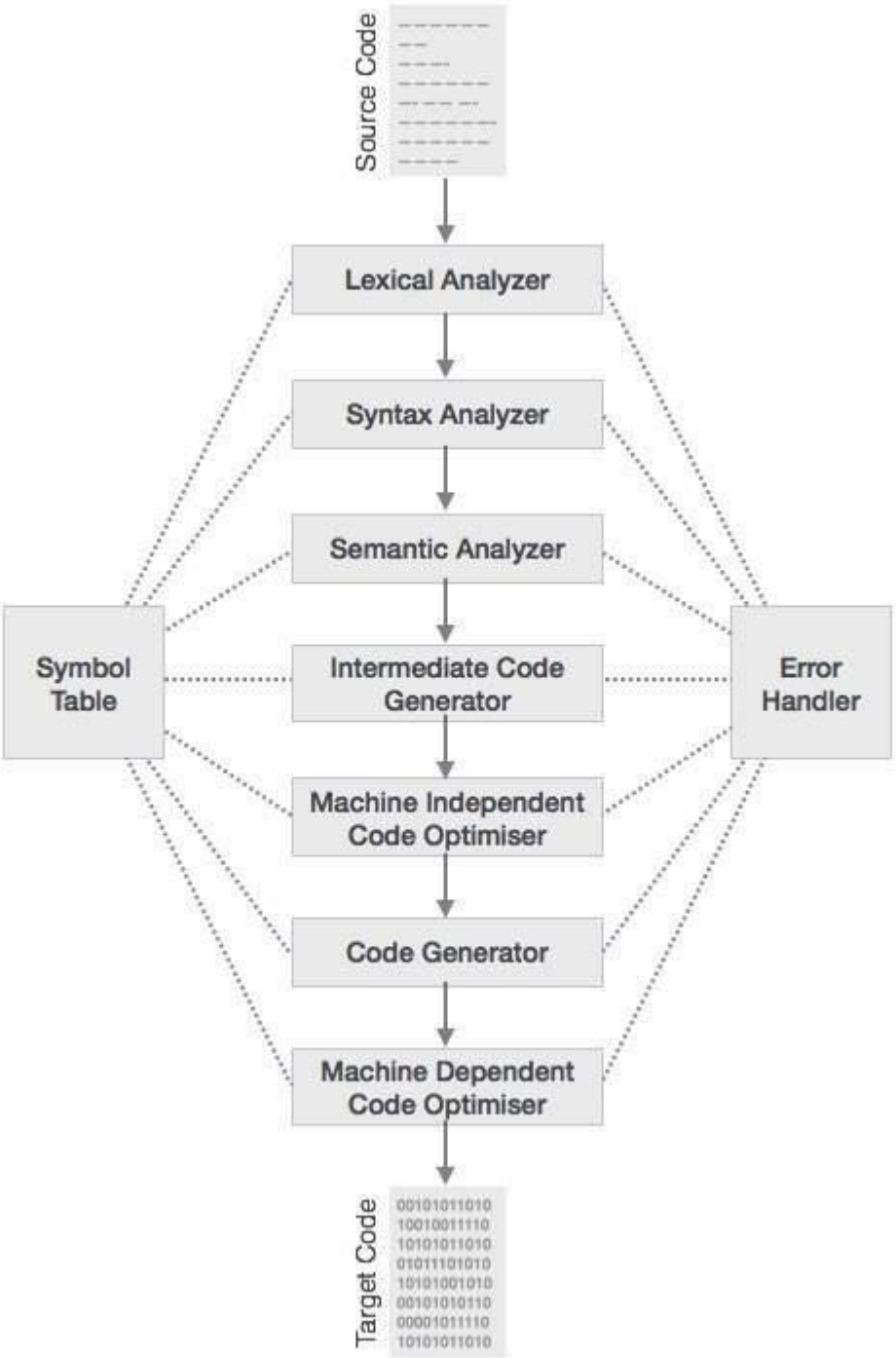
Structure of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

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 Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

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

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

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.

For example, suppose a source program contains the assignment statement
positio n = initia l + rat e * 60 (1.1)
The characters in this assignment could be grouped into the following lexemes
and mapped into the following tokens passed on to the syntax analyzer:
1. positio n is a lexeme that would be mapped into a token (id, 1), where id
is an abstract symbol standing for identifier and 1 points to the symbol table entry for position . The symbol-table entry for an identifier holds

information about the identifier, such as its name and type.
2. The assignment symbol = is a lexeme tha t is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. initia l is a lexeme tha t is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial .

4. + is a lexeme tha t is mapped into the token (+).
5. rat e is a lexeme that is mapped into the token (id, 3), where 3 points to the symbol-table entry for rate .
6. * is a lexeme that is mapped into the token (*).
7. 60 is a lexeme that is mapped into the token (60). 1
Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens
(id,l) <=) (id, 2) (+) (id, 3) (*) (60)
In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

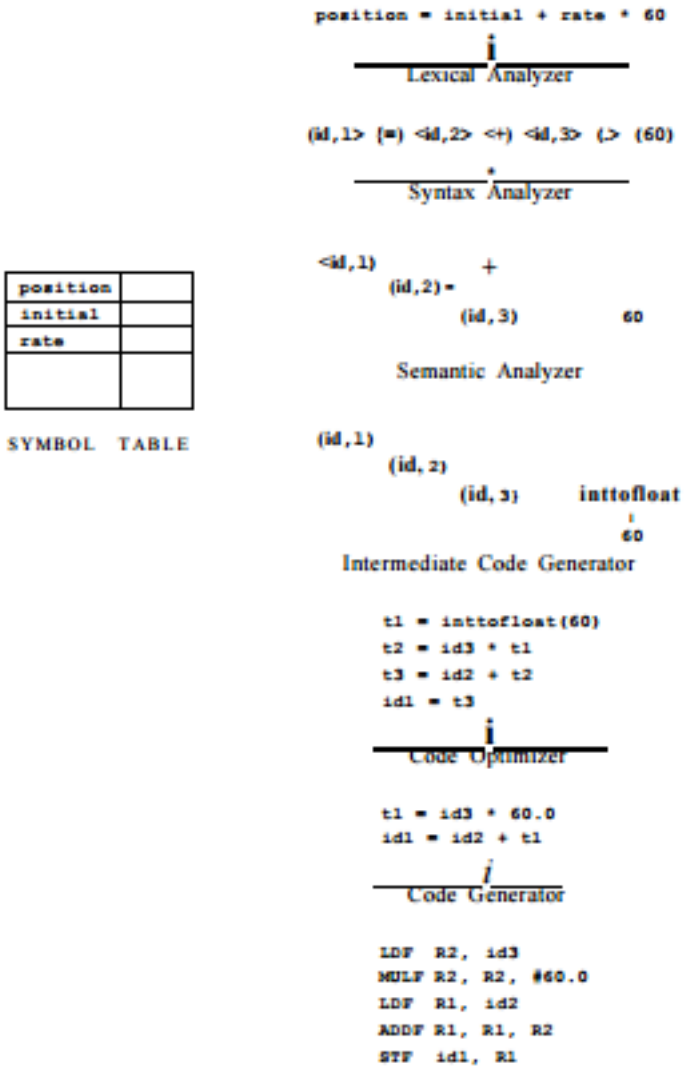


Figure 1.7: Translation of an assignment statement

Note:

Common programming errors can occur at many different levels.

- *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.
- *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).
- *Semantic* errors include type mismatches between operators and operands. An example is a `return` statement in a Java method with result type void.
- *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

Lexical Error:
A **lexical error** occurs when a sequence of characters does not match the pattern of any token. It typically happens during the execution of a program.
Types of Lexical Error:
Types of lexical error that can occur in a lexical analyzer are as follows:

1. Exceeding length of identifier or numeric constants.
Example: #include <iostream>
using namespace std;

int main() {

int a=2147483647 +1;
return 0;
}
This is a lexical error since signed integer lies between −2,147,483,648 and 2,147,483,647

2. Appearance of illegal characters
printf("Geeksforgeeks");$

3. Unmatched string
Example:

- C++

```
#include <iostream>
using namespace std;

int main() {
/* comment
```

```
    cout<<"GFG!";
    return 0;
}
```

This is a lexical error since the ending of comment "*/" is not present but the beginning is present.

4. Spelling Error

- C++

```
#include <iostream>
using namespace std;

int main() {

    int 3num= 1234;    /* spelling error as identifier
                    cannot start with a number*/
    return 0;
}
```

5. Replacing a character with an incorrect character.

- C++

```
#include <iostream>
using namespace std;

int main() {

    int x = 12$34; /*lexical error as '$' doesn't
                belong within 0-9 range*/
    return 0;
}
```

Other lexical errors include

6. Removal of the character that should be present.

- C++

```
#include <istream> /*missing 'o' character
                hence lexical error*/
using namespace std;

int main() {

    cout<<"GFG!";
    return 0;
}
```

7. Transposition of two characters.

- C++

```
#include <iostream>
using namespace std;

int mian()
{
   /* spelling of main here would be treated as an lexical
      error and won't be considered as an identifier,
      transposition of character 'i' and 'a'*/
   cout << "GFG!";
   return 0;
}
```

Regular Expression:

The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as
  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate { e, x, xx, xxx, xxxx, … }
- x+ means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.
  [a-z] is all lower-case alphabets of English language.
  [A-Z] is all upper-case alphabets of English language.
  [0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = $(sign)^{?}(digit)^{+}$

Identifier = (letter)(letter | digit)*

3.1.2 Tokens, Patterns, and Lexemes When discussing lexical analysis, we use three related but distinct terms: • A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name. • A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings. • A lexeme is a sequence of characters in the source program tha t matches the pattern for a token and is identified by the lexical analyzer as an instance of tha t token.

3.1.2 Tokens, Patterns, and Lexemes

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

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.
- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.
- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

A language that can be defined by a regular expression is called a *regular set.* If two regular expressions r and *s* denote the same regular set, we say they are *equivalent* and write *r* = *s.* For instance, **(a|b) = (b|a).** There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions *r,* *s,* and *t.*

LAW	DESCRIPTION
r|s = s|r	| is commutative
r|(s|t) = (r|s)|t	| is associative
r(st) = (rs)t	Concatenation is associative
r(s|t) = rs|rt; (s|t)r = sr|tr	Concatenation distributes over |
er = re = r	e is the identity for concatenation
*r** = (r|e)*	e is guaranteed in a closure
	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

Regular Definitions

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

letter- -+ **A | B | - - - | Z | a | b | - - - | z | _**
digit -> **0 j 1 j • • • j 9**
id —)• *letter- (letter- \ digit)**

Example 3.6: Unsigned numbers (integer or floating point) are strings such as **5280, 0.01234, 6.336E4,** or **1.89E-4.** The regular definition

digit **0 | 1 9**
digits -+ *digit digit**
optionalFraction -)• . *digits* | e
optionalExponent **(E (+ | - | e)** *digits* **) | e**
number -> *digits optionalFraction optionalExponent*

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent,* if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match **1.**, but does match **1.0.** •

Extensions of Regular Expressions

1. *One or more instances.* The unary, postfix operator ' represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then (r)' denotes the language (L(r))'. The operator ' has the same precedence and associativity as the operator *. Two useful algebraic laws, r* = r'\e and r' = rr* = r*r relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to r|e, or put another way, *L(r?)* = *L(r)* U {e}. The ? operator has the same precedence and associativity as * and +.

