Compiler Design

Presented By:
Prof. M. M.Math
Department of CSE
E-mail : mmmath@git.edu. mmmath@rediffmail.com
Phone: 9945001309

Topics covered

- Introduction to Language processor
- Phases of a Compiler
- Evolution of Programming languages
- The Role of Lexical Analyser
- Buffering
- Specification of Tokens
- Recognition of Tokens

Introduction to Language processor

- What is Language processing ?
- Examples of language processors
  - Compiler and Interpreter
  - Comparisons between Complier and interpreter
  - Hybrid compiler
  - Just-In-Time compiler
  - Incremental Compiler
  - Cross Compiler

  And Language processing system

What is Language processing?

- The different steps involved in converting instructions in high level language to machine level code is called language processing. There are different components involved in language processing.

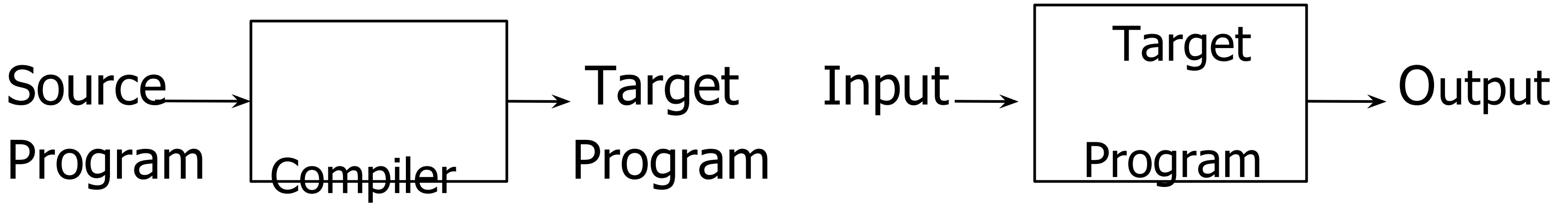
  They are Pre-processor, Compiler and Assembler.

Examples of Language processor

- Compiler

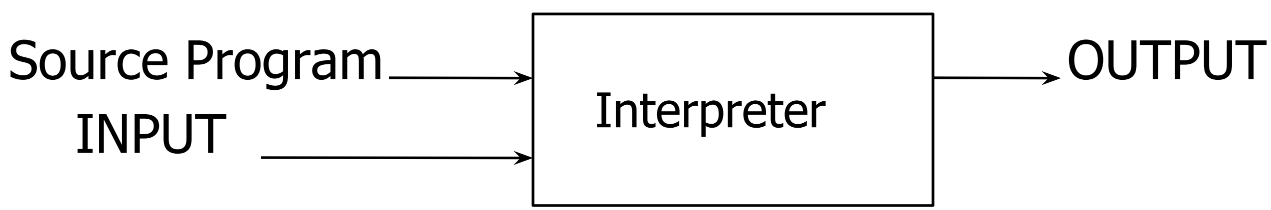
  It is a Translator program that can read program written in one language (Source language) and translate it into an equivalent programs in another language (target language). It also generates any error in the source program that it detects in the translation process

  The target program is executable machine language, called by the user to process the inputs and produce the required outputs.



Interpreter

It is also a translator program that can read program line by line and checks for the syntax if everything is correct then program is directly executed.



Compiler Vs Interpreters

- Generates target code with or without generating any intermediate code.
- Some of the languages which use interpreter are BASIC, LISP etc.
- Interpreter takes each line of source program and converts it to target code. But compiler considers entire program as a single unit. It scans the entire program generates intermediate code for the entire program and then only converts it to target code.
- As interpreters may not generate intermediate code there is less scope for code optimization. If speed of execution is primary concern then compilers are preferred, but if time taken to generated target code is primary concern then interpreters are referred.
- ***Used during development.***

Compiler	Interpreter
1.Compiler considers entire program as a single unit. It scans the entire program and then generates the target code.	1. Interpreter scans each line of source program and converts it to target code.
2. Compiler generates intermediate code.	2. Generates target code with or without generating any intermediate code.
3. It has to generate the intermediate code hence there is more scope for code optimization.	3. It may not generate Intermediate code hence there is a less scope for code optimization
4. If speed of execution is primary concern then compilers are preferred.	4. If time taken to generated target code is primary concern then interpreters are referred.
5. They cannot be used.	5. Interpreters are sometimes used during the development of a program.
6. Examples Pascal, C, C++ etc..	

- ## Hybrid Compiler :

This language processor combines the compilation and interpretation. i.e First source program may compiled into an intermediate form called BYTECODES and these bytes codes are interpreted by virtual machine. Example: JAVA

The advantage is that the bytecodes are compiled on one machine can be interpreted by another machine.

Just in Compiler :

In order to achieve faster processing of inputs to outputs, some Java compilers translate the BYTECODE into machine language immediately before they run the intermediate program to process the input. These are called Just-In-Compiler

Historical Perspective

- First Fortran Compiler – 1957 by John Backus and his team of 11 members

- Took 20 man years to write.

Languages

Classification Of Languages

- Natural Languages used for human Communication

  ---English,Kannada,Hindi
- Formal languages used for communications with a machine

  --- Fortran ,C,C++ etc.

Language processing system

The systematic steps involved in converting instructions in high level language to machine level code is called language processing. There are different components involved in language processing. They are

- Pre-processor
- Compiler
- Assembler

Steps in Language processing

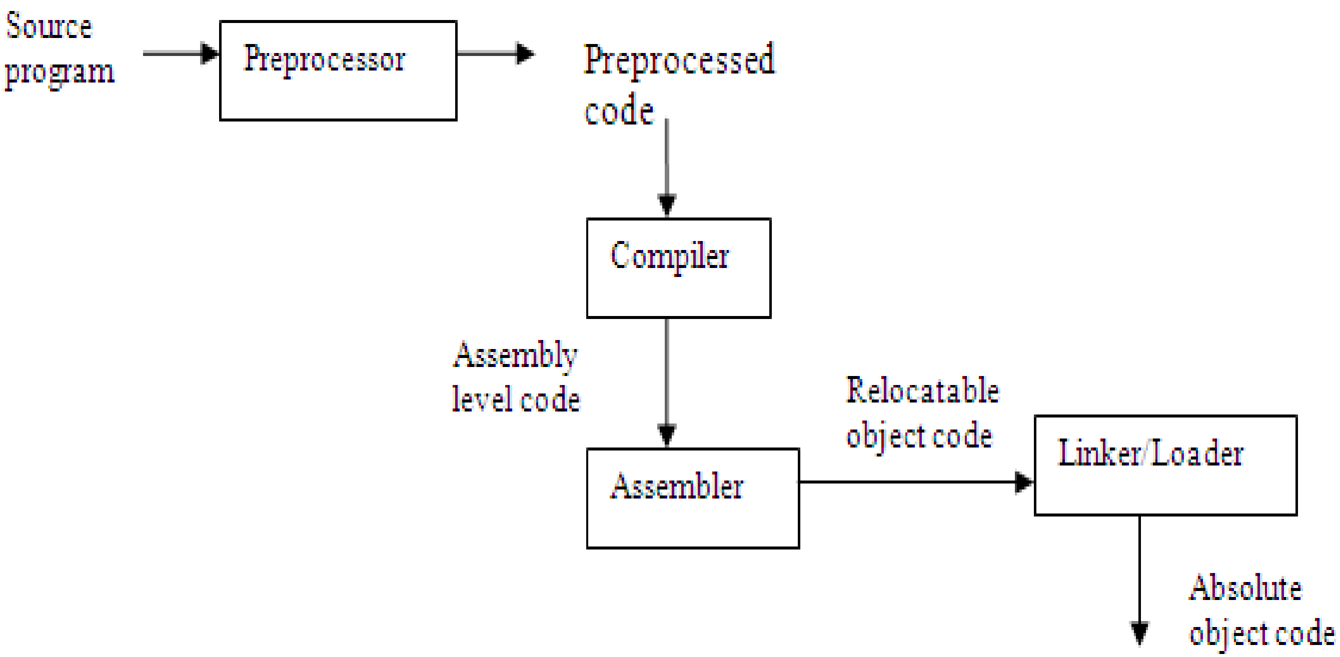


Fig 1.1 Steps in language processing.

Preprocessor

First step in language processing is preprocessing:

- Input to this phase is source program. Different parts of the source program may be stored in different files.
- Preprocessor collects all these files and creates a single file. It also performs macro expansion.
- In C #define and #include are expanded during preprocessing. Some preprocessors also delete comments from the source program.

