lists
begin

```
/* MARKING PHASE                                                    */
Unscanned ← ∅;
Scanned ← ∅;
foreach o ∈ root ∩ Unreached do
    Unreached ← Unreached \{o}; Unscanned ← Unscanned ∪{o};
end
while ∃ o ∈ Unscanned do
    Unscanned ← Unscanned \{o};
    Scanned ← Scanned ∪{o};
    foreach r ∈ references_in(o) do
        if r ∈ Unreached then
            Unreached ← Unreached \{r};
            Unscanned ← Unscanned ∪{r};
        end
    end
end
```

[Baker, 1992]

```
    /* SWEEPING PHASE                                          */
    Free ← Free ∪ Unreached;
    Unreached ← Scanned;
end
```

[Baker, 1992]

Move reachable objects in the heap to eliminate memory fragmentation

After identifying holes, relocate allocated objects at one end of the heap
Rest of the memory becomes a single free chunk

Two major approaches:

1. Mark-and-compact GC
2. Copying GC

The mark-and-compact collector follows:

1. Marking Phase: similar to the mark-and-sweep algorithms
2. Object Relocation:
   - Allocated regions of the heap are scanned
   - Address of each reachable object is computed from the low end of the heap
   - Addresses are stored in a structure named `NewLocation`
3. Object Copy:
   - Objects are copied to their new locations
   - References in the objects to point to are updated

Inputs : Root set of objects, heap, pointer marking the start of the free space (named `Free`)
Output : New value of pointer `Free`

```
begin
    /* MARKING PHASE                                                    */
    Unscanned ← copy_of(root);
    foreach o ∈ Unscanned do
        reached_bit[o] ← false;
    end
    while ∃ o ∈ Unscanned do
        Unscanned ← Unscanned \{o};
        foreach r ∈ references_in(o) do
            if neg reached_bit[r] then
                reached_bit[r] ← true;
                Unscanned ← Unscanned ∪{r};
            end
        end
    end
```

```
/* COMPUTE THE NEW LOCATIONS                                    */
NewLocation ← [ ];
Free ← first address in the heap; foreach c ∈ chunks[0..] do
    if reached_bit[c] then
        NewLocation[c] ← Free;
        Free ← Free size_of(c);
    end
end
```

```
    /* RETARGET THE REFERENCES AND MOVE REACHED OBJECTS                    */
    foreach c ∈ chunks[0..] do
        if reached_bit[c] then
            foreach r ∈ references_in(c) do
                c.r ← NewLocation[c.r]
            end
            Copy c to NewLocation[c];
        end
    end
    foreach r ∈ references_in(root) do
        r ← NewLocation[r]
    end
end
```

Copying GC reserves space to which the objects can move

- Memory space is partitioned into two semispaces A and B
- Mutator allocates in A until it fill up
- Mutator is stopped and GC copies the reachable objects to B
- When GC finished, the roles of A and B are reversed

Algorithm is proposed by C.J. Cheney [Cheney, 1970]

[Fenichel and Yochelson, 1969, Cheney, 1970]

- Basic Mark-and-sweep: Proportional to the number of chunks in heap
- Baker's Mark-and-sweep: Proportional to the number of reached objects
- Basic Mark-and-compact: Proportional to the number of chunks in the heap plus the total size of the reached objects
- Cheney's Copying Collector: Proportional to the total of the reached objects

Problem of the trace-based collectors

- Trace-based collectors do stop-the-world GC
- May introduce long pauses into execution of user programs

First Solution: Incremental Collection

Divide the work in time, by interleaving GC and mutation

Second Solution: Partial Collection

Divide the work in space, by collecting a subset of the garbage at a time

Incremental short-pause GC is conservative:

- While GC must not collect objects that are not garbage
- It does not have to collect all the garbage in each round

- Garbage left in memory is named floating garbage
- Incremental GC overestimates the set of reachable objects

1. Program's root set is processed automatically, without interference with the mutator
2. After finding the initial set of unscanned objects, the mutator's actions are interleaved with the tracing step
3. During this period, any of the mutator's actions that may change reachability are recorded succinctly, in a side table
4. Side table is used by GC to adjust the memory allocation when the mutator's actions resume their execution
5. If there is not enough memory space, GC blocks the mutator until it finished to collect garbage

Set of reachable objects when tracing finished is:

$$(R \cup New) \setminus Lost$$

- R: set of reachable objects at the beginning of garbage collection
- New: set of allocated objects during garbage collection
- $Lost$: set of objects that have become unreachable due to lost references