3. *Character classes.* A regular expression *aifal* • • • \a„ where the *a^s* are each symbols of the alphabet, can be replaced by the shorthand *[aia,2 • • - a].* More importantly, when 0 1 , 0 2 , . . . , a „ f ° ' " a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by oi-a„, that is, just the first and last separated by a hyphen. Thus, **[abc]** is shorthand for **a|b|c,** and **[a-z]** is shorthand for **a|b|.--|z.**

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

```
letter.   ->   [A-Za-z_]
digit     ->   [0-9]
id        ->   letter- ( letter \ digit )*
```

The regular definition of Example 3.6 can also be simplified:

```
digit    ->   [0-9]
digits   —>•  digit+
number   -»•  digits (. digits)?  ( E [+-]? digits )?
```

Recognition of Tokens:

```
stmt   —>   if expr then stmt
       |    if expr then stmt else stmt
       I    e
expr   —»•   term relop term
       |     term
term   ->    id
       |     number
```

Figure 3.10: A grammar for branching statements

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

Figure 3.11: Patterns for tokens of Example 3.8

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

Figure 3.12: Tokens, their patterns, and attribute values

Transition Diagrams

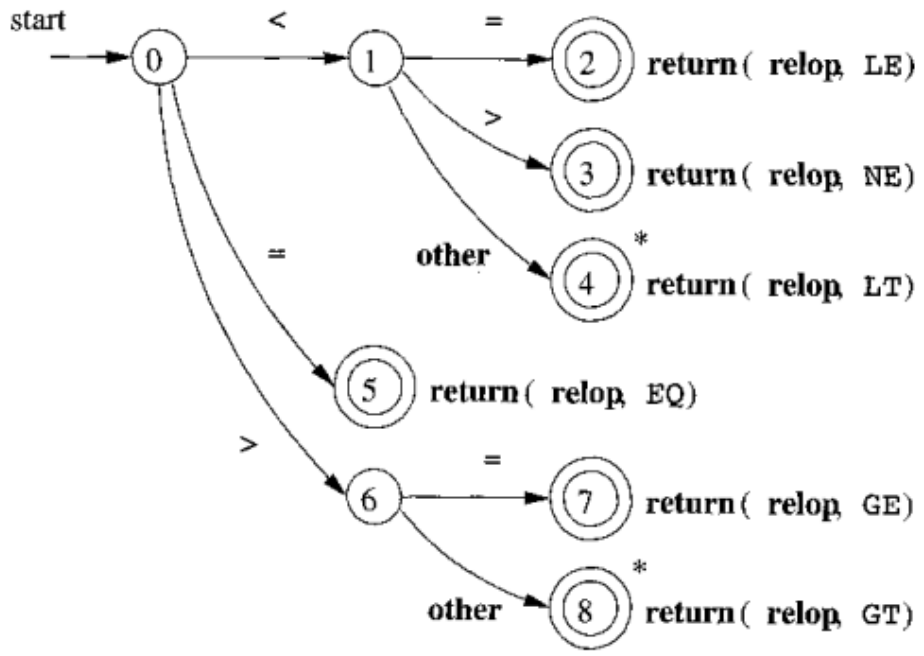


Figure 3.13: Transition diagram for **relop**

3.4.2 Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords if, **then,** and **else** of our running example.

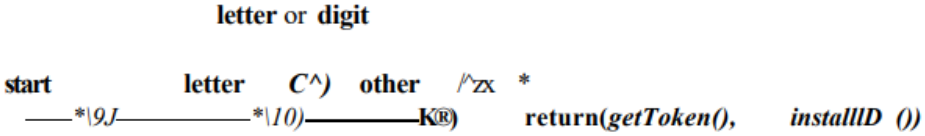


Figure 3.14: A transition diagram for **id**'s and keywords

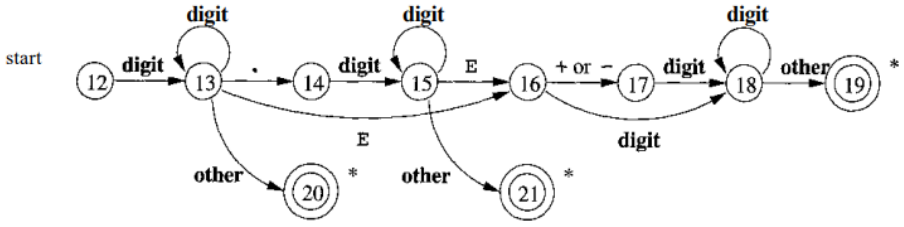


Figure 3.16: A transition diagram for unsigned numbers

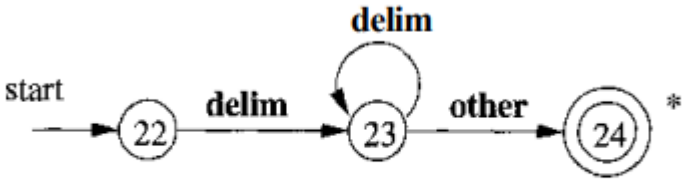


Figure 3.17: A transition diagram for whitespace

Syntax directed translation

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So we can say that

1. Grammar + semantic rule = SDT (syntax directed translation)
2. In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
3. In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
4. In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

Parse tree for SDT:

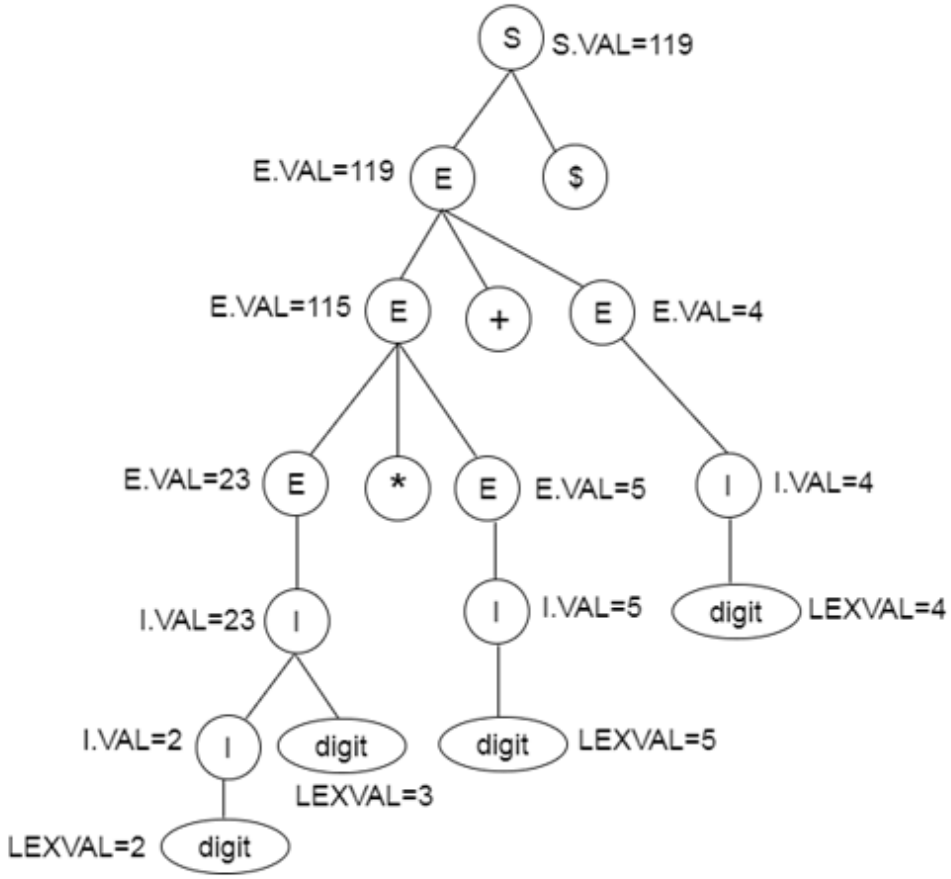


Fig: Parse tree

S – attributed and L – attributed SDTs in Syntax directed translation

- Before coming up to S-attributed and L-attributed SDTs, here is a brief intro to Synthesized or Inherited attributes **Types of attributes –** Attributes may be of two types – Synthesized or Inherited.
  1. **Synthesized attributes –** A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production). The non-terminal concerned must be in the head (LHS) of production. For e.g. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
  2. **Inherited attributes –** An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production). The non-terminal concerned must be in the body (RHS) of production. For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute because A is a parent here, and C is a sibling.

  Now, let's discuss about S-attributed and L-attributed SDT.

1. **S-attributed SDT :**
   - If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
   - S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
   - Semantic actions are placed in rightmost place of RHS.
2. **L-attributed SDT:**
   - If an 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 : S->ABC, Here attribute B can only obtain its value either from the parent – S or its left sibling A but It can't inherit from its right sibling C. Same goes for A & C – A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.

