Systems Programming Unit 4 : Compilers

Prof. Reshma Pise
Computer Engg. Dept
Vishwakarma University

Unit 4 : Compiler Design

- Introduction to Compiling
- Phases in Compilation
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing,
  - Bottom-Up Parsing,
  - Ambiguity in Grammar

Translators

- Programs written in high-level languages need to be translated into low-level (machine code) for processing and execution by the CPU. This is done by a translator program.

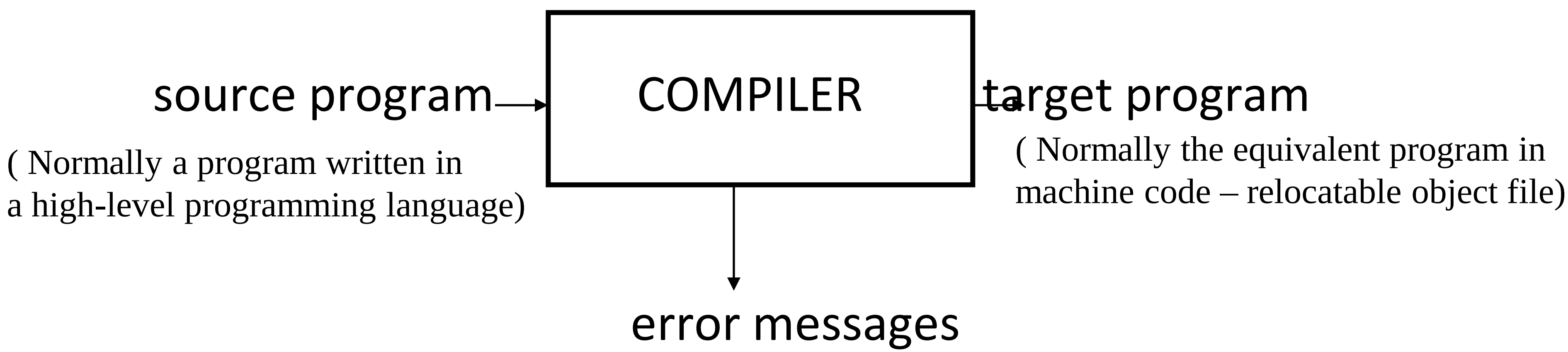
- There are two types of translator program:

  interpreters

  compilers

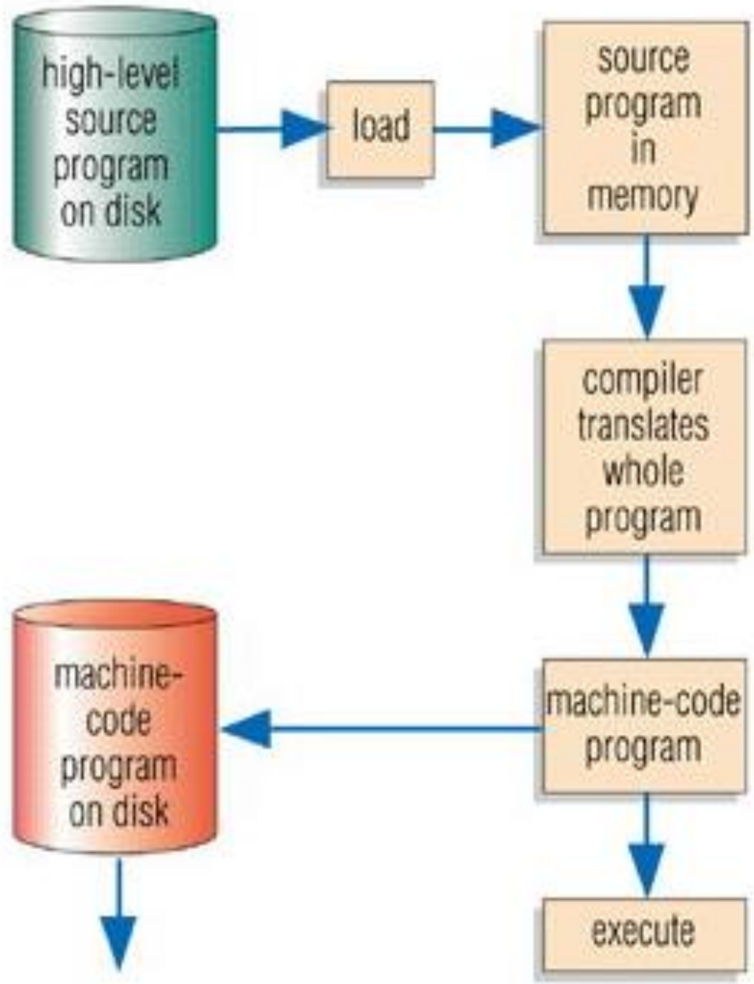
COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Compilers

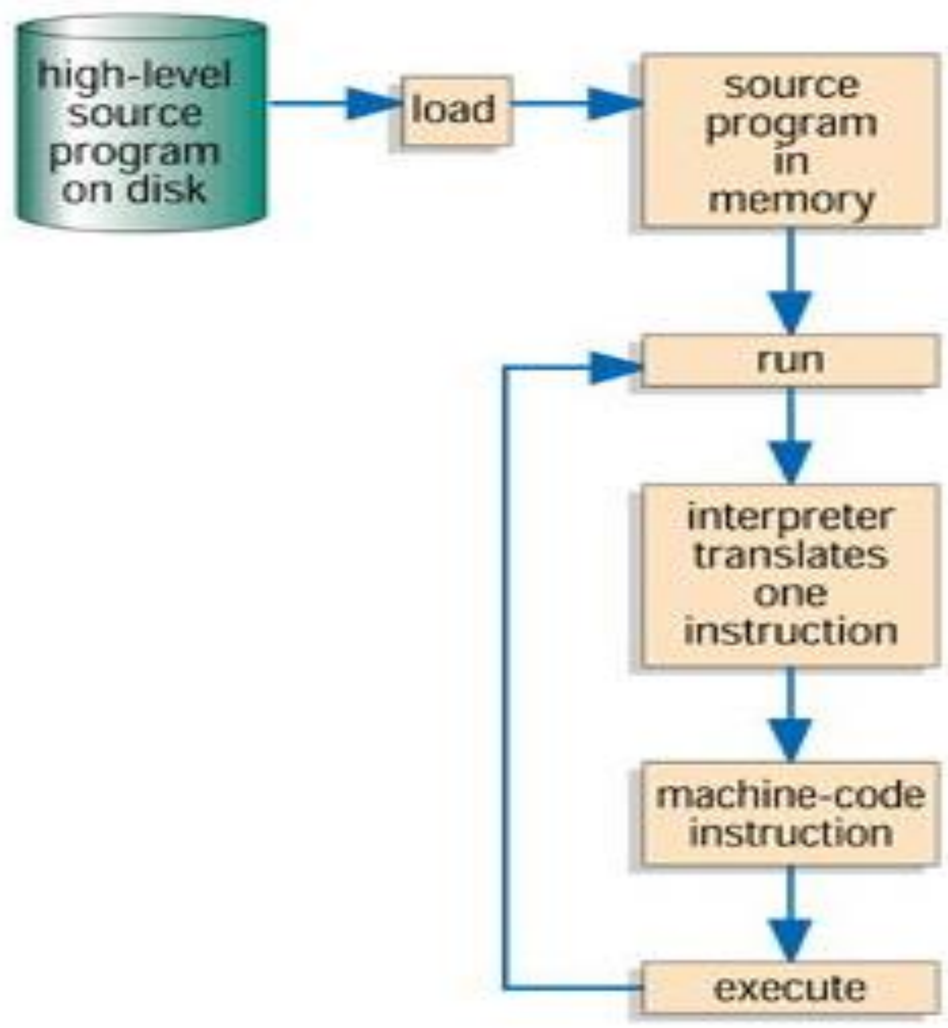
A Compiler program translates the whole program into a machine code version that can be run without the compiler being present.