Compiler

- Compiler takes preprocessed file and generates assembly level code.
- It also generates symbol table and literal table. Compiler has error handler which displays error messages and performs ***some error recovery if necessary.***
- In order to reduce the amount of ***time and memory taken for the*** execution and for better utilization of memory, compiler generates intermediate form of code and optimizes this code.
- Intermediate code is independent of machine architecture.
- Easy to perform ***language independent optimizations***.

Assembler

- Assembler takes assembly code as input and converts it into ***relocatable object code.***

- The instruction in assembly code will have three parts label,opcode part and an operand part.
  ex : mov r1,r2

- Opcode specifies the type of operation.

- The operand part consists of number of operands on which the operations are to be applied. These operands may be memory location, register or immediate data.

The structure of Compiler

- Basic Functions
- Phases of compiler
- Model and working of compiler

The Translation Process

- The process of converting source program to target code requires many functions to be done.
- Compilers can also be studied in two parts :

  Analysis part : breaks up the source program into consistent pieces and creates an intermediate representation of the source program.

  Synthesis part : This part constructs the designed target program from the intermediate representation of the two parts synthesis requires the most specilised techniques.
- ***Think of translating a Natural Language.***

Phases of the compiler

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation
- Code optimization
- Final code generation.

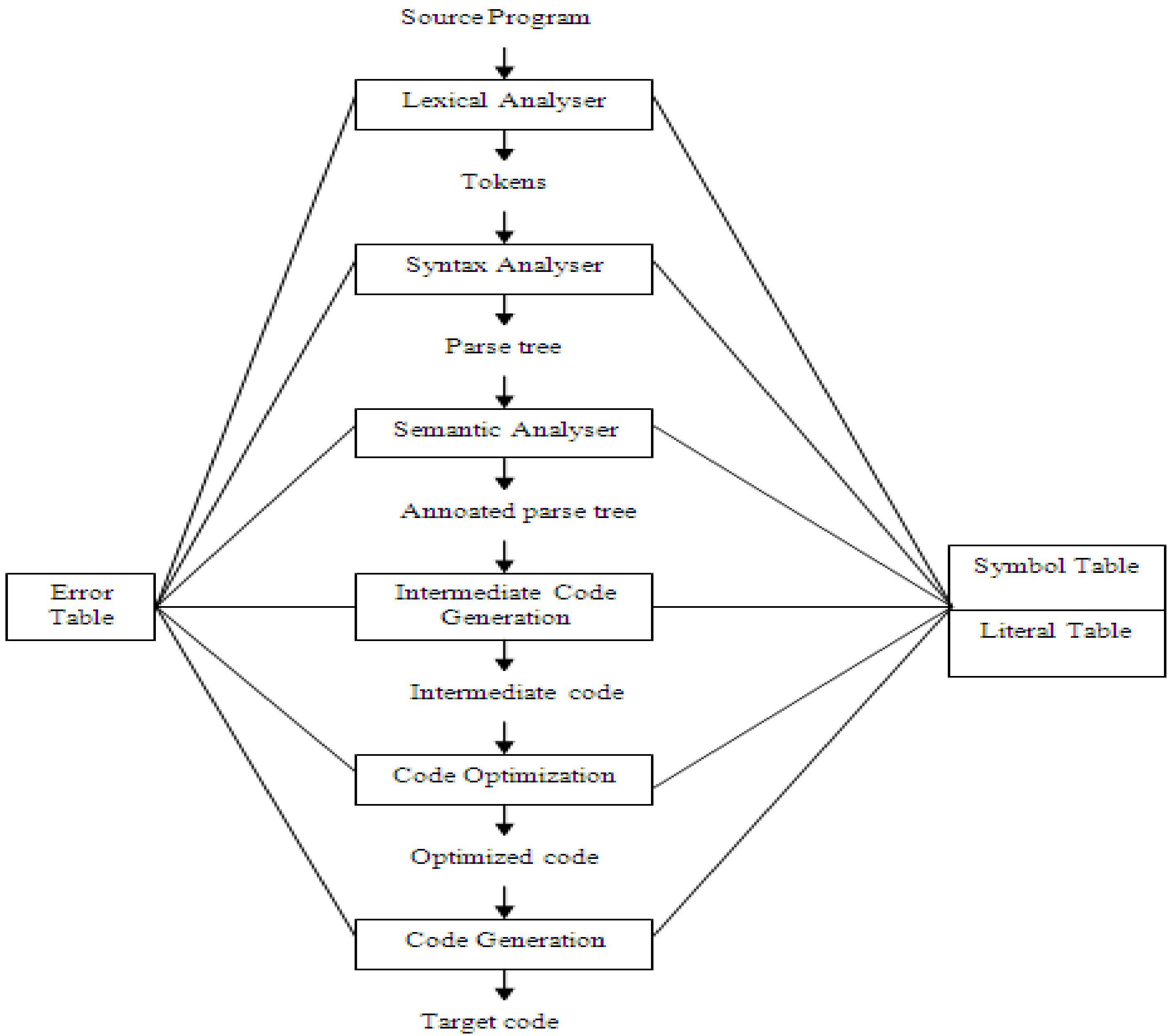


Fig 1.2 Phases of Compiler

Lexical Analysis

- The main function of Lexical Analyser is to break the source program into tokens.
- Tokens are classified as key words, identifiers, operators, etc.
- Lexical analyser then creates symbol table and literal table to store the identifiers and constants respectively.
- This phase takes care of detecting few lexical errors and applies some strategy and tries to recover from errors.

Syntax Analyser

- Syntax Analyser determines the structure of the program.
- The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
- Syntax Analyser uses context free grammar to define and validate rules for language construct.
- Output of Syntax Analyser is parse tree or syntax tree which is hierarchical / tree structure of the input.

Semantic Analyser

Input to semantic analysis phase is parse tree or syntax tree.

Important function of semantic analysis phase is:

1. Type checking.
2. Type conversion.
3. Checks for some semantic errors like real identifier used for array indexing.

Output of semantic analyser is an annotated parse tree.

Intermediate Code Generator

- Intermediate code generator generates intermediate code for annotated parse tree. This code is dependent on source program and ***independent of machine architecture.***

- The intermediate code uses three address codes and uses temporary variables to store the intermediate results.

Code optimization

- The purpose of code optimiser is to reduce the number of operations or reduce the amount of time taken for execution.

- It also takes care that it uses minimum temporaries to store intermediate values. Based on the amount of time taken to execute the instruction, most appropriate instruction will be selected.

- ***Code optimizer concentrates on that part of the code which will be executed many times*** and tries to optimise that code.

Final Code Generation

Input to code generation phase is the intermediate optimised code generated from code optimization phase or intermediate code with no optimization. Final phase is to generate a code that runs on target machine

Most difficult phase. ***It an NP problem***

Example

Syntax Definition:
For example consider as arithamatic statement:

Ex : 9-5+2
Ex : Cost = prize * qty * 1.2

list → list + digit
list → list – digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9|