Compiler Design | Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form s = f(b_1, b_2, ……b_k) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E_1 + T { E.val = E_1.val + T.val}

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of attributes – There are two types of attributes:

1. Synthesized Attributes – These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

Example:

E --> E_1 + T { E.val = E_1.val + T.val}

In this, E.val derive its values from E_1.val and T.val

Computation of Synthesized Attributes –

- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

Example: Consider the following grammar

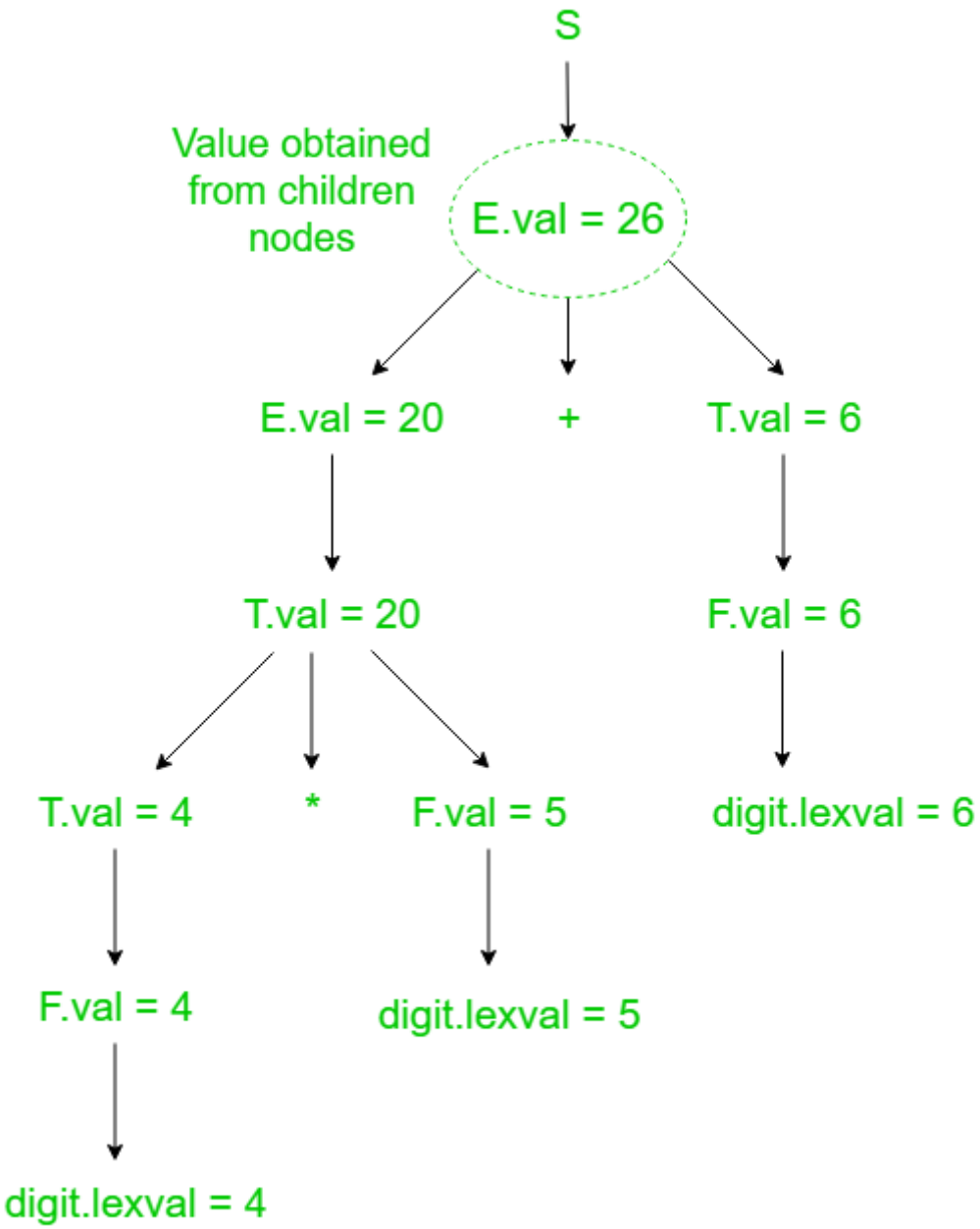
S --> E

E --> E_1 + T
E --> T
T --> T_1 * F
T --> F
F --> digit

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val . Hence, T.val = 4 * 5 = 20

Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

2. Inherited Attributes – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

Example:

A --> BCD { C.in = A.in, C.type = B.type }

Computation of Inherited Attributes –

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Example: Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

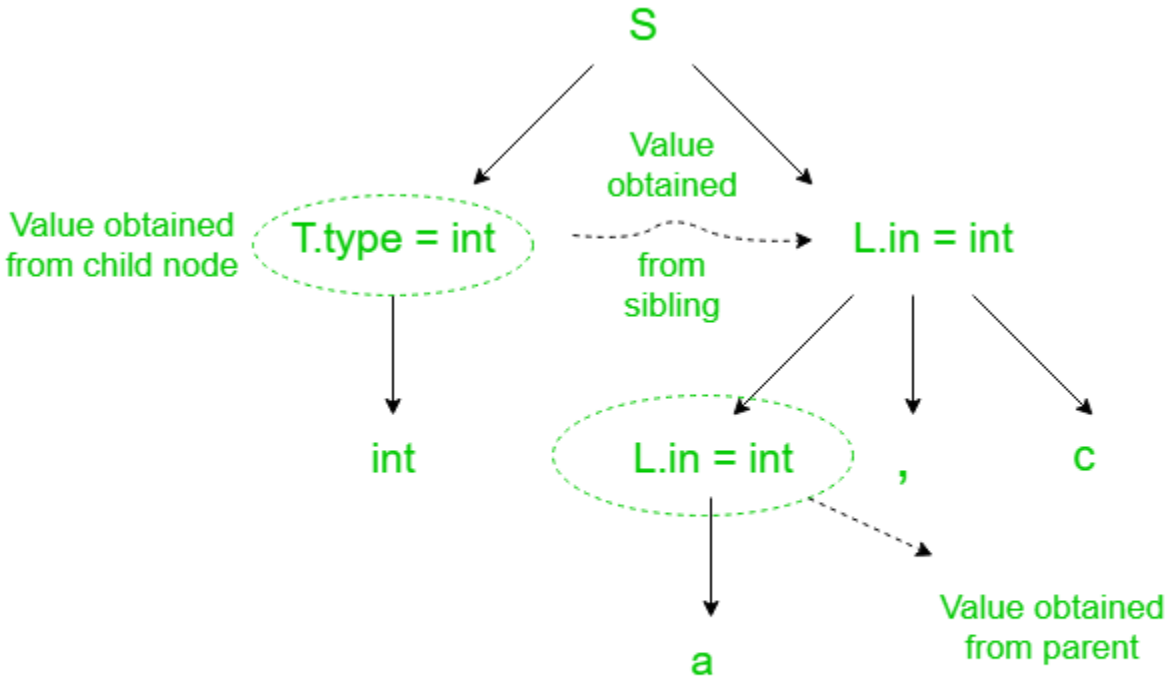
L --> L_1, id

L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

Symbol Table in Compiler

Definition

The symbol table is defined as the set of Name and Value pairs.

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built-in lexical and syntax analysis phases.
- The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.
- It is used by various phases of the compiler as follows:-
  1. **Lexical Analysis:** Creates new table entries in the table, for example like entries about tokens.
  2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
  3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
5. **Code Optimization:** Uses information present in the symbol table for machine-dependent optimization.
6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in the symbol table is associated with attributes that support the compiler in different phases.

Use of Symbol Table-

The symbol tables are typically used in compilers. Basically compiler is a program which scans the application program (for instance: your C program) and produces machine code.

During this scan compiler stores the identifiers of that application program in the symbol table. These identifiers are stored in the form of name, value address, type.

Here the name represents the name of identifier, value represents the value stored in an identifier, the address represents memory location of that identifier and type represents the data type of identifier.

Thus compiler can keep track of all the identifiers with all the necessary information.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by the compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, a pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Operations on Symbol Table :

Following operations can be performed on symbol table-

1. Insertion of an item in the symbol table.

2. Deletion of any item from the symbol table.

3. Searching of desired item from symbol table.

Implementation

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list
- Hash table
- Binary search tree

Symbol table are mostly implemented as hash table.

What is Parsing in Compiler Design?

The process of transforming the data from one format to another is called Parsing. This process can be accomplished by the parser. The parser is a component of the translator that helps to organise linear text structure following the set of defined rules which is known as grammar.

Types of Parsing:

There are two types of Parsing:

- The Top-down Parsing
- The Bottom-up Parsing

- **Top-down Parsing:** When the parser generates a parse with top-down expansion to the first trace, the left-most derivation of input is called top-down parsing. The top-down parsing initiates with the start symbol and ends on the terminals. Such parsing is also known as predictive parsing.
    - **Recursive Descent Parsing:** Recursive descent parsing is a type of top-down parsing technique. This technique follows the process for every terminal and non-terminal entity. It reads the input from left to right and constructs the parse tree from right to left. As the technique works recursively, it is called recursive descent parsing.
    - **Back-tracking:** The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules.

- **Bottom-up Parsing:** The bottom-up parsing works just the reverse of the top-down parsing. It first traces the rightmost derivation of the input until it reaches the start symbol.
    - **Shift-Reduce** Parsing: Shift-reduce parsing works on two steps: Shift step and Reduce step.
        - **Shift step:** The shift step indicates the increment of the input pointer to the next input symbol that is shifted.
        - **Reduce Step:** When the parser has a complete grammar rule on the right-hand side and replaces it with RHS.
    - **LR Parsing:** LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to right tracing, and R stands for the right to left tracing.

Left Recursion and Left Factoring:

We then consider several transformations tha t could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques — left-recursion elimination and left factoring — are useful for rewriting grammars so they become suitable for top-down parsing.

Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Eliminating Ambiguity:

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for an input string.
An ambiguous grammar or string can have multiple meanings. Ambiguity is often treated as a grammar bug, in programming languages, this is mostly unintended.

What is the dangling else problem with example?

It is an ambiguity in which it is not clear which if statement is associated with the else clause. For example, if X then if Y then E1 else E2. It is not unacceptable in programming. From the above statement, it is clear that there are two if statements and a single else statement.
The dangling else problem in syntactic ambiguity. It occurs when we use *nested if*. When there are multiple *"if"* statements, the *"else"* part doesn't get a clear view with which "*if*" it should combine.
For example:

```
if (condition) {

}

if (condition 1) {

}

 if (condition 2) {

}

   else
```

{

}

In the above example, there are multiple "*ifs"* with multiple conditions and here we want to pair the outermost if with the else part. But the else part doesn't get a clear view with which "*if"* condition it should pair. This leads to inappropriate results in programming.

Resolving Dangling-else Problem

The *first* way is to design non-ambiguous programming languages.
Secondly, we can resolve the dangling-else problems in programming languages by using braces and indentation.

Elimination of Left Recursion:

$$\begin{array}{lll} E & & E + T \mid T \\ T & \text{-)>} & T * F \mid F \\ F & & (E) \mid \text{id} \end{array}$$

Example 4.17: The non-left-recursive expression grammar (4.2), repeated here,

$$\begin{array}{l} E \text{ -> } T\ E' \\ E' \text{ ^ } + T\ E' \\ T \text{ ^ } FT' \\ T' \text{ -> } * F\ T \\ F\text{->}(E) \mid \textbf{id} \end{array}$$

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions E -» $E + T \mid T$ are replaced by E -> $T\ E'$ and E' -»• + T E' \ e. The new productions for T and T' are obtained similarly by eliminating immediate left recursion. •

Left Factoring:
Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead, consider this example:

Left Factoring is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more productions.
For example, going from:

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

to:

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta \mid \gamma$

In this case, a parser with a look-ahead will always choose the right production.

LL (1) Parsing or Predictive Parsing

LL (1) stands for, left to right scan of input, uses a Left most derivation, and the parser takes 1 symbol as the look ahead symbol from the input in taking parsing action decision. A non recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation.