Advantage: program runs fast as already in machine code, translator program only needed at the time of compiling

Disadvantage: slow to compile as whole program translated

Interpreters



Interpreter program translates HLL code into machine code one line at a time.

Advantage: easy to find errors, better for learners

Disadvantage: program runs slow as have to be continually interpreted, interpreter program always in memory to interpret program.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

  Analysis - > Front End

  Synthesis - > Back end

- In analysis phase, an intermediate representation is created from the given source program.

- In synthesis phase, the equivalent target program is created from this intermediate representation.

Phases of A Compiler

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

- Each phase transforms the source program from one representation into another representation.

- They communicate with error handlers.

- They communicate with the symbol table.

1. Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

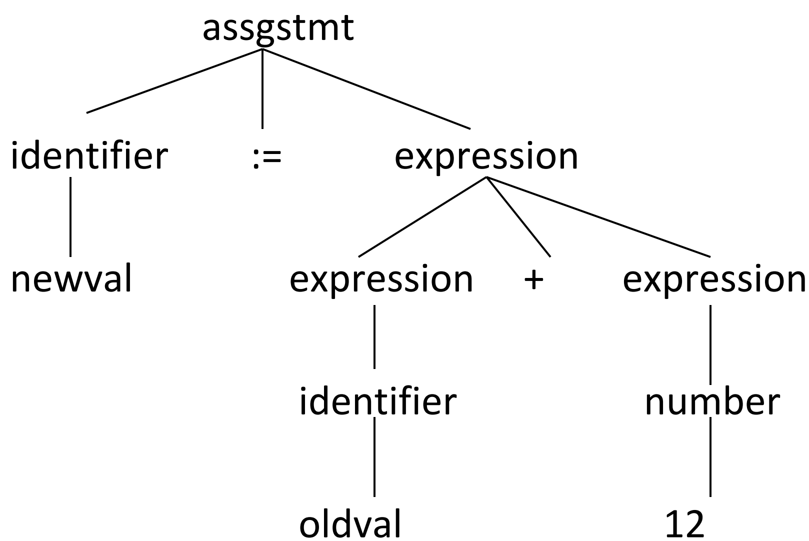
Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

2. Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

  assgstmt -> identifier := expression

  expression -> identifier

  expression -> number

  expression -> expression + expression

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with the simple non-recursive constructs of the language.
  - The syntax analyzer deals with the recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

3. Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

    - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

4. Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is closer to the level of machine codes.
- Ex:

newval := oldval * fact + 1

↓

id1 := id2 * id3 + 1

↓

temp1 := id2 * id3 *Intermediates Codes (three address code)*

temp2 := temp1 + 1

id1 := temp2

Intermediate Code Generation

- Properties of IR-

  Easy to produce

  Easy to translate into the target program
- IR can be in the following forms-

  Syntax trees

  Postfix notation

  Three address statements
- Properties of three address statements-

  At most one operator in addition to an assignment operator

  Must generate temporary names to hold the value computed at each

  instruction

  Some instructions have fewer than three operands

5. Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

    temp1 := id2 * id3

    id1 := temp1 + 1

6. Code Generator

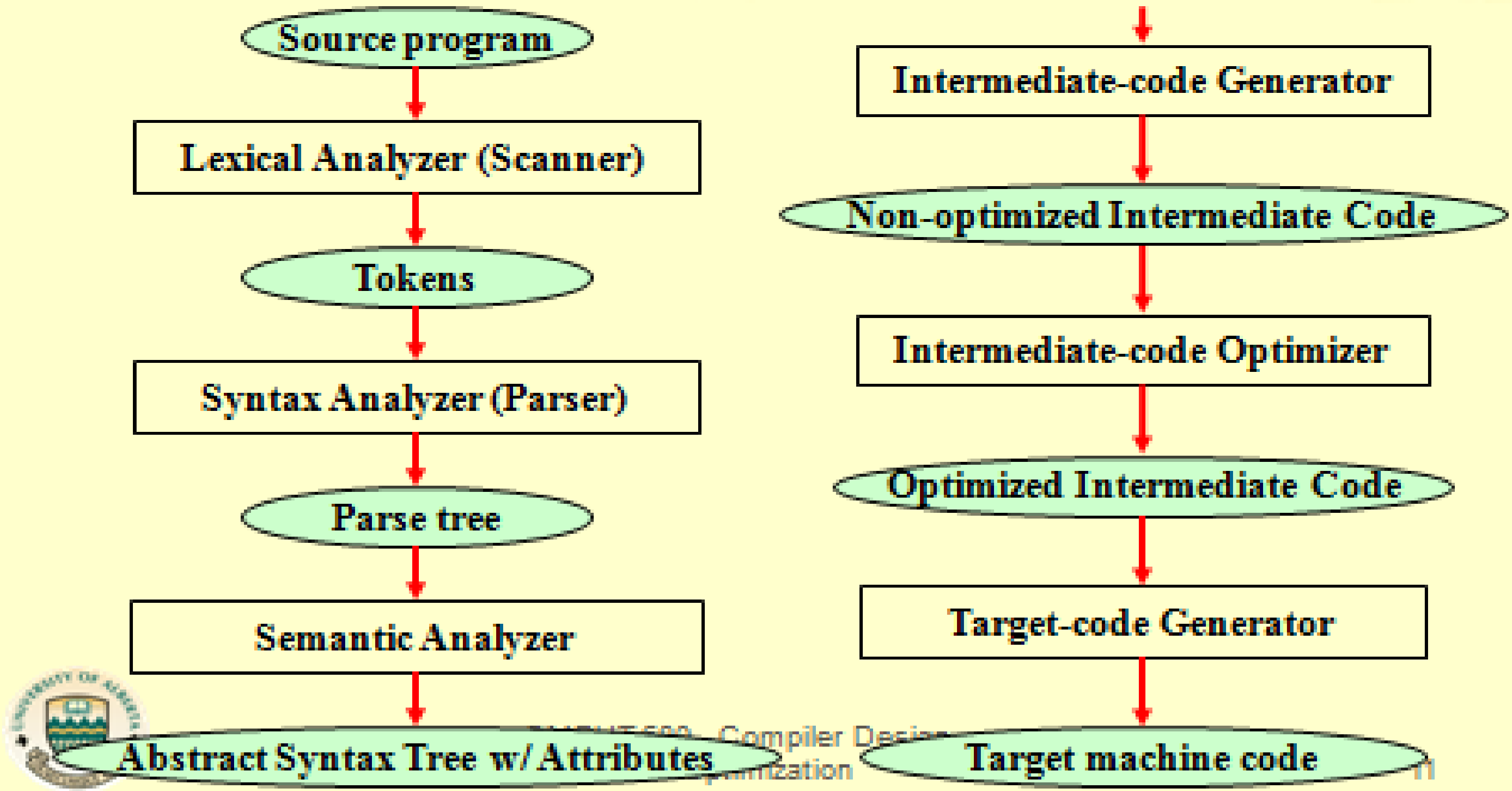
- Produces the target language in a specific architecture.
- The target program is normally a relocatable object file containing the machine code or assembly code.
- Intermediate instructions are each translated into a sequence of machine instructions that perform the same task.
- Ex:

(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

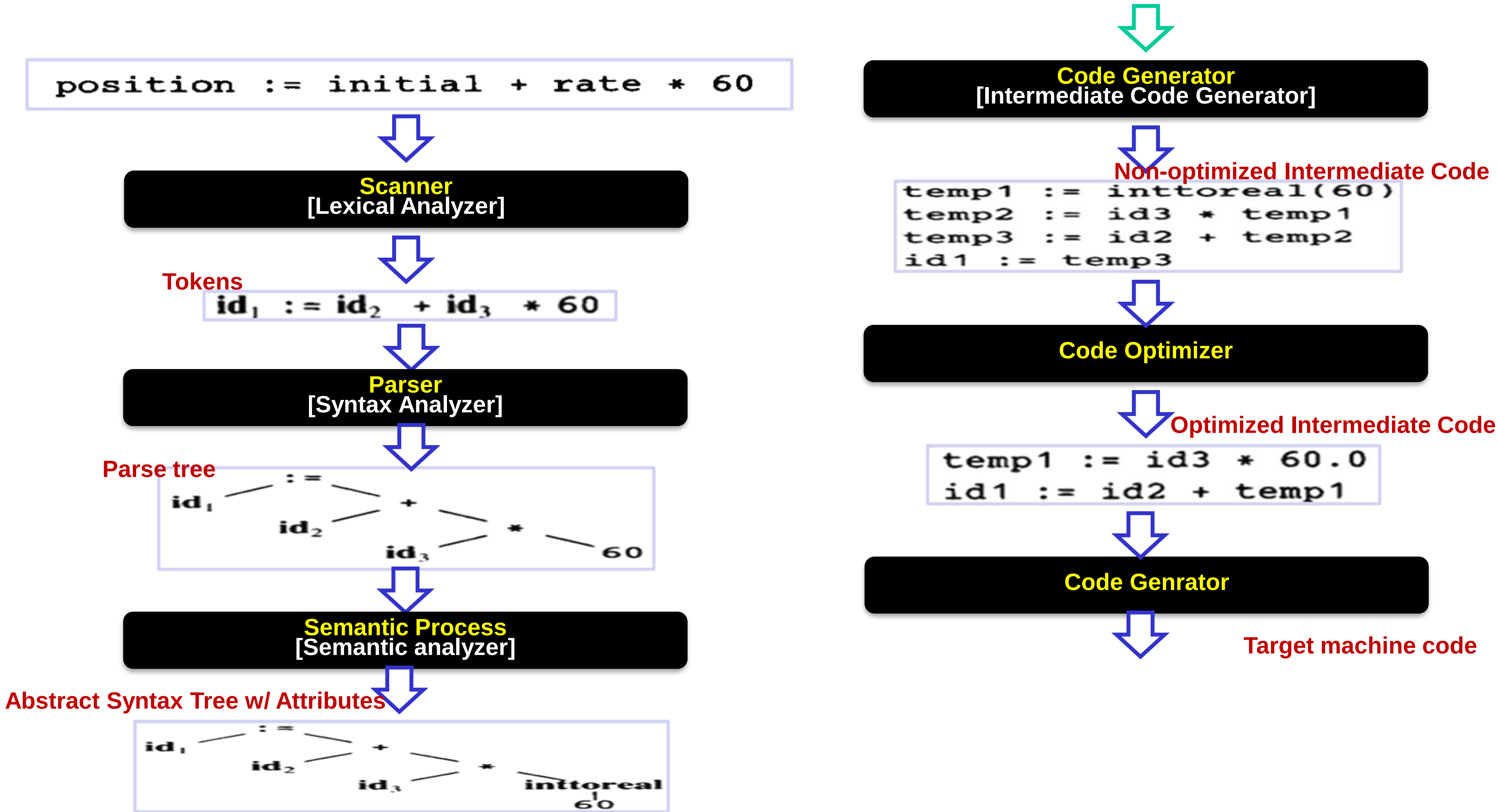
```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Phases of compiler

Phases of a Compiler



The Structure of a Compiler



The input program as you see it.

```
main ()
{
     int i,sum;
    sum = 0;
    for (i=1; i<=10; i++);
    sum = sum + i;
    printf("%d\n",sum);
}
```

The same program – as the compiler sees it (initially).

