DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

UNIT-I

INTRODUCTION TO LANGUAGE PROCESSING:

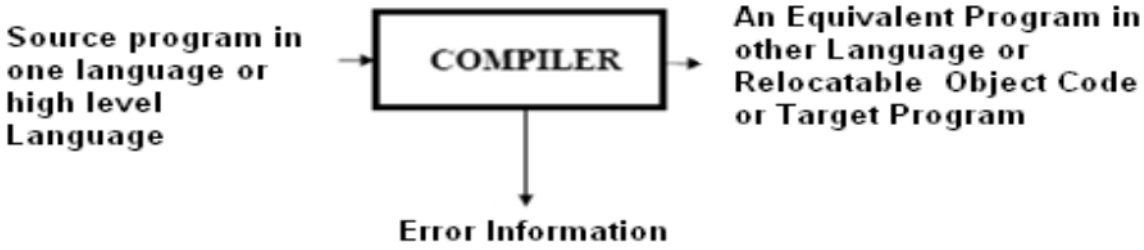
As Computers became inevitable and indigenous part of human life, and several languages with different and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software's have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGE TRANSLATORS :

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target]language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level languageprogram.

∑ Two commonly Used Translators are Compiler and Interpreter

1. **Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



∑ If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

Figure1.1: Running the target Program

2. **Interpreter:** An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by theuser.

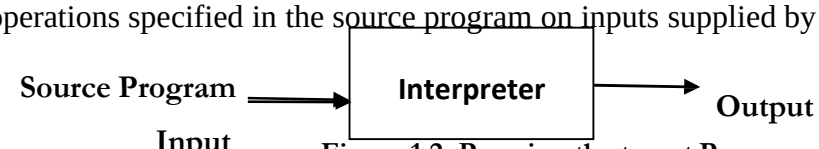


Figure 1.2: Running the target Program

LANGUAGE PROCESSING SYSTEM:

Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

Preprocessor: A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

- ∑ Collects all the modules, files in case if the source program is divided into different modules stored at different files.
- ∑ Expands short hands / macros into source languagestatements.

Compiler: Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

- ∑ Reports to its user the presence of errors in the source program.
- ∑ Facilitates the user in rectifying the errors, and execute the code.

Assembler: Is a program that takes as input an assembly language program and converts it into its equivalent machine language code.

Loader / Linker: This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

- ∑ **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.
- ∑ **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.

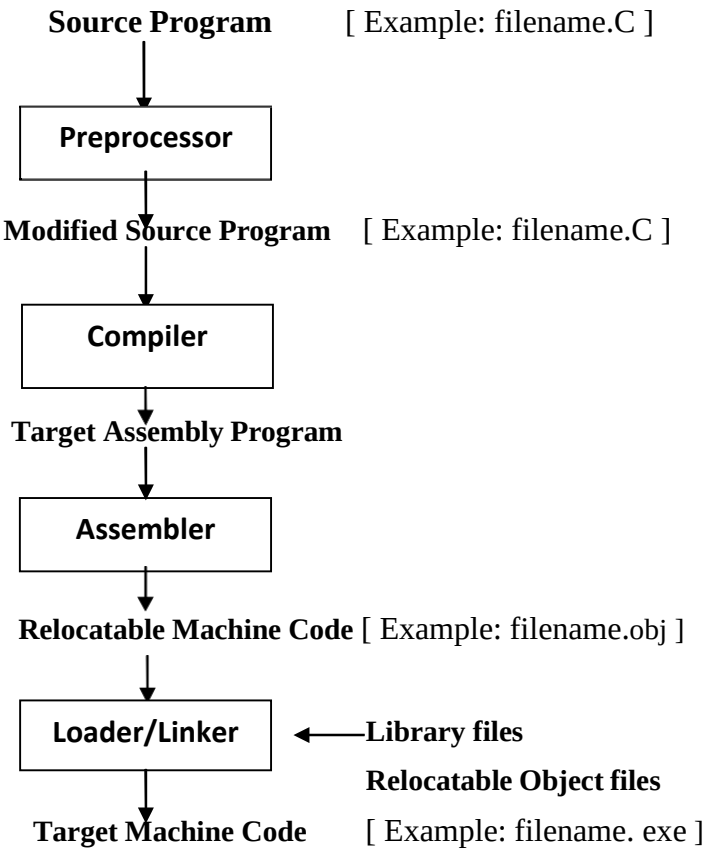


Figure1.3 : Context of a Compiler in Language Processing System

TYPES OF COMPILERS:

Based on the specific input it takes and the output it produces, the Compilers can be classified into the following types;

Traditional Compilers(C, C++, Pascal): These Compilers convert a source program in a HLL into its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code for another machine.

Incremental Compilers: These compilers separate the source into user defined–steps; Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into object code of another platform**.**

PHASES OF A COMPILER:

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions. Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)

Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. There fore a typical Compiler is having the following Phases:

1. Lexical Analyzer (Scanner), 2. Syntax Analyzer (Parser), 3.Semantic Analyzer, 4.Intermediate Code Generator(ICG), 5.Code Optimizer(CO) , and 6.Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some

cases. The description is given in next section.

The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

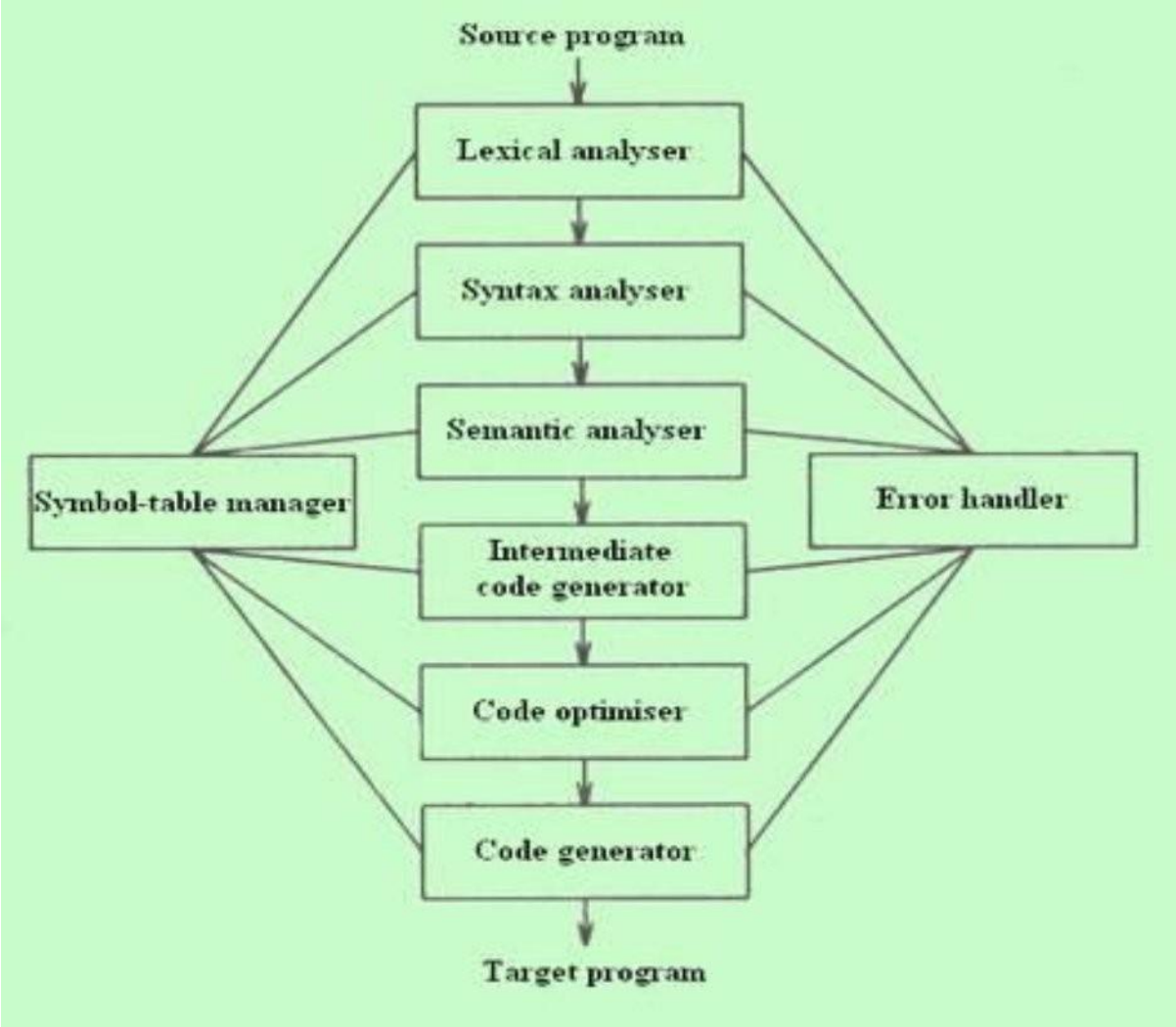


Figure1.4 : Phases of a Compiler

PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THE FRONT-END & BACK-END OF A COMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**, and **The Back-end**. This division is due to their dependence on either the Source Language or the Target machine. This model is called an Analysis & Synthesis model of a compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and those portions don‘t dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

- ∑ Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as anidentifier , a Keyword , a punctuation mark, a multi character operator like := .
- ∑ The character sequence forming a token is called a **lexeme** of the token.
- ∑ The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn‘t exist.
- ∑ Also removes the Comments, and unnecessary spaces.

The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- ∑ Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves are tokens.
- ∑ The interior node of this tree represents the stream of tokens that logically belongs together.
- ∑ It means it checks the syntax of program elements.

SEMANTIC ANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it

may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

∑ The Syntactically and Semantically correct structures are produced here in the form of a Syntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

∑ It should be easy to produce,and Easy to translate into the target program. Example intermediate code forms are:

∑ Three address codes,

∑ Polish notations, etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

∑ Attempts to improve the IC so as to have a faster machine code. Typical functionsinclude –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.

∑ Sometimes the data structures used in representing the intermediate forms may also be changed.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

∑ Memory locations are selected for each variable used, and assignment of variables to registers is done.

∑ Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

The input source program is **Position=initial+rate*60**

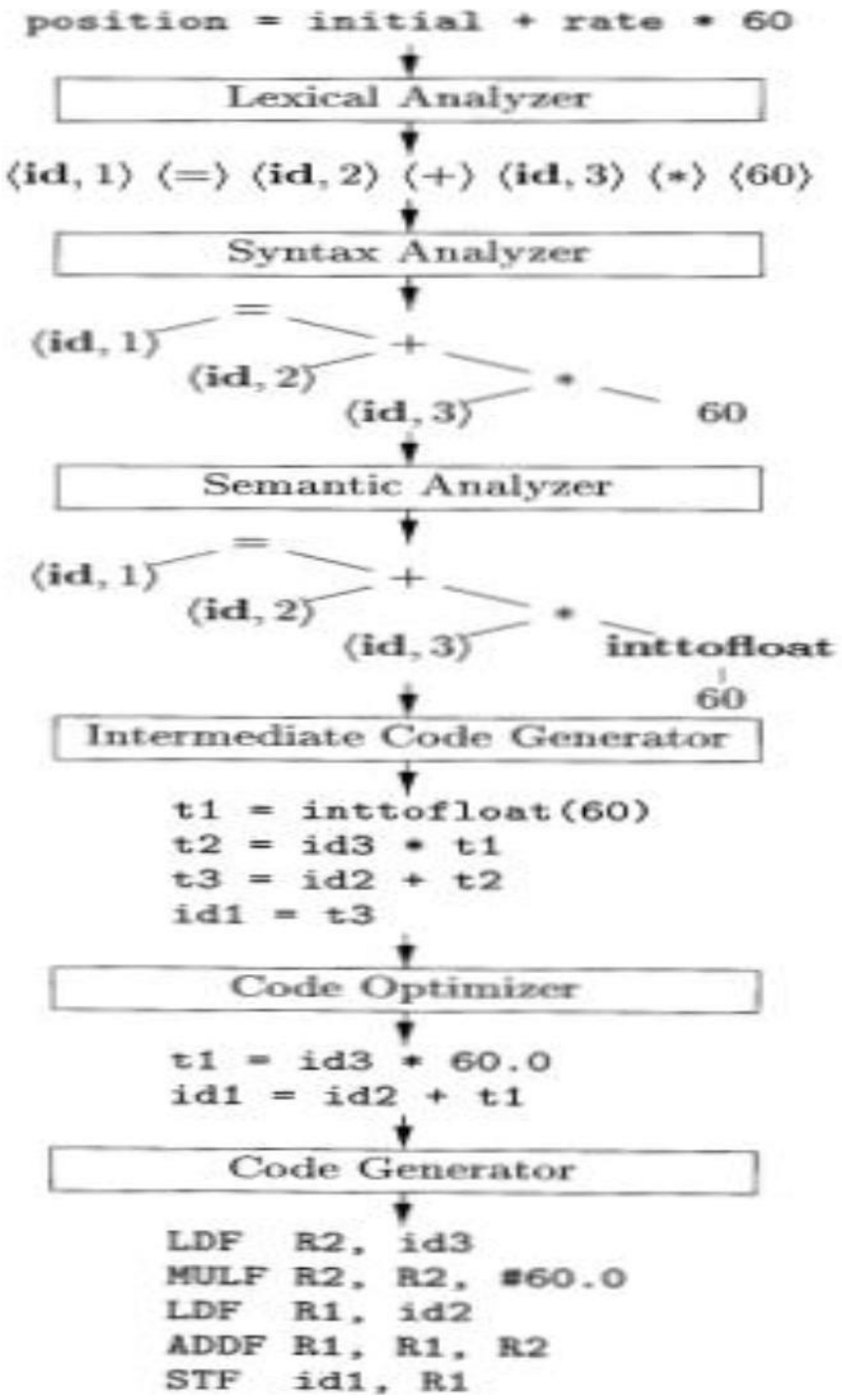


Figure1.5: Translation of an assignment Statement

LEXICAL ANALYSIS:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.