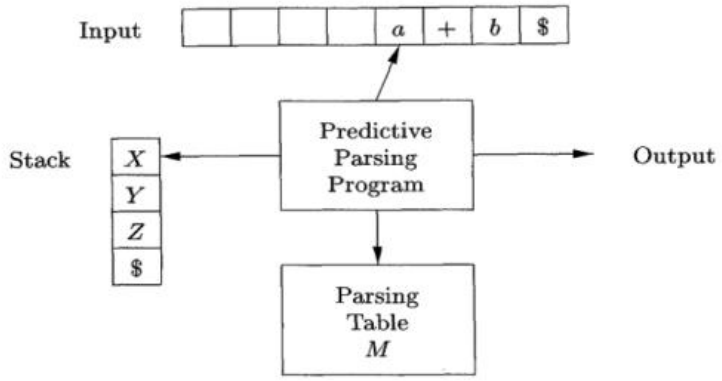


Figure 2.2: Model for table driven parsing

The Steps Involved In constructing an LL(1) Parser are:

1. Write the Context Free grammar for given input String
2. Check for Ambiguity. If ambiguous remove ambiguity from the grammar
3. Check for Left Recursion. Remove left recursion if it exists.
4. Check For Left Factoring. Perform left factoring if it contains common prefixes in more than one alternates.
5. Compute FIRST and FOLLOW sets
6. Construct LL(1) Table
7. Using LL(1) Algorithm generate Parse tree as the Output

Example : - Is the following grammar left recursive? If so, find a non left recursive grammar equivalent to it.

E → E + T | T

T → T * F | F

F → -E | (E) | id

Yes ,the grammar is left recursive due to the first two productions which are satisfying the general form of Left recursion, so they can be rewritten after removing left recursion from **E → E + T**, and **T→ T * F** is

E → TE′
E′→ +T E′ | €
T → F T′
T′→ *F T′ | €
F → (E) | id

FIRST and FOLLOW:

Computation of FIRST:

FIRST function computes the set of terminal symbols with which the right hand side of the productions begin. To compute FIRST (A) **for all grammar symbols**, apply the following rules until no more terminals or **Є** can be added to any FIRST set.

1. If A is a terminal, then FIRST {A} = {A}.
2. If A is a Non terminal and A->X1X2…Xi
   FIRST(A)=FIRST(X1) if X1is not null, if X1 is a non terminal and X1->Є, add FIRST(X2) to FIRST(A), if X2-> Є add FIRST(X3) to FIRST(A), … if Xi->Є , i.e., all X_i's for i=1..i are null, add Є FIRST(A).
3. If A ->Є is a production, then add Є to FIRST (A).

Computation Of FOLLOW:

Follow (A) is nothing but the set of terminal symbols of the grammar that are immediately following the Non terminal A. If **a** is to the immediate right of non terminal A, then Follow(A)= {**a**}. To compute FOLLOW (A) for **all non terminals** A, apply the following rules until no more symbols can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where *S* is the start symbol, and $ is the input right end marker.
2. If there is a production A-> αBβ, then everything in FIRST (β) except Є is in FOLLOW(B).
3. If there is a production A->αB or a production A-> αBβ with FIRST(β) contains Є, then FOLLOW (B) = FOLLOW (A).

Example: - Compute the FIRST and FOLLOW values of the expression grammar

1. **E → TE′**
2. **E′→+TE′ | Є**
3. **T → FT′**
4. **T′ →*FT′ | Є**
5. **F → (E) | id**

Computing FIRST Values:

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

FIRST (E′) = {+, Є}

FIRST (T′) = {*, Є}

Computing FOLLOW Values:

FOLLOW (E) = { $,), } Because it is the start symbol of the grammar.

FOLLOW (E′) = {FOLLOW (E)} satisfying the 3rd rule of FOLLOW()
= { $,) }

FOLLOW (T) = { FIRST E′} It is Satisfying the 2nd rule.
U { FOLLOW(E′) }
= {+, FOLLOW (E′)}
= { +, $,) }

FOLLOW (T′) = { FOLLOW(T)} Satisfying the 3rd Rule
= { +, $,) }

FOLLOW (F) = { FIRST (T′) } It is Satisfying the 2nd rule.
U { FOLLOW(E′) }
= {*, FOLLOW (T)}
= { *, +, $,)}

NON TERMINAL	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E′	{ +, Є }	{ $,) }
T	{ (, id}	{ +, $,) }
T′	{*, Є}	{ +, $,) }
F	{ (, id}	{ *, +, $,) }

Table 2.1: FIRST and FOLLOW values

Constructing Predictive Or LL (1) Parse Table:

It is the process of placing the all productions of the grammar in the parse table based on the FIRST and FOLLOW values of the Productions.

The rules to be followed to Construct the Parsing Table (M) are :

1. For Each production A-> α of the grammar, do the bellow steps.
2. For each terminal symbol 'a' in FIRST (α), add the production A-> α to M [A, a].
3. i. If Є is in FIRST (α) add production A->α to M [A, b], where b is all terminals in FOLLOW (A).
   ii. If Є is in FIRST(α) and $ is in FOLLOW (A) then add production A->α to M [A, $].
4. Mark other entries in the parsing table as error .

NON-TERMINALS	INPUT SYMBOLS					
	+	*	(	)	id	$
E			E TE′		E id	
E′	E′ +TE′			E′ Є		E′ Є
T			T FT′		T FT′	
T′	T′ Є	T′ *FT′		T′ Є		T′ Є
F			F (E)		F id	

Table 2.2: LL (1) Parsing Table for the Expressions Grammar

Note: if there are no multiple entries in the table for single a terminal then grammar is accepted by LL(1) Parser.

LL (1) Parsing Algorithm:

The parser acts on basis on the basis of two symbols

i. A, the symbol on the top of the stack
ii. a, the current input symbol

There are three conditions for A and 'a', that are used fro the parsing program.

1. If A=a=$ then parsing is Successful.
2. If A=a≠$ then parser pops off the stack and advances the current input pointer to the next.
3. If A is a Non terminal the parser consults the entry M [A, a] in the parsing table. If M[A, a] is a Production A-> $X_1X_2..X_n$, then the program replaces the A on the top of the Stack by $X_1X_2..X_n$ in such a way that X_1 comes on the top.

STRING ACCEPTANCE BY PARSER:

If the input string for the parser is **id + id * id,** the below table shows how the parser accept the string with the help of Stack.

Stack	Input	Action	Comments
$E	id+ id* id $	E TE`	E on top of the stack is replaced by TE`
$E`T	id+ id*id $	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id+ id*id $	F id	F on top of the stack is replaced by id
$E`T`id	id+ id*id $	pop and remove id	Condition 2 is satisfied
$E`T`	+id*id$	T` Є	T` on top of the stack is replaced by Є
$E`	+id*id$	E` +TE`	E` on top of the stack is replaced by +TE`
$E`T+	+id*id$	Pop and remove +	Condition 2 is satisfied
$E`T	id*id$	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id*id$	F id	F on top of the stack is replaced by id
$E`T`id	id * id$	pop and remove id	Condition 2 is satisfied
$E`T`	*id$	T` *FT`	T` on top of the stack is replaced by *FT`
$E`T`F*	*id$	pop and remove *	Condition 2 is satisfied
$E`T`F	id$	F id	F on top of the stack is replaced by id
$E`T`id	id$	Pop and remove id	Condition 2 is satisfied
$E`T`	$	T` Є	T` on top of the stack is replaced by Є
$E`	$	E` Є	E` on top of the stack is replaced by Є
$	$	Parsing is successful	Condition 1 satisfied

Table2.3 : Sequence of steps taken by parser in parsing the input **token stream id+ id* id**

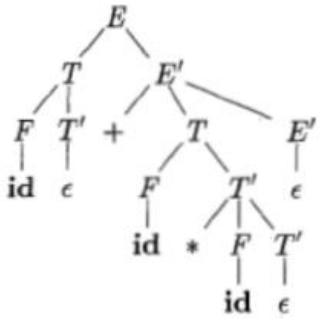


Figure 2.7: Parse tree for the input id + id* id

ERROR HANDLING (RECOVERY) IN PREDICTIVE PARSING:

In table driven predictive parsing, it is clear as to which terminal and Non terminals the parser expects from the rest of input. An error can be detected in the following situations:
1. When the terminal on top of the stack does not match the current input symbol.
2. when Non terminal A is on top of the stack, a is the current input symbol, and M[A, a] is empty or error
The parser recovers from the error and continues its process.

RECURSIVE DESCENT PARSING :

A recursive-descent parsing program consists of a set of recursive procedures, one for each non terminal. Each procedure is responsible for parsing the constructs defined by its non terminal, Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.
If the given grammar is
E -> TE′
E′ -> +TE′ | €
T -> FT′
T′ -> *FT′ | €
F-> (E) | id
Reccursive procedures for the recursive descent parser for the given grammar are given below.

```
procedure E( )
{
T( );
E′( );
}
procedure T ( )
{
F( );
T′( );
}
Procedure E′( )
{
if input = +
{
advance( );
T ( );
E′( );
return true;
}
else error;
}
procedure T′( )
{
if input = *
{
advance( );
F ( );
T′( );
return true;
}
else return error;
}
procedure F( )
```

```
{
if input = (
{
advance( );
E ( );
if input = )
advance( );
return true;
}
else if input = id
{
advance( );
return true;
}
else return error;
}
advance()
{
input = next token;
}
```

BOTTOM-UP PARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It involves "reducing an input string 'w' to the Start Symbol of the grammar. in each reduction step, a perticular substring matching the right side of the production is replaced by symbol on the left of that production and it is the Right most derivation. For example consider the following Grammar:

E → E+T|T

T → T*F

F → (E)|id

Bottom up parsing of the input string **"id * id "**is as follows:

INPUT STRING	SUB STRING	REDUCING PRODUCTION
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

Parse Tree representation is as follows:

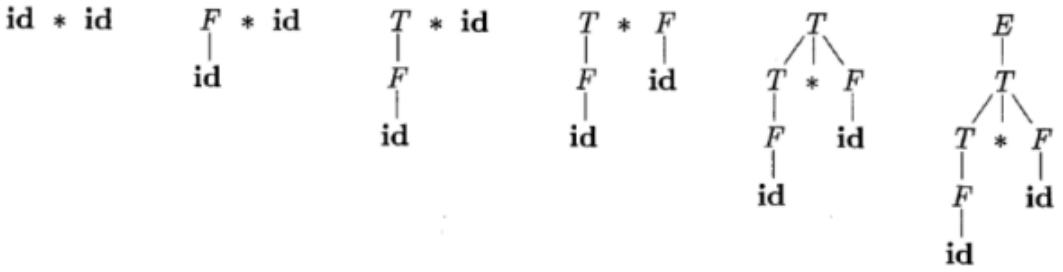


Figure 3.1 : A Bottom-up Parse tree for the input String "id*id"

Bottom up parsing is classified in to 1. Shift-Reduce Parsing, 2. Operator Precedence parsing , and 3. [Table Driven] L R Parsing

i. SLR(1)
ii. CALR (1)
iii.LALR(1)

SHIFT-REDUCE PARSING:

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed, We use $ to mark the bottom of the stack and also the right end of the input. And it makes use of the process of shift and reduce actions to accept the input string. Here, the parse tree is Constructed bottom up from the leaf nodes towards the root node.

