M.Sc. (C. S.)

SEMESTER - II (CBCS)

PAPER II

DESIGN AND IMPLEMENTATION OF MODERN COMPILERS

SUBJECT CODE: PSCS202

Dr. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology IDOL,
Univeristy of Mumbai – 400098

Course Co-ordinator : **Mr Sumedh Shejole**
Assistant Professor,
IDOL, University of Mumbai- 400098

Course Writers : **Shallu Hassija**
Pt. JLN Government College, Faridabad,
Sec-16, Faridabad, Haryana

: **Priyamwada Godbole**
Kirti College
Mumbai

: **Seema Nandal**
Pt. JLN Government College, Faridabad,
Sec-16, Faridabad, Haryana

May 2022, Print - 1

Published by : Director,
Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari,Mumbai - 400 098.

DTP composed and Printed by: Mumbai University Press

CONTENTS

M.Sc. (C. S.)

Semester - II PAPER II

DESIGN AND IMPLEMENTATION OF MODERN COMPILERS

SYLLABUS

Course Code	Course Title	Credits
PSCS202	**Design and implementation of Modern Compilers**	**04**

Unit I: Introduction to Compilers

The structure of a compiler, A simple approach to the design of lexical analyzers, Regular expressions, Finite automata, From regular expressions to finite automata, Minimizing the number of states of a DFA, Context-free grammars, Derivations and Parse trees, Parsers, Shift-reduce parsing, Operator-precedence parsing, Top- down parsing, Predictive parsers.

Unit II: Automatic Construction of Efficient Parsers

LR parsers, The canonical collection of LR(0) items, Constructing SLR parsing tables, Constructing canonical LR parsing tables, Constructing LALR parsing tables, Using ambiguous grammars, An automatic parser generator, Implementation of LR parsing tables, Constructing LALR sets of items.

Unit III: Advanced syntax analysis and basic semantic analysis

Syntax-directed translation schemes, Implementation of syntax-directed translators, Initial introduction to the ongoing Tiger compiler, bindings for the Tiger compiler, type-checking expressions, type-checking declarations, activation records, stack frames, frames in the Tiger compiler, translation to intermediate code, intermediate representation trees, translation into trees, declarations, basic blocks and traces, taming conditional branches, liveness analysis, solution of dataflow equations, liveness in the Tiger compiler, interference graph construction.

Unit IV: Dataflow analysis and loop optimization

The principle sources of optimization, Loop optimization: The DAG representation of basic blocks, Dominators, Reducible flow graphs, Depth-first search, Loop-invariant computations, Induction variable elimination, Some other loop optimizations. Dataflow Analysis: intermediate representation for flow analysis, various dataflow analyses, transformations using dataflow analysis, speeding up dataflow analysis, alias analysis.

Text book:

- Compilers: Principles, Techniques and Tools 2nd edition, Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman , Pearson (2011)
- Modern Compiler Implementation in Java, Second Edition, Andrew Appel and Jens Palsberg, Cambridge University Press (2004).

References:

- Principles of Compiler Design, Alfred Aho and Jeffrey D. Ullman, Addison Wesley (1997).
- Compiler design in C, Allen Holub, Prentice Hall (1990).

1

INTRODUCTION TO COMPILERS

Unit Structure

1.0 OBJECTIVES

After going through this unit, you will be able to:

- define the compiler, regular expression
- structure of compiler
- define the contact free grammars, parsers
- minimize the number of states of DFA
- top-down parsing

1.1 INTRODUCTION

Compiler is a software that translate one language into another language. The compiler converts the code from high-level language (source code) to low level language (machine code/object code) as shown in figure 1. From the compiler expected that it will give optimized result in terms of time and space.

Figure. 1

- Computer are the combination of hardware and software.
- Hardware is a mechanical device that will not work alone and these devices controlled by the software. Hardware devices will work on electronic charge viz positive and negative these charges handled by software programming i.e., binary languages. The binary language has two values 0 and 1.
- When compiler converts the source code into machine code then at first it checks the source code whether in the code having any syntax error that it will check from predefined keywords, tokens or values.
- If any syntax of particular keyword is different from predefined keyword values, then it will arise an error message. Compiler convert the source code into object code in one go.

Features of Compiler

1. Good error detection
2. Compile in one go
3. Fast in speed
4. Help in code debugging
5. Faultlessness
6. Easily detect the illegal errors

Types of Compilers

1. Single pass compiler 2. Two pass compilers 3. Multi pass compiler

1.2 STRUCTURE OF COMPILER

Compiler is an abstract machine. It is in between the high-level language and machine level language. Structure part define the phases of compiler. At first compiler get the data from user i.e. source code then convert these data in lexical part then data pass to another level. Every level gets the intermediate data. The output of first stage becomes the input of second stage. Structure of compiler describe in figure 2.

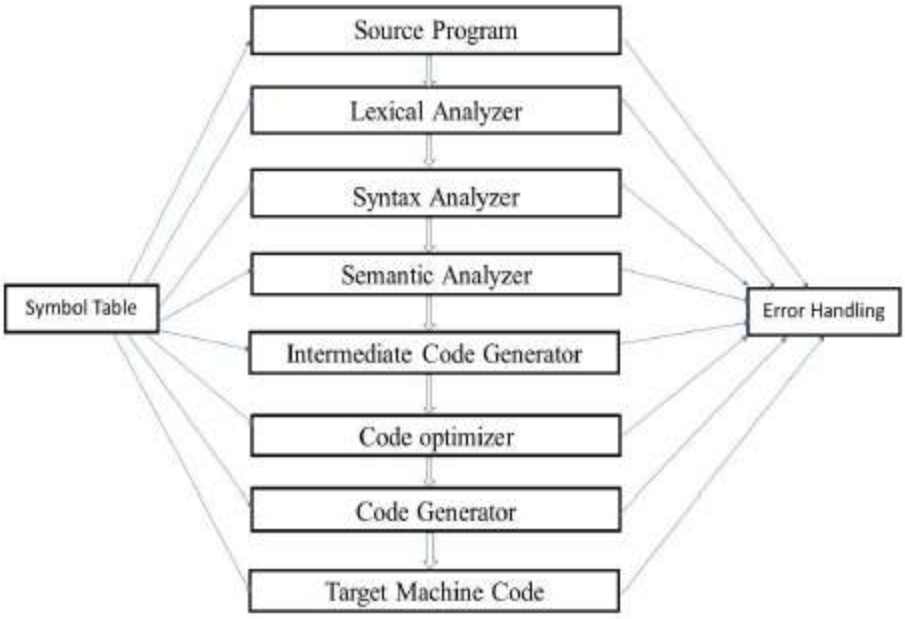


Figure 2.

1.2.1 LEXICAL ANALYSIS

1. It is the first level of compilation process.
2. At first it takes code as input from source code. Then it starts to convert the data.
3. It reads the source code one character at a time and then convert this source code into meaningful lexemes.
4. These lexemes are represented as a token in lexical analyzer.
5. It also removes the white space and comments.
6. It also uses to check and remove the lexical errors.
7. It reads the character or values from left to right.

1.2.2 SYNTAX ANALYSIS

1. It is a second phase of compiler.
2. It takes the input from lexical analysis as tokens then convert these tokens into parse tree.
3. When tokens convert into parse tree it will follow the rules of source code grammar.
4. These grammar codes as known as context free grammar.
5. This phase analyzes the parser and then check the input expressions that are syntactically correct or not.

1.2.3 SEMANTIC ANALYSIS

1. This level checks source code for semantic consistency with language definition for that it uses the syntax tree and for the information symbol table.

2. It collects all the information and checks the validity for variables, keyword, data and save it into syntax tree or in the symbol table.
3. In this analysis every operator checks weather it is having matching operands.
4. Type checking and flow checking is an important part of Semantic analysis.
5. Language specification may allow type conversion also it is known as coercions.
6. It also checks weather the language follows the rules or not.
7. It also verifies the parser tree of syntax analyzer.

For instance, coercions appear in figure 3. Assume that Interest, principal, rate have been declared to be floating point number and lexeme 70 is itself forms of an integer. The (*) operator is concern to a floating-point number rate and the integer value 70. In this case, integer value is translated into floating point number.

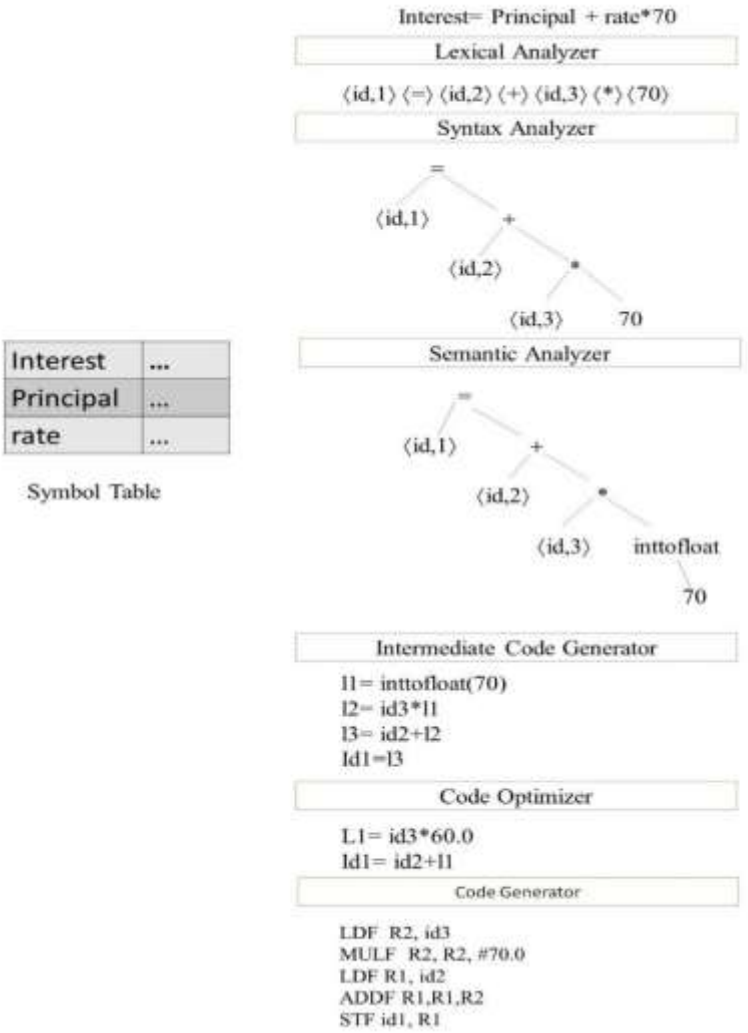


Figure 3

1.2.4 INTERMEDIATE CODE GENERATION

When compiler convert the source code into target code then compiler create one or more intermediate code. Syntax tree are the intermediate representation of syntax and semantic analysis. This code is similar for all the compilers. This intermediate representation having two properties

1. To convert into the object code or target machine
2. It produces the result in easy manner.

This is an example of intermediate code. This code consists three operators. It is also known as three-address code. Each instruction consists assignment operator. l1, l2, l3 are the temporary name that hold the values. In l1 statement integer value is converted into floating point value. In l2 statement multiplication operator are used. In l3 statement addition operator are used.

E.g.

l1= inttofloat(70)

l2= id3*l1

l3= id2+l2

Id1=l3

1.2.5 CODE OPTIMIZATION

1. In the code optimization step is to improve the intermediate code performance for better target code result. In the code optimizer firstly decide the code should be small, it will be giving the result very quickly and it will consume less power.
2. One special point is that the code should be user friendly.
3. The code optimizer also can reduce the compilation and execute time of compiler when it compiles the code.
4. This example shows that the conversion of integer value into floating point (60.0) at once after that it will use previous result.

   l1= id3*60.0

   Id1= id2+l1

1.2.6 CODE GENERATION

1. Code generation phase is an important part that takes the intermediate code value as a input and writes the code for target language.
2. Intermediate instructions are converted into sequence of target instruction code that perform a particular task.
3. During the code generation firstly decide about the variable names, keywords, operation which gives the result as per the requirement.
4. Example for code generation. F letter is use for the floating-point value and R1, R2 are the intermediate code and the first value of each statement specify the destination means where statements result will store.
5. # Symbol specifies the 70.0 is treated as immediate constants. E.g.

   LDF R2, id3

MULF R2, R2, #70.0 LDF R1, id2

ADDF R1, R1, R2 STF id1, R1

1.2.7 TARGET CODE GENERATER

1. After the completion of code generation phase execute that code and user will get the desired result.
2. If the result according to the requirements, then solve another problem if the result will not come according to the user requirements, then do some changes till the desired result will come.
3. This is final phase of compiler.
4. All the phases of compiler are divided into two parts:
    1. Front end
    2. Back end

1. Front end

In this phase all the phases come viz. lexical analysis, syntax analysis, semantic analysis and Intermediate code generation.

2. Back end

Code optimization and code generation phases comes under back-end section.

1.2.8 SYMBOL-TABLE MANAGMENT

1. Symbol table is a data structure that consist all the variable name, keyword, fields name.
2. With the help of symbol table user can easily store and get the data for each record with name quickly.
3. It collects the information for attribute of a name.
4. It also provides the detail or information for the storage, type, and its scope.
5. Different kind of data structure techniques are used to create a symbol table. Some of techniques are:
    1. Linked list
    2. Hash table
    3. Tree

E.g. int sum (int x, int y) {

add=0; add=x+y; return add;

}

Operations on Symbol table

1. Allocates the operations on symbol table
2. Insert the operations on symbol table

3. Set_attributes
4. Get_attributes
5. Free operation
6. Look up operation

1.3 LEXICAL ANALYZER

1. Lexical analyzer is the first and important phase of compiler.
2. It reads the character or value as input from the source program then make them group into lexemes and generate the output as a sequence of tokens for every lexeme.
3. All the tokens are pass to the parser for syntax analysis.
4. Lexical analyzer communicates with the symbol table.
5. When the lexical analyzer generates a lexeme, it must ne enter lexemes value into symbol table.
6. In few cases, some identifiers read from the symbol table then lexical analyzer determine the proper tokens it should be pass to the parser. The relationship between lexical analyzer, symbol table and parser as shown in figure 4.

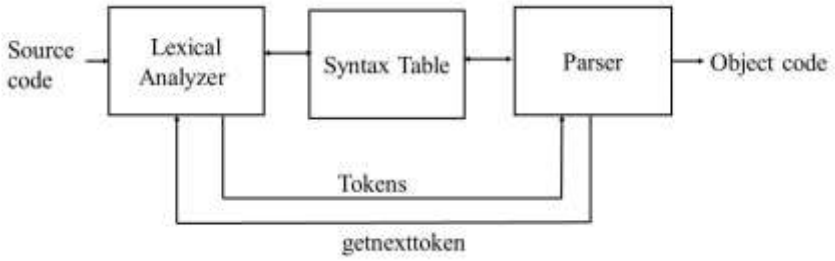


Figure 4. Interaction between lexical analyzer and parser

From this it is clear that lexical analyzer passes the tokens to parser then parser pass the system call command to lexical analyzer i.e., getnexttoken. Parser having the bidirectional link with symbol table. Symbol table is the data structure that contain the value, keywords, data types. Lexical analyzer also communicates with symbol table.

7. Its main task is to correct the error messages that are generated by the compiler.
8. It also keeps track of the number of newline character so it identifies the error message with line number.
9. Lexical analyzer is classified into two processes:
   a) Scanning contains the normal process that doesn't require tokenization of the input.
   b) Lexical analyzer produces the tokens from the result of the scanner.

10. Lexical analyzer having some important operations on languages viz. union, concatenation.
11. Union operation join two language statements.
12. Concatenation operation perform on two languages by taking a string or character from one language then take another string from second language then apply concatenation operation on them. Table 1 describes the operations on language.

OPERATION	DEFINATION AND NOTATION
Union of T and E	T U E={s1|s1 is in T or s1 is in E}
Concatenation of T and E	TE={hy|h is in T and Y is in E}
Kleene closure of T	$T^*=U_{j=0}^{\infty} T^j$
Positive closure of T	$T^*=U_{j=1}^{\infty} T^j$

Table 1. Operations on languages

Tokens Patterns and Lexemes

Tokens: Lexemes are the sequence of alphanumeric character that are known as tokens. Tokens represent a bit of information in a source code. In every language some predefined values are there that are known as keywords. All lexemes follow that predefined rule. Every programming language having some symbols i.e. keywords, punctuations, operators, operations, string these are consider as a tokens in lexical analysis

For example, in C language variables are declared as: float a=10.0;

float (keyword), a (identifier), = (operator), 10.0 (constant) ; (symbol).

Patterns: Patterns represent the description of tokens. Pattern is a structure that match with a string.

Lexemes: Lexemes are the combination of characters that find the matches for a token. Example: while (x<y) represented as that are shown in table 2:

E.g. while (x<y)

Lexeme	Tokens
while	while
(	lparen
x	identifier
<	Comparison
y	identifier
)	Rparen

Table 2.

1.4 REGULAR EXPRESSION

Regular expressions are a sequence of characters that define a search pattern. For specific patterns in regular expression having an important

notation. In every programming language having tokens these are described as regular expression. In regular expression anchors are using "^" and "$" as a character in the string. The "^" represent the starting of the string and "$" represent the ending of the string. Regular expression having a series of characters that matched a text pattern.

Examples of Regular Expression:

1. The string is with same alphabet (a,b) but the language is starting with a and ending with a. {a, aa, aaa, aaaa, aaaaa …..}

   {aba, abba, abbba, abbbba ……}

   These strings are infinity then the language also infinite.

2. ^..J[.]* : Sort the string those are having 3 letter of their name is J. {aajabab, abjbaba}

3. [.]*F : Sort the string those are having the ending letter will be F.

4. (b+a)* set of string b's and a's any amount of character including null string. So T={ ε, b, a, bb, ba, aa, ab, bbb …..}

5. (b+a)*bba set of string b's and a's that ending with the string bba. So T={bba, abba, aabba, babba……}

Table 3 is to describe the most common characters that used in regular expression.

Character	Description
[set]	Describe a character in parentheses that match any single character from a set.
*	* describe that any amount of character and any character combination will come.
.	The dot describes each string having single character. E.g. (..F..) means in the string having third character should be F Before F any letter will come and after F also any letter will come.
$	$ describe end of the string before that sign any letter is there that will become the last letter of a string. E.g ..H$ last character of a string is H.
^	^ describe the beginning of string.

Table 3. Regular expression character

Applications of Regular expression: simple parsing, data wrangling, data validation, data scraping.

Regular Language: Regular expression is formal grammar that are used for parse string and textual information this is known as Regular languages. languages are regular if the expression are regular expression.

For regular languages regular expressions are used. Expressions are regular if:

1. ϕ is a regular expression for regular language ϕ.
2. ε is a regular expression for regular language $\{\varepsilon\}$.
3. If $b \in \Sigma$ (Σ used for input alphabets), b is regular expression with language {b}.
4. If x and y are regular expression, x+y is also a regular expression with language {x,y}.
5. If x and y are regular expression, xy is also regular.
6. If b is regular expression, then b* (more times of b or 0) will be regular.

Closure Properties of Regular Languages

1. **Union:** If T1 and T2 are two regular languages, then union of T1 ∪ T2 will also be regular. For example, T1 = {bm | m ≥ 0} and T2 = {am | m ≥ 0} T3 = T1 ∪ T2 = {bm ∪ am | m ≥ 0} is also regular.
2. **Intersection:** If T1 and T2 are two regular languages, then intersection of T1 ∩ T2 will also be regular. For example, T1 = $\{b^m \mid m \geq 0\}$ and T2 = $\{a^m \mid m \geq 0\}$ T3 = T1 ∩ T2 = $\{b^m \cap a^m \mid m \geq 0\}$ is also regular.
3. **Concatenation:** If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example, T1 = $\{b^m \mid m \geq 0\}$ and T2 = $\{a^m \mid m \geq 0\}$ T3=T1.T2=$\{b^m.a^m \mid m \geq 0\}$ is also regular.
4. **Kleene Closure:** If T1 is a regular language, its Kleene closure T1* will also be regular. For example, T1 = (b ∪ a) T1* = (b ∪ a)*
5. **Complement:** If T(Y) is regular language, then its complement T'(Y) will also be regular. For example, T(Y) = $\{b^m \mid m > 3\}$ T'(Y) = $\{b^m \mid m <= 3\}$.

1.5 FINITE AUTOMATA

1. A finite automaton is a machine that accept patterns. it has a group of states and rules through these rules one state moves to another state.
2. To recognize patterns, use finite automata.
3. It takes the input from string and change that string into state. Transition state occur when the desired symbol will find.
4. When transition function occurs, the automata move from one state to another or again move on the same state.

5. Finite automata having two states, Accept or Reject state. When the automata reach its final state, it means the string processed successfully.
6. Finite automata consist of:
    - A finite set T of M states.
    - Start state.
    - Accepting or final state.
    - Moving from one state to another state use transition function.

Definition of FA

It is a collection of 5 tuples (Q, Σ, δ, q0, F)

1. Q : To represent the Finite state
2. ∑: To represent the input symbol
3. q0: To represent the initial state.
4. F: to represent the final state.
5. δ: perform the transition function on string

Finite Automata Construction

1. States: In FA states are represented by circle.
2. Start state: The start state pointed with arrow. It represents the starting state for finite automata.
3. Intermediate states: IN intermediate state having two arrows one pointing to and another arrow pointing out.
4. Final state: If the string will successfully accept then automata reach to its final state. It represents with a doble circle.
5. Transition: when string successfully accepted then initial state moves to another state. This process continues till it reach to its final state.

Finite Automata Model:

Finite automata represented by finite control and input tape as shown in figure 5.

- **Input tape:** In the input tape input are placed in each cell.

  Finite control : The finite control receives the input and then decides the next state. It takes the input from input tape. Tape reads the cells from left to right, and it read the input one by one.

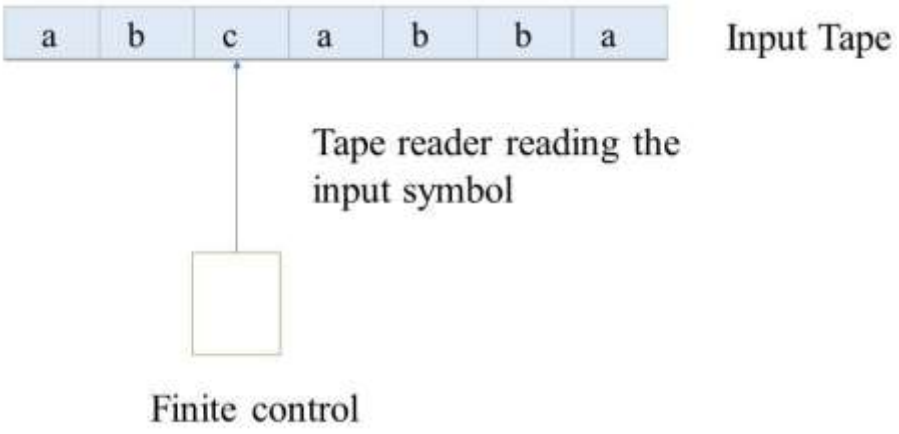


Figure 5. Finite automata model

Types of Automata:

- Automata classified into two categories as shown in figure 6:
  1. DFA (Deterministic finite automata)
  2. NFA (Non-deterministic finite automata)

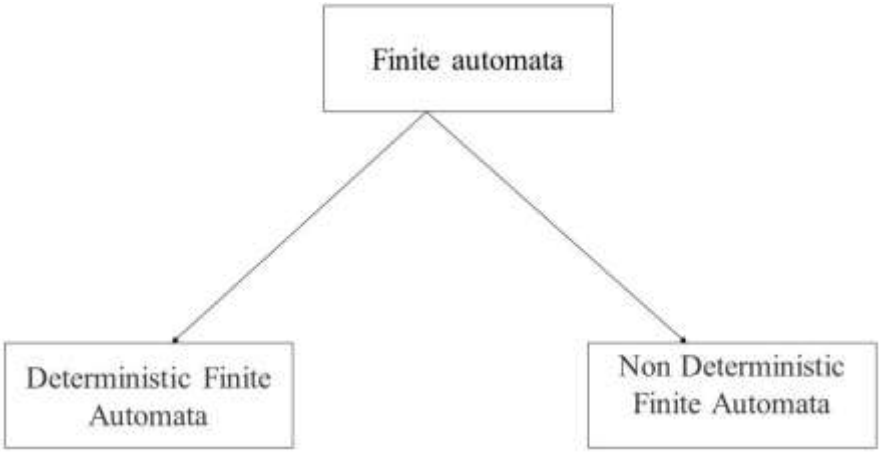


Figure 6. Types of Automata

DFA: DFA is the deterministic finite automata. In DFA, the machine having only one state for a single input character. It refers to a uniqueness of computation. It doesn't accept the null moves.

NFA: NFA refers to non-deterministic finite automata. In NFA, the machine having multiple states for a particular input character. It accepts the null move.

Important points about DFA and NFA:

1. Every NFA is not DFA but every DFA will be NFA.
2. There are multiple final states in NFA and DFA.
3. In compiler DFA used lexical analyser

DFA

1. DFA is the deterministic finite automata. In DFA, the machine having only one state for a single input character. It refers to a uniqueness of computation.
2. It doesn't accept the null moves.
3. In DFA, from the input state to output state there will be a one path for a particular input.
4. DFA having multiple final states that are use in lexical analyzer.