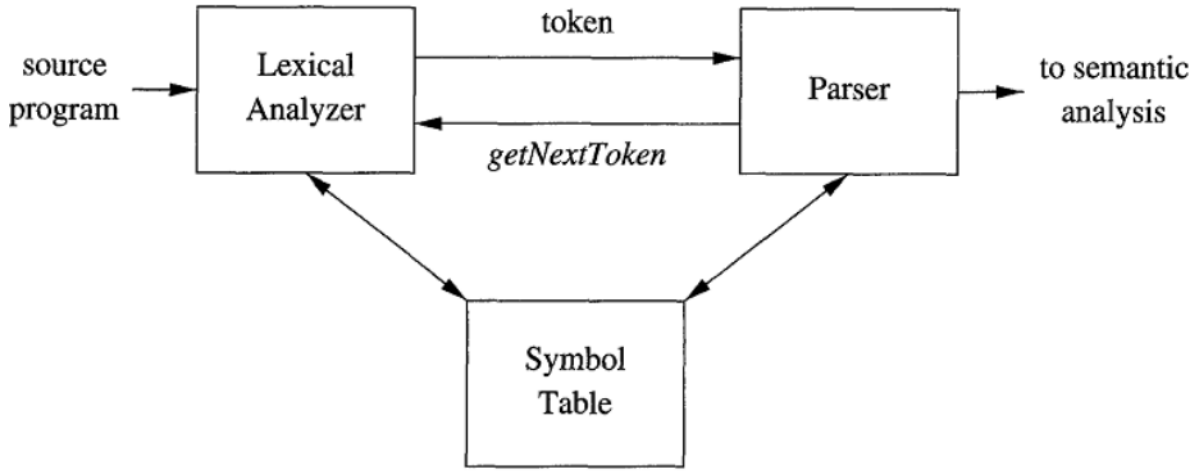


Figure 1.6 : Lexical Analyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

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 [or match]. 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.

Example: In the following C language statement ,

printf ("Total = %d\n‖, score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖** is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 1.7: Examples of Tokens

LEXICAL ANALYSIS Vs PARSING:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

∑ **1**. **Simplicity of design is the most important consideration.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic unitswould be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

∑ **2. Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

∑ **3. Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

INPUT BUFFERING:

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

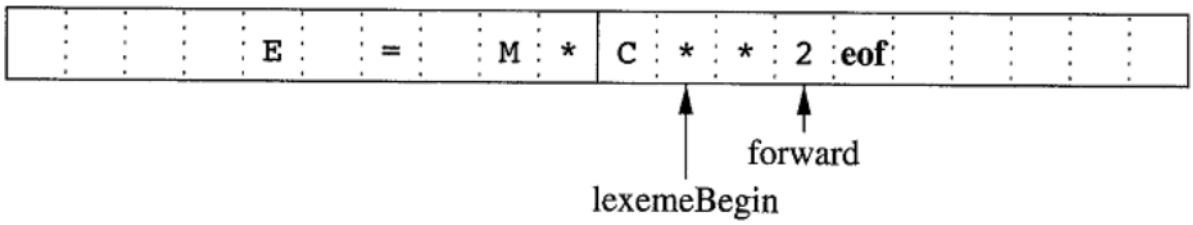


Figure1.8 : Using a Pair of Input Buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

∑ Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greaterthan N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and oneto determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

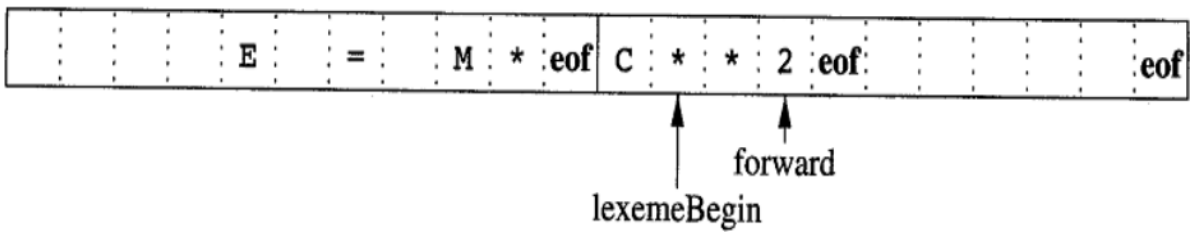


Figure1.8 : Sentential at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of

a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++  )
{
        case eof: if (forward is at end of first buffer )
                {
                  reload second buffer;
                  forward = beginning of second buffer;
                 }
                else if (forward is at end of second buffer  )
                {
                  reload first buffer;
                  forward = beginning of first buffer;
                }
                  else    /* eof within a buffer marks the end of input */
                    terminate lexical analysis;
            break;
}
```

Figure 1.9: use of switch-case for the sentential

SPECIFICATION OF TOKENS:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

LEX the Lexical Analyzer generator

Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex .yy .c, it is a c program given for C Compiler, gives the Object code. Here we need to know how to write the Lex language. The structure of the Lex program is given below.

Structure of LEX Program : A Lex program has the following form:

Declarations
%%
Translation rules
%%
Auxiliary functions definitions

The declarations section : includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. It appears between %{. . .%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the form

Pattern {Action}

The auxiliary function definitions section includes the definitions of functions used to install identifiers and numbers in the Symbol tale.

LEX Program Example:

```
%{
/* definitions of manifest constants LT,LE,EQ,NE,GT,GE, IF,THEN, ELSE,ID, NUMBER,
RELOP */
%}
/* regular definitions */
delim          [ \t\n]
ws      {      delim}+
letter          [A-Za-z]
digit           [o-91
id              {letter} ({letter} | {digit})  *
number          {digit}+ (\ . {digit}+)? (E  [+-I]?{digit}+)?
%%
{ws}             {/* no action and no return */}
if               {return(1F) ; }
```