Ex 1: Parse tree for 9-5+2

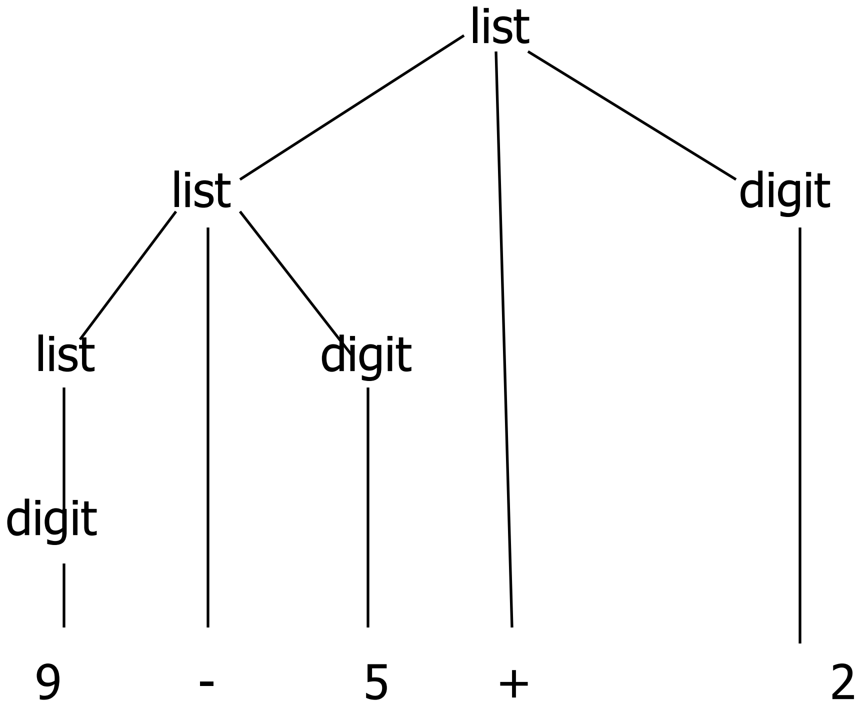
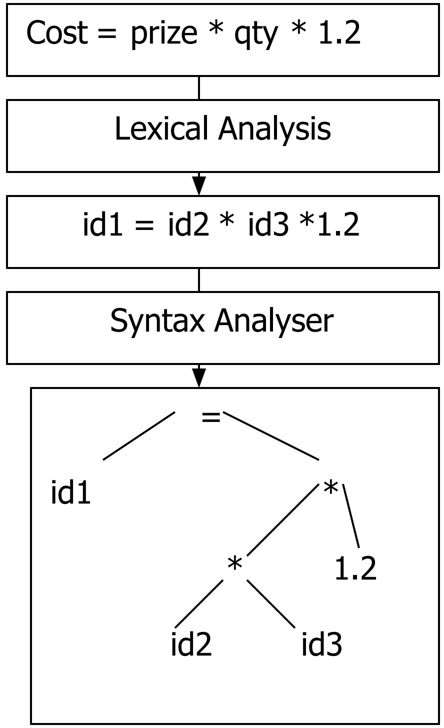
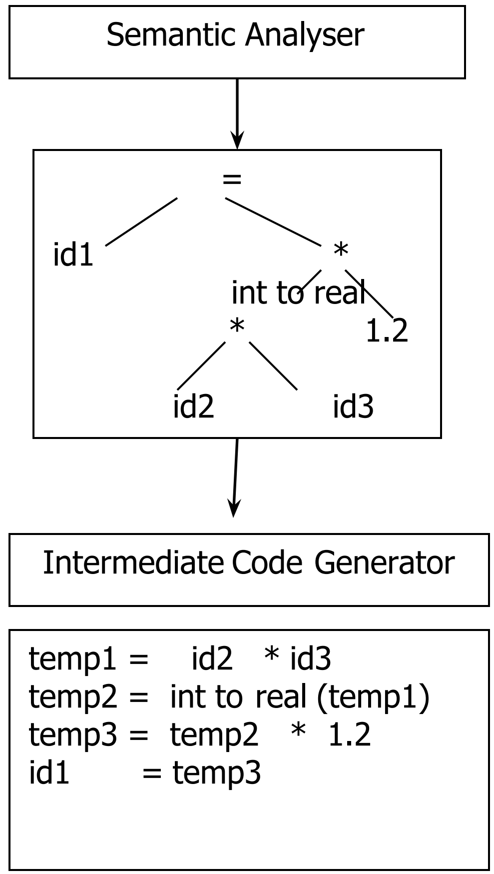


Illustration of phases of compiler through an example Cost = prize * qty * 1.2



Ex contd


Semantic Analyser
=
id1
*
int to real
*
1.2
id2
id3
Intermediate Code Generator
temp1 = id2 * id3
temp2 = int to real (temp1)
temp3 = temp2 * 1.2
id1 = temp3

Ex contd

Code Optimizer

```
temp1 = id2 * id3

temp2 = int to real(temp1)

td1 = temp2 * 1.2
```

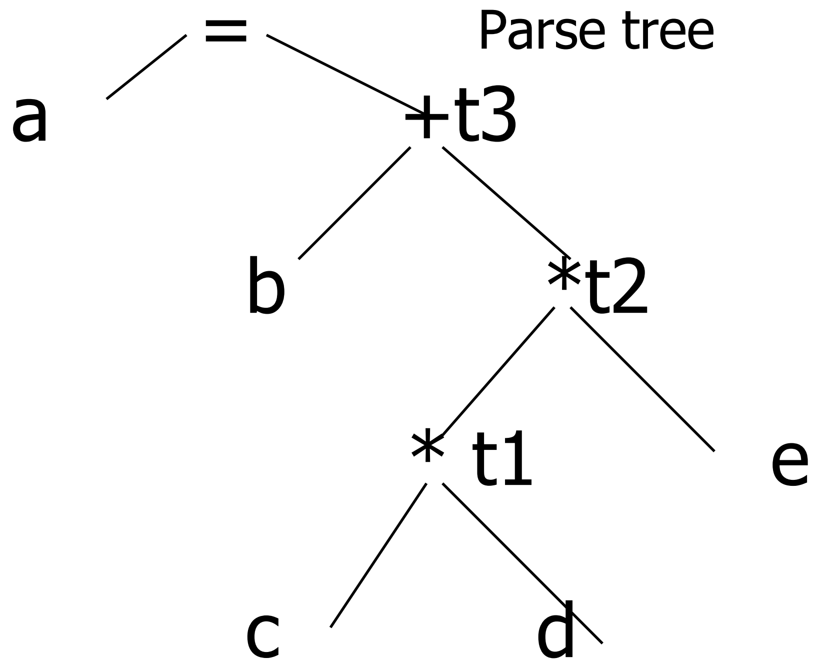
Code Generator

```
movf  id2, r2
movf  id3, r1
mulf  r2, r1
mulf  #1.2 ,r1
Movf  r1, id1
```