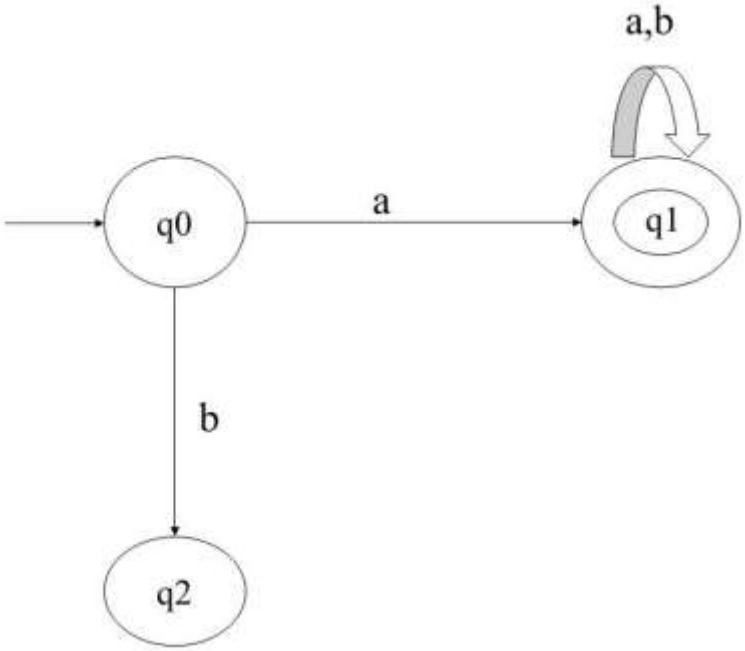


Figure 7. DFA

In the following diagram 7 it is showing that q0 is the initial state. q1 is the final state, when a input apply on state then the next state will become q1 that is final state. When b input apply on q0 state then the next state will be q2. State q1 having a self-loop.

Definition of DFA

DFA having 5 tuples.

1. Q: Represent the states.
2. ∑: To represent input symbol.
3. q0: Initial state.
4. F: Final state
5. δ: transition function

Transition function represented as:

δ: Q x ∑ ->Q

E.g.

1. Q = {q0,q1,q2}
2. ∑= {a,b}
3. q0 = {q0}
4. F= {q1}

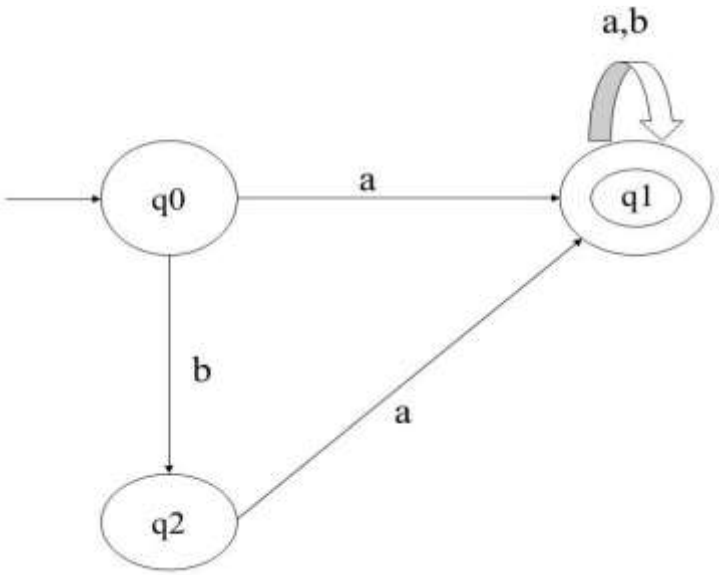


Figure 8.

Transition function: Table 4 represent the transition function

Present state	Next state for input a	Next state for input b
q0	a	q1
q0	b	q2
q2	a	q1
q1	a ,b	q1

Table 4.

1.5.1 CONVERSION FROM REGULAR EXPRESSION TO FINITE AUTOMATA

To convert the regular expression to finite automata, use subset method some steps are:

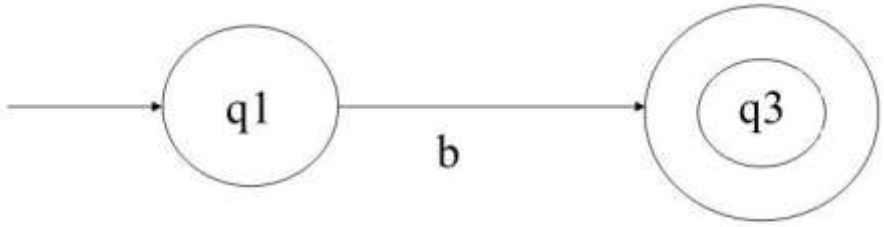
Step 1 − Construct a Transition diagram for a given RE by using non-deterministic finite automata (NFA) with ε moves.

Step 2 − Convert NFA with ε to NFA without ε.

Step 3 − Convert the NFA to the equivalent Deterministic Finite Automata (DFA).

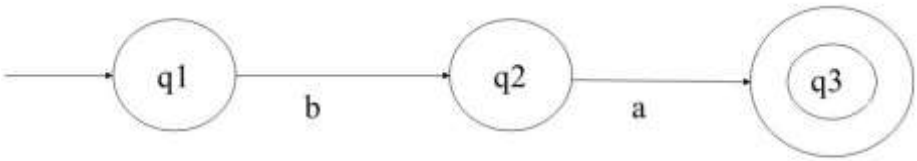
Example to convert R.E to Finite automata.

Case 1: Construct finite automata, for regular expression ‘b’.



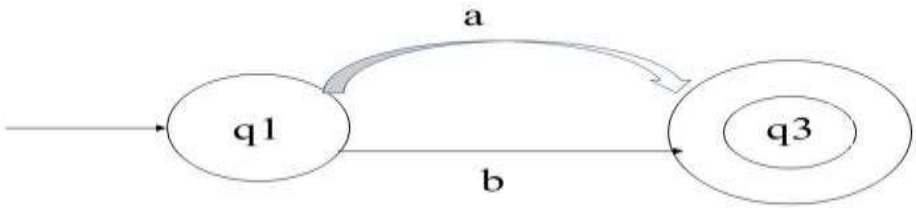
Finite Automata for R.E= b

Case 2: Construct finite automata, for regular expression ‘ba’.



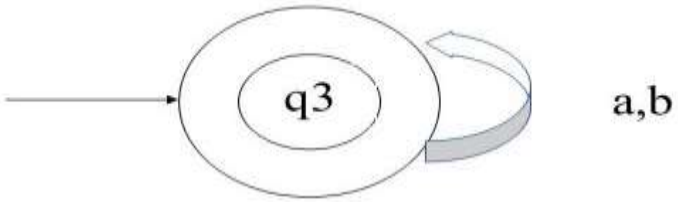
Finite Automata for R.E= ba

Case 3: Construct finite automata, for regular expression ‘(b+a)’.



Finite Automata for R.E= (b+a)

Case 4: Construct the finite automata, for regular expression ‘(b+a)*’.



Finite Automata for R.E= (b+a)*

1.5.2 MINIMIZING THE STATES OF DFA

Minimization means reducing the number of states of FA. There are some steps to minimize DFA.

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. $\delta (q, a) = p$
2. $\delta (r, a) = p$

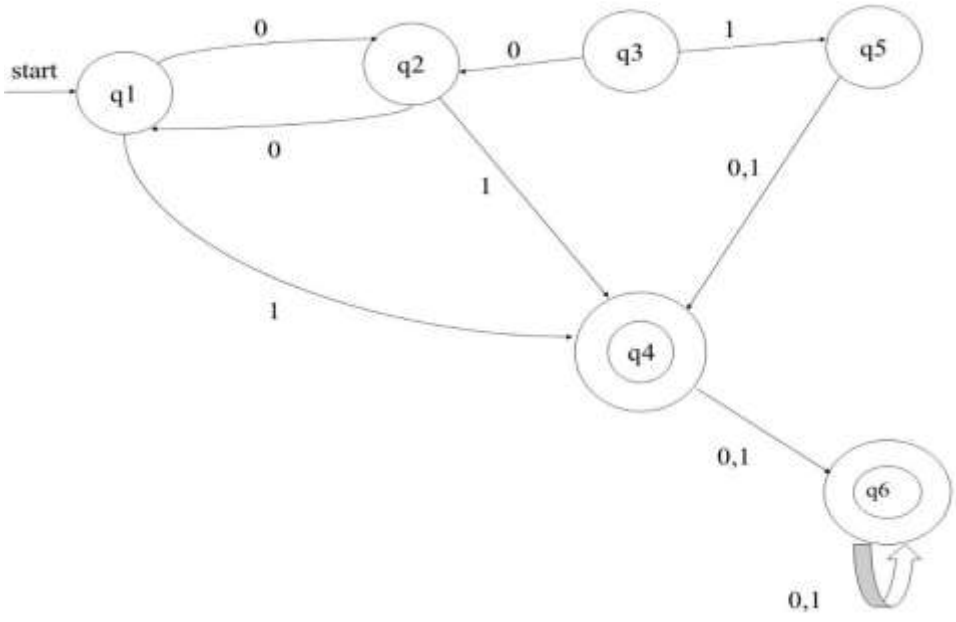
That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:



Solution:

Step 1: In the given DFA, q3 and q5 are the unreachable states so remove them.

Step 2: For the other states draw transition table.

State	0	1
→q1	q2	q4
q2	q1	q4
*q4	q6	q6
*q5	q6	q6

Step 3: Now break the transition table into two sets:

1. One set having non-final states:

State	0	1
q1	q2	q4
q2	q1	q4

2. Another set having final states.

State	0	1
q4	q6	q6
q6	q6	q6

Step 4: Set 1 doesn't have any similar rows so it will be the same.

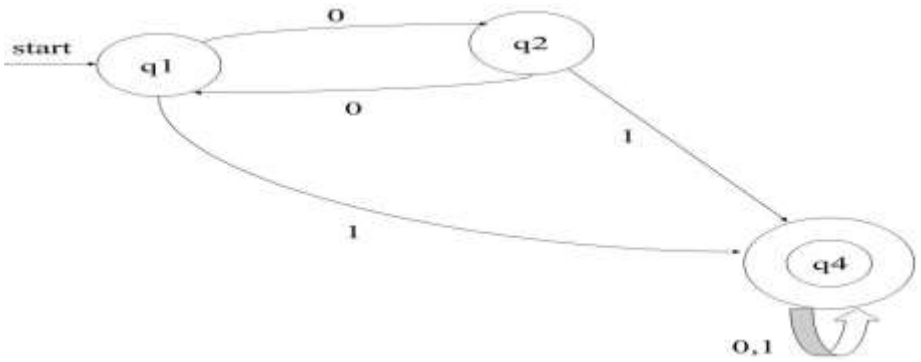
Step 5: In set 2, row 1 and row 2 having similar states q4 and q6 on 0 and 1.so skip q6 and replace with q4.

State	0	1
q4	Q4	q4

Step 6: Now join set 1 and set 2.

State	0	1
→q1	q2	q4
q2	q1	q4
*q4	q4	q4

Minimize DFA shown as:



1.6 CONTEXT FREE GRAMMAR

CFG (context free grammar) is a formal grammar. It is used to produce string in a formal language.

There are the various capabilities of CFG:

1. Context free grammar is very helpful to describe programming languages.
2. If the grammar properly defined then parser easily construct automatically.
3. It is efficient to describe the nested structure viz. if-then-else, balanced parentheses and so on.

CFG defined in 4 tuples. G = (V, T, P, S)

Where,

G define the grammar

T define a finite set of terminal symbols.

V define a finite set of non-terminal symbols

P define a set of production rules

S represent start symbol.

In CFG, S represent the start symbol that is used to proceed the string. The string is proceeded with a non-terminal until all the non-terminal symbol have been exchanged by terminal symbols.

Example:

L= {tft^R | t € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG baacaab.

1. S ⇒ bSb
2. S ⇒ baSab
3. S ⇒ baaSaab
4. S ⇒ baacaab

Applying the production S → bSb, S → aSa recursively and at last we get the final production S →c, now we get the final string baacaab.

Classification of Context Free Grammars

Context free grammar are divided into two properties:

1. Number of strings it generates.
    - If context free grammar producing number of strings in finite order, then that grammar will be non-recursive grammar.
    - If context free grammar producing number of strings in infinite order, then that grammar will be non-recursive grammar.
2. Number of derivation trees.
    - Context free grammar is unambiguous if only one derivation tree is there.
    - Context free grammar is ambiguous if more than one derivation tree is there.

Examples of Recursive and Non-Recursive grammar

1. S ->SbS

S->a

The language produced by this grammar: {a, aba, ababa, …….} this is infinite. So this grammar is recursive grammar.

2. S ->Ba B ->b | c

The language produced by this grammar: {ba, ca} this is finite. So, this grammar is non-recursive grammar.

Types of recursive grammar

A recursive grammar classified into 3 types

1. General recursive grammar
2. Left recursive grammar
3. Right recursive grammar

1.7 DERIVATION AND PARSE TREE

The process of preceding the string is known as derivation. At every step of derivation, make two decisions. Firstly, take the decision which non-terminal symbol should be replaced. Secondly take the decision for replacing the non-terminal symbol which production rule should be use. Every non-terminal symbol replaced with more than one derivation in the identical production rule but order of exchanging non-terminal symbol will be different. There are two types of derivation as shown in fig 9.

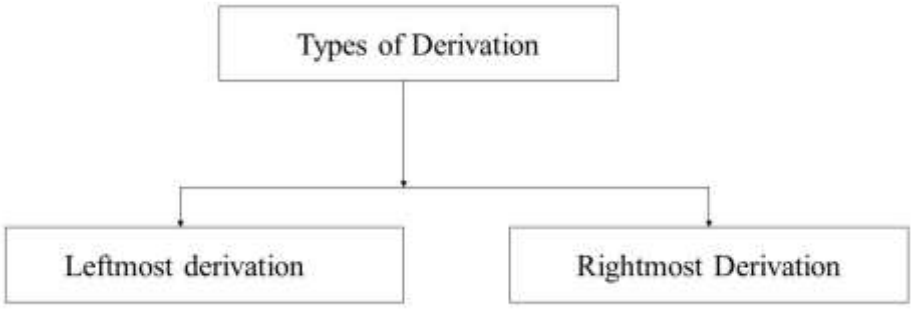


Figure 9.

1. Leftmost Derivation-

- The process of preceding the string by enlarge the rightmost non-terminal at every step is known as leftmost derivation.
- The rightmost derivation's representation in geometrical form is known as leftmost derivation tree.

Example-

Consider the following grammar-

S → bB | aA

S → bS | aAA | b

B → aS | bBB | a

(**Unambiguous Grammar**)

Let us consider a string w = bbbaabaaab

Now, let us derive the string w using leftmost derivation.

Leftmost Derivation-S → b**B**

→ bb**B**B (Using B → bBB)

→ bbb**B**BB (Using B → bBB)

→ bbba**BB (Using B → a)**

→ bbbaa**B (Using B→ bBB)**

→ bbbaab**B**B (Using B → bBB)

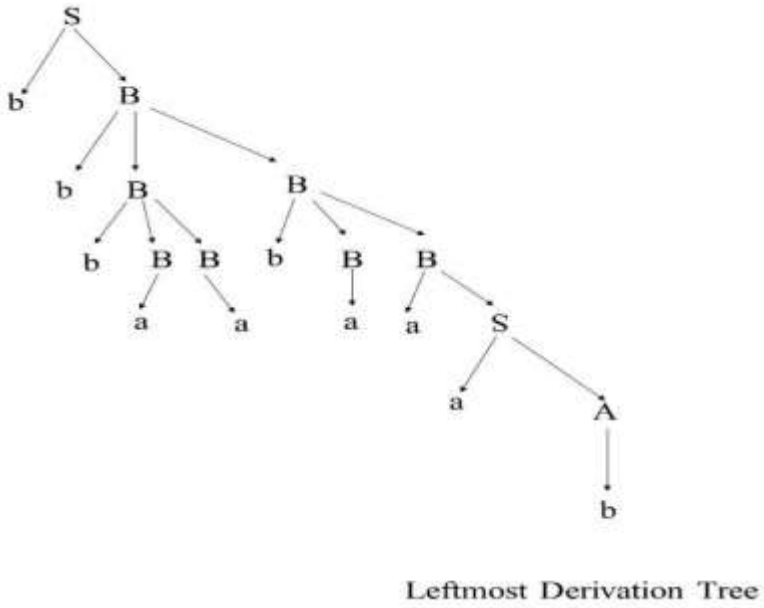
→ bbbaaba**B** (Using B→ a)

→ bbbaabaa**S** (Using B → aS)

→ bbbaabaaa**A** (Using S→ aA)

→ bbbaabaaab (Using A → b)

Parse tree in fig. 10.



Leftmost Derivation Tree

Figure 10. Leftmost Derivation Parse Tree

2. Rightmost Derivation-

- The process of preceding the string by enlarge the rightmost non-terminal at every step is known as rightmost derivation.
- The rightmost derivation's representation in geometrical form is known as rightmost derivation tree.

Example-

Consider the following grammar-

S → bB | aA

S → bS | aAA | b

B → aS | bBB | a

(**Unambiguous Grammar**)

Let us consider a string w = bbbaabaaab

Now, derive the string w using rightmost derivation.

Rightmost Derivation-

S →b**B**

→ bbB**B** (Using B → bBB)

→ bbBbB**B** (Using B → bBB)

→ bbBbBa**S** (Using B → aS)

→ bbBbBaa**A** (Using S → aA)

→ bbBb**B**aaB (Using A → a)

→ bbBbaaab (Using B→ a)

→ bbbB**B**baaab (Using B → bBB)

→bbbBabaaab (Using B → a)

→ bbbaabaaab (Using B a)

Parse tree in figure 11.

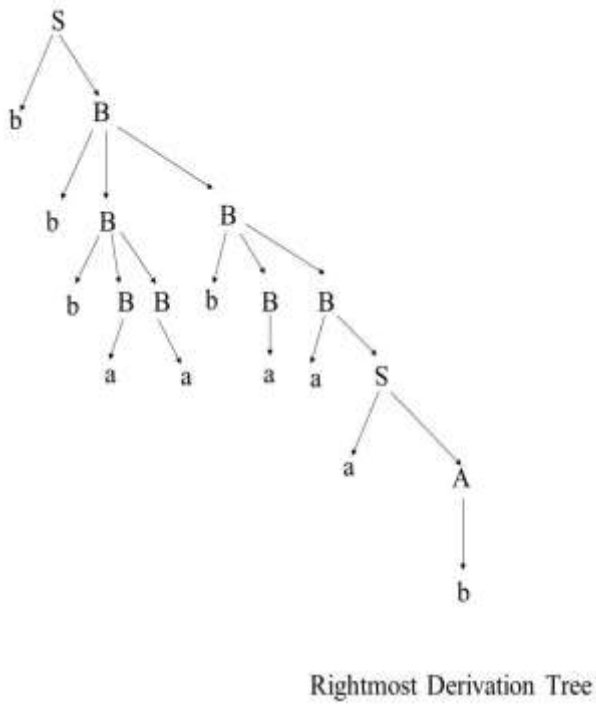


Figure 11. Rightmost Derivation Parse Tree

Properties Of Parse Tree-

1. The start symbol of grammar is known as Root node of a parse tree.
2. The terminal symbol of a parse tree is represented as a leaf node.
3. The non-terminal symbol is the interior node of a parse tree.
4. Parse tree is independent when the productions are used derivations.

1.8 PARSERS

- It is a compiler that divide the data into smaller elements that get from lexical analysis phase.
- Parser takes the input into set of tokens form then output will become in parse tree.
- It is having two types describe in figure 12:

Top-down parsing Bottom-up parsing

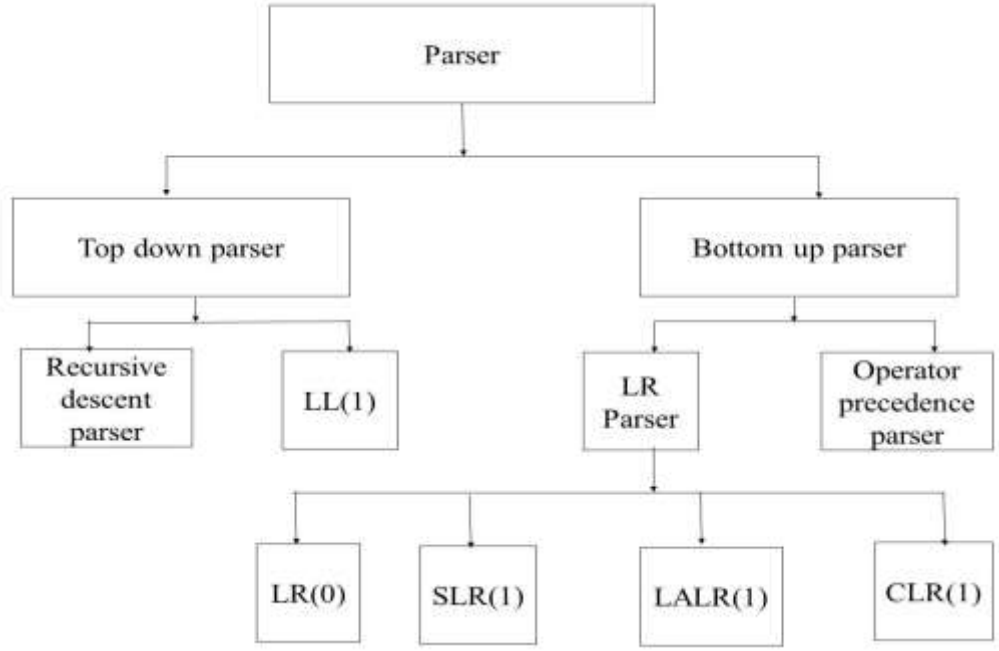


Figure 12.

Top-Down Parsing

1. Top-down parsing is called as recursive or predictive parsing.
2. To construct a parse tree use, bottom-up parsing.
3. In top down parsing the process start from the start symbol and convert it into input symbol.
4. Top-down parser are categories into 2 parts: Recursive descent parser, and non-recursive descent parser.

(i) Recursive descent parser:

It is also called as Brute force parser or backtracking parser.

(ii) Non-recursive descent parser:

To generates the parse tree, use parsing tree rather backtracking.

Bottom-up parsing

1. Bottom-up parsing is called as shift-reduce parsing.
2. To construct a parse tree use, bottom-up parsing.
3. In bottom up parsing the process start from the start symbol and design a parse tree from the start symbol by touching out the string from rightmost derivation in reverse.

Example: Production

1. T → P
2. P→ P * E
3. P → id
4. E → P
5. E → id

Parse Tree representation of input string "id * id" is as follows:

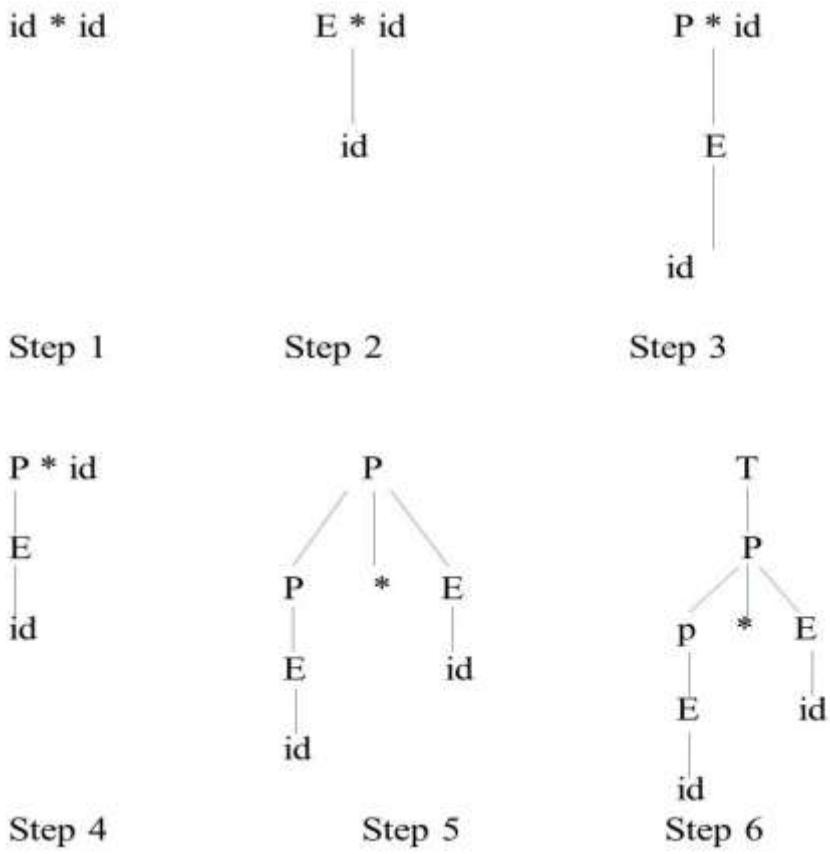


Figure 13.

Bottom-up parsing having various parsing techniques.

1. Shift-Reduce parser
2. Operator Precedence parser

3. Table Driven LR parser

- LR(1)
- SLR(1)
- CLR (1)
- LALR(1)

Further Bottom-up parser is classified into 2 types: LR parser, and Operator precedence parser. LR parser is of 4 types:

(a). LR(0) **(b).** SLR(1) **(c).** LALR(1) **(d).** CLR(1)

i) LR parser:

It generates the parse tree for a particular grammar by using unambiguous grammar. For the derivation it follows right most derivation. LR parser having 4 types:

(a). LR(0) **(b).** SLR(1) **(c).** LALR(1) **(d).** CLR(1)

ii) Operator precedence parser:

It generates the parse tree for a particular grammar or string with a condition i.e., two consecutives non terminal and epsilon doesn't come at the right-hand side of any production.

1.8.1 SHIFT-REDUCE PARSING

1. In Shift reduce parsing reduce a string of a grammar from the start symbol as figure 14.

2. a string of a grammar from the start symbol.

3. It uses a stack to hold the grammar and to hold the string it uses input tape.

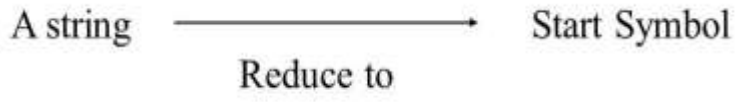


Figure 14.