```
main␣()↩{↩␣␣␣␣int␣i,sum;↩␣␣␣␣sum␣=␣0;↩␣␣␣␣
for␣(i=1;␣i<=10;␣i++);␣␣␣␣sum␣=␣sum␣+␣i;↩␣␣␣␣
printf("%d\n",sum);↩}
```

How do you make the compiler see what you see?

Lexemes, Tokens and Patterns

Definition: *Lexical analysis* is the operation of dividing the input program into a sequence of *lexemes* (*tokens*).

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i`, `sum`, `for`, `10`, `++`, `"%d\n"`, `<=`.

Lexemes, Tokens and Patterns

Definition: *Lexical analysis* is the operation of dividing the input program into a sequence of *lexemes* (*tokens*).

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i, sum, for, 10, ++, "%d\n", <=`.
- *tokens* – sets of similar lexemes.
  Examples –
  identifier = {`i, sum, buffer`, . . .}
  int_constant = {`1, 10`, . . .}
  addop = {`+, -`}

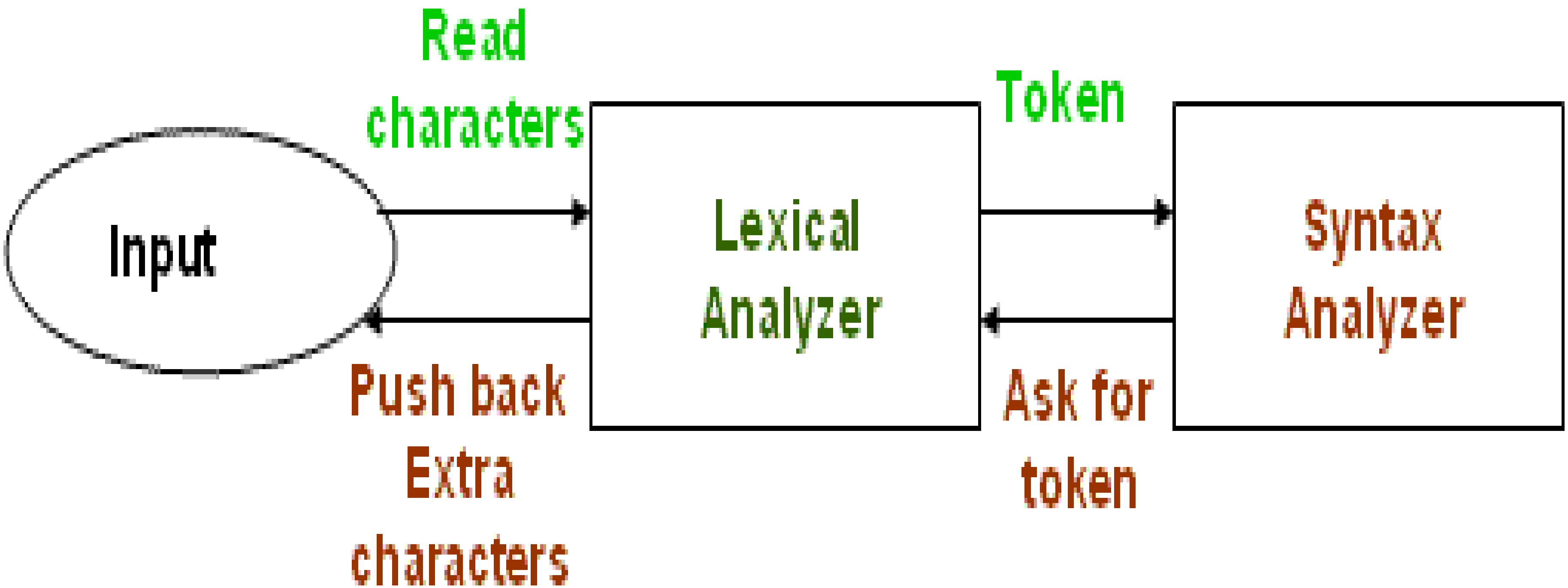
Lexemes, Tokens and Patterns

Things that are not counted as lexemes –

- white spaces – tab, blanks and newlines
- comments

These too have to be detected and ignored.

Role of the Lexical Analyzer



Tasks of Lexical Analyzer

- Reads source text and detects the token
- Stripe out comments, white spaces, tab, newline characters.
- Correlates error messages from compilers to source program

Approaches to implementation

. Use assembly language- Most efficient but most difficult to implement

. Use high level languages like C- Efficient but difficult to implement

. Use tools like lex, flex- Easy to implement but not as efficient as the first two cases

LEXICAL ERRORS

Primarily of two kinds:

1. Lexemes whose length exceed the bound specified by the language.
   - In Fortran, an identifier more than 7 characters long is a lexical error.
   - Most languages have a bound on the precision of numeric constants. A constant whose length exceeds this bound is a lexical error.
2. Illegal characters in the program.
   - The characters ~, & and @ occuring in a Pascal program (but not within a string or a comment) are lexical errors.
3. Unterminated strings or comments.

Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
   - Possibly more efficient.