It is expensive to compute an object's reachability every time

- Incremental GC does not attempt to collect all garbage at the end of the tracing
- Every garbage left behind (floating garbage) should be a subset of the Lost objects

$$((R \cup New) \setminus Lost) \subseteq S \subseteq (R \cup New)$$

- First, tracing algorithm is used to find the upper bounds of $R \cup New$
- Mutator behavior is modified during the tracing:
  - All references that existed before GC are preserved
  - All objects created are considered reachable immediately and are placed in the Unscanned state
- This scheme is conservative and finds R and New

But the cost is high because the algorithm intercept all the write operations and remembers all the overwritten references

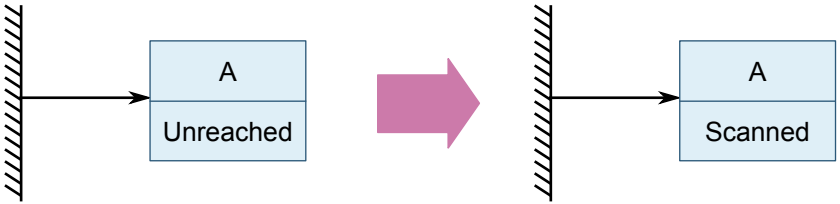
Following slides proposes a solution

If mutator and tracing GC algorithm are interleaved, then some reachable objects may be misclassified as unreachable

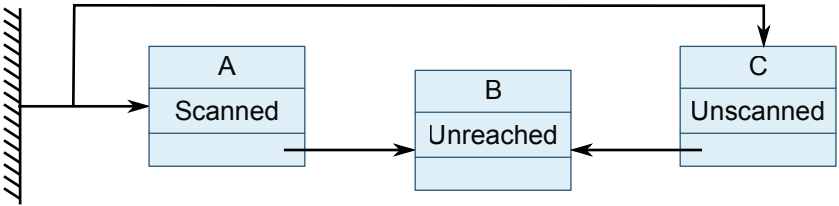
Because mutator may violate the following invariant of GC algorithm: Scanned object can only contain references to other scanned or unscanned objects, never unreached objects

[Dijkstra et al., 1978]

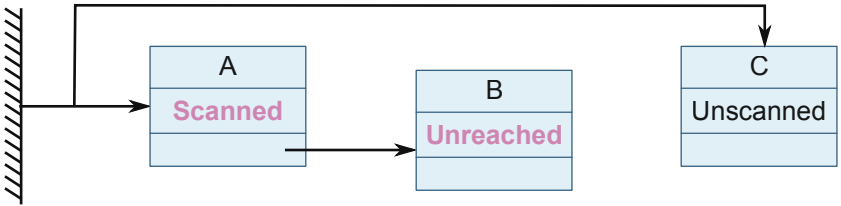
- GC finds reachable object A and scans pointers within A, thereby putting A in the Scanned state
- Mutator stores a reference to an Unreached (but reachable) object B into the Scanned object A. It does by copying a reference to B from an object C that is currently in the Unreached or Unscanned state
- Mutator loses reference to B in object C. It may have overwritten C's reference to B before the reference is scanned, or C may have become unreachable and never have reached the Unscanned state to have its reference scanned



- GC finds reachable object A and scans pointers within A, thereby putting A in the Scanned state
- Mutator stores a reference to an Unreached (but reachable) object B into the Scanned object A. It does by copying a reference to B from an object C that is currently in the Unreached or Unscanned state
- Mutator loses reference to B in object C. It may have overwritten C's reference to B before the reference is scanned, or C may have become unreachable and never have reached the Unscanned state to have its reference scanned



- GC finds reachable object A and scans pointers within A, thereby putting A in the Scanned state
- Mutator stores a reference to an Unreached (but reachable) object B into the Scanned object A. It does by copying a reference to B from an object C that is currently in the Unreached or Unscanned state
- Mutator loses reference to B in object C. It may have overwritten C's reference to B before the reference is scanned, or C may have become unreachable and never have reached the Unscanned state to have its reference scanned



- Write Barriers:
  - Intercepts writes of references into a Scanned object A
  - When the reference is to an Unreached object B, then classify the object B as reachable and place it in an Unscanned state
  - or put the object A in an Unscanned state
- Read Barriers:
  - Intercept the reads of references in Unreached or Unscanned objects
  - When the mutator reads reference to object A from an object in Unreached or Unscanned state, classify A as reachable and put it in the Unscanned state
- Transfer Barriers:
  - Intercept the loss of original reference in an Unreached or Unscanned object
  - When the mutator overwrites a reference in an Unreached or Unscanned object, save the reference being overwritten, classify it as reachable, and place the reference itself in the Unscanned state