4. It performs two actions: shift and reduce so it is known as shift reduce parsing,

5. When shifting process start then the current symbol of string move to the stack.

6. Shift reduce parsing having 2 categories:

Example

Operator Precedence Parsing

LR-Parser

$A \rightarrow A+A$ $A \rightarrow A-A$ $A \rightarrow (A)$ $A \rightarrow a$

Input string: x1-(x2+x3)

Parsing table: Describe in Table 5.

Input String: x1-(x2+x3)

Stack	Input string	Action
$	x1-(x2+x3)	shift x1
$x1	-(x2+x3)$	reduce by A
$A	-(x2+x3)$	shift -
$A-	(x2+x3)$	shift (
$A-(	x2+x3)$	shift x2
$A-(x2	+x3)$	reduce by A→a
$A-(A	+x3)$	shift +
$A-(A+	x3)$	shift x3
$A-(A+x3	)$	reduce by A→a
$A-(A+A	)$	shift)
$A-(A+A)	$	reduce by A→ A+A
$A-(A)	$	reduce by A→ (A)
$A-A	$	reduce by A→A-A
$A	$	Accept

Table 5.

1.8.2 OPERATOR-PRECEDENCE PARSING

1. It is related to small type of class operator grammar.
2. If the grammar is the type of operator precedence, then it should have two properties:
3. The production should not have any a∈ operator at right side.
4. Non-terminal symbols are not adjacent.
5. It can only perform the operation between the terminal symbols of grammar. It doesn't take any notice to non-terminals.
6. Operator precedence are categories into three relations ⋗ ⋖ ≐.

   x ⋗ y means that terminal "x "has greater precedence than the terminal "y".

   x ⋖ y means that terminal "y "has higher precedence than the terminal "x".

   x ≐ y means that precedence of terminal "x and y "are equal.
7. Operator precedence parser comes under bottom-up parser that interprets with operator grammar.
8. In this parser ambiguous grammar is not allowed.
9. There are 2 ways that determine which precedence relation should hold the pair of terminals.

1. Use precedence of order and conventional associativity.
2. Firstly, construct the unambiguous grammar for a particular language, that grammar reflects the correct precedence of parse tree.

Precedence table:

	+	*	(	)	id	$
+	⩾	⩽	⩽	⩾	⩽	⩾
*	⩾	⩾	⩽	⩾	⩽	⩾
(	⩽	⩽	⩽	≐	⩽	X
)	⩾	⩾	X	⩾	X	⩾
id	⩾	⩾	X	⩾	X	⩾
$	⩽	⩽	⩽	X	⩽	X

Table 6.

Parsing Action

1. At the end of string use $ symbol.
2. After that scan input string from left to right until ⩾ is encountered.
3. Now scan the string towards left above all the equal precedence Until first left ⩽ is encountered.
4. Now handle all the string value that lie between ⩽ and ⩾.
5. If at last we get $ it means the parsing is successfully accepted.

Example: Grammar:

S → S+E/E E → E* F/F F → id

Given string:

1. w = id + id * id

Let us consider a parse tree for it as follow:

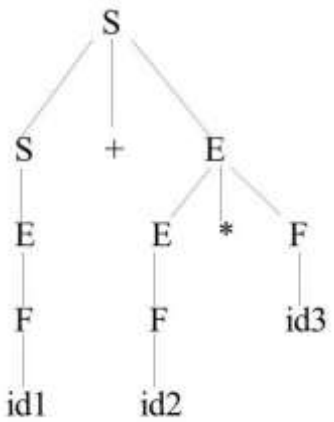


Figure 15. Parse tree

According to parse tree, we can design operator precedence table describe in table 7:

	S	E	F	id	+	*	$
S	X	X	X	X	≐	X	⋗
E	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

Table 7. Operator Precedence Table

Now process the string through precedence table as shown in figure 16.

$ ⋖ id1 ⋗ + id2 * id3 $
$ ⋖ F ⋗ + id2 * id3 $
$ ⋖ E ⋗ + id2 * id3 $
$ ⋖ S ≐ + ⋖ id2 ⋗ * id3 $
$ ⋖ S ≐ + ⋖ F ⋗ * id3 $
$ ⋖ S ≐ + ⋖ E ≐ * ⋖ id3 ⋗ $
$ ⋖ S ≐ + ⋖ E ≐ * ≐ F ⋗ $
$ ⋖ S ≐ + ≐ E ⋗ $
$ ⋖ S ≐ + ≐ E ⋗ $
$ ⋖ S ⋗ $
Accept

Figure 16.

Disadvantage of operator precedence parser

If we have n number of operators then the size for table will be n*n. Then complexity will be $0(n^2)$. To decrease the size of table, use operator function table. Operator precedence parsers use precedence function that plot terminal symbols to integer, and the relations between the symbol are affected by numerical comparison. Parsing table enclosed by two precedence function **f** and **g** that plot terminal symbols to integers.

1. f(a) < g(b) takes the precedence to b
2. f(a) = g(b) a and b have the same precedence
3. f(a) > g(b) takes the precedence over b

1.8.3 TOP-DOWN PARSING

1. Top-down parsing is called as recursive or predictive parsing.

2. To construct a parse tree use, bottom-up parsing.

3. In top down parsing the process start from the start symbol and convert it into input symbol.

4. It always uses left most derivation.

5. It is a parser that generate the parse for a particular string through the help of grammar production.

6. Top-down parser categories into two parts as shown in figure 17.

   a) Back tracking

   b) Non-backtracking

      - Predictive Parser
      - LL Parser

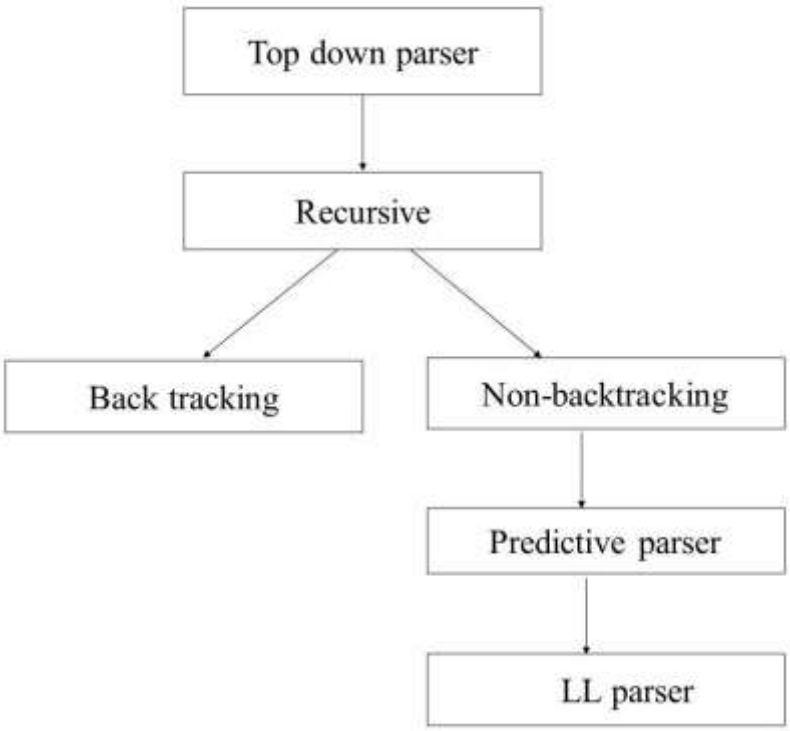


Figure. 17

Recursive Descent Parsing

Recursive descent parser is a top-down parser. It starts to construct the parse tree from the top and it reads the input from left to right. It is use to process each terminal and non-terminal entities. This technique is use to make the parse tree for that parser recursively pass the input. Context free grammar is recursive in nature so this grammar uses in recursive descent parsing.

A technique that doesn't require any backtracking that are known as predictive parser.

Back-tracking

Top-down parser match the input string against the production rule from the start node. Example for CFG:

X → pYd | pZd

Y → ka | wa

Z → ae

It will start the production rule from the root node i.e., X and start to find the match of a letter from left most input i.e., 'p'. The production of X (X → pYd) match with it. Then top-down parser proceeds to the next input string i.e., 'w'. Now parser find the match for non-terminal 'Y' and check the production for (Y → ka). This string doesn't match with input string. So top-down parser backtracks to get the result of Y. (Y → wa). The parser matches complete string in ordered manner. Hence string is accepted as shown in figure 18.

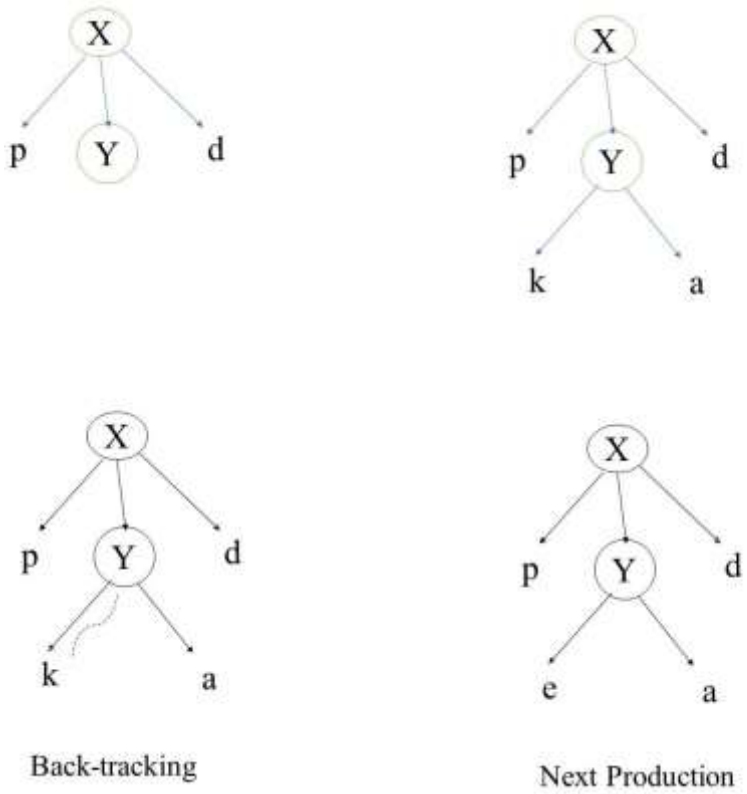


Figure 18.

Example for Top-down parser:

Input string: "adbc"

Representation of Parse Tree for input string "adbc" is as shown in figure 19

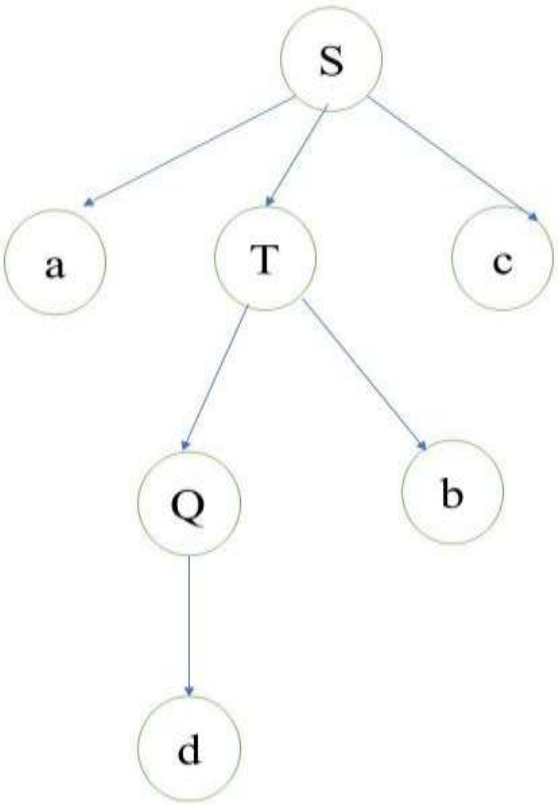


Figure 19. Parse tree

1.8.4 PREDICTIVE PARSING

1. Predictive parsing is a top-down parser it doesn't require backtracking.
2. It is a recursive descent parser but backtracking is not there.
3. Predictive parsing steps for Pre-processing
   - From the grammar remove left recursion.
   - On the resultant grammar perform left factoring.
   - From the grammar remove ambiguity.
4. Point to the next input symbols predictive parser use a look ahead pointer.
5. The predictive parser use some limit on the grammar to make parser free from back tacking.
6. It is also called as LL(1) parser.
7. The predictive parser use few constrains on the grammar and will get only in LL(k) grammar.

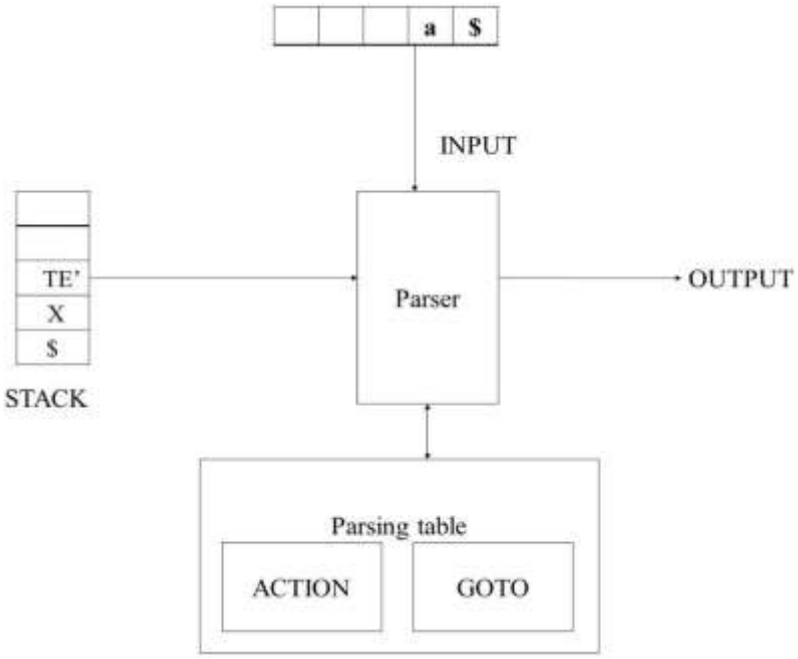


Figure 20.

Predictive parsing uses a parsing table to parse the input and for storing the data in the parse table it uses stack and then prepare the parse tree. Stack an input string contains $ symbol at the end. $ symbol represent the stack is empty and the inputs are used. Parser use the parsing table for the combination of input and stack elements. Describe in figure 20.

Describe the processing of parsers

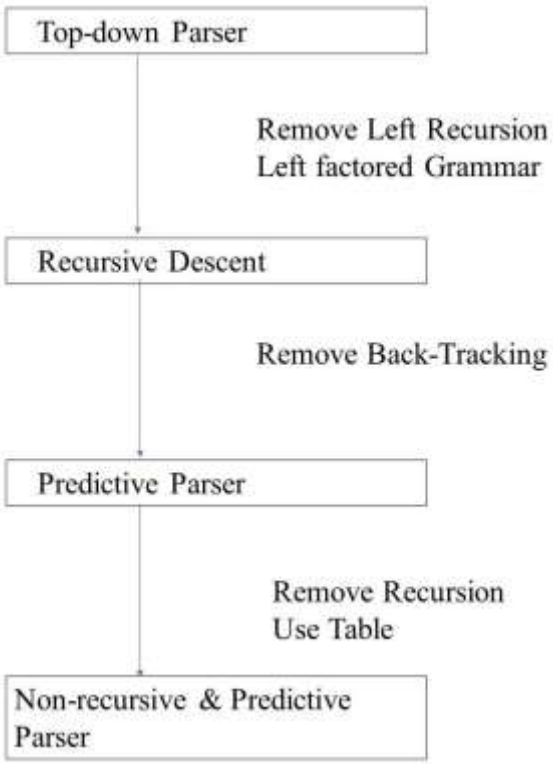


Figure 21.

Recursive descent parser has more than one input production then it chooses one production from the production, whereas predictive parser use table to generate the parse tree.

Predictive Parser Algorithm:

1. Construct a transition diagram (DFA/NFA) for each production of grammar.
2. By reducing the number of states, optimize DFA to produce the final transition diagram.
3. Simulate the string on the transition diagram to parse a string.
4. If the transition diagram reaches an accept state after the input is consumed, it is parsed.

LL Parser:

- LL Parser accept LL grammar.
- It is a subset of context free grammar.
- It follows the rule of context free grammar but with some restriction to get easier version. LL grammar implemented with recursive descent or table-driven algorithm.
- It is denoted as LL(K). first L represent the parse input from left to right, second L represent left most derivation and k represent look ahead symbols. Usually, k=1, so LL(K) also written as LL (1) as described in figure 22.

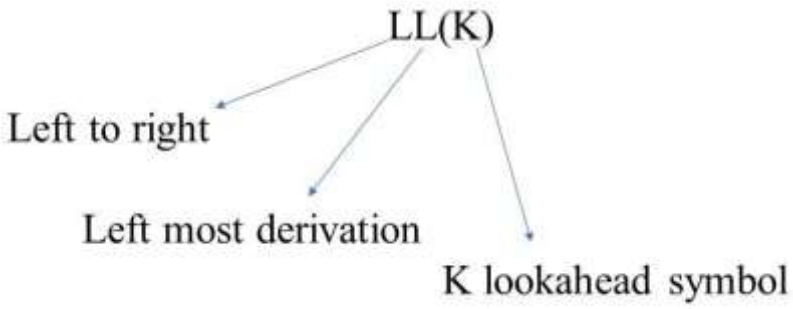


Figure 22.

1.9 SUMMARY

1. Computers are combination of hardware and software.
2. Hardware is a machinal devices these can't understand human language so we write programs into programming language that is high level language but these programs can't understand by computer then compiler convert this high-level language into machine level language.

3. Compilers are classified into three categories
   a. Single pass compiler
   b. Two pass compilers
   c. Multi pass compiler
4. Compiler is a software that convert high level language into machine level language.
5. When compiler convert the one language into another language then it doesn't change the meaning of code it only finds the syntax errors.
6. In lexical analyzer helps to identify the tokens from symbol table. 7. Lexical analysis implemented with DFA.
8. Lexical analyzer removes the white space and comments. 9. Lexical analyzer breaks the syntax into series of tokens.
10. Syntactic analysis collects all the information and checks the validity for variables, keyword, data and save it into syntax tree or in the symbol table.
11. Top-down parsing is called as recursive or predictive parsing. 12. Operator precedence are categories into three relations $\gtrdot \lessdot \doteq$.
13. Parser is a compiler that divide the data into smaller elements that get from lexical analysis phase.
14. DFA is a collection of 5 tuples (Q, Σ, δ, q0, F)
    a. Q : To represent the Finite state
    b. $\sum$: To represent the input symbol
    c. q0: To represent the initial state.
    d. F: to represent the final state.
    e. δ: perform the transition function on string
15. Derivation having two parts

1. Left most derivation
2. Right most derivation

16. CFG (context free grammar) is a formal grammar. It is used to produce string in a formal language.
17. Predictive parsing uses a parsing table to parse the input and for storing the data in the parse table it uses stack and then prepare the parse tree.
18. Operator precedence parsing can only perform the operation between the terminal symbols of grammar. It doesn't take any notice to non-terminals.
19. A recursive grammar classified into 3 types
    a. General recursive grammar
    b. Left recursive grammar

c. Right recursive grammar

20. Finite automata having two types;
    1. DFA (Deterministic finite automata)
    2. NFA (Non-deterministic finite automata)

1.10 EXCERSICE

1) Define Complier?
2) What is the difference between compiler and interpreter? 3) What is symbol table?
4) What are the phases/structure of compiler?
5) Define applications of compiler?
6) The regular expression (1*0)*1* denotes the same set as

   (A) 0*(10*)*

   (B) 0 + (0 + 10)*

   (C) (0 + 1)* 10(0 + 1)*

   (D) none of these
7) Which one of the following languages over the alphabet {1,0} is described by the regular expression? (1+0)*1(1+0)*1(1+0)*
8) Which of the following languages is generated by given grammar? X -> bS | aS | ∈
9) DFA with ∑ = {0, 1} accepts all ending with 1.
10) Assume FA accepts any three digit binary value ending in digit 0 FA = {Q(q0, qf), Σ(0,1), q0, qf, δ}
11) Consider the grammar

    S → aB | bA

    A → a | aS | bAA

    B → b| bS | aBB

    For the string w = aabbabab, find-
    1. Leftmost derivation
    2. Rightmost derivation
    3. Parse Tree
12) Consider the grammar-

    S → X1Y

    X → 0X | ∈

    Y → 0Y | 1Y | ∈

    For the string w = 11010, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

13) Construct Regular expression for the language L= {w ε{1,0}/w.

14) Define the parts of string?

15) Define DFA and NFA?

16) Differentiate between Recursive descent and Predictive parser?

17) Describe the language denoted by the R.E. (0/1)*0(0/1)(0/1).

18) Define the steps of lexical analyzer?

19) Explain Parsers and its types?

20) Write the R.E. for the set of statements over {x, y, z} that contain an even no of x's.

21) What is parse tree?

22) Write down the operations on languages?

23) What is regular expression? Write down the rules for R.E? 24) Define the types of top-down parser?

25) Explain Top-down parser and bottom-up parser?

❄❄❄❄❄❄❄

2

AUTOMATIC CONSTRUCTION OF EFFICIENT PARSERS

Unit Structure

2.1 OBJECTIVES

The main objective to use Efficient Parsers is it imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors.

2.2 INTRODUCTION

Every programming language has rules that prescribe the syntactic structure of well formed programs. The syntax of programming language constructs can be described by context- free grammars or BNF (Back us – Naur Form) notation. For certain classes of grammars, we can automatically construct an efficient parser that determines if a source program is syntactically well formed. In addition to this, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that does not remain undetected in the initial design phase of a language and its compiler.

2.3 OVERVIEW

At the end of this chapter you will know and understand the following concepts in detail :-

1) Parsing methods used in compilers.
2) Basic concepts.
3) Techniques used in Efficient Parsers.
4) Algorithms – to recover from commonly occurring errors.

2.4. BASIC CONCEPTS RELATED TO PARSERS

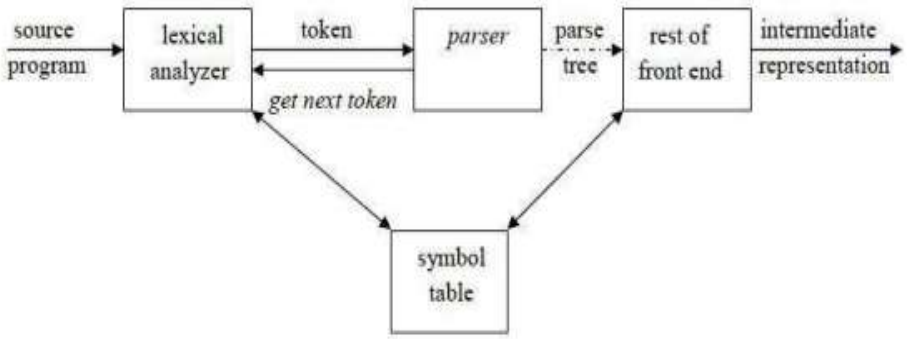
2.4.1 The Role of the Parser:-

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w, if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.

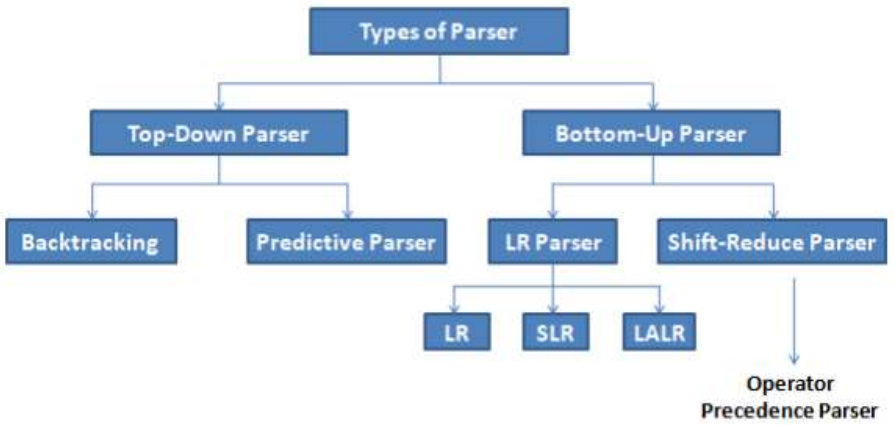
The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.



There are three general types of parsers for grammar's. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. But these methods are inefficient to use in production compilers. The Efficient methods commonly used in compilers are as follows:-



2.4.1.1 Syntax Error Handling:-

Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

Programs can contain errors at many different levels as follows:-

a) Lexical, such as misspelling an identifier, keyword or an operator.

b) Syntactic, such as an arithmetic expression with unbalanced parenthesis.

c) Semantic, such as an operator applied to an incompatible operand.

d) Logical, such as an infinitely recursive call.

For recovery from syntax errors, the error handler in a parser has simple-to-state goals:-

a) It should report the presence of errors clearly and accurately.

b) It should recover from each error quickly enough to be able to detect subsequent errors.

c) It should not significantly slow down the processing of correct programs.

2.4.1.2 Error – Recovery Strategies:-

There are many different general strategies that a parser can employ to recover from syntactic error. Here are few methods listed down which have broad applicability. :-

a) Panic mode – Simplest and adequate method and panic mode recovery does not work in an infinite loop.

b) Phrase level – local correction, one must be careful to choose replacements that do not lead to infinite loops. Difficulty in coping with situations in which the actual error has occurred before the point of detection.

c) Error productions – One can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

d) Global corrections – Too costly to implement in terms of time and space.

2.4.2 CONTEXT FREE GRAMMARS: -

A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.

2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.

3. In a grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.

4. The productions of a grammar specify the way the terminals and non-terminals can be combined to form strings. Each production consists of a non terminal, followed by an arrow, followed by a string of non-terminals and terminals.

   Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S). Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic- variables).P is a finite set of productions rules in the following form A → α where A is a non- terminal and α is a string of terminals and non-terminals (including the empty string).S is a start symbol (one of the non-terminal symbol).

