MOTHER TERESA WOMEN'S UNIVERSITY

DIRECTORATE OF DISTANCE EDUCATION

KODAIKANAL – 624 102

M.Sc COMPUTER SCIENCE

(P.G)

MATERIAL FOR COMPILER DESIGN

SUBJECT CODE

Compiled by Dr. K.Kavitha

COMPILER DESIGN

UNIT – 1: Introduction to Compiling : Compilers – Analysis of Source Program – The Phases of a Compiler –Compiler Construction tool – **A Simple One – pass Compiler**: Overview Syntax Definition – Syntax Directed Translation – Parsing – Lexical Analysis.

UNIT – II: Lexical Analysis: The Role of the Lexical Analyser – Input Buffering – Specification of Token – Recognition of Token –Finite Automata. Syntax Analysis: The Role of the parser – Context – free grammars – Top down Parsing – Bottom – up Parsing.

UNIT – III: Syntax – Directed Translation: Syntax – Directed definitions – Construction of Syntax trees – L-attributed definitions – Top down translation – Bottom – up evaluation of inherited attributes.

Type Checking: Type system- Specification of a simple type checker – Type conversion.

UNIT –IV: Intermediate code generation: Intermediate Language – Declaration – Assignment Statement – Boolean Expression – Case Statement – Back Batching – Procedure calls.

Code Generation: Issues in the design of a code generation – Run time storage management – Basic Blocks and flow graphs – A simple code generator – Register allocation and assignment.

UNIT – V: Code optimization: Introduction – The Principle sources of optimization – Optimization of basic blocks – Loops in flow graphs – Introduction to global data – flow analysis – Iterative solution of data flow equations – code improving transformation – Data – flow analysis of structured flow graphs – Efficient data – flow algorithms.

TEXT BOOK

1. Compilers Principles Techniques and Tools by Alfred V Aho Ravi Sethi Jeffrey D.Ullman, Published by Pearson Education.

UNIT 1

INTRODUCTION TO COMPILING

Introduction of language processing system

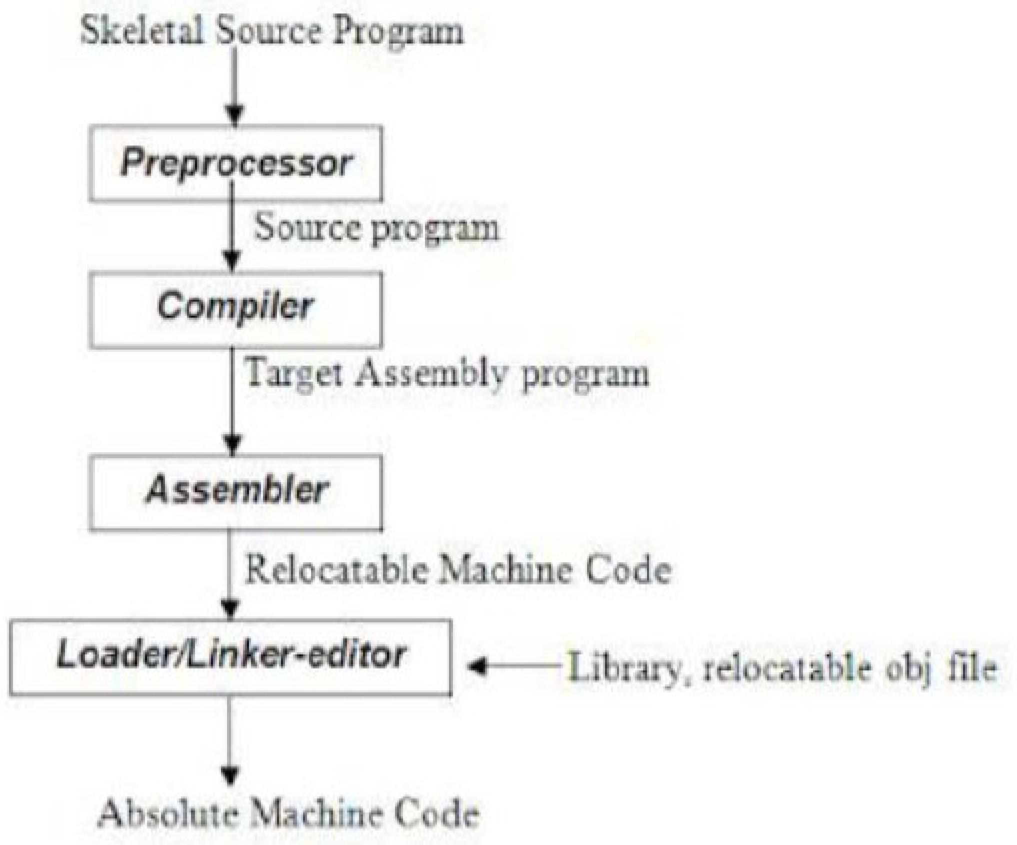


Fig 1.1: Language Processing System

Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

Macro processing: A preprocessor may allow a user to define macros that are short hands for
longer constructs.

File inclusion: A preprocessor may include header files into the program text.

Rational preprocessor: These preprocessors augment older languages with more modern flow-of control and data structuring facilities.

Language Extensions: These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

Compiler

The compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer. This translation process is called compilation. After compiling once, the program can continue to be run from its binary form without compiling again. This results in more efficient and faster execution of the code.

Example: C, C++, Cobol and Fortran

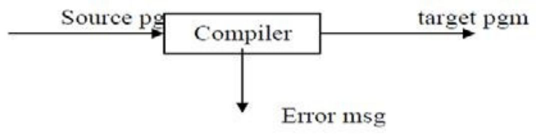


Fig 1.1: Structure of Compiler

HLL programming language basically consists of two parts. The source program must be compiled translated into an object program. Then the results object program is loaded into a memory executed.

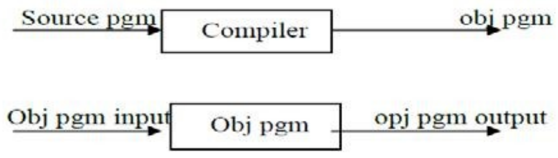


Fig 1.2: Execution process of the source program in Compiler

Analysis of Source Program:

In Compiling, analysis consists of three phases:

- Linear Analysis: In which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning.
- Hierarchical Analysis: In which characters or tokens are grouped hierarchically in to nested collections with collective meaning.
- Semantic Analysis: In which certain checks are performed to ensure that the components of a program fit together meaningfully.

The Phases of a Compiler:

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation

that takes source program in one representation and produces output in another representation. There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)

b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **‘phases’**. The phases of a compiler are shown in below.

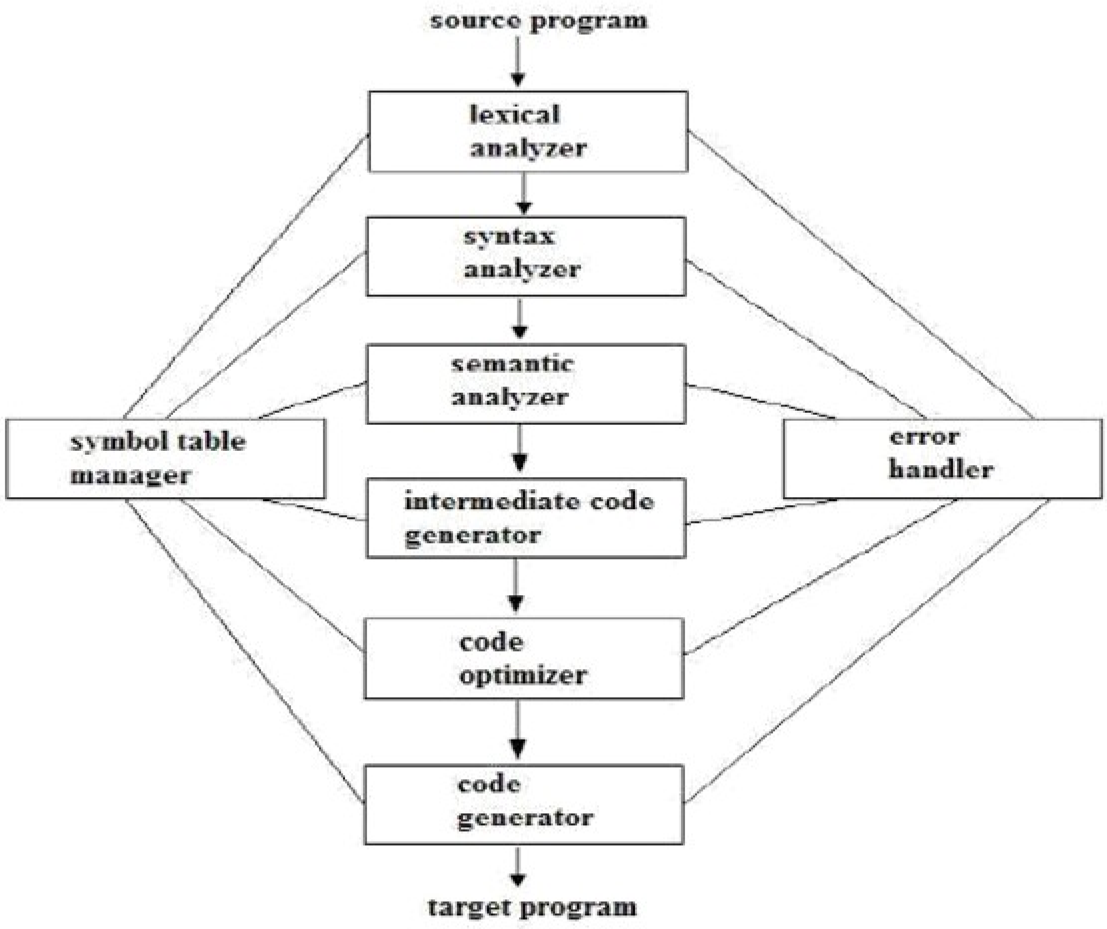


Fig 1.3:Phases of a Compiler

Lexical Analysis:-

Lexical Analysis or Scanners reads the source program one character at a time, carving the source program into a sequence of automatic units called tokens. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.

Syntax Analysis:-

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic analysis:-

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generations:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands. The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization :-

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length ofmachine language program** are carried out during this phase. The output of the code generator isthe machine language program of the specified computer.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Error Handlers:-

It is invoked when a flaw error in the source program is detected. The output of Lexical Analysis is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The Syntax Analysis groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that arepermitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression mightappear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression. Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)
1, divide A by B and then multiply by C or
2, multiply B by C and then use the result to divide A.
each of these two interpretations can be represented in terms of a parse tree.

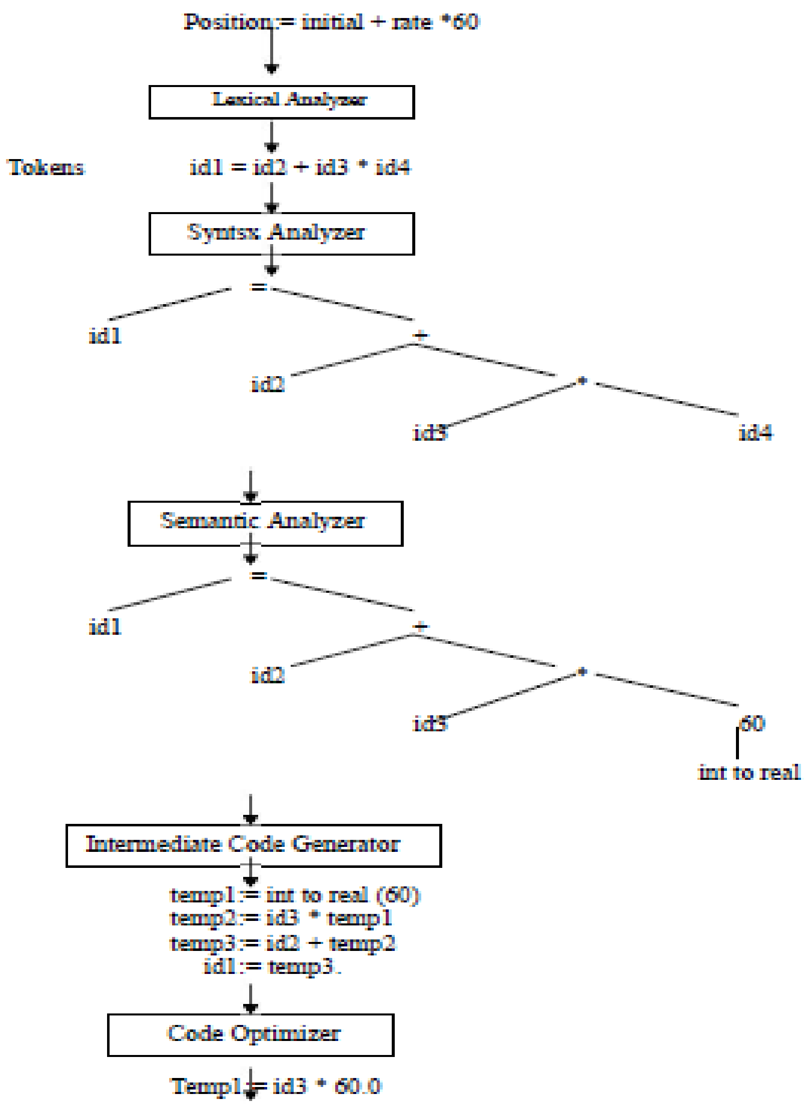


Figure 1.4 Compilation process of source code through phases

Compiler Construction tools –

Compiler construction tools were introduced as computer-related technologies spread all over the world. They are also known as a compiler- compilers, compiler-generators or translator.

These tools use specific language or algorithm for specifying and implementing the component of the compiler.

- **Scanner generators**: This tool takes regular expressions as input. For example LEX for Unix Operating System.
- **Syntax-directed translation engines**: These software tools offer an intermediate code by using the parse tree. It has a goal of associating one or more translations with each node of the parse tree.

- **Parser generators:** A parser generator takes a grammar as input and automatically generates source code which can parse streams of characters with the help of a grammar.
- **Automatic code generators**: Takes intermediate code and converts them into Machine Language
- **Data-flow engines**: This tool is helpful for code optimization. Here, information is supplied by user and intermediate code is compared to analyze any relation. It is also known as data-flow analysis. It helps you to find out how values are transmitted from one part of the program to another part.

A Simple One – pass Compiler:

In computer programming, a **one-pass compiler** is a compiler thatpasses through the parts of each compilation unit only once, immediately translating each part into its final machine code. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

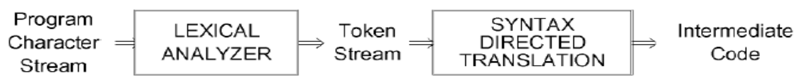
Overview Syntax Definition

Language Definition

- Appearance of programming language

  Vocabulary : Regular expression

  Syntax : Backus-Naur Form(BNF) or Context Free Form(CFG)
- Semantics : Informal language or some examples



Syntax

❖ To specify the syntax of a language : CFG and BNF

- **Example :**if-else statement in C has the form of statement → if (expression) statement else statement

❖ An alphabet of a language is a set of symbols.

- **Examples**: {0,1} for a binary number system(language)={0,1,100,101,...}

  {a,b,c} for language={a,b,c, ac,abcc..}

{if,(,),else ...} for a if statements={if(a==1)goto10, if--}

- A string over an alphabet
  - is a sequence of zero or more symbols from the alphabet.
  - Examples : 0,1,10,00,11,111,0202 ... strings for a alphabet {0,1}
  - Null string is a string which does not have any symbol of alphabet.
- Language
  - Is a subset of all the strings over a given alphabet.

oAlphabets Ai	Languages Li for Ai
A0={0,1}	L0={0,1,100,101,...}
A1={a,b,c}	L1={a,b,c, ac, abcc..}
A2={all of C tokens}	L2= {all sentences of C program }

- Example :Grammar for expressions consisting of digits and plus and minus signs.
  - Language of expressions L={9-5+2, 3-1, ...}
  - The productions of grammar for this language L are:

list → *list* + *digit*

list → *list* - *digit*

list → *digit*

digit → *0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

  - list, digit : Grammar variables, Grammar symbols.
  - **0,1,2,3,4,5,6,7,8,9,-,+** : Tokens, Terminal symbols.
- Convention specifying grammar
  - Terminal symbols : bold face string **if, num, id**
  - Nonterminal symbol, grammar symbol : italicized names, list, digit ,A,B
- Grammar G=(N,T,P,S)
  - N : a set of nonterminal symbols
  - T : a set of terminal symbols, tokens
  - P : a set of production rules
  - S : a start symbol, $S \in N$
- Grammar G for a language L = { 9-5+2, 3-1,….}
  - G=(N,T,P,S)
  - N={list,digit}
  - T={0,1,2,3,4,5,6,7,8,9,-,+}
  - P= *list* → *list* + *digit*

list → *list* - *digit*

list → digit

digit → 0|1|2|3|4|5|6|7|8|9

❖ Some definitions for a language L and its grammar G

- Derivation :

  A sequence of replacements S⇒α1⇒α2⇒…⇒αn is a derivation of αn.

  Example, A derivation 1+9 from the grammar G

  - left most derivation

    list⇒list + digit ⇒digit + digit ⇒1 + digit ⇒1 + 9

  - right most derivation

    list⇒list + digit ⇒list + 9 ⇒digit + 9 ⇒1 + 9

- Language of grammar L(G)

  L(G) is a set of sentences that can be generated from the grammar G.