- Write barriers: the most efficient of the barriers
- Read barriers: more expensive because typically there are many more reads than there are writes
- Transfer barriers: not competitive; because many objects "die young," this approach would retain many unreachable objects

Fact: "Objects typically die young"

- These objects becomes unreachable before the GC is invoked
- $\Rightarrow$ GC is cost effective with the approaches presented in the previous slides
- The same "mature" objects were found and copied at every round of the GC

Two major approaches for effective GC

- Generational GC
- Train Algorithm

Structure

Heap is divided in partitions: 0, 1, ..., n
(0 is for the younger data)

Behavior

- Objects are created in partition 0
- When the partition 0 fills up, this participation is GC and the reachable objects are moved in partition 1
- Same algorithm for partition 2 and 3, ... $n-1$ and n

[Lieberman and Hewitt, 1983]

Train algorithm uses fixed-length partitions, called cars

Car

Car is a single disk block, assuming there are no object larger than disk blocks
OR
Car size could be larger, but it is fixed once and for all

Train

Cars are organized into trains

No limit to the number of cars in a train
No limit to the number of trains

[Hudson and Moss, 1992]

Two approaches to collect the garbages

1. First car in lexicographic order is collected in one incremental garbage-collection step:
   - Step similar to collection of the first partition in the generational algorithm, since a "remembered" list of all points from outside the car is maintained
   - Objects with no references at all are identified, as well as garbage cycles that are contained completely within this car
   - Reachable objects in the car are always moved to some other car, so each garbage-collected car becomes empty and can be removed from the train
2. Sometimes, the first train has no external reference
   - There are no pointer from the root set to any car of the train, and the remembered sets for the cars contain only references from other cars in the train, not from other trains
   - In this situation, the train is a huge collection of cyclic garbage, and we delete the entire train

Two approaches to collect the garbages

1. First car in lexicographic order is collected in one incremental garbage-collection step:
   - Step similar to collection of the first partition in the generational algorithm, since a "remembered" list of all points from outside the car is maintained
   - Objects with no references at all are identified, as well as garbage cycles that are contained completely within this car
   - Reachable objects in the car are always moved to some other car, so each garbage-collected car becomes empty and can be removed from the train
2. Sometimes, the first train has no external reference
   - There are no pointer from the root set to any car of the train, and the remembered sets for the cars contain only references from other cars in the train, not from other trains
   - In this situation, the train is a huge collection of cyclic garbage, and we delete the entire train

- Generational GC works most frequently on the area of the heap that contains the youngest objects
  It tends to collect a lot of garbage for relatively little work
- Train algorithm does not spend a large proportion of time on young objects
  It does limit the pauses due to garbage collection
  Advantage is that we never have to do a complete garbage collection, as we do occasionally for generational garbage collection
- Good combination of strategies is to use generational collection for young objects, and once heap becomes sufficiently mature, to "promote" it to a separate heap that is managed by the train algorithm

- Run-time Organization: To implement the abstractions embodied in the source language, a compiler creates and manages a run-time environment in concert with the operating system and the target machine. The run-time environment has static data areas for the object code and the static data objects created at compile time. It also has dynamic stack and heap areas for managing objects created and destroyed as the target program executes
- Control Stack: Procedure calls and returns are usually managed by a run-time stack called the control stack. We can use a stack because procedure calls or activations nest in time; that is, if p calls q, then this activation of q is nested within this activation of p

- Stack Allocation: Storage for local variables can be allocated on a run-time stack for languages that allow or require local variables to become inaccessible when their procedures end. For such languages, each live activation has an activation record (or frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack
- Access to Nonlocal Data on the Stack: For languages like C that do not allow nested procedure declarations, the location for a variable is either global or found in the activation record on top of the run-time stack. For languages with nested procedures, we can access nonlocal data on the stack through access links, which are pointers added to each activation record. The desired nonlocal data is found by following a chain of access links to the appropriate activation record. A display is an auxiliary array, used in conjunction with access links, that provides an efficient short-cut alternative to a chain of access links