```
then        {return(THEN) ; }
else        {return(ELSE) ; }
(id)        {yylval = (int) installID(); return(1D);}
(number)    {yylval = (int) installNum() ; return(NUMBER) ; }
‖ <  ‖      {yylval = LT; return(REL0P) ; )}
— <=‖       {yylval = LE; return(REL0P) ; }
—=‖         {yylval =  EQ  ; return(REL0P) ;  }
–<>‖        {yylval = NE;  return(REL0P);}
—<‖         {yylval =  GT;  return(REL0P);)}
–<=‖        {yylval  =  GE;  return(REL0P);}
%%
int installID0() {/* function to install the lexeme, whose first character is pointed to by yytext,
                    and whose length is yyleng, into the symbol table and return a pointer
                    thereto */
int installNum() {/* similar to installID, but puts numerical constants into a separate table */}
```

Figure 1.10 : Lex Program for tokens common tokens

SYNTAX ANALYSIS (PARSER)

THE ROLE OF THE PARSER:

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the below Figure, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

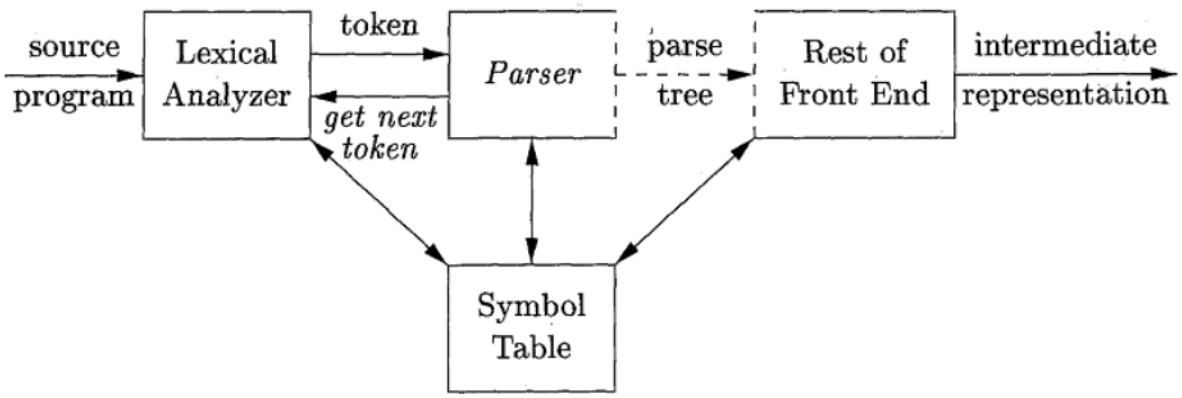


Figure2.1: Parser in the Compiler

During the process of parsing it may encounter some error and present the error information back to the user

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).

Based on the way/order the Parse Tree is constructed, **Parsing** is basically **classified** in to following two types:

1. **Top Down Parsing :** Parse tree construction start at the root node and moves to the children nodes (i.e., top down order).
2. **Bottom up Parsing:** Parse tree construction begins from the leaf nodes and proceeds towards the root node (called the bottom up order).

IMPORTANT (OR) EXPECTED QUESTIONS

1. What is a Compiler? Explain the working of a Compiler with your own example?
2. What is the Lexical analyzer? Discuss the Functions of Lexical Analyzer.
3. Write short notes on tokens, pattern and lexemes?
4. Write short notes on Input buffering scheme? How do you change the basic input buffering algorithm to achieve better performance?
5. What do you mean by a Lexical analyzer generator? Explain LEX tool.

ASSIGNMENT QUESTIONS:

1. Write the differences between compilers and interpreters?
2. Write short notes on token reorganization?
3. Write the Applications of the Finite Automata?
4. Explain How Finite automata are useful in the lexical analysis?
5. Explain DFA and NFA with an Example?

UNIT-II

TOP DOWN PARSING:

∑ Top-down parsing can be viewed as the problem of constructing a parse tree for the given input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first left to right).

∑ Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

It is classified in to two different variants namely; one which uses Back Tracking and the other is Non Back Tracking in nature.

Non Back Tracking Parsing: There are two variants of this parser as given below.

1. **Table Driven Predictive Parsing :**
   i. LL (1) Parsing
2. **Recursive Descent parsing**

Back Tracking

1. Brute Force method

NON BACK TRACKING:

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. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

The table-driven parser in the figure has

∑ 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 start symbol of the grammar on top of $.

∑ A parsing table containing the production rules to be applied. This is a two dimensional array M [Non terminal, Terminal].

∑ A parsing Algorithm that takes input String and determines if it is conformant to Grammar and it uses the parsing table and stack to take such decision.

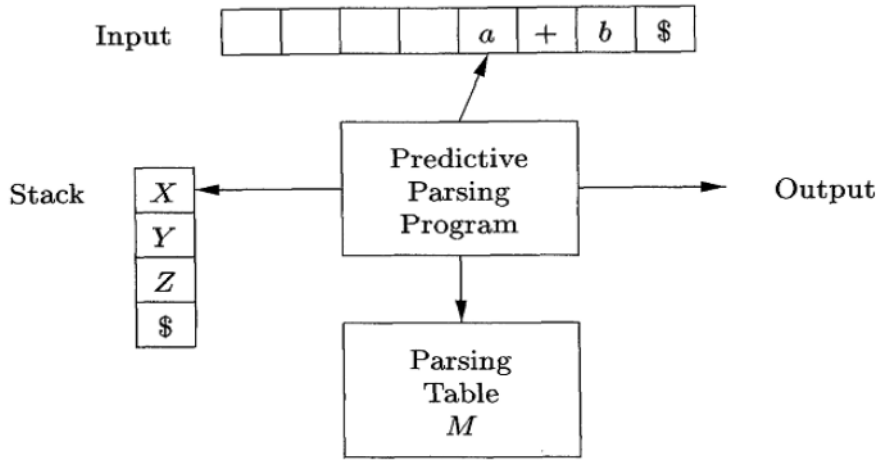


Figure 2.2: Model for table driven parsing

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

Context Free Grammar (CFG): CFG used to describe or denote the syntax of the programming language constructs. The CFG is denoted as G, and defined using a four tuple notation.

Let G be CFG, then G is written as, G= (V, T, P, S)

Where

∑ V is a finite set of Non terminal; Non terminals are syntactic variables that denote sets of strings. The sets of strings denoted by non terminals help define the language generated by the grammar. Non terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

∑ T is a Finite set of Terminal; Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for '"terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer.

∑ S is the Starting Symbol of the grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. P is finite set of Productions; the productions of a grammar specify the manner in which the

terminals and non terminals can be combined to form strings, each production is in α->β form, where α is a single non terminal, β is (VUT)*.Each production consists of:

(a) A non terminal called the head or left side of the production; this production defines some of the strings denoted by the head.

(b) The symbol ->. Some times: = has been used in place of the arrow.

(c) A body or right side consisting of zero or more terminals and non- terminals. The components of the body describe one way in which strings of the non terminal at the head can be constructed.

∑ Conventionally, the productions for the start symbol are listed first.

Example: Context Free Grammar to accept Arithmetic expressions.

The terminals are **+, *, -, (,), id**.

The **Non terminal symbols** are **expression**, **term**, **factor** and expression is the starting symbol.

expression → expression + term
expression → expression – term
expression → term
term → term * factor
term → term / factor
term → factor
factor → (expression)
factor → id

Figure 2.3 : Grammar for Simple Arithmetic Expressions

Notational Conventions Used In Writing CFGs:

To avoid always having to state that –these are the terminals," "these are the non terminals," and so on, the following notational conventions for grammars will be used throughout our discussions.

1. These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, e.

(b) Operator symbols such as **+**, ***,** and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1. . . 9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

2. These symbols are non terminals:

(a) Uppercase letters early in the alphabet, such as A, B, *C.*

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as *expr* or *stmt.*

(d) When discussing programming constructs, uppercase letters may be used to represent Non terminals for the constructs. For example, non terminal for expressions, terms, and factors are often represented by E, T, and F, respectively.

Using these conventions the grammar for the arithmetic expressions can be written as

E → E + T | E – T | T
T → T * F | T / F | F

F → (E) | id

DERIVATIONS:

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a Non terminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree as well as the bottom construction of theparse tree.

∑ Derivations are classified in to **Let most Derivation** and **Right Most Derivations.**

Left Most Derivation (LMD):

It is the process of constructing the parse tree or accepting the given input string, in which at every time we need to rewrite the production rule it is done with left most non terminal only.
Ex: - If the Grammar is E-> **E+E | E*E | -E| (E) | id** and the input string is **id + id* id**

The production **E -> - E** signifies that if E denotes an expression, then **–** E must also denote an expression. The replacement of a single E by - E will be described by writing

E => -E which is read as **"E derives _E"**

For a general definition of derivation, consider a non terminal A in the middle of a sequence of grammar symbols, as in αAβ, where α and β are arbitrary strings of grammar symbol. Suppose A ->γ is a production. Then, we write αAβ => αγβ. The symbol => means "derives in one step". Often, we wish to say, "Derives in zero or more steps." For this purpose,we can use the symbol $\overset{*}{\Rightarrow}$, If we wish to say, "Derives in $\overset{+}{\Rightarrow}$ one or more steps." We cn use the symbol $\overset{+}{\Rightarrow}$. If S $\overset{*}{\Rightarrow}$ a, where S is the start symbol of a grammar G, we say that α is a sentential form of G.

The Leftmost Derivation for the given input string **id + id* id** is

E => E +E

=> id + E

=> id + E * E

=> id + id * E

=> id + id * id

NOTE: Every time we need to start from the root production only, the under line using at Non terminal indicating that, it is the non terminal (left most one) we are choosing to rewrite the productions to accept the string.

Right Most Derivation (RMD):

It is the process of constructing the parse tree or accepting the given input string, every time we need to rewrite the production rule with Right most Non terminal only.

The Right most derivation for the given input string **id + id* id** is

E => E + E
=> E + E * E
=> E + E * id
=> E + id * id
=> id + id * id

NOTE: Every time we need to start from the root production only, the under line using at Non terminal indicating that, it is the non terminal (Right most one) we are choosing to rewrite the productions to accept the string.

What is a Parse Tree?

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals.

- ∑ Each interior node of a parse tree represents the application of a production.
- ∑ All the interior nodes are Non terminals and all the leaf nodes terminals.
- ∑ All the leaf nodes reading from the left to right will be the output of the parse tree.
- ∑ If a node n is labeled X and has children n1,n2,n3,…nk with labels X1,X2,…Xk respectively, then there must be a production A->X1X2…Xk in the grammar.

Example1:- Parse tree for the input string - **(id + id) using** the above Context free Grammar is

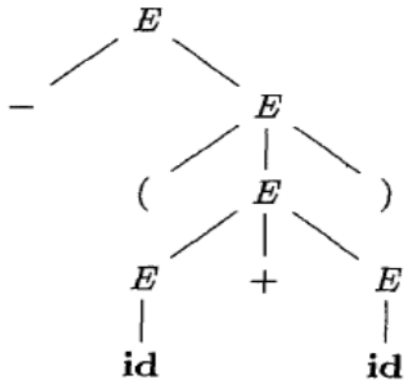


Figure 2.4 : Parse Tree for the input string - **(id + id)**

The Following figure shows step by step construction of parse tree using CFG for the parse tree for the input string - **(id + id).**

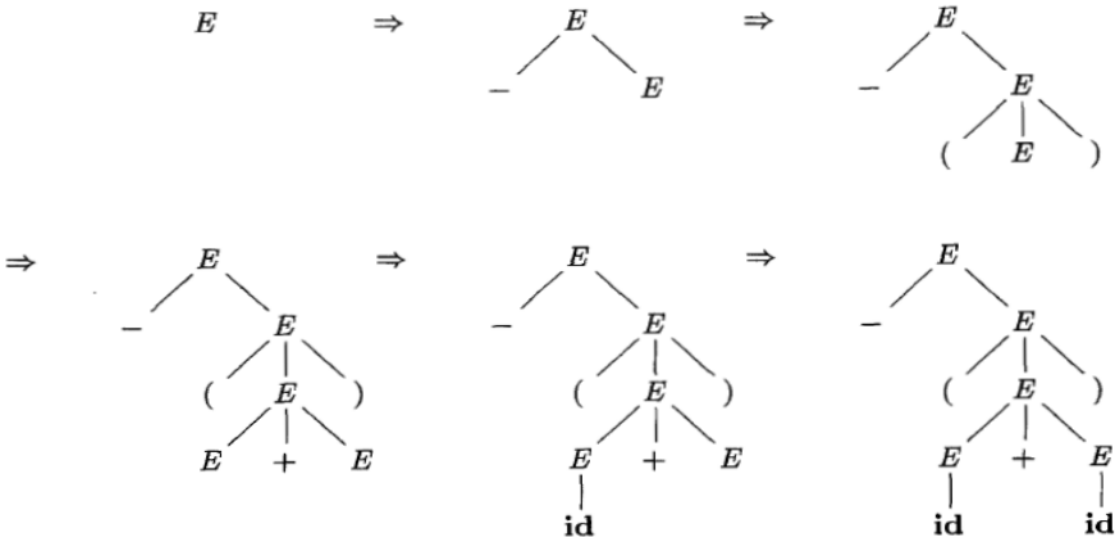


Figure 2.5 : Sequence outputs of the Parse Tree construction process for the input string **–(id+id)**

Example2:- Parse tree for the input string **id+id*id** using the above Context free Grammar is

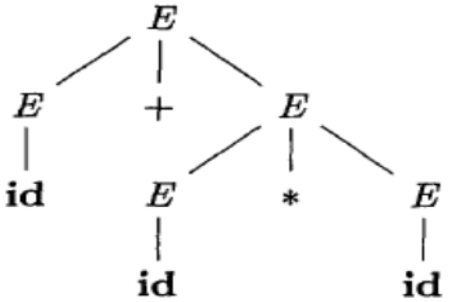


Figure 2.6: Parse tree for the input string id+ id*id

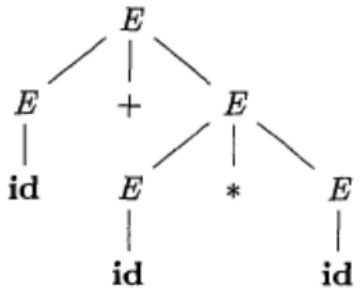
AMBIGUITY in CFGs:

Definition: A grammar that produces more than one parse tree for some sentence (input string) is said to be ambiguous.

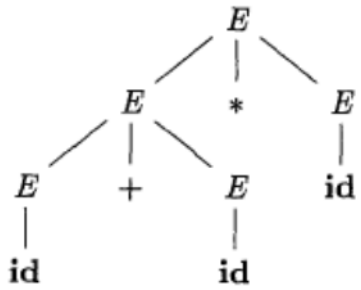
In other words, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Or If the right hand production of the grammar is having two non terminals which are exactly same as left hand side production Non terminal then it is said to an ambiguous grammar.
Example : If the Grammar is **E-> E+E | E*E | -E| (E) | id** and the Input String is **id + id* id**

Two parse trees for given input string are



(a)



(b)

Two Left most Derivations for given input String are :

E => E +E

=> id + E

=> id + E * E

=> id + id * E

=> id + id * id

(a)

E => E * E

=> E+E*E

=> id + E * E

=> id+ id* E

=> id + id * id

(b)

The above Grammar is giving two parse trees or two derivations for the given input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: Since Ambiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, **s**ometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

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

This can be written as (introduce one new non terminal in the place of second non terminal)

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \alpha\, A' \mid \varepsilon$$

Example : Let the grammar is **E⟶ E+E | E*E | -E| (E) | id .** It is shown that it is ambiguous that can be written as

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

In the above grammar the 1st and 2nd productions are having ambiguity. So, they can be written as

E-> E+E | E*E this production again can be written as
E-> E+E | β , where β is E*E
The above production is same as the general form. so, that can be written as
E->E+T|T
T->β

The value of β is E*E so, above grammar can be written as

1) **E->E+T|T**
2) **T-> E*E** **The first production is free from ambiguity** and substitute E->T in the 2nd production then it can be written as
   T-> T*T | -E| (E) | **id** this production again can be written as
   T->T*T | β where β is -E| (E) | id, introduce new non terminal in the Right hand side production then it becomes
   T->T*F | F
   F-> -E | (E) | id now the entire grammar turned in to it equivalent unambiguous,

The Unambiguous grammar equivalent to the given ambiguous one is

1) **E → E + T | T**
2) **T → T * F | F**
3) **F → -E | (E) | id**

LEFT RECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation A=>Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

The General form of Left Recursion is

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

The above left recursive production can be written as the non left recursive equivalent :

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid €$$

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

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

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

$$F \rightarrow -E \mid (E) \mid id$$

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 $E \rightarrow E + T$, and $T \rightarrow T * F$ is

$$E \rightarrow TE'$$
$$E' \rightarrow +T\,E' \mid €$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid €$$
$$F \rightarrow (E) \mid id$$

LEFT FACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.

For example, in the following grammar there are n A productions have the common prefix **α,** which should be removed or factored out without changing the language defined for A.

$$A \rightarrow \alpha A1 \mid \alpha A2 \mid \alpha A3 \mid \alpha A4 \mid \ldots \mid \alpha An$$

We can factor out the **α** from all n productions by adding a new A production $A \rightarrow \alpha A'$, and rewriting the A′ productions grammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1 \mid A2 \mid A3 \mid A4 \ldots \mid An$$

FIRST and FOLLOW:

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input (look a head) symbol.

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 **€** 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->€, add FIRST(X2) to FIRST(A), if X2-> € add FIRST(X3) to FIRST(A), … if Xi->€ , i.e., all X_i's for i=1..i are null, add € FIRST(A).
3. If A ->€ is a production, then add € 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 € is in FOLLOW(B).
3. If there is a production A->αB or a production A-> αBβ with FIRST(β) contains €, then FOLLOW (B) = FOLLOW (A).

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

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

Computing FIRST Values:

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}
FIRST (E′) = {+, €}
FIRST (T′) = {*, €}

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′	{ +, € }	{ $,) }
T	{ (, id}	{ +, $,) }
T′	{*, €}	{ +, $,) }
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 € is in FIRST (α) add production A->α to M [A, b], where b is all terminals in FOLLOW (A).
   ii. If € 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′ €		E′ €
T			T FT′		T FT′	
T′	T′ €	T′ *FT′		T′ €		T′ €
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` €	T` on top of the stack is replaced by €
$E`	+id*id$	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` €	T` on top of the stack is replaced by €
$E`	$	E` €	E` on top of the stack is replaced by €
$	$	Parsing is successful	Condition 1 satisfied

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

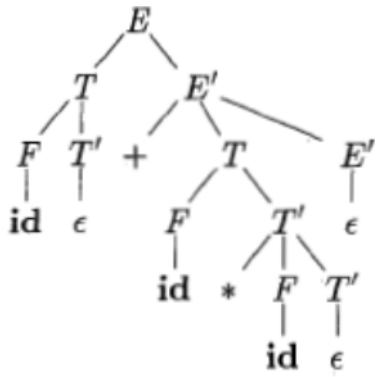


Figure 2.7: Parse tree for the input id + id* id

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. The following error recovery schemes are use in predictive parsing:

Panic mode Error Recovery :

It is based on the idea that when an error is detected, the parser will skips the remaining input until a synchronizing token is en countered in the input. Some examples are listed below:

1. For a Non Terminal A, place all symbols in FOLLOW (A) are adde into the synchronizing set of non terminal A. For Example, consider the assignment statement –c=;‖ Here, the expression on the right hand side is missing. So the Follow of this is considered. It is –;‖ and is taken as synchronizing token. On encountering it, parser emits an error message –Missing Expression‖.
2. For a Non Terminal A, place all symbols in FIRST (A) are adde into the synchronizing set of non terminal A. For Example, consider the assignment statement –22c= a + b;‖ Here, FIRST (expr) is 22. It is –;‖ and is taken as synchronizing token and then the reports the error as –extraneous token‖.

Phrase Level Recovery :

It can be implemented in the predictive parsing by filling up the blank entries in the predictive parsing table with pointers to error Handling routines. These routines can insert, modify or delete symbols in the input.

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 ( );
```