When we are parsing the given input string, if the match occurs the parser takes the reduce action otherwise it will go for shift action. And it can accept ambiguous grammars also.

For example, consider the below grammar to accept the input string "id * id", using S-R parser

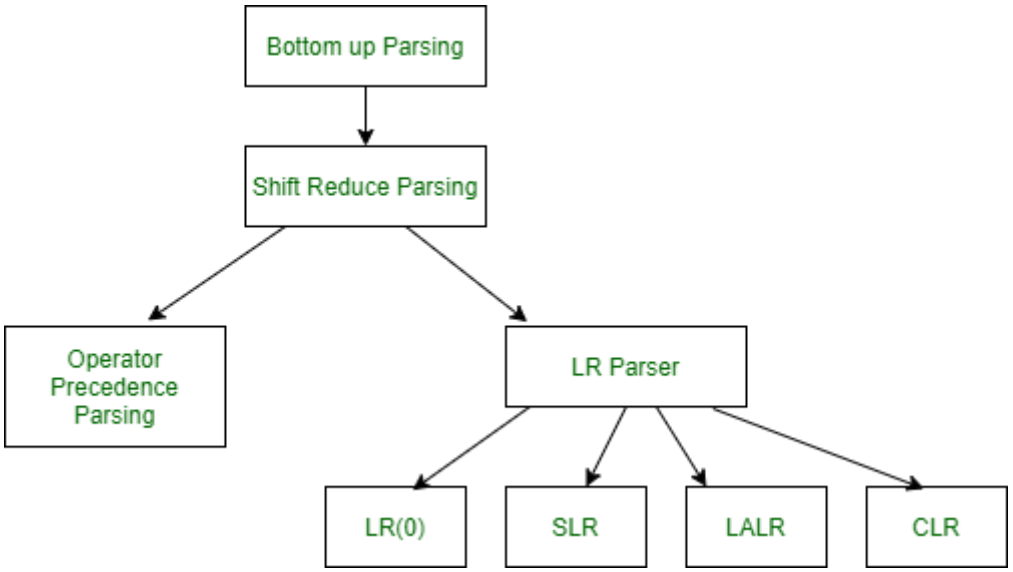
E → E+T|T
T → T*F | F
F → (E)|id

Actions of the Shift-reduce parser using Stack implementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	Reduce with F→d
$F	*id$	Reduce with T→F
$T	*id$	Shift
$T*	id$	Shift
$T*id	$	Reduce with F→id
$T*F	$	Reduce with T→T*F
$T	$	Reduce with E→T
$E	$	Accept

LR parser :

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.



Description of LR parser :

The term parser LR(k) parser, here the L refers to the left-to-right scanning, R refers to the rightmost derivation in reverse and k refers to the number of unconsumed "look ahead" input symbols that are used in making parser decisions. Typically, k is 1 and is often omitted. A context-free grammar is called LR (k) if the LR (k) parser exists for it. This first reduces the sequence of tokens to the left. But when we read from above, the derivation order first extends to non-terminal.

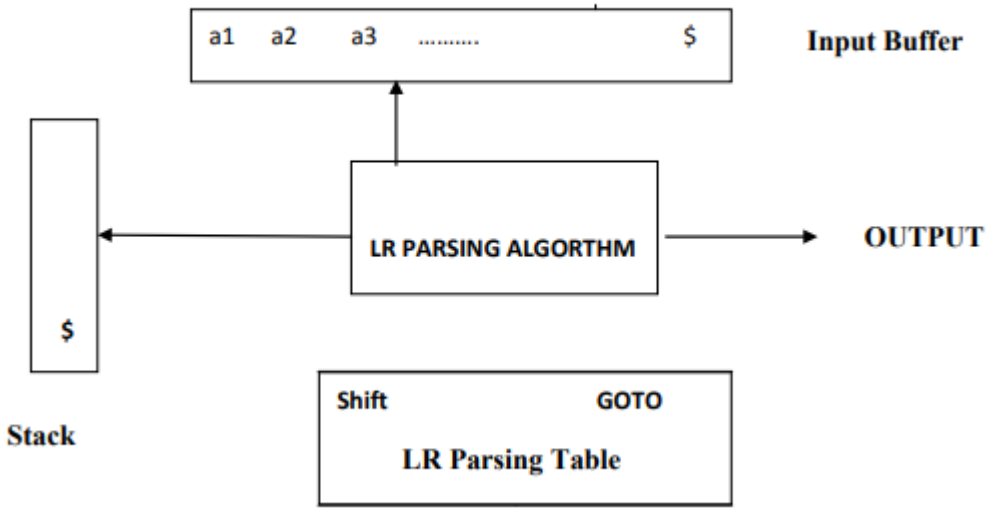


Figure 3.3: Components of LR Parsing

LR Parser Consists of

- An input buffer that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.
- A stack containing a sequence of grammar symbols with a $ at the bottom of the stack, which initially contains the Initial state of the parsing table on top of $.
- A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] and it contains two parts

1. **ACTION Part**
   The ACTION part of the table is a two dimensional array indexed by state and the input symbol, i.e. **ACTION**[state][input], An action table entry can have one of following four kinds of values in it. They are:
   1. Shift X, where X is a State number.
   2. Reduce X, where X is a Production number.
   3. Accept, signifying the completion of a successful parse.
   4. Error entry.
2. **GO TO Part**
   The GO TO part of the table is a two dimensional array indexed by state and a Non terminal, i.e. GOTO[state][NonTerminal]. A GO TO entry has a state number in the table.

∑ A parsing Algorithm uses the current State X, the next input symbol 'a' to consult the entry at action[X][a]. it makes one of the four following actions as given below:

1. If the action[X][a]=shift Y, the parser executes a shift of Y on to the top of the stack and advances the input pointer.
2. If the action[X][a]= reduce Y (Y is the production number reduced in the State X), if the production is Y->β, then the parser pops 2*β symbols from the stack and push Y on to the Stack.
3. If the action[X][a]= accept, then the parsing is successful and the input string is accepted.
4. If the action[X][a]= error, then the parser has discovered an error and calls the error routine.

The parsing is classified in to

1. LR (0)
2. Simple LR (1)
3. Canonical LR (1)
4. Look ahead LR (1)

LR (1) Parsing: Various steps involved in the LR (1) Parsing:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment production
4. Create Canonical collection of LR (0) items
5. Draw DFA
6. Construct the LR (0) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

Augment Grammar

The Augment Grammar G`, is G with a new starting symbol S` an additional production S` →S. this helps the parser to identify when to stop the parsing and announce the acceptance of the input. The input string is accepted if and only if the parser is about to reduce by S→S. For example let us consider the Grammar below:

E → E+T|T
T → T*F
F → (E) | id **the Augment grammar G` is Represented by**

E`→ E
E → E+T|T
T → T*F
F → (E) | id

NOTE: Augment Grammar is simply adding one extra production by preserving the actual meaning of the given Grammar G.

Example :

0.	**E`→ E**		**E`→• E**
1.	**E→ E+T**	**LR (0) items for the Grammar is**	**E →•E+T**
2.	**T→ F**		**T→ •F**
3.	**T→ T*F**		**T→ •T*F**
4.	**F→ (E)**		**F→ • (E)**
5.	**F→ id**		**F→ • id**

Closure (I_0) State

Add **E `→ • E** in I_0 State

Since, the '•' symbol in the Right hand side production is followed by A Non terminal E. So, add productions starting with E in to Io state. So, the state becomes

E ` • E →

0. E→ •E+T

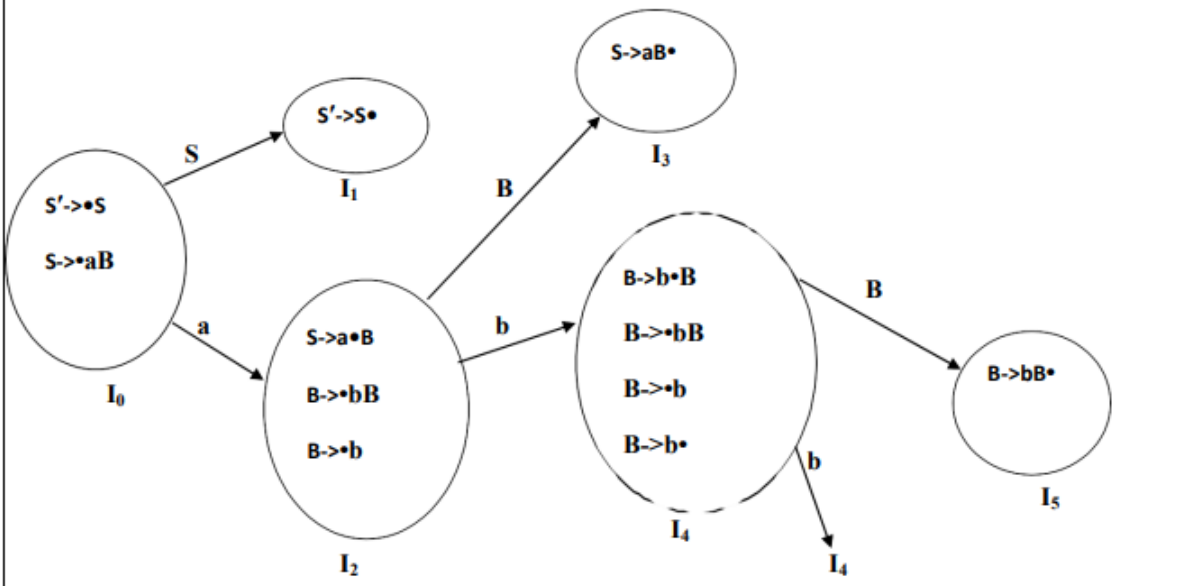
1. T → •F

The 1st and 2nd productions are satisfies the 2nd rule. So, add productions which are starting with E and T in I_0

Note: once productions are added in the state the same production should not added for the 2nd time in the same state. So, the state becomes

0. E`→ •E
1. E→ •E+T
2. T→ •F
3. T→ •T*F
4. F→ •(E)
5. F→ • id

Drawing Finite State diagram DFA: Following DFA gives the state transitions of the parser and is useful in constructing the LR parsing table.



LR Parsing Table:

States	ACTION			GOTO	
	a	B	$	S	B
I_0	S_2			1	
I_1			ACC		
I_2		S_4			3
I_3	R_1	R_1	R_1		
I_4	R_3	S_4/R_3	R_3		5
I_5	R_2	R_2	R_2		

Note: if there are multiple entries in the LR (1) parsing table, then it will not accepted by the LR(1) parser. In the above table I_3 row is giving two entries for the single terminal value 'b' and it is called as Shift- Reduce conflict.

