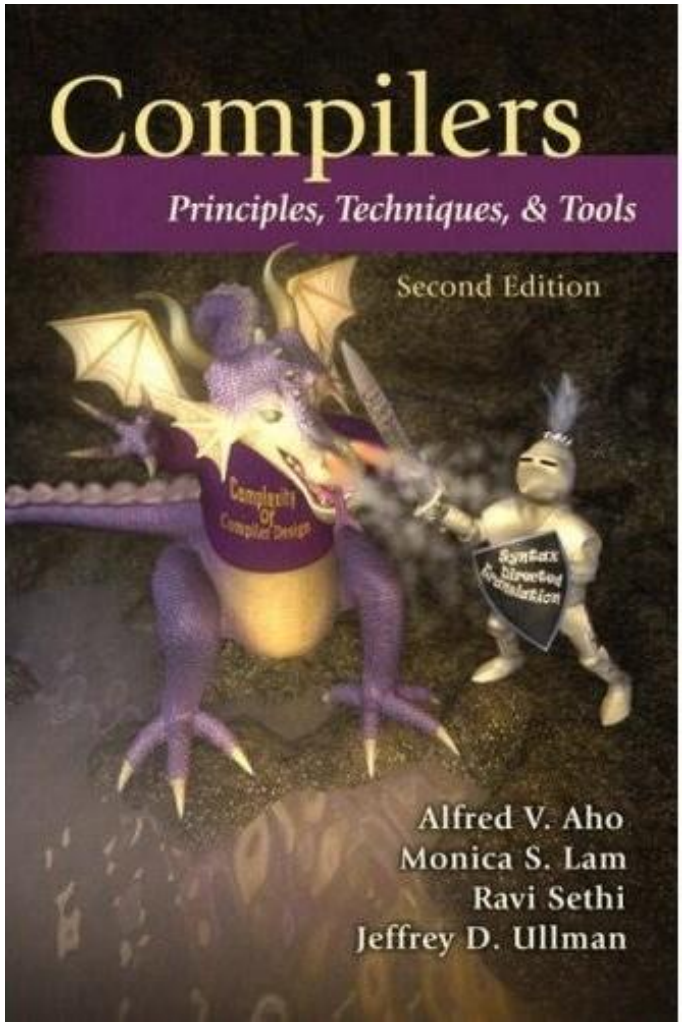
CSE 303 (Compilers)

Basic Concepts

Rubyeat Islam

Assistant Professor
Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST)
Mirpur Cantonment, Dhaka-1216, Bangladesh

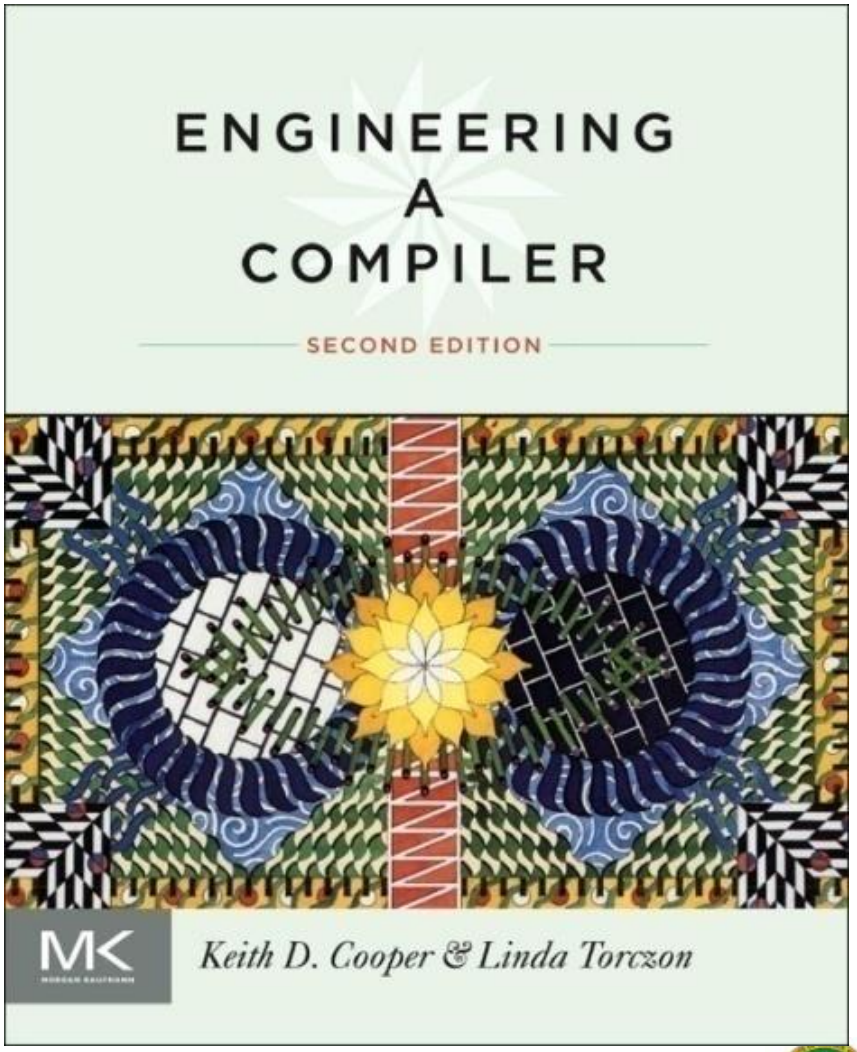
References



- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman
  Compilers: Principles, Techniques & Tools, Second edition.
  Pearson/Addison Wesley, 2007.

References

- Linda Torczon and Keith Cooper Engineering A Compiler, Second edition. Morgan Kaufmann Publishers Inc., 2011.



Class Tests

- This is a 3.0 credit course.
- So, there will be 2 + 1 = 3 class tests.
- Out of these, best two will be selected.
- One Mid Term (It is compulsory).

Course Group and Website

- An email group (https://classroom.google.com/u/0/c/Njc1NzcwNjI4Mjk4) has been created for the course.
- Class code 55dexcu

- It is mandatory to become a member of the group.

Course Group and Website

- All notifications and intimations sent over the Google classroom shall be considered as official.
- At the same time, all notices and statistics published over the course web site shall be considered as official.

Introduction

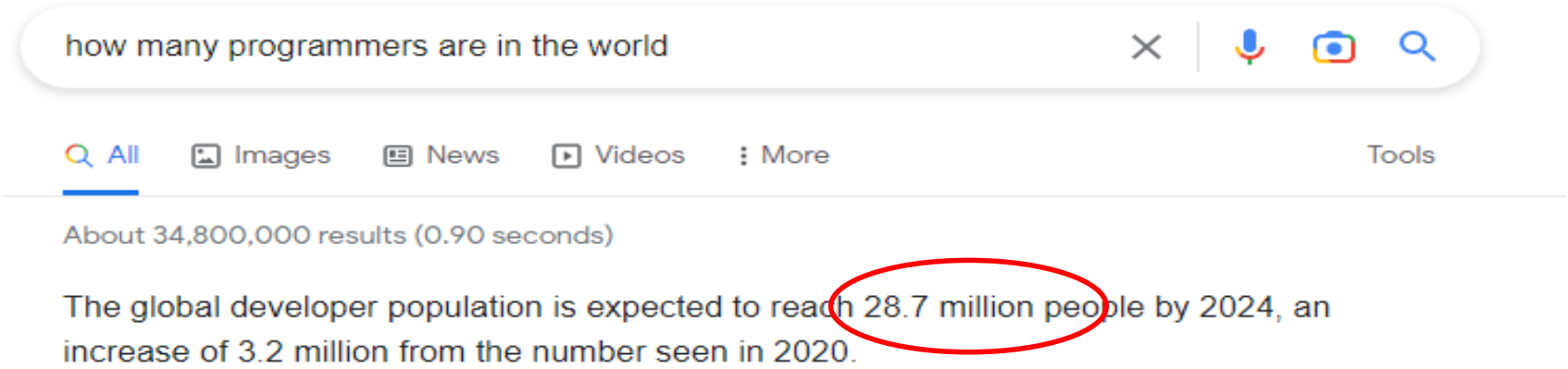
- To become a CSE Engineer, one should have basic three ideas:
  - How a programming language works, from the inside.
  - How a Compiler works.
  - How to build a new programming language.

Introduction

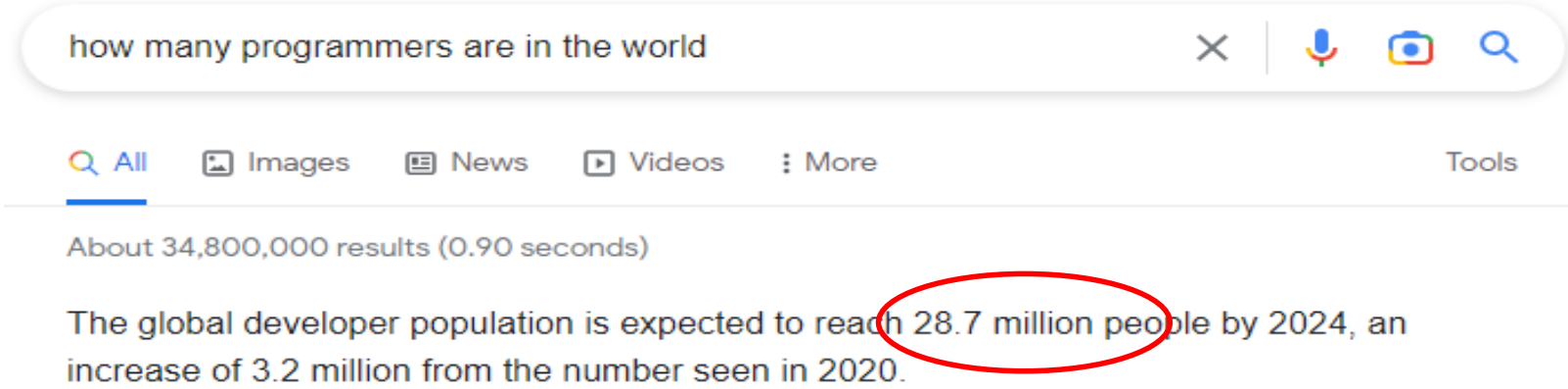
- To become a CSE Engineer, one should have basic three ideas:
  - How a programming language works, from the inside.
  - How a Compiler works.
  - How to build a new programming language.

The third idea appears to be really intriguing.
It's a good thing for you that we're diving deeper on this one.

Introduction



According to Google, there are about 28 Million software developers in the world. How many of them do you think would know how to create a new programming language?



According to Google, there are about 28 Million software developers in the world. How many of them do you think would know how to create a new programming language?

Very few!

Introduction

- Programmers are practical people:
  Will it build?
  Does it work? If so, then push.

Introduction

- Programmers are practical people:
  Will it build?
  Does it work? If so, then push.

- But, If you have a concept for creating a programming language that is significantly different from what is currently available, you should create your own compiler or interpreter.

- Perhaps you have imagined a new programming paradigm that you believe would be accepted by everyone, or a new syntax that you believe will be more expressive.

Introduction

- Programmers are practical people:
  Will it build?
  Does it work? If so, then push.

- But, If you have a concept for creating a programming language that is significantly different from what is currently available, you should create your own compiler or interpreter.

- Perhaps you have imagined a new programming paradigm that you believe would be accepted by everyone, or a new syntax that you believe will be more expressive.

So, here ART is blended with CODING!

Steps to create a programming language

- Seen from a very high perspective, three main processes are required to create a new programming language,
  - Define the grammar.
  - Build the front-end compiler for the source code.
  - Build the back-end code generator.

Steps to create a programming language

- Seen from a very high perspective, three main processes are required to create a new programming language,
  - Define the grammar.
  - Build the front-end compiler for the source code.
  - Build the back-end code generator.

The formal grammar defines what the source code of a program must look like, in this language.

For instance, in human language, sentence is made up of an object, a noun, and a verb,

It has to do with the syntactical conventions that programmers must Follow to in order to build programs in your language.

Steps to create a programming language

- Seen from a very high perspective, three main processes are required to create a new programming language,
  - Define the grammar.
  - Build the front-end compiler for the source code.
  - Build the back-end code generator.

The Front-End Compiler is a piece of software that takes the source code and produces some weird looking data structure.

More about this later in the course.

Steps to create a programming language

- Seen from a very high perspective, three main processes are required to create a new programming language,
  - Define the grammar.
  - Build the front-end compiler for the source code.
  - Build the back-end code generator.

Finally, the Back-End Code Generator is another piece of software that takes whatever was produced by the Front-End and creates a code that can actually be run.

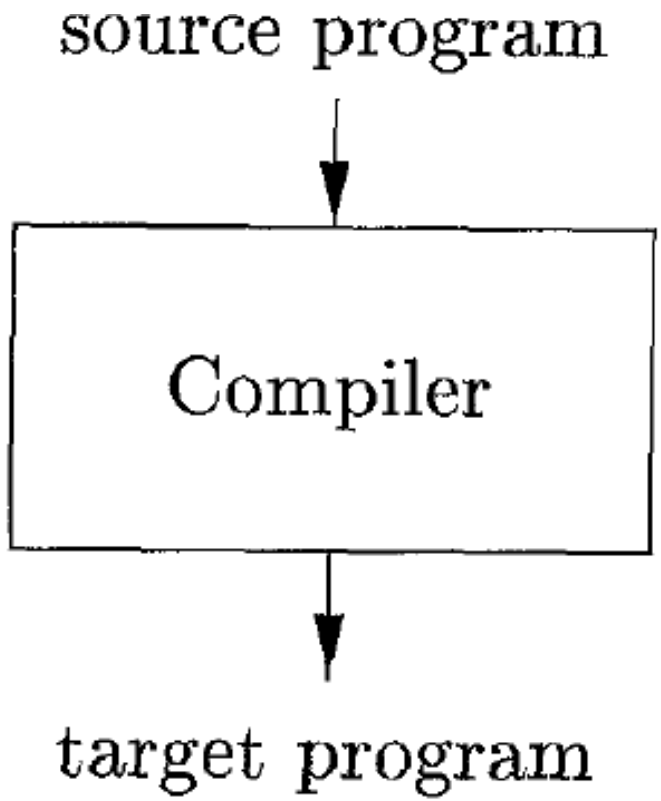
What's a compiler?

All computers only understand machine language

Therefore, high-level language instructions must be translated into machine language prior to execution

Language Processors- Compilers

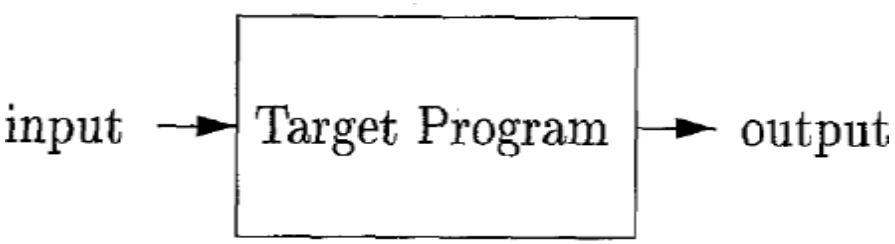
- A compiler is a program that can read a program in one language: the source language and translate it into an equivalent program in another language - the target language.
- An important role of the compiler is to report any errors in the source program that it detects during the translation process.



A compiler

Language Processors- Compilers

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



Running the target program

Language Processors- Compilers

program.c

```
while (c!='x')
   {
   if (c == 'a' || c == 'e' || c == 'i')
                                    printf("Congrats!");
   else
   if (c!='x')
                                    printf("You Loser!");
   }
```

Compiler

10000100011000101……

gcc -o prog program.c

Language Processors- Compilers

- After executing a C program file:

  hello.c → source code

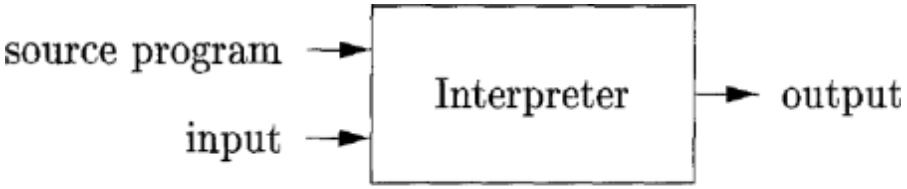
  hello.o → object file; contains function definitions in binary form

  hello.exe → linker/ executable file; links together a number of object files to produce a binary file which can be directly executed

- C language uses compiler

Language Processors — Interpreter

- An interpreter is another common kind of language processor.
- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.



An interpreter

Language Processors — Interpreter

- After executing a python program file:
  hello.py → source code
  hello.pyc → interpreted to bytecode; bytecode is executed by software called a virtual machine (VM)

- Python language does not use compilers.

Compiler vs Interpreter

Compiler works in two steps:

Step 1:

Step 2:

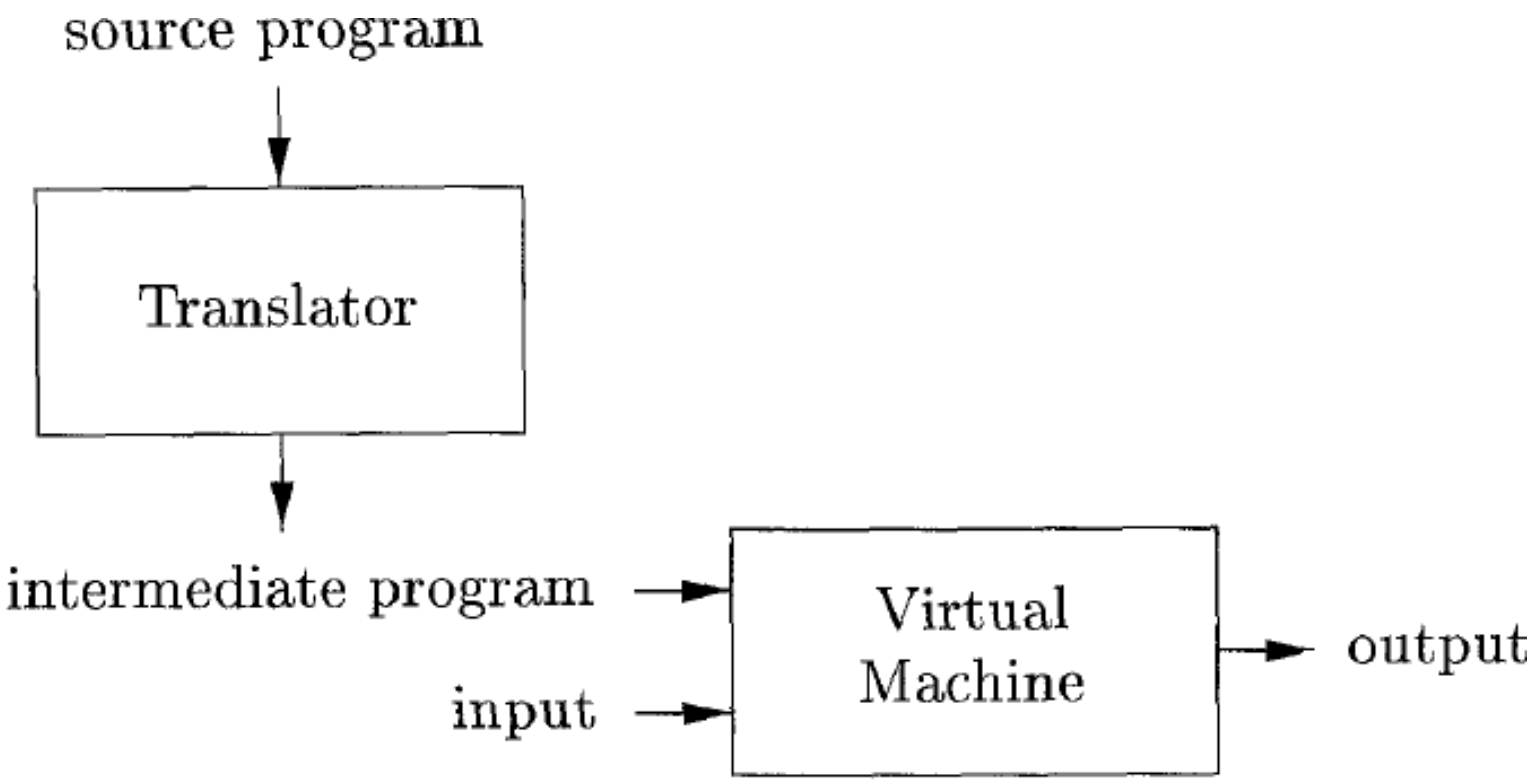
Interpreter works in one step:

Step 1:

Compiler vs Interpreter

- Compiler will scan the program at once and translate the whole program, interpreter will translate one by one statement.
- Compiler creates intermediate code, whereas interpreter does not.
- Thus compiler will require more memory to store the intermediate code.
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Hybrid Compiler

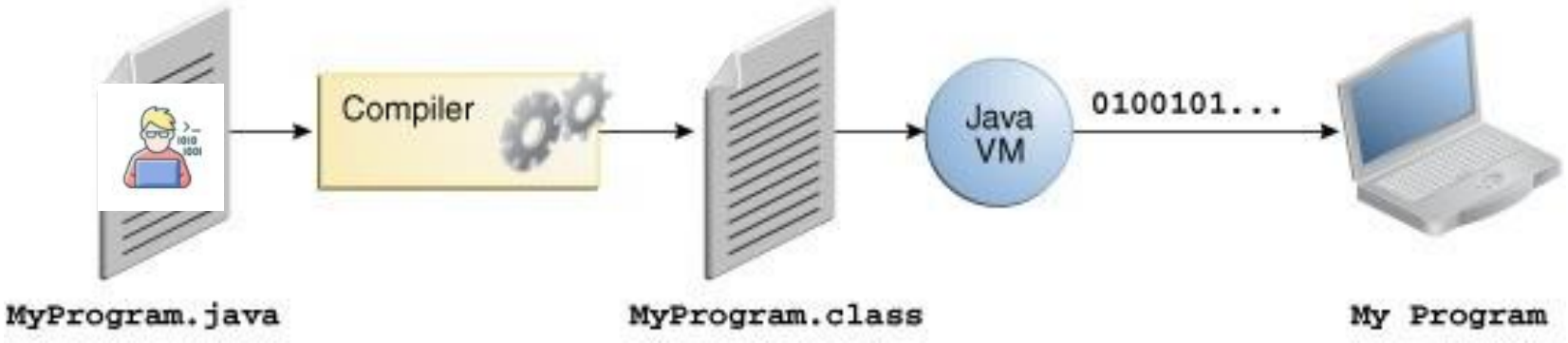


A hybrid compiler

- Java language processors combine compilation and interpretation.