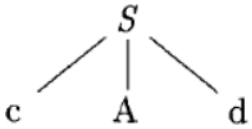
```
                T′( );
        return true;
        }
        else return error;
}
procedure F( )
{
        if input = ‗(‗
        {
                advance( );
                E ( );
                if input = ‗)‘
                advance( );
                return true;
        }
        else if input = ―id‖
        {

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

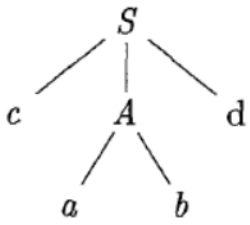
BACK TRACKING: This parsing method uses the technique called Brute Force method during the parse tree construction process. This allows the process to go back (back track) and redo the steps by undoing the work done so far in the point of processing.

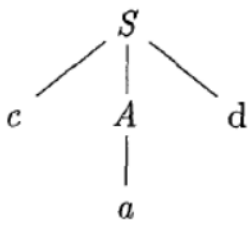
Brute force method: It is a Top down Parsing technique, occurs when there is more than one alternative in the productions to be tried while parsing the input string. It selects alternatives in the order they appear and when it realizes that something gone wrong it tries with next alternative.
For example, consider the grammar bellow.

S → cAd
A → ab | a

To generate the input string ―cad‖, initially the first parse tree given below is generated. As the string generated is not ―cad‖, input pointer is back tracked to position ―A‖, to examine the next alternate of ―A‖. Now a match to the input string occurs as shown in the 2nd parse trees given below.







(1) **(2)**

IMPORTANT AND EXPECTED QUESTIONS

1. Explain the components of working of a Predictive Parser with an example?
2. What do the FIRST and FOLLOW values represent? Give the algorithm for computing FIRST n FOLLOW of grammar symbols with an example?
3. Construct the LL (1) Parsing table for the following grammar?
   E → E+T|T

   T → T*F

   F → (E) | id
4. For the above grammar construct, and explain the Recursive Descent Parser?
5. What happens if multiple entries occurring in your LL (1) Parsing table? Justify your answer? How does the Parser

ASSIGNMENT QUESTIONS

1. Eliminate the Left recursion from the below grammar?
   A-> Aab | AcB|b
   B-> Ba | d
2. Explain the procedure to remove the ambiguity from the given grammar with your own example?
3. Write the grammar for the if-else statement in the C programming and check for the left factoring?
4. Will the Predictive parser accept the ambiguous Grammar justify your answer?
5. Is the grammar G = { S->L=R, S->R, R->L, L->*R | **id** } an LL(1) grammar?

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:

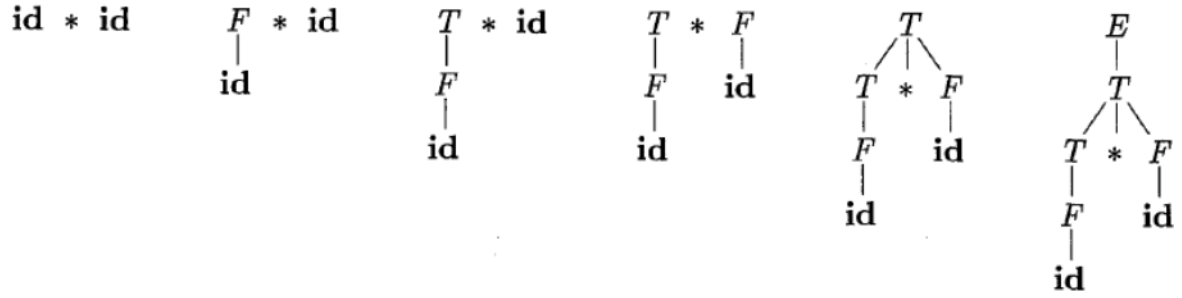


Figure 3.1 : A Bottom-up Parse tree for the input String "id*id"

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

Consider the following grammar:

S → aAcBe
A→ Ab|b
B→ d

Let the input string is –abbcde‖. The series of shift and reductions to the start symbol are as follows.

abbcde ⇨ aAbcde ⇨ aAcde ⇨ aAcBe ⇨ S

Note: in the above example there are two actions possible in the second Step, these are as follows :

1. Shift action going to 3rd Step
2. Reduce action, that is A->b

If the parser is taking the 1st action then it can successfully accepts the given input string, if it is going for second action then it can't accept given input string. This is called shift reduce conflict. Where, S-R parser is not able take proper decision, so it not recommended for parsing.

OPERATOR PRECEDENCE PARSING:

Operator precedence grammar is kinds of shift reduce parsing method that can be applied to a small class of operator grammars. And it can process ambiguous grammars also.

- ∑ An operator grammar has two important characteristics:
  1. There are no € productions.
  2. No production would have two adjacent non terminals.
- ∑ The operator grammar to accept expressions is give below:

  E ➡ E+E / E → E-E / E → E*E / E → E/E / E → E^E / E → -E / E → (E) / E → id

Two main Challenges in the operator precedence parsing are:

1. Identification of Correct handles in the reduction step, such that the given input should be reduced to starting symbol of the grammar.
2. Identification of which production to use for reducing in the reduction steps, such that we should correctly reduce the given input to the starting symbol of the grammar.

Operator precedence parser consists of:

1. An input buffer that contains string to be parsed followed by a$, a symbol used to indicate the ending of input.
2. A stack containing a sequence of grammar symbols with a $ at the bottom of the stack.
3. An operator precedence relation table O, containing the precedence ralations between the pair of terminal. There are three kinds of precedence relations will exist between the pair of terminal pair _a' and _b' as follows:
4. The relation a<•b implies that he terminal _a' has lower precedence than terminal _b'.
5. The relation a•>b implies that he terminal _a' has higher precedence than terminal _b'.
6. The relation a=•b implies that he terminal _a' has lower precedence than terminal _b'.

7. An operator precedence parsing program takes an input string and determines whether it conforms to the grammar specifications. It uses an operator precedence parse table and stack to arrive at the decision.

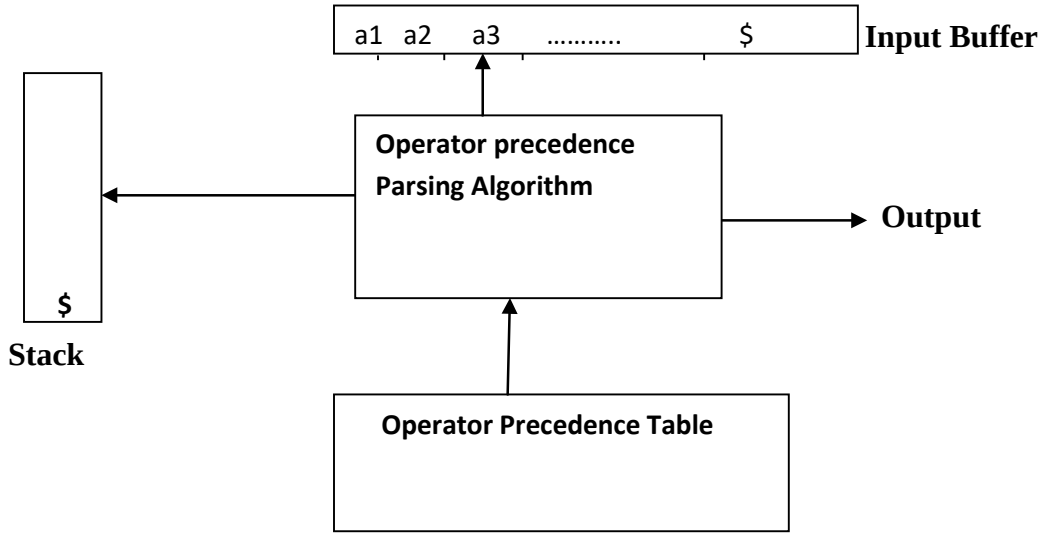


Figure3.2: Components of operator precedence parser

Example, If the grammar is

E → E+E
E → E-E
E → E*E
E → E/E
E → E^E
E → -E
E → (E)
E → id , Construct operator precedence table and accept input string **" id+id*id"**

The precedence relations between the operators are
(id) > (^) > (* /) > (+ -) > $, „^" operator is Right Associative and reaming all operators are Left Associative

	+	-	*	/	^	id	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
Id	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	=	**Err**
)	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	**Err**	**Err**

The intention of the precedence relations is to delimit the handle of the given input String with <• marking the left end of the Handle and •> marking the right end of the handle.

Parsing Action:

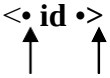
To locate the handle following steps are followed:

1. Add $ symbol at the both ends of the given input string.
2. Scan the input string from left to right until the right most •> is encountered.
3. Scan towards left over all the equal precedence's until the first <• precedence is encountered.
4. Every thing between <• and •> is a handle.
5. $ on S means parsing is success.

Example, Explain the parsing Actions of the OPParser for the input string is **"id*id"** and the grammar is:

E → E+E
E → E*E
E → id

1. $ <• id •> *<• id•> $



The first handle is _id' and match for the _id _in the grammar is E → id . So, id is replaced with the Non terminal E. the given input string can be written as

2. $ <• E •> *<• id•> $

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So , the string becomes

3. $ <• *<• id•> $

The next handle is _id' and match for the _id _in the grammar is E → id . So, id is replaced with the Non terminal E. the given input string can be written as

4. **$ <• *<• E•> $**

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So, the string becomes

5. **$ <• * •> $**

The next handle is _*' and match for the _ _in the grammar is E → E * E . So, id is replaced with the Non terminal E. the given input string can be written as

6. **$ E $**

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So, the string becomes

7. **$ $**

$ On $ means parsing successful.

Operator Parsing Algorithm:

The operator precedence Parser parsing program determines the action of the parser depending on

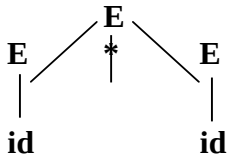
1. _a‘ is top most symbol on the Stack
2. _b‘ is the current input symbol

There are 3 conditions for _a‘ and _b‘ that are important for the parsing program

1. a=b=$, the parsing is successful
2. a <• b or a = b, the parser shifts the input symbol on to the stack and advances the input pointer to the next input symbol.
3. a •> b, parser performs the reduce action. The parser pops out elements one by one from the stack until we find the current top of the stack element has lower precedence than the most recently popped out terminal.

Example, the sequence of actions taken by the parser using the stack for the input string –id * id — and corresponding Parse Tree are as under.

STACK	INPUT	OPERATIONS
$	id * id $	$ <• id, shift _id‘ in to stack
$ id	*id $	id •> *, reduce _id‘ using E-> id