   L(G) is the language of G (the language generated by G) which is a set of sentences. A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) if S ω where ω is a string of terminals of G. If G is a context-free grammar (G) is a context-free language. Two grammars G1 and G2 are equivalent, if they produce same grammar.

   Consider the production of the form S α, if α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

   Example: Consider the grammar for simple arithmetic expressions:

   expr → expr op expr
   expr → (expr)
   expr → - expr
   expr → id
   op → +
   op → -
   op → *
   op → /
   op → ^
   Terminals : id + - * / ^ ( )
   Non-terminals : expr , op
   Start symbol : expr

2.4.2.1 Notational Conventions:

1. These symbols are terminals:
   i. Lower-case letters early in the alphabet such as a, b, c.
   ii. Operator symbols such as +, -, etc.
   iii. Punctuation symbols such as parentheses, comma etc.
   iv. Digits 0,1,…,9.
   v. Boldface strings such as id or if (keywords)
2. These symbols are non-terminals:
   i. Upper-case letters early in the alphabet such as A, B, C..
   ii. The letter S, when it appears is usually the start symbol.
   iii. Lower-case italic names such as expr or stmt.
3. Upper-case letters late in the alphabet, such as X,Y,Z, represent grammar symbols, that is either terminals or non-terminals.
4. Greek letters α , β , γ represent strings of grammar symbols.

   e.g., a generic production could be written as A → α.
5. If A → α1 , A → α2 , ... , A → αn are all productions with A , then we can write A

   → α1 | α2 |. . . . | αn , (alternatives for A).
6. Unless otherwise stated, the left side of the first production is the start symbol.

   Using the shorthand, the grammar can be written as:

   E → E A E | (E) | - E | id

   A → + | - | * | / | ^

2.4.2.2 Derivations:

A derivation of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

$\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in $\geq$ one step

$\overset{*}{\underset{G}{\Rightarrow}}$ indicates that the derivation utilizes the rules of grammar G

2.4.3.2.1. To create a string from a context-free grammar:

- Begin the string with a start symbol.
- Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

In general a derivation step is αAβ αγβ is sentential form and if there is a production rule A→γ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols α1 α2... αn (αn derives from α1 or α1 derives αn). There are two types of derivation:

1. **Leftmost Derivation (LMD):**
    - If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.
    - The sentential form derived by the left-most derivation is called the left-sentential form.
2. **Rightmost Derivation (RMD):**
    - If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
    - The sentential form derived from the right-most derivation is called the right sentential form.

      Example:

      Consider the G,

      E → E + E | E * E | (E) | - E | id

Derive the string id + id * id using leftmost derivation and rightmost derivation.

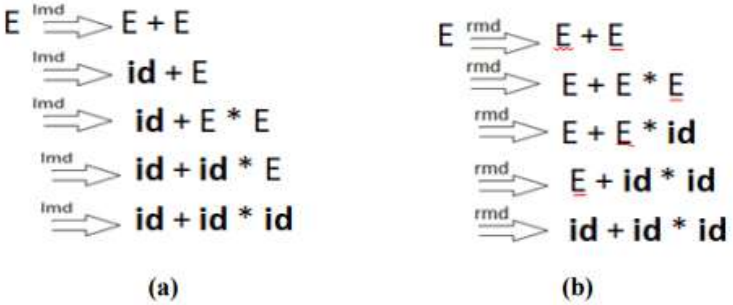


Fig 2.2 a) Leftmost derivation b) Rightmost derivation

Strings that appear in leftmost derivation are called left sentential forms. Strings that appear in rightmost derivation are called right sentential forms.

Sentential Forms:

Given a grammar G with start symbol S, if S => α , where α may contain non-terminals or terminals, then α is called the sentential form of G.

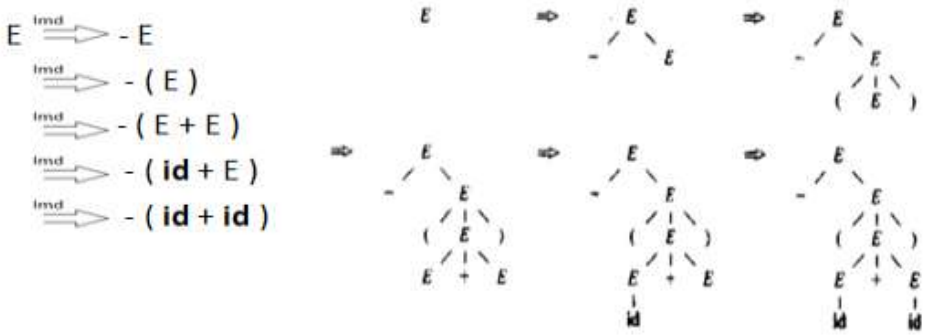
2.4.2.2.2 Parse Tree:

A parse tree is a graphical representation of a derivation sequence of a sentential form.

In a parse tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.



Yield or frontier of tree:

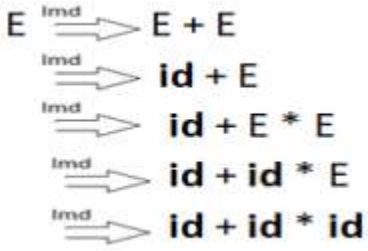
Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentential forms that are read from left to right. The sentential form in the parse tree is called yield or frontier of the tree.

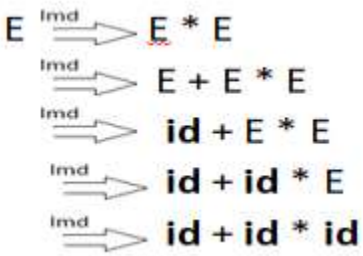
Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar. i.e. An ambiguous grammar is one that produce more than one leftmost or more than one rightmost derivation for the same sentence.

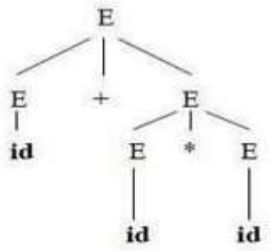
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

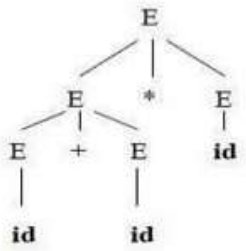
The sentence id+id*id has the following two distinct leftmost derivations:





The two corresponding parse trees are:-



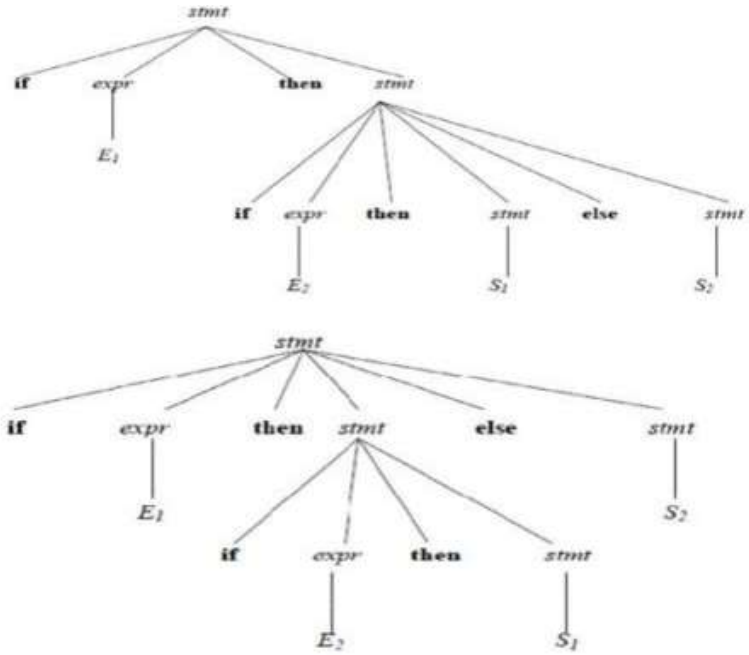


Consider another example –

stmt → if expr then stmt | if expr then stmt else stmt | other

This grammar is ambiguous since the string if E1 then if E2 then S1 else S2 has the following

Two parse trees for leftmost derivation :



Eliminating Ambiguity:

An ambiguous grammar can be rewritten to eliminate the ambiguity. e.g. Eliminate the ambiguity from "dangling-else" grammar,

stmt → if expr then stmt

| if expr then stmt else stmt

| other

Match each else with the closest previous unmatched then. This disambiguity rule can be incorporated into the grammar.

stmt → matched_stmt | unmatched_stmt

matched_stmt →if expr then matched_stmt else matched_stmt

| other

unmatched_stmt → if expr then stmt

| if expr then matched_stmt else unmatched_stmt

This grammar generates the same set of strings, but allows only one parsing for string.

Removing Ambiguity by Precedence & Associativity Rules:

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

– Precedence Constraints

– Associativity Constraints

These constraints are implemented using the following rules:

Rule-1:

• The level at which the production is present defines the priority of the operator

contained in it.

– The higher the level of the production, the lower the priority of operator.

– The lower the level of the production, the higher the priority of operator.

Rule-2:

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

Example: Consider the ambiguous Grammar:

E → E + E |E – E | E * E | E / E | (E) | id

Introduce new variable / non-terminals at each level of precedence,

- an expression E for our example is a sum of one or more terms. (+,-)
- a term T is a product of one or more factors. (*, /)
- a factor F is an identifier or parenthesised expression.

10

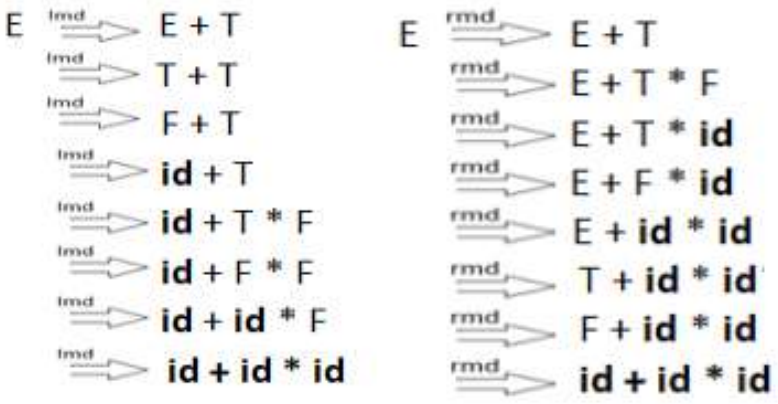
The resultant unambiguous grammar is:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$

Trying to derive the string id+id*id using the above grammar will yield one unique derivation.



Regular Expression vs. Context Free Grammar:

- Every construct that can be described by a regular expression can be described by a grammar.
- NFA can be converted to a grammar that generates the same language as recognized by the NFA.
- **Rules:**
- For each state i of the NFA, create a non-terminal symbol Ai .
- If state i has a transition to state j on symbol a, introduce the production Ai →a Aj
- If state i goes to state j on symbol ε, introduce the production Ai → Aj
- If i is an accepting state, introduce Ai → ε
- If i is the start state make Ai the start symbol of the grammar.

Example: The regular expression (a|b)*abb, consider the NFA

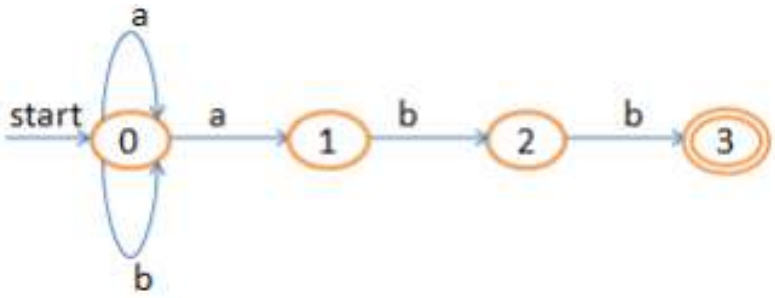


Fig 2.7 NFA for (a|b)*abb

Equivalent grammar is given by:

A0 → a A0 | b A0 | a A1

A1 → b A2

A2 → b A3

A3 → ε

2.4.2.2 LR Parsing:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

Why LR parsing:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to right scan of the input.
- The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

2.4.2.3 Bottom-Up Parsing:

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

2.4.2.3.1 Shift-Reduce Parsing:

Shift-reduce parsing is a type of bottom -up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The string to be recognized is abbcde. We want to reduce the string to S.

Steps of reduction:

Abbcde (b,d can be reduced)

aAbcde (leftmost b is reduced)

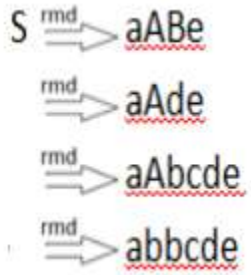
aAde (now Abc,b,d qualified for reduction)

aABe (d can be reduced)

S

Each replacement of the right side of a production by the left side in the above example is

called reduction, which is equivalent to rightmost derivation in reverse.



Handle:

A substring which is the right side of a production such that replacement of that substring by

the production left side leads eventually to a reduction to the start symbol, by the reverse of a

rightmost derivation is called a handle.

2.4.2.4 Stack Implementation of Shift-Reduce Parsing:

There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	w$

The parser operates by shifting zero or more input symbols onto the stack until a handle is on top of the stack. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Example: The actions a shift-reduce parser in parsing the input string id1+id2*id3, according to the ambiguous grammar for arithmetic expression.

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$	shift
(2)	id_1	$+ id_2 * id_3$	reduce by $E \rightarrow id$
(3)	E	$+ id_2 * id_3$	shift
(4)	$E +	$id_2 * id_3$	shift
(5)	$E + id_2$	$* id_3$	reduce by $E \rightarrow id$
(6)	$E + E$	$* id_3$	shift
(7)	$E + E *	id_3	shift
(8)	$E + E * id_3$	$	reduce by $E \rightarrow id$
(9)	$E + E * E$	$	reduce by $E \rightarrow E * E$
(10)	$E + E$	$	reduce by $E \rightarrow E + E$
(11)	E	$	accept

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

In above fig. Reductions made by Shift Reduce Parser.

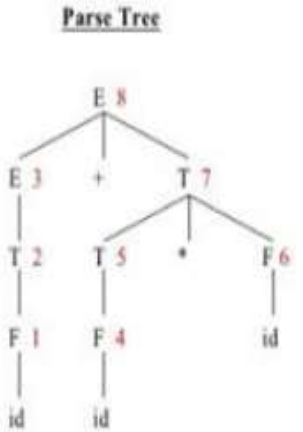
While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

(1) shift, (2) reduce,(3) accept, and (4) error.

- In a shift action, the next input symbol is shifted unto the top of the stack.
- In a reduce action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
- In an accept action, the parser announces successful completion of parsing.
- In an error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Figure below represents the stack implementation of shift reduce parser using unambiguous grammar.

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept



2.4.2.5 Operator Precedence Parsing:

Operator grammars have the property that no production right side is ε (empty) or has two adjacent non terminals. This property enables the implementation of efficient operator precedence parsers.

Example: The following grammar for expressions:

E→E A E | (E) | -E | id

A→ + | - | * | / | ^

This is not an operator grammar, because the right side EAE has two consecutive nonterminals. However, if we substitute for A each of its alternate, we obtain the following

operator grammar:

E→E + E |E – E |E * E | E / E | (E) | E ^ E | - E | id

In operator-precedence parsing, we define three disjoint precedence relations between pair of terminals. This parser relies on the following three precedence relations.

Relation	Meaning
$a <\cdot b$	*a* yields precedence to *b*
$a \doteq b$	*a* has the same precedence as *b*
$a \cdot> b$	*a* takes precedence over *b*

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Above figure - Operator Precedence Relation Table.

Example: The input string: id1 + id2 * id3

After inserting precedence relations the string becomes:

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows identifying handles as follows:

1. Scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any =’s until a <· is encountered.
3. Everything between the two relations <· and ·> forms the handle.

Stack	Rule	Input	Comments
$ < id > + < id > * < id > $	E → id	$ id + id * id $	Here the first "id" is looked as the handle and since we were able to reduce, we reduce it in the input
$ < + < id > * < id > $	E → id	$ E + id * id $	The second handle is also "id" since that is available between a pair of lesser than and greater than precedences
$ < + < * < id > $	E → id	$ E + E * id $	The third handle is also "id".
$ < + < * > $	E → E*E	$ E + E * E $	The fourth handle is E*E, and is popped in the stack and we push the greater than symbol.
$ < + > $	E → E+E	$ E + E $	The last handle is E+E and that is also reduced.
$$			The stack is empty and has only the $ symbol, we say the string is accepted.

Defining Precedence Relations:

The precedence relations are defined using the following rules:

Rule-01:

- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

1. ↑ is of highest precedence and right-associative,
2. * and / are of next highest precedence and left-associative, and
3. + and − are of lowest precedence and left-associative.

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Above fig. – Operator Precedence Relation Table

STACK		INPUT	COMMENT
$	<·	id+id*id $	shift id
$ id	·>	+id*id $	pop the top of the stack id
$	<·	+id*id $	shift +
$ +	<·	id*id $	shift id
$ +id	·>	*id $	pop id
$ +	<·	*id $	shift *
$ + *	<·	id $	shift id
$ + * id	·>	$	pop id
$ + *	·>	$	pop *
$ +	·>	$	pop +
$		$	accept

Above fig. – Stack Implementation

Implementation of Operator-Precedence Parser:

- An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
- More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Steps involved in Parsing:

1. Ensure the grammar satisfies the pre-requisite.
2. Computation of the function LEADING()
3. Computation of the function TRAILING()
4. Using the computed leading and trailing ,construct the operator Precedence Table
5. Parse the given input string based on the algorithm
6. Compute Precedence Function and graph.

Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.

- LEADING(A)={ a| A=>+

γaδ },where γ is ε or any non-terminal, =>+ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for LEADING(A):

{

1. 'a' is in LEADING(A) is A→ γaδ where γ is ε or any non-terminal.

2.If 'a' is in LEADING(B) and A→B, then 'a' is in LEADING(A).

}

Computation of TRAILING:

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- TRAILING(A)={ a| A=>+

γaδ },where δ is ε or any non-terminal, =>+ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for TRAILING(A):

{

1. 'a' is in TRAILING(A) is A→ γaδ where δ is ε or any non-terminal.

2.If 'a' is in TRAILING(B) and A→B, then 'a' is in TRAILING(A).

}

Example 1: Consider the unambiguous grammar,

E→E + T

E→T

T→T * F

T→F

F→(E)

F→id

Step 1: Compute LEADING and TRAILING:

LEADING(E)= { +,LEADING(T)} ={+ , * , (, id}

LEADING(T)= { *,LEADING(F)} ={* , (, id}

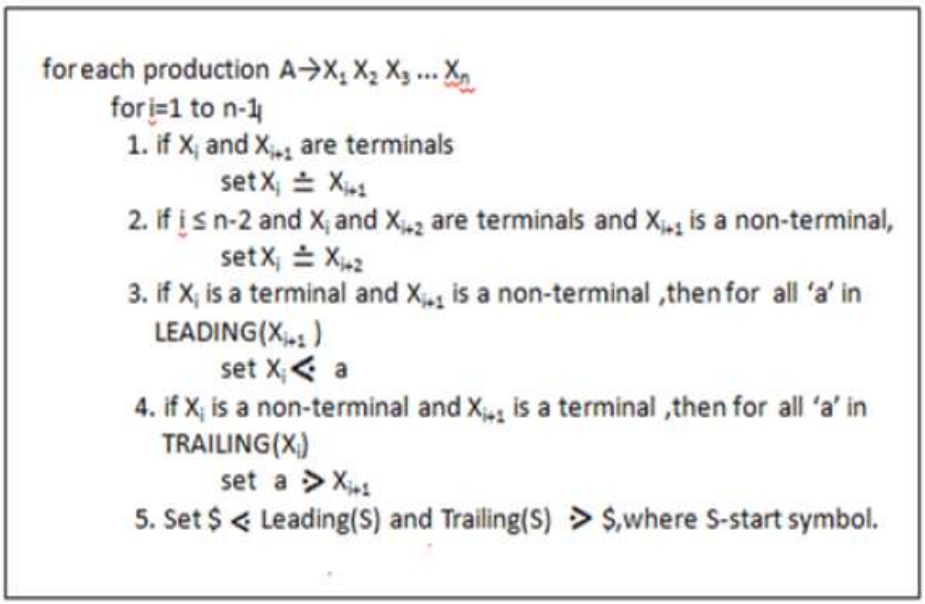
LEADING(F)= { (, id}

TRAILING(E)= { +, TRAILING(T)} ={+ , * ,) , id}

TRAILING(T)= { *, TRAILING(F)} ={* ,) , id}

TRAILING(F)= {) , id}

Step 2: After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the '$' symbol.



Above fig. – Algorithm for constructing Precedence Relation Table.

	+	*	id	(	)	$
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
$	<	<	<	<	e	Accept

Above fig. – Precedence Relation Table.

Step 3: Parse the given input string (id+id)*id$

```
Set ip to point to the first symbol of w$
Repeat forever
    if $ is on the top of the stack and ip points to $ then return
    else begin
        Let a be the top terminal on the stack, and b the symbol pointed to by ip
            if a <· b or a =· b then
                push b onto the stack
                advance ip to the next input symbol
            end
            else if a ·> b then
                repeat
                    pop the stack
                until the top stack terminal is related by <·
                    to the terminal most recently popped
            else error()
    end
```

Above fig. – Parsing Algorithm

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(<+	+id)*id$	Shift +
$(+	+ <id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* <id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Above fig. - Parse the input string(id+id)*id$

2.4.2.6 Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. f (a) < g(b) whenever a<·b.

2. f (a) = g(b) whenever a = b. and

3. f(a) > g(b) whenever a ·> b.

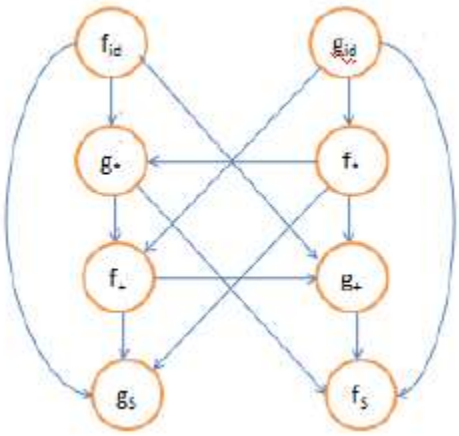
Algorithm for Constructing Precedence Functions:

1. Create functions fa for each grammar terminal a and for the end of string symbol.

2. Partition the symbols in groups so that fa and gb are in the same group if a = b (there can be symbols in the same group even if they are not connected by this relation).

3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if a <· b, otherwise if a ·> b place an edge from the group of fa to that of gb.

4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb respectively.



Precedence Graph.

There are no cycles,so precedence function exist. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1,so g(+)=1.There is a path from gid to f* to g* to f+ to g+ to f$,so g(id)=5.The resulting precedence functions are:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Example 2:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R

S→R

L→*R

L→id

R→L

Solution:

1. Computation of LEADING:

   LEADING(S) = {=, * , id}

   LEADING(L) = {* , id}

   LEADING(R) = {* , id}

2. Computation of TRAILING:

   TRAILING(S) = {= , * , id}

   TRAILING(L)= {* , id}

   TRAILING(R)= {* , id}

3. Precedence Table:

	=	*	id	$
=	e	<·	<·	·>
*	·>	<·	<·	·>
id	·>	e	e	·>
$	<·	<·	<·	accept

* All undefined entries are error (e).

4. Parsing the given input string:

   1. *id = id

STACK	INPUT STRING	ACTION
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

   2. id*id=id

STACK	INPUT STRING	ACTION
$	id*id=id$	$<·idPush
$id	*id=id$	Error

2.4.3 Top-Down Parsing- Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

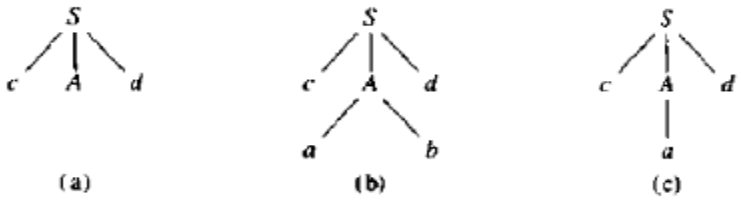
A general form top-down parsing called recursive descent parsing, involves backtracking, that is making repeated scans of the input. A special case of recursive descent parsing called predictive parsing, where no backtracking is required.

Consider the grammar

S → cAd

A → ab | a

and the input string w=cad. Construction of parse is shown in fig below:-



Above fig. – Steps in Top-Down Parse.

The leftmost leaf, labeled c, matches the first symbol of w, hence advance the input pointer to a, the second symbol of w. Fig 2.21(b) and (c) shows the backtracking required to match the input string.

Predictive Parser:

A grammar after eliminating left recursion and left factoring can be parsed by a recursive descent parser that needs no backtracking is a called a predictive parser. Let us understand how to eliminate left recursion and left factoring.

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA'

A' → αA' | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT '

T'→ *FT ' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

25

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order A1, A2 . . . An.

2. for i := 1 to n do begin

for j := 1 to i-1 do begin

replace each production of the form Ai → Aj γ

by the productions Ai → δ1 γ | δ2γ | . . . | δk γ.

where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;

end

eliminate the immediate left recursion among the Ai- productions

end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → αβ1 | αβ2 , it can be rewritten as

A → αA'

A’ → αβ1 | αβ2

Consider the grammar,

S → iEtS | iEtSeS | a

E → b

Here,i,t,e stand for if ,the,and else and E and S for “expression” and “statement”.