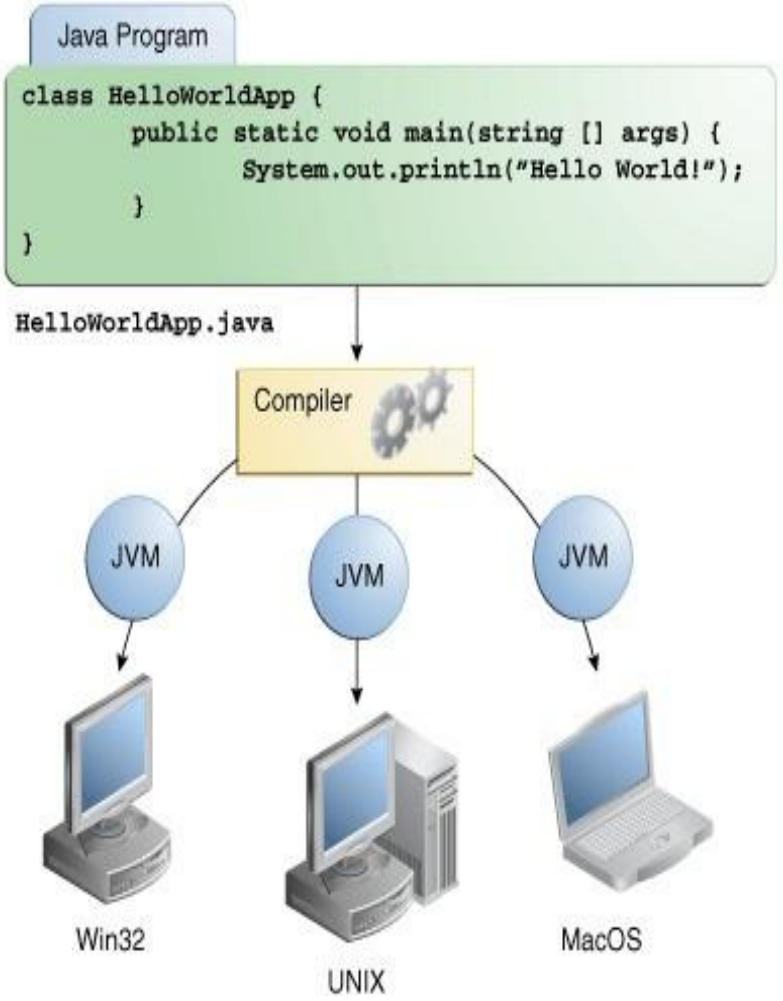
Hybrid Compiler

- Java language processors combine compilation and interpretation (hybrid compiler)

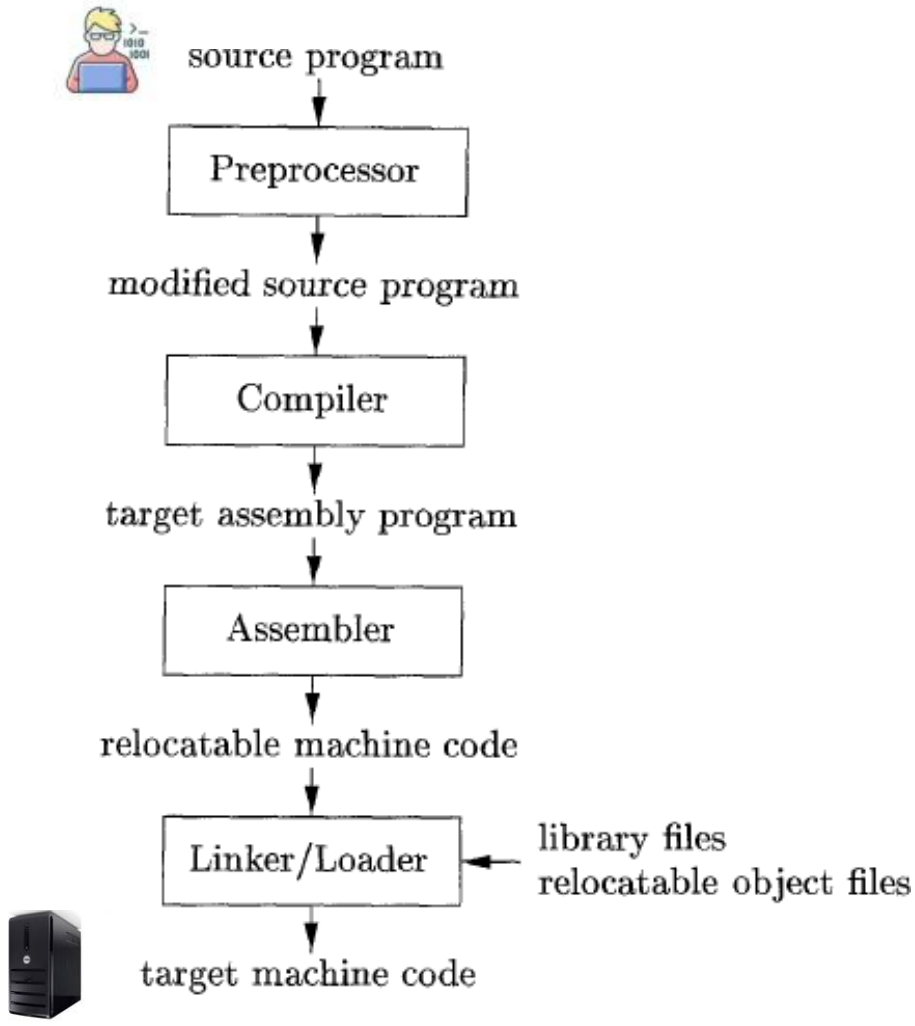


Hybrid Compiler

- Benefit: bytecodes compiled on one machine can be interpreted in another machine



Language Processors — *continued*



A language-processing system

Language Processors — *continued*

source program — High Level Language (HLL)

↓

Preprocessor

↓

modified source program — Pure HLL

↓

Compiler

↓

target assembly program — Mid level language

↓

Assembler — Machine dependent

↓

relocatable machine code — Binary object file (does not contain library files thus are not executable)

Can load the code at any point in the main memory

↓

Linker/Loader ← library files, relocatable object files

↓

target machine code

A language-processing system

Example- job of a linker

```c
#include<stdio.h>
#include "second.c"
int main()
{


    int a=10, b=5;
    int sum=add(a,b);
    printf("%d", sum);
}
```

first.c

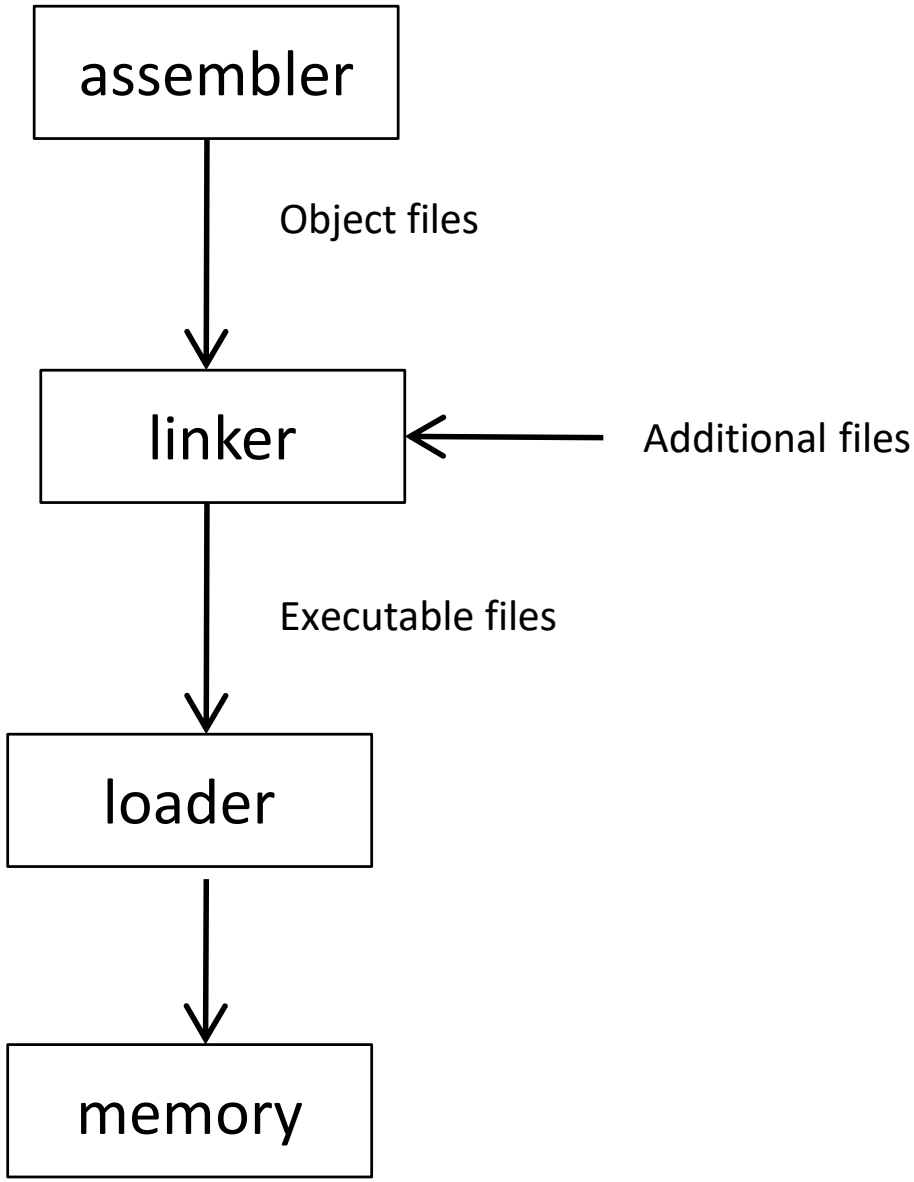
first.o

```c
int add(int x, int y)
{
    return x+y;
}
```

second.c

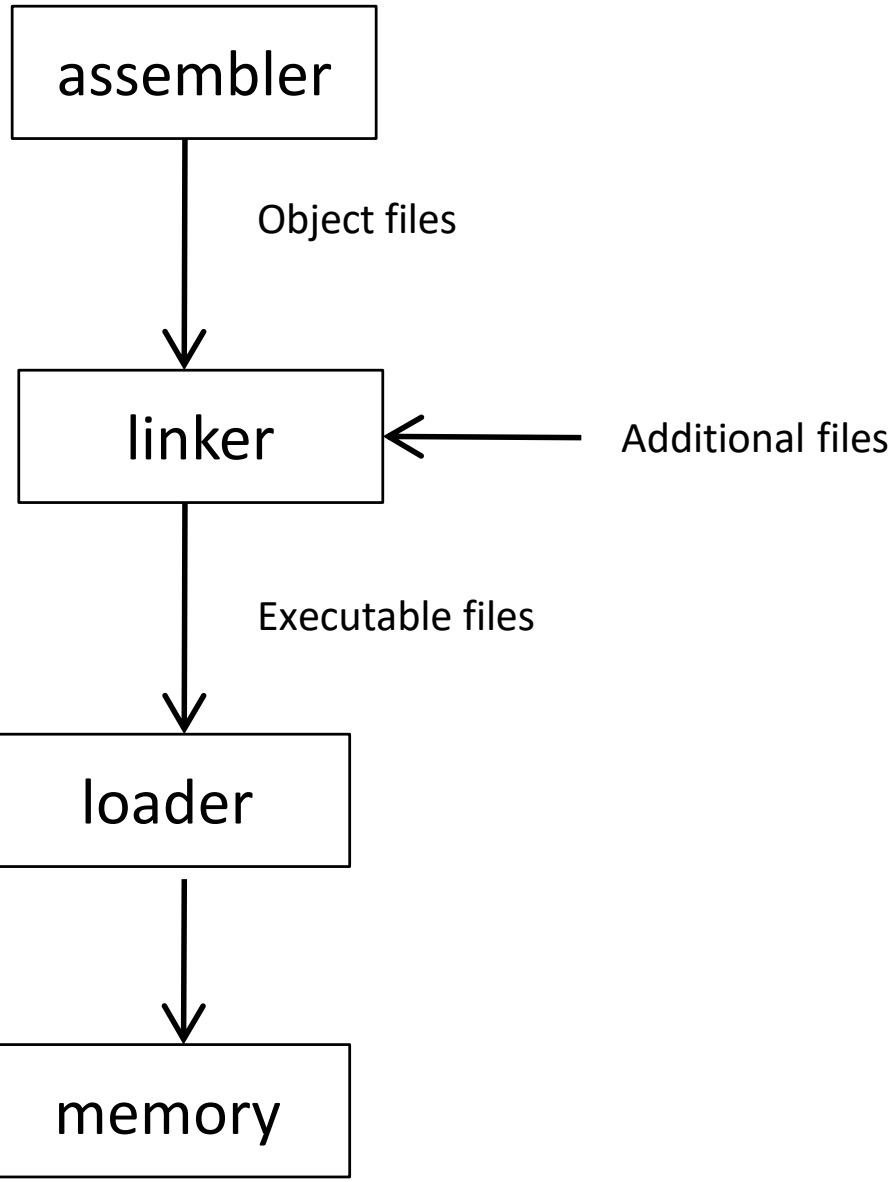
second.o

Linker and Loader



Linker and Loader

Now, state the difference between linker and loader..



Searches and appends all Library files and all the object files

Allocates memory for executable files

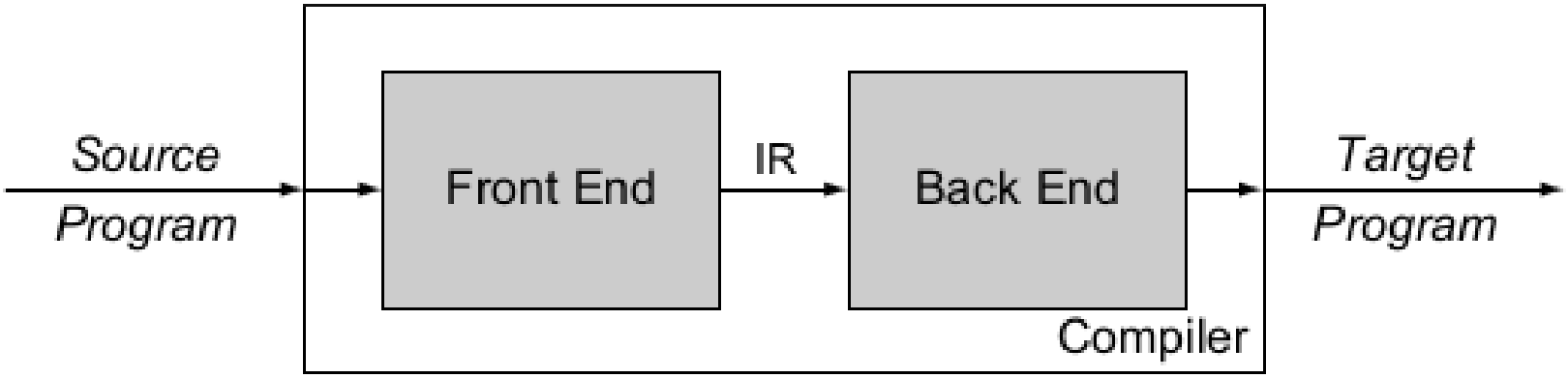
The Structure of a Compiler

- Two parts of compiler: **Analysis & Synthesis**

Analysis (Front end): checks for syntactic or semantic error; provide error message; stores information of source code in symbol table.

Synthesis (Back end): Constructs target program from Intermediate code and symbol table.

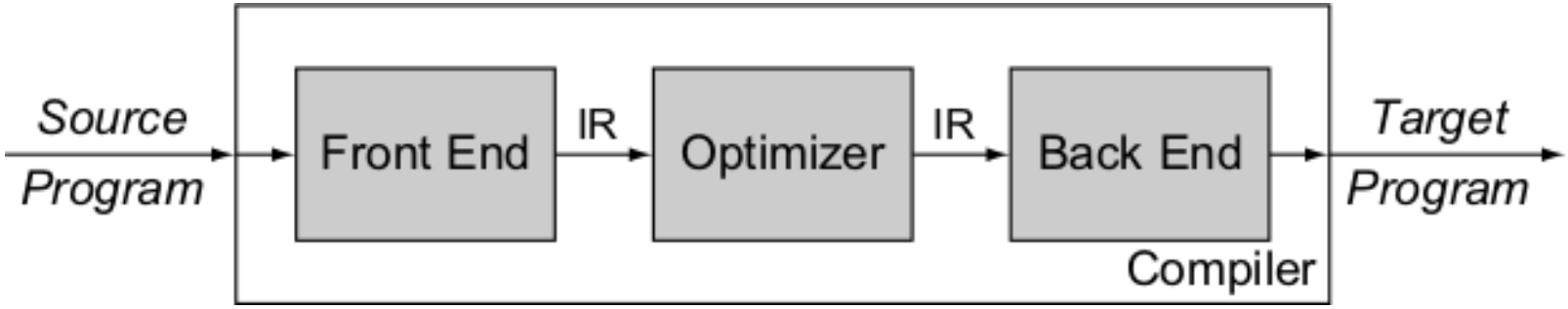
The Structure of a Compiler



Two pass compiler- benefits??

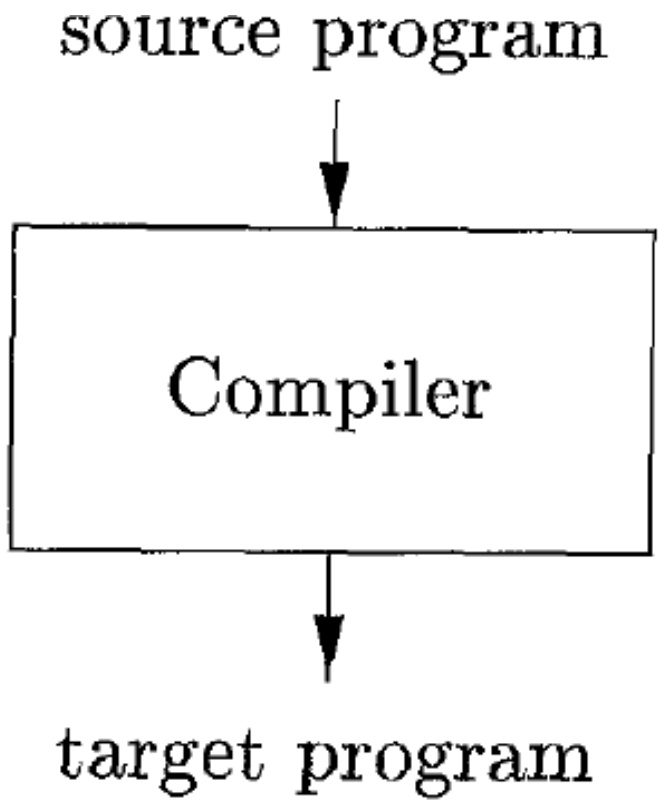
What is single pass compiler?

The Structure of a Compiler

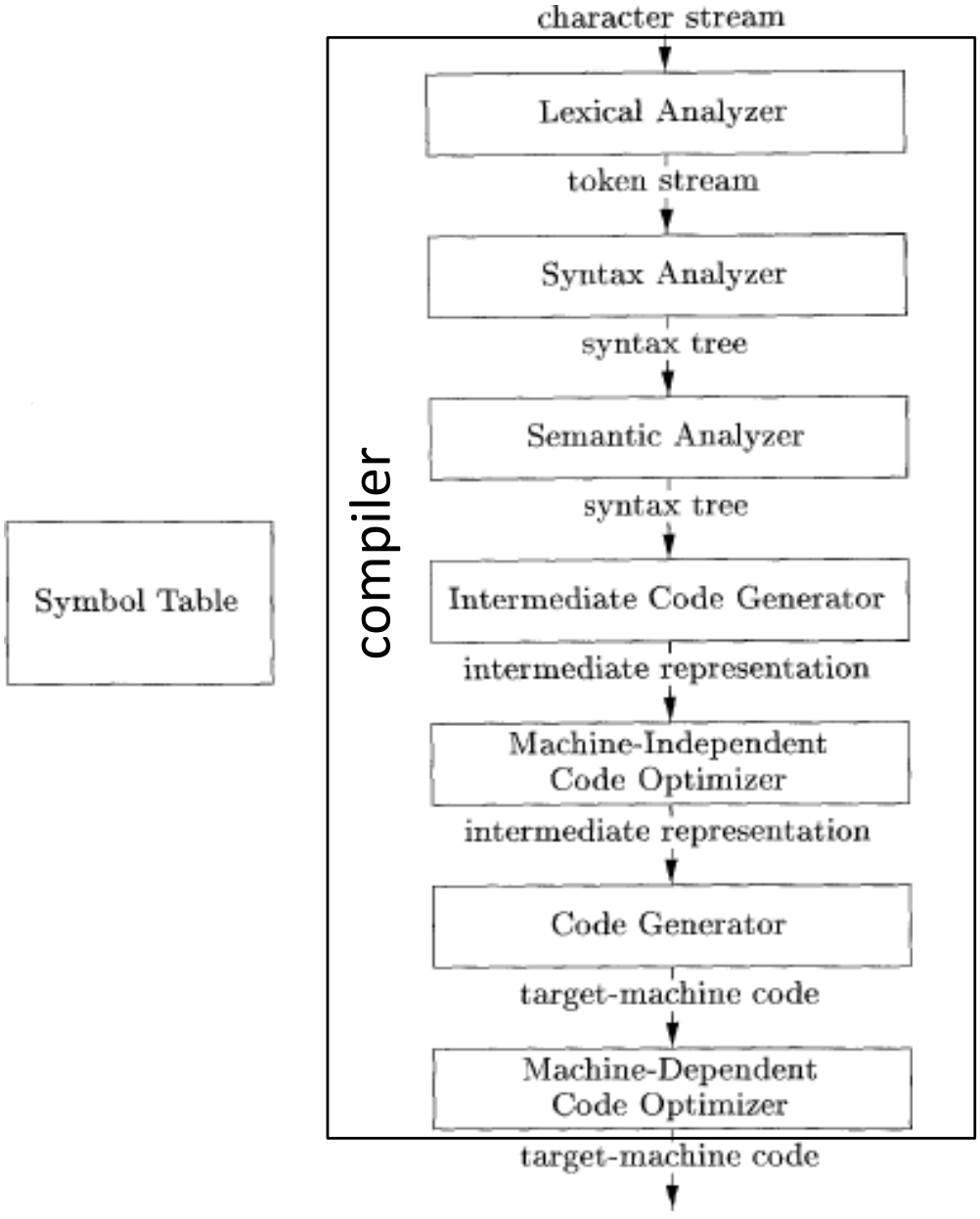


The Structure of a Compiler

- Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.



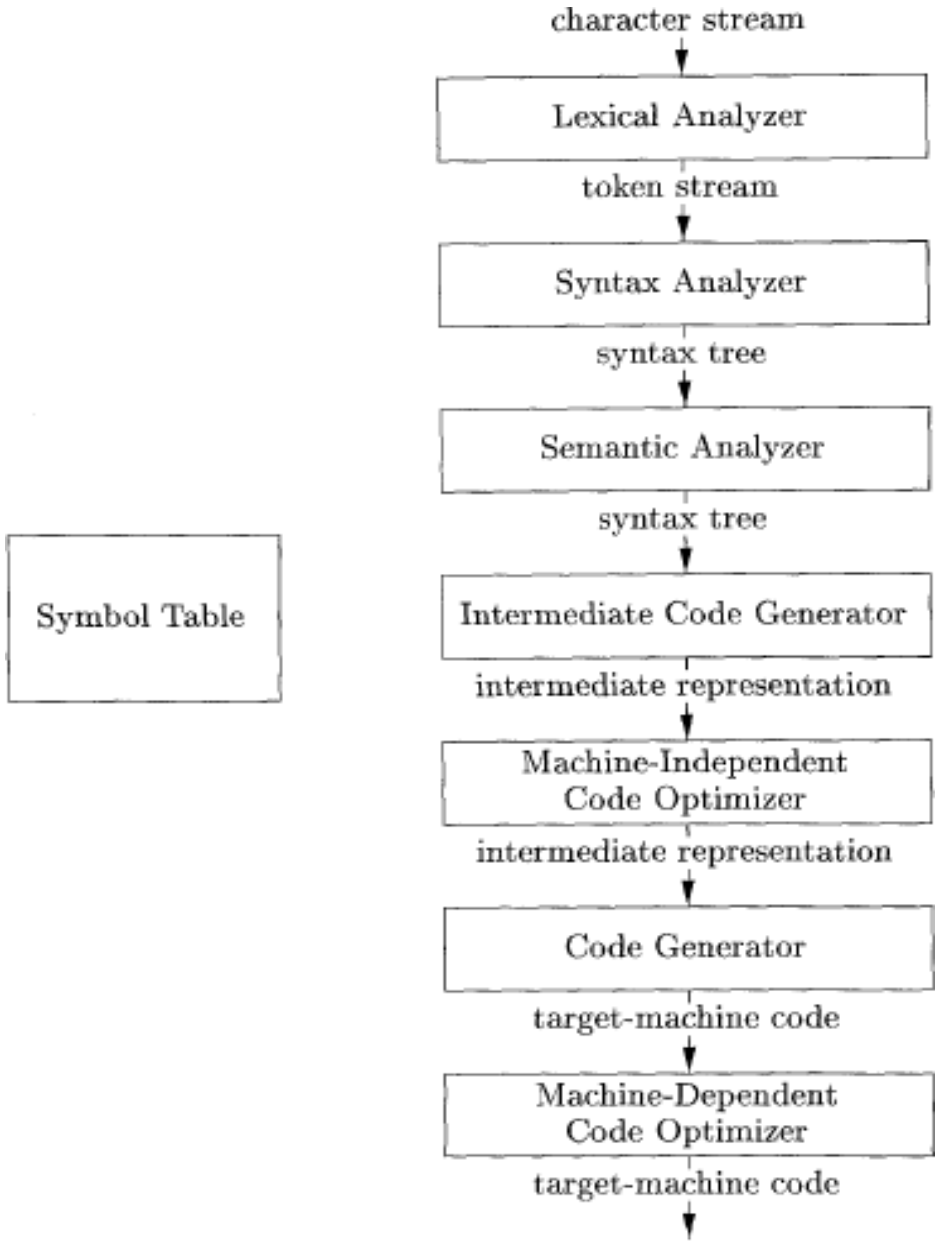
A compiler



Phases of a compiler

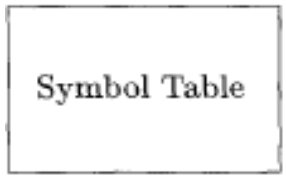
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table,

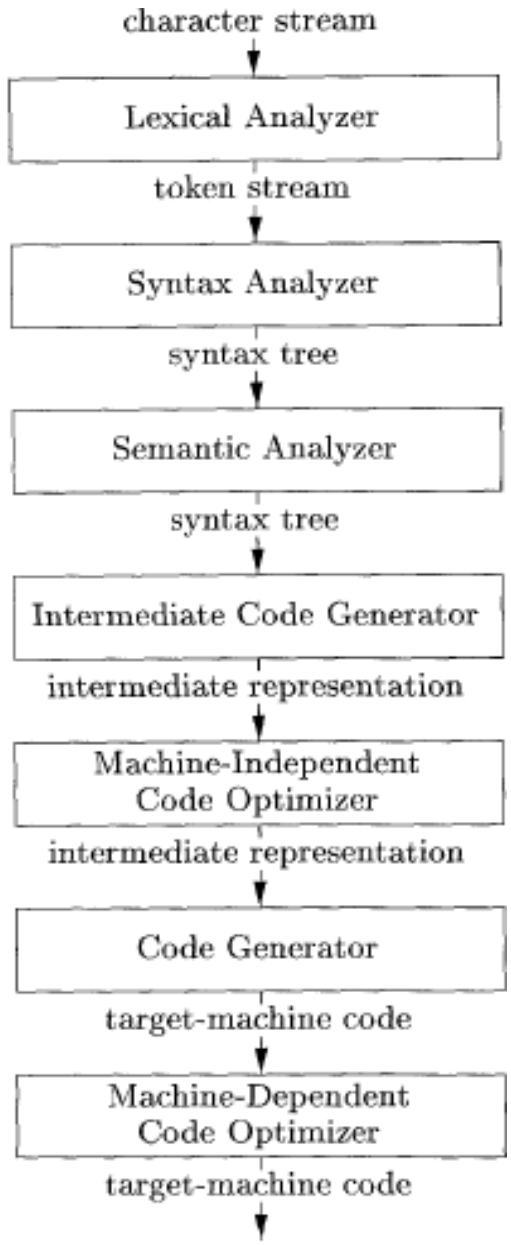
- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.