SLR Parser :

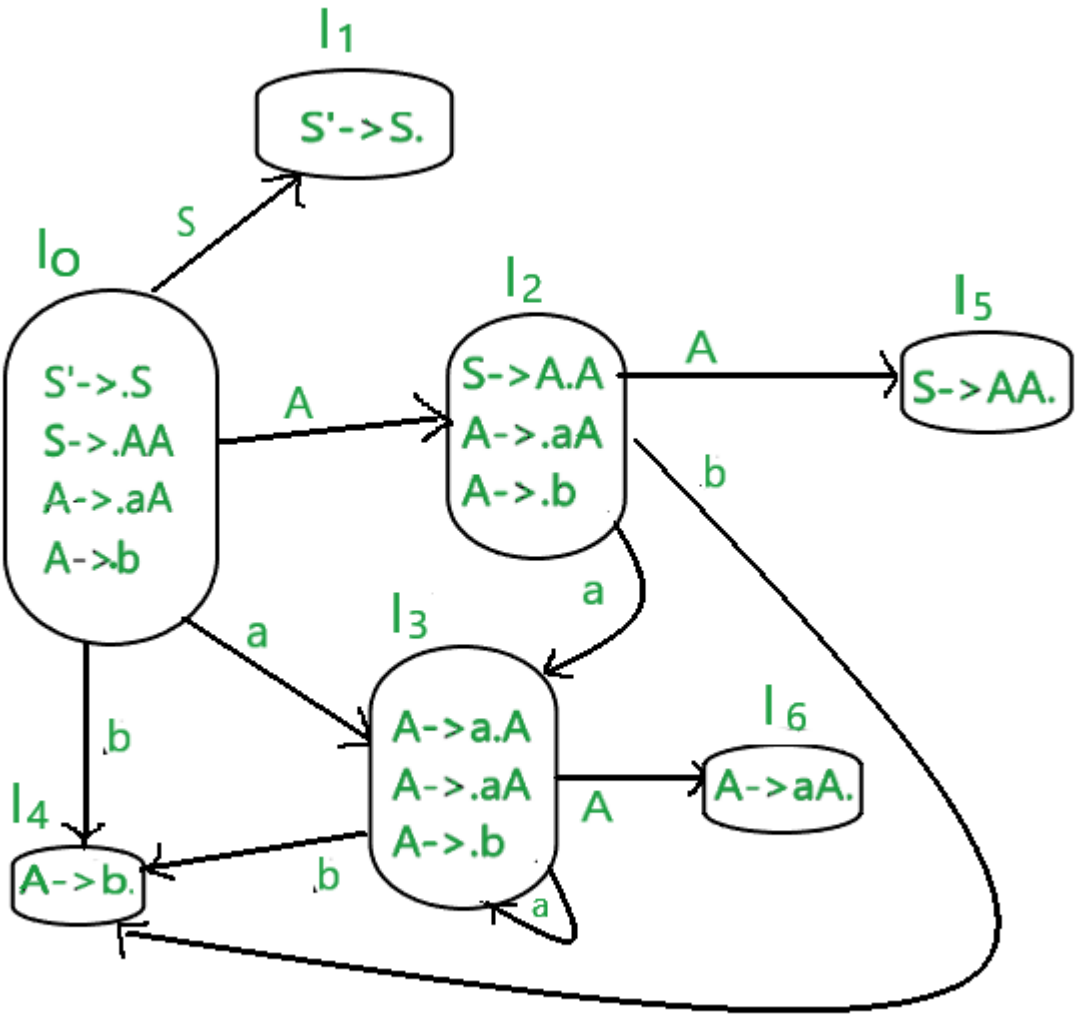
SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items

Steps for constructing the SLR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions:goto[list of terminals] and action[list of non-terminals] in the parsing table

EXAMPLE – Construct LR parsing table for the given context-free **grammar**
S–>AA
A–>aA|b
Solution:
STEP1 – Find augmented grammar
The augmented grammar of the given grammar is:-
S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

STEP2 – Find LR(0) collection of items
Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}.
The non-terminals of this grammar are {S,A}

RULE –
If any non-terminal has ‘ . ‘ preceding it, we have to write all its production and add ‘ . ‘ preceding each of its production.
RULE –
from each state to the next state, the ‘ . ‘ shifts to one place to the right.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ‘ . ‘ of 0th production is shifted towards the right of S(S’->S.). this state is the accepted state. S is seen by the compiler.
- Io goes to I2 when ‘ . ‘ of 1st production is shifted towards right (S->A.A) . A is seen by the compiler
- I0 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.
- I0 goes to I4 when ‘ . ‘ of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I2 goes to I5 when ‘ . ‘ of 1st production is shifted towards right (S->AA.) . A is seen by the compiler
- I2 goes to I4 when ‘ . ‘ of 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I2 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.
- I3 goes to I4 when ‘ . ‘ of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I3 goes to I6 when ‘ . ‘ of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler
- I3 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.

STEP3 –
Find FOLLOW of LHS of production
FOLLOW(S)=$
FOLLOW(A)=a,b,$

To find FOLLOW of non-terminals, please read follow set in syntax analysis.
STEP 4-
Defining 2 functions:goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

- $ is by default a nonterminal that takes accepting state.
- 0,1,2,3,4,5,6 denotes I0,I1,I2,I3,I4,I5,I6

- I0 gives A in I2, so 2 is added to the A column and 0 rows.
- I0 gives S in I1,so 1 is added to the S column and 1 row.
- similarly 5 is written in A column and 2 row, 6 is written in A column and 3 row.
- I0 gives a in I3 .so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4 .so S4(shift 4) is added to the b column and 0 row.
- Similarly, S3(shift 3) is added on a column and 2,3 row ,S4(shift 4) is added on b column and 2,3 rows.
- I4 is reduced state as ' . ' is at the end. I4 is the 3rd production of grammar(A–>.b).LHS of this production is A. FOLLOW(A)=a,b,$. write r3(reduced 3) in the columns of a,b,$ and 4th row
- I5 is reduced state as ' . ' is at the end. I5 is the 1st production of grammar(S–>.AA). LHS of this production is S.
  FOLLOW(S)=$. write r1(reduced 1) in the column of $ and 5th row
- I6 is a reduced state as ' . ' is at the end. I6 is the 2nd production of grammar(A–>.aA). The LHS of this production is A.
  FOLLOW(A)=a,b,$. write r2(reduced 2) in the columns of a,b,$ and 6th row

UNIT - V

Intermediate Code

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

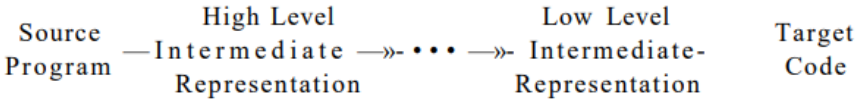


Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. For looping statements, for example, a syntax tree represents the components of a statement, whereas three-address code contains labels and jump instructions to represent the flow of control, as in machine language.

Types of Intermediate representations / forms: There are three types of intermediate

representation:-

1. Syntax Trees (**directed acyclic graph (hereafter called a DAG)**

2. Postfix notation

3. Three Address Code

1. Syntax Trees (**directed acyclic graph (hereafter called a DAG)**

DAG's can be constructed by using the same techniques tha t construct syntax trees. A directed acyclic graph (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is tha t a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

a + a * (b - c) + (b - c) * d

The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression b-c are represented by one node, the node labeled —. That node has two parents, representing its two uses in the subexpressions a*(b-c) and (b-c)*d. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression b-c. •

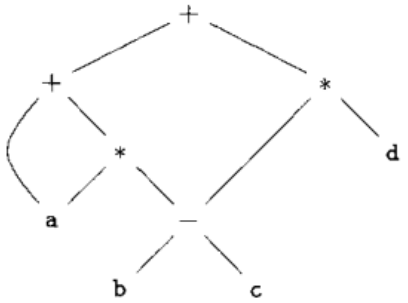


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

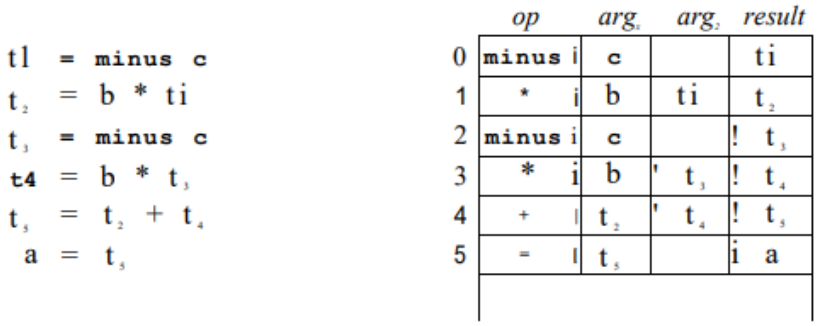
Semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees of for generating postfix notation.

Three-Address Code: A statement involving no more than three references(two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as a three address code. Three address statement is of form x = y op z, where x, y, and z will have address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.
Example: The three address code for the expression
a + b * c + d
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
where
T 1 , T 2 , T 3 are temporary variables.
Implementation of Three Address Code –
There are 3 representations of three address code namely
1. Quadruple
2. Triples

3. Indirect Triples

Example – Consider expression a = b * – c + b * – c.

The quadruples in Fig. 6.10(b) implement the three-address code in (a).



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

(a) Three-address code

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

For readability, we use actual identifiers like a, b, and c in the fields *arg_1*, *arg_2*, and *result* in Fig. 6.10(b), instead of pointers to their symbol-table entries.

1. Quadruple – It is a structure which consists of 4 fields namely **op, arg1, arg2 and result.** op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(b) Triples

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.
Example – Consider expression a = b * – c + b * – c

instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Figure 6.12: Indirect triples representation of three-address code

Code Optimization:

Elimination of unnecessary instructions in object code, or the replacement of one sequence ofinstructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization."

Optimizations are classified into two categories.

1. Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine

2. Machine dependant optimizations:
Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

Code Optimization is done in the following different ways:

1. Compile Time Evaluation:

C

```
(i)  x = 12.4
     y = x/2.3
     Evaluate x/2.3 as 12.4/2.3 at compile time.
```

2. Variable Propagation:

C

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4


//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

3. Constant Propagation:

- If the value of a variable is a constant, then replace the variable with the constant. The variable may not always be a constant.

Example:

C

```
(i)  A = 2*(22.0/7.0)*r
     Performs 2*(22.0/7.0)*r at compile time.
(ii)  x = 12.4
      y = x/2.3
      Evaluates x/2.3 as 12.4/2.3 at compile time.
(iii) int k=2;
      if(k) go to L3;
      It is evaluated as :
      go to L3 ( Because k = 2 which implies condition is always true)
```

4. Constant Folding:

- Consider an expression : a = b op c and the values b and c are constants, then the value of a can be computed at compile time.

Example:

C

```
#define k 5
x = 2 * k
y = k + 5

This can be computed at compile time and the values of x and y are :
 x = 10
 y = 10
```

Note: Difference between Constant Propagation and Constant Folding:

- In Constant Propagation, the variable is substituted with its assigned constant where as in Constant Folding, the variables whose values can be computed at compile time are considered and computed.