  L(G)={x| S ⇒* x} where x ∊a sequence of terminal symbols

- Example: Consider a grammar G=(N,T,P,S):

  N={S} T={a,b}

  S=S P ={S → aSb | ε }

  - isaabb a sentecne of L(g)? (derivation of string aabb)

    S⇒aSb⇒aaSbb⇒aaεbb⇒aabb(or S⇒* aabb) so, aabbεL(G)

  - there is no derivation for aa, so aa∉L(G)
  - note L(G)={anbn| n≧0} where anbnmeas n a's followed by n b's.

❖ **Parse Tree**

A derivation can be conveniently represented by a derivation tree(parse tree).

- The root is labeled by the start symbol.

oEach leaf is labeled by a token orε.

oEach interior none is labeled by a nonterminal symbol.

oWhen a production A→x1… xn is derived, nodes labeled by x1… xn are made aschildren

nodes of node labeled by A.

- root : the start symbol
- internal nodes : nonterminal
- leaf nodes : terminal

- Example :

  list -> list + digit | list - digit | digit

digit -> 0|1|2|3|4|5|6|7|8|9

- left most derivation for 9-5+2,

  list⇒list+digit⇒list-digit+digit⇒digit-digit+digit⇒9-digit+digit

  ⇒9-5+digit ⇒9-5+2

- right most derivation for 9-5+2,

  list⇒list+digit⇒list+2⇒list-digit+2 ⇒list-5+2⇒digit-5+2 ⇒9-5+2

parse tree for 9-5+2

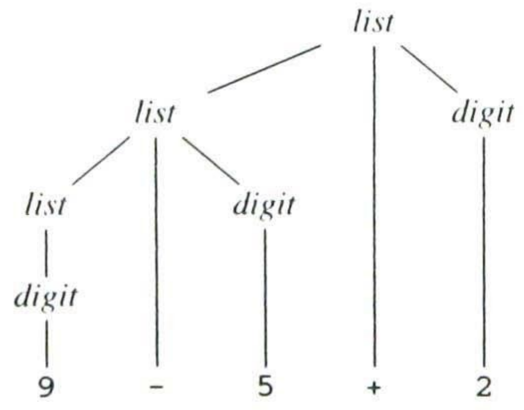


Fig 1.6. Parse tree for 9-5+2 according to the grammar in Example

Ambiguity

❖ A grammar is said to be ambiguous if the grammar has more than one parse tree for a given string of tokens

❖ Example . Suppose a grammar G that can not distinguish between lists and digits as in above example

- o G : string → string + string | string - string |0|1|2|3|4|5|6|7|8|9

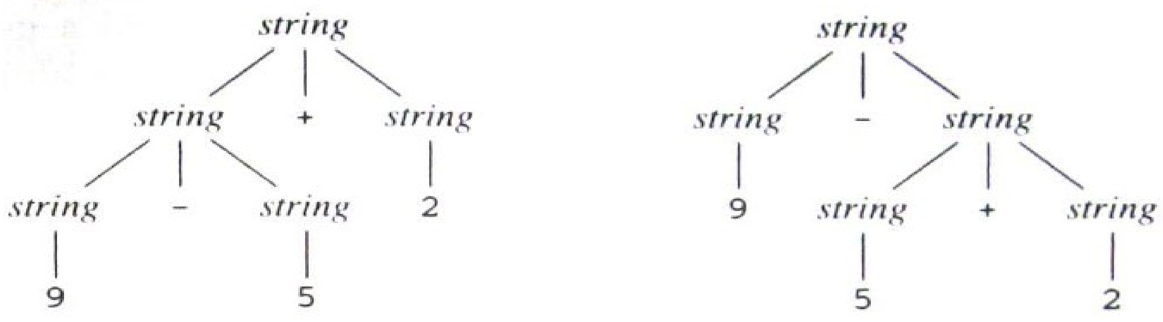


Fig 1.7. Two Parse tree for 9-5+2

- o 1-5+2 has 2 parse trees => Grammar G is ambiguous.

Associativity of operator

A operator is said to be left associative if an operand with operators on both sides of it is taken by the operator to its left.

Example: 9+5+2≡(9+5)+2, a=b=c≡a=(b=c)

- Left Associative Grammar :

  list → list + digit | list - digit

  digit →0|1|…|9

- Right Associative Grammar :

  right → letter = right | letter

  letter → a|b|…|z

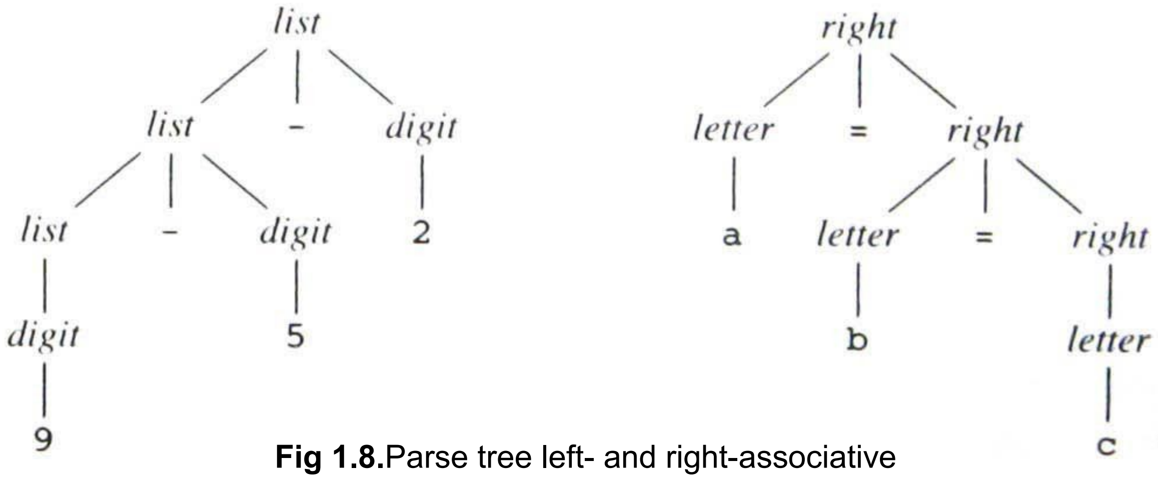


Fig 1.8. Parse tree left- and right-associative operators.

Precedence of operators

We say that aoperator(*) has higher precedence than other operator(+) if the operator(*) takes operands before other operator(+) does.

- Example: 9+5*2≡9+(5*2), 9*5+2≡(9*5)+2
- left associative operators : + , - , * , /
- right associative operators : = , **
- Syntax of full expressions

operator	Associative	precedence
+ , -	Left	1 low
* , /	Left	2 heigh

- *expr → expr + term | expr - term | term*

  *term → term * factor | term / factor | factor*

  factor → digit | (expr)

  digit → 0 | 1 | … | 9
- Syntax of statements
  - stmt → id = expr ;

    | if (expr) stmt ;

    | if (expr) stmt else stmt ;

    | while (expr) stmt ;

    expr → expr + term | expr - term | term

    term → term * factor | term / factor | factor

    factor → digit | (expr)

    digit → 0 | 1 | … | 9

SYNTAX-DIRECTED TRANSLATION(SDT)

A formalism for specifying translations for programming language constructs.

(attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation

Postfix notation for an expression E

- If E is a variable or constant, then the postfix nation for E is E itself (E.t≡E).
- if E is an expression of the form E1 op E2 where op is a binary operator
  - E1' is the postfix of E1, o E2' is the postfix of E2
  - then E1' E2' op is the postfix for E1 op E2
  - if E is (E1), and E1' is a postfix

    then E1' is the postfix for E

eg) $9 - 5 + 2 \Rightarrow 9\ 5\ -\ 2\ +$

$9 - (5 + 2) \Rightarrow 9\ 5\ 2\ +\ -$

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively for translation.
- A translation is an input-output mapping procedure for translation of an input X,

oConstruct a parse tree for X.

oSynthesize attributes over the parse tree.

- o Suppose a node n in parse tree is labeled by X and X.a denotes the value of attribute a of X at that node.

o Compute X's attributes X.a using the semantic rules associated with X.

Example : SDD for infix to postfix translation

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel$ '+'
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel$ '-'
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t :=$ '0'
$term \rightarrow 1$	$term.t :=$ '1'
. . .	. . .
$term \rightarrow 9$	$term.t :=$ '9'

Fig 1.9. Syntax-directed definition for infix to postfix translation.

An example of synthesized attributes for input X=9-5+2

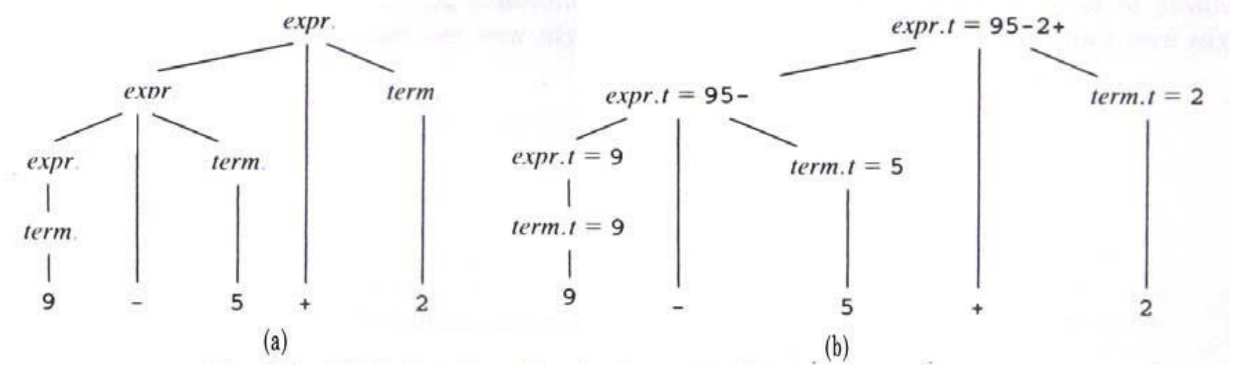


Fig 2.0 Attribute values at nodes in a parse tree.

Syntax-directed Translation Schemes(SDTS)

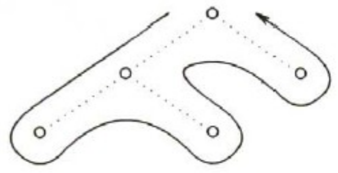
- A translation scheme is a context-free grammar in which program fragments called translation actions are embedded within the right sides of the production.

productions(postfix)	SDD for postfix to infix notation	SDTS
list → list + term	list.t = list.t || term.t || "+"	list → list + term { print("+")}

- {print("+");} : translation(semantic) action.
- SDTS generates an output for each sentence x generated by underlying grammar by executing actions in the order they appear during depth-first traversal of a parse tree for x.
  - 1. Design translation schemes(SDTS) for translation
  - 2. Translate :

    a)parse the input string x and

    b)emit the action result encountered during the depth-first traversal of parse tree.



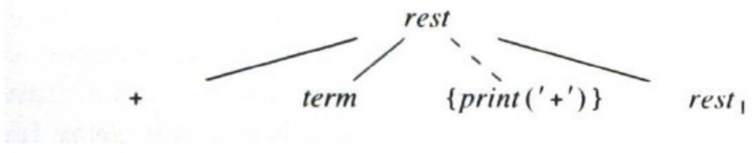


Fig 1.10 Example of a depth-first traversal of a tree. **Fig 1.11**An extra leaf is constructed for a semantic action.

Example

- SDD vs. SDTS for infix to postfix translation.

Productions	SDD	SDTS

expr → list + term
expr → list + term
expr → term
term → 0
term → 1
…
	expr.t = list.t || term.t || "+"
	expr.t = list.t || term.t || "-"
	expr.t = term.t
	term.t = "0"
	term.t = "1"
	…
		expr → list + term printf{"+")}
		expr → list + term printf{"-")}
		expr → term
		term → 0 printf{"0")}
		term → 1 printf{"1")}

term → 9	term.t = "9"	… term → 9 printf{"0")

- Action translating for input 9-5+2

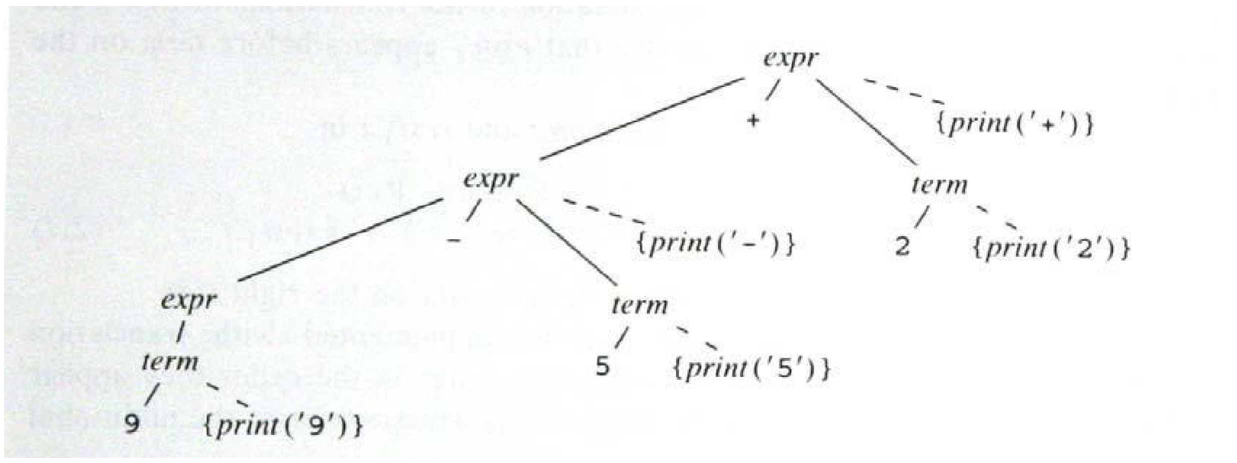


Fig 1.12. Actions translating 9-5+2 into 95-2+.

- 1) Parse.
- 2) Translate.

Do we have to maintain the whole parse tree ?

No, Semantic actions are performed during parsing, and we don't need the nodes (whose semantic actions done).

PARSING

if token string x ∈ L(G), then parse tree else error message

Top-Down parsing

1. At node n labeled with nonterminal A, select one of the productions whose left part is A and construct children of node n with the symbols on the right side of that production.
2. Find the next node at which a sub-tree is to be constructed.

Example. G: type → simple

|↑id

|array [simple] of type

simple → integer

|char

|numdotdotnum

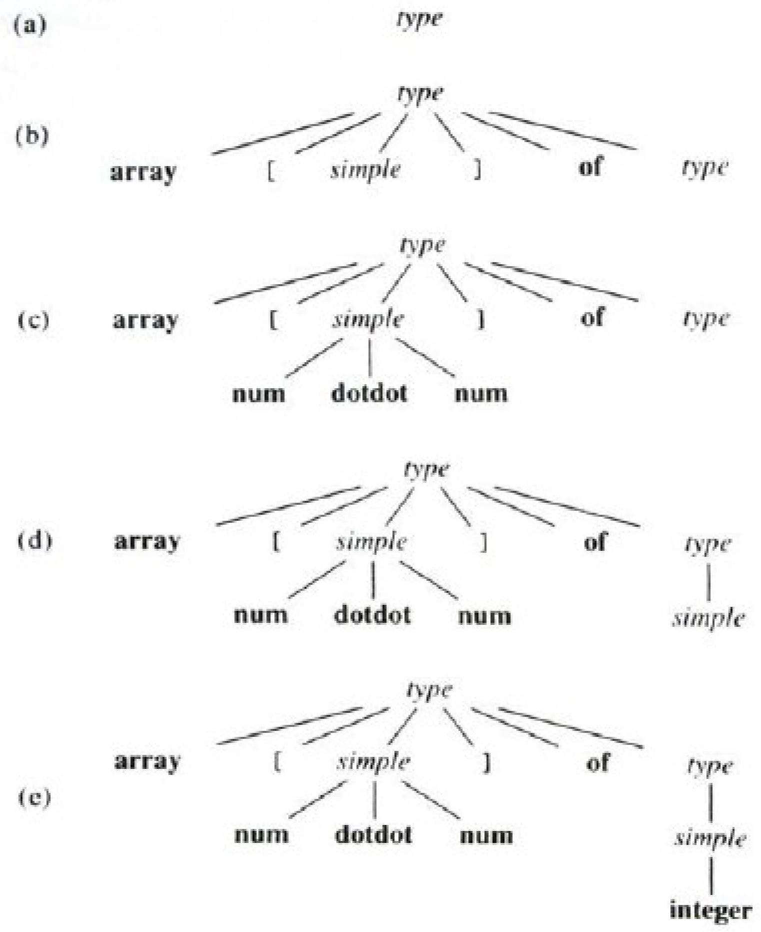


Fig 1.13. Steps in the top-down construction of a parse tree.

The selection of production for a nonterminal may involve trial-and-error.

=>backtracking

• G : { S->aSb | c | ab }

According to topdown parsing procedure, acb ,aabb∈L(G)?

• S/acb⇒aSb/acb⇒aSb/acb⇒aaSbb/acb⇒X

(S→aSb) move (S→aSb) backtracking

⇒aSb/acb⇒acb/acb⇒acb/acb⇒acb/acb

(s→c) move move

so, acb∈L(G)

Is is finished in 7 steps including one backtracking.

• S/aabb⇒aSb/aabb⇒aSb/aabb⇒aaSbb/aabb⇒aaSbb/aabb⇒aaaSbbb/aabb⇒X

(S→aSb) move (S→aSb) move (S→aSb) backtracking

⇒aaSbb/aabb⇒aacbb/aabb⇒X

(S→c) backtracking

⇒aaSbb/aabb⇒aaabbb/aabb⇒X

(S→ab) backtracking

⇒aaSbb/aabb⇒X

backtracking

⇒aSb/aabb⇒acb/aabb

(S→c) bactracking

⇒aSb/aabb⇒aabb/aabb⇒aabb/aabb⇒aabb/aabb⇒aaba/aabb

(S→ab) move movemove

so, aabb∈L(G)

but process is too difficult. It needs 18 steps including 5 backtrackings.