Phases of a compiler

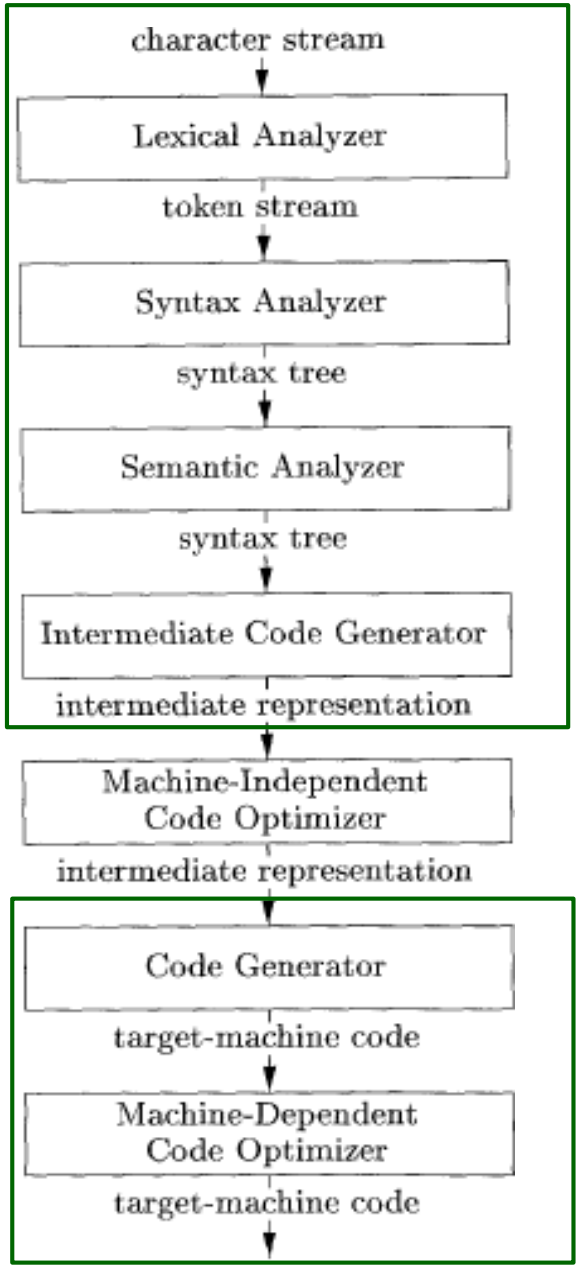
- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.





Phases of a compiler

Machine independent front end/ analysis part



Symbol Table

Machine dependent Back end/ synthesis part

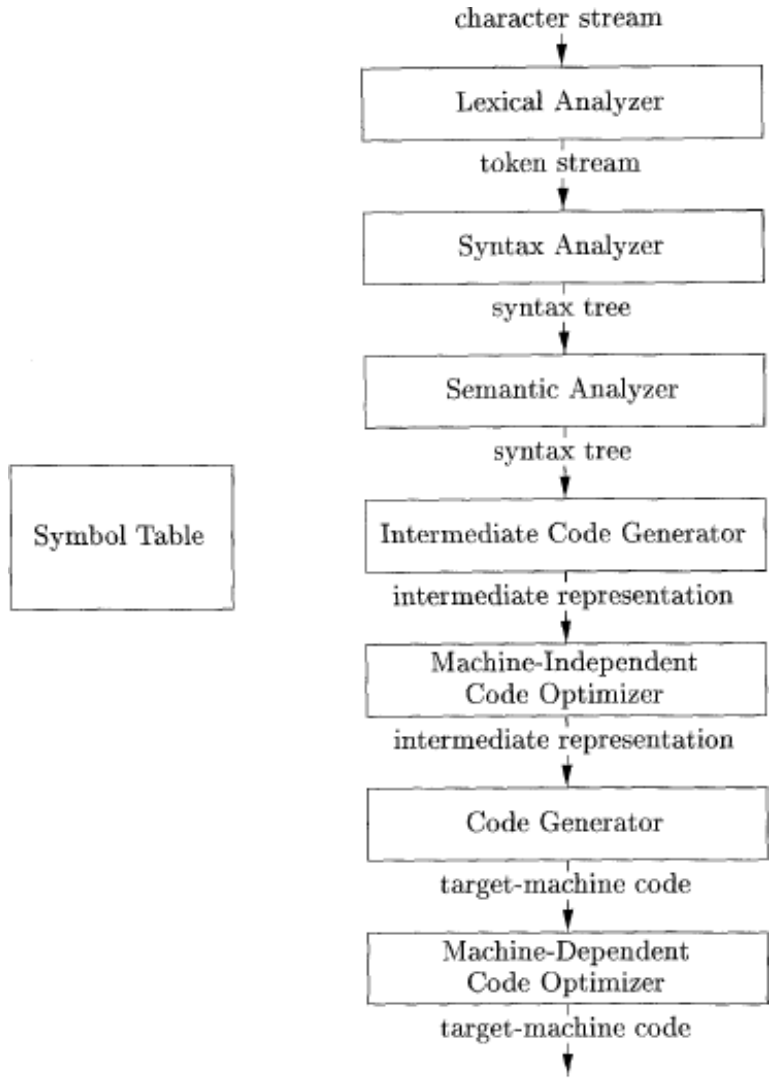
Phases of a compiler

Symbol Table

- To store the name of all entities in a structured form at one place
- To verify if a variable has been declared
- To determine the scope of a variable
- To implement type checking

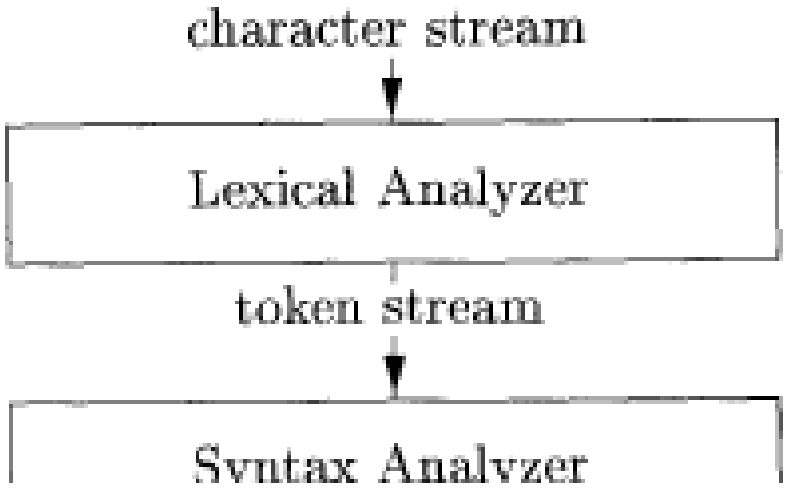
Lexical Analysis

- The first phase of a compiler is called lexical analysis or scanning.



Phases of a compiler

Lexical Analysis — *continued*



- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Lexical Analysis — *continued*

- For each lexeme, the lexical analyzer produces as output a token of the form

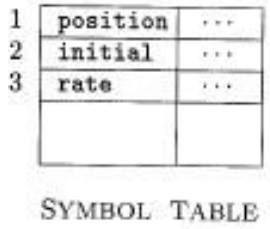
  $\langle$*token-name, attribute-value*$\rangle$

  that it passes on to the subsequent phase, syntax analysis.

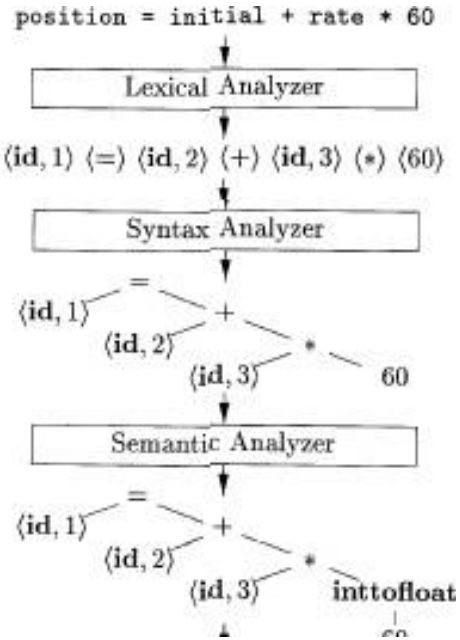
Lexical Analysis — *continued*

`position = initial + rate *60`

<id, 1> <=> <id, 2> <+> <id, 3> <∗><60>

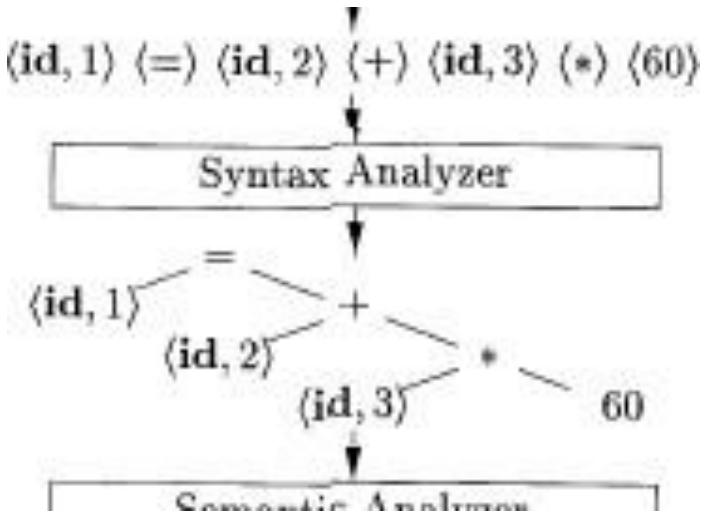


SYMBOL TABLE



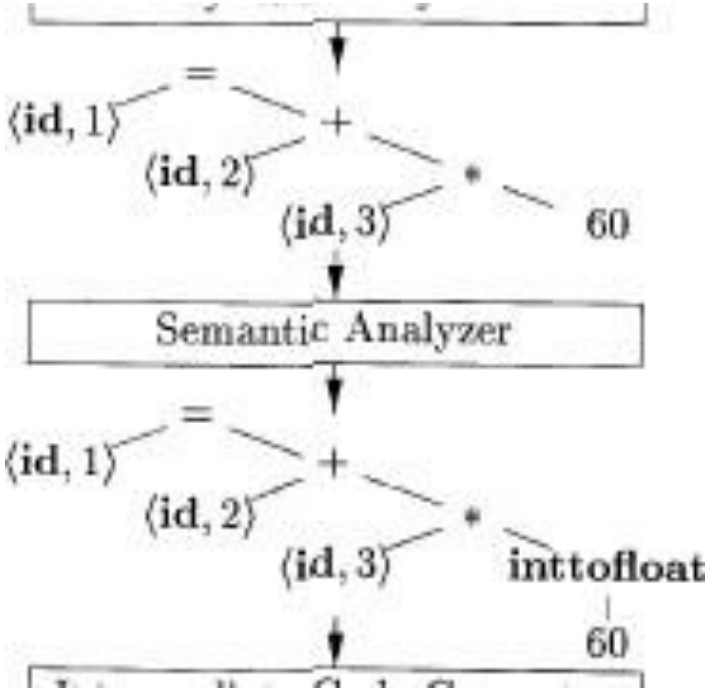
Syntax Analysis

- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.



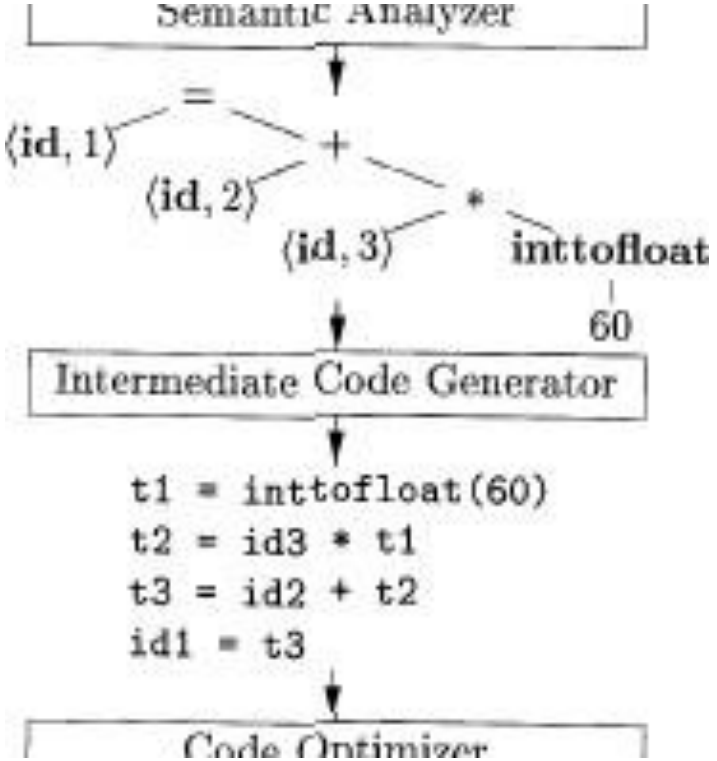
Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.



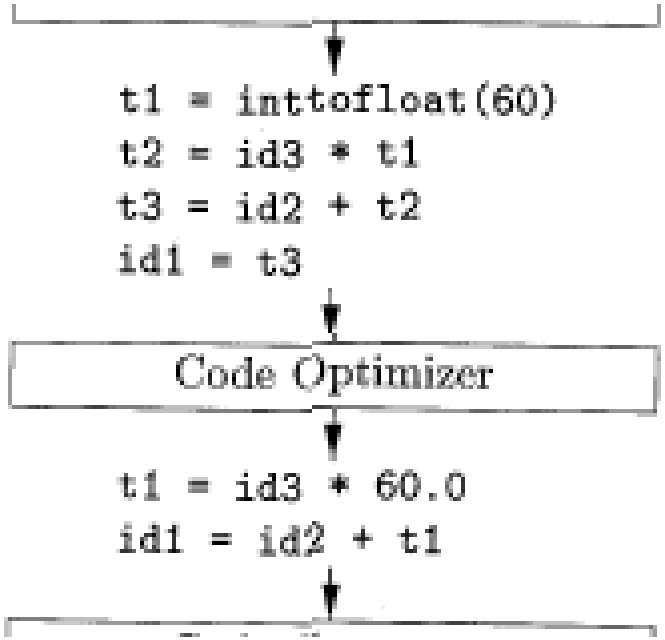
Intermediate Code Generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
- This intermediate representation should have two important properties:
  - it should be easy to produce
  - it should be easy to translate into the target machine.



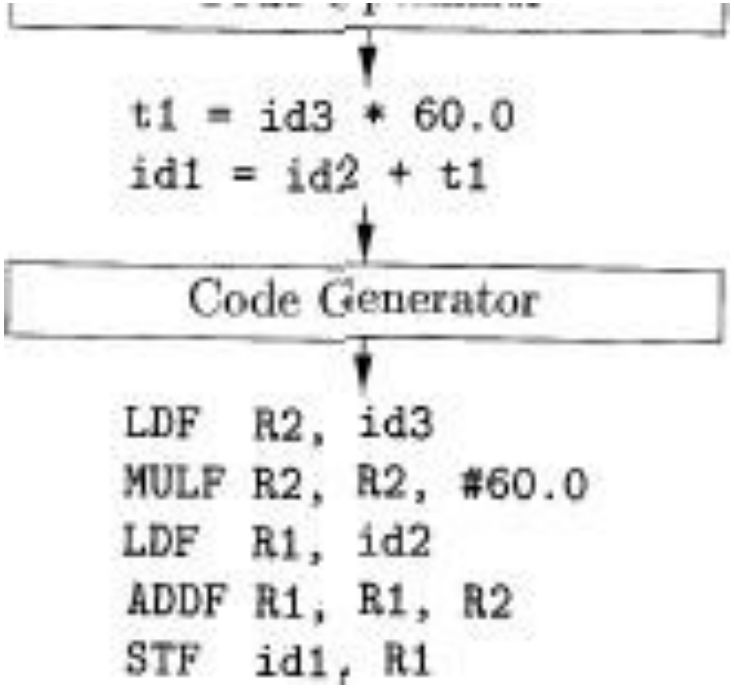
Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, shorter code, or target code that consumes less power.



Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.



Code Generation

- For example, using registers `R1` and `R2`, the intermediate code might get translated into the machine code

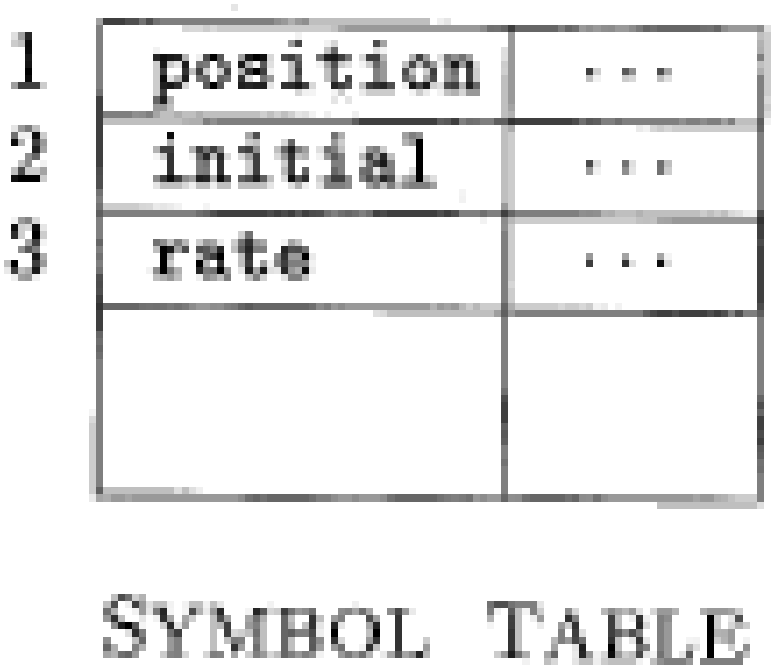
Position=initial+rate*60

```
t1 = id3 * 60.0
id1 = id2 + t1
```

LDF R1, id3
MULF R1, R1, #60.0
LDF R2, id2
ADDF R2, R2, R1
STF id1, R2

Symbol-Table Management

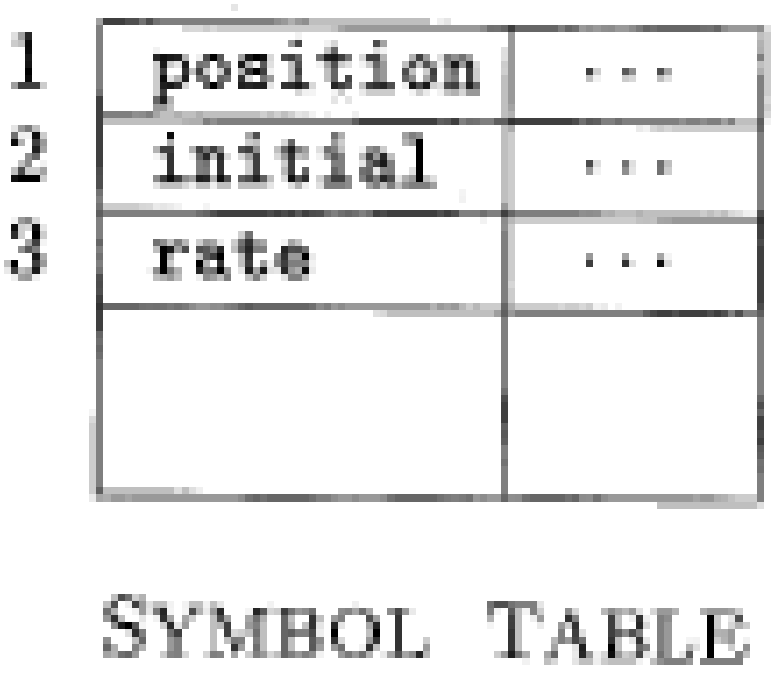
- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.



SYMBOL TABLE

Symbol-Table Management

- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used)
- In the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.



SYMBOL TABLE

The Grouping of Phases into Passes

- The front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- Code optimization might be an optional pass.
- Then there could be a back-end pass consisting of code generation for a particular target machine.

The Grouping of Phases into Passes — *continued*

- Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine.
- With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine.
- Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

Compiler-Construction Tools

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on.
- In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

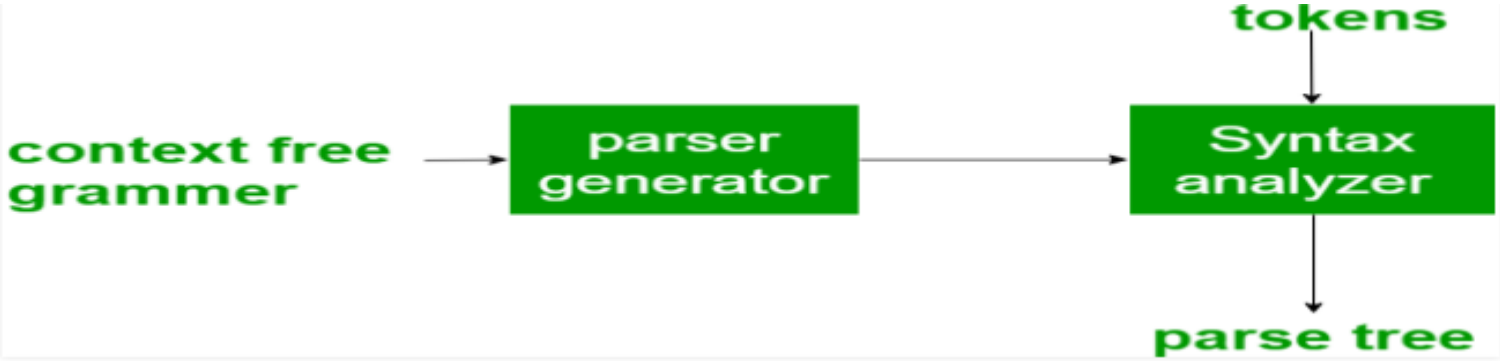
Compiler-Construction Tools

1. **Scanner generators** - produce lexical analyzers from a regular-expression description of the tokens of a language.

Example: Lex

2. **Parser generators** - It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar.

Example: PIC, EQM



Compiler-Construction Tools

3. Syntax directed translation engines – It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code.

4. Data-flow analysis engines – It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.

5. Automatic code generators – It generates the code generator to create machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator.