Ex 3 : a = b+c*d*e

Intermediate code

t1= c*d
t2 = t1*e
t3 = b+t2
a = t3

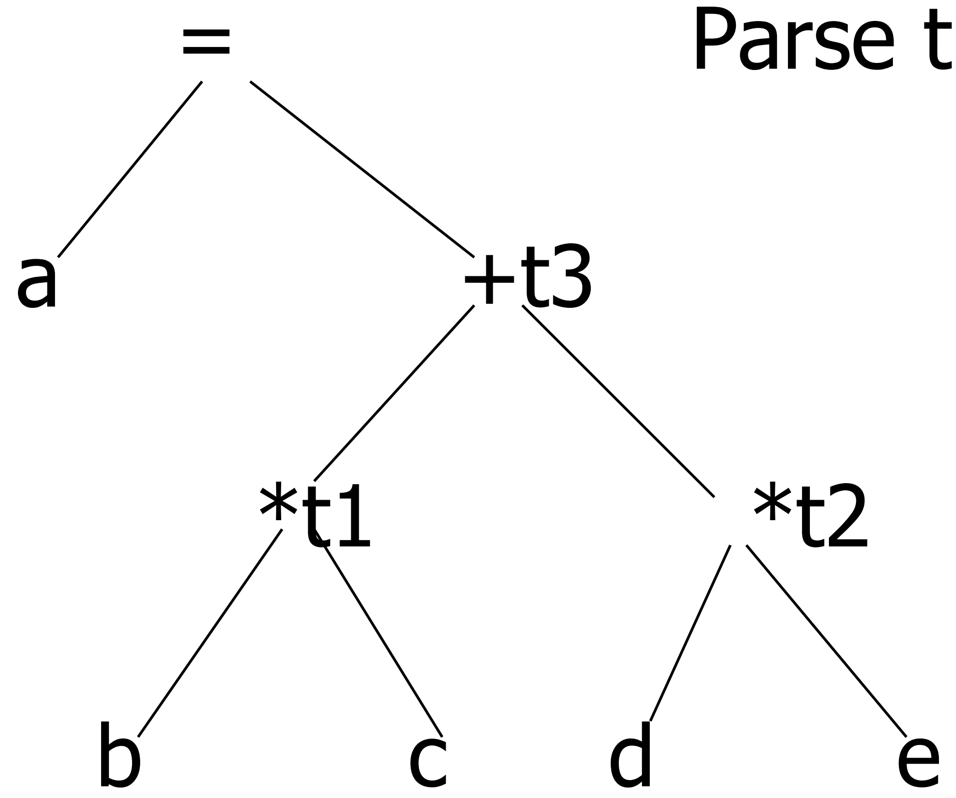


Ex 4: a = b*c + d*e

Intermediate code

t1 = b*c
t2 = d*e
t3 = t1+t2
a = t3

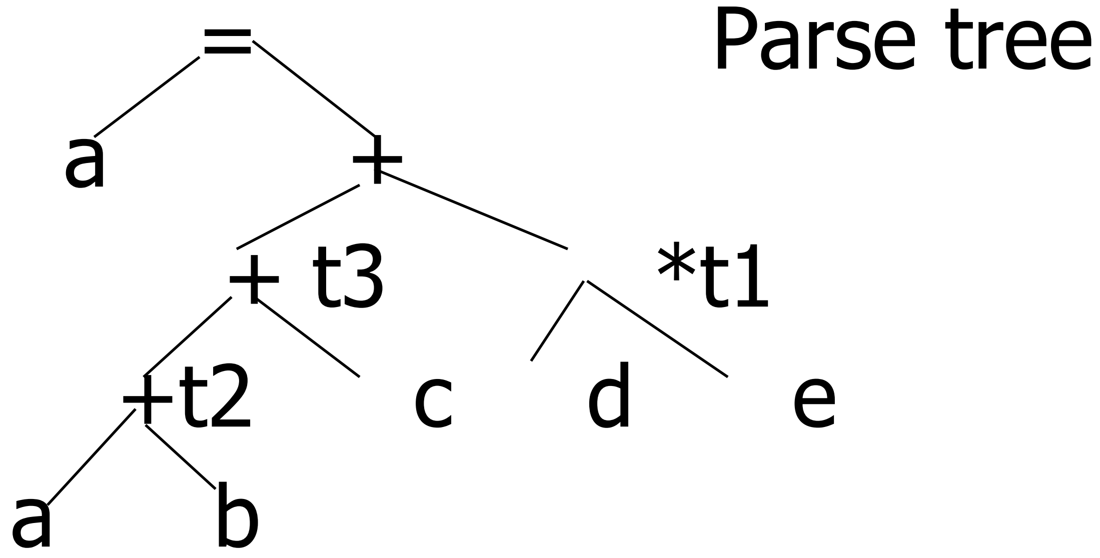
Parse tree



Ex 5: a = a+b+c+d*e

Intermediate code

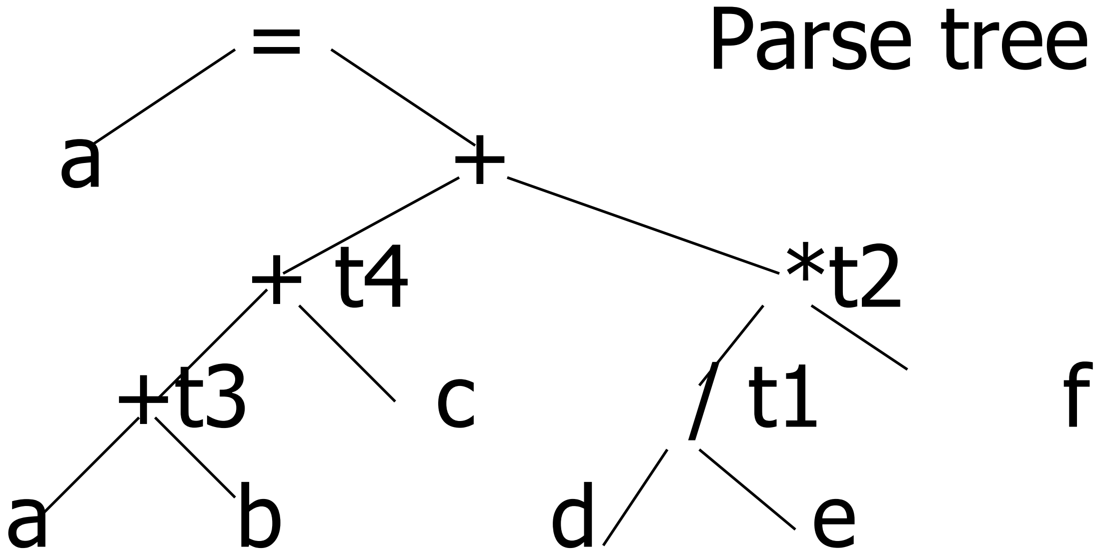
t1 = d*e
t2 = a + b
t3 = t2 +c
a = t3 + t1



Ex 6: a = a+b+c+d/e*f

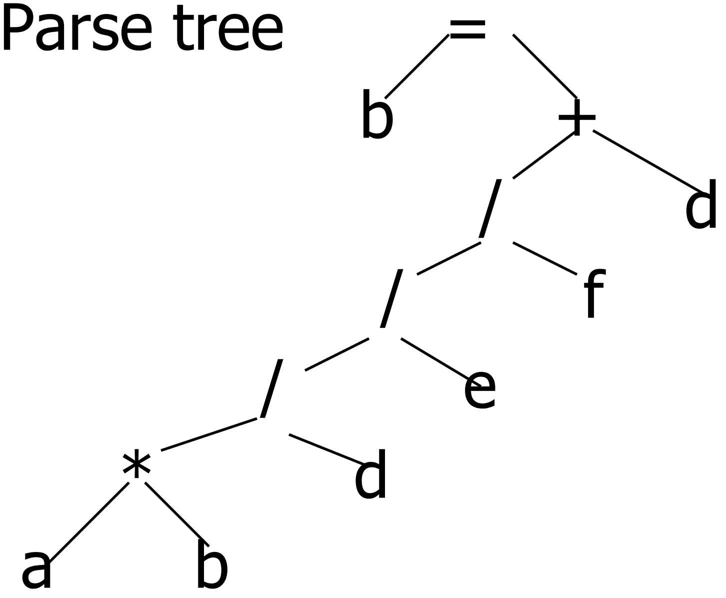
Intermediate code

t1 =d / e
t2 = t1*f
t3 = a+b
t4 = t2 + c
t5 = t4+t2



Ex 7: b = a*b/d/e/f+d

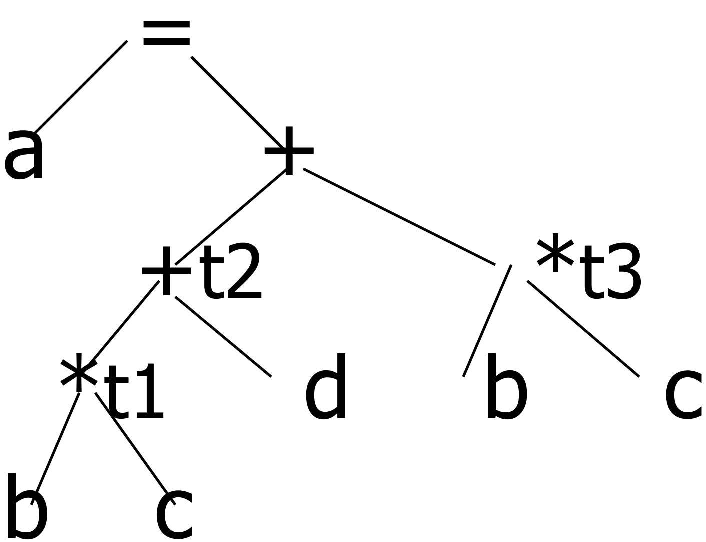
Parse tree



Intermediate code

t1 = a*b
t2 =t1/d
t3 = t2/e
t4 = f+d
b = t3/t4

An example for Code Optimization for exp a = b*c + d + b*c



Intermediate code

t1 = b*c

t2 = t1+d

t3 =b*c

t4 = t2 +t3

a =t4

Optimized Code

t1= b*c

t2 = t1 + d

t3 = t2 +t1

Compiler Tools

Tools used to construct the compiler are :

- Lex
- Yacc
- Automatic code generator
- Data flow engines

Evolution of Programming languages

Classification by Generation

* First Generation-Machine Language
* Second Generation- Assembly Language
* Third Generation - High Level Languages like C,C++.
* Fourth Generation –Specific Applications like SQL.
* Fifth Generation- For logic and constraint base

  languages like Prolog.

Contd..

Other Classifications:

- Imperative – Specifies ***how*** a computation is to be done.
- Declarative – Specifies ***what*** computation to be done.

Applications of Compiler Technology

- Implementation of High Level programming languages
- Optimizations for computer architectures
- Design of new computer architectures
- Program Translations
- Software productivity Tools.

The Role of Lexical Analyser

- The main function of Lexical Analyser is to break the source program into tokens.
- Tokens are classified as key words, identifiers, operators.
- Generates token with the help of regular expression or finite automata.
- Eliminates blank spaces, tab and new line characters.
- Detecting lexical errors if any, correlating error message with position of error, like line number, function where the error is detected or file where the error is found.

Lexical Analyser-Design

Lexical analyser is designed as two parts.

- Scanning Process

  The source program is given as input and it is broken into tokens.

- Analysis

  The tokens generated from the scanning process is analysed and grouped into specific category.

Buffering

- The source program is to be stored in buffers before scanning.
- The buffer is scanned to retrieve the token. For this purpose two pointers are used, viz, **Begin pointer and forward pointer**.
- Initially begin pointer and forward pointer both points to the beginning of buffer (starting of program).
- Forward pointer moves one symbol by another until a lexeme is found.
- Current lexeme is a string between begin pointer and forward pointer.
- This lexeme is converted to token and sent to parser.
- The token will be entered onto symbol table or literal table based on the type of token.