• procedure of top-down parsing

let a pointed grammar symbol and pointed input symbol be g, a respectively.

- if(g ∈N) select and expand a production whose left part equals to g next to current production.

  else if(g = a) then make g and a be a symbol next to current symbol.

  else if(g ≠a) back tracking.

  ¬ let the pointed input symbol a be the symbol that moves back to steps same with the number of current symbols of underlying production

  ¬ eliminate the right side symbols of current production and let the pointed

  symbol g be the left side symbol of current production.

Predictive parsing (Recursive Decent Parsing,RDP)

• A strategy for the general top-down parsing

Guess a production, see if it matches, if not, backtrack and try another.

• It may fail to recognize correct string in some grammar G and is tedious in processing.

• **Predictive parsing**

- is a kind of top-down parsing that predicts a production whose derived terminalsymbol is equal to next input symbol while expanding in top-down paring.
- without backtracking.

- Procedure decent parser is a kind of predictive parser that is implemented bydisjoint recursive procedures one procedure for each nonterminal, the proceduresare patterned after the productions.

• procedure of predictive parsing(RDP)

let a pointed grammar symbol and pointed input symbol be g, a respectively.

- if(g ∈N)
  - ¬ select next production P whose left symbol equals to g and a set of firstterminal symbols of derivation from the right symbols of the production P

    includes a input symbol a.

  - ¬ expand derivation with that production P.
- else if(g = a) then make g and a be a symbol next to current symbol.
- else if(g ≠a) error

• G : { S→aSb | c | ab } => G1 : { S->aS' | c S'->Sb | ab }

According to predictive parsing procedure, acb ,aabb∈L(G)?

- S/acb⇒confused in { S→aSb, S→ab }
- so, a predictive parser requires some restriction in grammar, that is, there shouldbe only one production whose left part of productions are A and each firstterminal symbol of those productions have unique terminal symbol.

• Requirements for a grammar to be suitable for RDP: For each nonterminal either

1. A → Bα, or
2. A → a1α1 | a2α2 | … | anαn

    1) for 1 ≦i, j ≦n and i≠ j, ai ≠ aj

    2) A ε may also occur if none of ai can follow A in a derivation and if we have A→ε

• If the grammar is suitable, we can parse efficiently without backtrack.

General top-down parser with backtracking

↓

Recursive Descent Parser without backtracking

↓

Picture Parsing (a kind of predictive parsing) without backtracking

Left Factoring

• If a grammar contains two productions of form

S→ aα and S → aβ

it is not suitable for top down parsing without backtracking. Troubles of this form cansometimes be removed from the grammar by a technique called the left factoring.

• In the left factoring, we replace { S→ aα, S→ aβ } by

{ S → aS', S'→ α, S'→ β } cf. S→ a(α|β)

(Hopefully α and β start with different symbols)

• left factoring for G { S→aSb | c | ab }

S→aS' | c cf. S(=aSb | ab | c = a (Sb | b) | c) → a S' | c

S'→Sb | b

• A concrete example:

<stmt> → IF <boolean> THEN <stmt> |

IF <boolean> THEN <stmt> ELSE <stmt>

is transformed into

<stmt>→ IF <boolean> THEN <stmt> S'

S' → ELSE <stmt> | ε

• **Example**,

- for G1 : { S→aSb | c | ab }

According to predictive parsing procedure, acb ,aabb∈L(G)?

- S/aabb⇒ unable to choose { S→aSb, S→ab ?}
- According for the feft factored gtrammar G1, acb ,aabb∈L(G)?

  G1 : { S→aS'|c S'→Sb|b} <= {S=a(Sb|b) | c }
- S/acb⇒aS'/acb⇒aS'/acb⇒aSb/acb⇒acb/acb⇒acb/acb⇒acb/acb

  (S→aS') move (S'→Sb⇒aS'b) (S'→c) move move

  so, acb∈ L(G)

  It needs only 6 steps whithout any backtracking.

  cf. General top-down parsing needs 7 steps and I backtracking.
- S/aabb⇒aS'/aabb⇒aS'/aabb⇒aSb/aabb⇒aaS'b/aabb⇒aaS'b/

aabb⇒aabb/aabb⇒⇒

(S→aS') move (S'→Sb⇒aS'b) (S'→aS') move (S'→b) move move

so, aabb∈L(G)

but, process is finished in 8 steps without any backtracking.

cf. General top-down parsing needs 18 steps including 5 backtrackings.

Left Recursion

• A grammar is left recursive iff it contains a nonterminal A, such that

A⇒+ Aα, where is any string.

- Grammar {S→ Sα | c} is left recursive because of S⇒Sα
- Grammar {S→ Aα, A→ Sb | c} is also left recursive because of S⇒Aα⇒Sbα

• If a grammar is left recursive, you cannot build a predictive top down parser for it.

1) If a parser is trying to match S & S→Sα, it has no idea how many times S must be
applied

2) Given a left recursive grammar, it is always possible to find another grammar that
generates the same language and is not left recursive.

3) The resulting grammar might or might not be suitable for RDP.

• After this, if we need left factoring, it is not suitable for RDP.

• Right recursion: Special care/Harder than left recursion/SDT can handle.

Eliminating Left Recursion

Let G be S→ S A | A

Note that a top-down parser cannot parse the grammar G, regardless of the order the productions
are tried.

⇒The productions generate strings of form AA…A

⇒They can be replaced by S→A S' and S'→A S'|ε

Example :

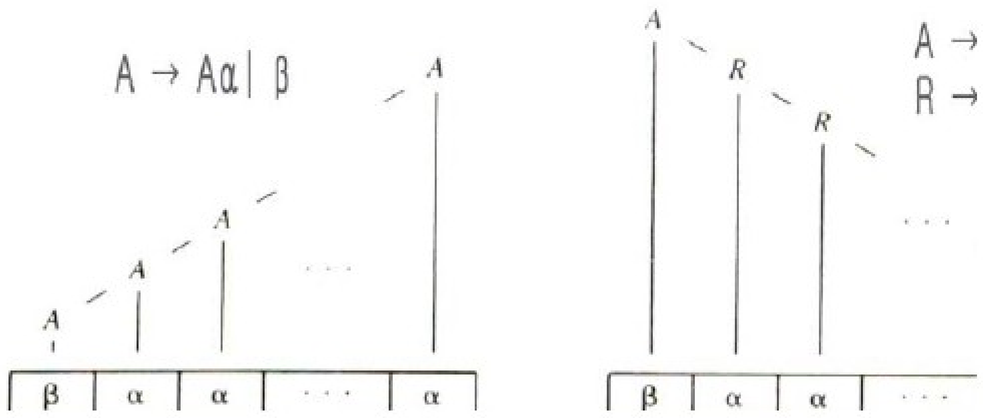
• A → Aα|β

=>

A → βR

R → αR | ε

F



Lexical Analysis:

- reads and converts the input into a stream of tokens to be analyzed by parser.
- lexeme : a sequence of characters which comprises a single token.
- Lexical Analyzer →Lexeme / Token → Parser

Removal of White Space and Comments

- Remove white space(blank, tab, new line etc.) and comments

Contsants

- Constants: For a while, consider only integers
- Example :x for input 31 + 28, output(token representation)?

  input : 31 + 28

  output: <num, 31><+, ><num, 28>

  num + :token

  31 28 : attribute, value(or lexeme) of integer token num

Recognizing

- Identifiers
  - Identifiers are names of variables, arrays, functions...
  - A grammar treats an identifier as a token.
  - eg) input : count = count + increment;

    output : <id,1><=, ><id,1><+, ><id, 2>;

  Symbol table

	tokens	attributes(lexeme)
0		
1	id	count
2	id	increment
3		

- Keywords are reserved, i.e., they cannot be used as identifiers.

Then a character string forms an identifier only if it is no a keyword.

- punctuation symbols
  - o operators : + - * / := <> …

Interface to lexical analyzer

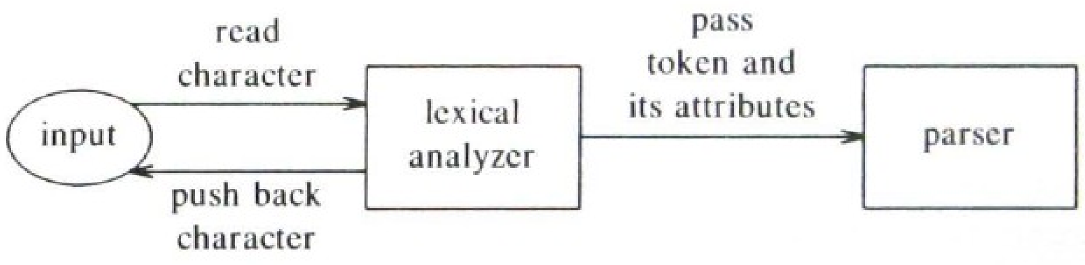


Fig 1.14. Inserting a lexical analyzer between the input and the parser

UNIT - II

LEXICAL ANALYSIS

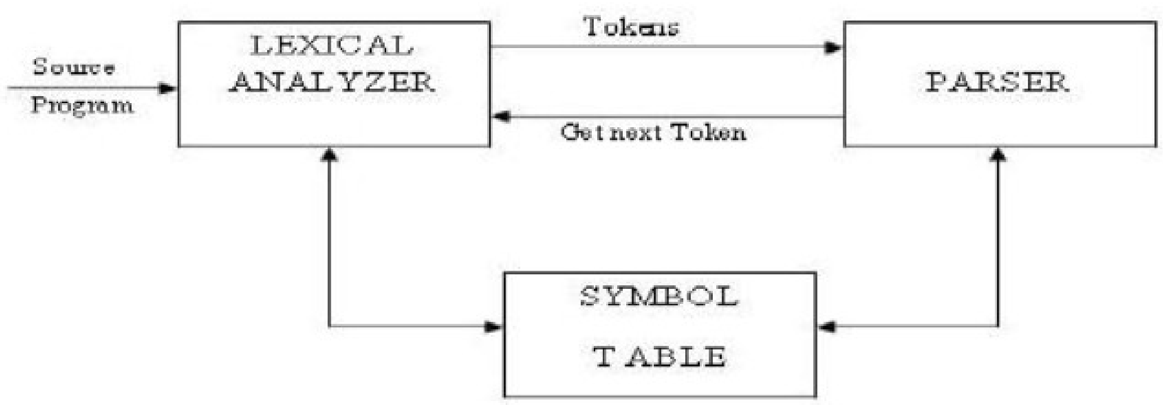
OVERVIEW OF LEXICAL ANALYSIS

oTo identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.

oSecondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

The Lexical Analysis is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The Lexical Analysis return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon. Lexical Analysis may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and newline characters. Another is correlating error message from the compiler with the source program.



TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set
of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the
pattern for a token.

INPUT BUFFERING

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

• Hence a two-buffer scheme is introduced to handle large lookaheads safely.

• Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Specification of Token

An alphabet or a character class is a finite set of symbols. Typical examples of symbols are letters and characters.The set {0, 1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.

Strings

A string over some alphabet is a finite sequence of symbol taken from that alphabet.
For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII computer alphabet. The empty string denoted by ∈, is a special string with zero symbols (i.e., string length is 0).

Languages

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.
Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then
Union: LUM = {dog, ba, na, house}
Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}
Expontentiation: L2 = LL

By definition: L0 ={∈} and L` = L

The kleene closure of language L, denoted by L*, is "zero or more Concatenation of" L.
L* = L0 U L` U L2 U L3 . . . U Ln . . .
For example, If L = {a, b}, then
L* = {∈, a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }

The positive closure of Language L, denoted by L+, is "one or more Concatenation of" L.
L+ = L` U L2 U L3 . . . U Ln . . .
For example, If L = {a, b}, then
L+ = {a, b, aa, ba, bb, aaa, aba, . . . }

Regular Expressions

The regular expressions over alphabet specifies a language according to the following rules.$\in$ is a regular expression that denotes {$\in$}, that is, the set containing the empty string.

If a is a symbol in alphabet, then a is a regular expression that denotes {a}, that is, the set containing the string a.

Suppose r and s are regular expression denoting the languages L(r) and L(s). Then

(r)|(s) is a regular expression denoting L(r) U L(s).

(r)(s) is a regular expression denoting L(r) L(s).

(r)* is a regular expression denoting (L(r))*.

(r) is a regular expression denoting L(r), that is, extra pairs of parentheses may be used around regular expressions.

Unnecessary parenthesis can be avoided in regular expressions using the following conventions:

The unary operator * (kleene closure) has the highest precedence and is left associative.

Concatenation has a second highest precedence and is left associative.

Union has lowest precedence and is left associative.

Regular Definitions

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letter and digits beginning with a letters.

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | 2 | . . . | 9

id → letter (letter | digit)*

The regular expression id is the pattern for the Pascal identifier token and defines letter and digit.

Where letter is a regular expression for the set of all upper-case and lower case letters in the alphabet and digit is the regular for the set of all decimal digits.

The pattern for the Pascal unsigned token can be specified as follows:

digit → 0 | 1 | 2 | . . . | 9

digit → digit digit*

Optimal-fraction → . digits | ε

Optimal-exponent → (E (+ | - | ∈) digits) | ε

num → digits optimal-fraction optimal-exponent.

This regular definition says thatAn optimal-fraction is either a decimal point followed by one or more digits or it is missing (i.e., an empty string).An optimal-exponent is either an empty string or it is the letter E followed by an ' optimal + or - sign, followed by one or more digits.

Recognition of Token

Consider the following grammar.

stmt → if expr then stmt

| if expr then stmt else stmt

| ε

expr → term relop term// relop is relational operator =,>, etc

| term

term → id

| number

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → <|<=|=|<>|>|>=

id → letter(letter|digit)*

num → digit$^+$ (.digit$^+$)?(E(+|-)?digit$^+$)?

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

Transition diagrams

It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

A transition diagram is similar to a flowchart for (a part of) the lexer. We draw one for each possible token. It shows the decisions that must be made based on the input seen. The two main components are circles representing *states* and arrows representing *edges.*

Finite Automata

The secret weapon used by lex et al to convert (compile) its input into a lexer.Finite automata are like the graphs we saw in transition diagrams but they simply decide if a sentence (input string) is in the language (generated by our regular expression). That is, they are recognizers of the language.

There are two types of finite automata

1. Deterministic finite automata (DFA) have for each state (circle in the diagram) exactly one edge leading out for each symbol. So if you know the next symbol and the current state, the next state is determined. That is, the execution is deterministic; hence the name.
2. Nondeterministic finite automata (NFA) are the other kind. There are no restrictions on the edges leaving a state: there can be several with the same symbol as label and some edges can be labeled with ε. Thus there can be several possible next states from a given state and a current lookahead symbol.

Both DFAs and NFAs are capable of recognizing the same languages, the regular languages, i.e., the languages generated by regular expressions (plus the automata can recognize the empty language).

Types of Finite Automata

Deterministic Automata

Non-Deterministic Automata.

DETERMINISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input.

A DFA is a special case of a NFA in which:-

1, it has no transitions on input € ,

2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, Σ, δ, qo, F), where

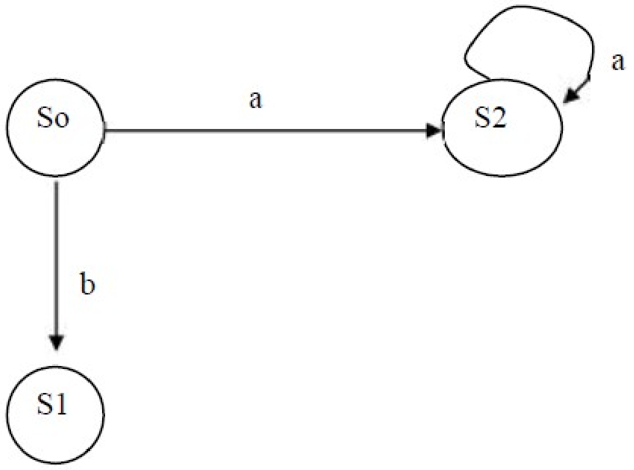
Q is a finite ‘set of states’, which is non empty.

Σ is ‘input alphabets’, indicates input set.

qo is an ‘initial state’ and qo is in Q ie, qo, Σ, Q

F is a set of ‘Final states’,

δ is a ‘transmission function’ or mapping function, using this function the next state can be determined.



The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.

NONDETERMINISTIC AUTOMATA

A NFA is a mathematical model that consists of

¬ A set of states S.

¬ A set of input symbols Σ.

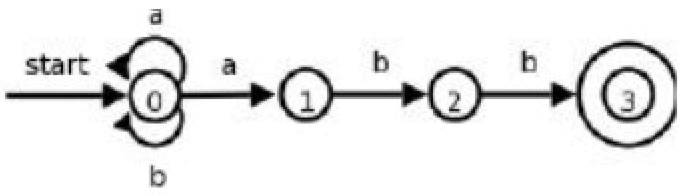
¬ A transition for move from one state to an other.

¬ A state so that is distinguished as the start (or initial) state.

¬ A set of states F distinguished as accepting (or final) state.

¬ A number of transition to a single symbol.

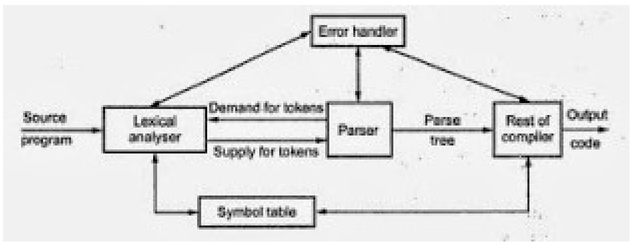
A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function. This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as by input symbols. The transition graph for an NFA that recognizes the language (a | b) * abb is shown



Syntax Analysis:

ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion.