After Left factored, the grammar becomes

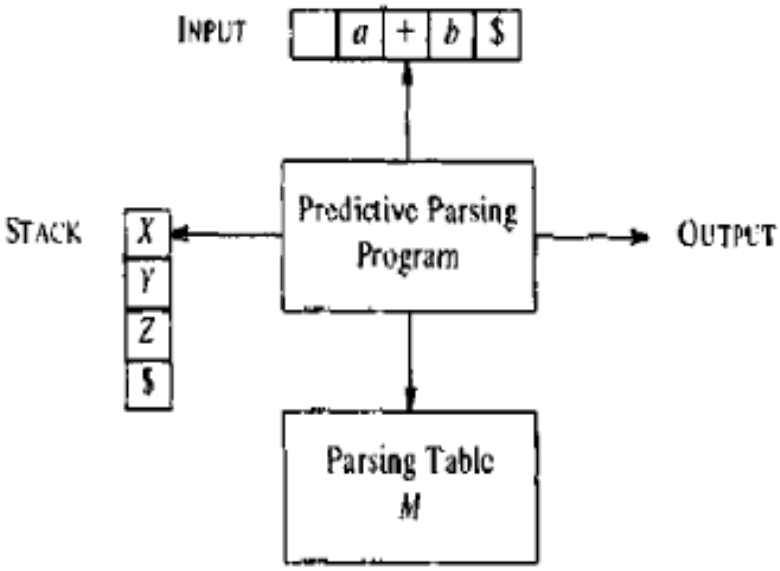
S → iEtSS' | a

S' → eS | ε

E → b

Non-recursive Predictive Parsing:

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of 26 determining the production to be applied for a non-terminal.



Above fig. - Model of a Non-recursive predictive parser.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of S. The parsing table is a two-dimensional array M[A,a],where A is a non-terminal, and a is a terminal or the symbol $.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If X = a =$,the parser halts and announces successful completion of parsing.

2. If X =a ≠$, the parser pops X off the stack and advances the input pointer to the next input symbol.

3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a] = {X→UVW}, the parser replaces X on top of the stack by WVU (with U on top). If M[X, a] = error, the parser calls an error recovery routine.

2.4.3.1 Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar

G .These functions are FIRST and FOLLOW.

Rules for FIRST():

1. If X is terminal, then FIRST(X) is {X}.

2. If X → ε is a production, then add ε to FIRST(X).

3. If X is non-terminal and X → aα is a production then add a to FIRST(X).

4. If X is non-terminal and X → Y 1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-1 => ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW():

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε,then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production A → α of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A, a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].
4. Make each undefined entry of M be error.

Algorithm : Non-recursive predictive parsing.

Input: A string w and a parsing table M for grammar G.

Output: If w is in L(G), a leftmost derivation of w; otherwise, an error .

Method: Initially, the parser is in a configuration in which it has $$ on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

set ip to point to the first symbol of w$:

repeat

let X be the top stack symbol and a the symbol pointed to by ip;

if X is a terminal or $ then

if X = a then

pop X from the stack and advance ip

else error()

else /* X is a non-terminal */

if M[X,a] =X→Y1Y2…….Yk, then

begin

pop X from the stack:

push Yk , Yk - 1 Y1 , onto the stack, with Y1 on top;

output the production X→Y1Y2 …..Yk

end

else error()

until X ≠$ /* stack is empty*/

LL(1) Grammars:

For some grammars the parsing table may have some entries that are multiply-defined. For example, if G is left recursive or ambiguous , then the table will have at least one multiply-defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

Example: Consider this following grammar:

S→ iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S→ iEtSS’ | a S’→ eS | ε

E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals. FIRST(S) ={ i, a }

FIRST(S’) = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = { $,e }

FOLLOW(E) = {t}

Parsing Table for the grammar:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.

2.4.4 LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions.. When (k) is omitted, it is assumed to be 1.

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

2.4.4.1 Types of LR parsing method:

1. SLR- Simple LR
    - Easiest to implement, least powerful.
2. CLR- Canonical LR

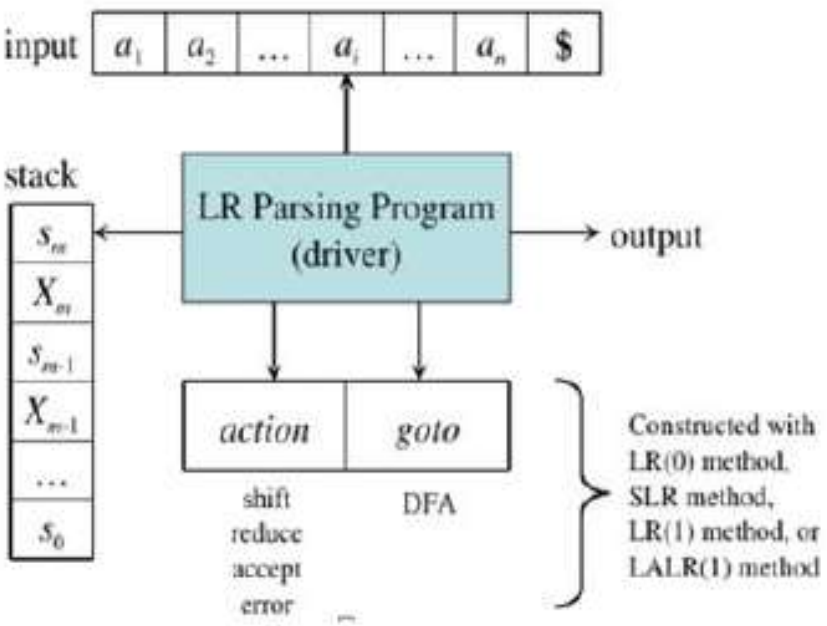
- Most powerful, most expensive.

3. LALR- Look -Ahead LR

- Intermediate in size and cost between the other two methods.

2.4.4.2 The LR Parsing Algorithm:

The schematic form of an LR parser is shown in Fig 2.25. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto).The driver program is the same for all LR parser. The parsing table alone changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form s0X1s1X2s2…… Xmsm , where sm is on top. Each Xi is a grammar symbol and each si is a symbol called a state.



Above fig. – Model of an LR Parser.

The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults action[sm,ai] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

2.4.5 CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

   LR(0) items:

   An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

   A → •XYZ

   A → X•YZ

   A → XY•Z

   A → XYZ•

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

   Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX•β] such that [A→ α•Xβ] is in I.Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G`
2. Construct the canonical collection of set of items C for G‟
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

2.4.5.1 Algorithm for construction of SLR parsing table:

Input : An augmented grammar G‟

Output : The SLR parsing table functions action and goto for G’

Method :

1. Construct C ={I0, I1, …. In}, the collection of sets of LR(0) items for G'.

2. State i is constructed from Ii. The parsing functions for state i are determined as follows:

   (a) If [A→α•aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.

   (b) If[A→α•] is in Ii , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).

   (c) If [S‟→S•] is in Ii, then set action[i,$] to "accept".

   If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all non-terminals A using the rule: If

   goto(Ii,A)= Ij, then goto[i,A] = j.

4. All entries not defined by rules (2) and (3) are made "error"

5. The initial state of the parser is the one constructed from the set of items containing [S'→•S].

2.4.5.2 SLR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G.

Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer. The parser then executes the following program:

set ip to point to the first input symbol of w$;

repeat forever begin

let s be the state on top of the stack and a the symbol pointed to by ip;

if action[s, a] =shift s‟ then begin

push a then s‟ on top of the stack;

advance ip to the next input symbol

end

else if action[s, a]=reduce A→β then begin

pop 2* |β |symbols off the stack;

let s‟ be the state now on top of the stack;

push A then goto[s‟, A] on top of the stack;

output the production A→ β

end

else if action[s, a]=accept then

return

else error( )

end

Example: Implement SLR Parser for the given grammar:

1.E→E + T

2.E→T

3.T→T * F

4.T→F

5.F→(E)

6.F→id

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

E'→E

35

E→E + T

E→T

T→T * F

T→F

F→(E)

F→id

Step 2 : Find LR (0) items.

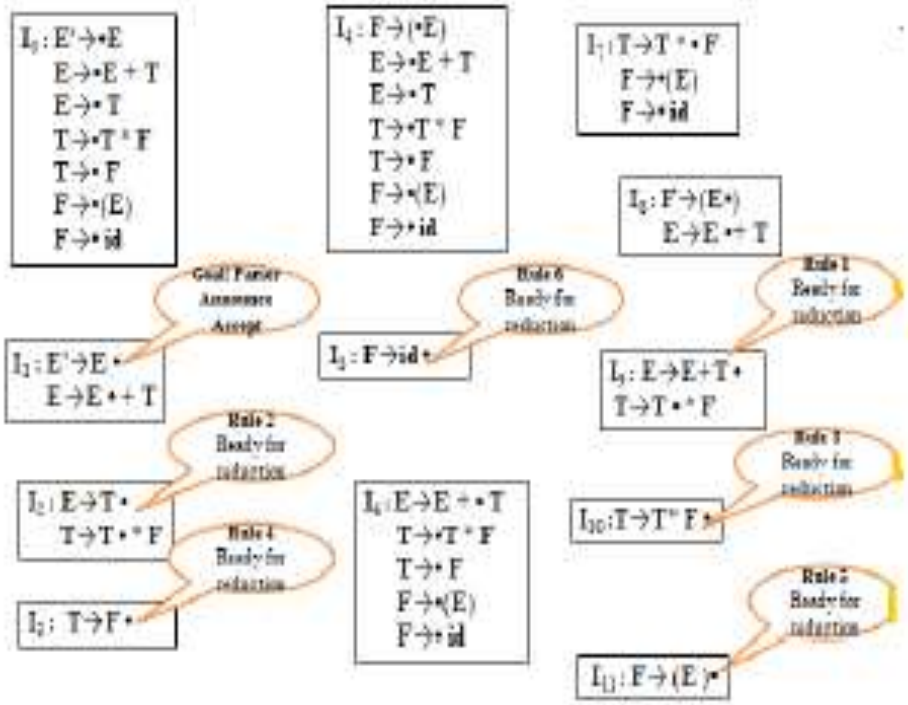
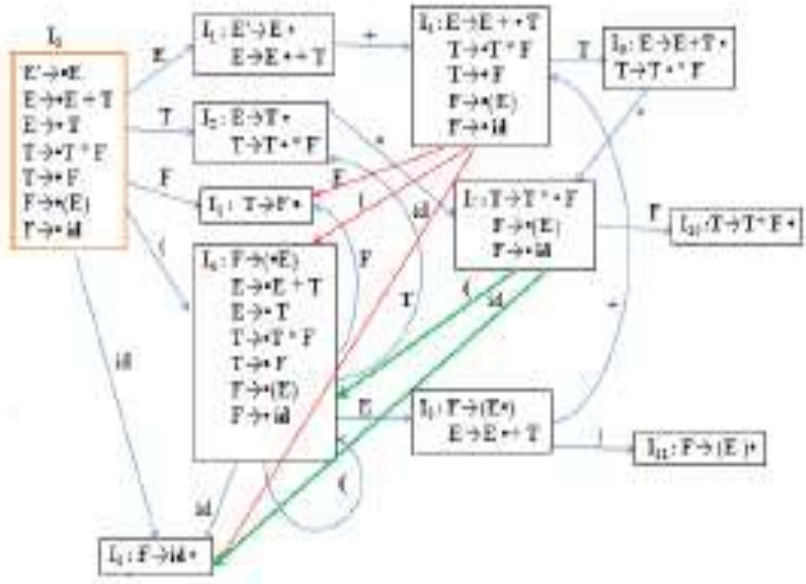


Fig 2.26 Canonical LR(0) collections



Step 3 : Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

FOLLOW(E) = {+,),$ }

FOLLOW(T) ={*,+,) ,$}

FOLLOW(F) ={*,+,) ,$}

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

1. si means shift and stack state i.

2. rj means reduce by production numbered j.

3. acc means accept.

4. Blank means error.

Step 4: Parse the given input. The Fig below shows the parsing the string id*id+id using stack implementation

Stack	Input	Action
0	id*id+id$	s5 Shift 5
0 id 5	*id+id$	r6 Reduce by F→id
0 F 3	*id+id$	r4 Reduce by T→F
0 T 2	*id+id$	s7 Shift 7
0 T 2 * 7	id+id$	s5 Shift 5
0 T 2 * 7 id 5	+id$	r6 Reduce by F→id
0 T 2 * 7 F 10	+id$	r3 Reduce by T→T*F
0 T 2	+id$	r2 Reduce by E→T
0 E 1	+id$	s6 Shift 6
0 E 1 + 6	id$	s5 Shift 5
0 E 1 + 6 id 5	$	r6 Reduce by F→id
0 E 1 + 6 F 3	$	r4 Reduce by T→F
0 E 1 + 6 T 9	$	r1 Reduce by E→E+T
0 E 1	$	Accept

2.4.6 Constructing Canonical or LR(1) parsing tables

LR or canonical LR parsing incorporates the required extra information into the state by redefining configurations to include a terminal symbol as an added component. LR(1) configurations have the general form:

A –> X1...Xi

- Xi+1...Xj , a

This means we have states corresponding to X1...Xi on the stack and we are looking to put states corresponding to Xi+1...Xj on the stack and then reduce, but only if the token following Xj is the terminal a. a is called the lookahead of the configuration. The lookahead only comes into play with LR(1) configurations with a dot at the right end:

A –> X1…Xj •, a

This means we have states corresponding to X1...Xj on the stack but we may only reduce when the next symbol is a. The symbol a is either a terminal or $ (end of input marker).

With SLR(1) parsing, we would reduce if the next token was any of those in Follow(A).

With LR(1) parsing, we reduce only if the next token is exactly a. We may have more than one symbol in the lookahead for the configuration, as a convenience, we list those symbols separated by a forward slash. Thus, the configuration A –> u•, a/b/c says that it is valid to reduce u to A only if the next token is equal to a, b, or c. The configuration lookahead will always be a subset of Follow(A).

Recall the definition of a viable prefix from the previous handout. Viable prefixes are those prefixes of right sentential forms that can appear on the stack of a shift-reduce parser. Formally we say that a configuration [A –> u•v, a] is valid for a viable prefix α if there is a rightmost derivation S =>* βAw =>* βuvw where α = βu and either a is the first symbol of w or w is ∂ and a is $.

For example:

S –> ZZ

Z –> xZ | y

There is a rightmost derivation S =>* xxZxy => xxxZxy. We see that configuration

[Z –> x•Z, x] is valid for viable prefix α = xxx by letting β = xx, A = Z, w = xy, u = x and

v = Z. Another example is from the rightmost derivation S =>* ZxZ => ZxxZ, making

[Z –> x•Z, $] valid for viable prefix Zxx.

Often we have a number of LR(1) configurations that differ only in their lookahead components. The addition of a lookahead component to LR(1) configurations allows us to make parsing decisions beyond the capability of SLR(1) parsers. There is, however, a big price to be paid. There will be more distinct configurations and thus many more possible configurating sets. This increases the size of the goto and action tables considerably. In the past when memory was smaller, it was difficult to find storageefficient ways of representing these tables, but now this is not as much of an issue. Still, it's a big job building LR tables for any substantial grammar by hand.

The method for constructing the configurating sets of LR(1) configurations is essentially the same as for SLR, but there are some changes in the closure and successor operations because we must respect the configuration lookahead. To compute the closure of an LR(1) configurating set I:

Repeat the following until no more configurations can be added to state I:

— For each configuration [A –> u•Bv, a] in I, for each production B –> w in G', and for each terminal b in First(va) such that [B –> •w, b] is not in I: add [B –> •w, b] to I.

What does this mean? We have a configuration with the dot before the non-terminal B.

In LR(0), we computed the closure by adding all B productions with no indication of what was expected to follow them. In LR(1), we are a little more precise— we add each B production but insist that each have a lookahead of va. The lookahead will be First(va) since this is what follows B in this production. Remember that we can compute first sets not just for a single non-terminal, but also a sequence of terminal and non-terminals.

First(va) includes the first set of the first symbol of v and then if that symbol is nullable, we include the first set of the following symbol, and so on. If the entire sequence v is nullable, we add the lookahead a already required by this configuration.

The successor function for the configurating set I and symbol X is computed as this:

Let J be the configurating set [A –> uX•v, a] such that [A –> u•Xv, a] is in I.

successor(I,X) is the closure of configurating set J.

We take each production in a configurating set, move the dot over a symbol and close on the resulting production. This is basically the same successor function as defined for LR(0), but we have to propagate the lookahead when computing the transitions.

We construct the complete family of all configurating sets F just as we did before. F is initialized to the set with the closure of [S' –> S, $]. For each configurating set I and each grammar symbol X such that successor(I,X) is not empty and not in F, add successor (I,X) to F until no other configurating set can be added to F.

LR(1) grammars

Every SLR(1) grammar is a canonical LR(1) grammar, but the canonical LR(1) parser may have more states than the SLR(1) parser. An LR(1) grammar is not necessarily SLR(1), the grammar given earlier is an example. Because an LR(1) parser splits states based on differing lookaheads, it may avoid conflicts that would otherwise result if using the full follow set.

A grammar is LR(1) if the following two conditions are satisfied for each configurating set:

1. For any item in the set [A –> u•xv, a] with x a terminal, there is no item in the set of the form [B –> v•, x]. In the action table, this translates no shift-reduce conflict for any state. The successor function for x either shifts to a new state or reduces, but not both.

2. The lookaheads for all complete items within the set must be disjoint, e.g. set cannot have both [A –> u•, a] and [B –> v•, a] This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which is appropriate from the next input token.

   As long as there is a unique shift or reduce action on each input symbol from each state, we can parse using an LR(1) algorithm. The above state conditions are similar to what is required for SLR(1), but rather than the looser constraint about disjoint follow sets and so on, canonical LR(1) computes a more precise notion of the appropriate lookahead within a particular context and thus is able to resolve conflicts that SLR(1) would encounter.

2.4.7 LALR Table Construction

A LALR(1) parsing table is built from the configurating sets in the same way as canonical LR(1); the lookaheads determine where to place reduce actions. In fact, if there are no mergable states in the configuring sets, the LALR(1) table will be identical to the corresponding LR(1) table and we gain nothing.

In the common case, however, there will be states that can be merged and the LALR table will have fewer rows than LR. The LR table for a typical programming language may have several thousand rows, which can be merged into just a few hundred for LALR. Due to merging, the LALR(1) table seems more similar to the SLR(1) and LR(0) tables, all three have the same number of states (rows), but the LALR may have fewer reduce actions—some reductions are not valid if we are more precise about the lookahead. Thus, some conflicts are avoided because an action cell with conflicting actions in SLR(1) or LR(0) table may have a unique entry in an LALR(1) once some erroneous reduce actions have been eliminated.

Brute Force?

There are two ways to construct LALR(1) parsing tables. The first (and certainly more obvious way) is to construct the LR(1) table and merge the sets manually. This is sometimes referred as the "brute force" way. If you don't mind first finding all the multitude of states required by the canonical parser, compressing the LR table into the LALR version is straightforward.

1. Construct all canonical LR(1) states.

2. Merge those states that are identical if the lookaheads are ignored, i.e., two states being merged must have the same number of items and the items have the same core (i.e., the same productions, differing only in lookahead). The lookahead on merged items is the union of the lookahead from the states being merged.

3. The successor function for the new LALR(1) state is the union of the successors of the merged states. If the two configurations have the same core, then the original successors must have the same core as well, and thus the new state has the same successors.

4. The action and goto entries are constructed from the LALR(1) states as for the canonical LR(1) parser. Consider the LR(1) table for the grammar given on page 1 of this handout. There are nine states.

State on top of stack	Action			Goto	
	a	b	$	S	X
0	s3	s4		1	2
1			Acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Looking at the configurating sets, we saw that states 3 and 6 can be merged, so can 4 and 7, and 8 and 9. Now we build this LALR(1) table with the six remaining states:

State on top of stack	Action			Goto	
	a	b	$	S	X
0	S36	s47		1	2
1			acc		
2	S36	s47			5
36	S36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Having to compute the LR(1) configurating sets first means we won't save any time or effort in building an LALR parser. However, the work wasn't all for naught, because when the parser is executing, it can work with the compressed table, thereby saving memory. The difference can be an order of magnitude in the number of states.

However there is a more efficient strategy for building the LALR(1) states called step-by-step merging. The idea is that you merge the configurating sets as you go, rather than waiting until the end to find the identical ones. Sets of states are constructed as in the LR(1) method, but at each point where a new set is spawned, you first check to see 6 whether it may be merged with an existing set. This means examining the other states to see if one with the same core already exists. If so, you merge the new set with the existing one, otherwise you add it normally.

Here is an example of this method in action:

S' –> S

S –> V = E

E –> F | E + F

F –> V | int | (E)

V –> id

Start building the LR(1) collection of configurating sets as you would normally:

I0: S' –> •S, $

S –> •V = E, $

V –> •id, =

I1: S' –> S•, $

I2: S' –> V• = E, $

I3: V –> id•, =

I4: S –> V =•E, $

E –> •F, $/+

E –> •E + F, $/+

F –>•V, $/+

F –>•int, $/+

F –>•(E), $/+

V –>•id, $/+

I5: S –> V = E•, $

E –> E• + F, $/+

I6: E –> F•, $/+

I7: F–> V•, $/+

I8: F–> int•, $/+

I9: F–> (•E), $/+

E –> •F,)/+

E –> •E + F,)/+

F –> •V,)/+

F –> •int,)/+

F –> •(E),)/+

V –> •id)/+

I10: F–> (E•), $/+

E –> E• + F,)/+

When we construct state I11, we get something we've seen before:

I11: E –>F•,)/+

It has the same core as I6, so rather than add a new state, we go ahead and merge with that one to get:

I611: E –>F•, $/+/)

We have a similar situation on state I12 which can be merged with state I7. The algorithm continues like this, merging into existing states where possible and only adding new states when necessary. When we finish creating the sets, we construct the table just as in LR(1).

2.4.8 An automatic parser generator

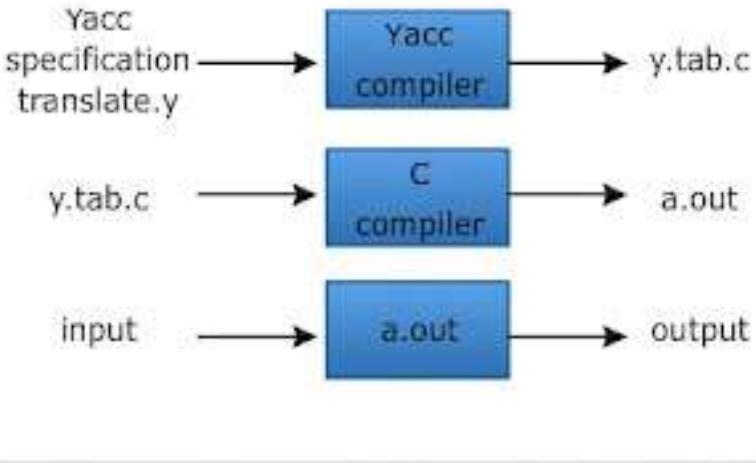
A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a parser, which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an abstract syntax tree (AST).

Antlr is a mature and widely-used parser generator for Java, and other languages as well.

Example tool for parser generator is YACC:



Creating an Input/Output Translator with YACC

YACC is a automatic tool that generates the parser program

YACC stands for Yet Another Compiler Compiler. This program is available in UNIX OS The construction of LR parser requires lot of work for parsing the input string. Hence, the process must involve automation to achieve efficiency in parsing an input.

Basically YACC is a LALR parser generator that reports conflicts or uncertainties (if at all present) in the form of error messages.

The typical YACC translator can be represented as shown in the image

2.5 SUMMARY

We have studied the below concepts:

1) Parsing methods used in compilers.

2) Basic concepts.

3) Techniques used in Efficient Parsers.

4) Algorithms – to recover from commonly occurring errors.

2.6 REFERENCE FOR FURTHER READING

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA:

Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

❄❄❄❄❄❄❄

3

ADVANCED SYNTAX ANALYSIS AND BASIC SEMANTIC ANALYSIS

Unit Structure

3.0 OBJECTIVES

The aim of this chapter is to explain the role of the syntax analysis and to introduce the important techniques used in the syntax analysis and semantic analysis. After going through this unit, you will be able to understand:

- Syntax directed definitions
- Syntax directed translations
- SDT schemes
- Introduction to Tiger compiler
- Bindings for Tiger compiler
- Type-checking expressions, declarations
- Activation records and stack frames in Tiger compiler
- Intermediate code and its representation trees
- Liveliness in Tiger compiler

3.1 INTRODUCTION

A compiler's analysis step breaks down a source program into its components and generates intermediate code, which is an internal representation of the program. The intermediate code is translated into the target program during the synthesis process. The syntax of the language to be compiled is used to conduct the analysis. The syntax of a programming language explains the correct format of its programs, but the semantics of the language defines what each program means when it runs. We offer a commonly used notation for describing syntax termed context-free grammars.

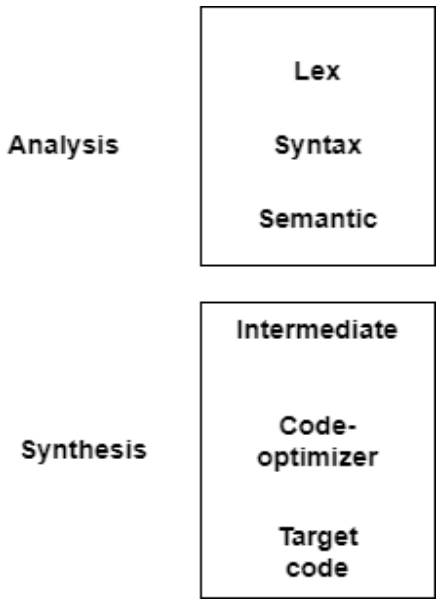


Fig 3.1 Phases of compiler

3.2 SYNTAX-DIRECTED TRANSLATION

Background: Parser by using CFG (Context free Grammar) validate any input statement and generate output for the next phase of the compiler. This output could be represented in the form of either a parse tree or abstract syntax tree. At this stage semantic analysis is associated with the syntax analysis phase of compiler by using Syntax Directed Translation.**Definition**: *Syntax Directed Translation are additional notations to the grammar that make semantic analysis more simple and effective.* These additional informal notations are called *Semantic Rules*. In general, syntax directed translation states that meaning of any input statement is related to its syntactic structure (Parse Tree).