2. *Use a generator* – To generate the lexical analyser from a formal description.
   - The generation process is faster.
   - Less prone to errors.

Automatic Generation of Lexical Analysers

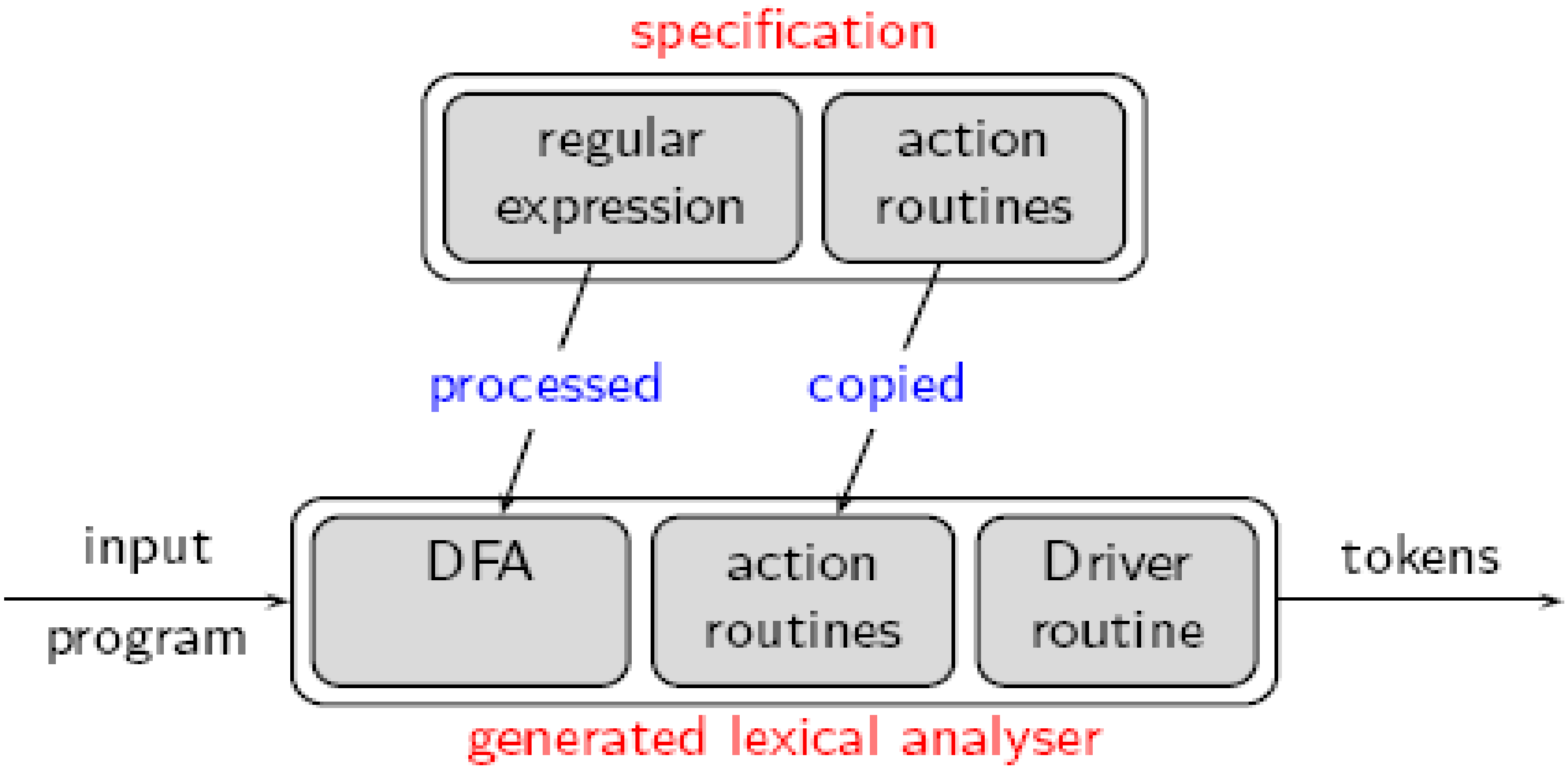
Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, consisting of:
  - a regular expression describing each token, and
  - a code fragment describing the action to be performed, on identifying each token.

The generated lexical analyser consists of:

- A *deterministic finite automaton (DFA)* constructed from the token specification.
- A *code fragment* (a driver routine) which can traverse *any DFA*.
- Code for the *action specifications*.

Automatic Generation of Lexical Analysers

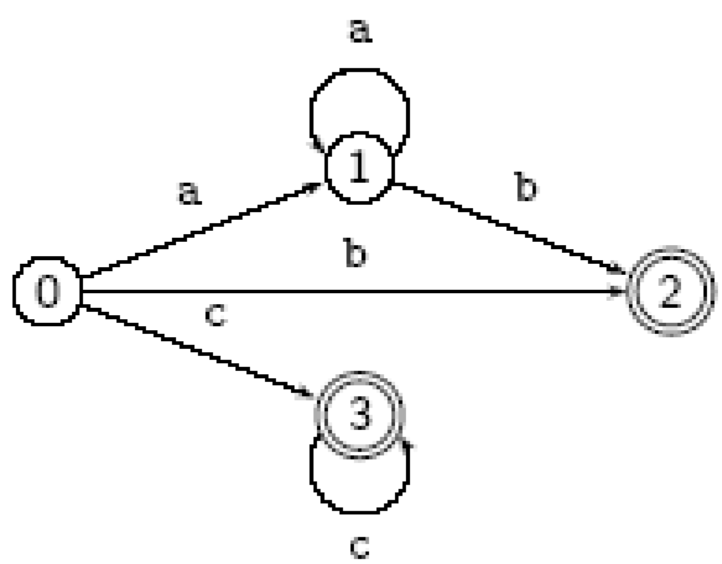


Example of Lexical Analyser Generation

Suppose a language has two tokens

Pattern	Action
`a*b`	`{ printf( "Token 1 found");}`
`c+`	`{ printf( "Token 2 found");}`

From the description, construct a structure called a deterministic finite automaton (DFA).



Example of Lexical Analyser Generation

In summary:

- The DFA, the driver routine and the action routines taken together, constitute the lexical analyser.
-
  - actions are supplied as part of specification.
  - driver routine is common to all generated lexical analyzers

  The only issue – how are the patterns, specified by regular expressions, converted to a DFA.

  In two steps:

  - Convert regular expression into NFA.
  - Convert NFA to DFA.

Example of Lexical Analyser Generation

Consider a language with the following tokens:

- *begin* – representing the lexeme `begin`
- *integer* – Examples: 0, -5, 250
- *identifier* – Examples: `a`, `A1`, `max`

lex Programming Utility

General Information:

- Input is stored in a file with *.l extension
- File consists of three main sections
- lex generates C function stored in lex.yy.c

Using lex:

1) Specify words to be used as tokens (Extension of regular expressions)
2) Run the lex utility on the source file to generate **yylex( )**, a C function
3) Declares global variables char* yytext and int yyleng

lex Programming Utility

Three sections of a lex input file:

```
/* C declarations and #includes lex definitions */
%{ #include “header.c”
int i; }%

%%
/* lex patterns and actions */
{INT}          {sscanf (yytext, “%d”, &i);
                printf(“INTEGER\n”);}

%%
/* C functions called by the above actions */
{ yylex(): }
```

flex - fast lexical analyzer generator

- Flex is a tool for generating scanners.
- Flex source is a table of regular expressions and corresponding program fragments.
- Generates *`lex.yy.c`* which defines a routine *`yylex()`*

Format of the Input File

- The flex input file consists of three sections, separated by a line with just %% in it:

```
definitions
%%
rules
%%
user code
```

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
```

Rules Section

- The rules section of the flex input contains a series of rules of the form:

  ```
  pattern action
  ```

- Example:

  ```
  {ID} printf( "An identifier: %s\n", yytext );
  ```

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

Action

- If the action contains a `'{'`, the action spans till the balancing `'}'` is found, as in C.
- An action consisting only of a vertical bar (`'|'`) means "same as the action for the next rule.“
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

Precedence Problem

- For example: a “<“ can be matched by “<“ and “<=“.
- The one matching most text has higher precedence.
- If two or more have the same length, the rule listed first in the flex input has higher precedence.

User Code Section

- The user code section is simply copied to `lex.yy.c` verbatim.
- The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with the `%{}`'s removed).

A Simple Example

```
%{
   int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()  {
   yylex();
   printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
}
```

Syntax Analyzer

Introduction to the parser

- Context-free grammars
- Writing a grammar

- Using ambiguous grammars

Syntax Analyzer

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.

Context Free Grammar

A CFG G is formally defined to have four components (N, T, S, P):

1. T is a finite set of terminals.
2. N is a finite set of nonterminals.
3. S is a special nonterminal (from N) called the *start* symbol.
4. P is a finite set of production rules of the form such as $A \rightarrow \alpha$, where A is from N and α from $(N \bigcup T)^*$

$$
\begin{array}{lcl}
\mathit{declaration} & \rightarrow & \mathit{type}\ \mathit{idlist}\ ; \\
\mathit{idlist} & \rightarrow & \mathbf{id} \mid \mathit{idlist}\ ,\ \mathbf{id} \\
\mathit{type} & \rightarrow & \texttt{integer} \mid \texttt{real}
\end{array}
$$

start symbol

terminals

non-terminals

production

Derivation

Example : $G = (\{list\}, \{\mathbf{id}, \ ,\}, list, \{list \rightarrow list, \mathbf{id}, \ \ list \rightarrow \mathbf{id}\})$

A derivation is traced out as follows

$$\begin{aligned} list &\Rightarrow list, \mathbf{id} \\ &\Rightarrow list, \mathbf{id}, \mathbf{id} \\ &\Rightarrow \mathbf{id}, \mathbf{id}, \mathbf{id} \end{aligned}$$

- The transformation of a string of grammar symbols by replacing a non-terminal by the corresponding right hand side of a production is called a *derivation*.
- The set of all possible *terminal strings* that can be derived from the start symbol of a CFG is the language generated by the CFG.

This grammar generates a list of one or more **id**s separated by commas.

Why the Term Context Free ?

Why the term context free ?

1. The only kind of productions permitted are of the form *non-terminal* $\longrightarrow$ *sequence of terminals and non-terminals*
2. Rules are used to replace an occurrence of the lhs non-terminal by its rhs. The replacement is made regardless of the context (symbols surrounding the non-terminal).

Notational Conventions

Symbol type	Convention
single terminal	letters `a, b, c, operators` delimiters, keywords
single nonterminal	letters A, B, C and names such as *declaration* , *list* and S is the start symbol
single grammar symbol (symbol from $\{N \cup T\}$)	X, Y, Z
string of terminals	letters `x , y , z`
string of grammar symbols	α, β, γ
null string	ϵ

FOrmal Definitions

- The *language* $L(G)$ generated by a context free grammar G is defined as $\{w \mid S \overset{+}{\Longrightarrow} w, w \in T^*\}$. Strings in $L(G)$ are called *sentences* of G.
- A string α, $\alpha \in (N \bigcup T)^*$, such that $S \overset{*}{\Longrightarrow} \alpha$, is called a *sentential form* of G.
- Two grammars are *equivalent*,if they generate the same language.

Basic Concepts in Parsing

- For constructing a derivation, there are choices at each sentential form.
  - choice of the nonterminal to be replaced
  - choice of a rule corresponding to the nonterminal.
- Instead of choosing the nonterminal to be replaced, in an arbitrary fashion, it is possible to make an uniform choice at each step.
  - replace the leftmost nonterminal in a sentential form
  - replace the rightmost nonterminal in a sentential form

  The corresponding derivations are known as *leftmost* and *rightmost* derivations respectively.
- Given a sentence w of a grammar G, there are several distinct derivations for w.

Parse Trees

A *parse tree* is a pictorial form of depicting a derivation.

1. root of the tree is labeled with S
2. each leaf node is labeled by a token or by ϵ
3. an internal node of the tree is labeled by a nonterminal
4. if an internal node has A as its label and the children of this node from left to right are labeled with $X_1, X_2, \ldots, X_n$ then there must be a production

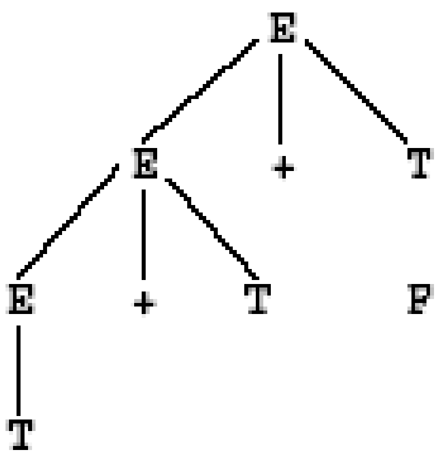
$$A \rightarrow X_1 X_2 \ldots X_n$$

where X_i is a grammar symbol.

Illustration

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathtt{id}
\end{aligned}
$$

The parse tree:



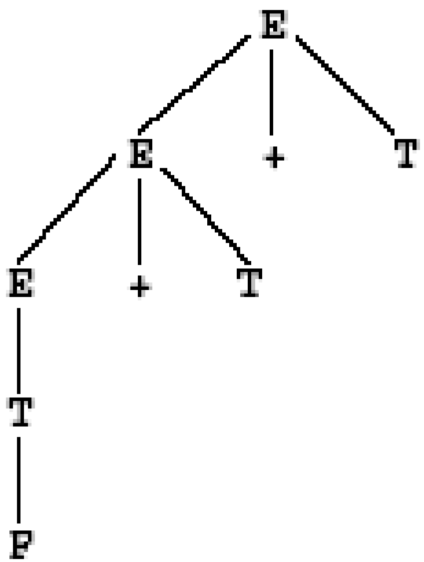
Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{E} + T + T \\
&\Rightarrow \underline{T} + T + T
\end{aligned}
$$

Illustration

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathtt{id}
\end{aligned}
$$

The parse tree:



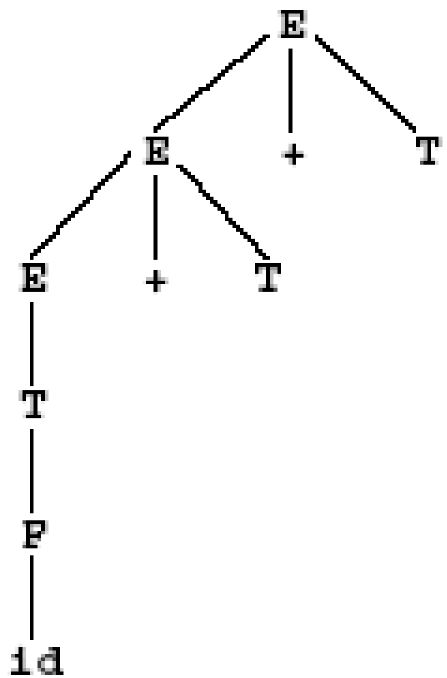
Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{E} + T + T \\
&\Rightarrow \underline{T} + T + T \\
&\Rightarrow \underline{F} + T + T
\end{aligned}
$$

Illustration

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

The parse tree:



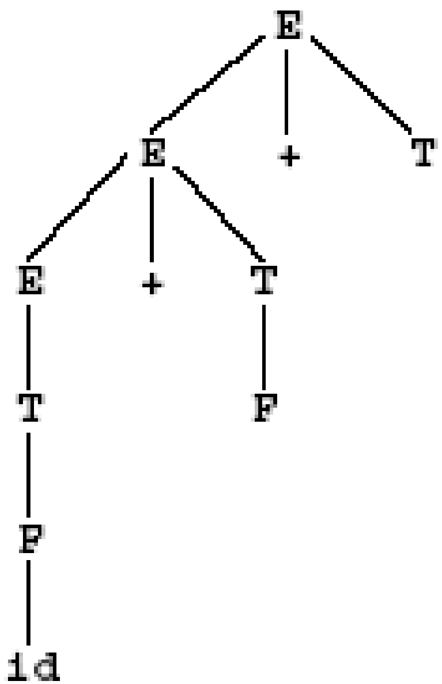
Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \Rightarrow & \underline{E} + T \\
 & \Rightarrow & \underline{E} + T + T \\
 & \Rightarrow & \underline{T} + T + T \\
 & \Rightarrow & \underline{F} + T + T \\
 & \Rightarrow & id + \underline{T} + T
\end{array}
$$

Illustration

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \texttt{id}
\end{aligned}
$$

The parse tree:



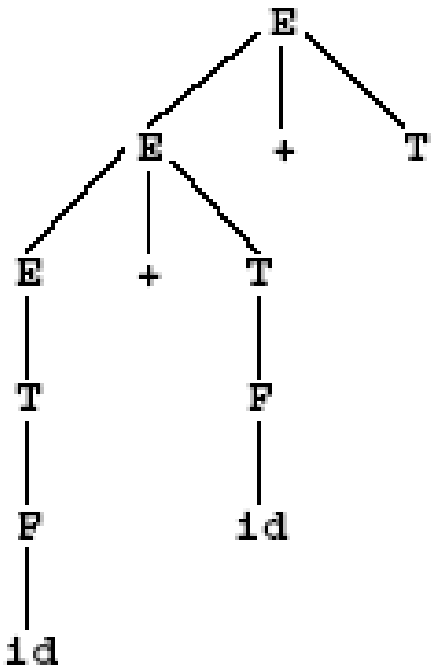
Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{E} + T + T \\
&\Rightarrow \underline{T} + T + T \\
&\Rightarrow \underline{F} + T + T \\
&\Rightarrow id + \underline{T} + T \\
&\Rightarrow id + \underline{F} + T
\end{aligned}
$$

Illustration

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathtt{id} \end{array}$$

The parse tree:



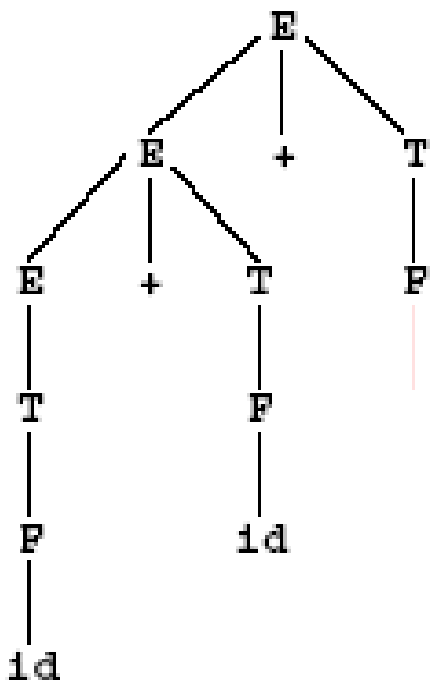
Leftmost derivation:

$$\begin{array}{lcl} \underline{E} & \Rightarrow & \underline{E} + T \\ & \Rightarrow & \underline{E} + T + T \\ & \Rightarrow & \underline{T} + T + T \\ & \Rightarrow & \underline{F} + T + T \\ & \Rightarrow & id + \underline{T} + T \\ & \Rightarrow & id + \underline{F} + T \\ & \Rightarrow & id + id + \underline{T} \end{array}$$

Illustration

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \texttt{id}
\end{aligned}
$$

The parse tree:



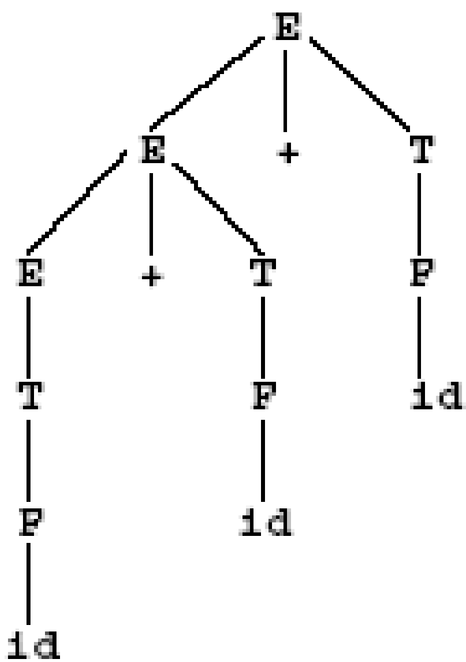
Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{E} + T + T \\
&\Rightarrow \underline{T} + T + T \\
&\Rightarrow \underline{F} + T + T \\
&\Rightarrow id + \underline{T} + T \\
&\Rightarrow id + \underline{F} + T \\
&\Rightarrow id + id + \underline{T} \\
&\Rightarrow id + id + \underline{F}
\end{aligned}
$$