The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-downparsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top down parsers.

$$\mathbf{E} \longrightarrow \mathbf{E + E \mid E * E \mid id}$$

Solution:

$$w = \mathbf{id + id * id}$$

$$\mathbf{E} \xrightarrow[lm]{} \mathbf{E+E}$$

$$\mathbf{E} \xrightarrow[lm]{} \mathbf{id+E} \qquad [\mathbf{E} \longrightarrow \mathbf{id}]$$

$$\mathbf{E} \xrightarrow[lm]{} \mathbf{id+E * E} \qquad [\mathbf{E} \longrightarrow \mathbf{E * E}]$$

$$\mathbf{E} \xrightarrow[lm]{} \mathbf{id + id * E} \qquad [\mathbf{E} \longrightarrow \mathbf{id}]$$

$$\mathbf{E} \xrightarrow[lm]{} \mathbf{id + id * id} \qquad [\mathbf{E} \longrightarrow \mathbf{id}]$$

RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands forleftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form .A syntax of the form defines zero or one occurrence of the form. A syntax of the form defines zero or more occurrences of the form

.

A usual implementation of an LL(1) parser is:

initialize its data structures,
get the lookahead token by calling scanner routines, and
call the routine that implements the start symbol.

```
procsyntaxAnalysis()
begin
initialize(); // initialize global data and structures
nextToken(); // get the lookahead token
program(); // parser routine that implements the start symbol
end;
```

Context free grammars

It involves four quantities.
CFG contain terminals,
N-T, start symbol and production.
Terminal are basic symbols form which string are formed.
N-terminals are synthetic variables that denote sets of strings In a Grammar, one
N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.
The production of the grammar specify the manor in which the terminal and N-T can be combined to form strings.
Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

Bottom – up Parsing.

Bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:
Input string : a + b * c
Production rules
$S \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow E * T$

E → T

T → id

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S

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:

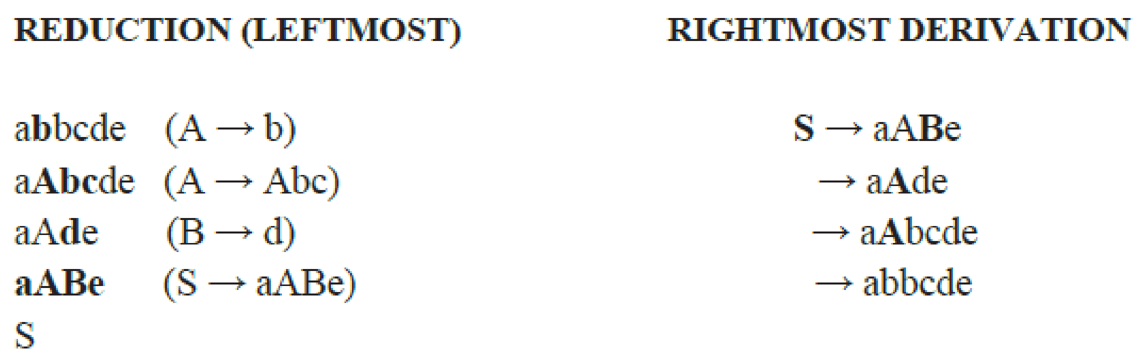
Consider the grammar:

S → aABe

A → Abc | b

B → d

The sentence to be recognized is abbcde.



REDUCTION (LEFTMOST)		RIGHTMOST DERIVATION
abbcde	(A → b)	S → aABe
aAbcde	(A → Abc)	→ aAde
aAde	(B → d)	→ aAbcde
aABe	(S → aABe)	→ abbcde
S		

This reductions trace out the right most derivations in reverse.

Handles

A handle of a string is a substring that matches the right side of a production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a right most derivation.

Example

Consider the grammar

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

And the input string is id+id*id

The right most derivation is:

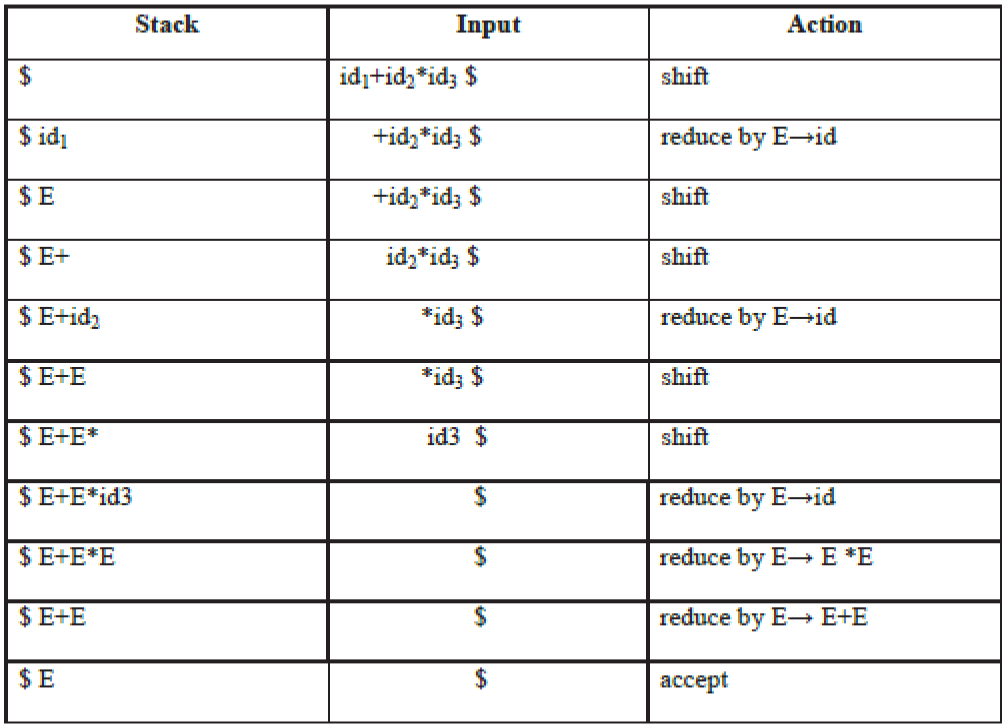
$E \rightarrow \underline{\mathbf{E+E}}$
$\rightarrow E+\underline{\mathbf{E*E}}$
$\rightarrow E+E*\underline{\mathbf{id_3}}$
$\rightarrow E+\underline{\mathbf{id_2}}*id_3$
$\rightarrow \underline{\mathbf{id_1}}+id_2*id_3$

In the above derivations underlined substrings are called handles.

Handle Pruning:

HANDLE PRUNING is the general approach used in shift-and-reduce parsing. A **Handle** is a substring that matches the body of a production. **Handle** reduction is a step in the reverse of rightmost derivation. A rightmost derivation in reverse can be obtained by **handle pruning**.

Stack implementation of shift reduce parsing



Stack	Input	Action
\$	$id_1+id_2*id_3$ \$	shift
\$ id_1	$+id_2*id_3$ \$	reduce by $E \rightarrow id$
\$ E	$+id_2*id_3$ \$	shift
\$ E+	id_2*id_3 \$	shift
\$ $E+id_2$	$*id_3$ \$	reduce by $E \rightarrow id$
\$ E+E	$*id_3$ \$	shift
\$ E+E*	id3 \$	shift
\$ E+E*id3	\$	reduce by $E \rightarrow id$
\$ E+E*E	\$	reduce by $E \rightarrow E * E$
\$ E+E	\$	reduce by $E \rightarrow E+E$
\$ E	\$	accept

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a⋗ b means that terminal "a" has the higher precedence than terminal "b".

a⋖ b means that terminal "a" has the lower precedence than terminal "b".

a≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example

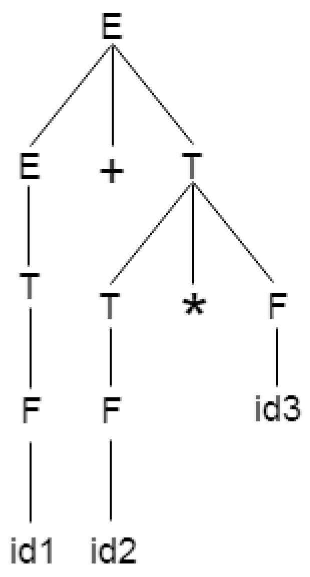
Grammar:

1. E → E+T/T
2. T → T*F/F
3. F → id

Given string:

1. w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	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

Now let us process the string with the help of the above precedence table:

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

LR Parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.In the LR parsing, "L" stands for left-to-right scanning of the input."R" stands for constructing a right most derivation in reverse."K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts:

- LR (0) parsing,
- SLR parsing,
- CLR parsing
- LALR parsing.

LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

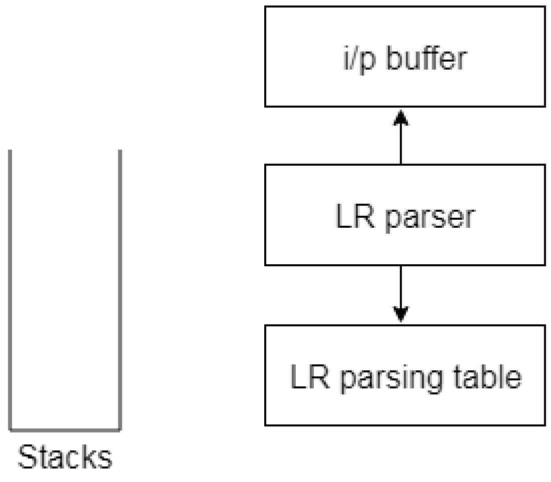


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augment Grammar

Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

Given grammar

1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$

The Augment grammar G` is represented by

1. S`→ S
2. S → AA
3. A → aA | b

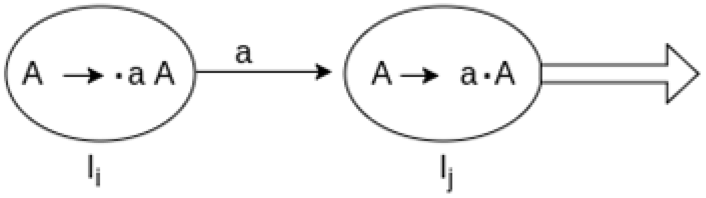
SLR (1) Parsing

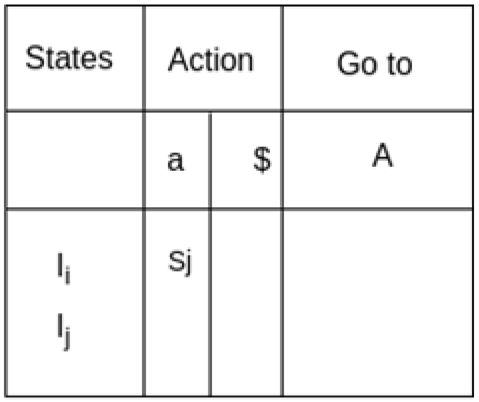
SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table.To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

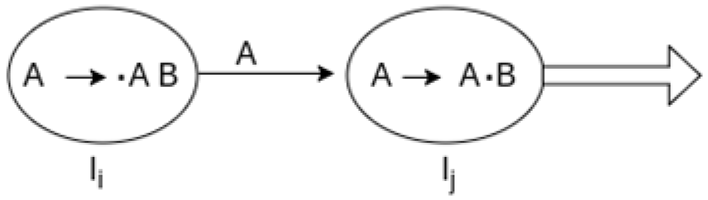
- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table
- SLR (1) Table Construction
- The steps which use to construct SLR (1) Table is given below:
- If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.

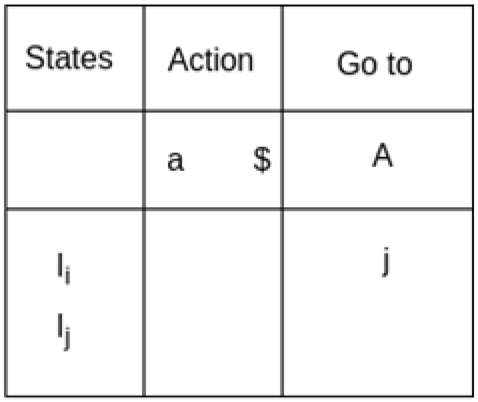




States	Action		Go to
	a	$	A
I_i	Sj		
I_j			

- If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.





States	Action		Go to
	a	$	A
I_i			j
I_j			

- If a state (I_i) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

CLR (1) Parsing

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

LR (1) item

LR (1) item is a collection of LR (0) items and a look ahead symbol.

LR (1) item = LR (0) item + look ahead

The look ahead is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

Example

CLR (1) Grammar

1. S → AA
2. A → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

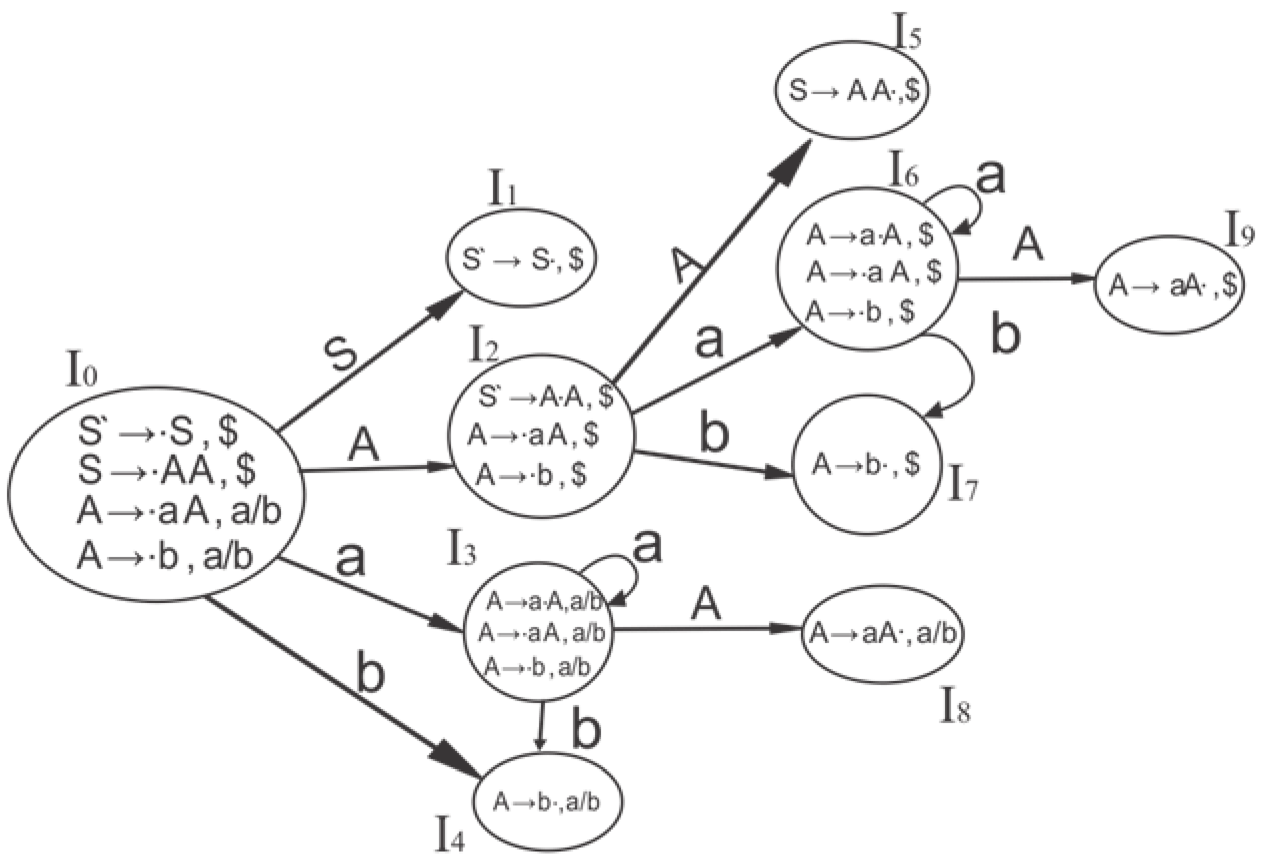
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



CLR (1) Parsing table:

States	a	b	$	S	A
I0	S3	S4			2
I1			Accept		
I2	S6	S7			5
I3	S3	S4			8
I4	R3	R3			
I5			R1		
I6	S6	S7			9
I7			R3		
I8	R2	R2			
I9			R2		

Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA ....(2)
3. A → b ... (3)

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives (A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.

I5 contains the final item which drives (S → AA•, $), so action {I5, $} = R1.

I7 contains the final item which drives (A → b•,$), so action {I7, $} = R3.

I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.

I9 contains the final item which drives (A → aA•, $), so action {I9, $} = R2.