$E	*id $	$ <• *, shift _*‘ in to stack
$E*	id$	* <• id , shift _id‘ in to Stack
$E*id	$	id •> $, reduce _id‘ using E->id
$E*E	$	*•> $, reduce _*‘ using E->E*E
$E	$	$=$=$, so parsing is successful



Advantages and Disadvantages of Operator Precedence Parsing:

The following are the advantages of operator precedence parsing

1. It is simple and easy to implement parsing technique.
2. The operator precedence parser can be constructed by hand after understanding the grammar. It is simple to debug.

The following are the disadvantages of operator precedence parsing:

1. It is difficult to handle the operator like _-_which can be either unary or binary and hence different precedence‘s and associativities.
2. It can parse only a small class of grammar.

3. New addition or deletion of the rules requires the parser to be re written.
4. Too many error entries in the parsing tables.

LR Parsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of the given input string, R is Right Most derivation in reverse and K is no of input symbols as the Look ahead.

- ∑ It is the most general non back tracking shift reduce parsing method
- ∑ The class of grammars that can be parsed using the LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- ∑ An LR parser can detect a syntactic error as soon as it is possible to do so, on a left to right scan of the input.

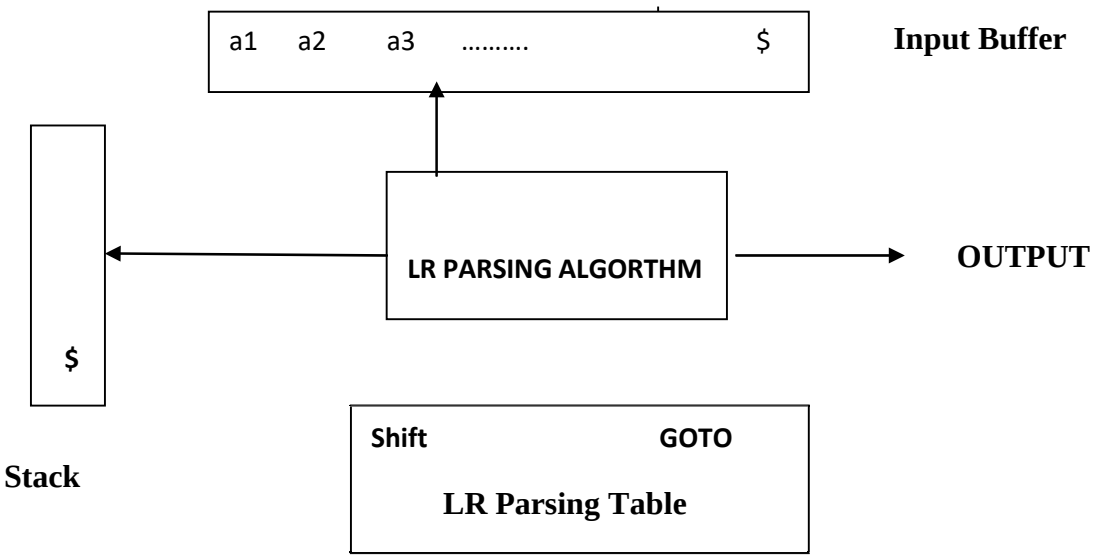


Figure 3.3: Components of LR Parsing

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.

Canonical collection of LR (0) items

LR (0) items

An LR (0) item of a Grammar is a production G with dot at some position on the right side of the production. An item indicates how much of the input has been scanned up to a given point in the process of parsing. For example, if the Production is X → YZ then, The LR (0) items are:

1. X → •AB, indicates that the parser expects a string derivable from AB.
2. X → A•B, indicates that the parser has scanned the string derivable from the A and expecting the string from Y.
3. X → AB•, indicates that he parser has scanned the string derivable from AB.

If the grammar is X → € the, the LR (0) item is

X → •, indicating that the production is reduced one.

Canonical collection of LR(0) Items:

This is the process of grouping the LR (0) items together based on the closure and Go to operations

Closure operation

If I is an initial State, then the Closure (I) is constructed as follows:

1. Initially, add Augment Production to the state and check for the • symbol in the Right hand side production, if the • is followed by a Non terminal then Add Productions which are Stating with that Non Terminal in the State I.

2. If a production X→α•Aβ is in I, then add Production which are starting with X in the State I. Rule 2 is applied until no more productions added to the State I(meaning that the • is followed by a Terminal symbol).

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

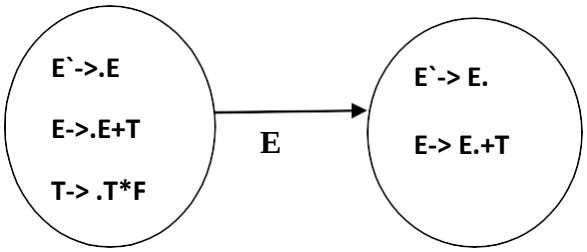
GO TO Operation

Go to (I_0, X), where I_0 is set of items and X is the grammar Symbol on which we are moving the „•" symbol. It is like finding the next state of the NFA for a give State I_0 and the input symbol is X. For example, if the production is **E →E+T**

Go to **(I_0, E)** is **E`→ •E, E→ E•+T**

Note: Once we complete the Go to operation, we need to compute closure operation for the output production

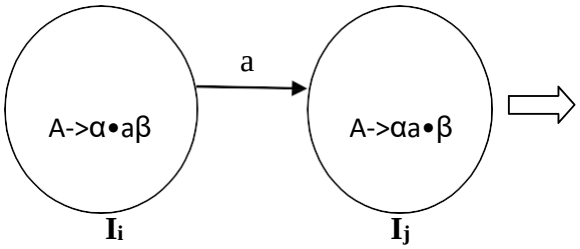
Go to **(I_0, E)** is **E→ E•+T,E`→ E. = Closure ({E`→E•, E →E•+T})**



Construction of LR (0) parsing Table:

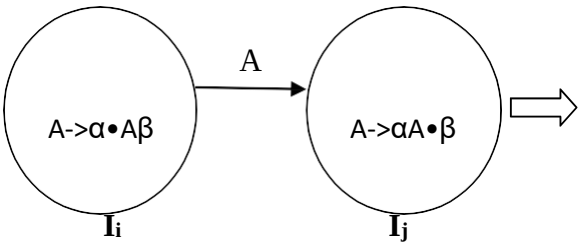
Once we have Created the canonical collection of LR (0) items, need to follow the steps mentioned below:

If there is a transaction from one state (I_i) to another state(I_j) on a terminal value then, we should write the shift entry in the action part as shown below:



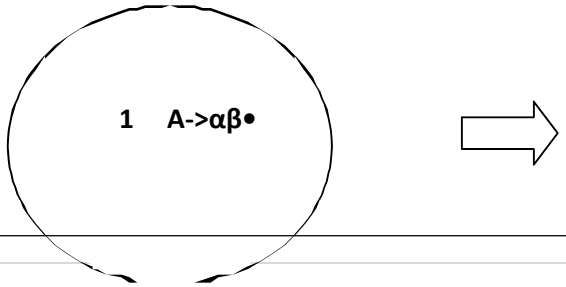
States	ACTION		GO TO
	a	$	A
I_i	S_j		
I_j			

If there is a transaction from one state (I_i) to another state (I_j) on a Non terminal value then, we should write the subscript value of I_i in the GO TO part as shown below: part as shown below:



States	ACTION		GO TO
	a	$	A
I_i			j
I_j			

If there is one state (I_i), where there is one production which has no transitions. Then, the production is said to be a reduced production. These productions should have reduced entry in the Action part along with their production numbers. If the Augment production is reducing then, write accept in the Action part.



States	ACTION		GO TO
	a	$	A
I_i	r_1	r_1	

n

I_i

I_i

For Example, Construct the LR (0) parsing Table for the given Grammar (G)

S → aB

B → bB | b

Sol: 1. Add Augment Production and insert „•" symbol at the first position for every production in G

0. S′ → •S

1. S → •aB

2. B → •bB

3. B →•b

I_0 State:

1. Add Augment production to the I_0 State and Compute the Closure

I_0 = Closure (S′→ •S)

Since _•' is followed by the Non terminal, add all productions starting with S in to I_0 State. So, the I_0 State becomes

I_0 = S′ → •S

S → •aB Here, in the S production _.' Symbol is followed by a terminal value so close the state.

I_1= Go to (I_0, S)

S`→ S•

Closure(S`→ S•) = S′→ S• Here, The Production is reduced so close the State.

I_1= S′ → S•

I_2= Go to (I_0, a) = closure (S → a•B)

Here, the _•' symbol is followed by The Non terminal B. So, add the productions which are Starting B.

I_2= B → •bB

B → •b Here, the _•' symbol in the B production is followed by the terminal value. So, Close the State.

I_2= S → a•B

B → •bB

B → •b

I_3= Go to (I_2 ,B) = Closure (S → aB•) = S → aB•

I_4= Go to (I_2 , b) = closure ({B → b•B, B→ b•})

Add productions starting with B in I_4.

B → •bB
B → •b The Dot Symbol is followed by the terminal value. So, close the State.

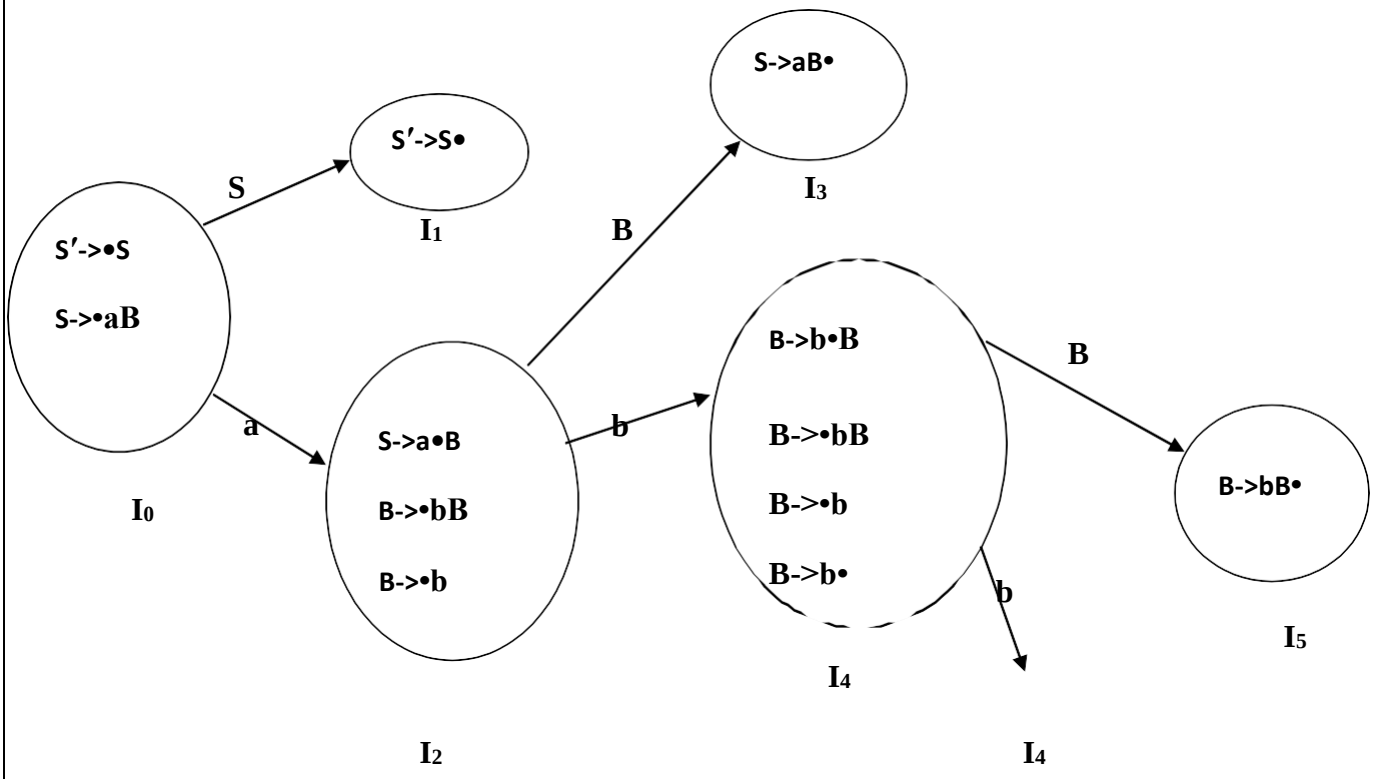
I_4= B → b•B
B → •bB
B → •b
B → b•

I_5= Go to (I_2, b) = Closure (B → b•) = B → b•

I_6= Go to (I_4, B) = Closure (B → bB•) = B → bB•

I_7 = Go to (I_4 , b) = I_4

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



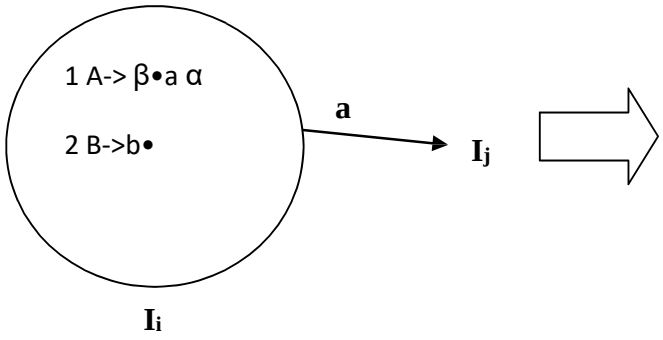
LR Parsing Table:

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.

Shift-Reduce Conflict in LR (0) Parsing: Shift Reduce Conflict in the LR (0) parsing occurs when a state has

1. A Reduced item of the form A → α• and
2. An incomplete item of the form A → β•aα as shown below:

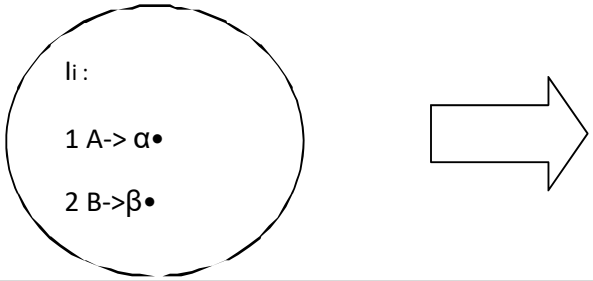


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2	r_2		
I_j				

Reduce - Reduce Conflict in LR (0) Parsing:

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or more reduced items of the form

1. **A → α•**
2. **B → β•** as shown below:



States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$	r_1/ r_2		

SLR PARSER CONSTRUCTION: What is SLR (1) Parsing

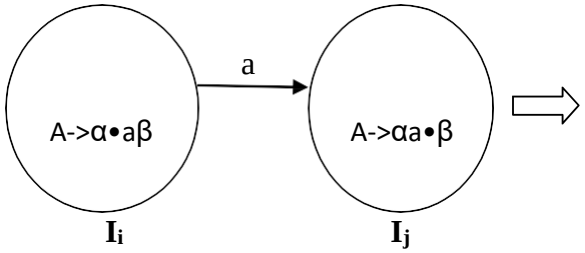
Various steps involved in the SLR (1) Parsing are:

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 SLR (1) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

SLR (1) Parsing Table Construction

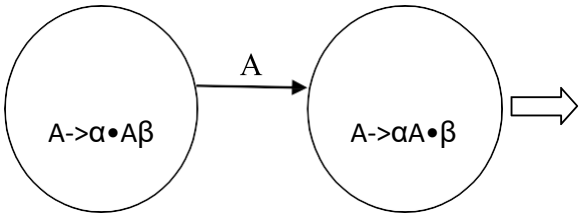
Once we have Created the canonical collection of LR (0) items, need to follow the steps mentioned below:

If there is a transaction from one state ($\mathbf{I_i}$) to another state($\mathbf{I_j}$) on a terminal value then, we should write the shift entry in the action part as shown below:



States	ACTION		GO TO
	a	$	A
$\mathbf{I_i}$	$\mathbf{S_j}$		
$\mathbf{I_j}$			

If there is a transaction from one state ($\mathbf{I_i}$) to another state ($\mathbf{I_j}$) on a Non terminal value then, we should write the subscript value of $\mathbf{I_i}$ in the GO TO part as shown below: part as shown below:



States	ACTION		GO TO
	a	$	A
$\mathbf{I_i}$			**j**
$\mathbf{I_j}$			

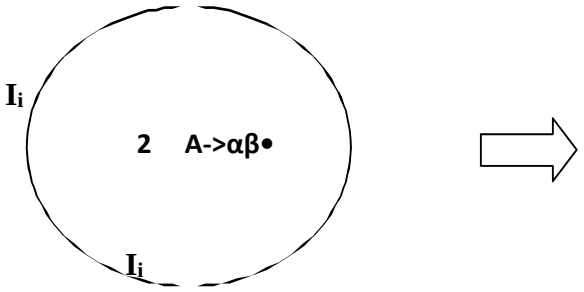
I_i I_j

1 If there is one state (**I_i**), where there is one production (**A->αβ•**) which has no transitions to the next State. Then, the production is said to be a reduced production. For allterminals X in FOLLOW (A), write the reduce entry along with their production numbers. If the Augment production is reducing then write accept.

1 S -> •aAb

2 A->αβ•

Follow(S) = {$}
Follow (A) = (b}



States	ACTION			GO TO	
	a	b	$	S	A
I_i		r_2			

SLR (1) table for the Grammar

S → aB
B → bB | b

Follow (S) = {$}, Follow (B) = {$}

States	ACTION			GOTO	
	A	**b**	**$**	**S**	**B**