In Syntax Directed Translation, we attach attributes to the grammar symbols representing the language constructs. Values for these attributes are calculated by the semantic rules augmented with the grammar. These semantic rules use:

- Lexical values of nodes (returned by lexical analyzer)
- Constants
- Attributes associated to the node.

Notations to attach semantic rules:

1. Syntax Directed Definitions (SDD)

- A syntax-directed definition (SDD) is a generalized context-free grammar along with both attributes and rules. Attributes set is associated with grammar symbols and semantic rules are associated with productions for computing the attribute value.

- For example:

Production	Semantic Rules
$L \rightarrow X$	$L.val := X.val$
$X \rightarrow X + T$	$X.val := X.val + T.val$
$X \rightarrow T$	$X.val := T.val$
$T \rightarrow T * F$	$T.val := T.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow const$	$F.val := const.lexval$

Fig 3.2 Syntax directed definition of desk calculator

Where,

X is a symbol and *val* is one of its attributes.

X.val denotes the value of that attribute at a particular parse-tree node X.

lexval is integer valued attribute returned by lexical analyzer.

Attributes may attain value of any type: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

If nodes of the parse tree are implemented as records or objects, then the attributes of X can be implemented by data fields in record corresponding to node X.

There are two types of attributes:

- Synthesized Attributes: These are derived from the values of the children node's attributes.
- Inherited Attributes: These are calculated from the values of the siblings' and parent's attributes.

2. **Translation Schemes**.

   SDDs hide many implementation details and give high-level specification whereas translation schemes are more implementation oriented and indicate the order of evaluation of semantic rules.

Grammar + Semantic Rules = SDT

Applications of Syntax Directed Translation (SDT)

1. Executing Arithmetic Expression
2. Conversion from infix to postfix
3. Conversion from infix to prefix
4. Conversion from binary to decimal
5. Counting number of reductions
6. Creating syntax tree
7. Generating intermediate code
8. Type checking
9. Storing type information into symbol table

A syntax-directed translation scheme (SDT) is a context-free grammar with some program component embedded within production bodies. These program components are called semantic actions. These actions are enclosed between curly braces at any position where action is to be performed.

3.3 IMPLEMENTATION OF SYNTAX-DIRECTED TRANSLATORS

Any SDT is implemented by first creating a parse tree and then performing the actions in preorder traversal.

Syntax Directed Translation (SDT) Schemes

1. Postfix Translation Schemes
2. Parser-Stack Implementation of Postfix SDT's
3. SDT's With Actions Inside Productions
4. SDT for L-Attributed Definitions

3.3.1 Postfix Translation Scheme:

In this scheme we parse the grammar bottom up and each action is placed at the end of production i.e. all the actions are at right ends of the productions. This SDT is called Postfix SDT.

Fig 3.2 implements Desk calculator SDD of Fig 3.1 as a postfix SDT. "Print a value" action is performed for first production and rest of the actions are equivalent to the semantic rules.

Production	Semantic Actions
$L \rightarrow X$	$\{Print\ (X.val);\}$
$X \rightarrow X + T$	$\{X.val = X.val + T.val;\}$
$X \rightarrow T$	$\{X.val = T.val;\}$
$T \rightarrow T * F$	$\{T.val = T.val * F.val;\}$
$T \rightarrow F$	$\{T.val = F.val;\}$
$F \rightarrow (E)$	$\{F.val = E.val;\}$
$F \rightarrow const$	$\{F.val = const.lexval;\}$

Fig 3.3 Postfix SDT implementation of desk calculator

Example 3.1: Parse tree for evaluating expression 23*4+5

SDD uses only synthesized attributes and semantic rules are evaluated by bottom-up, post order traversal.

Production	Semantic Actions
$L \rightarrow X$	$\{Print\ (X.val);\}$
$X \rightarrow X1 + X2$	$\{X.val = X1.val + X2.val;\}$
$X \rightarrow X1 * X2$	$X.val = X1.val * X2.val;\}$
$X \rightarrow (X1)$	$\{X.val = X1.val;\}$
$X \rightarrow T$	$\{X.val = T.val;\}$
$T \rightarrow T1const$	$T.val = 10 * T1.val + const.lexval;\}$
$T \rightarrow const$	$\{F.val = const.lexval;\}$

Fig 3.4 Grammar and Semantic actions to solve expression 23*4+5

1. S-attributed SDT :

- If any SDD contains only synthesized attributes, it is called as S-attributed SDD.
- S-attributed SDDs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

2. L-attributed SDT:

- If an SDTD has both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDD.
- Attributes in L-attributed SDDs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS

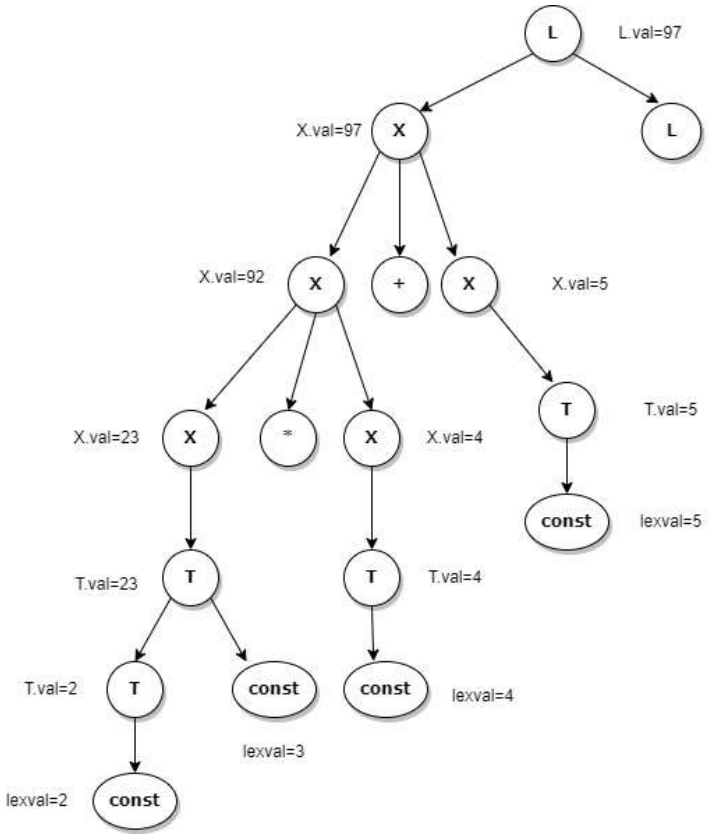


Fig 3.5 The Annotated parse tree

3.3.2 Parser-Stack Implementation of Postfix SDT's

During LR parsing, post x SDTs may be built by executing the actions when reductions occur. Each grammar symbol's attribute(s) can be placed on the stack at a location where they can be discovered throughout the reduction.

The optimal strategy is to store the characteristics, as well as the grammar symbols in records on the stack. The parser stack record contains one field for grammar symbols (or parser state) and, below it, a field for its any attribute. For example:

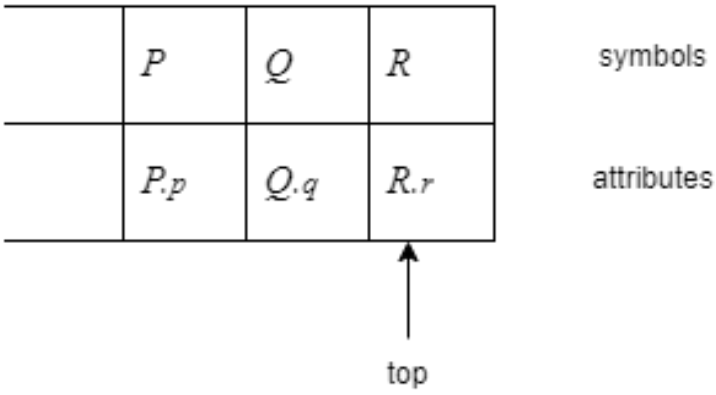


Fig 3.6 Parser stack

Here, P,Q,R on the top of stack are grammar symbols and when these are reduced according to any production then their attributes are stored in lower field of the stack. Say *P.p* is one of its attributes. Any number of attributes are stored on stack. If attributes are of large size like character strings then attribute value is stored elsewhere and the pointer to that value is put in the stack.

Production	Semantic Actions
$L \rightarrow X$	$\{Print\ (stack[top-1].val);\ top = top - 1;\}$
$X \rightarrow X + T$	$\{stack[top-2].val = stack[top-2].val + stack[top].val;\ top = top - 2\}$
$X \rightarrow T$	
$T \rightarrow T * F$	$\{stack[top-2].val = stack[top-2].val * stack[top].val;\ top = top - 2\}$
$T \rightarrow F$	
$F \rightarrow (E)$	$\{stack[top-2].val = stack[top-1].val; top = top - 2\}$
$F \rightarrow const$	

Fig 3.7 Implementing desk calculator on bottom-up parser stack

3.3.3 SDT's With Actions inside Productions

In this scheme action is placed anywhere inside the production and is performed immediately after all left side terminals. For example, if we have a production $X \rightarrow Y\{a\}\,Z$, here action a is performed after processing of Y.

In bottom-up parsing, action a is performed as soon as Y comes at the top in parser stack.

In top-down parsing, action a is performed just before the expansion of non-terminal Y.

3.3.4 SDT for L-Attributed Definitions

In this scheme the grammar is parsed top-down. Here actions in a parse tree are executed in pre-order traversal of the tree. Principles to convert L-attributed SDD into SDT are:

- Action to compute inherited attributes of a non-terminal symbol is embedded just before this symbol in the production. Attributes are calculated in the manner, that first required attributes are evaluated first.
- Action to compute synthesized attribute is placed at the end of the production.

3.4 SEMANTIC ANALYSIS

As we have studied Syntax Directed Translation led to generate Abstract Syntax Tree. It is fed to Semantic Analysis phase, where meaning of each phrase is determined. This process relates uses of variables to their definitions, checks types of declarations and expressions and requests translation of each phrase into representation easy to generate machine code.

3.4.1 Introduction to Tiger Compiler

A modern Compiler design has many phases, each for different aspects of language. This section focuses on compiling Tiger, a small but non-trivial language from Algol family. It is description language with syntax trees, nested structures, heap allocated record values with embedded pointers, arrays, strings and integer variables, and some control structures. These constructs make Tiger both functional and object oriented. In this chapter compilation of real programming is illustrated by using Tiger code fragments.

3.4.2 Bindings for Tiger Compiler

Symbol Tables:

Symbol tables are also called environments. As we know symbol table maps identifiers to their types and locations. As the declarations of identifiers are processed, each literal, variable and function is bound to "its meaning" in symbol table. Whenever these identifiers are faced in executable statements, they are searched in the symbol tables.

A symbol table is a set of bindings denoted by arrow ($\mapsto$).Each declared variable in the program has its scope in which it is known & can be visible. As semantic analyzer reaches to the end of that scope, the variable bindings in the table are discarded.

Assume example in Tiger language:

```
1    function f ( a:int, b:int ) =
2    ( print_int (a+b);
3    let var m:= a+b
4    var a := "tiger"
5    in print (a); print_int (m)
6    end;
7    print_int (b)
8    )
```

Suppose this program has initial symbol table $\sigma°$, after processing

Line 1, we get

Table $\sigma 1 = \sigma° + \{a \mapsto \text{int}, b \mapsto \text{int}\}$

i.e new bindings for a and b are added to $\sigma°$.

Line 2, identifiers are searched in $\sigma 1$.

At line 3, we get

Table $\sigma 2 = \sigma 1 + \{m \mapsto \text{int}\}$

At line 4, we get

Table $\sigma 3 = \sigma 2 + \{a \mapsto \text{string}\}$

Now same symbol has different bindings, so right hand side table {a $\mapsto$ string} overrides bindings in the left ($\sigma 2$).

At line 6, tables $\sigma 2, \sigma 3$ are discarded.

At line 7, table $\sigma 1$ is used to search for b.

At line 8, table $\sigma 1$ is discarded and go back to $\sigma°$.

In Tiger there are two name spaces: one is for types, and the other for functions & variables. Any type 'a' can be present with variable a or function a in same scope at the same type but both variable and function can't have same name within scope simultaneously. A type identifier is associated with a Ty_ty. The types module describes the structure of types. For eg.

```
/* types.h*/
typedef struct Ty_ty_ *Ty_ty;
typedef struct Ty_tyList_ *Ty_tyList;
struct Ty_ty_ {enum
{ty_record, Ty_int, Ty_string, Ty_array, Ty_name, Ty_nill, Ty_void}
kind;
}
```

The primitive types in Tiger are int and string, other types are constructed by using records, arrays or primitive types. Record types hold the names and fields types. Arrays are just like records and carries the type of array element.

A symbol table provide mappings from symbols to bindings and We need two environments one is value environment and the other is type environment. Consider following example:

```
let type a = int
var a: a := 5
var b: a := a              //here a denoting type of b
in b+a
end
```

here, in syntactic context, symbol a denotes the type a where type identifier is expected and denotes the variable a where variables are expected.

3.4.3 Type-checking Expressions

For each type identifier we need to remember only the type it stands for. Hence a type environment maps symbols to Ty_ty_ and the lookup function for symbol table of that environment will always return Ty_ty pointers.

For each value identifier we need to know whether it is a variable or a function; if a variable, what is its type; if a function, what are its parameters and type of result, and so on. Type enventry will hold all this information in following type-checking environment:

```
typedef struct E_enventry_ *E_enventry;
struct E_enventry_ {enum     {E_varEntry, E_funEntry} Kind;
union {struct {Ty_ty ty;} var;
struct {Ty_tyList formals; Ty_ty result;} fun;
} u;
};
```

A variable is mapped to varEntry where its type is found and if we are looking for a function, it is mapped to funEntry which contains:

formals- type list of formal arguments.

result- return type of function.

During processing of expressions for each identifier type-checker consult these environments.

The type-checker is a recursive function of the abstract syntax tree. This is transExp which is also used for translating the expression into intermediate code.

In several languages, addition (+) is overloaded, if one integer operand is added to real operand then real result is produced. Here integer operand is implicitly converted into real. But compiler need to convert it explicitly in the machine code.

3.4.4 Type-checking Declarations

In Tiger, any declaration occurs only by let expression. Type-checking of let expression is very simple and tansDec module is used to translate declaration. The call for transDec, enhance the value environment (venv) and type environment (tenv) with new declarations.

- **Variable declarations-** It is very simple to process, a variable is declared without any type constraint. E.g:

  var x := exp.

- **Type declarations**- When a variable is declared with type constraint and initializing expression. E.g:

  var x: type-id := exp.

- **Function declarations**- In Type-checking implementation of function declaration, first
  - transDec will search result-type identifier in the tenv.
  - Then, a local function will traverse through the formal arguments list and return their types. This information is entered into venv.
  - Formal parameters are entered into the value environment.
  - This augmented environment is used to process the body (expressions).
- **Recursive declaration**- Above procedure will not work with recursive type or function declarations. Because undefined identifiers (type or functions) are encountered while processing and this error cannot be handled by above implementation. The solution for these recursive things is to put all the prototypes (headers) in the environment first. Then process all the bodies in that environment. While processing of body all newly faced identifiers are looked up in the environment.

3.5 ACTIVATION RECORDS

In several languages such as C, Pascal, Tiger etc., local variables are created at the time of entry to the function and destroyed when function returns. Several function calls may exist simultaneously and each call has its own instances of variables. Consider below tiger function:

```
function f (a:int):int =

let var b := a+a

in if y<50

then f(b)

else b-1

end
```

Each time new instance of a is created when f is called and for each a instance of b is also created when entered into the body. Because this is recursive call, many a's exist simultaneously. A function returns in LIFO manner i.e it returns when all its called functions have returned. So we can use a Stack (LIFO) to hold local variables.

3.5.1 Stack Frames

Two operations are performed on stack data structures. On entry to the function, local variables are pushed into the stack and popped on exit in large batches. All variables are not initialized at time of push and we keep accessing all variables deep in stack. This way we need a different suitable model.

In this model, stack is used as array with special register (called **stack pointer**), which locate variable in this big array. Size of stack increases with entries and shrinks with exit from the function. These locations on the stack allocated to the local variables, formal parameters, return and other temporary identifiers are called that function's **Activation Records** or **Stack Frames**. For example consider following stack frame:

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>incoming arguments
frame pointer →</td><td>argument n
argument 2
argument 1
static link</td><td>↑ higher addresses
previous frame</td></tr>
<tr><td></td><td>local
variables</td><td rowspan="2">current frame</td></tr>
<tr><td></td><td>return address
temporaries
saved registers</td></tr>
<tr><td>outgoing arguments
stack pointer →</td><td>argument m
argument 2
argument 1
static link</td><td></td></tr>
<tr><td></td><td></td><td>next frame

↓ lower addresses</td></tr>
</table>

Fig 3.8

Features of this stack frame:

i. Stack starts from higher addresses and grow towards lower addresses.

ii. In previous frame incoming arguments are passed by the caller and stored at known offset from frame pointer.

iii. Return address is created by call statement, it tells where control should return after completion of currently called function. Some local variables are stored in current frame others are stored in machine register. Machine register variables sometimes shifted to the frame to create space in register.

iv. When this function calls another function (nested function) then outgoing argument space is used to pass parameters.

v. Stack pointer is pointing to the first argument passed by calling function. On entry of function new frame is allocated and size of that frame is subtracted from SP to generate new SP. At this time old SP is called Frame Pointer FP. i.e. FP=SP+ frame size.

vi. When function exits FP is copied back to SP and current FP attains old FP value.

vii. Return address is the address of instruction which is just next to call statement in the calling function.

3.5.2 Frames in the Tiger Compiler

Including Tiger there are many languages (Pascal, ML), which support **block structure** feature. That feature allow, in nested functions, the inner function can also use variables declared in outer function.

Following arrangements can achieve this.

i. When a function f is called, it is passed a pointer to the frame of that function which enclosed f statistically. This pointer is called **static link**.

ii. A global array is maintained which contains static nesting depth pointers, this array is called **Display**.

iii. When a function f is called, all the variables of calling function are also passed as extra arguments (and same way passed to nested functions of f). This method is called **lambda lifting**.

If we uses C functions in Tiger then, Tiger compiler uses standard stack frame layout, and abstract semantic analysis module which hides the internal representation of the symbol tables. This abstract implementation makes module machine independent.

Tiger compiler have two layers of abstraction between semantic analysis and frame layout:

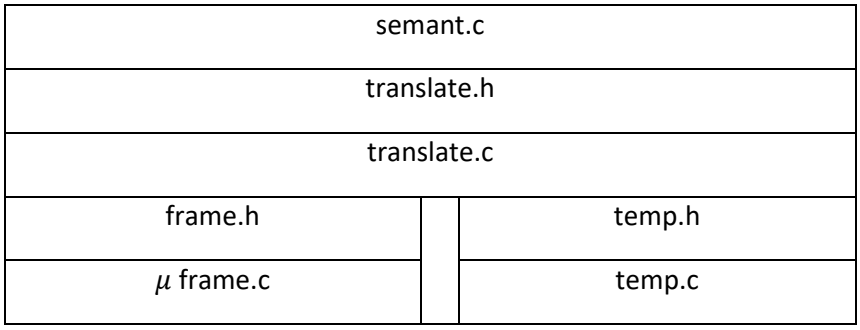


Fig 3.9

Here,

- frame.h and temp.h are interfaces to provide machine independent views for memory and register variables.
- Translate module handles nested scope (block structure feature) by using static links and provide abstract interface translate.h to semant module.
- μ stands for target machines. Abstraction is used to separate source code semantics and machine dependent frame layout. So that frame module become independent of specific language being compiled.

3.6 TRANSLATION TO INTERMEDIATE CODE

After type-checking semantic phase of tiger compiler request to translate abstract syntax into abstract machine code which is independent of machine target or any source language. It increases portability and modularity. This code is called Intermediate representation (IR).

Front end of any portable compiler perform analysis: Lexical analysis, parsing, semantic analysis, and intermediate representations. The back end compiler does optimization of IR and translate it to target machine instructions.

Without IR we require m*n compilers to compile m number of languages of n number of machines, whereas m front ends generate one IR only n backend compilers are required to convert it into machine code. For e.g.

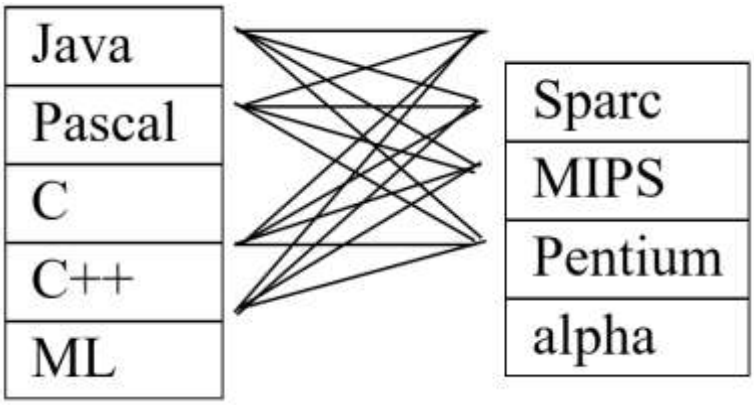


Fig 3.10 without IR

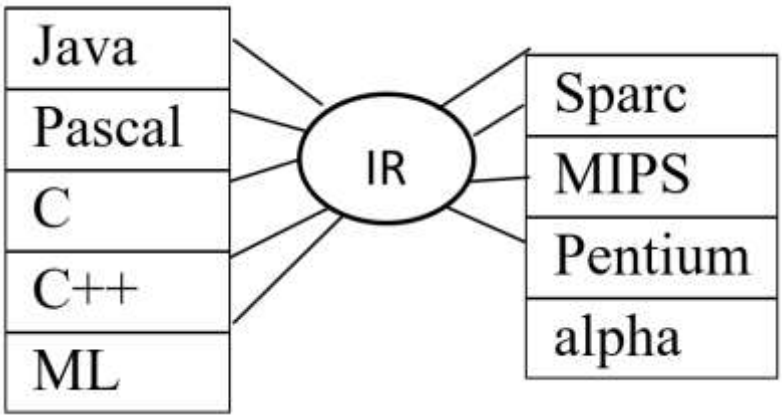


Fig 3.11 with IR

3.6.1 Intermediate Representation Trees

In tiger compiler implementation interface *tree.h* defines intermediate tree representation. A good IR must have following qualities:

It must be convenient to produce for semantic analysis phase.

For all the target machines, it must be convenient to translate it into machine code.

Code optimizing implementation requires to rewrite the IR, so each constructs should have clear and simple meaning.

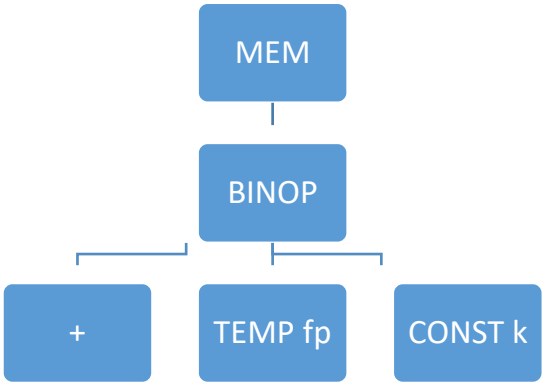
3.6.2 Translation into Trees

In some architectures it is found that complex structures (array subscript, procedure call etc.) of abstract syntax tree doesn't lead to corresponding complex machine instructions. These structures are transformed into abstract machine instructions. Therefore IR should have individual simple things- fetch, store, jump or add and then these abstract instructions are grouped together into clumps to form target machine instructions.

Translation of abstract syntax expression to intermediate tree:

Expressions: Abstract syntax expression *A_exp* is represented as *T_exp* in tree language, which computes a value. Those expressions which return no value (e.g procedure calls, while statements etc.) are represented as *T_stm.* Expressions with conditional jumps (e.g a>b) are represented as the combination of *T_stm* and *Temp_labels* (destinations).

Simple variables: In section 3.4.4 and 3.4.5 semantic analyzer function type checks a variable in type environment (tenv) and value environment (venv). At this stage semants of *exp* are modified to generate intermediate representation translation. A simple variable *v* declared in stack frame is translated as:



MEM(BINOP(PLUS, TEMP fp, CONST k))

Where, mem: content of memory word size.

BINOP: binary operation

TEMP fp: temporary frame pointer

CONST k: offset of variable *v* in the current frame

Array variables: All languages handle array elements differently. Like, in pascal array variable stands for its content where as in C, array variables are like pointer constants. So there is translate function which handles array subscripts for records fields, expression etc.

3.6.3 Declarations

Modules for type-check **Let** expressions are enhanced to translate into tree expression i.e *TransDec* and *TransExp* will accept more arguments now.

- Variable: as we know *transDec* updates *tenv* and *venv*. Now *transDec* will also return *Tr_exp* for assignment expressions to perform initializations.
- Function: Each tiger function is translated into assembly language with a prologue, a body, and an epilogue.

  Prologue specifies:

  - The beginning of the function and label definition for function name.
  - Adjust the stack pointer to allocate new frame.
  - Maintain callee-save registers and return address registers.

  Then comes body of function.

  - After body epilogue comes which specifies:
  - Pops the stack
  - Return value
  - Restore callee save register
  - Reset stack pointer
  - Jump to the return address
  - End of a function

3.7 BASIC BLOCKS AND TRACES

While translating trees generated by semantic analysis into machine language, operators capable on most of the machines are chosen. Some aspects of tree languages doesn't correspond to machine code and interfere with compiler optimization. For eg: the order of evaluating subexpression of any expression doesn't matter but if tree expression includes side effects ESEQ (expression sequence), CJUMP and CALL then mismatch between tree and machine language can occur. Because they make different orders of evaluation yielding different results.