6. Compiler construction toolkits – provide an integrated set of routines for constructing various phases of a compiler.

Thank You

CSE 303 (Compilers)

Lexical Analysis

Rubyeat Islam
Assistant Professor
Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST)
Mirpur Cantonment, Dhaka-1216, Bangladesh

Courtesy: Dr. Muhammad Masroor Ali, Professor, BUET and Lec Tasmiah Tamzid Anannya, MIST

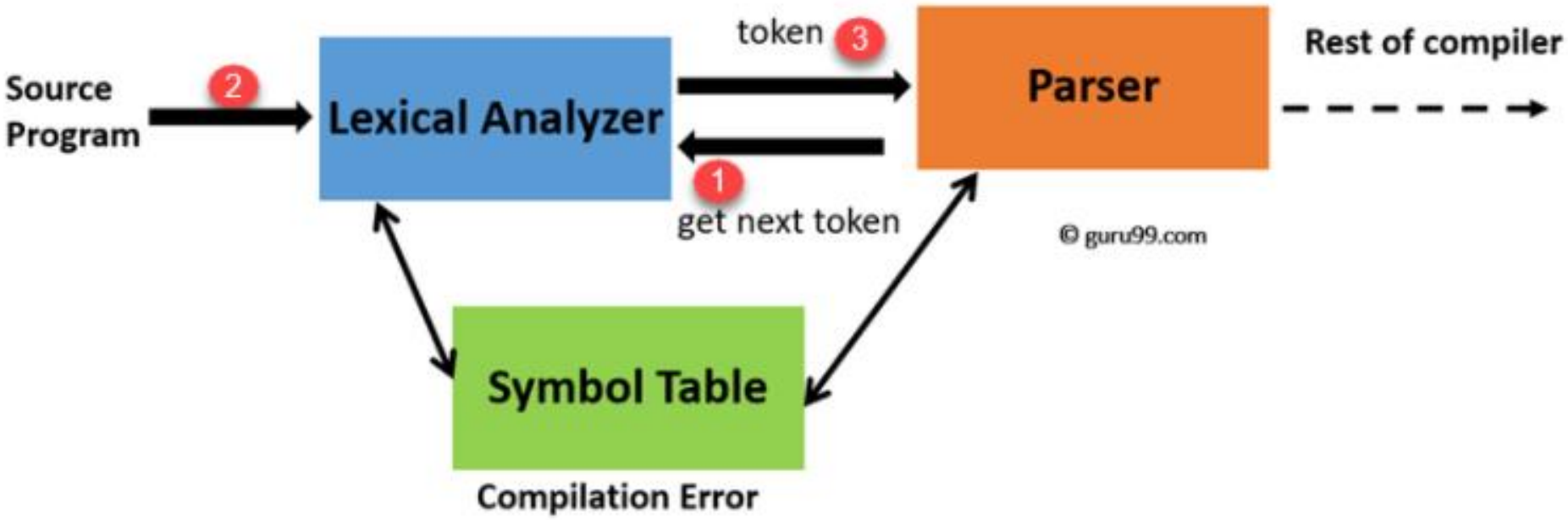
The Role of the Lexical Analyzer

- As the first phase of a compiler, the main task of the lexical analyzer is to,
  - read the input characters of the source program,
  - group them into lexemes,
  - and produce as output a sequence of tokens for each lexeme in the source program.

The Role of the Lexical Analyzer — *continued*

- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

The Role of the Lexical Analyzer — *continued*



- "Get next token" is a command which is sent from the parser to the lexical analyzer.
- On receiving this command, the lexical analyzer scans the input until it finds the next token.
- It returns the token to Parser.

The Role of the Lexical Analyzer — *continued*

- Converts source code to stream of tokens
- Removing comments
- Stripping out whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Recognizing keywords, identifiers and operators, constants, separator.
- Show errors when lexeme does not match any pattern.
- Correlating error messages generated by the compiler with the source program.

The Role of the Lexical Analyzer — *continued*

- Sometimes, lexical analyzers are divided into a cascade of two processes:
  a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
  b) *Lexical analysis* is the more complex portion, where the scanner produces the sequence of tokens as output.

The Role of the Lexical Analyzer — *continued*

- Sometimes, lexical analyzers are divided into a cascade of two processes:
  a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
  b) *Lexical analysis* is the more complex portion, where the scanner produces the sequence of tokens as output.

Why Lexical Analyzer is Separated from Syntax Analyzer?

- Because it makes both analyzers easier to debug.

Why Lexical Analyzer is Separated from Syntax Analyzer?

1. Simplicity –modular approach
   - By breaking down the task of language processing into smaller, more manageable parts, we can reduce the complexity of the overall system and make it easier to develop, test, and maintain.
2. Efficiency
3. Portability

Lexical Analysis Versus Parsing — *continued*

2. Compiler efficiency is improved.
   - Specific techniques to improve lexical analyzer or syntax analyzer alone can be implemented
   - In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
   - You can add new syntax without ever changing the lexical analyzer - for example add a new use for an existing keyword.
   - You can change the lexical definition of the language (for example change what you expect a valid identified is) without breaking the syntax.

Lexical Analysis Versus Parsing — *continued*

3. Compiler portability is enhanced.
   - Input-device-specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns, and Lexemes

- When discussing lexical analysis, we use three related but distinct terms:

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

- A pattern is a description of the form that the lexemes of a token may take.

- A lexeme is a sequence of characters in the source program that matches the pattern for a token

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Examples of tokens

- Figure gives some typical tokens, their informally described patterns, and some sample lexemes.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Examples of tokens

- To see how these concepts are used in practice, in the C statement `printf("Total = %d\n", score);` both `printf` and `score` are lexemes matching the pattern for token id, and `"Total = %d\n"` a lexeme matching literal.

Tokens, Patterns, and Lexemes — *continued*

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

1. One token for each keyword.
   - The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
- For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

Attributes for Tokens — *continued*

- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token.
- The token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

Attributes for Tokens — *continued*

- Normally, information about an identifier e.g.,
  - its lexeme,
  - its type,
  - and the location at which it is first found— is kept in the symbol table.

- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example

- The token names and associated attribute values for the Fortran statement

  `E = M * C ** 2`

  are written below as a sequence of pairs.

  <id, pointer to symbol-table entry for `E`**>**

  <assign-op>

  <id, pointer to symbol-table entry for `M`**>**

  <mult-op>

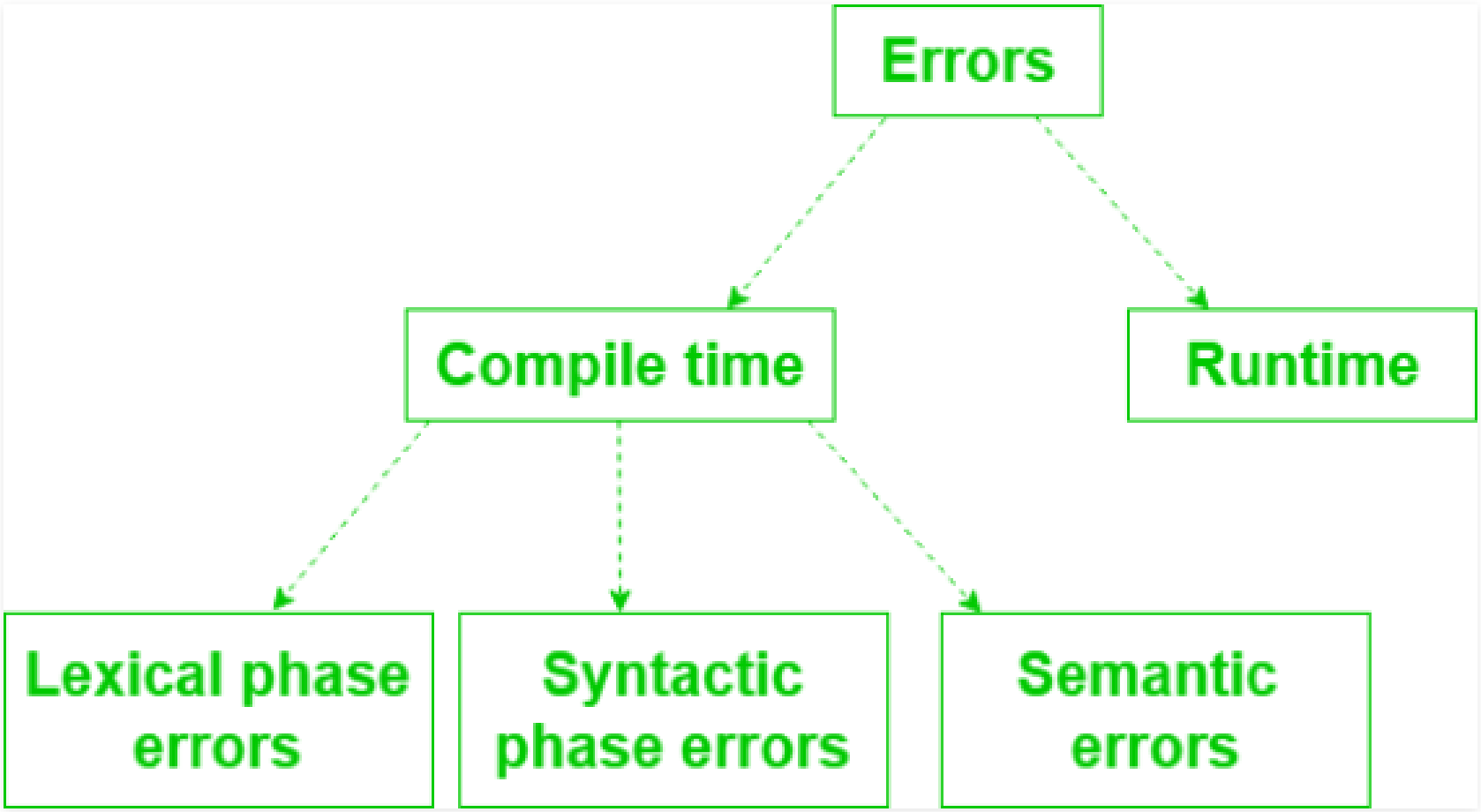
  <id, pointer to symbol-table entry for `C`**>**

  <exp-op>

  <number, integer value 2**>**

- In certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value.
- In this example, the token **number** has been given an integer-valued attribute.

Errors


Errors
Compile time
Runtime
Lexical phase errors
Syntactic phase errors
Semantic errors

Lexical Errors

- if the string `fi` is encountered for the first time in a C program in the context:
  `fi ( a == f(x)) ……`
- a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier.
- Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser.

Lexical Errors — *continued*

- Exceeding length of identifier.
- Appearance of illegal characters
- Unmatched string

Lexical Errors — *continued*

- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters
- Unmatched string/pattern.

Ex: printf(“Hello World”); $
This is a lexical error since an illegal character appears at the end of the statement.

Lexical Errors — *continued*

- Suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input.
- Perhaps the simplest recovery strategy is "panic mode" recovery.
- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as; or }
- The advantage is that it is easy to implement and guarantees not to go into an infinite loop
- The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

.

Lexical Errors — *continued*

Other possible error-recovery actions are:

1. deleting an extraneous character,
2. inserting a missing character,
3. replacing an incorrect character by a correct character,
4. transposing two adjacent characters.

Lexical Errors — *continued*

lexerror1.cpp

```
#include <iostream>
int main()
{
  `int i, j, k;
  return 0;
}
```

Response from gcc

```
lexerror1.cpp:4: error: stray ` in program
```

Lexical Errors — *continued*

lexerror2.cpp

```
int main()
{
  int 5test;

  return 0;
}
```

Response from gcc

```
lexerror2.cpp:3:7: error: invalid suffix
"test" on integer constant
```

Input Buffering

- Let us examine some ways that the simple but important task of reading the source program can be speeded.

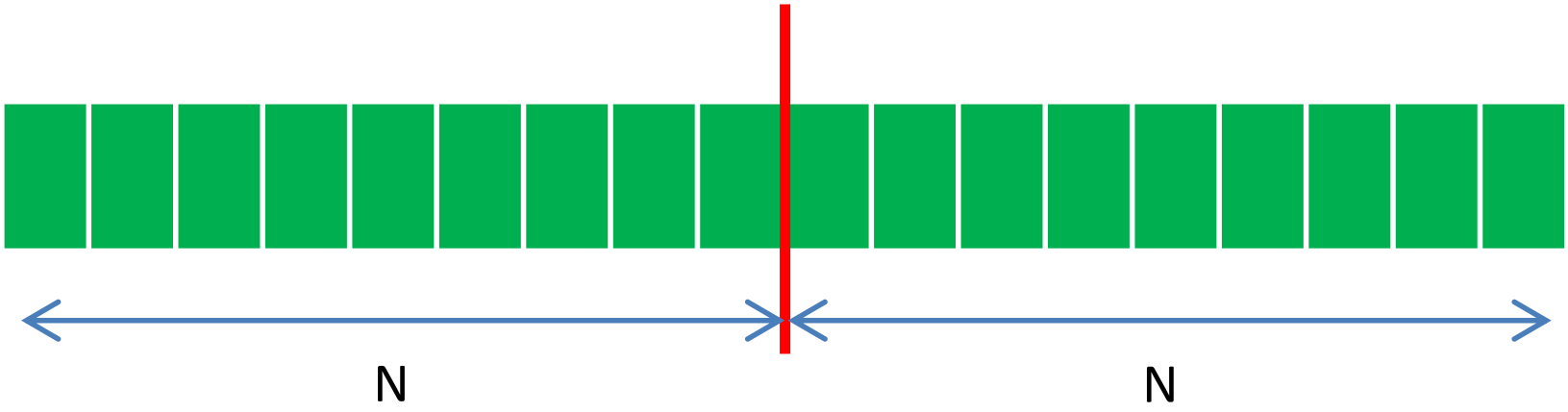
- There are many situations where we need to look at least one additional character ahead.

Input Buffering — *continued*

- For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.

- In C, single-character operators like −, =, or < could also be the beginning of a two-character operator like − >, ==, or <=.

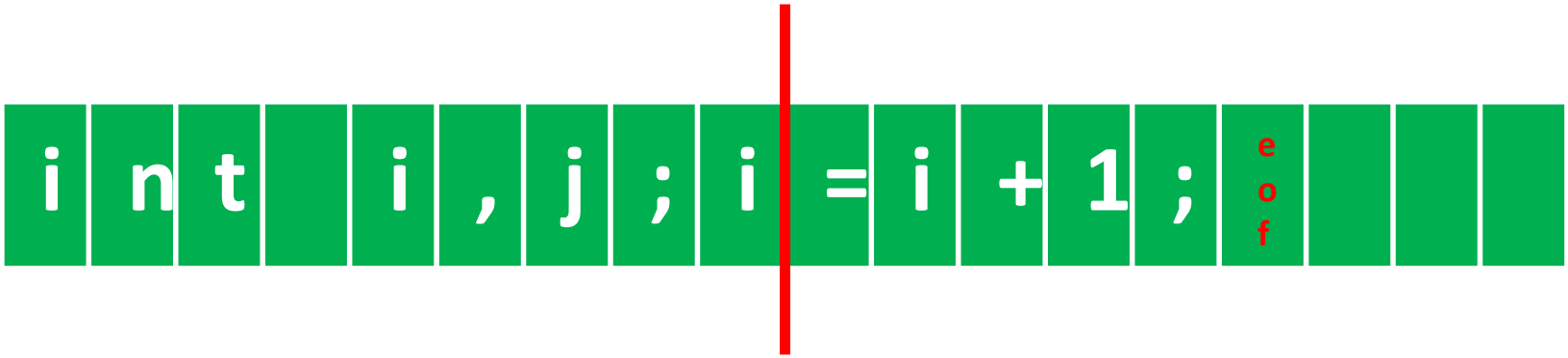
Input Buffering — *continued*



Buffer is divided into two N-characters half.

N is usually the size of disk blocks.

Buffer Pairs — *continued*



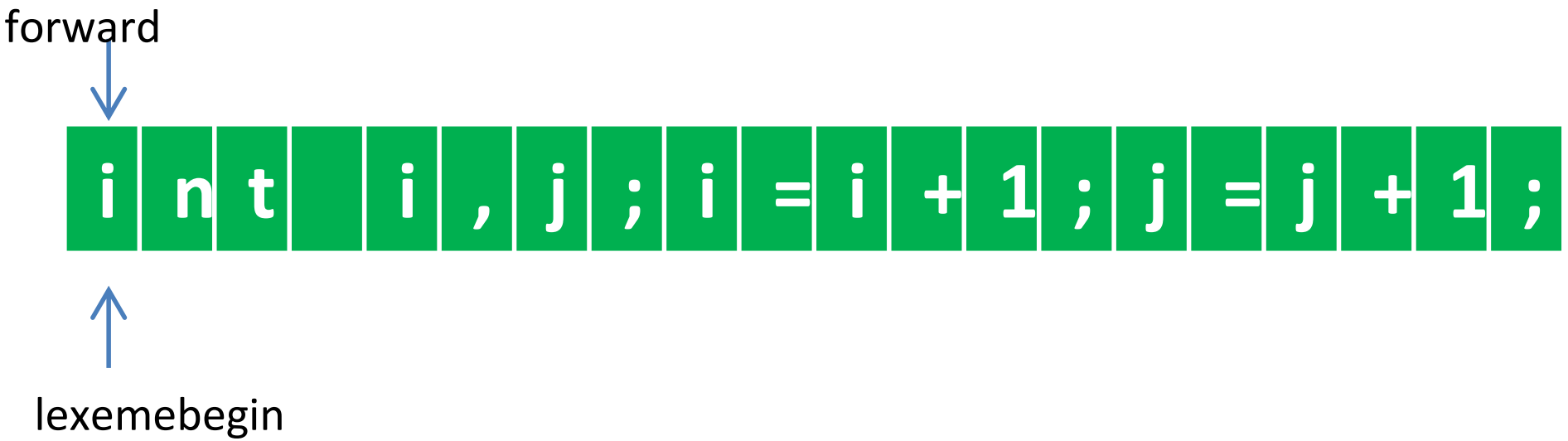
- If fewer than *N* characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- This **eof** is different from any possible character of the source program.

Buffer Pairs — *continued*

- Two pointers to the input are maintained:
  1. Pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are attempting to determine.

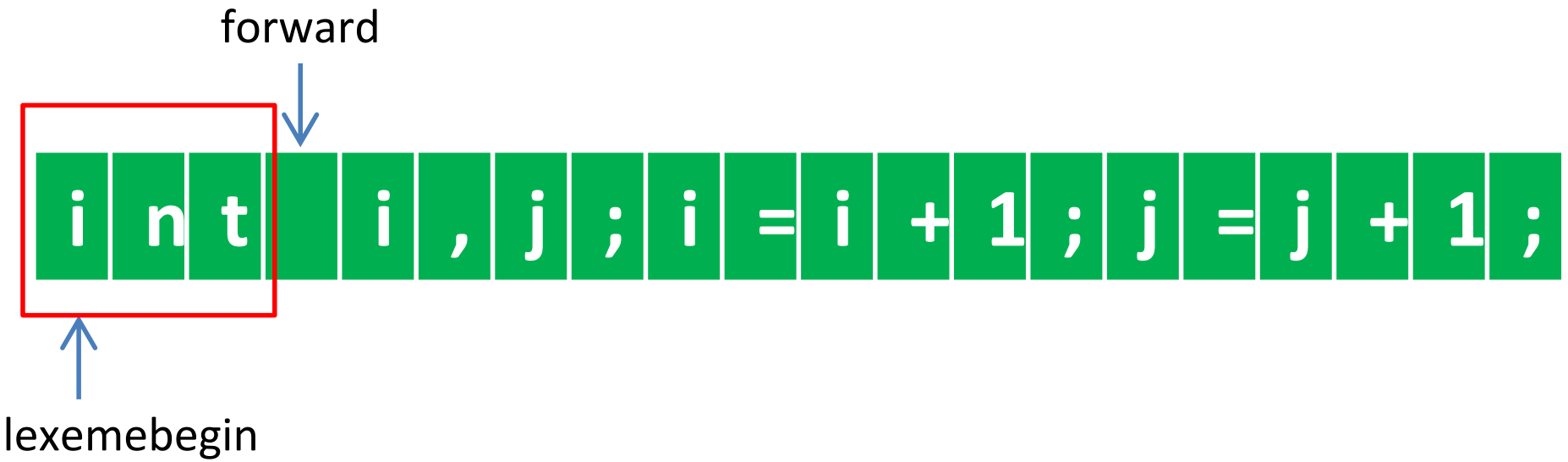
  2. Pointer `forward` scans ahead until a pattern match is found.

Input Buffering — *continued*



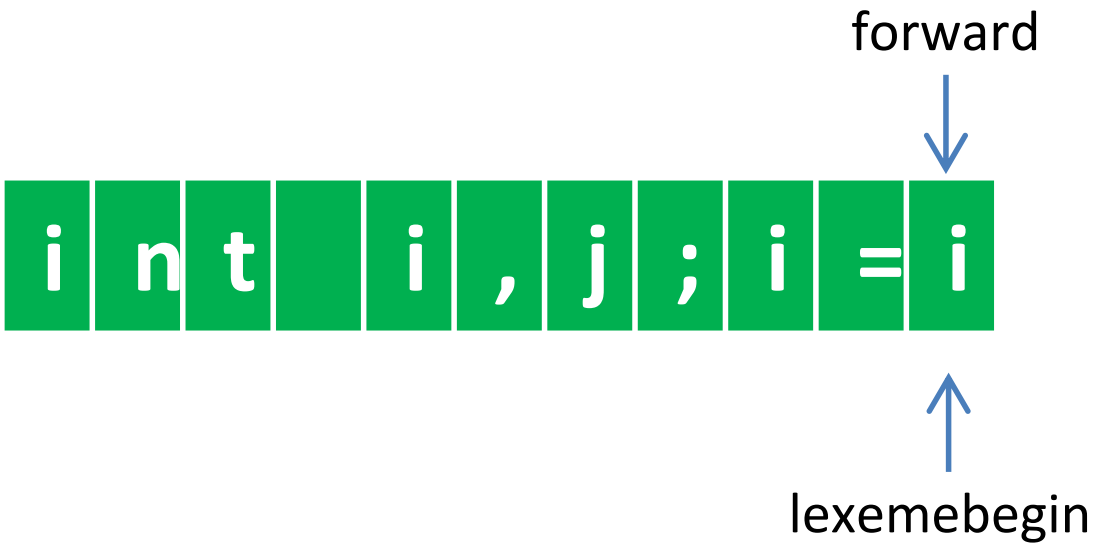
- Initially both the pointers point to the first character of the input string

Input Buffering — *continued*



- The forward ptr moves ahead to search for end of lexeme.
- As soon as the blank space is encountered, it indicates end of lexeme.
- In above example as soon as forward ptr encounters a blank space the lexeme “int” is identified.

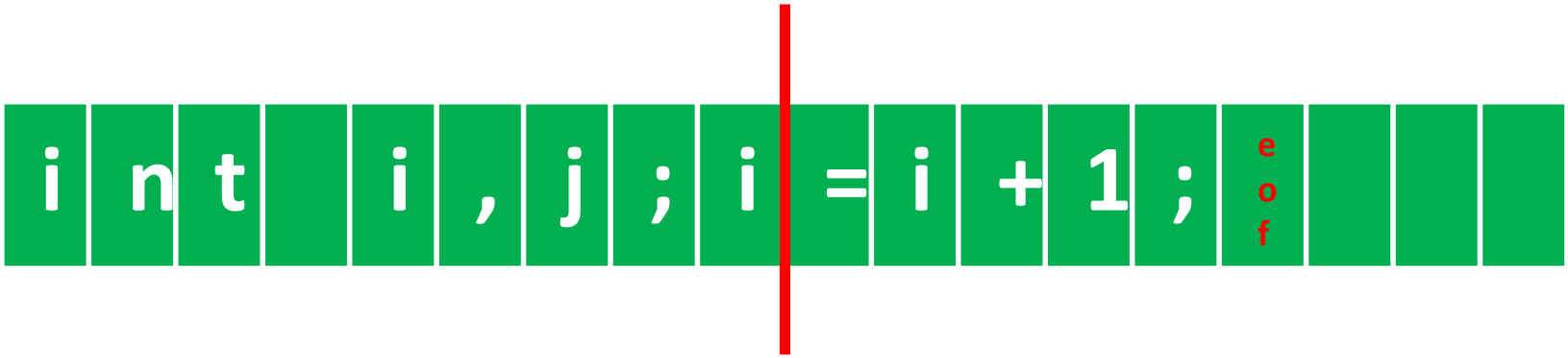
Input Buffering — *continued*



One Scheme Buffer

- if lexeme is very long then it crosses the buffer boundary
- to scan rest of the lexeme the buffer has to be refilled
- which makes overwriting the first of lexeme

Input Buffering — *continued*



Two Scheme Buffer

- The first buffer and second buffer are scanned alternately
- When end of current buffer is reached the other buffer is filled.