Specification of tokens

- Alphabet is a finite set of all characters, digits, operators and punctuation marks that can be used in the source language.
- Example in C language ∑= {0,1,…9, a, b,c, ….z, A,B, ….Z,+,- , *, / , (,), &, <, > , =,…..}
- String is a finite sequence of symbols drawn from the alphabet.
- Example: w = abc or s = abc234

Recognition of Tokens

- Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token.

- Using the token, code has been generated that examines the input string and finds a prefix that is lexeme matching one of the patterns.

Recognition of Tokens and Implementation

- Grammar fragment for Branching statement

$$
\begin{aligned}
Stmt &\rightarrow \textbf{if } expr \textbf{ then } stmt \\
&\quad \mid \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
expr &\rightarrow term \textbf{ relop } term \\
&\quad \mid term \\
term &\rightarrow \textbf{id} \\
&\quad \mid \textbf{number}
\end{aligned}
$$

Description of Patterns for the above tokens using Regular definitions :

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 → < | > | <= | >= | = | <>

Example -1

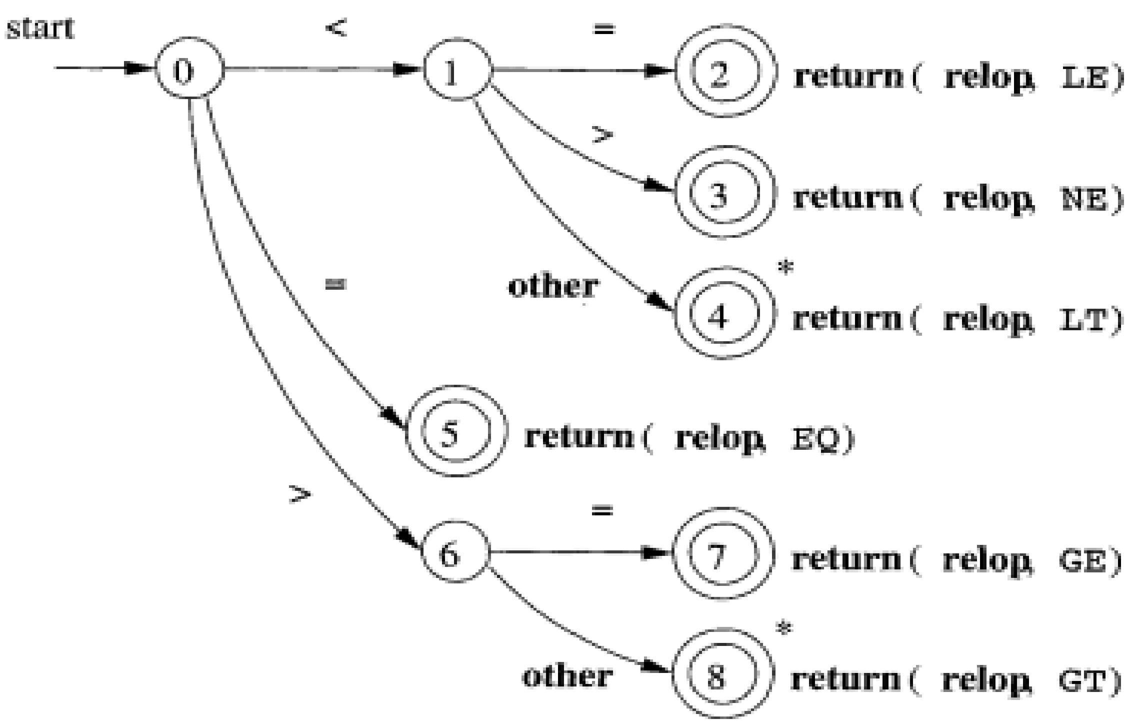


Figure 3.13: Transition diagram for **relop**

Example -1

Method for Translating the transition Diagram into a program using the following steps:

1. Every **STATE** gets piece of code that reads a character from the input buffer.
2. The character read is compared with the labels of the outgoing edges for the current **STATE**. If any match then control is transferred to the respective **Next-STATE** by calling the code associated with it, otherwise there is a failure to recognize the token and next transition diagram may be activated for another token.
3. If state is a **FINAL STATE** then no character is read and the **token definition** along with **NAME** and **ATTRIBUTE** is returned.
4. If a **FINAL STATE** is marked with '*' then one or more previously read characters, are not part of the lexeme hence retract is performed accordingly to move the **FORWARD** pointer one or more positions back.

- getRelop() ☐ It is C++ function whose job is to simulate the transition diagram for **relational operators** and returns an object of type **TOKEN,** that is pair consisting of the **token name** and an **attribute value.**

  It also first creates a new object **retToken** and initializes its first component to **RELOP**, the symbolic code for **token relop .**
- **nextChar()** ☐ It obtains the next character from the input buffer and advances the forward pointer to next character.
- **retract()** ☐ It moves the forward pointer one or more position back to reconsider the processed input character which are not part of lexeme
- **Fail()** ☐ It resets the forward pointer to lexmebegin and activates the next transition diagram to be applied to the true beginning of the unprocessed input. It might also change the **STATE** value to be the **START STATE** for next transition diagram. If there are no transition diagram to be applied, it can initiate the error recovery routine to repair the input to find a lexeme

Implementation Relational Operator

```
// it is assumed that the current state is stored in variable state
TOKEN getRelop ()
{
    TOKEN retToken = new(RELOP) // initialization of Token's first component to be 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();
                   break;
        Case 1:    C = nextchar( );
                     if( C==‘=’ ) state = 2;
                   else if (C==‘>’) state = 3;
                   else  state =4 ;
                   break;
        Case 2:    retToken.attrtibute = LE;
                   return(retToken)
                   break;
```

Implementation Contd..

```
Case 3:    retToken.attrtibute = NE;
           return(retToken)
           break;
Case 4:    retract();
          retToken.attribute = LT;
          return(retToken)
Case 5:    retToken.attribute = EQ;
          return(retToken);
Case 6: ---------------
 ---------
Case 8:     retract();
          retToken.attribute = GT;
          return(retToken)
     } // end of switch
  } // end of while
}// end of function
```

Recognition of Identifiers and keywords

- Usually all the keywords like **if**, **then** **else** etc… are not identifiers even though they look like identifiers and the transition diagrams used for identifying the identifiers will also recognize all the keywords . Hence recognition of keywords and identifiers creates a problem. This can be handled in two different ways

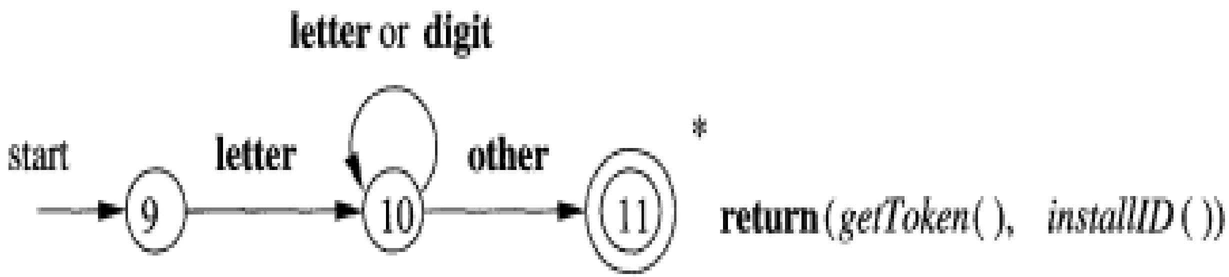


Figure 3.14: A transition diagram for **id**'s and keywords

1. Installation of all the keywords in the symbol table.

Here the field of symbol table entry indicates that these strings are keywords and never ordinary identifiers and also tells which token they represent. When the lexme is recognized we can use proper mechanism to differentiate the keywords and identifier tokens.

- To do that we execute the following two functions
- InstallId(): When the identifier lexme is found this function is invoked that checks the symbol table for the lexme for the following conditions.
  - If found and marked as keyword then it returns **ZERO.**
  - If found and is a program variable then it returns a **pointer to the symbol table entry.**
  - If not found then it is installed as a variable and returns **a pointer to the newly created entry**

- getToken() : When this function is invoked it examines the symbol table and returns the token type that is either ID or keword itself.

2. Separate transition diagram for each keywords:

Here every keywords gets a transition diagram that increases the number of states and is tedious to write a lexical analyzer. Also we prioritize the tokens so that the keywords are recognized in precedence to Identifier.