LALR (1) Parsing:

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

LALR (1) Grammar

1. S → AA
2. A → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $

A → •aA, a/b

A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b

A → •aA, a/b

A → •b, a/b

}

I6= { A → a•A, $

A → •aA, $

A → •b, $

}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$

A → •aA, a/b/$

A → •b, a/b/$

}

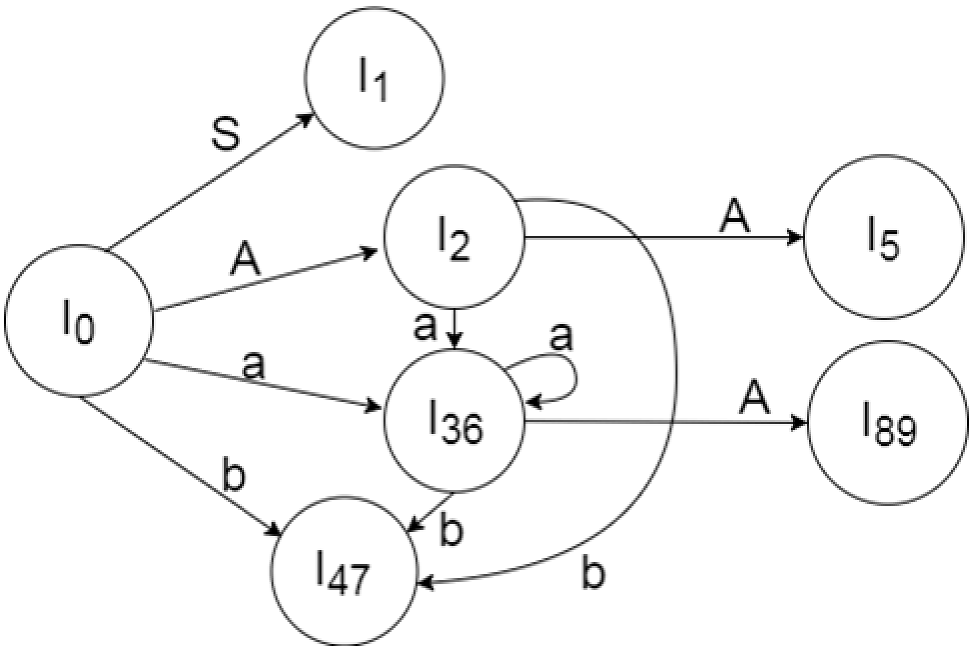
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

UNIT - III

SYNTAX – DIRECTED TRANSLATION

Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

Example

```
E -> E+T | T
```

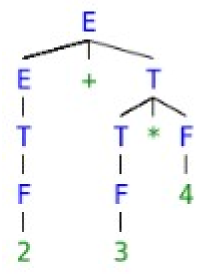
```
T -> T*F | F
F -> INTLIT
```

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don’t have any semantic assertions to check in this very basic example.

```
E -> E+T     { E.val = E.val + T.val }   PR#1
E -> T       { E.val = T.val }           PR#2
T -> T*F     { T.val = T.val * F.val }   PR#3
T -> F       { T.val = F.val }           PR#4
F ->INTLIT  {F.val = INTLIT.lexval }   PR#5
```

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.

Let’s take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.

Syntax Directed definitions

Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.It is a context free grammar with attributes and rules together which are associated with grammar symbols and productions respectively.

The process of syntax directed translation is two-fold:

• Construction of syntax tree and

• Computing values of attributes at each node by visiting the nodes of syntax tree.

Semantic actions

Semantic actions are fragments of code which are embedded within production bodies by syntax directed translation.They are usually enclosed within curly braces ({ }).It can occur anywhere in a production but usually at the end of production.

Example

E---> E1 + T {print '+'}

Types of translation

• L-attributed translation

o It performs translation during parsing itself.

o No need of explicit tree construction.

o L represents 'left to right'.

• S-attributed translation

o It is performed in connection with bottom up parsing.

o 'S' represents synthesized.

Types of attributes

• **Inherited attributes**

o It is defined by the semantic rule associated with the production at the parent of node.

o Attributes values are confined to the parent of node, its siblings and by itself.

o The non-terminal concerned must be in the body of the production.

• **Synthesized attributes**

o It is defined by the semantic rule associated with the production at the node.

o Attributes values are confined to the children of node and by itself.

o The non terminal concerned must be in the head of production.

o Terminals have synthesized attributes which are the lexical values (denoted by lexval) generated by the lexical analyzer.

Construction of Syntax trees

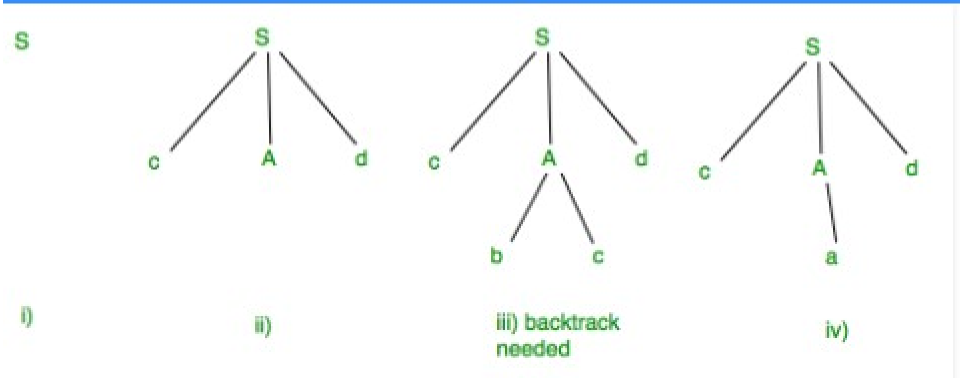
Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, error is reported by syntax analyzer.The Grammar for a Language consists of Production rules.

<u>Example:</u>

Suppose Production rules for the Grammar of a language are:

```
S ->cAd
A ->bc|a
And the input string is “cad”.
```

Now the parser attempts to construct syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string “cad” it uses the rules as shown in the given diagram:



In the step iii above, the production rule A->bc was not a suitable one to apply (because the string produced is “cbcd” not “cad”), here the parser needs to backtrack, and apply the next production rule available with A which is shown in the step iv, and the string “cad” is produced.

Thus, the given input can be produced by the given grammar, therefore the input is correct in syntax. But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.

L-attributed definitions

- If an SDT uses 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 SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

Example,

```
A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
```

is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.

Example – Consider the given below SDT.

```
P1: S ->MN  {S.val= M.val + N.val}
P2: M ->PQ  {M.val = P.val * Q.val  and P.val =Q.val}
```

Select the correct option.

A. Both P1 and P2 are S attributed.

B. P1 is S attributed and P2 is L-attributed.

C. P1 is L attributed but P2 is not L-attributed.

D. None of the above

Explanation

The correct answer is option C as, In P1, S is a synthesized attribute and in L-attribute definition synthesized is allowed. So P1 follows the L-attributed definition. But P2 doesn't follow L-attributed definition as P is depending on Q which is RHS to it.

TopDown Translation

- For each non-terminal A, construct a function that ν Has a formal parameter for each inherited attribute of A
- Returns the values of the synthesized attributes of A
- The code associated with each production does the following
- Save the s-attribute of each token X into a variable X.x.
- Generate an assignment B.s=parseB(B.i1,B.i2,…,B.ik) for each non-terminal B, where B.i1,…,B.ik are values for the Lattributes of B and B.s is a variable to store s-attributes of B.
- Copy the code for each action, replacing references to attributes by the corresponding variables.

Bottom Up Evaluation of inherited attributes

S-attributed definitions

- Syntax-directed definitions with only synthesized attributes
  - Can be evaluated through post-order traversal of parse tree π
- Synthesized attributes and bottom-up parsing
  - Keep attribute values of grammar symbols in stack
  - Evaluate attribute values at each reduction
- In top-down parsing, the return value of each parsing routine

TYPE SYSTEMS. A type system is a set of rules assigning type expressions to different parts of the program.

- Type systems can (usually) be implemented in a syntax-directed way.

- The implementation of a type system is called a type checker.

Specification of simple Type Checker:

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E

D → D ; D | id : T

T → char | integer | array [num] of T | ↑ T

E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E

D → D ; D

D → id : T { addtype (id.entry , T.type) }

T → char { T.type : = char }

T → integer { T.type : = integer }

T → ↑ T1 { T.type : = pointer(T1.type) }

T → array [num] of T1 { T.type : = array (1… num.val , T1.type) }

In the above language,

→ There are two basic types : char and integer ; → type_error is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression

pointer (integer).

T

Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1. E → literal { E.type : = char } E→num { E.type : = integer }
Here, constants represented by the tokens literal and num have type char and integer.

2. E → id { E.type : = lookup (id.entry) }

lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 mod E2 { E.type : = if E1. type = integer and E2. type = integer then integer
elsetype_error }

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type_error.

4. E → E1 [E2] { E.type : = if E2.type = integer and E1.type = array(s,t) then t
elsetype_error }

In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

5. E → E1 ↑ { E.type : = if E1.type = pointer (t) then t
elsetype_error }
The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement: S→id: = E

S→id: = E { S.type : = if id.type = E.type then void
else type_error }

2. Conditional statement: S→if E then S1

S→if E then S1 { S.type : = if E.type = boolean then S1.type
else type_error }

3. While statement:

S → while E do S1

S → while E do S1 { S.type : = if E.type = boolean then S1.type
else type_error }

4. Sequence of statements:

S → S1 ; S2 { S.type : = if S1.type = void and S1.type = void then void else type_error }

S → S1 ; S2 { S.type : = if S1.type = void and
S1.type = void then void
else type_error }

Type checking of functions

The rule for checking the type of a function application is : E → E1 (E2) { E.type : = if
E2.type = s and
E1.type = s → t then t else type_error }

Type Conversion:

In computer science, **type conversion** or typecasting refers to changing an entity of one datatype into another. There are two **types** of **conversion**: implicit and explicit. The term for implicit **type conversion** is coercion. Explicit **type conversion** in some specific way is known as **casting**.

UNIT - IV

INTERMEDIATE CODE GENERATION

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

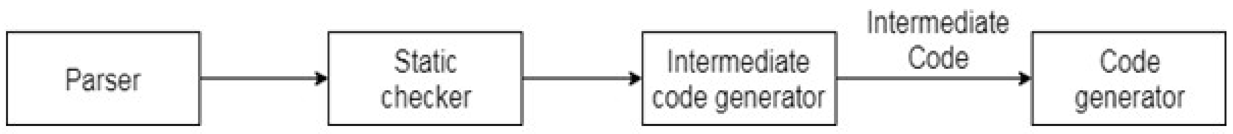


Figure: Position of the intermediate code generator

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.

- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Intermediate Language

Three ways of intermediate representation:

- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Graphical Representations: Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.

A syntax tree and dag for the assignment statement a : = b * - c + b * - c are shown in Fig.:

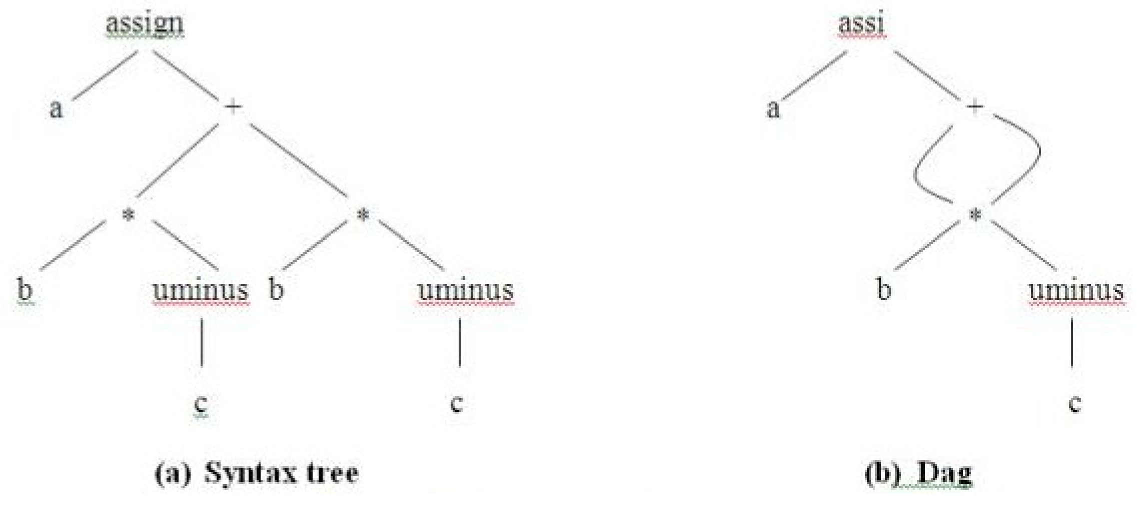


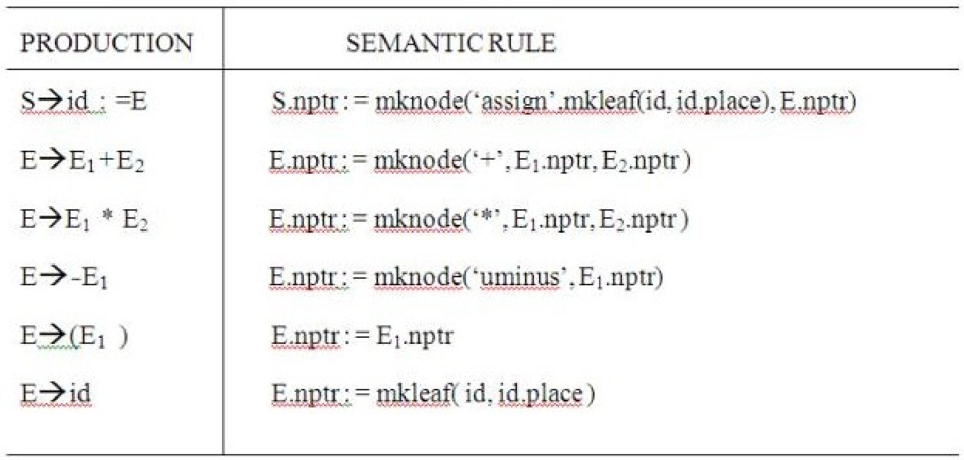
Figure : a : = b * - c + b * - c

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

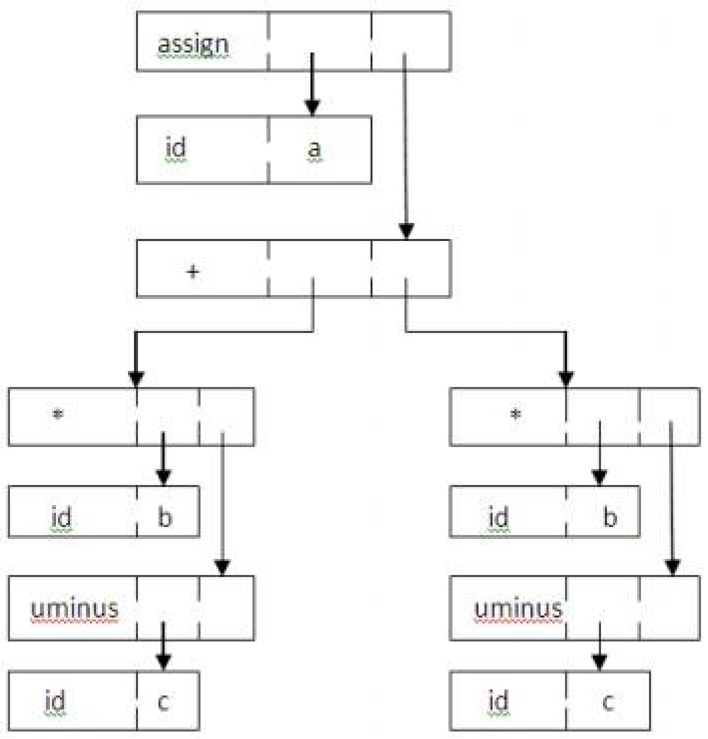


PRODUCTION	SEMANTIC RULE
S→id : =E	S.nptr : = mknode('assign'.mkleaf(id, id.place), E.nptr)
E→E_1+E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E→E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E→-E_1	E.nptr : = mknode('uminus', E_1.nptr)
E→(E_1)	E.nptr : = E_1.nptr
E→id	E.nptr : = mkleaf(id, id.place)

Fig. Syntax-directed definition to produce syntax

The token id has an attribute place that points to the symbol-table entry for the toen identifier. A symbol-table entry can be found from an attribute id.name, representing the lexeme associated with that occurrence of id. If the lexical analyzer holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodesare allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

Fig. Two representations of the syntax tree

Three-address code

Three-address code is a sequence of statements of the general form x : = y op z where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

```
t1    :    = y  * z
t2    :    = x  + t1
```

where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

* The unraveling of complicated arithmetic expressions and of statements makes three-address code desirable for target code generation and optimization. * The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged - unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

```
t1 : =  - c
t2 : =  b*  t1
t3 : =  - c
t4 : =  b*  t3
t5 : =  t2+  t4
a : =   t5
```

(a) Code for the syntax tree

```
t1 : = -c
t2 : = b * t1
t5 : = t2 + t2
a : = t5
```

(b) Code for the dag

Fig. Three-address code corresponding to the syntax tree and dag

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. Copy statements of the form x : = y where the value of y is assigned to x.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y as in the usual sequence.

6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. For example,

```
param x1
param x2
.
.
param xn
call p,n
```

generated as part of a call of the procedure p(x1, x2, …. ,xn).

7.Indexed assignments of the form x : = y[i] and x[i] : = y.

8.Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, id : = E consists of code to evaluate E into some temporary t, followed by the assignment id.place : = t.

Given input a : = b * - c + b * - c, the three-address code is as shown in Fig. The synthesized attribute S.code represents the three-address code for the assignment S.

The nonterminal E has two attributes :

1. E.place, the name that will hold the value of E , and
2. E.code, the sequence of three-address statements evaluating E.

Syntax-directed definition to produce three-address code for assignments.