Sentinels

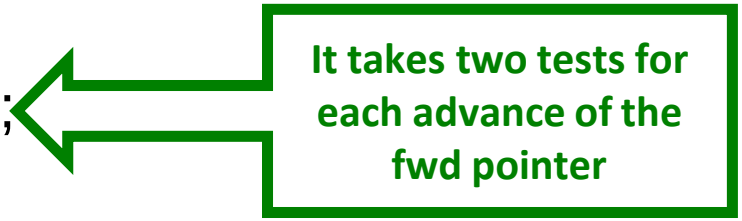
- If we use the previous scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers.
- If we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests:
  - one for the end of the buffer.
  - And one to determine what character is read (the latter may be a multiway branch).

Sentinels

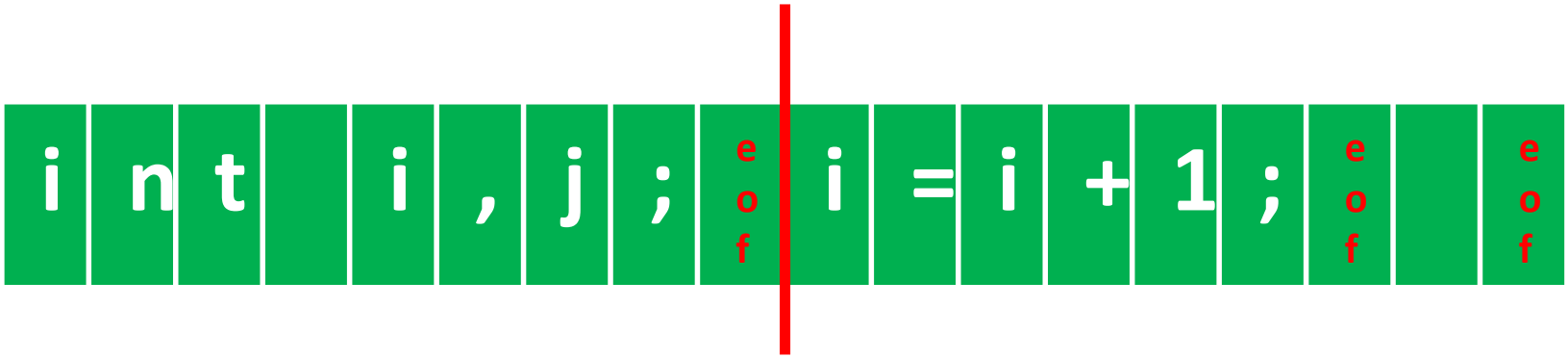
```
if (fwd at the end of the first half)
    reload the second half
    set fwd to point to the beginning of second half

else if (fwd at the end of the second half)
    reload the first half
    set fwd to point to the beginning of first half

else
    *fwd++;
```

Sentinels

```
if (fwd at the end of the first half)
    reload the second half
    set fwd to point to the beginning of second half

else if (fwd at the end of the second half)
    reload the first half
    set fwd to point to the beginning of first half

else
    *fwd++;
```



Sentinels— *continued*



- Adds extra eof at the end of each buffer as sentinels
- It optimizes the code by reducing the number of tests

Sentinels

```
*fwd++;
If(*fwd == eof)
        if (fwd at the end of the first half)
            reload the second half
            set fwd to point to the beginning of second
        half

        else if (fwd at the end of the second half)
            reload the first half
            set fwd to point to the beginning of first half

        else  /* end of input*/
            terminate lexical analysis;
```

Recognition of Tokens

- Our discussion will make use of the following running example.

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

A grammar for branching statements

Example

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a** or **b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal "**a+b**" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Example — *continued*

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

- To simplify matters, we make the common assumption that keywords are also ***reserved*** *words*.
- They are not identifiers, even though their lexemes match the pattern for identifiers.

Example — *continued*

- In addition, we assign the lexical analyzer the job of stripping out white- space, by recognizing the “token” *ws* defined by:

$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

- Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names.
- Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser.
- We rather restart the lexical analysis from the character that follows the whitespace.
- It is the following token that gets returned to the parser.

Recognition of Tokens-continued

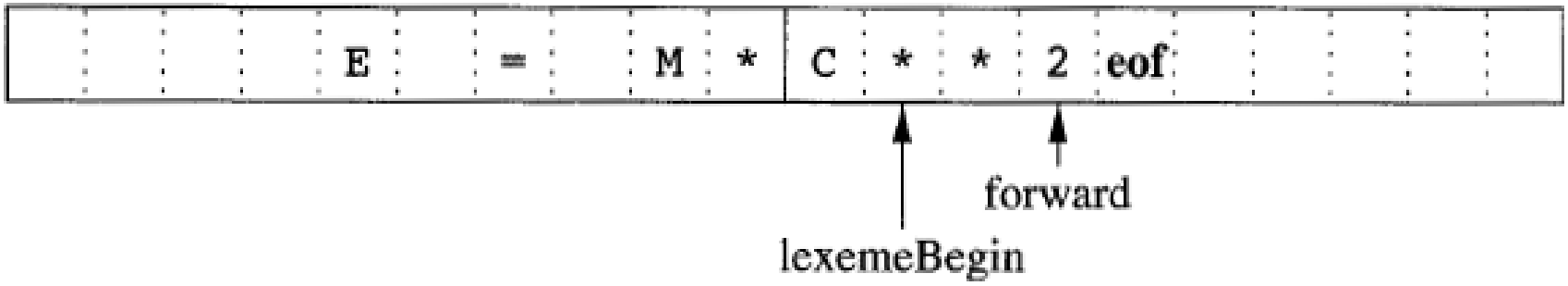
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

- Our goal for the lexical analyzer is summarized in figure.

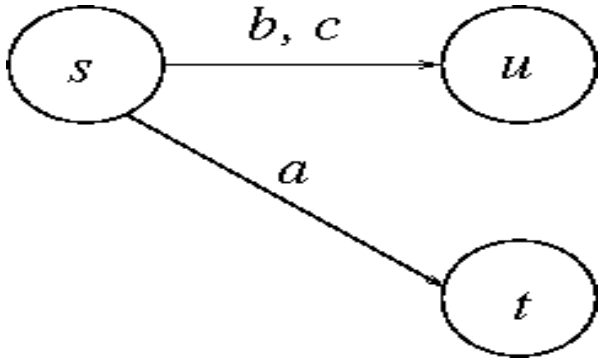
Transition Diagrams — *continued*

- Transition diagrams have a collection of nodes or circles, called states.
- We may think of a state as summarizing all we need to know about what characters we have seen between the `lexemeBegin` pointer and the `forward` pointer.



Transition Diagrams — *continued*

- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a *symbol* or *set of symbols*.
- If we are in some state *s*, and the next input symbol is *a*, we look for an edge out of state *s* labeled by *a* .
- If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads.



Transition Diagrams — *continued*

- We shall assume that all our transition diagrams are *deterministic*, meaning that *there is never more than one edge out of a given state with a given symbol among its labels*.

Transition Diagrams — *continued*

Some important conventions about transition diagrams are:

1. Certain states are said to be **accepting, or final**.

- These states indicate that a lexeme has been found.
- We always indicate an accepting state by a **double circle**.
- If there is an action to be taken — typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.

Transition Diagrams — *continued*

Some important conventions about transition diagrams are:

2. In addition, if it is necessary to **retract** the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state.

- In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.

Transition Diagrams — *continued*

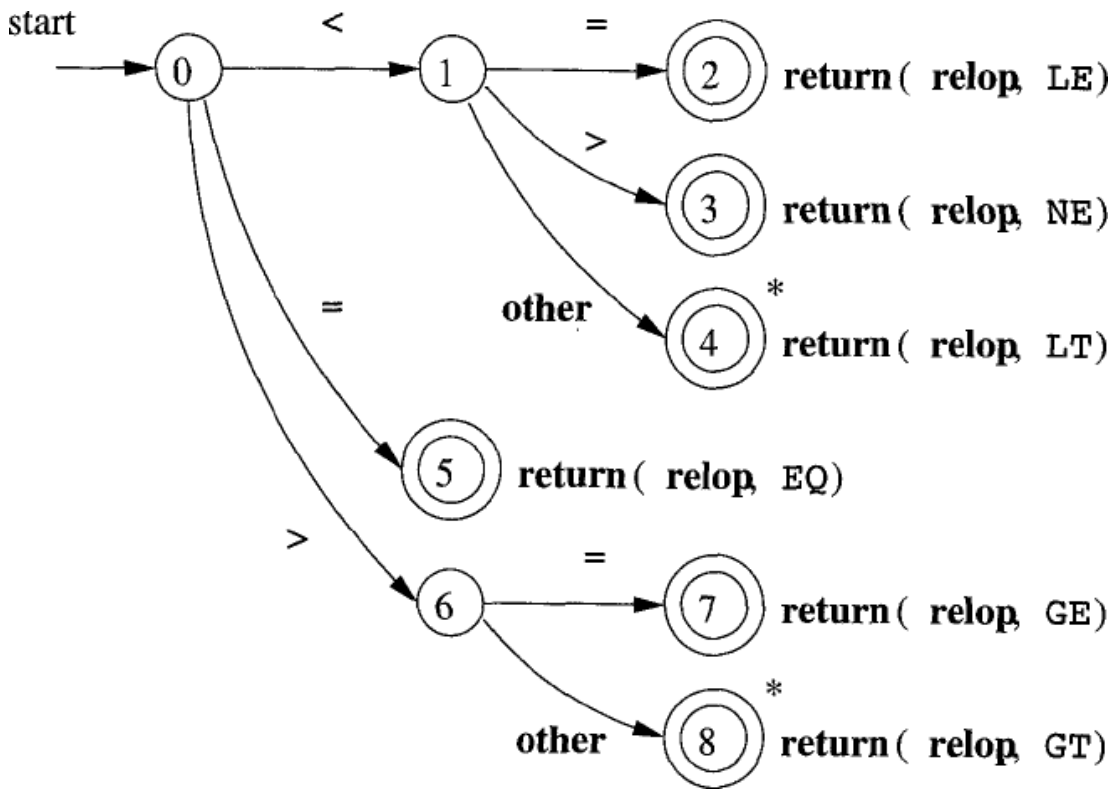
Some important conventions about transition diagrams are:

3. One state is designated the **start state**, or **initial state** it is indicated by an edge, labeled "**start** ," entering from nowhere.

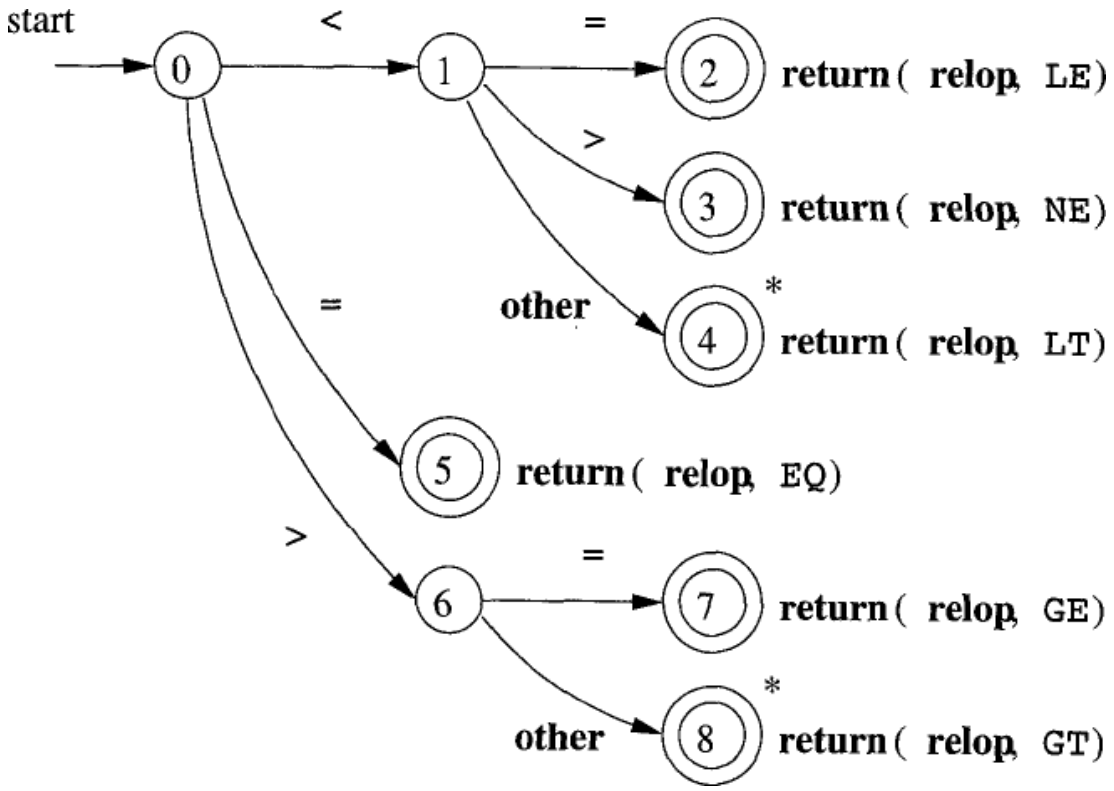
- The transition diagram always begins in the start state before any input symbols have been read.

Example — *continued*

- We begin in state 0, the start state.
- If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=.
- We therefore go to state 1, and look at the next character.

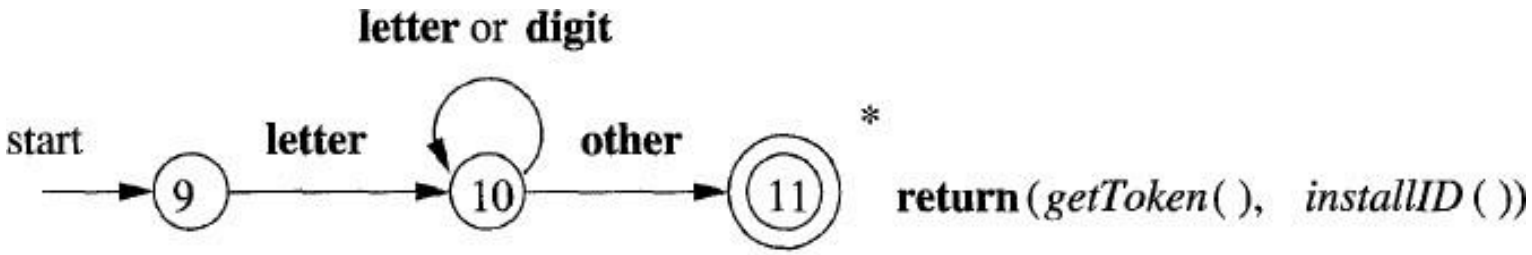


Example — *continued*



Note that if, in state 0, we see any character besides <, =, or >, we can not possibly be seeing a **relop** lexeme, so this transition diagram will not be used.

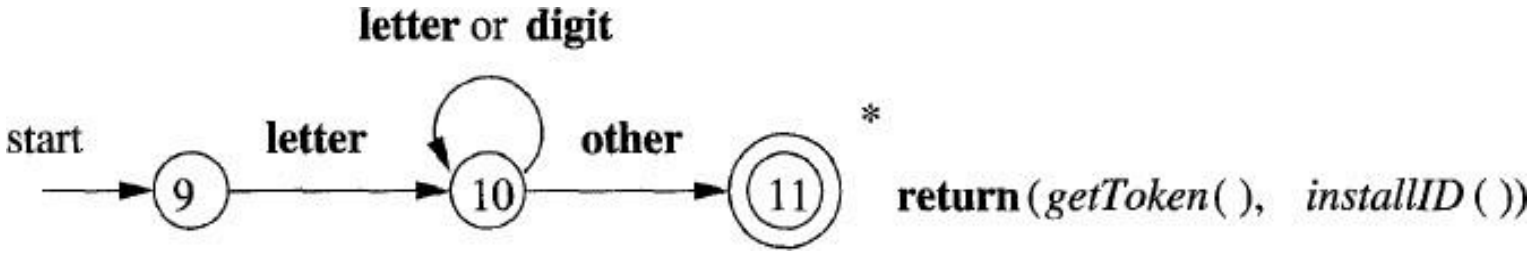
Recognition of Reserved Words and Identifiers



A transition diagram for **id**'s and keywords

- But, this diagram will also recognize the keywords `if`, `then`, and `else`.

Recognition of Reserved Words and Identifiers — *continued*



1. Install the reserved words in the symbol table initially.

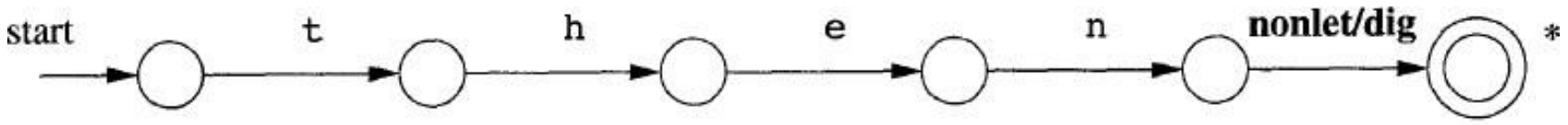
- **installID()** checks if the lexeme is already in the table. If it is not present, the lexeme is installed as an id token.

- Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id**.

- **gettoken()** examines the lexeme and returns the token name, either id or a name corresponding to a reserved keyword.

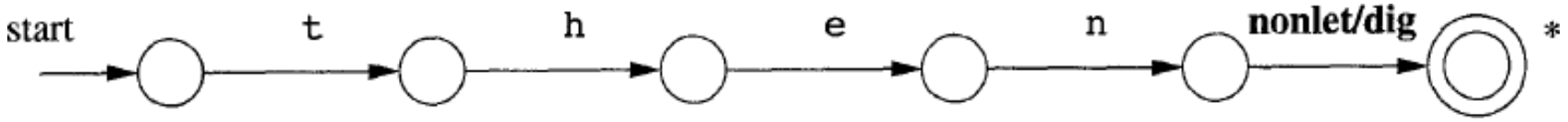
Recognition of Reserved Words and Identifiers — *continued*

2. Create separate transition diagrams for each keyword.
   - An example for the keyword `then` is shown in figure.



Hypothetical transition diagram for the keyword `then`

Recognition of Reserved Words and Identifiers — *continued*



- Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., **any character that cannot be the continuation of an identifier**.

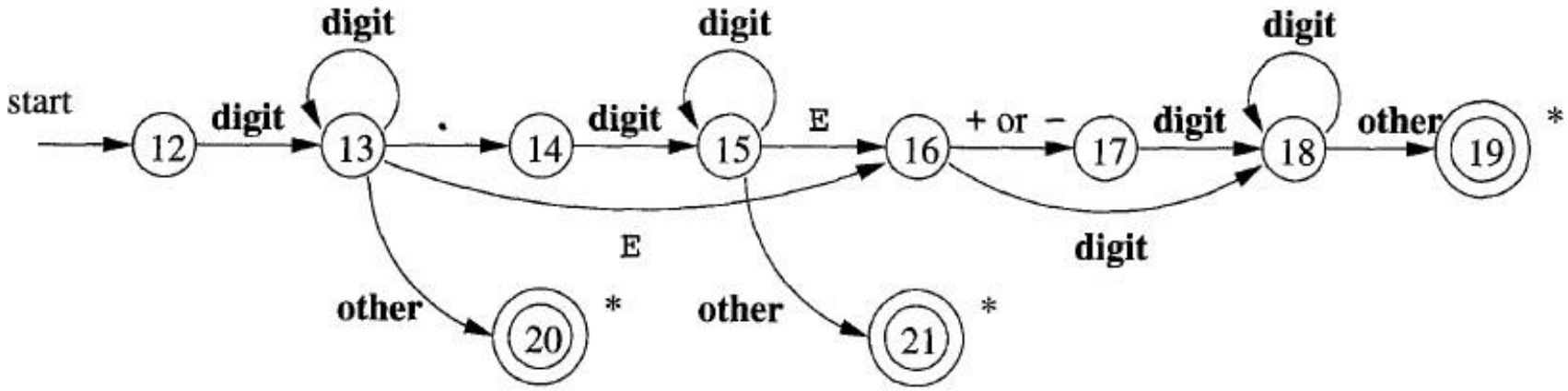
- It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was **id**, with a lexeme like `thenextvalue` that has `then` as a proper prefix.

Recognition of Reserved Words and Identifiers — *continued*

- If we adopt this approach, then we must prioritize the tokens so that the **reserved-word tokens are recognized in preference to id**, when the lexeme matches both patterns.

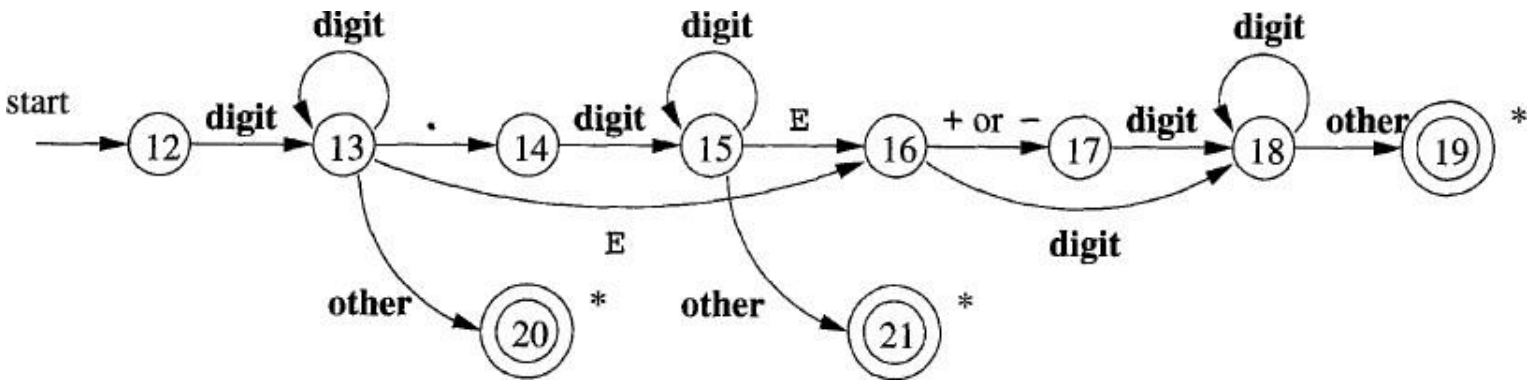
Completion of the Running Example — *continued*

- The transition diagram for token **number** is shown in figure.



A transition diagram for unsigned numbers

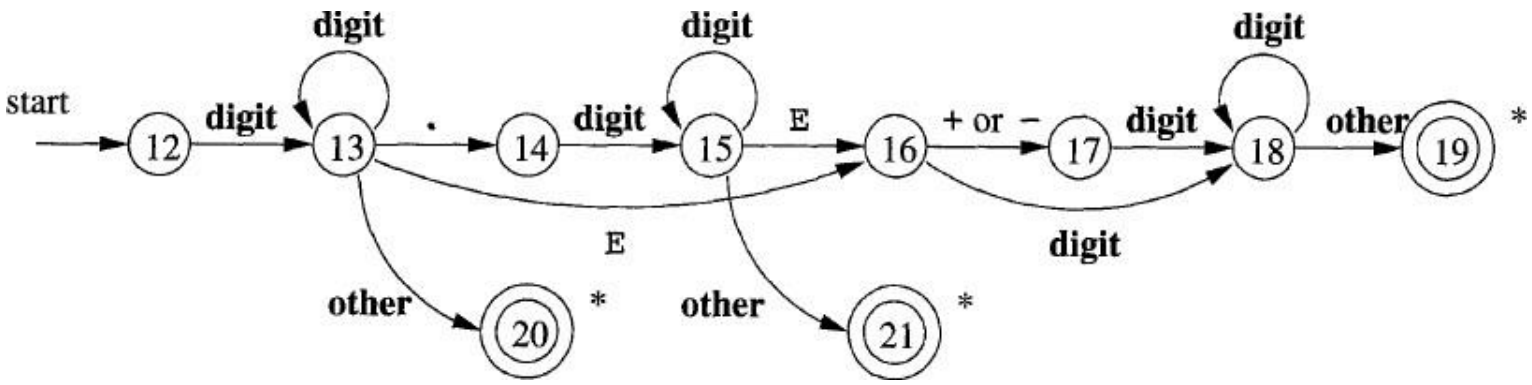
Completion of the Running Example — *continued*



123

- Beginning in state 12, if we see a digit, we go to state 13.
- In that state, we can read any number of additional digits.
- However, if we see anything but a digit or a dot, we have seen a number in the form of an integer.
- That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered.