I_0	S_2			1	
I_1			ACCEPT		
I_2		S_4			3
I_3			R_1		
I_4		S_4	R_3		5
I_5			R_2		

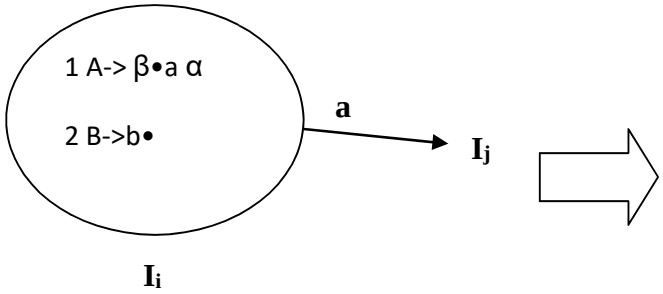
Note: When Multiple Entries occurs in the SLR table. Then, the grammar is not accepted by SLR(1) Parser.

Conflicts in the SLR (1) Parsing :

When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in SLR (1) Parsing : Shift Reduce Conflict in the LR (1) parsing occurs when a state has

1. A Reduced item of the form A → α• and Follow(A) includes the terminal value ‗a'.
2. An incomplete item of the form A → β•aα as shown below:



States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce - Reduce Conflict in SLR (1) Parsing

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or more reduced items of the form

1. **A → α•**
2. **B → β• and Follow (A) ∩ Follow(B) ≠ null** as shown below:

If The Grammar is

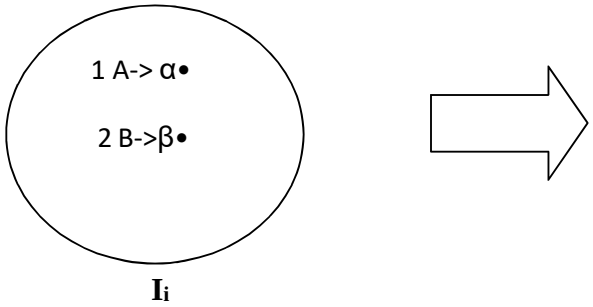
S-> αAaBa

A-> α

B-> β

Follow(S)= {$}

Follow(A)={a} and Follow(B)= {a}



States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

Canonical LR (1) Parsing: Various steps involved in the CLR (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 (1) items
5. Draw DFA
6. Construct the CLR (1) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

LR (1) items :

The LR (1) item is defined by **production**, **position of data** and a **terminal symbol**. The terminal is called as ***Look ahead symbol***.

General form of LR (1) item is

S->α•Aβ , $
A-> •γ, FIRST(β,$)

Rules to create canonical collection:

1. Every element of I is added to closure of I
2. If an LR (1) item [**X-> A•BC, a]** exists in I, and there exists a production B->b_1b_2....., then add item **[B->• b_1b_2, z]** where z is a terminal in **FIRST(Ca),** if it is not already in Closure(I).keep applying this rule until there are no more elements adde.

For example, if the grammar is

S->CC
C->cC
C->d

The Canonical collection of LR (1) items can be created as follows:

0. S′->•S (Augment Production)
1. S->•CC
2. C->•cC
3. C->•d

I_0 State : Add Augment production and compute the Closure, the look ahead symbol for the Augment Production is $.

S′->•S, $= Closure(S′->•S, $)

The dot symbol is followed by a Non terminal S. So, add productions starting with S in I_0 State.

S->•CC, FIRST ($), using 2nd rule

S->•CC, $

The dot symbol is followed by a Non terminal C. So, add productions starting with C in I_0 State.

C->•cC, FIRST(C, $)
C->•d, FIRST(C, $)

FIRST(C) = {c, d} so, the items are

C->•cC, c/d
C->•d, c/d

The dot symbol is followed by a terminal value. So, close the I_0 State. So, the productions in the **I_0** are

S'->•S , $
S->•CC , $
C->•cC, c/d
C->•d , c/d

I_1 = Goto (I_0, S)= S'->S•,$

I_2 = Go to (I_0 , C)= Closure(S-> C•C, $)

S-> **C->•cC , $**
C->•d,$ So, the I_2 State is

S->C•C,$
C->•cC , $
C->•d,$

I_3= Goto(I_0,c)= Closure(C->c•C, c/d)
C->•cC, c/d
C->•d , c/d So, the I_3 State is

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/d

I_5 = Goto (I_2, C)= closure(S->CC•,$)= S->CC•, $

I_6= Goto (I_2, c)= closure(C->c•C , $)=
C->•cC, $
C->•d , $ S0, the I_6 State is

C->c•C , $
C->•cC , $
C->•d,$

I_7 = Go to (I_2 , d)= Closure(C->d•,$) = C->d•, $

Goto(I_3, c)= closure(C->•cC, c/d)= I_3.

I_8= Go to (I_3 , C)= Closure(C->cC•, c/d) = C->cC•, c/d

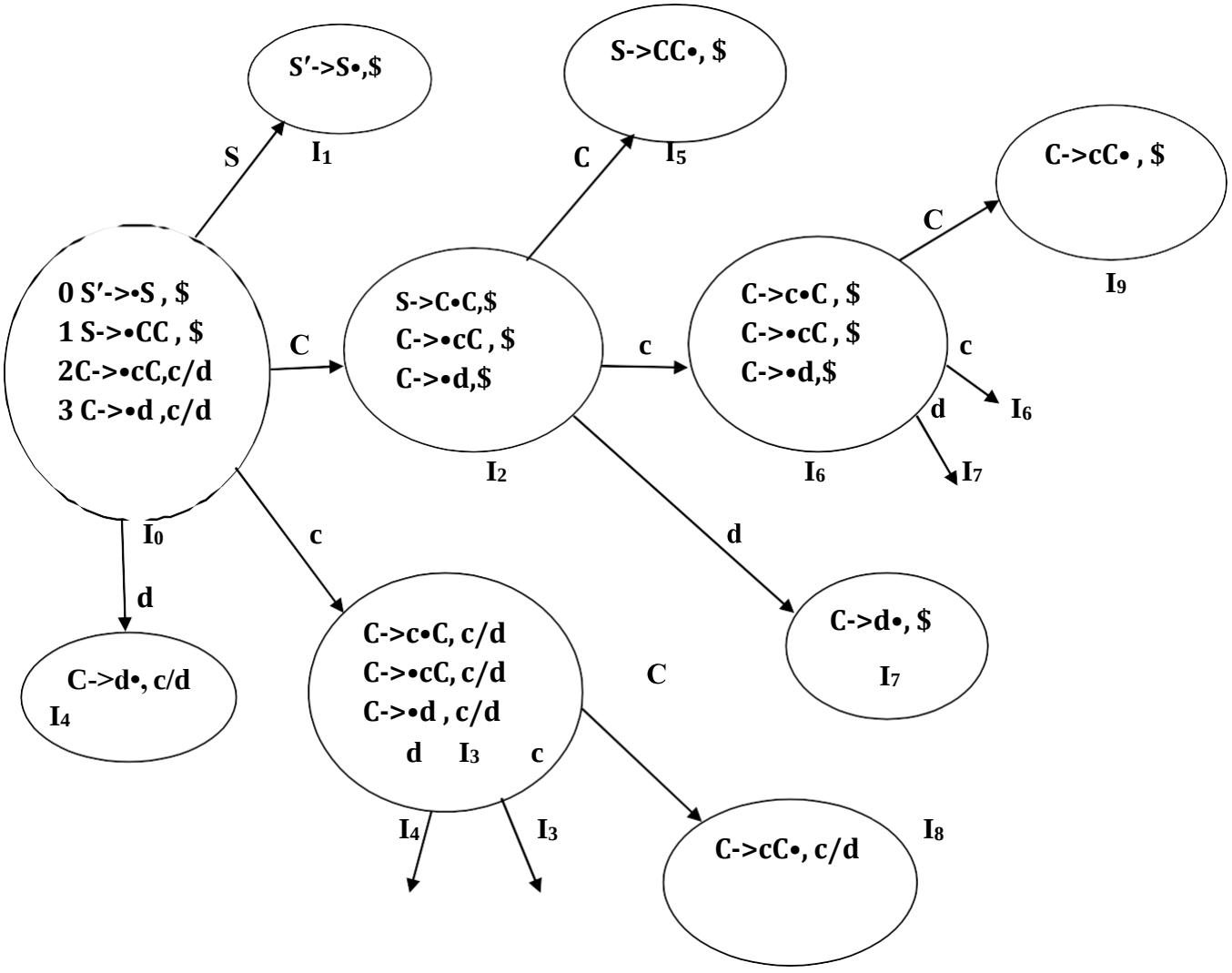
Go to (I_3 , c)= Closure(C->c•C, c/d) = I_3

Go to (I_3 , d)= Closure(C->d•, c/d) = I_4

I_9= Go to (I_6 , C)= Closure(C->cC• , $) = C->cC• , $
Go to (I_6 , c)= Closure(C->c•C , $) = I_6

Go to (I_6 , d)= Closure(C->d•,$) = I_7

Drawing the Finite State Machine DFA for the above LR (1) items



Construction of CLR (1) Table

Rule1: if there is an item [A->α•Xβ,b] in I_i and goto(I_i,X) is in I_j then action [I_i][X]= Shift j, Where X is Terminal.

Rule2: if there is an item [A->α•, b] in I_i and (A≠S`) set action [I_i][b]= reduce along with the production number.

Rule3: if there is an item [S`->S•, $] in I_i then set action [I_i][$]= Accept.

Rule4: if there is an item [A->α•Xβ,b] in I_i and go to(I_i,X) is in I_j then goto [I_i][X]= j, Where X is Non Terminal.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Table : LR (1) Table

LALR (1) Parsing

The CLR Parser avoids the conflicts in the parse table. But it produces more number of States when compared to SLR parser. Hence more space is occupied by the table in the memory. So LALR parsing can be used. Here, the tables obtained are smaller than CLR parse table. But it also as efficient as CLR parser. Here LR (1) items that have same productions but different look-aheads are combined to form a single set of items.

For example, consider the grammar in the previous example. Consider the states I_4 and I_7 as given below:

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/d

I_7 = Go to (I_2 , d)= Closure(C->d•,$) = C->d•, $

These states are differing only in the look-aheads. They have the same productions. Hence these states are combined to form a single state called as I_{47}.

Similarly the states I_3 and I_6 differing only in their look-aheads as given below:

I_3= Goto(I_0,c)=

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_6= Goto (I_2, c)=
C->c•C , $
C->•cC , $
C->•d,$

These states are differing only in the look-aheads. They have the same productions. Hence these states are combined to form a single state called as I_{36}.

Similarly the States I_8 and I_9 differing only in look-aheads. Hence they combined to form the state I_{89}.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			ACCEPT		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		5
I_5			R_1		
I_{89}	R_2	R_2	R_2		

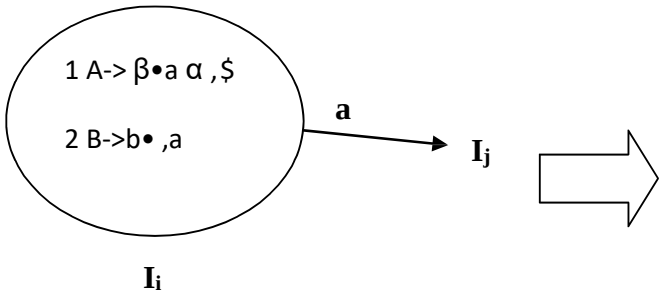
Table: LALR Table

Conflicts in the CLR (1) Parsing : When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in CLR (1) Parsing

Shift Reduce Conflict in the CLR (1) parsing occurs when a state has

3. A Reduced item of the form A → α•, a and
4. An incomplete item of the form A → β•aα as shown below:

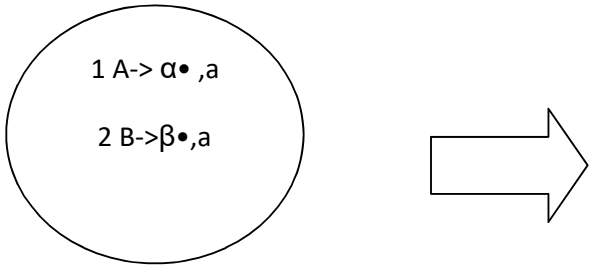


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce / Reduce Conflict in CLR (1) Parsing

Reduce- Reduce Conflict in the CLR (1) parsing occurs when a state has two or more reduced items of the form

3. **A → α•**
4. **B → β• If two productions in a state (I) reducing on same look ahead symbol as shown below:**



I_i

States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$			

String Acceptance using LR Parsing:

Consider the above example, if the input String is **cdd**

States	ACTION			GOTO	
	c	D	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

0 S′->•S (Augment Production)

1 S->•CC

2 C->•cC

3 C->•d

STACK	INPUT	ACTION
$0	cdd$	Shift S_3
$0c3	dd$	Shift S_4
$0c3d4	d$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0c3C	d$	Goto (I_3, C)=8Shift S_6

$0c3C8	d$	Reduce with R_2 ,C->cC, pop 2*β symbols from the stack
$0C	d$	Goto (I_0, C)=2
$0C2	d$	Shift S_7
$0C2d7	$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0C2C	$	Goto (I_2, C)=5
$0C2C5	$	Reduce with R_1,S->CC, pop 2*β symbols from the stack
$0S	$	Goto (I_0, S)=1
$0S1	$	Accept

Handing Ambiguous grammar

Ambiguity: A Grammar can have more than one parse tree for a string . For example, consider grammar.

string ~~str~~ing + string
| string - string
| 0 | 1 | . | 9

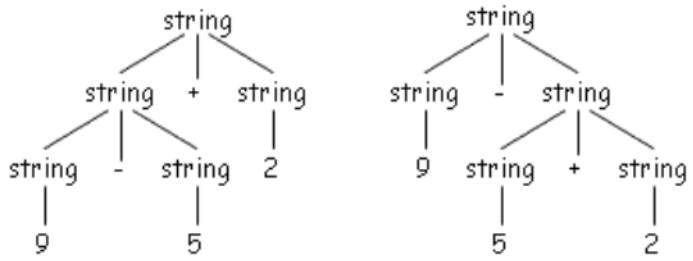
String 9-5+2 has two parse trees

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation). A language is inherently ambiguous if it can only be generated by ambiguous grammars.

For example, consider the following grammar:

string ~~str~~ing + string
| string - string
| 0 | 1 | . | 9

In this grammar, the string 9-5+2 has two possible parse trees as shown in the next slide.



Consider the parse trees for string 9-5+2, expression like this has more than one parse tree. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). The second parenthesization gives the expression the value 2 instead of 6.

- ∑ Ambiguity is problematic because meaning of the programs can be incorrect
- ∑ Ambiguity can be handled in several ways

- Enforce associativity and precedence

- Rewrite the grammar (cleanest way)

There are no general techniques for handling ambiguity, but

. It is impossible to convert automatically an ambiguous grammar to an unambiguous one