5. Copy Propagation:

- It is extension of constant propagation.
- After a is assigned to x, use a to replace x till a is assigned again to another variable or value or expression.
- It helps in reducing the compile time as it reduces copying.

Example :

C

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4


//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

6. Common Sub Expression Elimination:

- In the above example, a*b and x*b is a common sub expression.

7. Dead Code Elimination:

- Copy propagation often leads to making assignment statements into dead code.
- A variable is said to be dead if it is never used after its last definition.
- In order to find the dead variables, a data flow analysis should be done.

Example:

C

```
c = a * b
x = a
```

```
till
d = a * b + 4

//After elimination :
c = a * b
till
d = a * b + 4
```

8. Unreachable Code Elimination:

- First, Control Flow Graph should be constructed.
- The block which does not have an incoming edge is an Unreachable code block.
- After constant propagation and constant folding, the unreachable branches can be eliminated.

C++

```
#include <iostream>
using namespace std;

int main() {
  int num;
  num=10;
    cout << "GFG!";
    return 0;
  cout << num; //unreachable code
}
//after elimination of unreachable code
int main() {
  int num;
  num=10;
    cout << "GFG!";
    return 0;
 }
```

9. Function Inlining:

- Here, a function call is replaced by the body of the function itself.
- This saves a lot of time in copying all the parameters, storing the return address, etc.

10. Function Cloning:

- Here, specialized codes for a function are created for different calling parameters.
- **Example:** Function Overloading

11. Induction Variable and Strength Reduction:

- An induction variable is used in the loop for the following kind of assignment i = i + constant. It is a kind of Loop Optimization Technique.
- Strength reduction means replacing the high strength operator with a low strength.

Examples:

C

```
Example 1 :
```

Multiplication with powers of 2 can be replaced by shift left operator which is less expensive than multiplication

```
a=a*16
// Can be modified as :
a = a<<4
```

Example 2 :

```
i = 1;
while (i<10)
{
   y = i * 4;
}
```

```
//After Reduction
i = 1
t = 4
{
  while( t<40)
  y = t;
  t = t + 4;
}
```

Loop Optimization:

1. Code Motion or Frequency Reduction:

- The evaluation frequency of expression is reduced.
- The loop invariant statements are brought out of the loop.

Example:

C

```
a = 200;
 while(a>0)
 {
    b = x + y;
    if (a % b == 0)
    printf("%d", a);
   }
```

```
//This code can be further optimized as
```

```
a = 200;
b = x + y;
while(a>0)
 {
    if (a % b == 0}
    printf(“%d”, a);
  }
```

2. Loop Jamming:

- Two or more loops are combined in a single loop. It helps in reducing the compile time.

Example:

C

```
// Before loop jamming
for(int k=0;k<10;k++)
{
 x = k*2;
}

for(int k=0;k<10;k++)
{
 y = k+3;
}

//After loop jamming
for(int k=0;k<10;k++)
{
 x = k*2;
 y = k+3;
}
```

3. Loop Unrolling:

- It helps in optimizing the execution time of the program by reducing the iterations.
- It increases the program’s speed by eliminating the loop control and test instructions.

Example:

C

```
//Before Loop Unrolling

for(int i=0;i<2;i++)
{
  printf("Hello");
}

//After Loop Unrolling
```

```
printf("Hello");
printf("Hello");
```

Basic Blocks and Flow Graphs:

The representation is constructed as follows:
1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that
(a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
(b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

Algorith m 8.5 : Partitioning three-address instructions into basic blocks.

METHOD: First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Example 8.6: The intermediate code in Fig. 8.7 turns a 10 x 10 matrix a into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix a is stored in row-major form.

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j) = 0.0;
for i from 1 to 10 do
    a[i, ij = 1.0;
```

Figure 8.8: Source code for Fig. 8.7

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto
10)  i = i + 1
11)  if i <= 10 goto
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto
```

Figure 8.7: Intermediate code to set a 10 x 10 matrix to an identity matrix

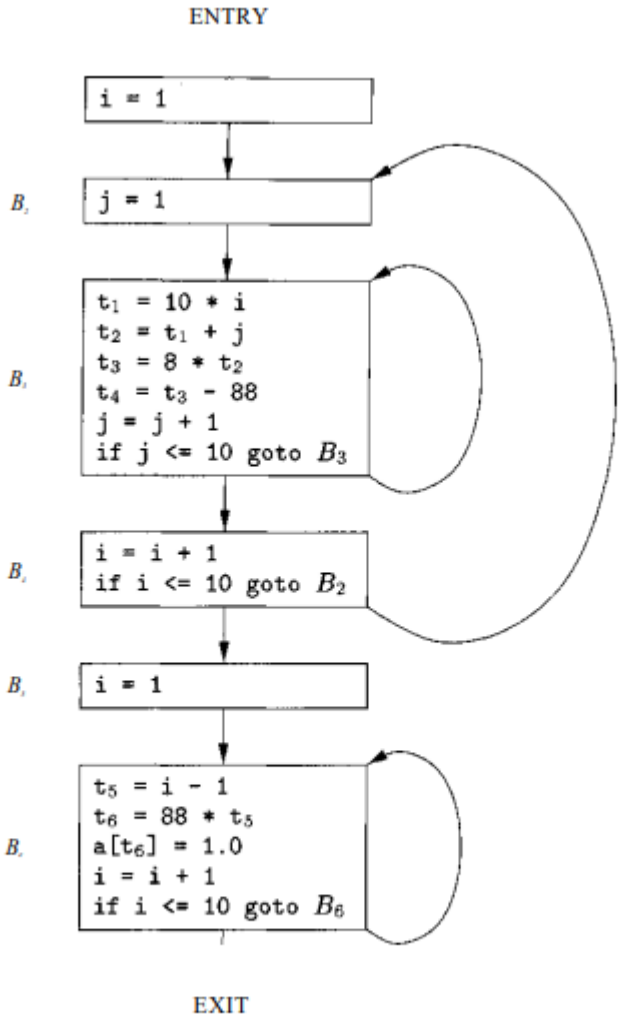


Figure 8.9: Flow graph from Fig. 8.7

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
  {
  int y, z;
  y = 10;
  z = x + y;
  return z;
  }
```

```
int add_ten(int x)
  {
  int y;
  y = 10;
  y = x + y;
  return y;
  }
```

```
int add_ten(int x)
  {
  int y = 10;
  return x + y;
  }
```

```
int add_ten(int x)
  {
  return x + 10;
  }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
  return x + 10;
  printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :  GOTO L2
L2 :  INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :  INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Flow Graph in Code Generation

A basic block is a simple combination of statements. Except for entry and exit, the basic blocks do not have any branches like in and out. It means that the flow of control enters at the beginning and it always leaves at the end without any halt. The execution of a set of instructions of a basic block always takes place in the form of a sequence.

The first step is to divide a group of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, the control will flow from one instruction to the next in sequential order.

The algorithm for the construction of the basic block is described below step by step:

Algorithm: The algorithm used here is partitioning the three-address code into basic blocks.

Input: A sequence of three-address codes will be the input for the basic blocks.
Output: A list of basic blocks with each three address statements, in exactly one block, is considered as the output.
Method: We'll start by identifying the intermediate code's leaders. The following are some guidelines for identifying leaders:

1. The first instruction in the intermediate code is generally considered as a leader.
2. The instructions that target a conditional or unconditional jump statement can be considered as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement can be considered as a leader.

Each leader's basic block will contain all of the instructions from the leader until the instruction right before the following leader's start.

Example of basic block:
Three Address Code for the expression a = b + c – d is:

T1 = b + c

T2 = T1 - d

a = T2

This represents a basic block in which all the statements execute in a sequence one after the other.

Basic Block Construction:

Let us understand the construction of basic blocks with an example:

Example:
0 seconds of 15 secondsVolume 0%

1. PROD = 0

2. I = 1

3. T2 = addr(A) – 4

4. T4 = addr(B) – 4

5. T1 = 4 x I

6. T3 = T2[T1]

7. T5 = T4[T1]

8. T6 = T3 x T5

9. PROD = PROD + T6

10. I = I + 1

11. IF I <=20 GOTO (5)

Using the algorithm given above, we can identify the number of basic blocks in the above three-address code easily-

There are two Basic Blocks in the above three-address code:
- **B1 –** Statement 1 to 4
- **B2 –** Statement 5 to 11

Transformations on Basic blocks:

Transformations on basic blocks can be applied to a basic block. While transformation, we don't need to change the set of expressions computed by the block.

There are two types of basic block transformations. These are as follows:

Optimization of Basic Blocks:

Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

There are two type of basic block optimization. These are as follows:

1. Structure-Preserving Transformations
2. Algebraic Transformations

1. Structure preserving transformations:

The primary Structure-Preserving Transformation on basic blocks is as follows:

- Common sub-expression elimination
- Dead code elimination

- Renaming of temporary variables
- Interchange of two independent adjacent statements

(a) Common sub-expression elimination:

In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.

1. a : = b + c
2. b : = a - d
3. c : = b + c
4. d : = a - d

In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:

1. a : = b + c
2. b : = a - d
3. c : = b + c
4. d : = b

(b) Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
- Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

(c) Renaming temporary variables

A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

(d) Interchange of statement

Suppose a block has the following two adjacent statements:

1. t1 : = b + c
2. t2 : = x + y

These two statements can be interchanged without affecting the value of block when value of t1 does not affect the value of t2.

2. Algebraic transformations:

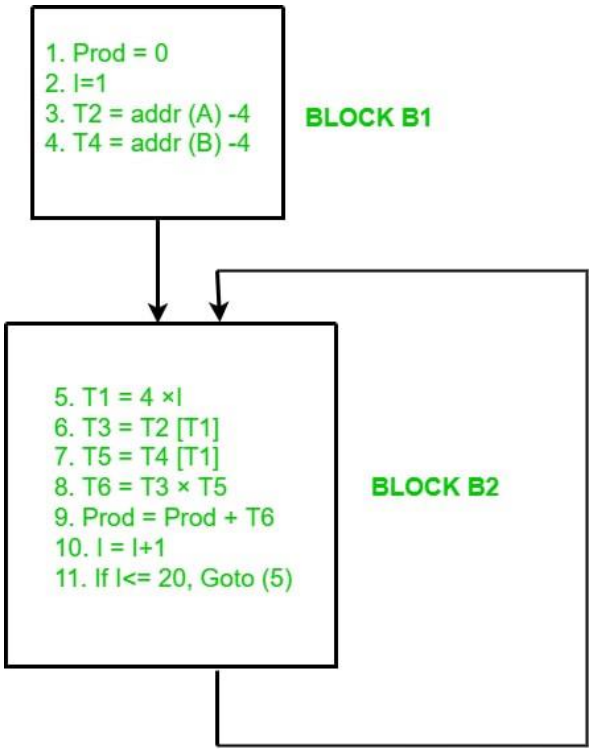
- In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expression.
- Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression 5*2.7 would be replaced by13.5.
- Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc.
- Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments**2.**

This can be eliminated from a basic block without changing the set of expressions.