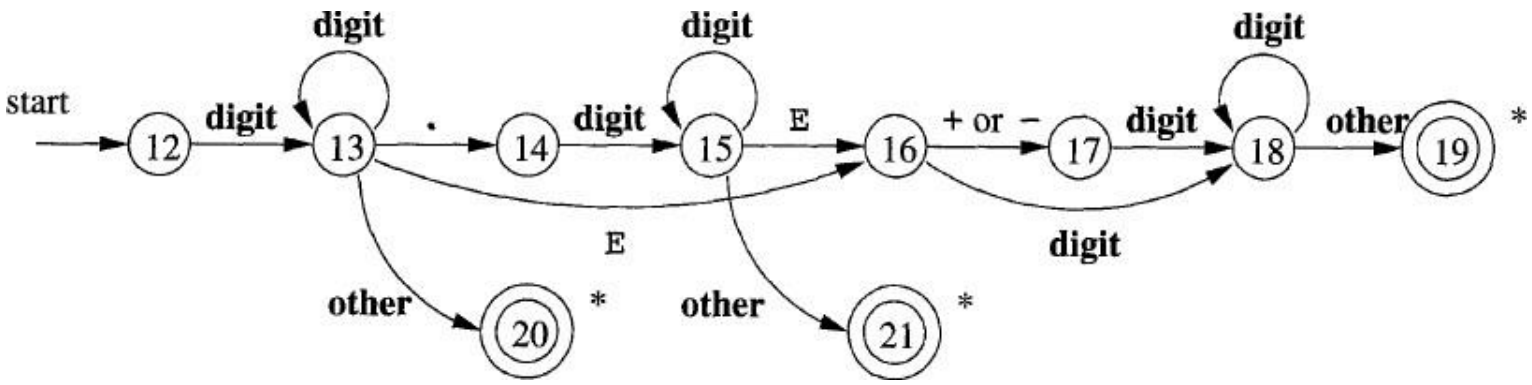
Completion of the Running Example — *continued*



123.456

- If we instead see a dot in state 13, then we have an "optional fraction."
- State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose.

Completion of the Running Example — *continued*

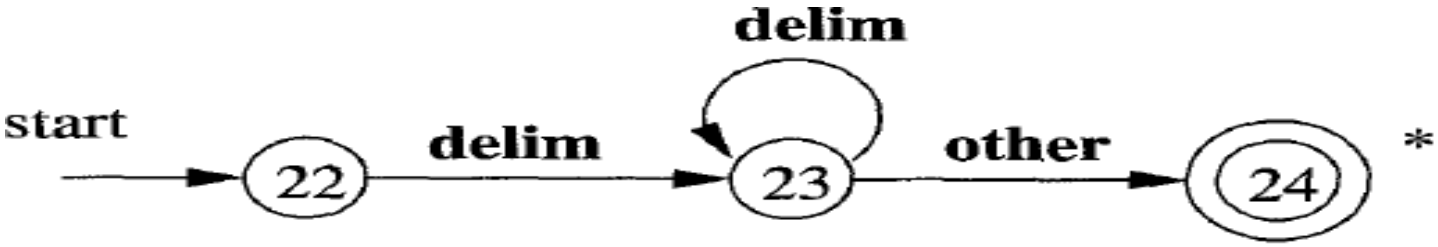


123.456E789 123.456E+789 123.456E-789

- If we see an E, then we have an "optional exponent," whose recognition is the job of states 16 through 19.
- Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

Completion of the Running Example — *continued*

- Transition diagram for whitespace is shown.



A transition diagram for whitespace

- The delim in the diagram represents any of the whitespace characters, say space, tab, and newline.
- The final star is there because we needed to find a non-whitespace character in order to know when the whitespace ends and this character begins the next token.
- There is no action performed at the accepting state.

Architecture of a Transition-Diagram-Based Lexical Analyzer

- The idea is that we write a piece of code for each decision diagram.
- This piece of code contains a case for each state, which typically reads a character and then goes to the next case depending on the character read.
- The numbers in the circles are the names of the cases.
- Accepting states often need to take some action and return to the parser. Many of these accepting states (the ones with stars) need to restore one character of input. This is called retract() in the code.

Architecture of a Transition-Diagram-Based Lexical Analyzer

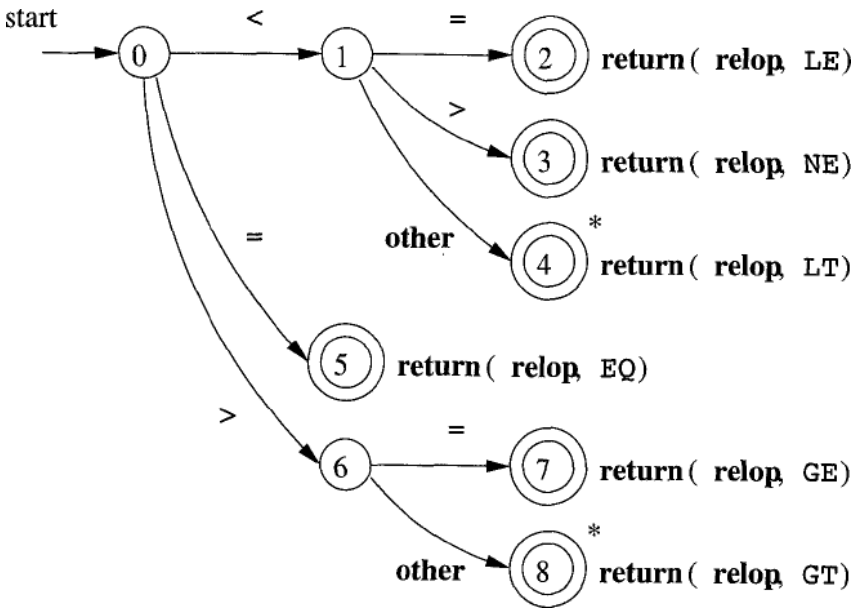
- What should the code for a particular diagram do if at one state the character read is not one of those for which a next state has been defined? That is, what if the character read is not the label of any of the outgoing arcs?

- This means that we have failed to find the token corresponding to this diagram.
- The code calls **fail().** This is not an **error case**.
- It simply means that the current input does not match this particular token.
- So, we start the next diagram at the point where this failing diagram started.
- If we have tried all the diagram, then we have a **real failure** and need to print an error message and perhaps try to repair the input.

Example

- In figure we see a sketch of `getRelop()`, a C++ function whose job is to simulate the transition diagram for **relop**.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```




```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

- Function `getRelop()` returns an object of type `TOKEN`, that is, a pair consisting of the token name **relop** and an attribute value (the code for one of the six comparison operators in this case).
- `getRelop()` first creates a new object `retToken` and initializes its first component to RELOP, the symbolic code for token **relop**.

Architecture of a . . . Lexical Analyzer — *continued*

- To place the simulation of one transition diagram in perspective, let us consider the ways code could fit into the entire lexical analyzer.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

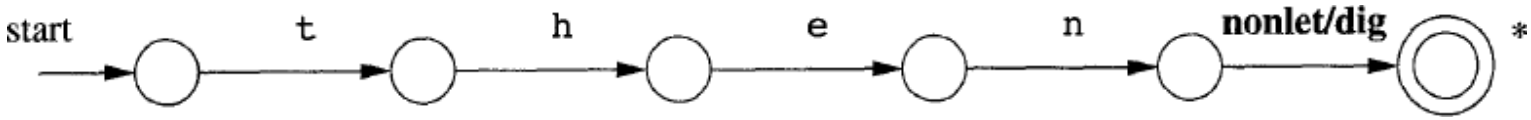
Architecture of a . . . Lexical Analyzer — *continued*

1. We could arrange for the transition diagrams for each token to be tried sequentially.
   - Then, the function `fail()` resets the pointer forward and starts the next transition diagram, each time it is called.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Architecture of a . . . Lexical Analyzer — *continued*

- This method allows us to use transition diagrams for the individual keywords.



- We have only to use these before we use the diagram for **id**, in order for the keywords to be reserved words.

Architecture of a . . . Lexical Analyzer — *continued*

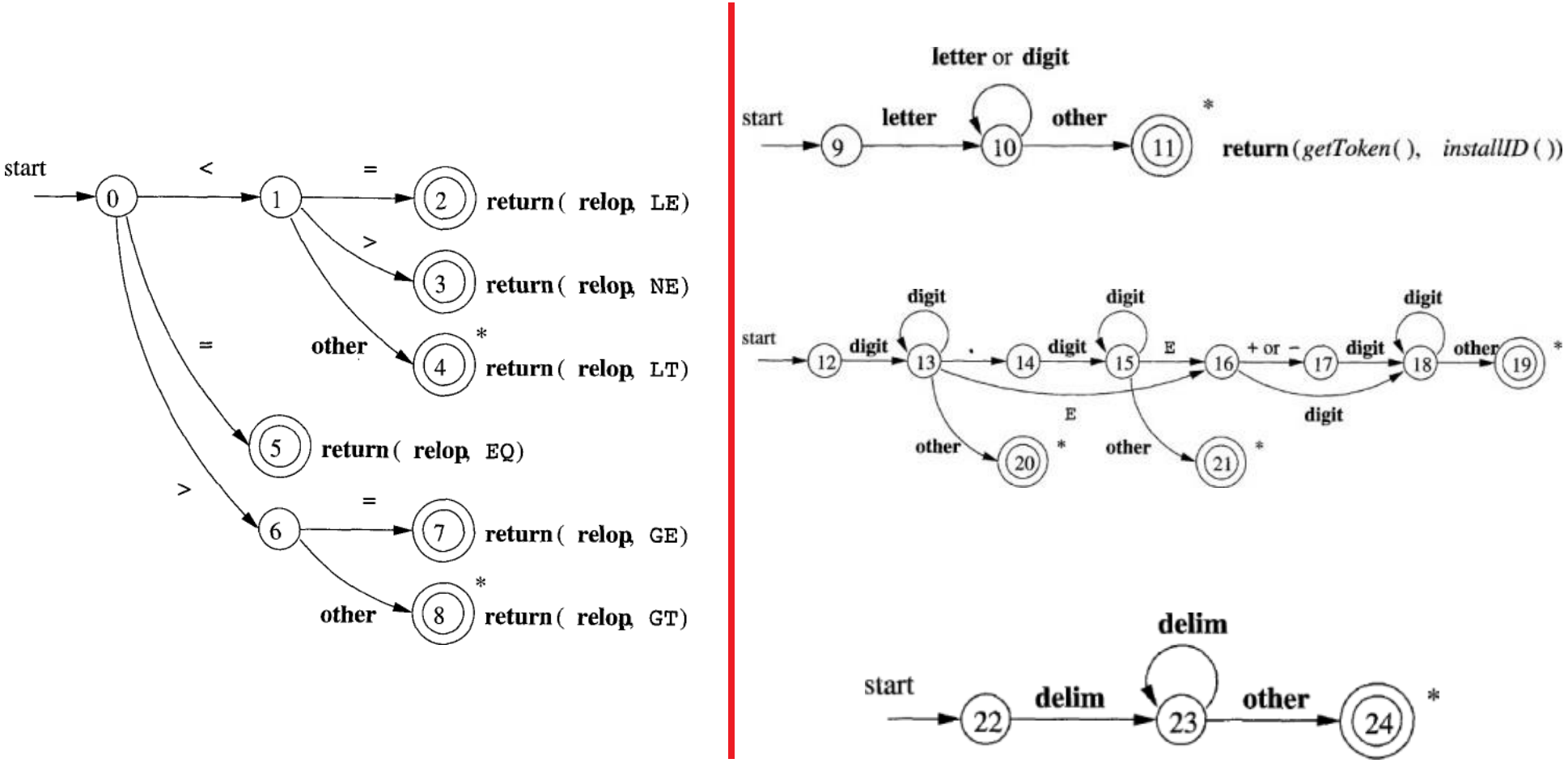
2. Unlike the method above, which tries the diagrams one at a time, the first new method tries them in parallel. That is, each character read is passed to each diagram (that hasn't already failed).

- Care is needed when one diagram has accepted the input, but others still haven't failed and may accept a longer prefix of the input..
- The normal strategy is to take the longest prefix of the input that matches any pattern.
- That rule allows us to prefer identifier `thenext` to keyword `then`, or the operator `->` to `-`, for example.

Architecture of a . . . Lexical Analyzer — *continued*

3. The preferred approach, is to combine all the transition diagrams into one.
   - We allow the transition diagram to read input until there is no possible next state.
   - And then take the longest lexeme that matched any pattern, as we discussed in item (2) above.

Architecture of a . . . Lexical Analyzer — *continued*



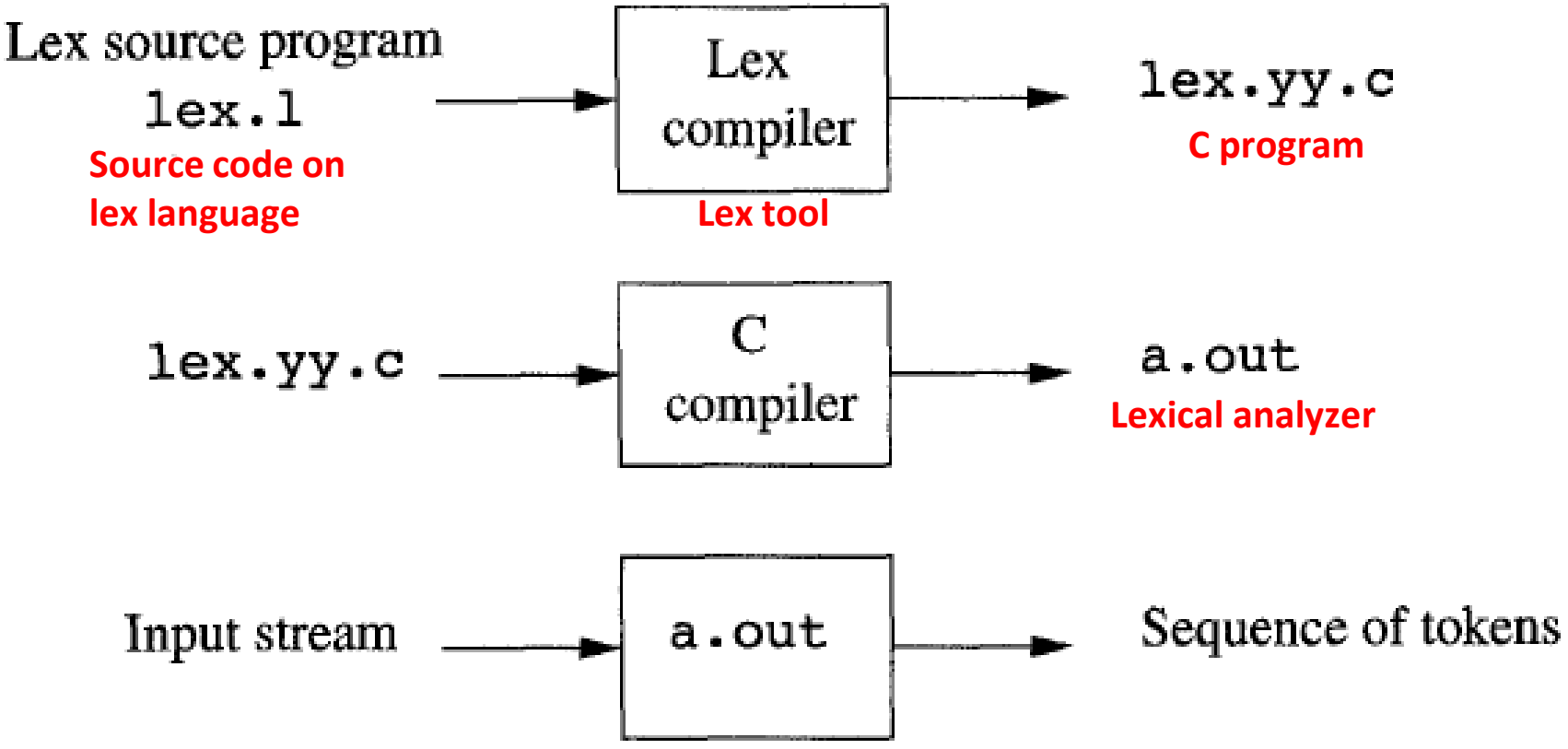
- Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact.

The Lexical- Analyzer Generator `Lex`

- We introduce a tool called `Lex`, or in a more recent implementation `Flex`.
- This allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- The input notation for the `Lex` tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*.
- Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram.

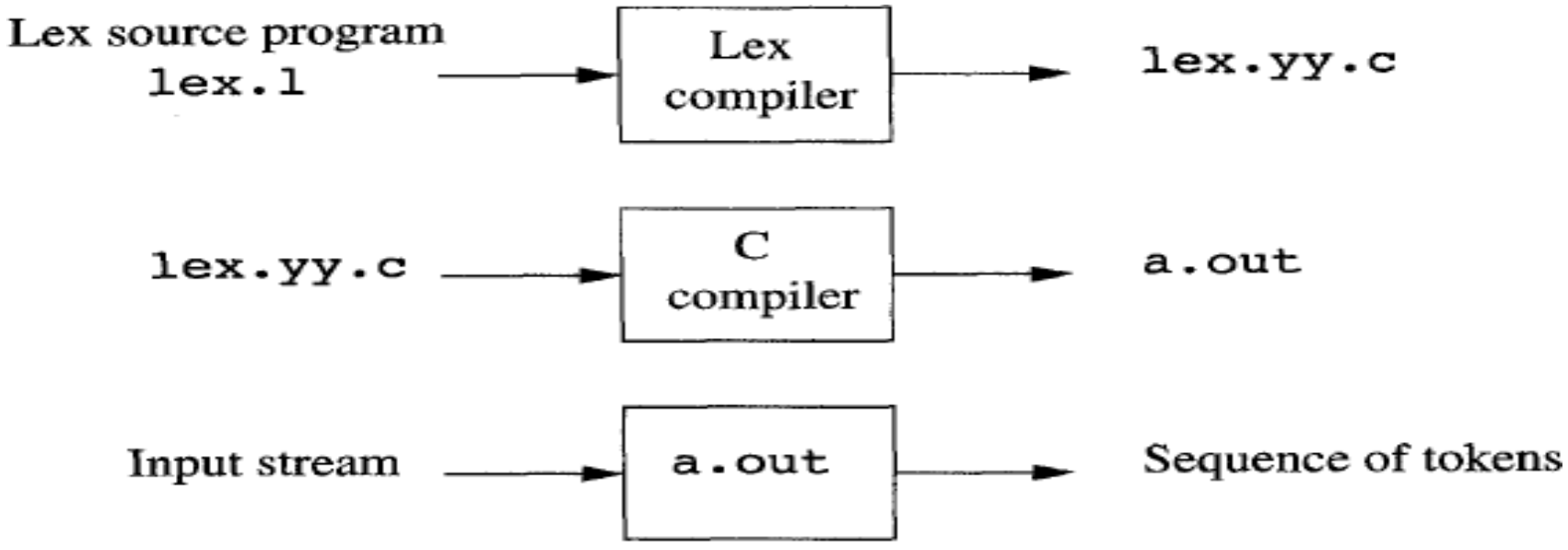
Use of Lex

- Figure suggests how Lex is used.



Creating a lexical analyzer with Lex

Use of Lex — *continued*



Creating a lexical analyzer with Lex

- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable **`yylval`** which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

Structure of Lex Programs

- A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

Structure of Lex Programs — *continued*

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The declarations section includes
  - declarations of variables,
  - manifest constants (identifiers declared to stand for a constant, e.g., the name of a token),
  - and regular definitions.

Structure of Lex Programs — *continued*

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The translation rules each have the form

  `Pattern {Action }`

- Each pattern is a regular expression, which may use the regular definitions of the declaration section.
- The actions are fragments of code, typically written in C.

Structure of Lex Programs — *continued*

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The third section holds whatever additional functions are used in the actions.
- Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

Example

- Figure is a Lex program that recognizes the tokens of the definitions and returns the token found.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{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) installNum(); 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 installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Figure 3.23: Lex program for the tokens of Fig. 3.12

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
```

- In the declarations section we see a pair of special brackets, `%{` and `%}`.
- Anything within these brackets is copied directly to the file `lex.yy.c`, and is not treated as a regular definition.
- In our example, we have listed in a comment the names of the manifest constants, `LT`, `IF`, and so on, but have not shown them defined to be particular integer.

Example — *continued*

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

%%
```

- Also in the declarations section is a sequence of regular definitions.
- These use the extended notation for regular expressions.

```
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 installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

- In the auxiliary-function section, we see two such functions, `installID()` and `installNum()`.
- Like the portion of the declaration section that appears between `%{...%}` everything in the auxiliary section is copied directly to file `lex.yy.c`, but may be used in the actions.

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); 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);}

%%
```

- First, `ws`, an identifier declared in the first section, has an associated empty action.
- If we find whitespace, we do not return to the parser, but look for another lexeme.

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); 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);}

%%
```

- Should we see the two letters `if` on the input, and they are not followed by another letter or digit, then the lexical analyzer consumes these two letters from the input and returns the token name `IF`, that is, the integer for which the manifest constant `IF` stands.