Ambiguity is harmful to the intent of the program. The input might be deciphered in a way which was not really the intention of the programmer, as shown above in the 9-5+2 example. Though there is no general technique to handle ambiguity i.e., it is not possible to develop some feature which automatically identifies and removes ambiguity from any grammar. However, it can be removed, broadly speaking, in the following possible ways:-

1) Rewriting the whole grammar unambiguously.

2) Implementing precedence and associatively rules in the grammar. We shall discuss this technique in the later slides.

If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator

. In a+b+c b is taken by left +
. +, -, *, / are left associative
. ^, = are right associative

Grammar to generate strings with right associative operators right à letter = right | letter letter
⟶ a| b |.| z

A binary operation * on a set S that does not satisfy the associative law is called non-associative. A left-associative operation is a non-associative operation that is conventionally evaluated from left to right i.e., operand is taken by the operator on the left side.
For example,
6*5*4 = (6*5)*4 and not 6*(5*4)
6/5/4 = (6/5)/4 and not 6/(5/4)

A right-associative operation is a non-associative operation that is conventionally evaluated from right to left i.e., operand is taken by the operator on the right side.

For example,

6^5^4 => 6^(5^4) and not (6^5)^4)
x=y=z=5 => x=(y=(z=5))

Following is the grammar to generate strings with left associative operators. (Note that this is left recursive and may go into infinite loop. But we will handle this problem later on by making it right recursive)

left ⟶ left + letter | letter
letter ⟶ a | b | .....| z

IMPORTANT QUESTIONS

1. Discuss the the working of Bottom up parsing and specifically the OperatorPrecedence Parsing with an exaple?
2. What do you mean by an LR parser? Explain the LR (1) Parsing technique?
3. Write the differences between canonical collection of LR (0) items and LR (1) items?
4. Write the Difference between CLR (1) and LALR(1) parsing?
5. What is YACC? Explain how do you use it in constructing the parser using it.

ASSIGNMENT QUESTIONS

1. Explain the conflicts in the Shift reduce Parsing with an example?
2. E → E+T|T
   T → T*F
   F → (E)|id , construct the LR(1) Parsing table? And explain the Conflicts?
3. E → E+T|T
   T → T*F
   F → (E)|id , construct the SLR(1) Parsing table? And explain the Conflicts?
4. E → E+T|T
   T → T*F
   F → (E)|id, construct the CLR(1) Parsing table? And explain the Conflicts?
5. E → E+T|T
   T → T*F
   F → (E) |id, construct the LALR (1) Parsing table? And explain the Conflicts?

UNIT-III

INTERMEDIATE CODE GENERATION

In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements.

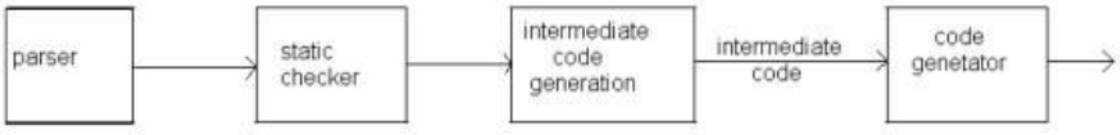


Figure 4.1 : Intermediate Code Generator

Intermediate code is:

- ∑ The output of the Parser and the input to the Code Generator.
- ∑ Relatively machine-independent and allows the compiler to be retargeted.
- ∑ Relatively easy to manipulate (optimize).

What are the Advantages of an intermediate language?

Advantages of Using an Intermediate Language includes :

1. Retargeting is facilitated - Build a compiler for a new machine by attaching a new code generator to an existing front-end.

2. Optimization - reuse intermediate code optimizers in compilers for different languages and different machines.

Note: the terms ―intermediate code‖, ―intermediate language‖, and ―intermediate representation‖ are all used interchangeably.

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

1. Syntax Trees

2. Postfix notation

3. Three Address Code

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.

Graphical Representations

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 sub-expressions are identified. A syntax tree for the assignment statement a:=b*-c+b*-cappear in the following figure.

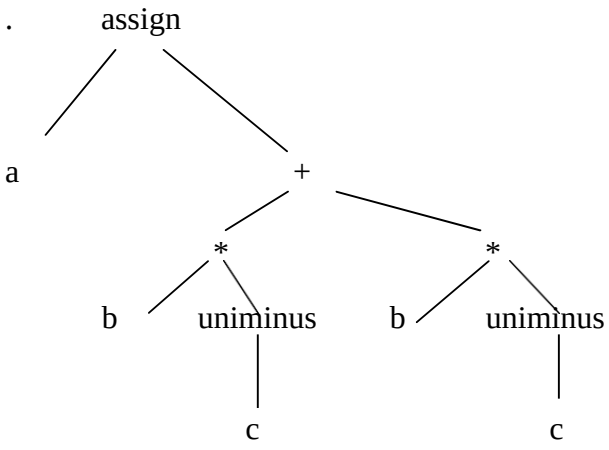


Figure 4.2 : Abstract Syntax Tree for the statement a:=b*-c+b*-c

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

a b c **uminus** + b c **uminus** * + **assign**

The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered in the order in which the nodes appear and the no. of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a staff, of an expression in postfix notation.

What is 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. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. 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. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-addresscode desirable for target code generation and optimization. The use of names for the intermediatevalues computed by a program allow- three-address code to be easily rearranged – unlike postfix notation. Three - address code is a linearzed representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

Intermediate code using Syntax for the above arithmetic expression

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

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. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer tc a symbol-table entry for that name.

Types of Three-Address Statements

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ‖back patching,‖ discussed in Section 8.6. Here are the common three-address statements used in the remainder of this book:

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 goto L is executed next, 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. Their typical use is as the sequence of three-address statements

param x1

param x2

param xn

call p, n

Generated as part of a call of the procedure p(x,, x~,..., x‖). The integer n indicating the number of actual parameters in ‖call p, n‖ is not redundant because calls can be nested. The implementation of procedure calls is outline d in Section 8.7.

7. **Indexed assignments** of the form x: = y[i] and x [i]: = y. The first of these sets x to the value in the location i memory units beyond location y. The statement x[i]:=y sets the contents ofthe location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects.

8. **Address and pointer assignments** of the form x:= &y, x:= *y and *x: = y. The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object!. In the statement x: = ~y, presumablyy is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r- value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

SYNTAX DIRECTED TRANSLATION OF THREE ADDRESS CODE

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non- terminal E on the left side of E □ E1 + E will be

computed into a new temporary t. In general, the three- address code for id: = E consists of code to evaluate E into some temporary t, followed by the assignment id.place: = t. If an expression is a single identifier, say y, then y itself holds the value of the expression. For the moment, we create a new name every time a temporary is needed; techniques for reusing temporaries aregiven in Section S.3. The S-attributed definition in Fig. 8.6 generates three-address code for assignment statements. Given input a: = b+ – c + b+ – c, it produces the code in Fig. 8.5(a). The synthesized attribute S.code represents the three- address code for the assignment S. The non- terminal 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.

The function newtemp returns a sequence of distinct names t1, t2,... in response to successive calls. For convenience, we use the notation gen(x ': =' y '+' z) in Fig. 8.6 to represent the 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 operands, like '+', are taken literally. In practice, three- address statements might be sent to an output file, rather than built up into the code attributes. Flow-of-control statements can be added to the language of assignments in Fig. 8.6 by productions and semantic rules)like the ones for while statements in Fig. 8.7. In the figure, the code for S - while E do S, 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.

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ id := E	$S.code := E.code \parallel gen(id.place\ ':='\ E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '+'\ E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '*'\ E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place\ ':='\ 'uminus'\ E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow$ id	$E.place :=$ id.place; $E.code := ''$

These attributes represent labels created by a function new label that returns a new label every time it is called.