- Heap Management: The heap is the portion of the store that is used for data that can live indefinitely, or until the program deletes it explicitly. The memory manager allocates and deallocates space within the heap. Garbage collection finds spaces within the heap that are no longer in use and can therefore be reallocated to house other data items. For languages that require it, the garbage collector is an important subsystem of the memory manager
- Exploiting Locality: By making good use of the memory hierarchy, memory managers can influence the run time of a program. The time taken to access different parts of memory can vary from nanoseconds to milliseconds. Fortunately, most programs spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. A program has temporal locality if it is likely to access to same memory locations again soon; it has spatial locality if it is likely to access nearby memory locations soon

- Reducing Fragmentation: As the program allocates and deallocates memory, the heap may get fragmented, or broken into large numbers of small noncontiguous free spaces or holes. The best fit strategy (allocate the smallest available hole that satisfies a request) has been found empirically to work well. While best fit tends to improve space utilization, it may not be best for spatial locality. Fragmentation can be reduced by combining or coalescing adjacent holes
- Manual Deallocation: Manual memory management has two common failings: not deleting data that can not be referenced is a memory-leak error, and referencing deleted data is a dangling-pointer-reference error
- Reachability: Garbage is data that cannot be referenced or reached. There are two basic ways of finding unreachable objects: either catch the transition as a reachable object turns unreachable, or periodically locate all the reachable objects and infer that all remaining objects are unreachable

- Reference-Counting Collectors: maintain a count of the references to an object; when the count transitions to zero, the object becomes unreachable. Such collectors introduce the overhead of maintaining references and can fail to find "cyclic" garbage, which consists of unreachable objects that reference each other, perhaps through a chain of references
- Trace-Based Garbage Collectors: iteratively examine or trace all references to find reachable objects, starting with the root set consisting of objects that can be accessed directly without having to dereference any pointers
- Mark-and-Sweep Collectors: visit and mark all reachable objects in a first tracing step and then sweep the heap to free up unreachable objects
- Mark-and-Compact Collectors: improve upon mark-and-sweep; they relocate objects in the heap to eliminate memory fragmentation

- Copying Collectors: break the dependency between tracing and finding free space. They partition the memory into two semispaces, A and B. Allocation requests are satisfied from one semispace, say A, until it fills up, at which point the garbage collector takes over, copies the reachable objects to the other space, say B, and reverses the roles of the semispaces
- Incremental Collectors: Simple trace-based collectors stop the user program while garbage is collected. Incremental collectors interleave the actions of the garbage collector and the mutator or user program. The mutator can interfere with incremental reachability analysis, since it can change the references within previously scanned objects. Incremental collectors therefore play it safe by overestimating the set of reachable objects; any "floating garbage" can be picked up in the next round of collection

- Partial Collectors: also reduce pauses; they collect a subset of the garbage at a time. The best known of partial-collection algorithms, generational garbage collection, partitions objects according to how long they have been allocated and collects the newly created objects more often because they tend to have shorter lifetimes. An alternative algorithm, the train algorithm, uses fixed length partitions, called cars, that are collected into trains. Each collection step is applied to the first remaining car of the first remaining train. When a car is collected, reachable objects are moved out to the other cars, so this car is left with garbage and can be removed from the train. These two algorithms can be used together to create a partial collector that applies the generational algorithm to younger objects and the train algorithm to more mature objects

Baker, Jr, H. (1992).
The treadmill: real-time garbage collection without motion sickness.
ACM SIGPLAN Notices, 27(3):66–70.

Cheney, C. (1970).
A nonrecursive list compacting algorithm.
Comm. ACM, 13(11):677–678.

Dijkstra, E. (1960).
Recursive programming.
Numerishe Math., 2:312–318.

Dijkstra, E., Lamport, L., Martin, A., Scholten, C., and Steffens, E. (1978).
On-the-fly garbage collection: an exercise in cooperation.
Comm. ACM, 21(11):966–975.

Fenichel, R. and Yochelson, J. (1969).
A lisp garbage-collector for virtual memory computer systems.
Comm. ACM, 12(11):611–612.

Hudson, R. and Moss, J. (1992).
Incremental collection of mature objects.
In *Intl. Workshop on Memory Management, Lecture Notes in Computer Science*, number 637, pages 388–403.

Johnson, S. and Ritchie, D. (1981).
The C language calling sequence.
Computing Science Technical Report 102, Bell Laboratories, Murray Hill, NJ.

Knuth, D. (1968).
Art of Computer Programming, Fundamental Algorithms, volume 1.
Addison-Wesley, Boston, MA.