Flow Graph:

A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information. A control flow graph is used to depict how the program control is being parsed among the blocks. A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks. When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow from one block X to another block Y.

Let’s make the flow graph of the example that we used for basic block formation:



Flow Graph for above Example

Firstly, we compute the basic blocks (which is already done above). Secondly, we assign the flow control information.

Register Allocation Algorithms in Compiler Design

Register allocation is an important method in the final phase of the compiler .
Registers are faster to access than cache memory .
Registers are available in small size up to few hundred Kb .Thus it is necessary to use ***minimum number of registers for variable allocation*** .

There are three popular Register allocation algorithms .
1. Naive Register Allocation
2. Linear Scan Algorithm
3. Chaitin's Algorithm

These are explained as following below.

1. Naïve Register Allocation :

Naive (no) register allocation is based on the assumption that variables are stored in Main Memory .
We can't directly perform operations on variables stored in Main Memory .
Variables are moved to registers which allows various operations to be carried out using ALU .
ALU contains a temporary register where variables are moved before performing arithmetic and logic operations .

Once operations are complete we need to store the result back to the main memory in this method .

Transferring of variables to and fro from Main Memory reduces the overall speed of execution .

a = b + c
d = a
c = a + d
Variables stored in Main Memory :

a	b	c	d
2 fp	4 fp	6 fp	8 fp

Machine Level Instructions :

```
LOAD  R1, _4fp
LOAD  R2, _6fp
ADD   R1, R2
STORE R1, _2fp
LOAD  R1, _2fp
STORE R1, _8fp
LOAD  R1, _2fp
LOAD  R2, _8fp
ADD   R1, R2
STORE R1, _6fp
```

Advantages :

Easy to understand operations and the flow of variables from Main memory to Registers and vice versa .

Only 2 registers are enough to perform any operations .

Design complexity is less .

Disadvantages :

Time complexity increases as variables is moved to registers from main memory .

Too many LOAD and STORE instructions .

To access a variable second time we need to STORE it to the Main Memory to record any changes made and LOAD it again .

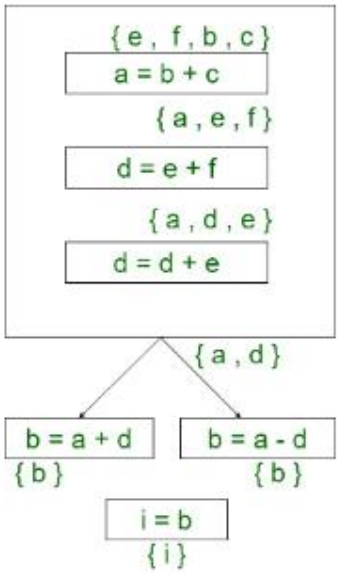
This method is not suitable for modern compilers .

2. Linear Scan Algorithm :

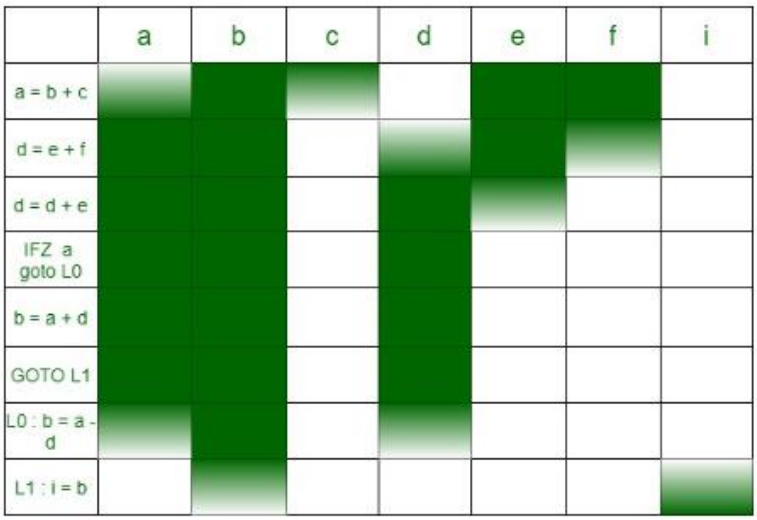
- Linear Scan Algorithm is a **global register allocation** mechanism .
- It is a bottom up approach .
- ***If n variables are live at any point of time then we require 'n' registers*** .
- In this algorithm the variables are scanned linearly to determine the live ranges of the variable based on which the registers are allocated .
- The main idea behind this algorithm is that to allocate minimum number of registers such that these registers can be used again and this totally depends upon the live range of the variables .
- For this algorithm we need to implement live variable analysis of Code Optimization

```
a = b + c
d = e + f
d = d + e
IFZ a goto L0
b = a + d
goto L1
L0 : b = a - d
L1 : i = b
```

Control Flow Graph :



- At any point of time the maximum number of live variables is 4 in this example . Thus we require 4 registers at maximum for register allocation .



If we draw horizontal line at any point on the above diagram we can see that we require exactly 4 registers to perform the operations in the program .

3.Graph Coloring (Chaitin's Algorithm) :

- Register allocation is interpreted as a graph coloring problem .
- Nodes represent live range of the variable.
- Edges represent the connection between two live ranges .
- Assigning color to the nodes such that no two adjacent nodes have same color .
- Number of colors represents the minimum number of registers required .

A k-coloring of the graph is mapped to k registers .

Steps :

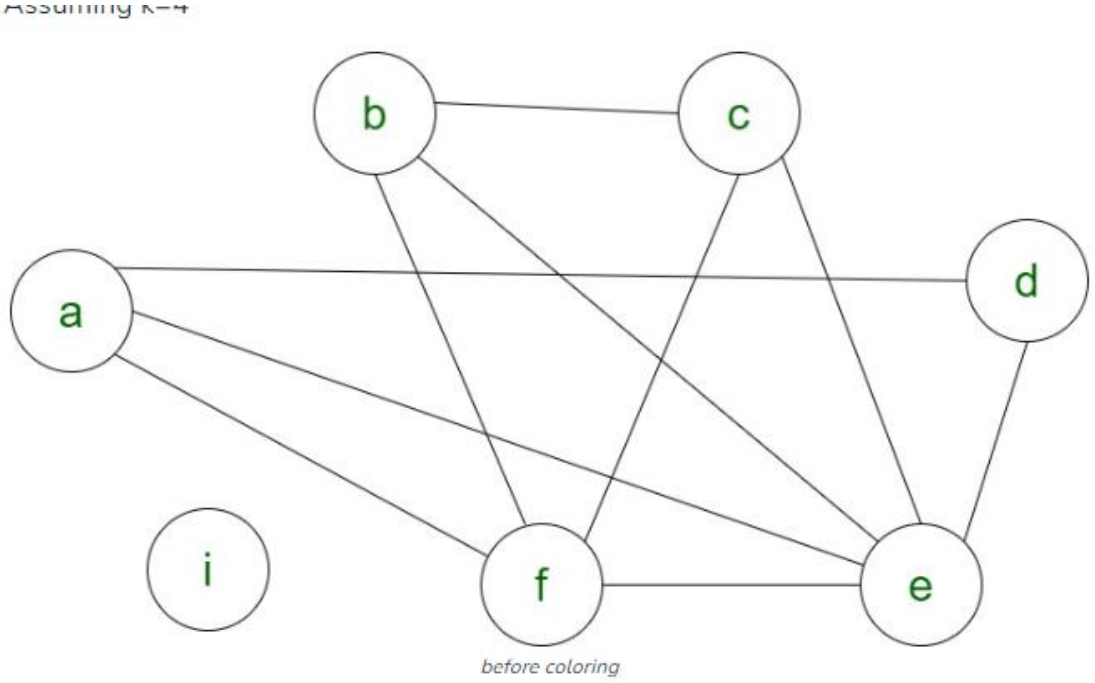
1. Choose an arbitrary node of degree less than k .

2. Push that node onto the stack and remove all of it's outgoing edges .
3. Check if the remaining edges have degree less than k, if YES goto 5 else goto #
4. If the degree of any remaining vertex is less than k then push it onto to the stack .
5. If there is no more edge available to push and if all edges are present in the stack POP each node and color them such that no two adjacent nodes have same color.
6. Number of colors assigned to nodes is the minimum number of registers needed .

spill some nodes based on their live ranges and then try again with same k value . If problem persists it means that the assumed k value can't be the minimum number of registers .Try increasing the k value by 1 and try the whole procedure again .

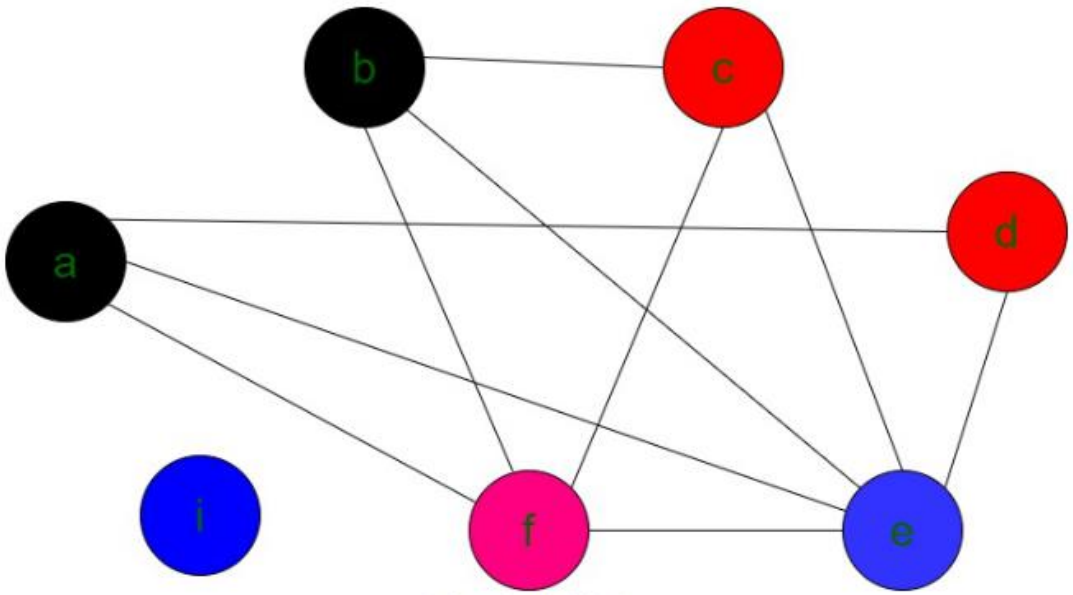
For the same instructions mentioned above the graph coloring will be as follows :

Assuming k=4



before coloring

After performing the graph coloring, final graph is obtained as follows



final graph with k(4) colors

Note : Any color(register) can be assigned to 'i' as it has no edge to any other nodes .