Illustration

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathtt{id}
\end{aligned}
$$

The parse tree:



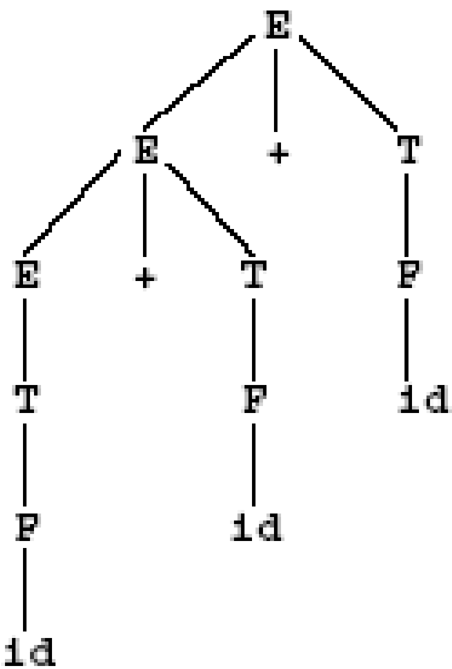
Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{E} + T + T \\
&\Rightarrow \underline{T} + T + T \\
&\Rightarrow \underline{F} + T + T \\
&\Rightarrow id + \underline{T} + T \\
&\Rightarrow id + \underline{F} + T \\
&\Rightarrow id + id + \underline{T} \\
&\Rightarrow id + id + \underline{F} \\
&\Rightarrow id + id + id
\end{aligned}
$$

Illustration

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

The parse tree:



Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \Rightarrow & \underline{E} + T \\
& \Rightarrow & \underline{E} + T + T \\
& \Rightarrow & \underline{T} + T + T \\
& \Rightarrow & \underline{F} + T + T \\
& \Rightarrow & id + \underline{T} + T \\
& \Rightarrow & id + \underline{F} + T \\
& \Rightarrow & id + id + \underline{T} \\
& \Rightarrow & id + id + \underline{F} \\
& \Rightarrow & id + id + id
\end{array}
$$

Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \Rightarrow & E + \underline{T} \\
& \Rightarrow & E + \underline{F} \\
& \Rightarrow & \underline{E} + id \\
& \Rightarrow & E + \underline{T} + id \\
& \Rightarrow & E + \underline{F} + id \\
& \Rightarrow & \underline{E} + id + id \\
& \Rightarrow & \underline{T} + id + id \\
& \Rightarrow & \underline{F} + id + id \\
& \Rightarrow & id + id + id
\end{array}
$$

Derivations and Parse Trees

The following summarize some interesting relations between the two concepts

- Parse tree filters out the choice of replacements made in the sentential forms.
- Given a left (right) derivation for a sentence, one can construct a unique parse tree for the sentence.
- For every parse tree for a sentence there is a unique leftmost and a unique rightmost derivation.
- Can a sentence have more than one distinct parse trees, and therefore more than one left (right) derivations?

Syntax Analysis

How are parsers constructed ?

- Till early seventies, parsers (in fact all of the compiler) were written manually.
- A better understanding of parsing algorithms has resulted in tools that can automatically generate parsers.
- Examples of parser generating tools:
  - Yacc/Bison: Bottom-up (LALR) parser generator
  - Antlr: Top-down (LL) scanner cum parser generator.

Parsers (cont.)

- We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.

- unambiguous grammar

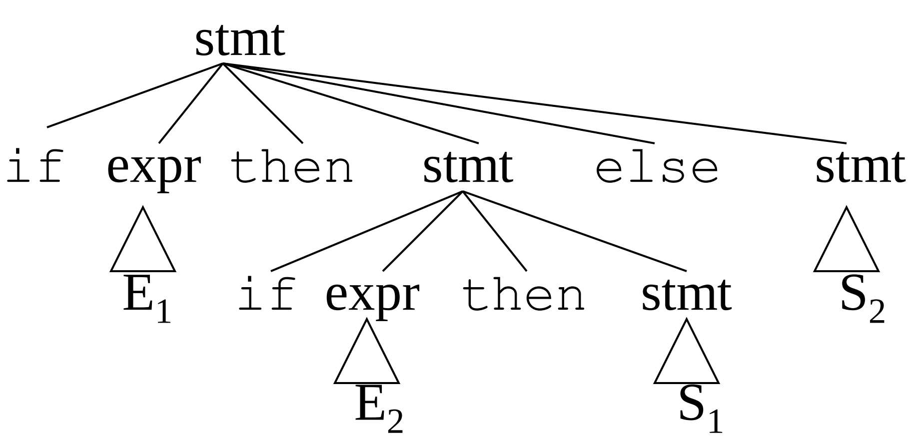
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

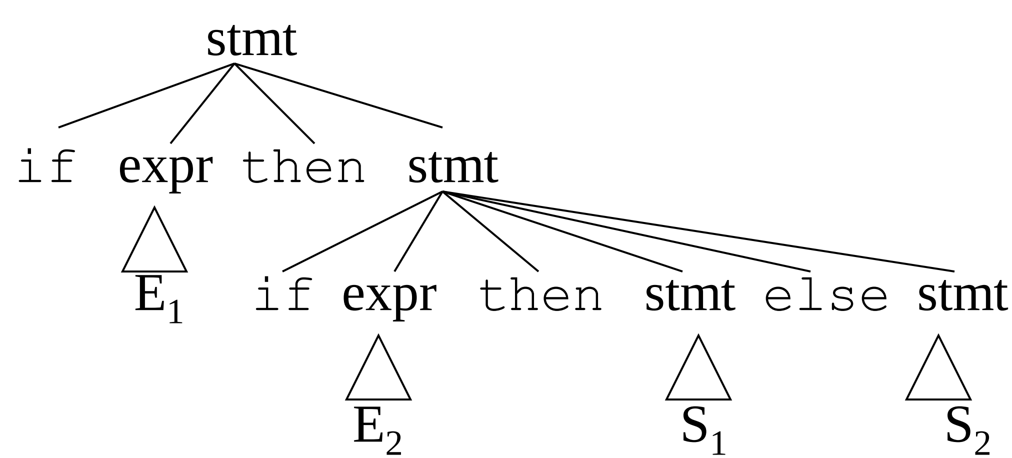
Ambiguity (cont.)

stmt → `if` expr `then` stmt |
`if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



1



2

Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt | unmatchedstmt

matchedstmt → `if` expr `then` matchedstmt `else` matchedstmt | otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

Thank You