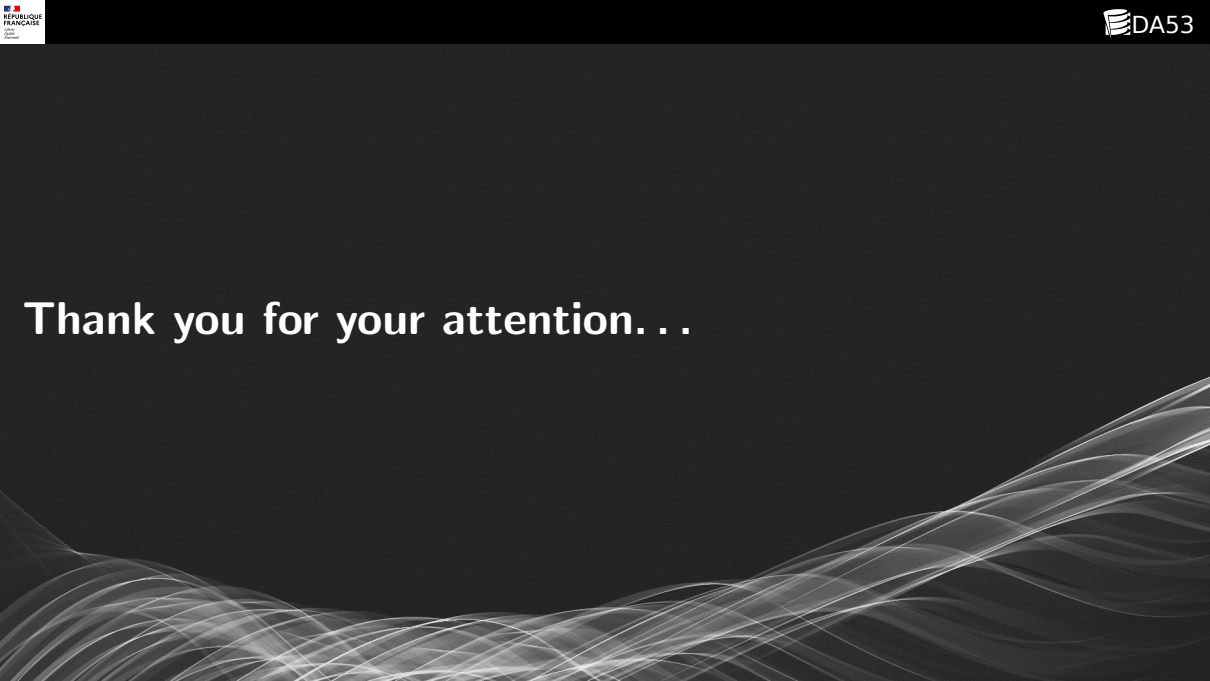
Lieberman, H. and Hewitt, C. (1983).
A real-time garbage collector based on the lifetimes of objects.
Comm. ACM, 26(6):419–429.

McCarthy, J. (1960).
Recursive functions of symbolic expressions and their computation by machine.
Comm. ACM, 3(4):184–195.

Randell, B. and Russel, L. (1964).
ALGOL 60 Implementation.
Academic Press.

Wilson, P. (1994).
Uniprocessor garbage collection techniques.

Thank you for your attention...



Appendix

History

2012-2021: Slides for the module LO46
Since 2021: Renaming LO46 to DA53

Sources

The LaTeX code of this document is available at
`https://github.com/gallandarakhneorg/da53-lessons`

Generation

This document is generated January 9, 2023 with the following tools:

- LaTeX
- Beamer
- CIAD style for Beamer (version 2022/02/10) — `https://github.com/gallandarakhneorg/tex-templates`
- AutoLaTeX — `http://www.arakhne.org/autolatex`

Full Professor

Université de Technologie de Belfort-Montbéliard
Université de Bourgogne-Franche-Comté, France

Deputy Director of the CIAD Laboratory

Topics: Multiagent systems, Agent-based simulation, Agent-oriented software engineering, Mobility and traffic modeling

Web page: `http://www.ciad-lab.fr/author-10836/`
Email: stephane.galland@utbm.fr

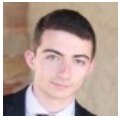
Open-source contributions:

- `http://www.sarl.io`
- `http://www.janusproject.io`
- `http://www.aspecs.org`
- `http://www.arakhne.org`
- `https://github.com/gallandarakhneorg/`

Computer Science Department

Université de Technologie de Belfort-Montbéliard

Université de Bourgogne-Franche-Comté, France



Web page: `https://therolf.fr`
Email: yann.le-vagueres@utbm.fr

Contributions:

- In this document: fixing of issues in the text and examples.
- Open Source: `https://github.com/TheRolfFR`