Translation is done in three stages:

1. A tree is rewritten into equivalent trees i.e canonical trees without these SEQ or ESEQ, CALL labels, internal jumps etc. Hence, canonical trees have no SEQ or ESEQ

2. This list of trees is grouped into **basic blocks**.

3. Then basic blocks are ordered in such a way that each CJUMP is followed by its false label. These arrangements are called **traces**.

3.7.1 TAMING CONDITIONAL BRANCHES

Most of the machine languages doesn't have direct equivalent of CJUMP instruction. In the tree language CJUMP is designed with two way branch, i.e it can jump either of two target labels whereas in real machine, the conditional jump can either transfer control on true condition or comes to immediate next instruction. So to transfer it into machine language every CJUMP (condition, Tlabel, Flabel) is arranged in such a way that it is immediately followed by its false label *Flabel*. Then each CJUMP can directly be translated as conditional branching with only true label *Tlabel.*

BASIC BLOCKS

In order to determine where to jump in a program we analyze control flow. The flow of control in a program doesn't know whether the jump is for true or false value. So control flows sequentially till it faces a jump statement. We can group these sequential non branching instructions (without any jump) into a basic block and analyze flow of control between these basic blocks.

A basic block is a set of non-branch instructions to be executed sequentially. We enter into a basic block at the beginning and exit at the end. i.e

- First statement of a basic block is a label.
- There are no other JUMPs, LABELs or CJUMPs.
- Only the last statement is a jump statement (CJUMP). We can say after that control enters into another basic block. Where ever a label is found a new basic block is started and that basic block ends whenever a JUMP or CJUMP is found.

This arrangement is applied to every function body. The last basic block has no JUMP at the end, so label *done* is appended at last to indicate the beginning of epilogue.

TRACES

Now the order of basic blocks doesn't affect execution result. Control jumps to new appropriate place at the end of each block. So we arrange these blocks in such a way that every CJUMP is followed by its false label. Also many target labels are immediate next to their unconditional jumps. So that deletion of these unconditional JUMPs makes compilation of program faster.

A trace is a order of basic blocks (sequence of statements) that can be consecutively executed during the program execution. A program can have many traces. While arranging CJUMPs and False-labels, a set of traces should cover the program. If we reduce the number of jumps from one trace to another, we will have few traces. For example, if blocks b1 ends with a jump to b6 and b6 has a jump to b4, then the trace will be b1, b6, b4. Now imagine block b4 have last instruction CJUMP (condition, b3,b7). Since it can't be decided at compile time which is false label, by assuming b7 is false label and some execution will follow it, we append b7 to our trace b1, b6, b4, b7 and b3 will fall under different trace.

3.8 LIVENESS ANALYSIS

The front end compiler translates program into an intermediate language with many number of temporaries. All of these are never written explicitly. But machine has limited number of registers and many temporaries can fit into these few registers if all of them are not in use at the same time. Other excess temps can be kept in memory.

Liveness analysis is process of analyzing intermediate representation to find which temporary variable are used at the same time and are live. A variable is live if its value is needed in future. To analyze this control flow a graph is made, which determines the order of statements to be executed. For example, in the following flow graph A is live at edge {2, 5, and 5 to 2}. Variable B is not live at edge 1, 2, 5 and 6 because it is not used at this time. It is assigned into at statement 2, so its live range is {3, 4}. If C is a formal parameter, it is live at entry and exit of the code. So only two registers can hold values of three variables because variable A and B are not live at same time so one register is for A and B and the other is for C.

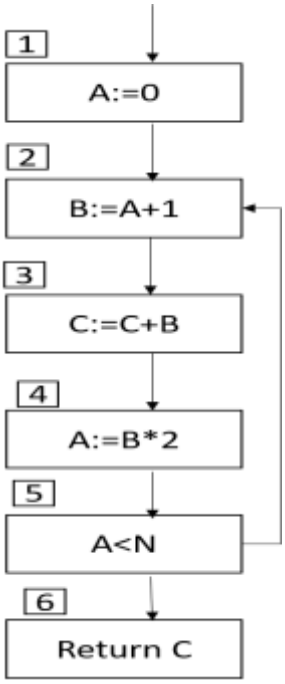


Fig 3.12 flow graph

3.8.1 SOLUTION OF DATAFLOW EQUATIONS

The edges that come to any node from predecessor nodes are called **in-edges** of that node and the edges that are going towards successor nodes are called **out-edges** of a that node. If any variable is live at any in-edge then it is called live-in and if it is live at any out-edge then it is live-out. We can analyze liveness of any variable by using two terms:

Def: def of a variable is the set of those graph nodes which defines that variable.

Uses: use of variable is the set of those variables or graph nodes which uses that variable.

Live range of each variable across the dataflow graph is calculated by following equation:

$$\boldsymbol{in}[\boldsymbol{n}] = \boldsymbol{use}[\boldsymbol{n}] \cup (\boldsymbol{out}[\boldsymbol{n}] - \boldsymbol{def}[\boldsymbol{n}])$$

$$\boldsymbol{out}[\boldsymbol{n}] = \bigcup_{\boldsymbol{s} \in \boldsymbol{succ}[\boldsymbol{n}]} \boldsymbol{in}[\boldsymbol{s}]$$

Where,

Pred[n]: set of all predecessors of node n

Succ[n]: set of all successors of node n

From above equation we can say:

A variable is live-in at node n, if it is in use set of that node i.e in use[n].

If a variable is live-out at n but it is not defined at n i.e not in def[n], then this variable must be line-in at n.

Live-out variable at n is live-in at all nodes s in succ[n].

3.8.2 INTERFERENCE GRAPH CONSTRUCTION

Liveness information is used in compiler optimization. Some optimization algorithms need to know at each node in the flow graph, which set of variables are live. Registers are allocated to the temporaries accordingly. If we have a set of temporary variables v1,v2,v3,……, which are to be allocated to registers r1,r2,r3,……, then the condition due to which we can't allocate same register to v1 & v2 is called an **interference**. This may occur due to overlapping live period i.e v1 & v2 both are live at same time in the program. In this case we can't assign same register to these variables. Interference can also occur when any variable v1 is generated by such instruction which doesn't address register r1, in this case v1 and r1 interfere. Interference information is represented as matrix of variables by marking x on the inference. This matrix can be expressed as undirected graph. Each node of graph is representing variables and edge between two nodes (variables) represent interference. Interference matrix and corresponding graph of fig 3.9 is:-

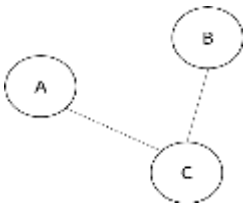


Fig 3.13 interference representation

	A	B	C
A			x
B			x
C	x	x	

3.8.3 LIVENESS IN THE TIGER COMPILER

In tiger compiler, first control-flow graph is generated and then liveness of a variable is analyzed. This analysis is expressed as interference graph. To represent these two types of graph, an abstract data type *Graph* is created.

G_Graph() := function to create empty directed graph.

G_Node(g,x) := adds new node in graph g with additional information x.

G_addEdge(n,m) := creates directed edge from n to m, now m is available in G_succ(n) list and n is present in G_pred(m) list. If instruction n is followed by instruction m (even by a jump), then there will be an edge between n and m in the control-flow graph. In flow graph each node contains information about the following:

a) FG_def(n): a set of all temporaries defined at n

b) FG_use(n): a set of temporary variables used at n

c) FG_isMove(n): represents any Move instruction at n

LIVENESS ANALYSIS

The liveness module takes flow-graph as input and produces:

- Interference graph
- List of node-pairs (having Move instruction). These are assigned to the same register to eliminate Move)

What happens if a freshly defined temporary isn't active right away? If a variable is defined but never utilized, this is the situation. It appears that there is no need to enter it in a register; hence, it will not conflict with any other temporary identifiers. However, if the defining instruction is executed it will write to a register, which must not contain any other live variables. As a result, any live ranges that overlap zero-length live ranges will interfere.

3.9 SUMMARY

This chapter gives the translation of languages guided by context-free grammars. The translation techniques in this chapter are applied for type checking and intermediate-code generation in compiler design. The techniques are also useful for implementing little languages for specialized tasks. To illustrate the issues in compiling real programming languages, code snippets of Tiger (a simple but nontrivial language of the Algol family, with nested scope and heap-allocated records) are discussed. These code snippets can be implemented in C-language or java. For complete code refer book by A.Andrew et.al., Modern Compiler Implementation in java (2004).

3.10 EXCERCISE

Q1. What are inherited and synthesized attributes?

Q2. What is the difference between syntax directed definition and syntax directed translation?

Q3. What are implementation scheme of syntax directed translation?

Q4. Differentiate between L-attributed and S-attributed SDT.

Q5. How compiler checks declarations and expressions in a program?

Q6. How local variables are managed during function calls?

Q7. What are blocks and traces?

Q8. Write short note on liveness of variables using Tiger compiler.

3.11 REFERENCE FOR FURTHER READING

- Modern Compiler Implementation in Java, Second Edition, Andrew Appel and Jens Palsberg, Cambridge University Press (2004).
- Principles of Compiler Design, Alfred Aho and Jeffrey D. Ullman, Addison Wesley (1997).
- Compiler design in C, Allen Holub, Prentice Hall (1990).
- Mogensen, T. Æ. (2017). Syntax Analysis. In *Introduction to Compiler Design* (pp. 39-95). Springer, Cham.

❄❄❄❄❄❄❄

4

DATAFLOW ANALYSIS AND LOOP OPTIMIZATION

Unit Structure

4.0 OBJECTIVES

After going through this chapter you will be able to understand the following concepts in detail:-

DAG Representation, Dominators, Reducible flow graphs, Depth-first search, Loop invariant computations, Induction variable elimination, various loop optimizations.

Also Intermediate representation, dataflow analyses, transformations, speeding up and alias analysis.

4.1 INTRODUCTION

To create an efficient target language program, a programmer needs more than an optimizing compiler. We mention the types of code-improving transformations that a programmer and a compiler writer can be expected to use to improve the performance of a program. We also consider the representation of programs on which transformations will be applied.

4.2 OVERVIEW

The code produced by straight forward compiling algorithms can be made to run faster or take less space or both. This process is achieved by program transformations that are traditionally called optimizations.

The maximum optimization benefit can be obtained if we can identify the frequently executed parts of the program and make these parts as efficient as possible. Generally inner loops in the program written using while or for statements are good candidates for optimization.

4.3 THE PRINCIPLE SOURCES OF OPTIMIZATION

4.3.1 Loop optimization:

Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities.

4.3.2. The DAG representation of basic blocks

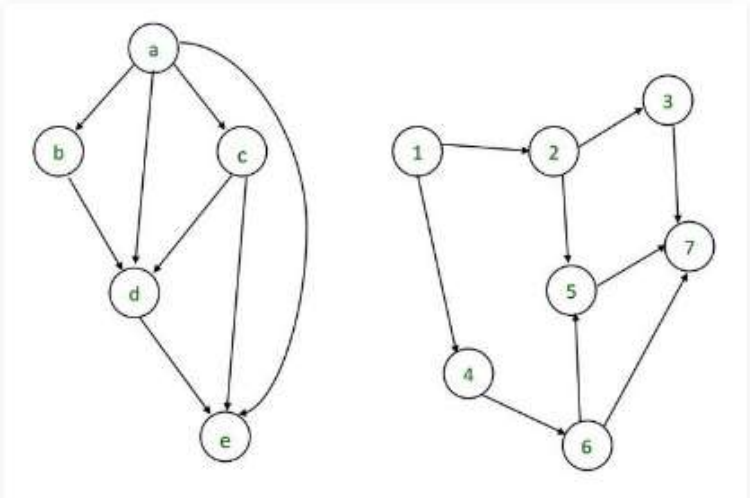
A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

A three address statement x: = y + z is said to define x and use y and z. A name in a basic block is said to be live at a given point if its value is used after that point in the program perhaps in another basic block.

A DAG for basic block is a directed acyclic graph with the following labels on nodes:

- The leaves of graph are labelled by unique identifier and that identifier can be variable names or constants.
- Interior nodes of the graph are labelled by an operator symbol.
- Nodes are also given a sequence of identifiers for labels to store the computed value.
- DAGs are a type of data structure. It is used to implement transformations on basic blocks.
- DAG provides a good way to determine the common sub-expression.
- It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Examples of directed acyclic graph :



Algorithm for construction of Directed Acyclic Graph :

There are three possible scenarios for building a DAG on three address codes:

Case 1 – x = y op z

Case 2 – x = op y

Case 3 – x = y

Directed Acyclic Graph for the above cases can be built as follows :

Step 1 –

If the y operand is not defined, then create a node (y).

If the z operand is not defined, create a node for case(1) as node(z).

Step 2 –

Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).

For the case (2), see if there is a node operator (OP) with one child node (y).

Node n will be node(y) in case (3).

Step 3 –

Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.

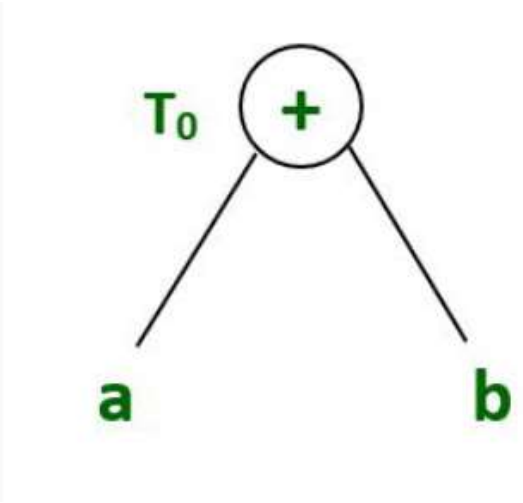
Example :

T0 = a + b —Expression 1

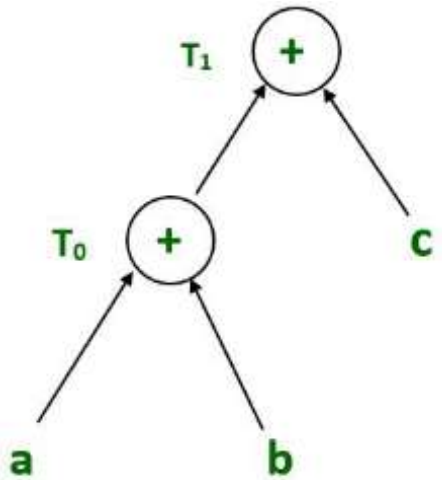
T1 = T0 + c —-Expression 2

d = T0 + T1 ——Expression 3

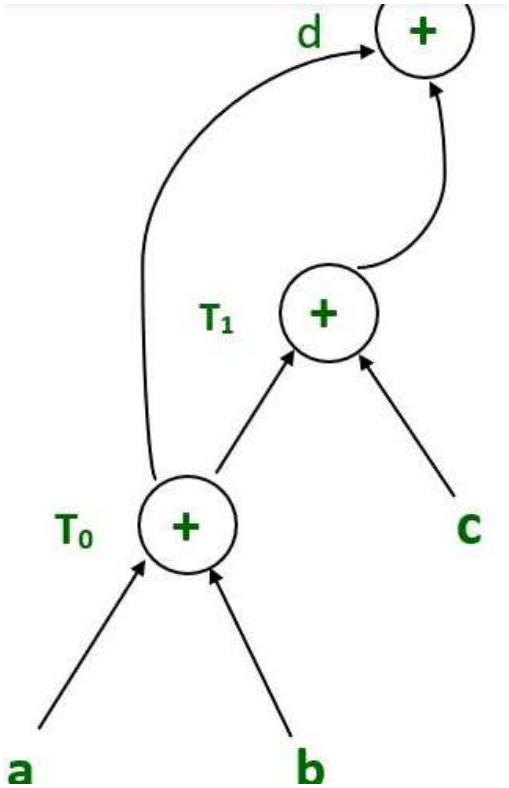
Expression 1 : T0 = a + b



Expression 2: T1 = T0 + c



Expression 3 : d = T0 + T1

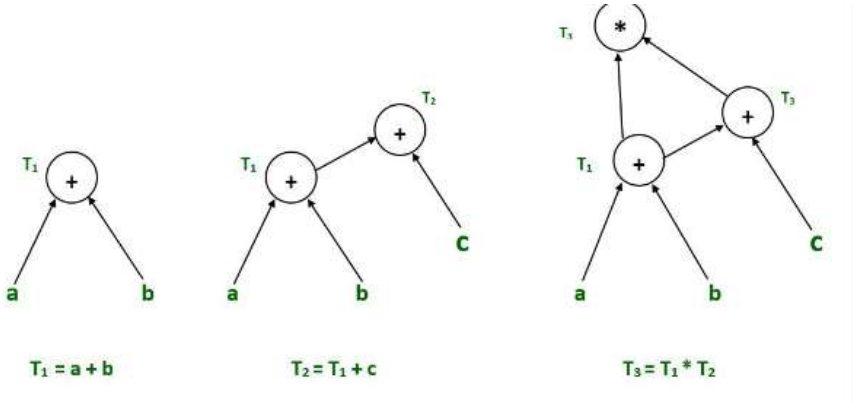


Example :

T1 = a + b

T2 = T1 + c

T3 = T1 x T2



Example :

T1:= 4*I0

T2:= a[T1]

T3:= 4*I0

T4:= b[T3]

T5:= T2 * T4

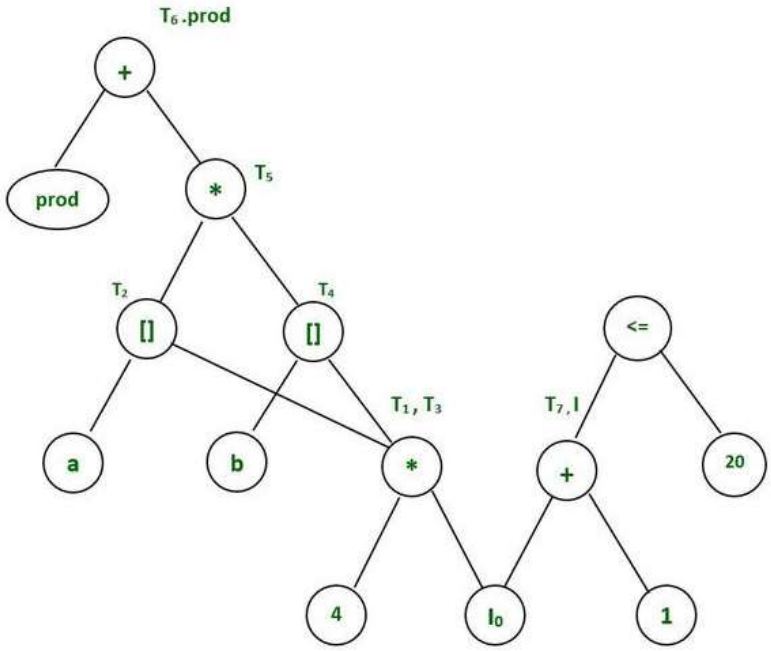
T6:= prod + T

prod:= T6

T7:= I0 + 1

I0:= T7

if I0 <= 20 goto 1



Application of Directed Acyclic Graph:

Directed acyclic graph determines the subexpressions that are commonly used.

Directed acyclic graph determines the names used within the block as well as the names computed outside the block.

Determines which statements in the block may have their computed value outside the block.

Code can be represented by a Directed acyclic graph that describes the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently.

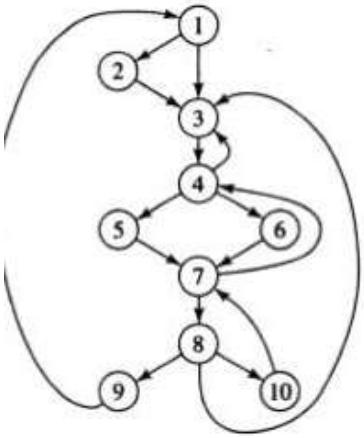
Several programming languages describe value systems that are linked together by a directed acyclic graph. When one value changes, its successors are recalculated; each value in the DAG is evaluated as a function of its predecessors.

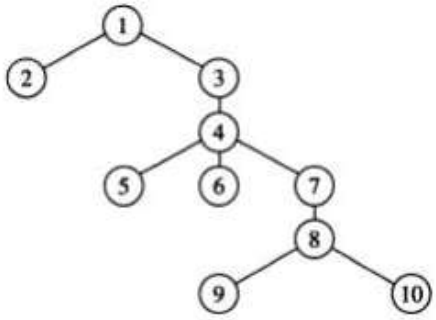
4.3.3 Dominators

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d Dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

In the flow graph below,*Initial node, node1 dominates every node. *node 2 dominates itself *node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3. *node 5 and 6 dominates only themselves, since flow of control can skip around either by going through the other. *node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10. *node 9 and 10 dominates only themselves.





The way of presenting dominator information is in a tree, called the dominator tree, in which

- The initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.

The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the Dom relation, the immediate dominator m has the property is d=! n and d Dom n, then d Dom m.

D (1) ={1}

D (2)={1,2}

D (3)={1,3}

D (4)={1,3,4}

D (5)={1,3,4,5}

D (6)={1,3,4,6}

D (7)={1,3,4,7}

D (8)={1,3,4,7,8}

D (9)={1,3,4,7,8,9}

D (10)={1,3,4,7,8,10}

4.3.4 Reducible flow graphs

Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently. Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

The most important properties of reducible flow graphs are that

1. There are no jumps into the middle of loops from outside;
2. The only entry to a loop is through its header

Definition:

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

1. The forward edges from an acyclic graph in which every node can be reached from initial node of G.
2. The back edges consist only of edges where heads dominate theirs tails.

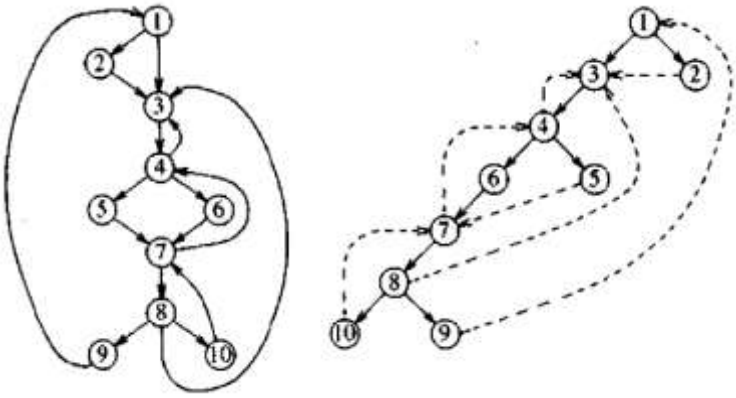
Example: The above flow graph is reducible. If we know the relation DOM for a flow graph, we can find and remove all the back edges. The remaining edges are forward edges. If the forward edges form an acyclic graph, then we can say the flow graph reducible. In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

4.3.5 Depth-first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

A depth first ordering can be used to detect loops in any flow graph; it also helps speed up iterative data flow algorithms.

One possible DFS representation of the data flow on left given on the right side figure.



4.3.6 Loop-invariant computations

- Definition
- A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop.
- A loop invariant is some predicate (condition) that holds for every iteration of the loop.
- For example, let's look at a simple for loop that looks like this:
- Int j = 9;
- for (int i=0; i<10; i++)
- J--;
- In this example it is true (for every iteration) that i + j == 9.
- A weaker invariant that is also true is that i >= 0 && i <= 10.
- One may get confused between the loop invariant, and the loop conditional (the condition which controls termination of the loop).
- The loop invariant must be true:
    - before the loop starts
    - before each iteration of the loop
    - after the loop terminates (although it can temporarily be false during the body of the loop).
    - On the other hand the loop conditional must be false after the loop terminates, otherwise, the loop would never terminate.
- **Usage:**

Loop invariants capture key facts that explain why code works. This means that if you write code in which the loop invariant is not obvious, you should add a comment that gives the loop invariant. This helps other programmers understand the code, and helps keep them from accidentally breaking the invariant with future changes.

A loop Invariant can help in the design of iterative algorithms when considered an assertion that expresses important relationships among the variables that must be true at the start of every iteration and when the loop terminates. If this holds, the computation is on the road to effectiveness. If false, then the algorithm has failed.

Loop invariants are used to reason about the correctness of computer programs. Intuition or trial and error can be used to write easy algorithms however when the complexity of the problem increases, it is better to use formal methods such as loop invariants.

Loop invariants can be used to prove the correctness of an algorithm, debug an existing algorithm without even tracing the code or develop an algorithm directly from specification.

A good loop invariant should satisfy three properties:

- Initialization: The loop invariant must be true before the first execution of the loop.
- Maintenance: If the invariant is true before an iteration of the loop, it should be true also after the iteration.
- Termination: When the loop is terminated the invariant should tell us something useful, something that helps us understand the algorithm.

Loop Invariant Condition:

Loop invariant condition is a condition about the relationship between the variables of our program which is definitely true immediately before and immediately after each iteration of the loop.

For example: Consider an array A{7, 5, 3, 10, 2, 6} with 6 elements and we have to find maximum element max in the array.

```
max = -INF (minus infinite)
for (i = 0 to n-1)
  if (A[i] > max)
max = A[i]
```

In the above example after the 3rd iteration of the loop max value is 7, which holds true for the first 3 elements of array A. Here, the loop invariant condition is that max is always maximum among the first i elements of array A.

This technique can be used to optimize various sorting algorithms like – selection sort, bubble sort, quick sort etc.

4.3.7 Induction variable elimination

A variable x is called an induction variable of loop L every time the variable x changes values, it is incremented or decremented by some constant

Example 1:

```
int i, max = 10, r;
r = max-1;
for(i=10;i<=r;i++)
{
Printf("%d", i);
}
```

In the above code, variable i is called induction variable as values of I get incremented by 1, i.e., 0,1,2,3,4,5,6,7,8,9,10

4.3.8 Some other loop optimizations.

Loop Optimization Techniques:

4.3.8.1 Frequency Reduction (Code Motion):

In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

4.3.8.2 Loop Unrolling:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

4.3.8.3 Loop Jamming:

Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops.