Implementation of IDENTIFIER

```
TOKEN getID ()
{
   TOKEN retToken
   While(1)                        // repeat character processing until a return or failure
    occurs
   {
        Switch (State){
        Case 9:     C = nextchar( );
                     if isletter (C) state = 10;
                    else fail();
                    break;
        Case 10:   C = nextchar( );
                     if  isletter ( C ) || isdigit ( C ) state = 10;
                    else  state =11 ;
                    break;
```

```
Case 11:   retract ();
           retToken = getToken();
           retToken.attrtibute = install_ID ();
           return(retToken)
           break;
    } // End of Switch
  } // End of while
}
```

```
int start =0, state=0;
int fail()
{
     forward = lexme_beginning;
      switch( start )
      {
              case 0   :  start =9; break;
              case 9   :  start =12; break;
              case 12 :  start =22; break;
              case 22 :  recover();
              default  :   // compiler error
      }
}
```

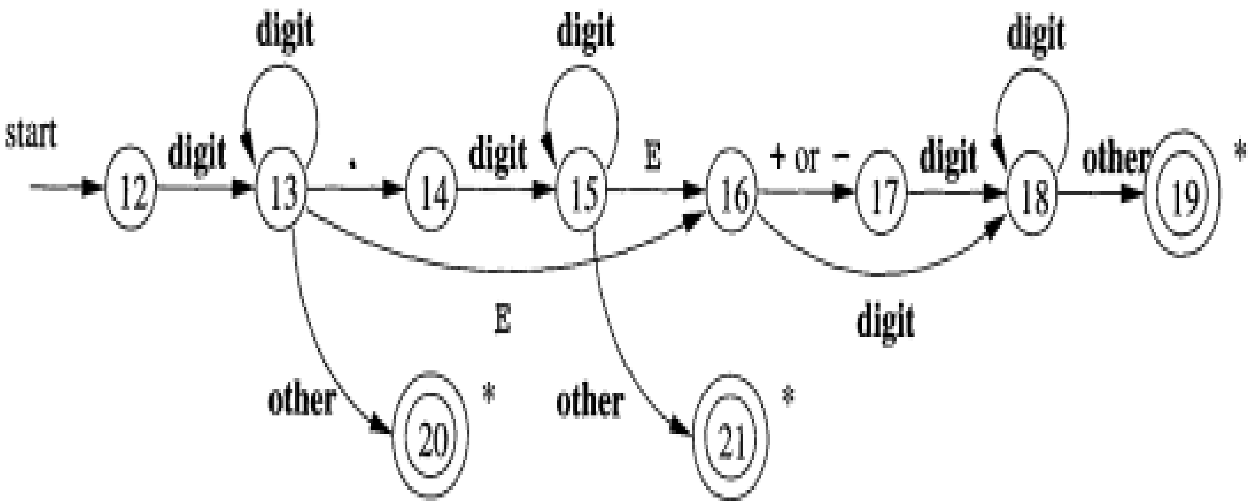
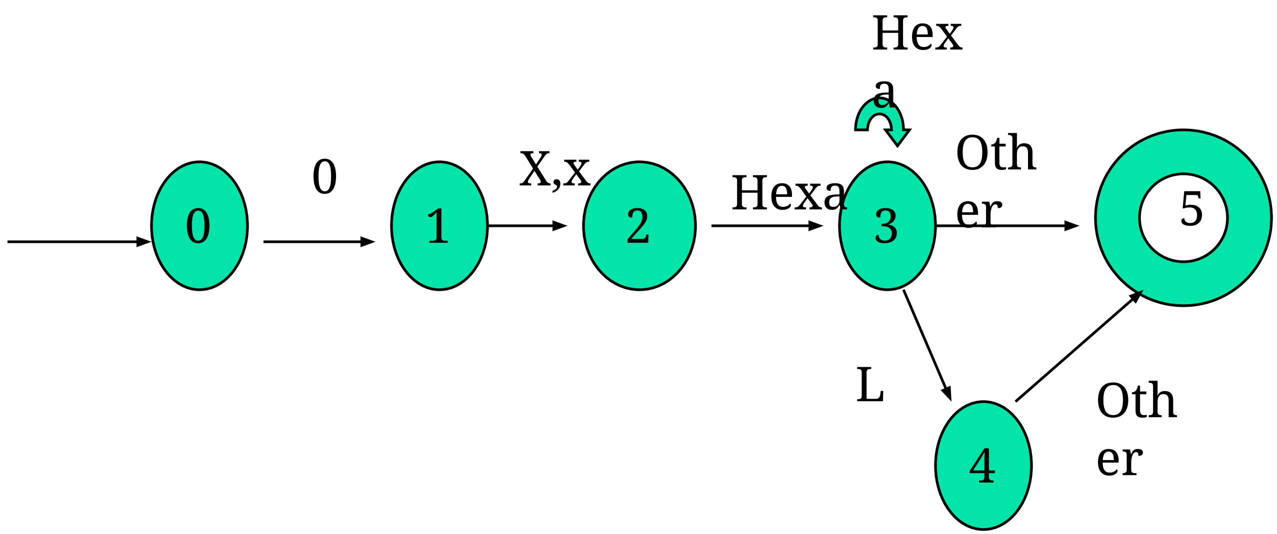


Figure 3.16: A transition diagram for unsigned numbers

Example

- DFA for Hexadecimal Number



Implementation

Representation of Hexadecimal number in C

// it is assumed that the current state is stored in variable state

```
TOKEN getHNUM{
TOKEN retTOKEN = new(HNUM)
While(1){/* repeat character processing until a return or failure occurs*/
Switch (State){
Case 0:     C = nextchar( );
             if (C ==‘0’) state = 1;
             else fail( );
            break;
Case 1:     C = nextchar( );
            if( C==‘X’ || C== ‘x’) state = 2;
          else fail( );
          break;
Case 2:     C = nextchar( );
            if (ishexa (C)) state = 3;
          else fail( );
          break;
```

Implementation Contd..

```
Case 3:      C = nextchar( );
            if (ishexa (C)) state = 3;
            else if (C==‘L’) state = 4;
            else if (isother(C)) state = 5;
            else fail( );
            break;
Case 4:  C = nextchar( );
          if (isother(C)) state = 5;
          else fail( );
           break;
Case 5:   retract( );
          retToken.attribute = HNUM;
          Return(retToken);
}
}
}
```

Topics covered

- Introduction
- Context Free Grammar
- Writing a grammar
- Role of Parser
- Top Down Parsing
- Conclusion

Syntax Analyser

- Syntax Analyser determines the structure of the program.
- The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
- Syntax Analyser uses context free grammar to define and validate rules for language construct.
- Output of Syntax Analyser is parse tree or syntax tree which is hierarchical / tree structure of the input.

Context Free Grammar

- Context Free Grammars (CFG) are used to represent the grammatical structure of all programming languages like C, Java etc.
- Understanding of CFG is important from the point of parsing an input sentence.
- Parser takes CFG as a database for checking the grammatical correctness of a statement in a programming language.

Definition of CFG

Denoted by G = (V, T, P, S)

Where,

- V is a set of variables, (non-terminals)
- T is a set of terminals,
- P is a set of productions,
- S is unique start symbol and S € V

Context Sensitive Grammar

- A context sensitive Grammar is an unrestricted Grammar in which every production has the form
  $\alpha \rightarrow ß$ where $|ß| \geq |\alpha|$ and α & ß are variables or terminals