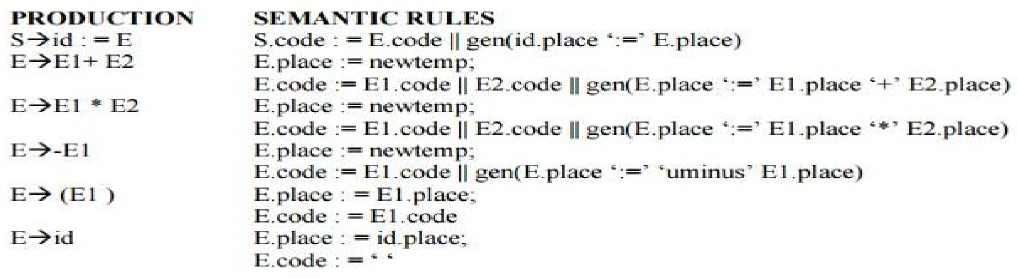
```
PRODUCTION : SEMANTIC RULES
S->whileEdoS1: S.begin := newlabel;
S.after := newlabel;
S.code := gen(S.begin ‘:’) ||
E.code ||
gen ( ‘if’ E.place ‘=’ ‘0’ ‘goto’ S.after)|| S1.code ||
gen ( ‘goto’ S.begin) || gen ( S.after ‘:’)
```

The function newtemp returns a sequence of distinct names t1,t2,….. in response to successive calls.

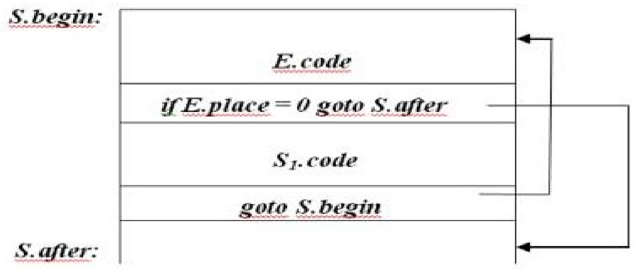
Ø Notation gen(x ‘:=’ y ‘+’ z) is used to represent three-address statement x := y + z. Expressions appearing instead of variables like x, y and z are evaluated when passed to gen, and quoted operators or operand, like ‘+’ are taken literally.

Ø Flow-of-control statements can be added to the language of assignments. The code for S
while E do S1 is generated using new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code for S, respectively.

The function newlabel returns a new label every time it is called.
We assume that a non-zero expression represents true; that is when the value of E becomes zero, control leaves the while statement.



PRODUCTION	SEMANTIC RULES
S→id : = E	S.code : = E.code || gen(id.place ':=' E.place)
E→E1+ E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place)
E→E1 * E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)
E→-E1	E.place := newtemp;
	E.code := E1.code || gen(E.place ':=' 'uminus' E1.place)
E→ (E1)	E.place : = E1.place;
	E.code : = E1.code
E→id	E.place : = id.place;
	E.code : = ' '



○

Fig.Semantic rules generating code for a while statement

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;

Allocation process:
{offset = 0}

   int a;
   id.type = int
   id.width = 2

offset = offset + id.width
{offset = 2}

   float b;
   id.type = float
   id.width = 4

offset = offset + id.width
{offset = 6}
```

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

```
P    M D

M    ε

D    D ; D | id : T | proc id ; N D ; S

N    ε
```

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

```
S    id : = E        { p : = lookup ( id.name);
                       if p ≠ nil then
                       emit( p ‘ : =’ E.place)
                       else error }

E    E1 + E2         { E.place : = newtemp;
                       emit( E.place ‘: =’ E1.place ‘ + ‘ E2.place ) }

E    E1 * E2         { E.place : = newtemp;
                       emit( E.place ‘: =’ E1.place ‘ * ‘ E2.place ) }
```

E - E_1

{ E.place : = newtemp;

emit (E.place ‘: =’ ‘uminus’ E_1.place) }

E (E_1)

{ E.place : = E_1.place }

```
E    id              { p : = lookup ( id.name);

                        if p ≠ nil then

                            E.place : = p

                        else error }
```

Reusing Temporary Names

The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.

Temporaries can be reused by changing newtemp. The code generated by the rules for E E$_1$ + E$_2$ has the general form:

```
evaluate E1 into t1

evaluate E2 into t2

t : = t1  + t2
```

The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.

Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use \$c and increase c by 1.

For example, consider the assignment x := a * b + c * d – e * f

Three-address code with stack temporaries

statement	value of c
	0
\$0 := a * b	1

\$1 := c * d	2
\$0 := \$0 + \$1	1
\$1 := e * f	2
\$0 := \$0 - \$1	1
x := \$0	0

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the ith element of array A begins in location

base + (i – low) x w

where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is, base is the relative address of A[low].

The expression can be partially evaluated at compile time if it is rewritten as

i x w + (base – low x w)

The subexpression c = base – low x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.

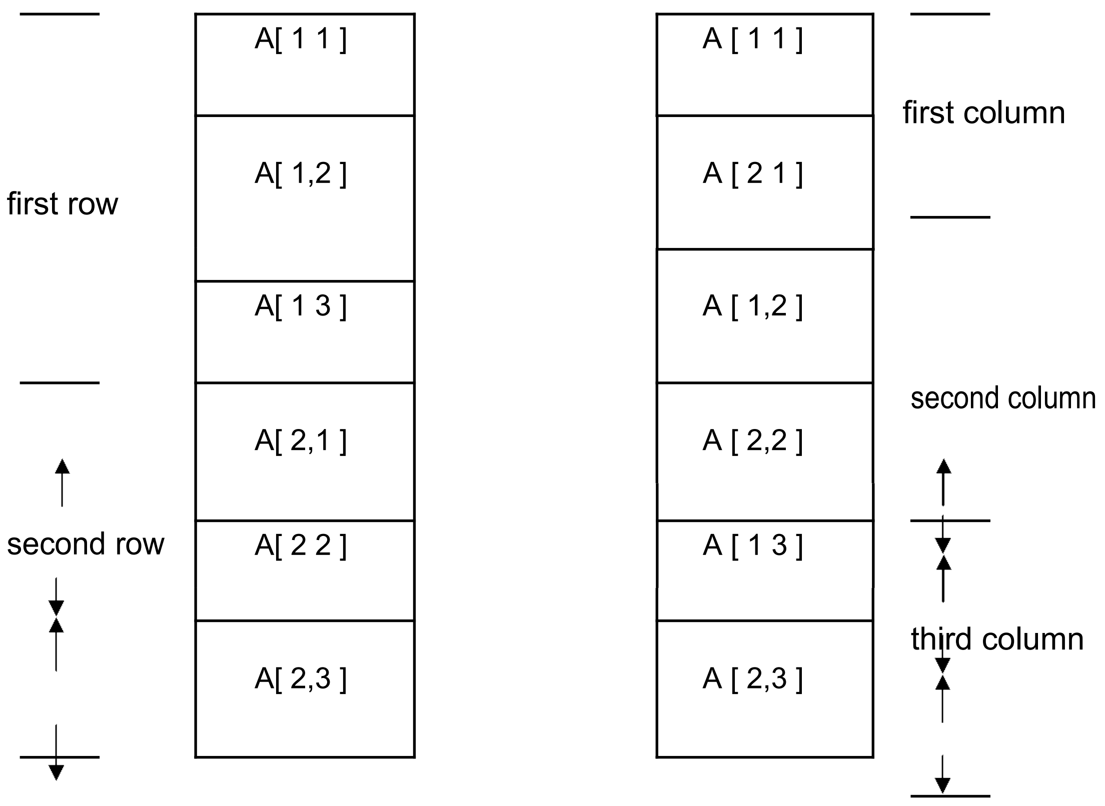
Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

Row-major (row-by-row)

Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR (b) COLUMN-MAJOR

In the case of row-major form, the relative address of A[i_1 , i_2] can be calculated by the formula

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + i_2 - \text{low}_2) \times w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if high_2 is the upper bound on the value of i_2, then $n_2 = \text{high}_2 - \text{low}_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of $A[i_1,i_2,\ldots,i_k]$

$$((\ldots((i_1n_2 + i_2)\,n_3 + i_3)\ldots)\,n_k + i_k) \times w + base - ((\ldots((low_1n_2 + low_2)n_3 + low_3)\ldots)\,n_k + low_k) \times w$$

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) SL : = E

(2) EE + E

(3) E(E)

(4) EL

(5) LElist]

(6) Lid

(7) ElistElist , E

(8) Elistid [E

We generate a normal assignment if L is a simple name, and an indexed assignment into the location denoted by L otherwise :

```
(1) S   L : = E          { if L.offset = null then / * L is a simple id */
                              emit ( L.place ‘: =’ E.place ) ;
                           else
                              emit ( L.place ‘ [‘ L.offset ‘ ]’ ‘: =’ E.place) }
(2) E   E1 + E2          { E.place : = newtemp;
                           emit ( E.place ‘: =’ E1.place ‘ +’ E2.place ) }
(3) E   ( E1 )           { E.place : = E1.place }
```

When an array reference L is reduced to E , we want the r-value of L. Therefore we use indexing to obtain the contents of the location L.place [L.offset] :

```
(4) E   L                    { if L.offset = null then  /* L is a simple id* /
                                   E.place : = L.place
                               else begin
                                   E.place : = newtemp;
                                   emit ( E.place ‘: =’ L.place ‘ [‘ L.offset ‘]’)
                               end }

(5) L   Elist ]               { L.place : = newtemp;
                                L.offset : = newtemp;
                                emit (L.place ‘: =’ c( Elist.array ));
                                emit (L.offset ‘: =’ Elist.place ‘*’ width (Elist.array)) }

(6) L   id                    { L.place := id.place;
                                 L.offset := null }

(7) Elist   $Elist_1$ , E        { t := newtemp;
                                m : = $Elist_1$.ndim + 1;
                                emit ( t ‘: =’ $Elist_1$.place ‘*’ limit ($Elist_1$.array,m));

                                emit ( t ‘: =’ t ‘+’ E.place);

                                Elist.array : = $Elist_1$.array;


                                Elist.place : = t;

                                Elist.ndim : = m }


(8)  Elist   id [ E{ Elist.array : = id.place;

                                Elist.place : = E.place;

                                Elist.ndim : = 1 }
```

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute E.type, whose value is either real or integer. The semantic rule for E.type associated with the production E E + E is :

```
E        E + E          { E.type : =

                              if $E_1$.type = integer and

                                 $E_2$.type = integer then integer

                              else real }
```

The entire semantic rule for E E + E and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Semantic action for E E_1 + E_2 $\Rightarrow$

```
E.place := newtemp;

if $E_1$.type = integer and $E_2$.type = integer then
     begin emit( E.place ‘: =’ $E_1$.place ‘int +’
     $E_2$.place); E.type : = integer

end

else if $E_1$.type = real and $E_2$.type = real then
       begin emit( E.place ‘: =’ $E_1$.place ‘real +’
       $E_2$.place); E.type : = real

end

else if $E_1$.type = integer and $E_2$.type = real then
      begin u : = newtemp;

      emit( u ‘: =’ ‘inttoreal’ $E_1$.place);

      emit( E.place ‘: =’ u ‘ real +’ $E_2$.place);

      E.type : = real
```

```
end

else if $E_1$.type = real and $E_2$.type =integer then
        begin u : = newtemp;

        emit( u ‘: =’ ‘inttoreal’ $E_2$.place);

        emit( E.place ‘: =’ $E_1$.place ‘ real +’ u);

        E.type : = real

end

else

        E.type : = type_error;
```

For example, for the input x : = y + i * j

assuming x and y have type real, and i and j have type integer, the output would look like

```
$t_1$ : = i int* j

$t_3$ : = inttoreal $t_1$

$t_2$ : = y real+ $t_3$

x : = $t_2$
```

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 relop E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E   E or E | E and E | not E | (E) | id relop id | true | false

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.

To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

The translation for

a or b and not c

is the three-address sequence

t_1 : = not c

t_2 : = b and t_1

t_3 : = a or t_2

A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103

101 :   t : = 0

102 :   goto 104

103 :   t : = 1

104 :
```

Translation scheme using a numerical representation for booleans

E E_1 or E_2

E not E_1

E E_1 and E_2

E (E_1)

E id_1 relop id_2

{ E.place : = newtemp;

emit(E.place ‘: =’ E_1.place ‘or’ E_2.place) }

{ E.place : = newtemp;

emit(E.place ‘: =’ E_1.place ‘and’ E_2.place) }

E   true

{ E.place : = newtemp;

E   false

emit(E.place ‘: =’ ‘not’ E_1.place) }

{ E.place : = E_1.place }

{ E.place : = newtemp;

emit(‘if’ id_1.place relop.op id_2.place ‘goto’ nextstat + 3);

emit(E.place ‘: =’ ‘0’);

emit(‘goto’ nextstat +2);

emit(E.place ‘: =’ ‘1’) }

{ E.place : = newtemp;

emit(E.place ‘: =’ ‘1’) }

{ E.place : = newtemp;

emit(E.place ‘: =’ ‘0’) }

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short-circuit” or “jumping” code. It is possible to evaluate boolean expressions without generating code for the boolean operators and, or, and not if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

```
100 : if  a < b goto  103
101 :  t1 : = 0
102 : goto 104
103 : t1  : = 1
104 : if  c < d goto  107
105 : t2  : = 0
106 : goto 108
107 : t2 : = 1
108 : if e < f goto 111
109 : t3 : = 0
110 : goto 112
111 : t3 : = 1
112 : t4 : = t2 and t3
113 : t5 : = t1 or t4
```

Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S if E then S_1

| if E then S_1 else S_2

| while E do S_1

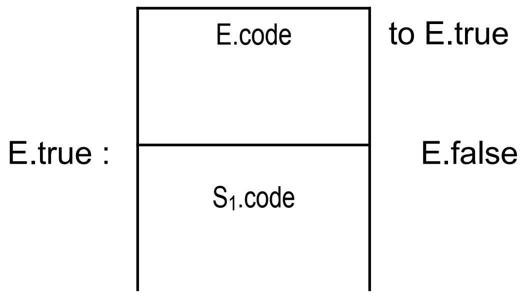
In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.

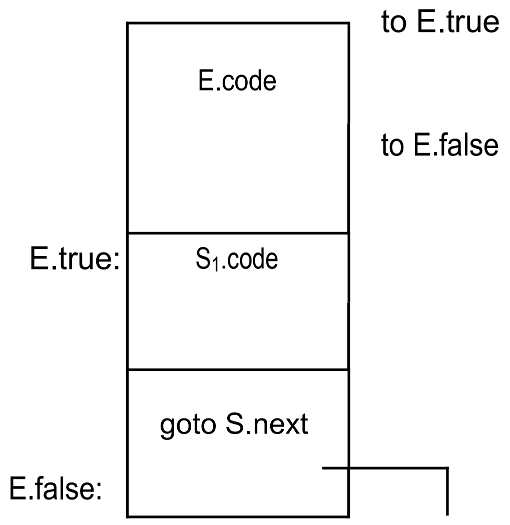
E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.

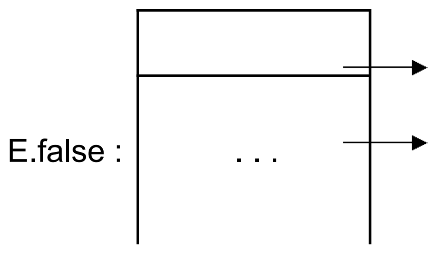
The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.

S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

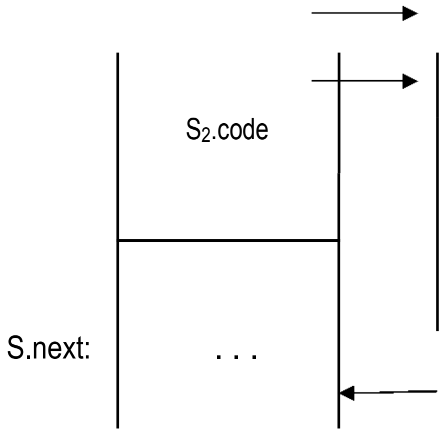
Code for if-then , if-then-else, and while-do statements



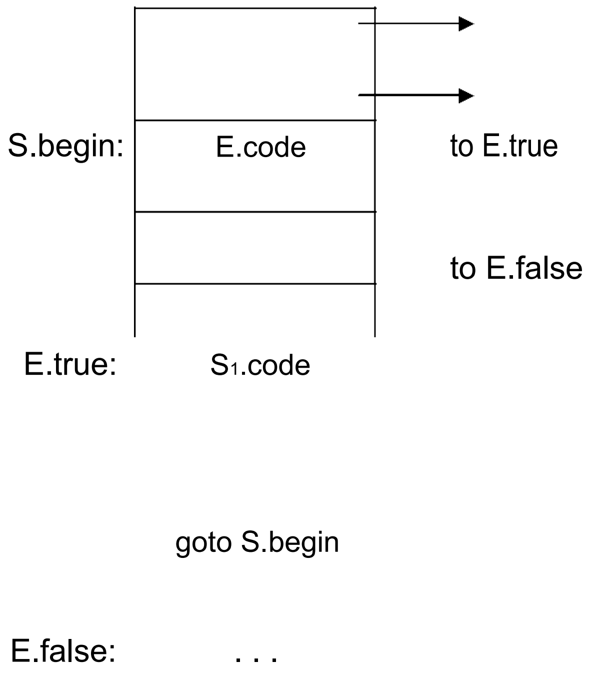




(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
S if E then S_1	E.true : = newlabel;
	E.false : = S.next;
	S_1.next : = S.next;
	S.code : = E.code || gen(E.true ‘:’) || S_1.code
S if E then S_1 else S_2	E.true : = newlabel;
	E.false : = newlabel;
	S_1.next : = S.next;
	S_2.next : = S.next;
	S.code : = E.code || gen(E.true ‘:’) || S_1.code ||
	gen(‘goto’ S.next) ||
	gen(E.false ‘:’) || S_2.code
S while E do S_1	S.begin : = newlabel;
	E.true : = newlabel;
	E.false : = S.next;
	S_1.next : = S.begin;
	S.code : = gen(S.begin ‘:’)|| E.code ||
	gen(E.true ‘:’) || S_1.code ||
	gen(‘goto’ S.begin)

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES

E E_1 or E_2	E_1.true : = E.true;
	E_1.false : = newlabel;
	E_2.true : = E.true;
	E_2.false : = E.false;
	E.code : = E_1.code || gen(E_1.false ':') || E_2.code
E E_1 and E_2	E.true : = newlabel;
	E_1.false : = E.false;
	E_2.true : = E.true;
	E_2.false : = E.false;
	E.code : = E_1.code || gen(E_1.true ':') || E_2.code
E not E_1	E_1.true : = E.false;
	E_1.false : = E.true;
	E.code : = E_1.code
E (E1)	E_1.true : = E.true;
	E_1.false : = E.false;
	E.code : = E_1.code
E id_1 relop id_2	E.code : = gen('if' id_1.place relop.op id_2.place 'goto' E.true) || gen('goto' E.false)
E true	E.code : = gen('goto' E.true)
E false	E.code : = gen('goto' E.false)

CASE STATEMENTS

The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

```
switch expression
    begin
        case value : statement
        case value : statement
            . . .
        case value : statement
        default :    statement
    end
```

There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, including a default “value” which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

By a sequence of conditional goto statements, if the number of cases is small.