4.4 DATAFLOW ANALYSIS

Data flow analysis is a process for collecting information about the use, definition, and dependencies of data in programs. The data flow analysis algorithm operates on a control flow graph generated from an AST. You can use a control flow graph to determine the parts of a program to which a particular value assigned to a variable might propagate.

An execution path (or path) from point p1 to point pn is a sequence of points p1, p2, ..., pn such that:

for each i = 1, 2, ..., n − 1, either 1 pi is the point immediately preceding a statement and pi+1 is the point immediately following that same statement, **or**

pi is the end of some block and pi+1 is the beginning of a successor block.

In general, there is an infinite number of paths through a program and there is no bound on the length of a path. Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts.

No analysis is necessarily a perfect representation of the state.

Process of dataflow analysis:

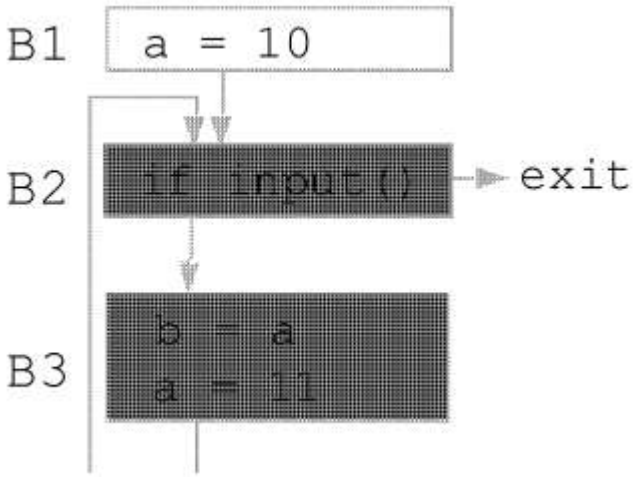
1) Build a flow graph(nodes = basic blocks, edges = control flow)
2) set up a set of equations between in[b] and out[b] for all basic blocks b.

   Effect of code in basic block:

   - Transfer function fb relates in[b] and out[b], for same b.
   - Effect of flow of control:
   - Relates out[b], in[b] if b1 and b2 are adjacent

3) Find a solution to the equations

   Static Program vs. Dynamic Execution:-



- Statically: Finite program
- Dynamically: Can have infinitely many possible execution paths
- Data flow analysis abstraction:
- For each point in the program:

combines information of all the instances of the same program point.

4.4.1 Intermediate representation for flow analysis

A compiler transforms the source program to an intermediate form that is mostly independent of the source language and the machine architecture. This approach isolates the front-end and the back-end.

The portion of the compiler that does scanning, parsing and static semantic analysis is called the front-end.

The translation and code generation portion of it is called the back-end.

The front-end depends mainly on the source language and the back-end depends on the target architecture.

More than one intermediate representation may be used for different levels of code improvement. A high level intermediate form preserves source language structure. Code improvements on loop can be done on it.

A low level intermediate form is closer to target architecture.

Parse tree is a representation of complete derivation of the input. It has intermediate nodes labeled with non-terminals of derivation.This is used (often implicitly) for parsing and attribute synthesis.

A syntax tree is very similar to a parse tree where extraneous nodes are removed.

It is a good representation that is close to the source-language as it preserves the structure of source constructs.

It may be used in applications like source-to-source translation, or syntax-directed editor etc.

Linear Intermediate Representation:-

Both the high-level source code and the target assembly codes are linear in their text.

The intermediate representation may also be linear sequence of codes. with conditional branches and jumps to control the flow of computation.

A linear intermediate code may have one operand address a , two-address b, or three-address like RISC architectures.

GCC Intermediate Codes:-

The GCC compiler uses three intermediate representations:

1. GENERIC - it is a language independent tree representation of the entire function.
2. GIMPLE - is a three-address representation generated from GENERIC.
3. RTL - a low-level representation known as register transfer language.

Consider the following C function.

```
double CtoF(double cel) { return cel * 9 / 5.0 + 32 ;}
```

C program with if:-

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

```
int l, m ;
scanf("%d", &l);
if(l < 10) m = 5*l;
else m = l + 10;
printf("l: %d, m: %d\n", l, m);
return 0;
}
```

C program with **for:**-

```
#include <stdio.h>
int main()
{
int n, i, sum=0 ;
scanf("%d", &n);
for(i=1; i<=n; ++i) sum = sum+i;
printf("sum: %d\n", sum);
return 0;
}
```

Representation of Three-Address Code:-

Any three address code has two essential components: operator and operand.

There can be at most three operands and one operator.

The operands are of three types, a name from the source program, a temporary name generated by the compiler or a constant a.

a There are different types of constants used in a programming language.

There is another category of name, a label in the sequence of three-address codes.

A three-address code sequence may be represented as a list or array of structures.

Quadruple:-

A quadruple is the most obvious first choice a.

It has an operator, one or two operands, and the target field.

Triple:-

A triple is a more compact representation of a three-address code.

It does not have an explicit target field in the record.

When a triple u uses the value produced by another triple d, then u refers to the value number (index) of d.
Example:-

t1 = a * a

t2 = a * b

t3 = t1 + t2

t4 = t3 + t2

t5 = t1 + t4

Indirect Triple:-

It may be necessary to reorder instructions for the improvement of execution.

Reordering is easy with a quad representation, but is problematic with triple representation as it uses absolute index of a triple.

As a solution indirect triples are used, where the ordering is maintained by a list of pointers (index) to the array of triples.

The triples are in their natural translation order and can be accessed by their indexes.

But the execution order is maintained by an array of pointers (index) pointing to the array of triples.

Static Single-Assignment (SSA) Form:-

This representation is similar to three-address code with two main differences.

Every definition a has a distinct name (virtual register).

Each use of a value refers to a particular definition.

e.g. t7= a + t3.

If the same user variable is defined on more than one control paths a, they are renamed as distinct variables with appropriate subscripts.

When more than one control-flow paths join, a φ-function is used to combine the variables.

The φ-function selects the value of its arguments depending on the control-flow path (data-flow under control-flow).

Each name is defined at one place a. Use of a name contains information about the location of its definition (data-flow).

SSA-form tries to encode data-flow under flow-control.

Consider the following C code:

for(f=i=1; i<=n; ++i) f = f*i;

The corresponding three-address codes and SSA codes are as follows.

i = 1 i0 = 1

f = 1 f0 = 1

L2: if i>n goto - if i0 > n goto L1

L2: i1 =

φ(i0, i2)

f1 =

φ(f0, f2)

f = f*i f2 = f1*i1

i = i + 1 i2 = i1 + 1

goto L2 if i2 <= n goto L2

L1: i3 =

φ(i0, i2)

f3 =

φ(f0, f2)

4.4.2 Various dataflow analyses

A data-flow value for a program point represents an abstraction of the set of all possible program states that can be observed for that point The set of all possible data-flow values is the domain for the application under consideration Example: for the reaching definitions problem, the domain of data-flow values is the set of all subsets of definitions in the program A particular data-flow value is a set of definitions IN[s] and OUT[s]: data-flow values before and after each statement s The data-flow problem is to find a solution to a set of constraints on IN[s] and OUT[s], for all statements.

Two kinds of constraints :

Those based on the semantics of statements (transfer functions)

Those based on flow of control

- A DFA schema consists of:
- A control-flow graph
- A direction of data-flow (forward or backward)
- A set of data-flow values
- A confluence operator (normally set union or intersection)
- Transfer functions for each block

We always compute safe estimates of data-flow values

A decision or estimate is safe or conservative, if it never leads to a change in what the program computes (after the change)

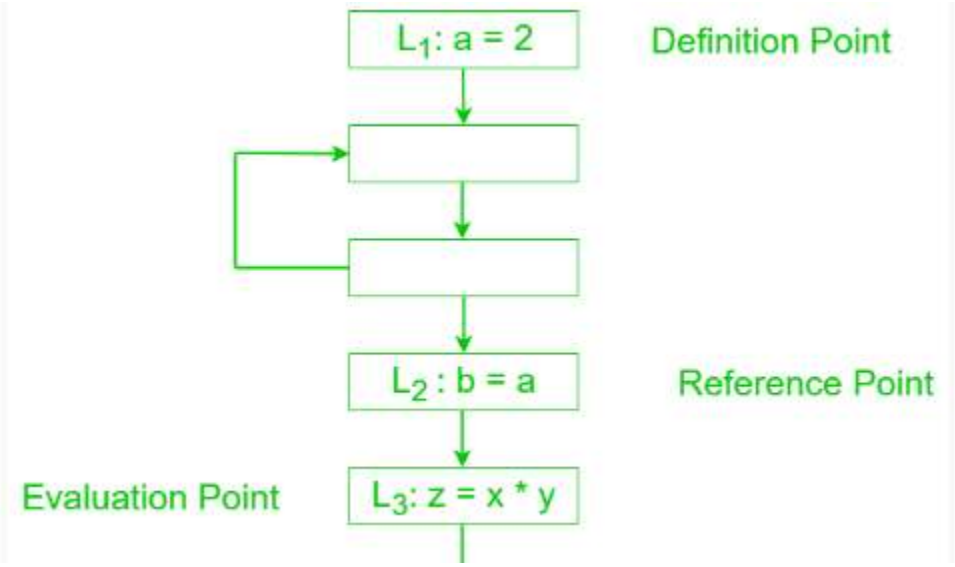
These safe values may be either subsets or supersets of actual values, based on the application

Basic Terminologies –

Definition Point: a point in a program containing some definition.

Reference Point: a point in a program containing a reference to a data item.

Evaluation Point: a point in a program containing evaluation of expression.



4.4.3 Transformations using dataflow analysis

In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph. A compiler could take advantage of "reaching definitions" , such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

Data-flow information can be collected by setting up and solving systems of equations of the form :

out [S] = gen [S] U (in [S] - kill [S])

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement." Such equations are called data-flow equation.

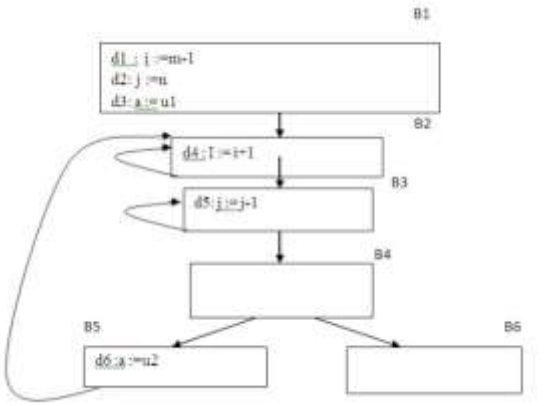
1. The details of how data-flow equations are set and solved depend on three factors. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[S] in terms of in[S], we need to proceed backwards and define in[S] in terms of out[S].

2. Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

1. Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and pi+1 is the beginning of a successor block.

Reaching definitions

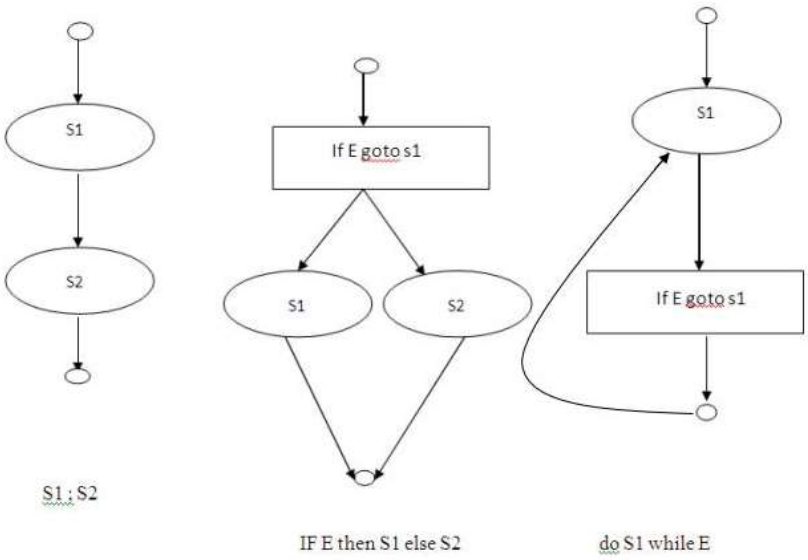
A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x. These

statements certainly define a value for x, and they are referred to as unambiguous definitions of x. There are certain kinds of statements that may define a value for x; they are called ambiguous definitions.

The most usual forms of ambiguous definitions of x are:

1. A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
2. An assignment through a pointer that could refer to x. For example, the assignment *q:=y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the appearing later along one path.



Data-flow analysis of structured programs:

Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

S->id: = E| S; S | if E then S else S | do S while E

E->id + id| id

Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

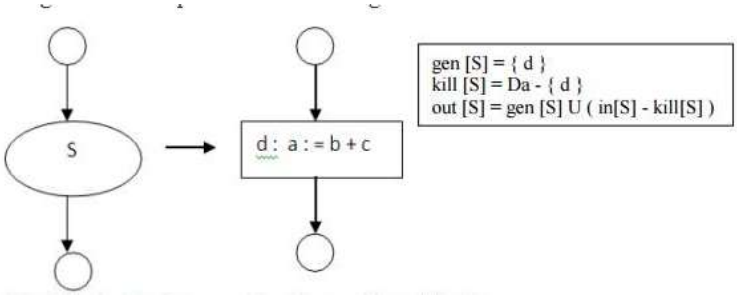
We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the

header. The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

We say that the beginning points of the dummy blocks at the statement's region are the beginning and end points, respective equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S. gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.

Consider the following data-flow equations for reaching definitions :

i)



Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

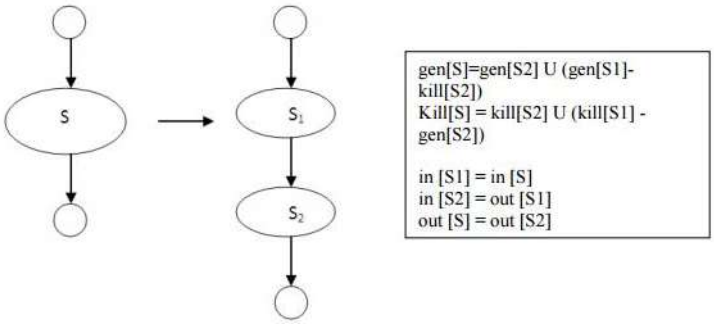
gen[S]={d}

On the other hand, d "kills" all other definitions of a, so we write

Kill[S] = Da - {d}

Where, Da is the set of all definitions in the program for variable a.

ii)



Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write

gen[S]=gen[S2] U (gen[S1]-kill[S2])

Similar reasoning applies to the killing of a definition, so we have

Kill[S] = kill[S2] U (kill[S1] - gen[S2])

Conservative estimation of data-flow information:

There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.

We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input. When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

Many data-flow problems can be solved by synthesized translation to compute gen and kill. It can be used, for example, to determine computations. However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S. Assuming we know in[S] we compute out by equation, that is

Out[S] = gen[S] U (in[S] - kill[S])

Considering cascade of two statements S1; S2, as in the second case. We start by observing in[S1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].

Consider the if-statement. we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S. That is,

in[S1] = in[S2] = in[S]

If a definition reaches the end of S if and only if it reaches the end of one or both sub-statements; i.e,

out[S]=out[S1] U out[S2]

Representation of sets:

Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implement complement of B and then using logical and to compute A

Local reaching definitions:

Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B

is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred. Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax directed manner. When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods. However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

4.4.4 Speeding up dataflow analysis

There are several ways to speed up the evaluation of dataflow equations.

4.4.4.1 Bit vectors

- Many dataflow analyses can be expressed using simultaneous equations on finite sets.
- A set S over a finite domain can be represented by a bit vector.
- The ith bit in the vector is a 1 if the element i is in the set S.
- In the bit-vector representation,

    1. unioning two sets S and T is done by a bitwise-or of the bit vectors,
    2. intersection can be done by bitwise-and,
    3. set complement can be done by bitwise complement, and so on.

If the word size of the computer is W, and the vectors are N bits long, then such a merging operation needs a sequence of N/W instructions.

- Of course, 2N/W fetches and N/W stores will also be necessary, as well as indexing and loop overhead.

It would be inadvisable to use bit vectors for dataflow problems where the sets are expected to be very sparse (so the bit vectors would be almost all zeros), in which case a different implementation of sets would be faster.

4.4.4.2 Basic blocks

- Suppose we have a node n in the flow graph that has only one predecessor, p, and p has only one successor, n.
- we can combine the gen and kill effects of p and n and replace nodes n and p with a single node.
- Such a single node is called a basic block.
- A basic block is a sequence of statements that is always entered at the beginning and exited at the end, that is:
  1. The first statement is a label.
  2. The last statement is a jump or cjump.
  3. There are no other labels, jumps, or cjumps.
- The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The sequence is scanned from beginning to end;
  1. whenever a label is found, a new block is started (and the previous block is ended);
  2. whenever a jump or cjump is found, a block is ended (and the next block is started).
  3. If this leaves any block not ending with a jump or cjump, then a jump to the next block's label is appended to the block.
  4. If any block has been left without a label at the beginning, a new label is invented and stuck there.

  We introduce a new label done which mean the beginning of the epilogue, and put a jump(name done) at the end of the last block.
- Taking reaching definitions as an example, we can combine all the statements of a basic block as follows:
  1. Consider what definitions reach out of the node n:

     $out[n] = gen[n] \cup (in[n] - kill[n])$
  2. We know in[n] is just out[p]; therefore

     $out[n] = gen[n] \cup ((gen[p] \cup (in[p] - kill[p])) - kill[n])$

3. By using the identity $(A \cup B) - C = (A - C) \cup (B - C)$ and then $(A - B) - C = A - (B \cup C)$, we have

   $out[n] = gen[n] \cup (gen[p] - kill[n]) \cup (in[p] - (kill[p] \cup kill[n]))$

4. If we regard p and n as a single node pn, the appropriate gen and kill sets for pn are:

   $gen[pn] = gen[n] \cup (gen[p] - kill[n])$

   $kill[pn] = kill[p] \cup kill[n]$

   The control-flow graph of basic blocks is much smaller than the graph of individual statements.

- The multipass iterative dataflow analysis works much faster on basic blocks.
- Once the iterative dataflow analysis algorithm is completed, we may recover the dataflow information of an individual statement (such as n) within a block (such pn in our example) as follows:

  1. start with the in set computed for the entire block and,
  2. apply the gen and kill sets of the statements that precede n in the block.

4.4.4.3 Ordering the nodes

- If we could arrange that every node was calculated before its successors in a forward dataflow problem, the dataflow analysis would terminate in one pass through the nodes.
  - This would be possible if the control-flow graph had no cycles.
- quasi-topologically sorting a cyclic graph by depth-first search helps to reduce the number of iterations required on cyclic graphs; in quasi-sorted order, most nodes come before their successors.
  - Information flows forward quite far through the equations on each iteration.

  1. Depth-first search topologically sorts an acyclic graph graph, or quasi-topologically sorts a cyclic graph, quite efficiently.
  2. Using sorted, the order computed by depth-first search, the iterative solution of dataflow equations should be computed as

```
Topological-sort:
N ← number of nodes
for all nodes i
    mark[i] ← false
DFS(start-node)

function DFS(i)
    if mark[i] = false
        mark[i] ← true
        for each successor s of node i
            DFS(s)
        sorted[N] ← i
        N ← N − 1
```

Figure 11: Algorithm: Topological sort by depth-first search.

```
repeat
    for i ← 1 to N
        n ← sorted[i]
        in ← ⋃_{p∈pred[n]} out[p]
        out[n] ← gen[n] ∪ (in − kill[n])
until no out set changed in this iteration
```

There is no need to make in a global array, since it is used only locally in computing out.

- For backward dataflow problems such as liveness analysis, we use a version of Algorithm 11 starting from exit-node instead of start-node, and traversing predecessor instead of successor edges.

```
W ← the set of all nodes
while W is not empty
    remove a node n from W
    old ← out[n]
    in ← ⋃_{p∈pred[n]} out[p]
    out[n] ← gen[n] ∪ (in − kill[n])
    if old ≠ out[n]
        for each successor s of n
            if s ∉ W
                put s into W
```

4.4.4.4 Work-list algorithms

- If any out set changes during an iteration of the repeat-until loop of an iterative solver, then all the equations are recalculated.

  Most of the equations may not be affected by the change.

- A work-list algorithm keeps track of just which out sets must be recalculated.

Whenever node n is recalculated and its out set is found to change, all the successors of n are put onto the work-list (if they're not on it already).

4.4.4.5 Use-def and def-use chains

- Use-def chains: a list of the definitions of x reaching that use for each use of a variable x.

  Information about reaching definitions can be kept as use-def chains,

  Use-def chains do not allow faster dataflow analysis per se, but allow efficient implementation of the optimization algorithms that use the results of the analysis.

- A generalization of use-def chains is static single-assignment form.

  SSA form not only provides more information than use-def chains, but the dataflow analysis that computes it is very efficient.

- Def-use chains: a list of all possible uses of that definition for each definition.

  SSA form also contains def-use information.

4.4.4.6 Wordwise approach

- Bit vector approach + basic block approach + worklist approach + wordwise
- Wordwise approach deals with the largest chunk of a bit vector which can be processed in one machine operation.
  - Typically, a machine word

1. Select a word.
4. Process it over required area of a control flow graph

- Wordwise approach results in considerable savings in the work to be performed since all parts may not require processing for all nodes of the control flow graph.

4.5 ALIAS ANALYSIS

If two or more expressions denote the same memory address we can say that the expressions are aliases of each other.

How do aliases arise?

- Pointers
- Call by reference (parameters can alias each other or non-locals)
- Array indexing
- C union, Pascal variant records, Fortran EQUIVALENCE and COMMON blocks

Alias analysis techniques are usually classified by flow-sensitivity and context-sensitivity. They may determine may-alias or must-alias information. The term alias analysis is often used interchangeably with points-to analysis, a specific case.

Alias analysers intend to make and compute useful information for understanding aliasing in programs.

Example code:

p.foo = 1;

q.foo = 2;

i = p.foo + 3;

There are three possible alias cases here:

The variables p and q cannot alias (i.e., they never point to the same memory location).

The variables p and q must alias (i.e., they always point to the same memory location).

It cannot be conclusively determined at compile time if p and q alias or not.

If p and q cannot alias, then i = p.foo + 3; can be changed to i = 4. If p and q must alias, then i = p.foo + 3; can be changed to i = 5 because p.foo + 3 = q.foo + 3. In both cases, we are able to perform optimizations from the alias knowledge (assuming that no other thread updating the same locations can interleave with the current thread, or that the language memory model permits those updates to be not immediately visible to the current thread in absence of explicit synchronization constructs).

On the other hand, if it is not known if p and q alias or not, then no optimizations can be performed and the whole of the code must be executed to get the result. Two memory references are said to have a may-alias relation if their aliasing is unknown.

In alias analysis, we divide the program's memory into alias classes. Alias classes are disjoint sets of locations that cannot alias to one another. For the discussion here, it is assumed that the optimizations done here occur on a low-level intermediate representation of the program. This is to say that the program has been compiled into binary operations, jumps, moves between registers, moves from registers to memory, moves from memory to registers, branches, and function calls/returns.

There are two ways for Alias Analysis:

Type-based alias analysis

Flow-based alias analysis

4.6 SUMMARY

An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program. Common requirements are to minimize a program's execution time, memory footprint, storage size, and power consumption (the last three being popular for portable computers).

Compiler optimization is generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources or executes faster.

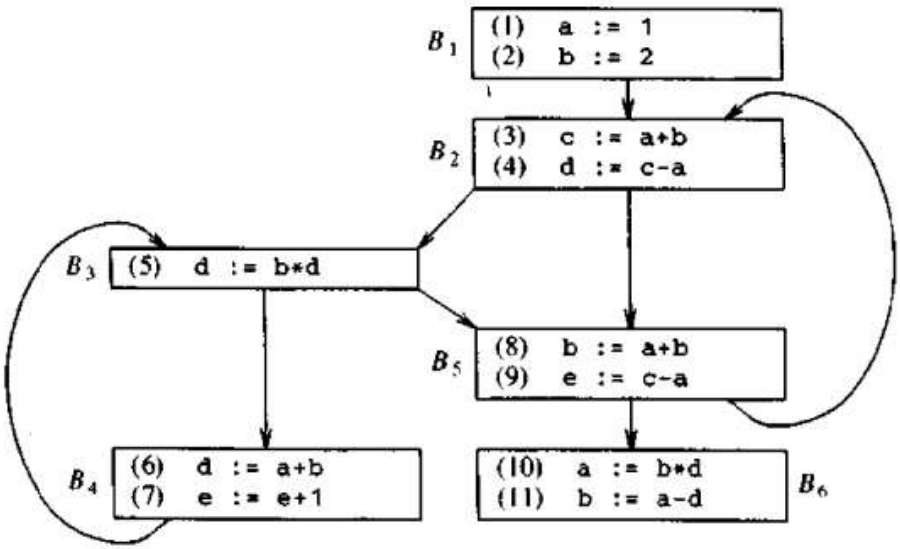
Data-flow optimizations, based on data-flow analysis, primarily depend on how certain properties of data are propagated by control edges in the control-flow graph.

We have learnt about various techniques of loop optimization and dataflow analysis as applicable for compiler design.

4.7 BIBLIOGRAPHY

Compilers – Principles, Techniques and Tools. By – Alfred Aho, Ravi Sethi, Jeffrey D. Ulman

4.8 UNIT END EXERCISES



1. For the above graph find the live expressions at the end of each block.
2. For the above graph find the available expressions
3. Are there any expressions which may be hoisted in above example, if so hoist them
4. Is there any constant folding possible in above graph. If so, do it.
5. Eliminate any common sub-expressions in the above figure
6. In the above figure, what is the limit flow graph? Is the flow graph reducible.
7. Give an algorithm in time O(n) on an n-node flow graph to find the extended basic block ending at each node.

❄❄❄❄❄❄❄