For Example,

$S \rightarrow aAb$

$aA \rightarrow bAA$

$bA \rightarrow aa$

Non-Context Free Grammars

$L1 = \{wcw \mid w \text{ is in } (a|b)^*\}$

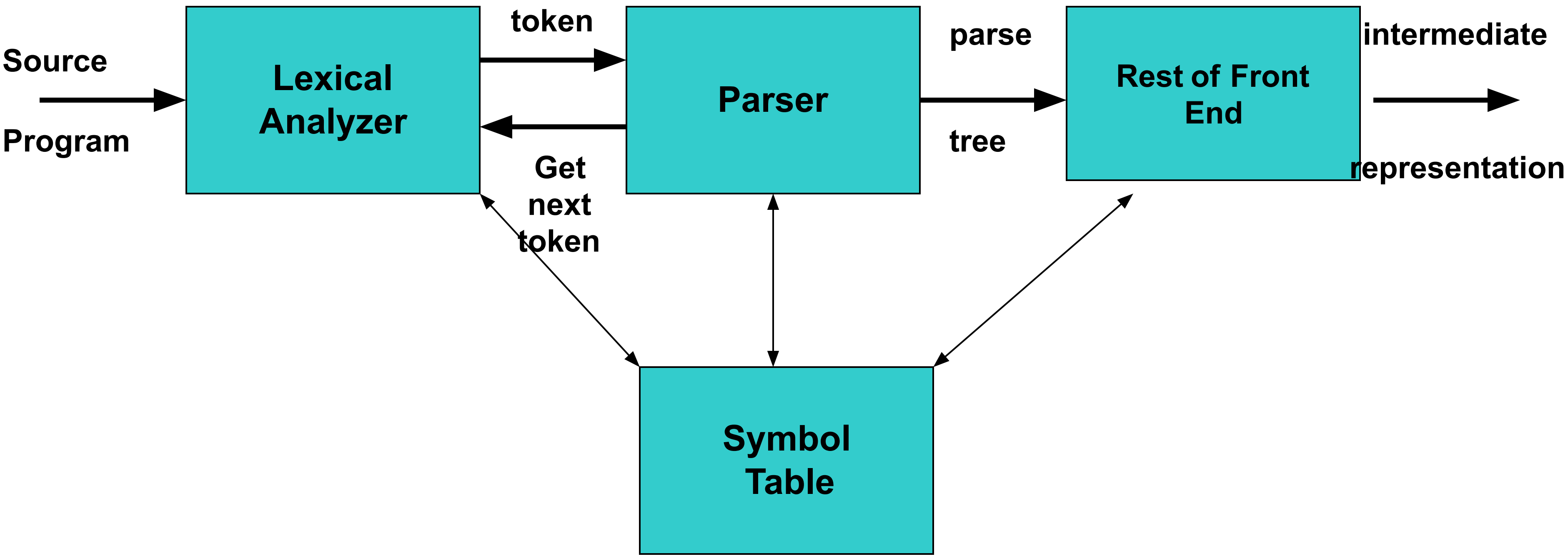
$L2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$

Unrestricted Grammar

- An unrestricted Grammar is a 4-tuple G=(V,∑,S,P) where V and ∑ are disjoint sets of variables and terminals; S is the Start Symbol and P is the set of productions of the form

  $\alpha \rightarrow ß$ where α, ß Є (V U ∑)* and α contains atleast one variable
- In this type there is no restriction on the length of α and ß
- The only restriction is that α cannot be ε.i.e, ε cannot appear on the left hand side of any production

Position of Parser in Compiler model



Role of a parser.

The stream of tokens is input to the syntax analyzer. The job of the parser is:

- To identify the valid statement represented by the stream of tokens as per the syntax of the language. If it is a valid statement, it will be represented by a parse tree.

- If it is not a valid statement, then a suitable error message is displayed, so that the programmer is able to correct the syntax error.

Classification of Parser

Parsing

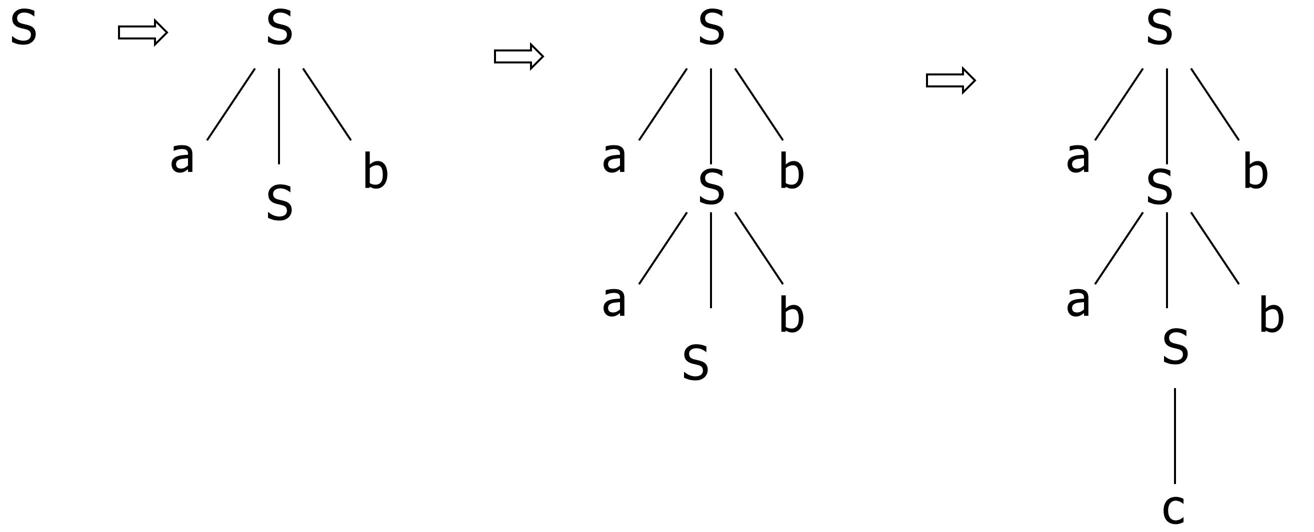
- **Top down parsing**

  In top down parsing we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence.

- **Bottom-up parsing**

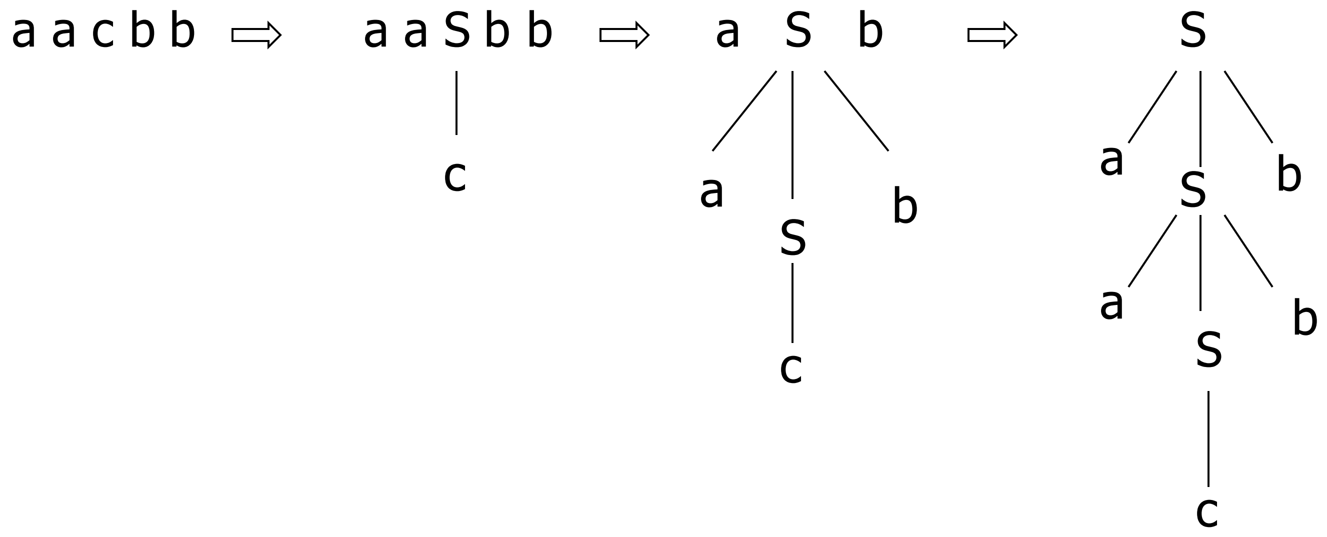
  In bottom-up parsing we start from the given sentence and using various production, we try to reach the start symbol.

Consider the grammar S -> aSb | c for the string aacbb



Top down parsing

Consider the grammar S -> aSb | c for the string aacbb



Bottom-up parsing

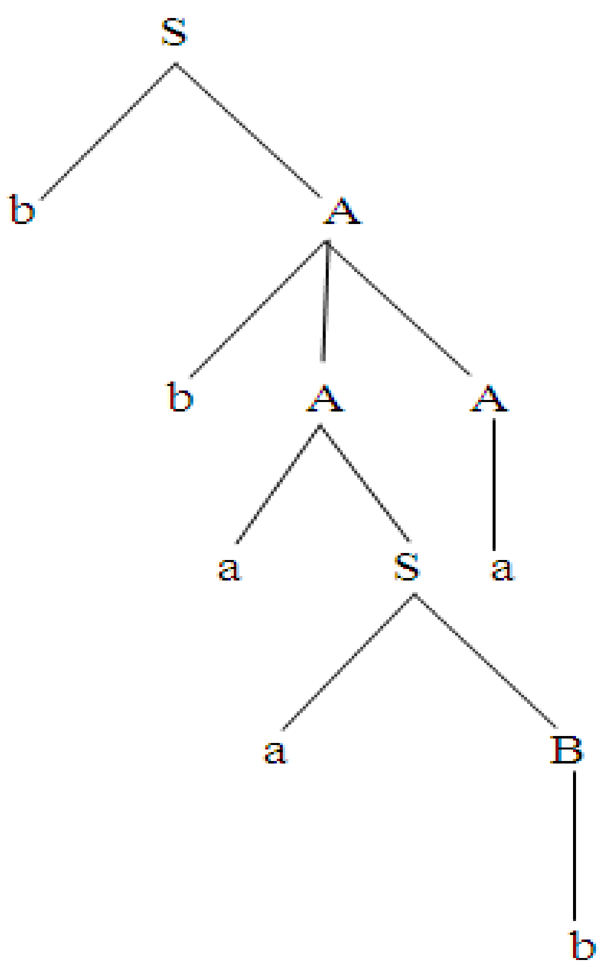
Example

Consider the grammar

- $S \rightarrow b\,A \mid a\,B$
- $A \rightarrow b\,A\,A \mid a\,b \mid a$
- $B \rightarrow a\,B\,B \mid b\,S \mid b$

i) bbaaba

ii) bbbaaaba (Home Work)