By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.

If the number of cases s large, it is efficient to construct a hash table.

There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of $i_{max} - i_{min}$, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset $j - i_{min}$ and the label for the default in entries not filled otherwise. To perform switch, evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset j-i_{min} .

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```
switch E

      begin
            case $V_1$ :   $S_1$
            case $V_2$ :   $S_2$

                        . . .
            case $V_{n-1}$ :  $S_{n-1}$
            default :     $S_n$

      end
```

This case statement is translated into intermediate code that has the following form :

Translation of a case statement

```
                code to evaluate E into t
                goto test
$L_1$ :          code for $S_1$
```

```
                goto next
L2 :            code for S2
                goto next
                   . . .
Ln-1 :          code for Sn-1
                goto next
Ln :            code for Sn
                goto next
test :          if  t = V1 goto L1
                if  t = V2 goto L2
                   . . .
                if  t = Vn-1 goto Ln-1
                goto Ln
next :
```

To translate into above form :

When keyword switch is seen, two new labels test and next, and a new temporary t are generated.

As expression E is parsed, the code to evaluate E into t is generated. After processing E , the jump goto test is generated.

As each case keyword occurs, a new label L_i is created and entered into the symbol table. A pointer to this symbol-table entry and the value V_i of case constant are placed on a stack (used only to store cases).

Each statement case V_i : S_i is processed by emitting the newly created label L_i, followed by the code for S_i , followed by the jump goto next.

Then when the keyword end terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

case V_1 L_1

case V_2 L_2

. . .

case V_{n-1} L_{n-1}

case t L_n

label next

where t is the name holding the value of the selector expression E, and L_n is the label for the default statement.

BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

1. makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
2. merge(p_1,p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) E E_1 or M E_2

(2) | E_1 and M E_2

(3) | not E_1

(4) | (E_1)

(5) | id_1 relop id_2

(6) | true

(7) | false

(8) M ε

Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by E.truelist and E.falselist.

Consider production E E_1 and M E_2. If E_1 is false, then E is also false, so the statements on

E_1.falselist become part of E.falselist. If E_1 is true, then we must next test E_2, so the target for the statements E_1.truelist must be the beginning of the code generated for E_2. This target is obtained using marker nonterminal M.

Attribute M.quad records the number of the first statement of E_2.code. With the production M

ε we associate the semantic action

{ M.quad : = nextquad }

The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E_1.truelist when we have seen the remainder of the production E E_1 and

M E_2. The translation scheme is as follows:

(1) E E_1 or M E_2	{ backpatch (E_1.falselist, M.quad);
	E.truelist : = merge(E_1.truelist, E_2.truelist);
	E.falselist : = E_2.falselist }
(2) E E_1 and M E_2	{ backpatch (E_1.truelist, M.quad);
	E.truelist : = E_2.truelist;
	E.falselist : = merge(E_1.falselist, E_2.falselist) }
(3) E not E_1	{ E.truelist : = E_1.falselist;
	E.falselist : = E_1.truelist; }
(4) E (E_1)	{ E.truelist : = E_1.truelist;
	E.falselist : = E_1.falselist; }
(5) E id_1 relop id_2	{ E.truelist : = makelist (nextquad);
	E.falselist : = makelist(nextquad + 1);
	emit('if' id_1.place relop.op id_2.place 'goto_')
	emit('goto_') }
(6) E true	{ E.truelist : = makelist(nextquad);
	emit('goto_') }

```
(7)  E   false                  { E.falselist : = makelist(nextquad);
                                   emit('goto_') }
(8)  M   ε                      { M.quad : = nextquad }
```

Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

```
(1)       S   if E then S
(2)            |  if E then S else S
(3)            |  while E do S
(4)            |  begin L end
(5)            |  A
(6)       LL ; S
(7)           |  S
```

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes E.truelist and E.falselist. L and S also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes L..nextlist and S.nextlist. S.nextlist is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and L.nextlist is defined similarly.

The semantic rules for the revised grammar are as follows:

(1) S if E then M_1 S_1 N else M_2 S_2

{ backpatch (E.truelist, M_1.quad);

backpatch (E.falselist, M_2.quad);

S.nextlist : = merge (S_1.nextlist, merge (N.nextlist, S_2.nextlist)) }

We backpatch the jumps when E is true to the quadruple M_1.quad, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S_2. The list S.nextlist includes all jumps out of S_1 and S_2, as well as the jump generated by N.

(2)	N ε	{ N.nextlist : = makelist(nextquad);
		emit(‘goto _’) }

(3)	M ε	{ M.quad : = nextquad }

(4)	S if E then M S_1	{ backpatch(E.truelist, M.quad);
		S.nextlist : = merge(E.falselist, S_1.nextlist) }

(5)	S while M_1 E do M_2 S_1	{ backpatch(S_1.nextlist, M_1.quad);
		backpatch(E.truelist, M_2.quad);
		S.nextlist : = E.falselist
		emit(‘goto’ M_1.quad) }

(6)	S begin L end	{ S.nextlist : = L.nextlist }
(7)	S A	{ S.nextlist : = nil }

The assignment S.nextlist : = nil initializes S.nextlist to an empty list.

(8)	L L1 ; M S	{ backpatch(L_1.nextlist, M.quad);

L.nextlist : = S.nextlist }

The statement following L_1 in order of execution is the beginning of S. Thus the L1.nextlist list is backpatched to the beginning of the code for S, which is given by M.quad.

(9) L S { L.nextlist : = S.nextlist }

PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement

(1) Scall id (Elist)

(2) Elist Elist , E

(3) Elist E

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence :

When a procedure call occurs, space must be allocated for the activation record of the called procedure.

The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.

Environment pointers must be established to enable the called procedure to access data in enclosing blocks.

The state of the calling procedure must be saved so it can resume execution after the call.

Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.

Finally a jump to the beginning of the code for the called procedure must be generated. For example, consider the following syntax-directed translation

```
(1)  S   call id ( Elist )

              { for each item p on queue do
                      emit (‘ param’ p );


                   emit (‘call’ id.place) }

(2)  Elist   Elist , E

               {  append E.place to the end of queue }


(3)  Elist   E

               {  initialize queue to contain only E.place }
```

Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement.

queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

Code Generation:

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate a correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

Issues in the design of a code generation

1. Input to code generator

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's etc. Assume that they are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Memory Management

Mapping the names in the source program to addresses of data objects is done by the front end and the code generator. A name in the three address statement refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

Instruction selection

Selecting best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered.But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

Register allocation issues

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

1. During Register allocation – we select only those set of variables that will reside in the registers at each point in the program.
2. During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increase, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register.

Evaluation order

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in general case is a difficult NP-complete problem.

Approaches to code generation issues:

Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Maintainable

Run time storage management

Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure. The two standard storage allocation strategies are:

1. Static allocation 2. Stack allocation

- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a

procedure. The record is popped when the activation ends.

The following three-address statements are associated with the run-time allocation and deallocation of activation records:

1. Call,
2. Return,
3. Halt, and
4. Action, a placeholder for other statements.

We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV #here + 20, callee.static_area /*It saves return address*/

GOTO callee.code_area /*It transfers control to the target code for the called procedure */

where,

callee.static_area - Address of the activation record callee.code_area - Address of the first instruction for called procedure

#here + 20 - Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure callee is implemented by : GOTO *callee.static_area. This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP /* initializes stack */
Code for the first procedure
HALT /* terminate execution */
```

Basic Blocks and flow graphs

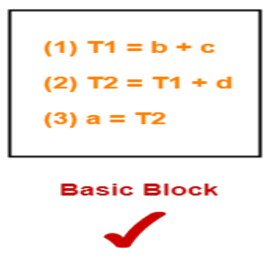
The basic block is a set of statements that always executes in a sequence one after the other.

The characteristics of basic blocks are-

- They do not contain any kind of jump statements in them.
- There is no possibility of branching or getting halt in the middle.
- All the statements execute in the same order they appear.
- They do not lose lose the flow control of the program.

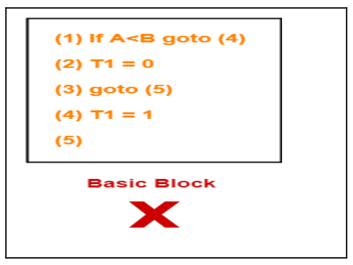
Example Of Basic Block-

Three Address Code for the expression **a = b + c + d** is-



```
(1) T1 = b + c
(2) T2 = T1 + d
(3) a = T2
```

Basic Block



```
(1) If A<B goto (4)
(2) T1 = 0
(3) goto (5)
(4) T1 = 1
(5)
```

Basic Block

Example Of Not A Basic Block-Three Address Code for the expression If A<B then 1 else 0 is-

Here,

- The statements do not execute in a sequence one after the other.
- Thus, they do not form a basic block.

Partitioning Intermediate Code Into Basic Blocks-

Any given code can be partitioned into basic blocks using the following rules-

Rule-01: Determining Leaders-

Following statements of the code are called as **Leaders**–

- First statement of the code.
- Statement that is a target of the conditional or unconditional goto statement.
- Statement that appears immediately after a goto statement.

Rule-02: Determining Basic Blocks-

- All the statements that follow the leader (including the leader) till the next leader appears form one basic block.

- The first statement of the code is called as the first leader.
- The block containing the first leader is called as Initial block.

A simple code generator

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.A code generator generates target code for a sequence of three- address statements andeffectively uses registers to store operands of the statements.

• For example: consider the three-address statement a :=b+c It can have the following sequence of codes:

```
ADD Rj, Ri Cost = 1
        (or)
ADD c, Ri Cost = 2
        (or)
MOV c, Rj Cost = 3
ADD Rj, Ri
```

Register allocation and assignment.

Register and Address Descriptors:

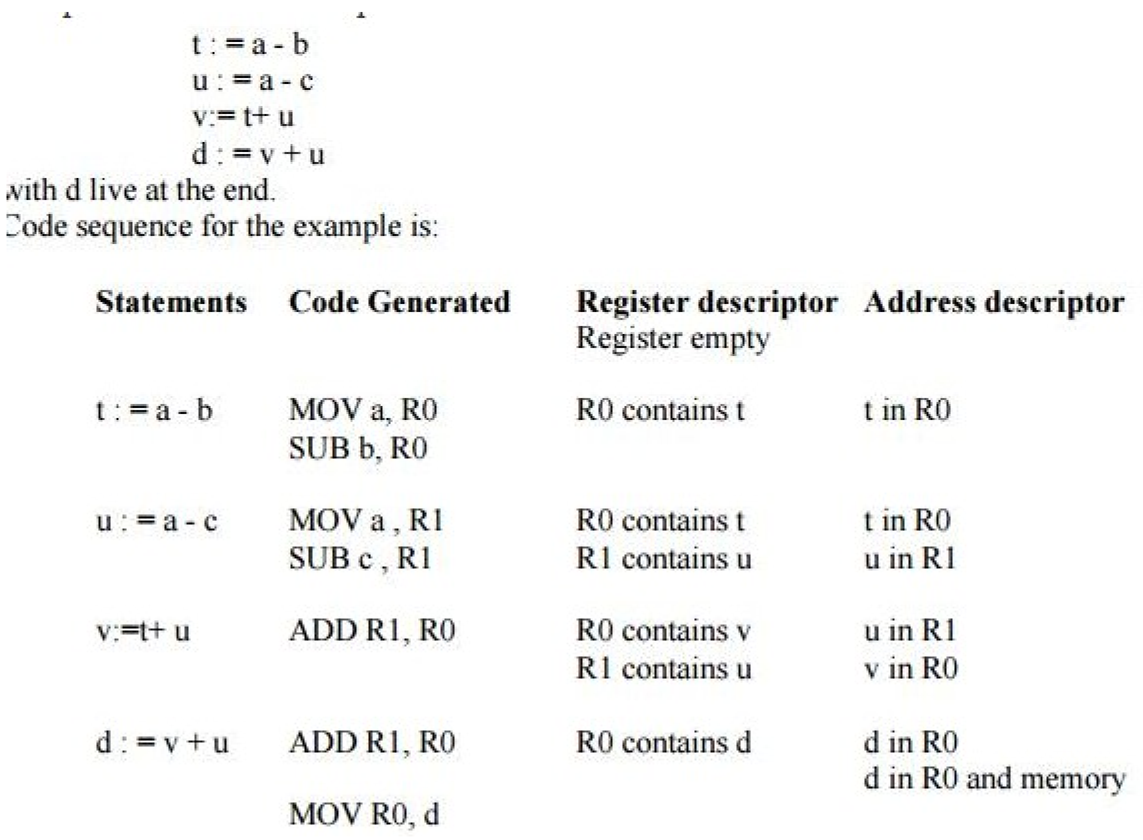
• A register descriptor is used to keep track of what is currently in each registers. The register
descriptors show that initially all the registers are empty.

• An address descriptor stores the location where the current value of the name can be found at run time.

Generating Code for Assignment Statements:

• The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

Code sequence for the example is:



```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

with d live at the end.
Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0	R0 contains d	d in R0 d in R0 and memory
	MOV R0, d		

UNIT – V

CODE OPTIMIZATION

Introduction:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

The Principle sources of optimization

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

¬‸The transformations

ᆸ Common sub expression elimination,

ᆸ Copy propagation,

ᆸ Dead-code elimination, and

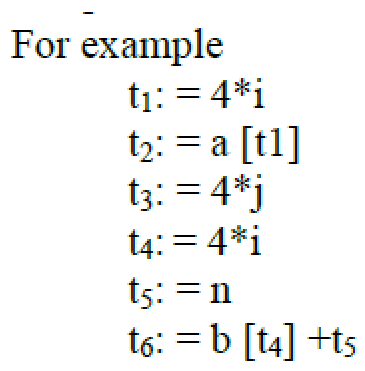
ᆸ Constant folding

are common examples of such function-preserving transformations. The othertransformations come up primarily when global optimizations are performed.

¬‸Frequently, a program will include several calculations of the same value, such as anoffset in an array. Some of the duplicate calculations cannot be avoided by theprogrammer because they lie below the level of detail accessible within the sourcelanguage.

Common Sub expressions elimination:

¬‸An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.



For example

```
        t1: = 4*i
        t2: = a [t1]
        t3: = 4*j
        t4: = 4*i
        t5: = n
        t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

$t_1: = 4*i$
$t_2: = a\ [t_1]$
$t_3: = 4*j$
$t_5: = n$
$t_6: = b\ [t_1] + t_5$

The common sub expression t4: =4*i is eliminated as its computation is already in t1. and value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The ideabehind the copy-propagation transformation is to use g for f, whenever possible after thecopy statement f: = g. Copy propagation means use of one variable instead of another.This may not appear to be an improvement, but as we shall see it gives us an opportunityto eliminate x.

For example:

x=Pi;

……

A=x*r*r;

The optimization using copy propagation can be done as follows:

A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise,it is dead at that point. A related idea is dead or useless code, statements that computevalues that never get used. While the programmer is unlikely to introduce any dead codeintentionally, it may appear as the result of previous transformations. An optimization canbe done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, ‘if’ statement is dead code because this condition will never get satisfied.

Constant folding:

¬ˆ We can eliminate both the test and printing from the object code. More generally,deducing at compile time that the value of an expression is a constant and using theconstant instead is known as constant folding.