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); 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);}

%%
```

- The action taken when id is matched is threefold.

1. Function `installID()` is called to place the lexeme found in the symbol table.

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); 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);}

%%
```

2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler.

```
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); 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);}

%%
```

- Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that Lex generates:
  (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin`.
  (b) `yyleng` is the length of the lexeme found.

```
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); 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);}

%%
```

3. The token name `ID` is returned to the parser.

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); 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);}

%%
```

- The action taken when a lexeme matching the pattern number is similar, using the auxiliary function `installNum()`.

Thank You

CSE 303 (Compilers)

Syntax Analysis

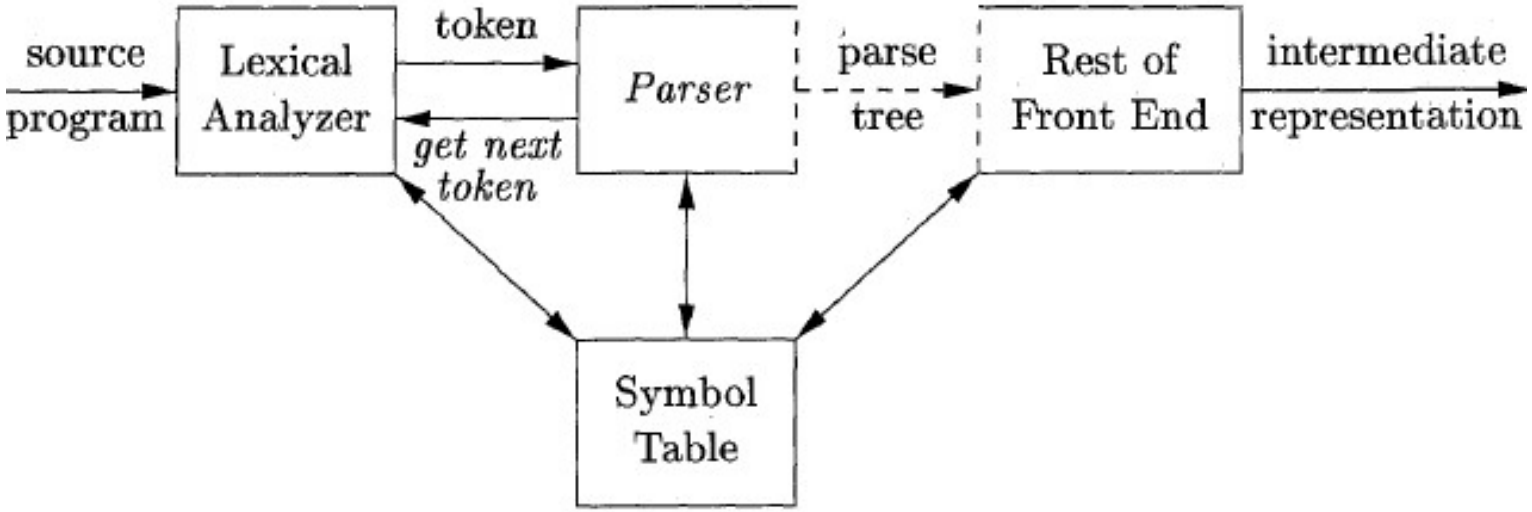
Rubyeat Islam

Assistant Professor

Department of Computer Science and Engineering
Military Institute of Science and Technology
(MIST) Mirpur Cantonment, Dhaka-1216,
Bangladesh

Courtesy: Dr Mohammad Masroor Ali, Professor, BUET
and Lec Tasmiah Tamzid Anannya

The Role of the Parser — *continued*



Position of parser in compiler model

- We expect the parser
  - to report any syntax errors in an intelligible fashion
  - and to recover from commonly occurring errors to
  - continue processing the remainder of the program.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Types of Parser

- What is parsing?
  - It is a process of deriving string from a given grammar that means whether the string belongs to a proper grammar or not.

Types of Parser— *continued*

- There are three general types of parsers for
  - grammars: universal,
  - top-down,
  - and
  - bottom-up.
- Universal parsing methods can parse any grammar.

- These general methods are, however, too inefficient to use in production compilers.

Types of Parser— *continued*

- The methods commonly used in compilers can be classified as being either **top-down** or **bottom-up**.
- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves).
- Bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

Representative Grammars — *continued*

- The following grammar treats + and $*$ alike.

$$E \rightarrow \quad E + E \;\mid\; E * E \;\mid\; (\, E \,) \mid \textbf{id}$$

- So it is useful for illustrating techniques for handling ambiguities during parsing.
- Here, E represents expressions of all types.
- This grammar permits more than one parse tree for expressions like $a + b * c$.

Syntax Error Handling — *continued*

- Most programming language specifications do not describe how a compiler should respond to errors.
- Error handling is left to the compiler designer.
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

Syntax Error Handling — *continued*

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

Elimination of Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} Aa$ for some string a.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- In simple left recursion there was one production of the form $A \rightarrow A\,\alpha$.
- Here we study the general case.

Elimination of Left Recursion — *continued*

- Left-recursive pair of productions $A \rightarrow A\,\alpha \mid \beta$ can be replaced by the non-left-recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example

$$A \rightarrow A\alpha \mid \beta$$

to be replaced by

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \in$$

- Grammar for arithmetic expressions,

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

Eliminating the immediate left recursions we obtain,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \in$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \in$$
$$F \rightarrow (E) \mid \textbf{id}$$

Elimination of Left Recursion — *continued*

- No matter how many A-productions there are, we can eliminate immediate left recursion from them.
- First, we group the A-productions as,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

  where no β_i, begins with an A.
- Then, we replace the A-productions by,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \in$$

- It does not eliminate left recursion involving derivations of two or more steps.

Elimination of Left Recursion — *continued*

- It does not eliminate left recursion involving derivations of two or more steps.
- Consider the grammar,

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

Algorithm

Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm to G. Note that the resulting non-left-recursive grammar may have ϵ-productions.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

Algorithm

Grammar with cycles: Grammar where derivations of the form

$$A \overset{+}{=} \quad \Rightarrow \quad A \text{ occurs.}$$

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

■

In the first iteration for $i = 1$, the outer `for`-loop of lines (2) through (7) eliminates any immediate left recursion among A_1-productions.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- Any remaining A_1 productions of the form $A_1 \rightarrow A_l\alpha$ must therefore have $l > 1$.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- After the $i - 1$st iteration of the outer `for`- loop, all nonterminals A_k, where $k < i$, are "cleaned".
- That is, any production $A_k \rightarrow A_l \alpha$, must have $l > k$.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- As a result, on the ith iteration, the inner loop of lines (3) through (5) progressively raises the lower limit in any production $A_i \rightarrow A_m \alpha$, until we have $m \geq i$.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

Then, eliminating immediate left recursion for the A_i productions at line (6) forces m to be greater than i.

Example

Input Grammar G with no cycles or ϵ-productions.

- We apply the procedure to grammar,

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- Technically, the algorithm is not guaranteed to work, because of the ϵ -production.
- But in this case the production $A \rightarrow \epsilon$ turns out to be harmless.

Example — *continued*

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- We order the nonterminals S, A.
- $A_1 = S$, $A_2 = A$

Example — *continued*

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- We order the nonterminals S, A.
- $A_1 = S$, $A_2 = A$

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- $i = 1$, $A_1 = S$
- $j = 1$ to $j = 1 - 1 = 0$, the loop is *not* entered

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- $i = 1$, $A_i = A_1 = S$

Example — *continued*

6) eliminate the immediate left recursion among the A_i-productions;

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- There is no immediate left recursion among the *S*-productions, so nothing happens for the case *i* = 1. (A_i=A_1 = *S*)

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- $j = 1$ to $j = 2 - 1 = 1$, the loop is entered

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

- $i = 2$, $A_i = A_2 = A$
- $j = 1$ to $j = 2 - 1 = 1$, the loop is entered

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow Sy$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid$
- b Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid$
- b Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid$
- b Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid$
- b Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $S \rightarrow Aa \mid b$ to be put in A, $A \rightarrow Sd$
- We substitute $S \rightarrow Aa \mid b$ in $A \rightarrow Sd$ to get the following A-productions,

$$A \rightarrow Aad \mid bd$$

Example — *continued*

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-RecursiveGrammar

$S \rightarrow$

$A \rightarrow$

- $S \rightarrow Aa \mid b$ to be put in A, $A \rightarrow Sd$
- We substitute $S \rightarrow Aa \mid b$ in $A \rightarrow Sd$ to get the following A-productions,

$$A \rightarrow Aad \mid bd$$

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

Example — *continued*

- All $A_i = A_2 = A$-productions together,

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the A-productions yields the following,

$$A \rightarrow bdA^j \mid A^j$$

$$A^j \rightarrow cA^j \mid adA^j \mid s$$

Example — *continued*

6) eliminate the immediate left recursion among the A_i-productions;

- All $A_i = A_2 = A$-productions together,

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the A-productions yields the following,

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

Example — *continued*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

i has attained the value of $n = 2$ and the loops are no more

Example — *continued*

Left-RecursiveGrammar

$$S \rightarrow$$
$$A \rightarrow$$

- Put together we get the following non-left-recursive grammar,

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i - 1$) {
4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

Conceptual Technique Summary (AGAIN)

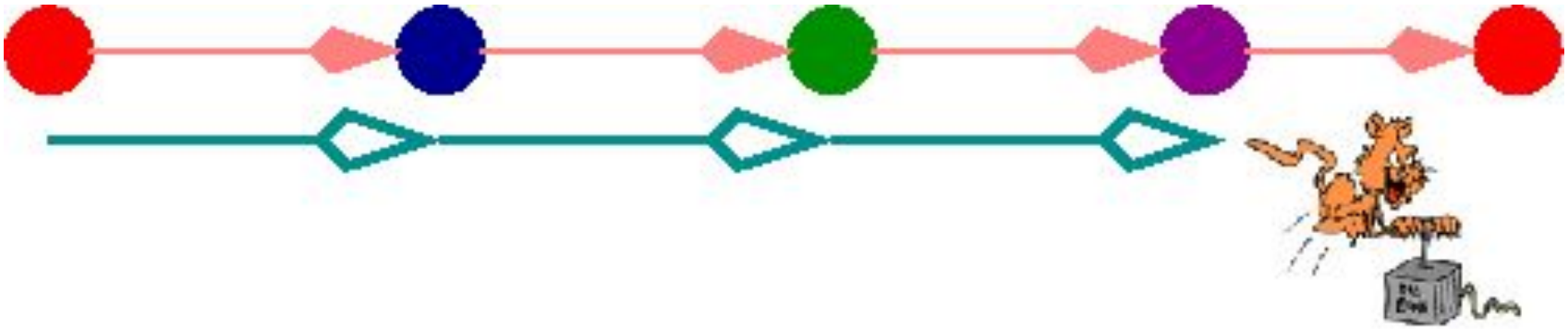
- Put some order in the nonterminals.
- Start by making first nonterminal productions left-recursion-free.
- Put the first nonterminal left-recursion-free productions into those of the second one.
- Now make the productions of second nonterminal left-recursion-free.
- Thus keep on growing the set of left-recursion-free productions.

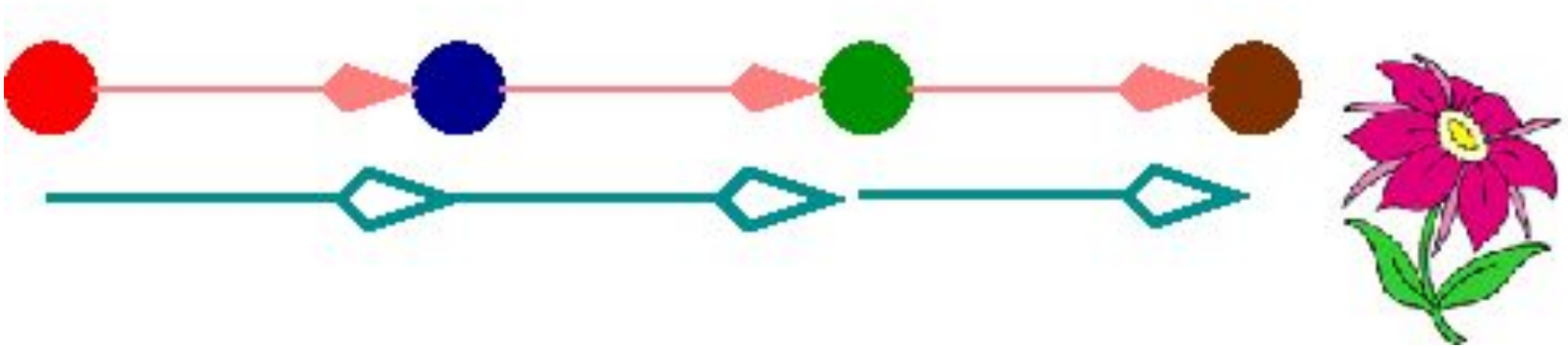
Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal *A*.
- We may be able to rewrite the *A*-productions to defer the decision until we have seen enough of the input to make the right choice.

Left Factoring — *continued*

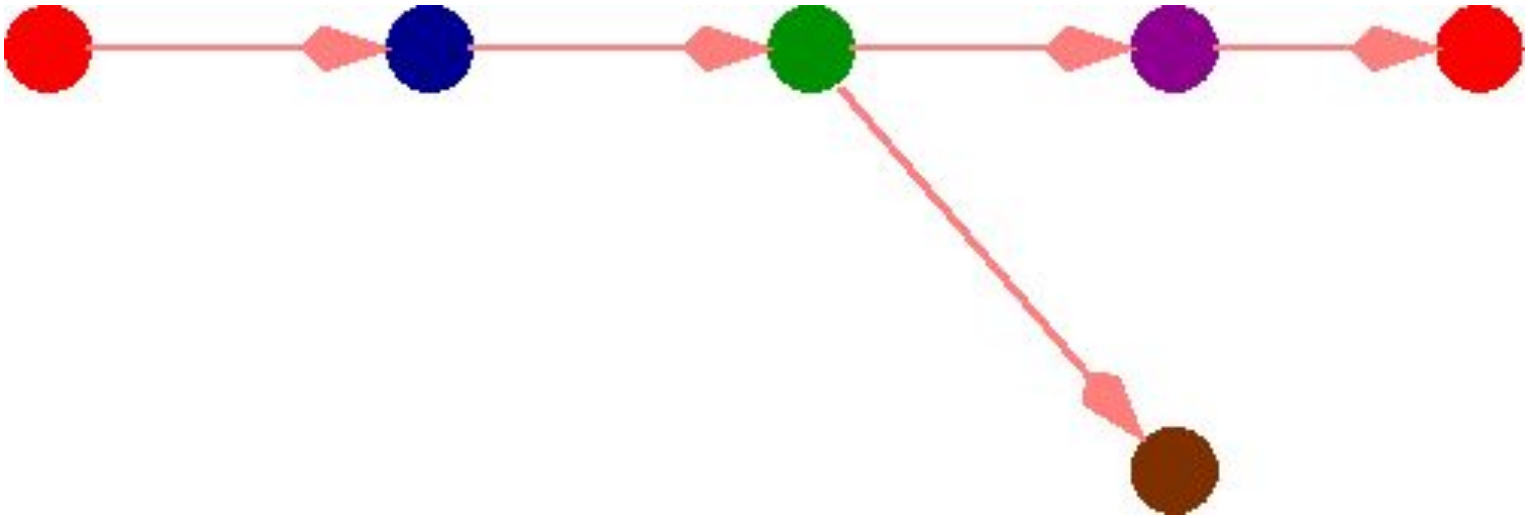
Road Direction: *Red* → *Blue* → *Green* → *Brown*





Left Factoring — *continued*

Defer the decision until we have seen enough of the input to make the right choice.



Left Factoring — *continued*

- We have the two productions,

$$\textit{stmt} \rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt}$$
$$\mid\ \ \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}$$

- On seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*.

Left Factoring — *continued*

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions.
- The input begins with a nonempty string derived from α .
- We do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding A to $\alpha A'$.
- Then, after seeing the input derived from α we expand A' to β_1 or β_2.
- Left-factored, the original productions become,

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Left Factoring Algorithm

INPUT. Grammar G.

OUTPUT An equivalent left-factored grammar.

Left Factoring Algorithm — *continued*

Method.

- For each nonterminal A find the longest prefix α common to two or more of itsalternatives.
- If $\alpha \neq \epsilon$ (there is a nontrivial common prefix), replace all the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' = \beta_1 \mid \beta_2 \mid \beta_3 \ldots\ldots\ldots\ldots \mid \beta_n$$

  where A' is a new nonterminal.
- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example

- The following grammar abstracts the dangling-else problem:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

- Here *i*, *t* , and *e* stand for **if**, **then** and **else**, *E* and *S* for "expression" and "statement."
- Left-factored, this grammar becomes:

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

Example — *continued*

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

- Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ.

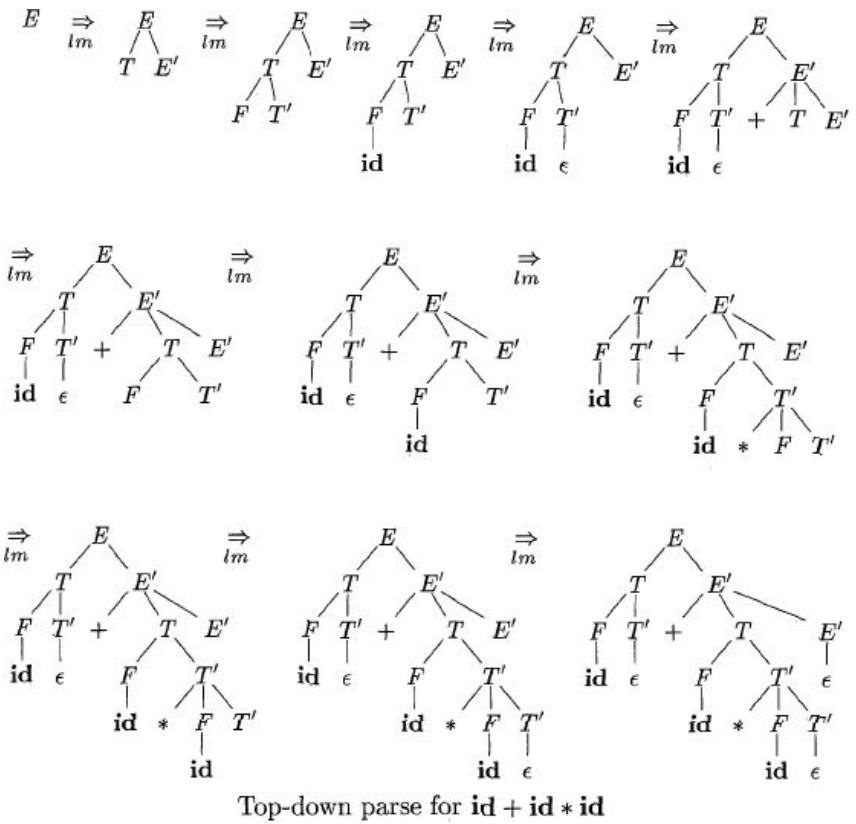
Top-Down Parsing

- Top-down parsing can be viewed as the problem
  - of constructing a parse tree for the input
  - string, starting from the root and
  - creating the nodes of the parse tree in preorder (depth-first).
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example

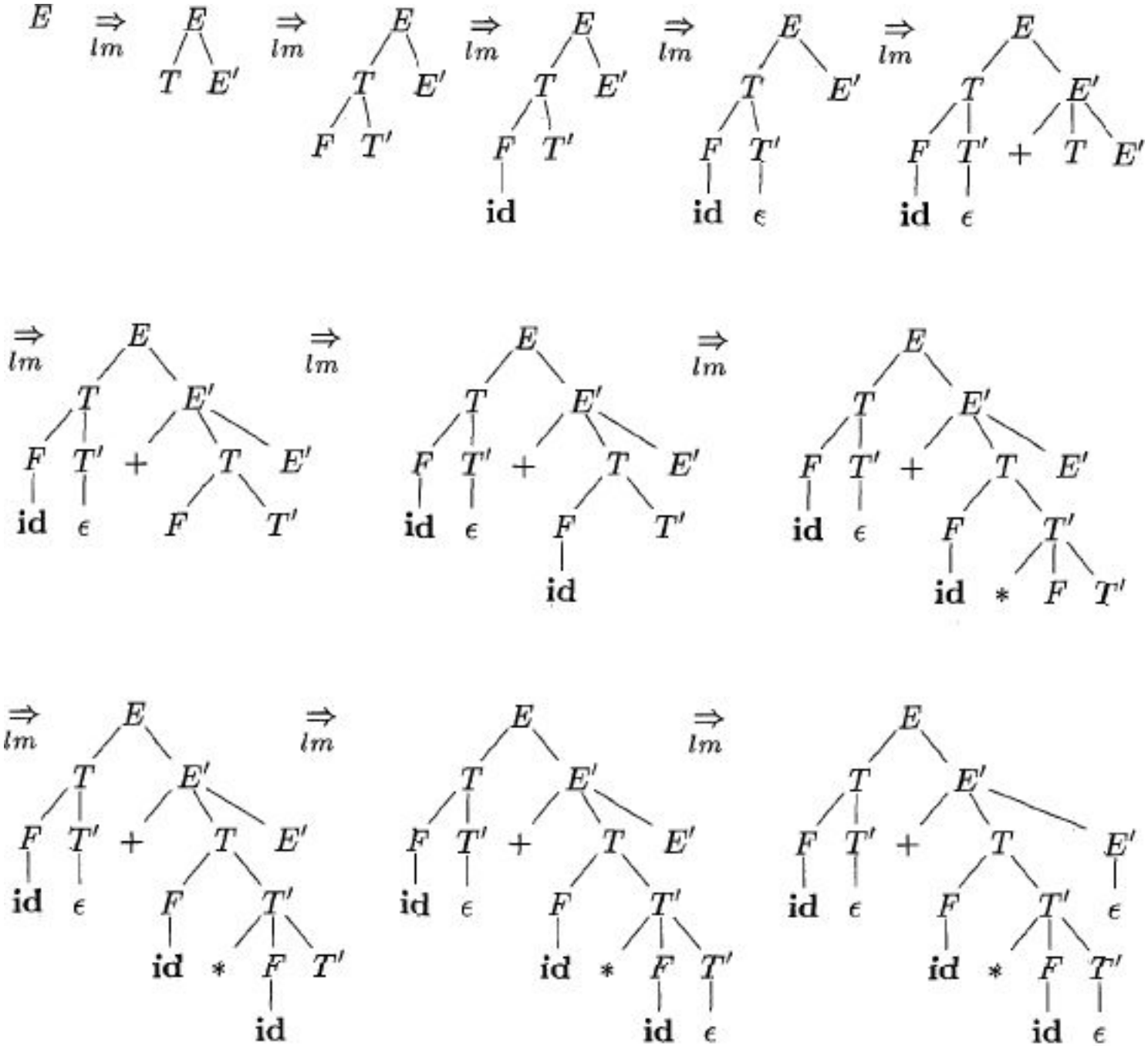
- The sequence of parse trees for the input **id** + **id** $*$ **id** is a top-down parse according to grammar.

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$



Top-down parse for **id** + **id** $*$ **id**

- This sequence of trees corresponds to a leftmost derivation of the input.



Top-down parse for **id + id * id**

Top-Down Parsing — *continued*

- The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the LL(k) class.
- We will discuss LL(1) parser.

Types of Top-Down Parsing

- Recursive- Descent Parsing- requires back tracking
- Predictive Parsing – makes explicit choice of parsing

Recursive- Descent Parsing

```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

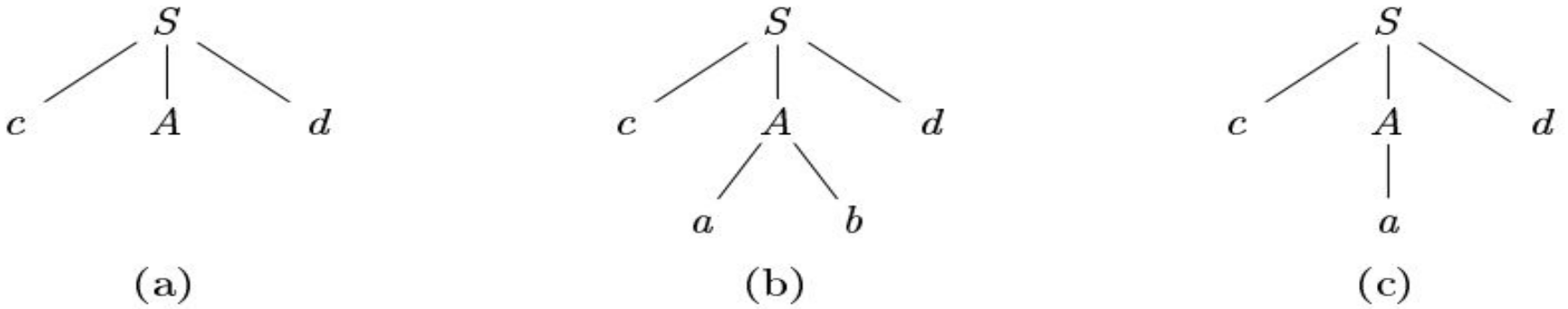
Figure 4.13: A typical procedure for a nonterminal in a top-down parser

- Recursive- Descent Parsing consists of a set of procedures, one for each non terminal.
- It may require back-tracking which is not very efficient.

Recursive- Descent Parsing

S -> cAd

A-> ab | a



- Left recursive grammar can cause a recursive descent parser, even one with backtracking, to go into an infinite loop.

FIRST and FOLLOW

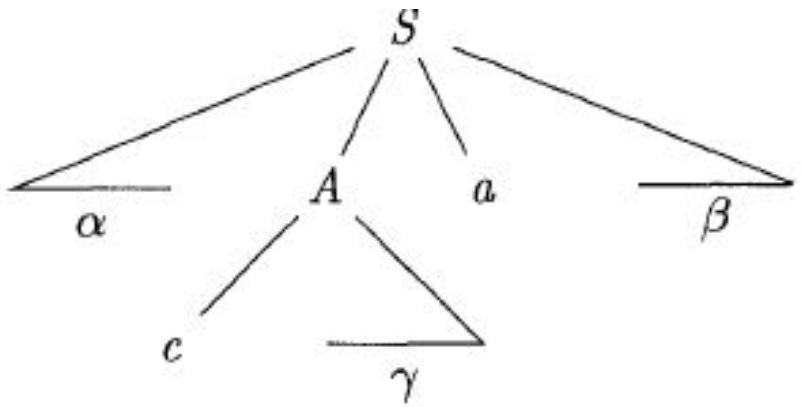
- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar *G*.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens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

FIRST and FOLLOW

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

FIRST and FOLLOW — *continued*

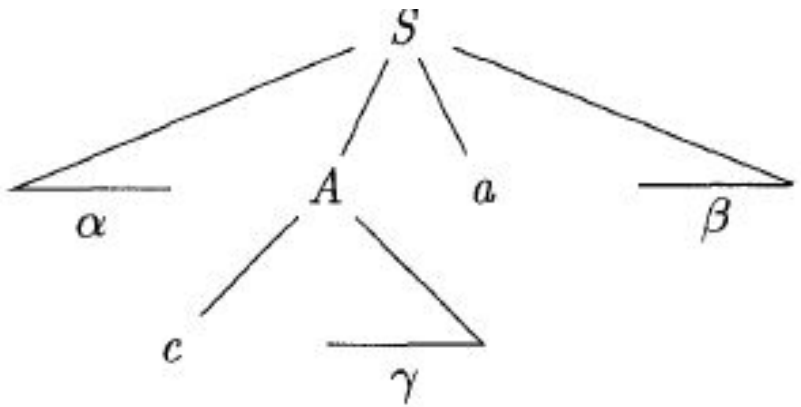
- Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.
- If $\alpha \Rightarrow \epsilon$, then ϵ is also in FIRST(α).
- For example, in figure $A \Rightarrow c\gamma$, so c is in FIRST(A).



Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST and FOLLOW — *continued*

- Define FOLLOW(A), nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form.
- That is, the set of terminals a such that there exists a derivation of the form $S \Rightarrow \alpha A a \beta$, 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 derived ϵ and disappeared.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST and FOLLOW — *continued*

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. For a production rule $X \rightarrow \in$, First(X) = { $\in$ }

2. For any terminal symbol 'a', First(a) = { a }

3. For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\epsilon \notin$ First(Y_1), then First(X) = First(Y_1)
- If $\epsilon \in$ First(Y_1), then First(X) = { First(Y_1) $- \epsilon$ } $\cup$ First(Y_2Y_3)
- Then, If $\epsilon \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\epsilon \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) $- \epsilon$ } $\cup$ First(Y_3)
- Similarly, we can make expansion for any production rule

$X \rightarrow Y_1Y_2Y_3.....Y_n$.

FIRST and FOLLOW — *continued*

To compute FOLLOW(*A*) for all nonterminals *A*, apply the following rules until nothing can be added to any FOLLOW et.

1. For the start symbol S, place $ in Follow(S).

2. For any production rule A → αB, Follow(B) = Follow(A)

3. For any production rule A → αBβ,

- If $\epsilon \notin$ First(β), then Follow(B) = First(β)
- If $\epsilon \in$ First(β), then Follow(B) = { First(β) – ϵ } ∪ Follow(A)

FIRST and FOLLOW — *continued*

- ϵ may appear in the FIRST function of a non-terminal.
- ϵ will never appear in the FOLLOW function of a non-terminal.
- Before calculating the FIRST and FOLLOW functions, eliminate Left Recursion from the grammar, if present.

Example

1. For a production rule $X \rightarrow \in$, First(X) = { $\in$ }
2. For any terminal symbol 'a', First(a) = { a }
3. For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\epsilon \notin$ First(Y_1), then First(X) = First(Y_1)
- If $\epsilon \in$ First(Y_1), then First(X) = { First(Y_1) $- \epsilon$ } $\cup$ First(Y_2Y_3)
- Then, If $\epsilon \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\epsilon \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) $- \epsilon$ } $\cup$ First(Y_3)
- Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Grammar,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid \mathbf{id}$

Then,

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

Example — *continued*

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid \textbf{id}$

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)	$,)	+, $,)	+, $,)	*, +, $,)

Practice problems- FIRST and FOLLOW

Problem-1 :

S-> ACB | CbB | Ba

A->da | BC

B-> g | ϵ

C-> h | ϵ

Problem-2 :

S-> (L) | a

L-> SL′

L′-> ,SL′ | ϵ

Problem-3:

S-> A

A-> aB | Ad

B-> b

HINT:
This grammar on prolem-3 is left recursive, you must eliminate left recursion before finding FIRST.

LL(1) Grammars

- Predictive parsers, that needs no backtracking, can be constructed for a class of grammars called LL(1).
- The first “L” in LL(1) stands for scanning the input from left to right.
- The second “L” for producing a leftmost derivation.
- And the “1” for using one input symbol of lookahead at each step to make parsing action decisions.

Algorithm for Construction of a Predictive Parsing Table

INPUT: Grammar *G*.

OUTPUT: Parsing table *M*.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$.

   If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table).

Algorithm for Construction of a Predictive Parsing Table

For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

LL(1) Grammars — *continued*

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G the following conditionshold:

1. For no terminal a do both α and β derive strings beginning with a. Meaning, FIRST(α) and FIRST(β) needs to be disjoint sets.
2. At most one of α and β can derive the empty string.
3. If $\beta \Rightarrow \epsilon$ then FIRST(α)and FOLLOW(A)needs to be disjoint. Likewise, $\alpha \Rightarrow \in$, then FIRST(β) and FOLLOW(A) needs to be disjoint.

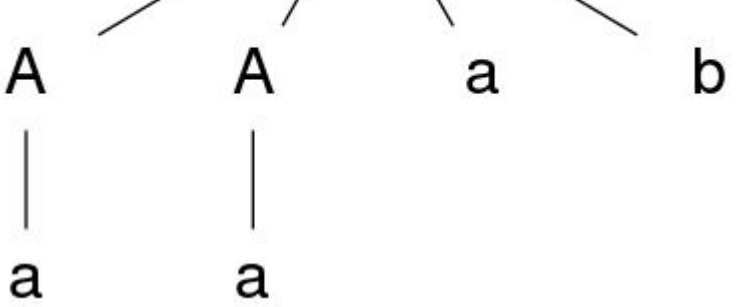
A Case of a non-LL(1) Grammar

$S \rightarrow AAab \mid BbBa$

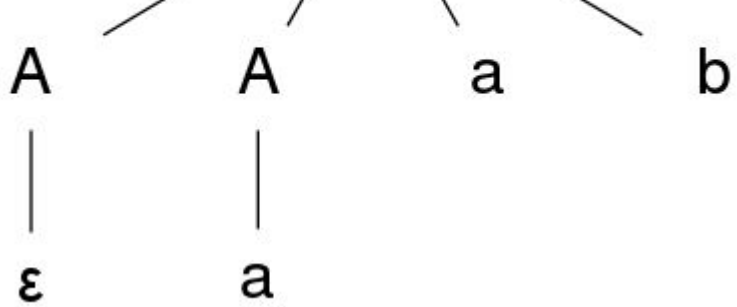
$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

aaab



aab



Example — *continued*

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

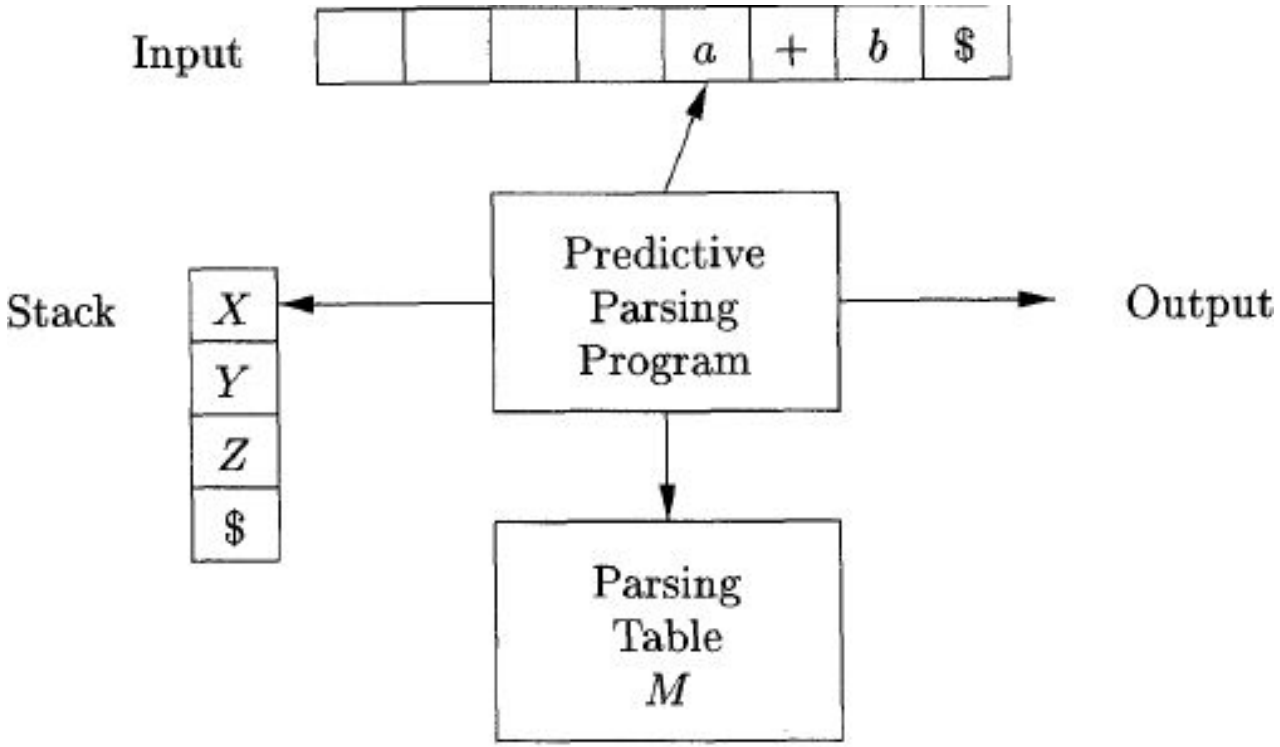
NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Nonrecursive Predictive Parsing

- A non-recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The 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

$$S \stackrel{lm}{\Longrightarrow} wa$$

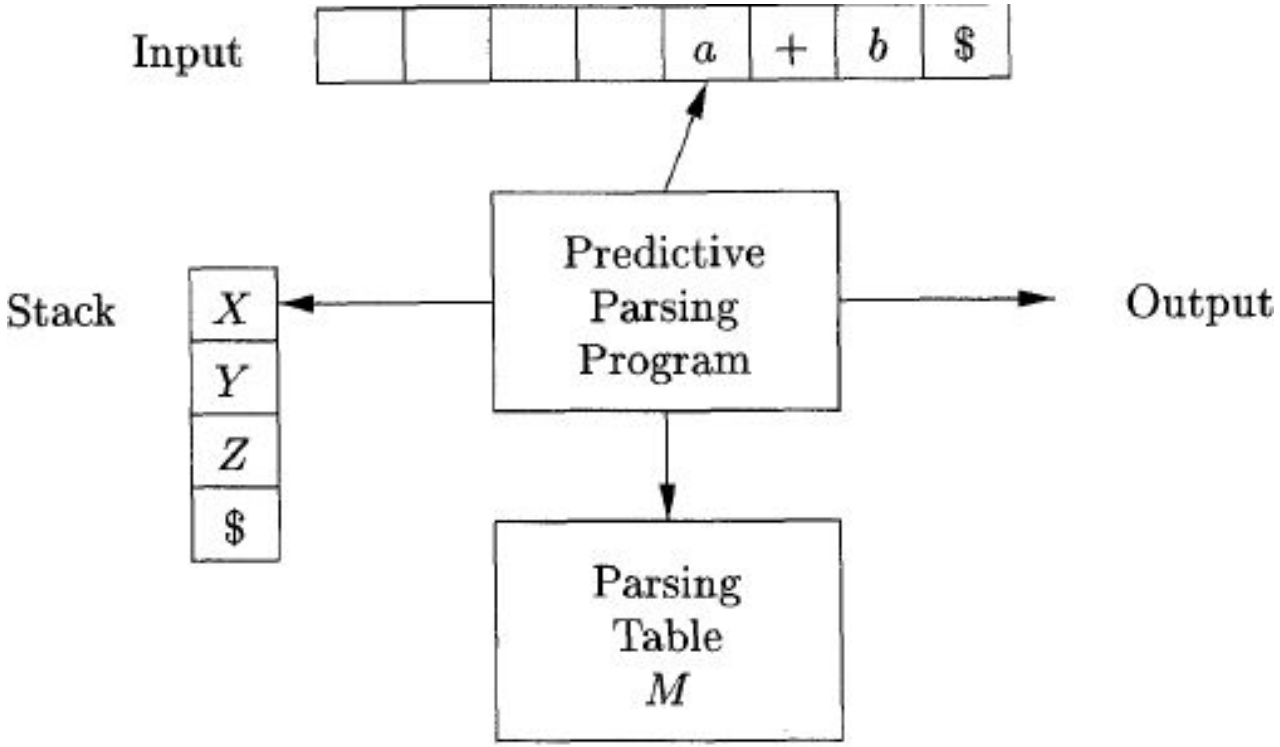
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- The table-driven parser in figure has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by algorithm, and an output stream.

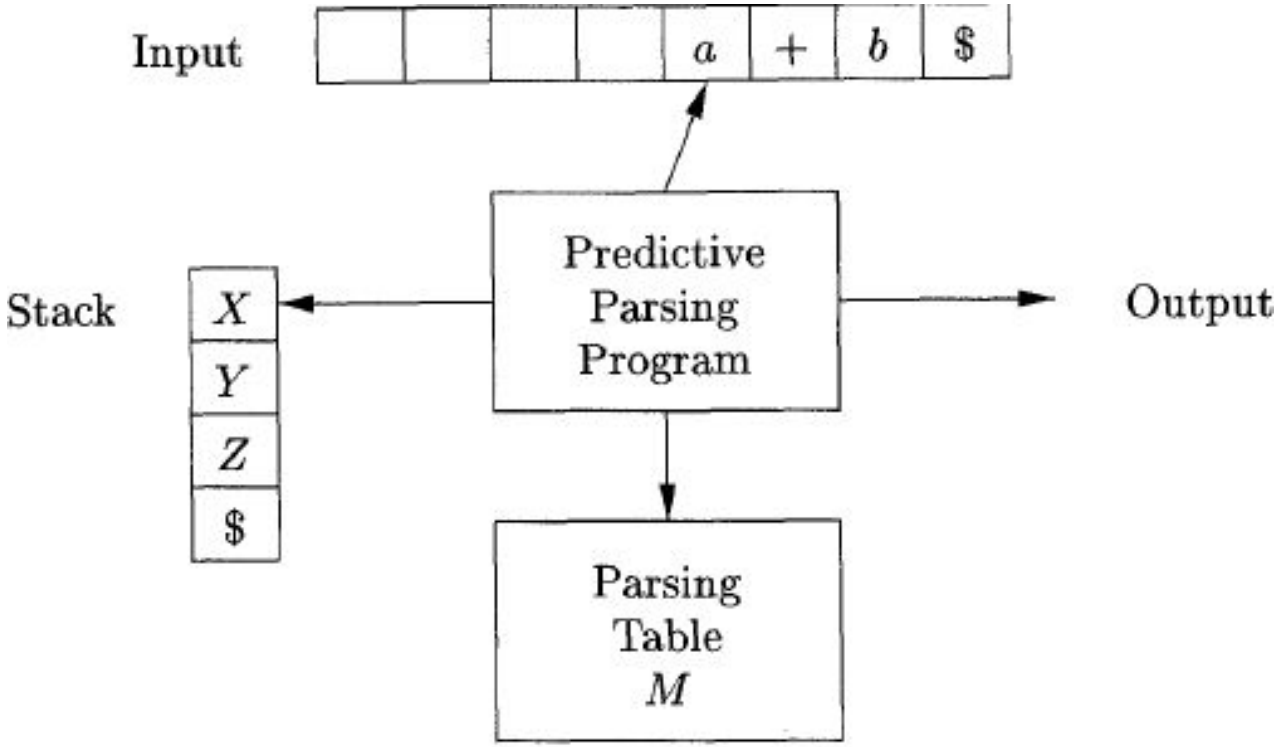
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- The parser is controlled by a program that considers X , the symbol on top of the stack, and a, the current input symbol.

Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- If *X* is a nonterminal, the parser chooses an *X* -production by consulting entry *M*[*X*, *a*] of the parsing table *M*.
- Additional code could be executed here, for example, code to construct a node in a parse tree.

Nonrecursive Predictive Parsing — *continued*

set ip to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if (X is a) pop the stack and advance ip;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    set X to the top stack symbol;
}

Predictive parsing algorithm

Example

- We consider grammar:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid s \\
T &\rightarrow \\
FT' &\rightarrow *FT' \mid s \\
T & \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned}
$$

- We have already seen its parsing table.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Example

- On input **id + id** $*$ **id**, the nonrecursive predictive parser algorithm makes the sequence of moves,

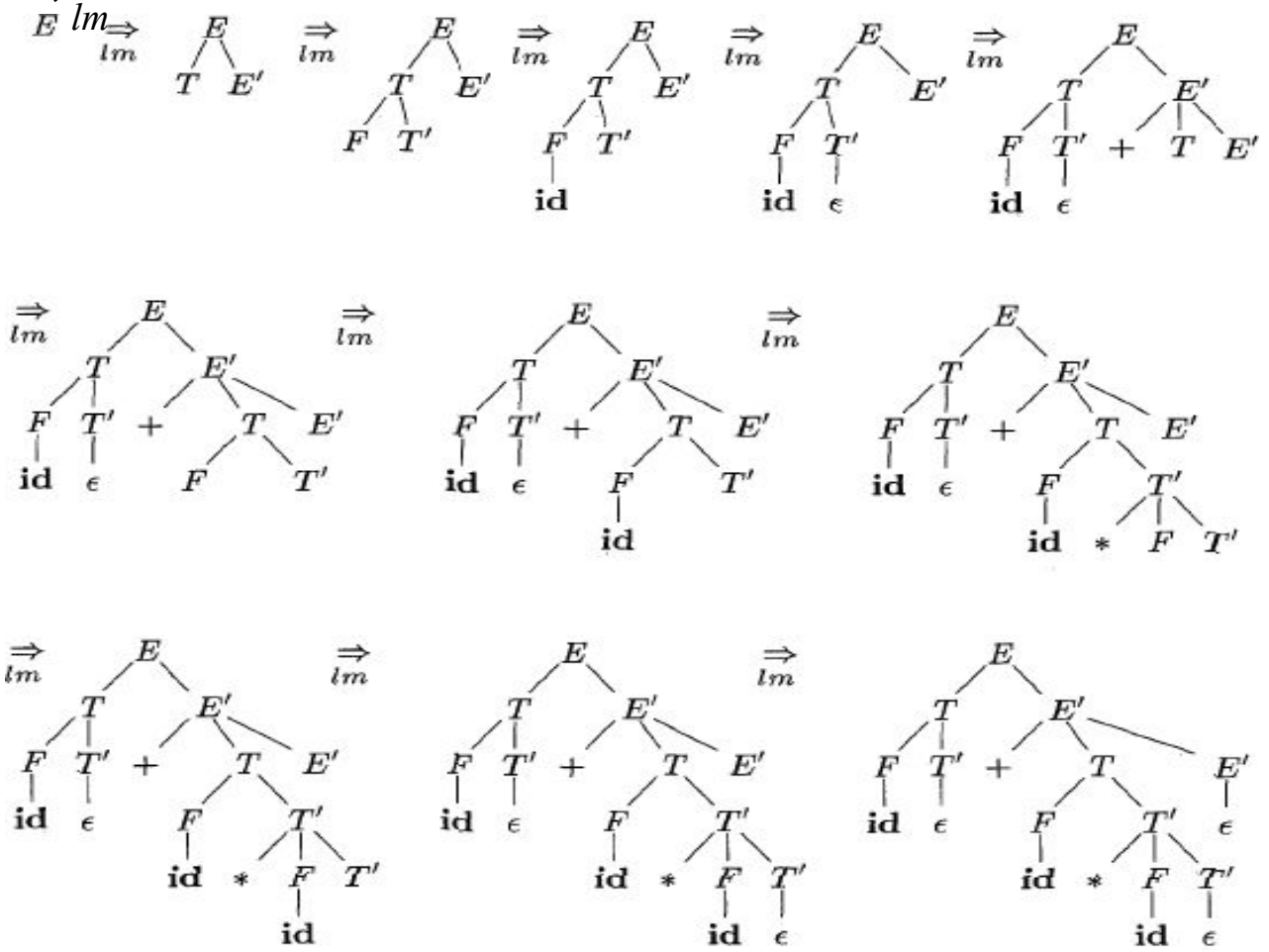
MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match **id**
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match **id**
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match **id**
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Example

These moves correspond to a leftmost derivation,

$$E \Rightarrow_{lm} TE' \Rightarrow_{lm} FT'E' \Rightarrow_{lm} \mathbf{id}\,T'E' \Rightarrow_{lm} \mathbf{id}\,E' \Rightarrow_{lm} \mathbf{id} + TE' \Rightarrow_{lm} \ldots$$



Top-down parse for **id + id * id**

Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

STACK	INPUT	ACTION
E $	**id + id** *	
↑	**id**$	
TE' $	↑	output *E* → *TE'*
	id + id *	
	id$	

Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

STACK	INPUT	ACTION
TE'$ ↑ *FTE*$	**id + id** * **id**$ ↑ **id + id** * **id**$	output $T \to FT'$

Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

STACK	INPUT	ACTION
FT'E'$	**id + id** *	
↑ , ,	**id**$	
id*T E* $	↑	output $F \to$ **id**
	id + id *	
	id$	

Example — *continued*

STACK	INPUT	ACTION
id*T'E'*\$ ↑	**id + id** * **id**\$ ↑	match **id**

Both are terminals and match. So, popped from the stack and input pointer advanced

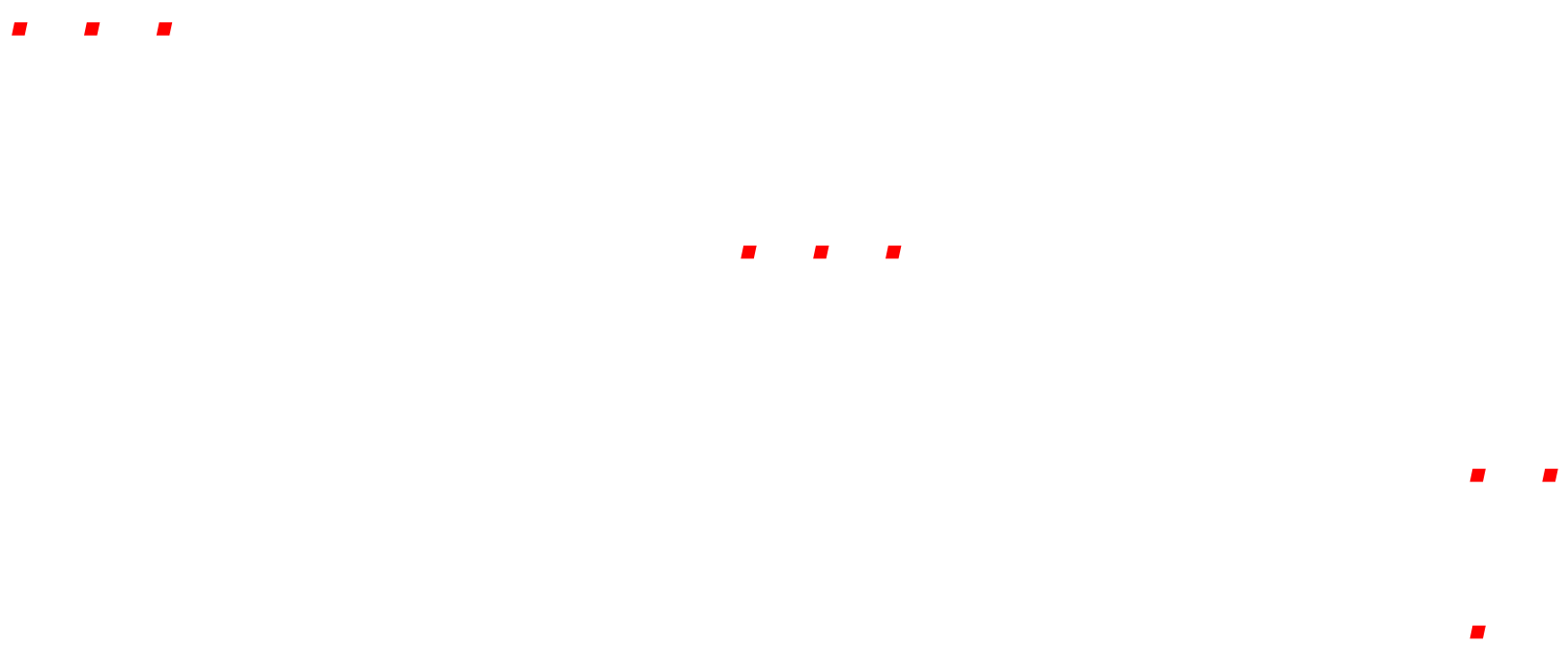
T'E'\$ ↑	**+id** *	**id**\$

Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

STACK	INPUT	ACTION
$T'E'\$$	**+id** $*$ **id**$	
$E\$$	**+id** $*$ **id**$	

Example — *continued*



Example — *continued*

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

STACK	INPUT	ACTION
$E'\$$	$\$$	
↑	↑	
$\$$	$\$$	

Example — *continued*

STACK	INPUT	ACTION
$ ↑	$ ↑	

Both are $, the parser halts and announces successful completion of parsing.

Example

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

For a leftmost derivation the production rules in the ACTION column (outputs only) are to be used from top to bottom.

Some useful links to check

You may check the following links where you can give input a grammar and will get specific output like:
elimination of left recursion, left factoring, finding FIRST and FOLLOW:

1. Eliminating Left Recursion
2. Left Factoring
3. Finding FIRST & FOLLOW

Thank You