Leftmost Derivation

$$S \Rightarrow b\,A$$
$$\Rightarrow b\,b\,\underline{A}\,A$$
$$\Rightarrow b\,b\,a\,\underline{S}\,A$$
$$\Rightarrow b\,b\,a\,a\,\underline{B}\,A$$
$$\Rightarrow b\,b\,a\,a\,b\,A$$
$$\Rightarrow b\,b\,a\,a\,b\,a$$

Rightmost derivation

$$S \Rightarrow b\,\underline{A}$$
$$\Rightarrow b\,b\,A\,\underline{A}$$
$$\Rightarrow b\,b\,\underline{A}\,a$$
$$\Rightarrow b\,b\,a\,\underline{S}\,a$$
$$\Rightarrow b\,b\,a\,a\,\underline{B}\,a$$
$$\Rightarrow b\,b\,a\,a\,b\,a$$

Fig. 3.5 Parse Tree for the string bbaaba

Ambiguous grammar

If for any given sentence there are two or more parse trees or syntax trees, then the grammar is ambiguous.

Why Ambiguous grammars can not be used?

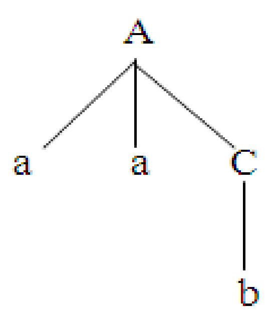
Example: Consider the following ambiguous grammar

$A \rightarrow BC \mid aaC$

$B \rightarrow a \mid Ba$

$C \rightarrow b$

Tree 1

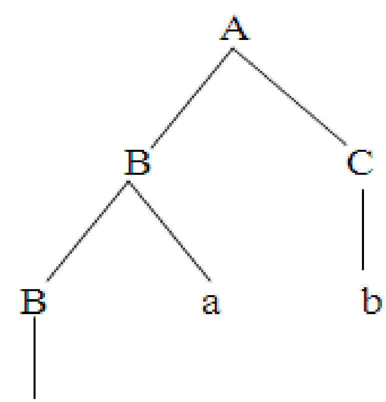


Leftmost derivation

$A \Rightarrow a\ a\ \underline{C}$

$\Rightarrow a\ a\ b$

Tree 2



Leftmost derivation

$A \Rightarrow \underline{B}\,C$

$\Rightarrow \underline{B}\ a\ c$

$\Rightarrow a\ a\ c \Rightarrow a\ a\ b$

Fig. 3.20 Two leftmost derivation for string a a b

Top-down Parsers

Top-down parsers are suitable for constructing parser by hand

Top-down parsers are again classified as

- Recursive Descent Parser
- Predictive Parser

Prerequisites for topdown parsers

- Elimination of Left-recursion
- Left Factoring
- First Set
- Follow Set

Elimination of Left-recursion

Example:

$S \rightarrow a\,S \,|\, b\,S \,|\, c\,S \,|\, S\,d \,|\, S\,e \,|\, S\,f \,|\, g \,|\, h$

Then

$S \rightarrow S\,d \,|\, S\,e \,|\, S\,f$ contains left recursion

$\alpha_1 = d,\ \ \alpha_2 = e,\ \ \alpha_3 = f$

The remaining terms do not contain left recursion they are β terms. Namely

$\beta_1 = aS\ ,\ \beta_2 = b\,|\,S,\ \beta_3 = c\,|\,S,\ \beta_4 = g,\ \beta_5 = h$

Therefore grammar containing without left recursion is as follow. Where S^1 is a new non-terminal introduced.

$S \rightarrow a\,S\,S^1 \,|\, b\,S\,S^1 \,|\, c\,S\,S^1 \,|\, g\,S^1 \,|\, h\,S^1$

$S^1 \rightarrow d\,S^1 \,|\, e\,S^1 \,|\, f\,S^1 \,|\, \epsilon$

Left factoring:

Example : Consider the grammar

$S \rightarrow b\,S \mid b\,S\,B \mid b\,S\,D \mid e$

Here left factor is ‘bS’, introduce a new non-terminal say ‘E’

$S \rightarrow b\,S\,E \mid e$

Now E should produce all the terms remaining is the original production.

$E \rightarrow \epsilon \mid B \mid D$

In 1st production only bs exists therefore λ is added. In the 2nd production after removing bs only ‘B’ remains and similarly D in 3rd production.

First Set

$S \longrightarrow aSb \mid cDe \mid fGh \mid i$

$D \longrightarrow d \mid \varepsilon$

$G \longrightarrow g \mid \varepsilon$

First(S) = {a,c,f,i}

First(D) = {d,ε}

First(G) = {g,ε}

Follow Set

$S \longrightarrow aSb \mid cDe \mid fGh \mid i$

$D \longrightarrow d \mid \varepsilon$

$G \longrightarrow g \mid \varepsilon$

Follow(S) = {b,$}

Follow(D) = {e}

Follow(G) = {h}

LL(1) Parser

LL (1) Parser stands for Left to right scan and Leftmost derivation, which it tries to derive. This is also a topdown parser, it implements recursive descent parser efficiently without using recursion.

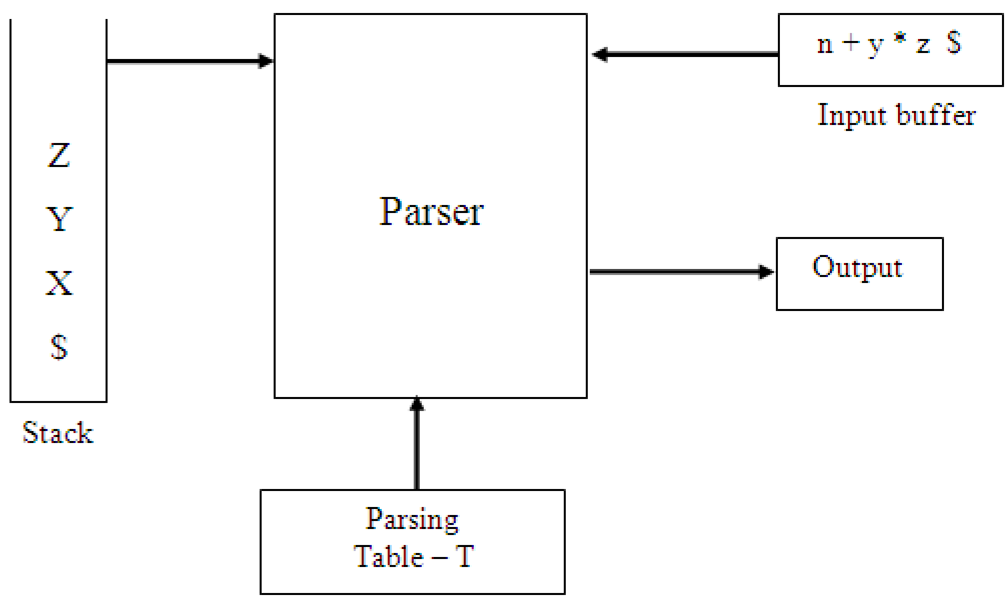


Fig. 4.1 LL(1) Parser

Assume a Grammar for L = { a^n c b^n | n ≥ 0}

$S \rightarrow a\ S\ b \mid c$

One parsing table T will be as follows

Non Terminal	a	b	c
S	$S \longrightarrow aSb$		$S \longrightarrow c$

Stack	Input Buffer	Action
S$	aacbb$	
~~a~~Sb$	~~a~~acbb$	S->aSb
Sb$	acbb$	Match a
~~a~~Sbb$	~~a~~cbb$	S->aSb
Sbb$	cbb$	Match a
~~c~~bb$	~~c~~bb$	S->c
~~b~~b$	~~b~~b$	Match c
~~b~~$	~~b~~$	Match b
$	$	Match b
		Accept

Questions ...?

Thank You