¬ˆ One advantage of copy propagation is that it often turns the copy statement into deadcode.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

Optimization of basic blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

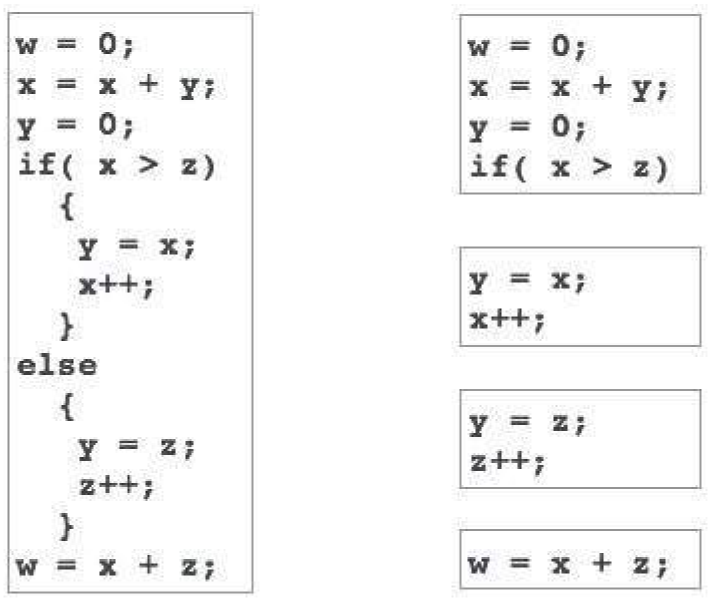
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.
    - Statements that are target of any branch (conditional/unconditional).
    - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

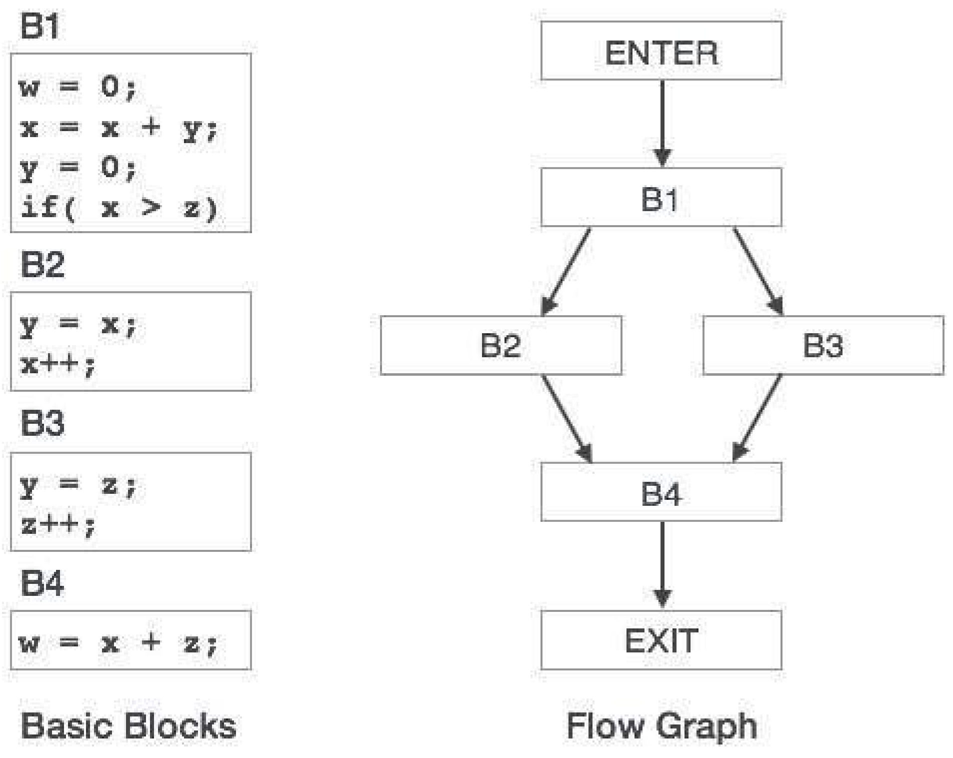
```
w = x + z;
```

Source Code Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks Flow Graph

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- Invariant code : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- Induction analysis : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- Strength reduction : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Loops in flow graphs

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a
code-generation algorithm. Nodes in the flow graph represent computations, and the edges
represent the flow of control.

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.

Introduction to global data flow analysis

In order to do code optimization and a good job of code generation , compiler needs tocollect information about the program as a whole and to distribute this information toeach block in the flow graph.

¬ʌ A compiler could take advantage of “reaching definitions” , such as knowing where avariable like debug was last defined before reaching a given block, in order to performtransformations are just a few examples of data-flow information that an optimizingcompiler collects by a process known as data-flow analysis.

¬ʌ Data-flow information can be collected by setting up and solving systems of equations ofthe 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 generatedwithin the statement , or enters at the beginning and is not killed as control flows throughthe statement.”

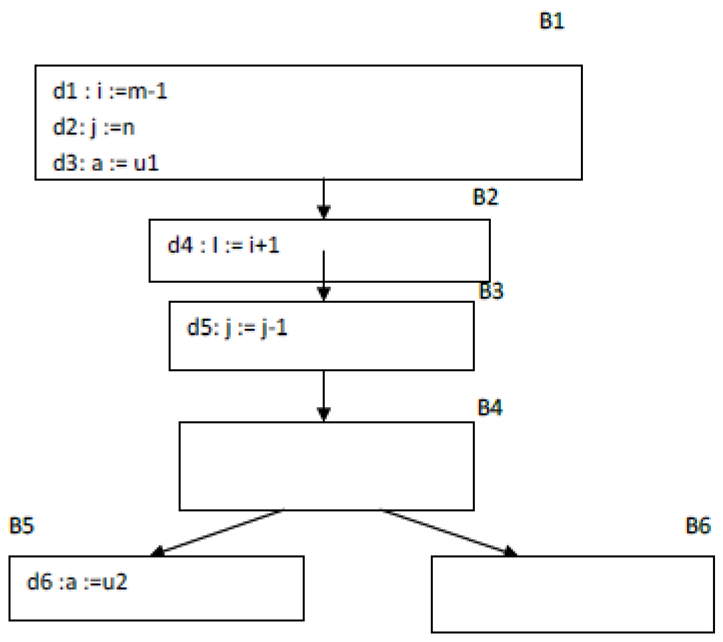
¬ʌ The details of how data-flow equations are set and solved depend on three factors.

ᴴ The notions of generating and killing depend on the desired information, i.e., on the dataflow analysis problem to be solved. Moreover, for some problems, instead of proceedingalong with flow of control and defining out[s] in terms of in[s], we need to proceedbackwards and define in[s] in terms of out[s].

ᴴ Since data flows along control paths, data-flow analysis is affected by the constructs in aprogram. In fact, when we write out[s] we implicitly assume that there is unique endpoint where control leaves the statement; in general, equations are set up at the level ofbasic blocks rather than statements, because blocks do have unique end points. ᴴ There are subtleties that go along with such statements as procedure calls, assignmentsthrough 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 thepoint before the first statement and after the last. Thus, block B1 has four points: onebefore 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 p1to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

ㅂ Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or

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

Reaching definitions:

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

ㄱㅅ These statements certainly define a value for x, and they are referred to as unambiguousdefinitions of x. There are certain kinds of statements that may define a value for x; theyare called ambiguous definitions. The most usual forms of ambiguous definitions of xare:

ㅂ 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.

ㅂ 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 assignmentthrough a pointer is a definition of every variable.

ㄱㅅ We say a definition d reaches a point p if there is a path from the point immediatelyfollowing d to p, such that d is not “killed” along that path. Thus a point can be reached

d1 :i :=m-1

d2: j :=n

d3: a := u1

d4 : I := i+1

d5: j := j-1

d6 :a :=u2

by an unambiguous definition and an ambiguous definition of the same variableappearing later along one path.

Iterative Solution for data flow equations

The most common way of **solving** the **data-flow equations** is by using an **iterative algorithm**. It starts with an approximation of the in-state of each block. The out-states are then computed by applying the transfer functions on the in-states. From these, the in-states are updated by applying the join operations.

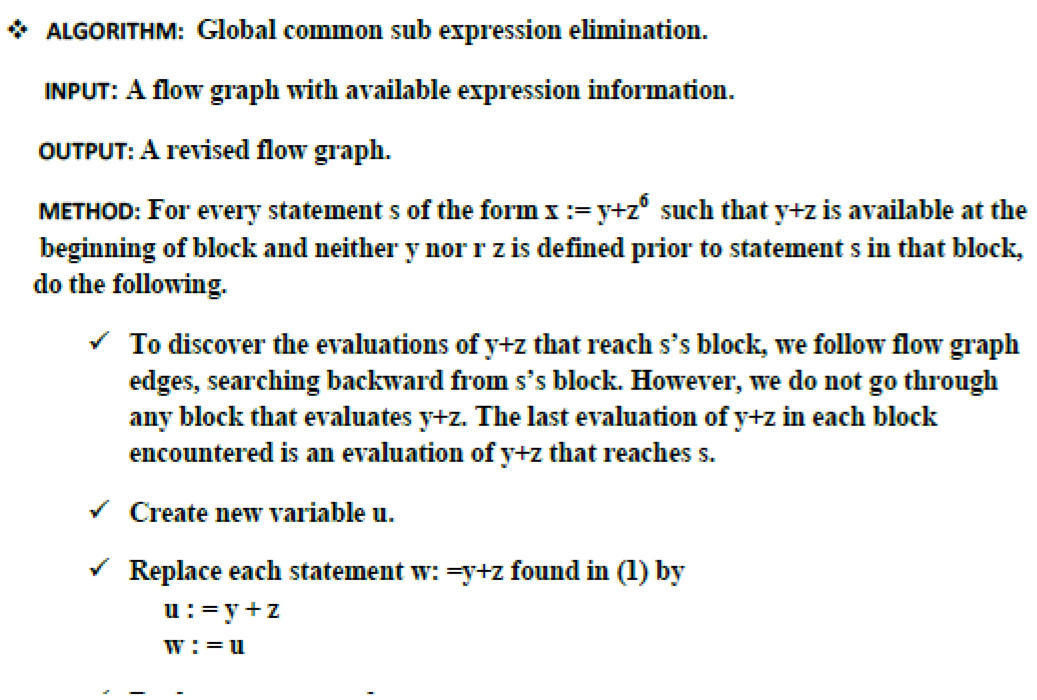
CODE IMPROVING TRANSFORMATIONS

¬ʌ Algorithms for performing the code improving transformations rely on data-flowinformation. Here we consider common sub-expression elimination, copy propagation andtransformations for moving loop invariant computations out of loops and for eliminatinginduction variables.

¬ʌ Global transformations are not substitute for local transformations; both must be performed.

Elimination of global common sub expressions:

¬ʌ The available expressions data-flow problem discussed in the last section allows us todetermine if an expression at point p in a flow graph is a common sub-expression. Thefollowing algorithm formalizes the intuitive ideas presented for eliminating common subexpressions.



❖ **ALGORITHM:** Global common sub expression elimination.

INPUT: A flow graph with available expression information.

OUTPUT: A revised flow graph.

METHOD: For every statement s of the form x := y+z^6 such that y+z is available at the beginning of block and neither y nor r z is defined prior to statement s in that block, do the following.

- ✓ To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.
- ✓ Create new variable u.
- ✓ Replace each statement w: =y+z found in (1) by
  u : = y + z
  w : = u

Copy propagation:

¬ʌ Various algorithms introduce copy statements such as x :=copies may also be generateddirectly by the intermediate code generator, although most of these involve temporarieslocal to one block and can be removed by the dag construction. We may substitute y for xin all these places, provided the following conditions are met every such use u of x.

¬ʌ Statement s must be the only definition of x reaching u.

¬ʌ On every path from s to including paths that go through u several times, there are noassignments to y.

¬ʌ Condition (1) can be checked using ud-changing information. We shall set up a new dataflow
analysis problem in which in[B] is the set of copies s: x:=y such that every pathfrom initial node to the beginning of B contains the statement s, and subsequent to thelast occurrence of s, there are no assignments to y.

Efficient Data Flow Algorithms

ALGORITHM: Copy propagation.

INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, andwith c_in[B] representing the solution to equations that is the set of copies x:=y thatreach block B along every path, with no assignment to x or y following the lastoccurrence of x:=y on the path. We also need ud-chains giving the uses of eachdefinition.

OUTPUT: A revised flow graph.

METHOD: For each copy s : x:=y do the following:

ㅂ Determine those uses of x that are reached by this definition of namely, s: x: =y.

ㅂ Determine whether for every use of x found in (1) , s is in c_in[B], where B is theblock of this particular use, and moreover, no definitions of x or y occur prior to thisuse of x within B. Recall that if s is in c_in[B]then s is the only definition of x thatreaches B.

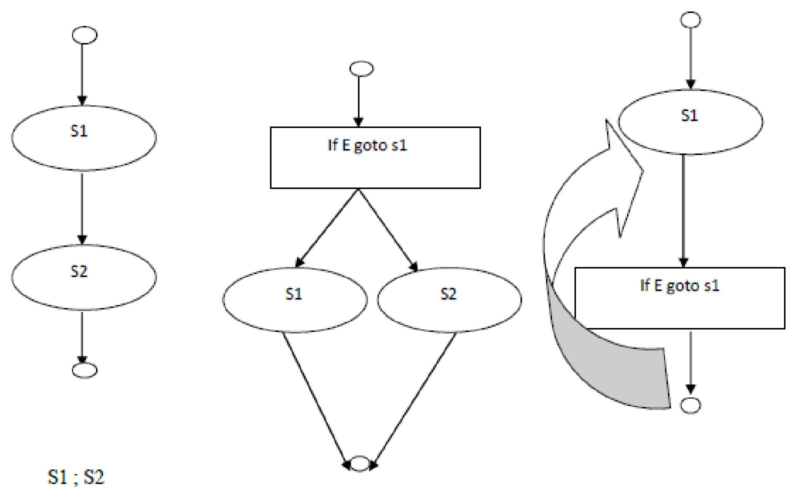
ㅂ If s meets the conditions of (2), then remove s and replace all uses of x found in (1)by y.

Data-flow analysis of structured programs:

¬ㅅ Flow graphs for control flow constructs such as do-while statements have a usefulproperty: there is a single beginning point at which control enters and a single end pointthat control leaves from when execution of the statement is over. We exploit this propertywhen 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

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



S1 ; S2

IF E then S1 else S2 do S1 while E

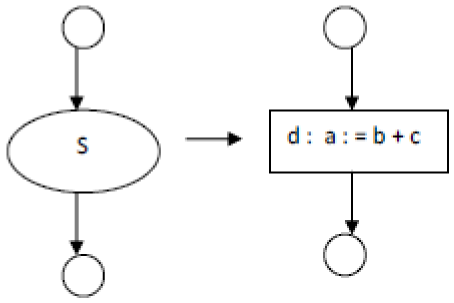
We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.

¬⅄ The portion of flow graph corresponding to a statement S is a region that obeys thefurther restriction that control can flow to just one outside block when it leaves theregion.

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

Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

Conservative estimation of data-flow information:

¬ There is a subtle miscalculation in the rules for gen and kill. We have made theassumption 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 goeither 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 isalways a subset of the computed gen. on the other hand, the true kill is always a supersetof the computed kill.

¬ These containments hold even after we consider the other rules. It is natural to wonderhether these differences between the true and computed gen and kill sets present aserious 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 merelystops 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 achange in the program that changes what the program computes. For the case of reachingdefinitions, then, we call a set of definitions safe or conservative if the estimate is asuperset of the true set of reaching definitions. We call the estimate unsafe, if it is notnecessarily a superset of the truth.

¬⌃ Returning now to the implications of safety on the estimation of gen and kill for reachingdefinitions, note that our discrepancies, supersets for gen and subsets for kill are both inthe safe direction. Intuitively, increasing gen adds to the set of definitions that can reach apoint, 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 translations similar to those usedto compute gen and kill. It can be used, for example, to determine loop-invariantcomputations.

¬⌃ However, there are other kinds of data-flow information, such as the reaching-definitionsproblem. It turns out that in is an inherited attribute, and out is a synthesized attributedepending on in. we intend that in[S] be the set of definitions reaching the beginning ofS, taking into account the flow of control throughout the entire program, includingstatements 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 distinctionbetween out[S] and gen[S]. The latter is the set of definitions that reach the end of Swithout 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 byobserving 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. Nowwe can compute out[S2], and this set is equal to out[S].

¬ʌ Considering if-statement we have conservatively assumed that control can follow eitherbranch, a definition reaches the beginning of S1 or S2 exactly when it reaches thebeginning of S.In[S1] = in[S2] = in[S]

¬ʌ If a definition reaches the end of S if and only if it reaches the end of one or both substatements; i.e,
Out[S]=out[S1] U out[S2]

Representation of sets:
¬ʌ Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bitvectors. We assign a number to each definition of interest in the flow graph. Then bitvector representing a set of definitions will have 1 in position I if and only if thedefinition numbered I is in the set.

¬ʌ The number of definition statement can be taken as the index of statement in an arrayholding pointers to statements. However, not all definitions may be of interest duringglobal data-flow analysis. Therefore the number of definitions of interest will typically be -7recorded in a separate table.

¬ʌ A bit vector representation for sets also allows set operations to be implementedefficiently. The union and intersection of two sets can be implemented by logical or andlogical and, respectively, basic operations in most systems-oriented programminglanguages. The difference A-B of sets A and B can be implemented by taking thecomplement of B and then using logical and to compute A .

Local reaching definitions:
¬ʌ Space for data-flow information can be traded for time, by saving information only atcertain points and, as needed, recomputing information at intervening points.

Basicblocks are usually treated as a unit during global flow analysis, with attention restricted toonly those points that are the beginnings of blocks.

¬⅄ Since there are usually many more points than blocks, restricting our effort to blocks is asignificant 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-definitionchains" or "ud-chains", which are lists, for each use of a variable, of all the definitionsthat reaches that use. If a use of variable a in block B is preceded by no unambiguousdefinition of a, then ud-chain for that use of a is the set of definitions in in[B] that aredefinitions of a.in addition, if there are ambiguous definitions of a ,then all of these forwhich 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 thecomputation of data-flow information using specifications. Specifically, the onlyconstraint on the evaluation order for the gen, kill, in and out sets for statements is thatimposed by dependencies between these sets. Having chosen an evaluation order, we arefree 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 thatdata-flow equations may have circular dependencies.

General control flow:

¬ᄉ Data-flow analysis must take all control paths into account. If the control paths areevidentfrom the syntax, then data-flow equations can be set up and solved in a syntaxdirectedmanner.

¬ᄉ When programs can contain goto statements or even the more disciplined break andcontinue statements, the approach we have taken must be modified to take the actualcontrol paths into account.

¬ᄉ 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 arereducible, such constraints can be handled systematically using the interval-basedmethods

¬ᄉ However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.