IMPLEMENTATIONS OF THREE-ADDRESS STATEMENTS:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

QUADRUPLES:

A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. H.S(a) are for the assignment a: = b+ – c + b i – c. They are obtained from the three-address code
.The contents of fields arg 1, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

TRIPLES:

To avoid entering temporary names into the symbol table. We might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2, as Shown below. The fields arg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer-defined names or constants) or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples.‘ Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes, as in

	op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		A

Table 8.8 (a) : Qudraples

	op	Arg1	Arg2
(0)	uminus	C	
(1)	*	B	(0)
(2)	uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	:=	A	(4)

Table8.8(b) : Triples :Triples

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the arg 1 and arg2 fields can be encoded into the op field or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that

the copy statement a:= t5 is encoded in the triple representation by placing a in the arg 1 field and using the operator assign. A ternary operation like x[i]: = y requires two entries in the triple structure, as shown in Fig. 8.9(a), while x: = y[i] is naturally represented as two operations in Fig. 8.9(b).

	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

Fig. 8.9. More triple representations.

Indirect Triples

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order. Then the triples in Fig. 8.8(b) might be represented as in Fig. 8.10.

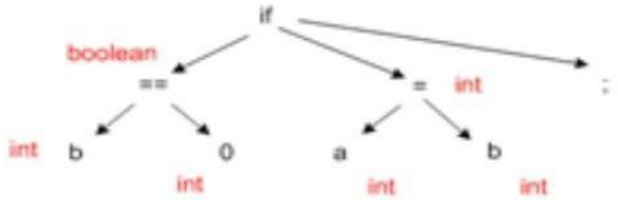
	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Figure 8.10 : Indirect Triples

SEMANTIC ANALYSIS : This phase focuses mainly on the

. Checking the semantics ,
. Error reporting
. Disambiguate overloaded operators
. Type coercion ,
. Static checking



- Type checking
-Control flow checking
- Uniqueness checking
- Name checking aspects of translation

Assume that the program has been verified to be syntactically correct and converted into some kind of intermediate representation (a parse tree). One now has parse tree available. The next phase will be semantic analysis of the generated parse tree. Semantic analysis also includes error reporting in case any semantic error is found out.

Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains symbol tables in which it stores what each symbol (variable names, function names, etc.) refers to.

FOLLOWING THINGS ARE DONE IN SEMANTIC ANALYSIS:

Disambiguate Overloaded operators: If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

TYPE CHECKING: The process of verifying and enforcing the constraints of types is called type checking. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

UNIQUENESS CHECKING: Whether a variable name is unique or not, in the its scope.

Type coersion: If some kind of mixing of types is allowed. Done in languages which are not strongly typed. This can be done dynamically as well as statically.

NAME CHECKS: Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier (Ex. int in java).

- ∑ Parser cannot catch all the program errors
- ∑ There is a level of correctness that is deeper than syntax analysis
- ∑ Some language features cannot be modeled using context free grammar formalism

- Whether an identifier has been declared before use, this problem is of identifying a language $\{w \alpha w \mid w \varepsilon \Sigma^*\}$

- This language is not context free

A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis. Typical features of semantic analysis cannot be modeled using context free grammar formalism. If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore.

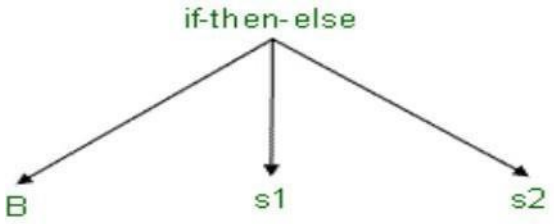
Example: in

string x; int y;
y = x + 3 the use of x is a type errorint
a, b;
a = b + c c is not declared

An identifier may refer to different variables in different parts of the program . An identifier may be usable in one part of the program but not another These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis. The third point can be explained like this: An identifier x can be declared in two separate functions in the program,once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts. The fourth point can be explained in this manner: A variable declared within one function cannot be used within the scope of the definition of the other function unless declared there separately. This is just an example. Probably you can think of many more examples in which a variable declared in one scope cannotbe used in another scope.

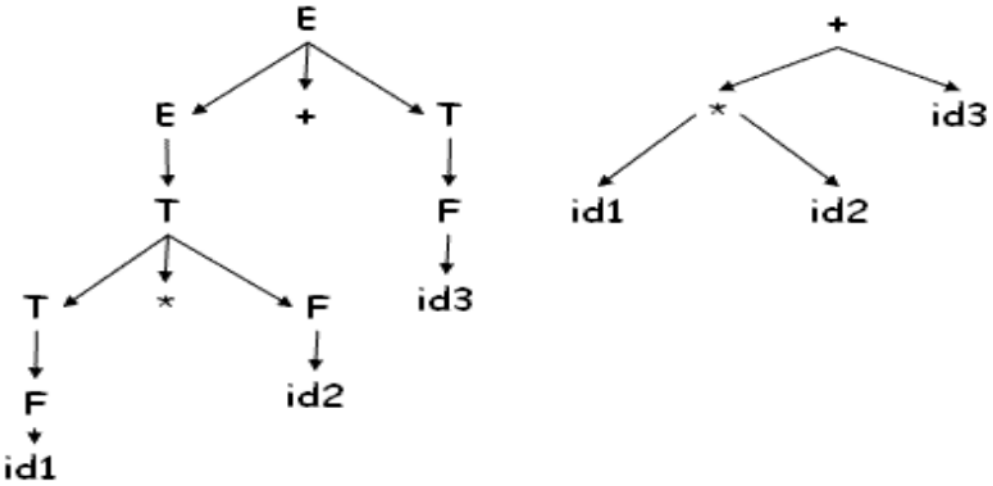
ABSTRACT SYNTAX TREE: Is nothing but the condensed form of a parse tree, It is

- ∑ Useful for representing language constructs so naturally.
- ∑ The production S ⟶if B then s1 else s2 may appear as



In the next few slides we will see how abstract syntax trees can be constructed from syntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normally operators and keywords appear as leaves but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the parse tree. This is clearly indicated by the examples in these slides.

Chain of single productions may be collapsed, and operators move to the parent nodes



Chain of single productions are collapsed into one node with the operators moving up to become the node.

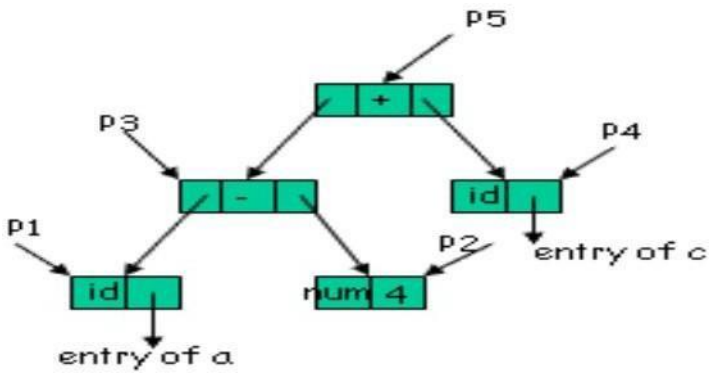
CONSTRUCTING ABSTRACT SYNTAX TREE FOR EXPRESSIONS:

In constructing the Syntax Tree, we follow the convention that :

. Each node of the tree can be represented as a record consisting of at least two fields to store operators and operands.
.***operators*** **:** one field for operator, remaining fields ptrs to operands mknode(op,left,right)
.***identifier*** **:** one field with label id and another ptr to symbol table mkleaf(id, id.entry)
.***number*** **:** one field with label num and another to keep the value of the number mkleaf(num,val)

Each node in an abstract syntax tree can be implemented as a record with several fields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

For Example: the following sequence of function calls creates a parse tree for **w= a- 4 + c**



P 1 = mkleaf(id, entry.a)
P 2 = mkleaf(num, 4)
P 3 = mknode(-, P 1 , P 2)
P 4 = mkleaf(id, entry.c)

P 5 = mknode(+, P 3 , P 4)

An example showing the formation of an abstract syntax tree by the given function calls for the expression a-4+c.The call sequence can be defined based on its postfix form, which is explained blow.

A- Write the postfix equivalent of the expression for which we want to construct a syntax tree

For above string w=a-4+c, it is **a4-c+**

B- Call the functions in the sequence, as defined by the sequence in the postfix expression which results in the desired tree. In the case above, call mkleaf() for a, mkleaf() for 4, mknode() for -, mkleaf() for c , and mknode() for + at last.

1. P1 = **mkleaf**(id, a.entry) : A leaf node made for the identifier a, and an entry for a is made in the symbol table.

2. P2 = **mkleaf**(num,4) : A leaf node made for the number 4, and entry for its value.

3. P3 = **mknode**(-,P1,P2) : An internal node for the -, takes the pointer to previously made nodes P1, P2 as arguments and represents the expression a-4.

4. P4 = **mkleaf**(id, c.entry) : A leaf node made for the identifier c , and an entry for c.entry made in the symbol table.

5. P5 = **mknode**(+,P3,P4) : An internal node for the + , takes the pointer to previously made nodes P3,P4 as arguments and represents the expression a- 4+c .

Following is the syntax directed definition for constructing syntax tree above

E ⟶E 1 + T	E.ptr = mknode(+, E1 .ptr, T.ptr)
E⟶ T	E.ptr = T.ptr
T ⟶T 1 * F	T.ptr := mknode(*, T1 .ptr, F.ptr)
T ⟶F	T.ptr := F.ptr
F ⟶(E)	F.ptr := E.ptr
F ⟶id	F.ptr := mkleaf(id, id.entry)
F⟶ num	F.ptr := mkleaf(num, val)

Now we have the syntax directed definitions to construct the parse tree for a given grammar. All the rules mentioned in slide 29 are taken care of and an abstract syntax tree is formed.

ATTRIBUTE GRAMMARS: A CFG G=(V,T,P,S), is called an Attributed Grammar iff , where in G, each grammar symbol XƐ VUT, has an associated set of attributes, and each production, p ƐP, is associated with a set of attribute evaluation rules called Semantic Actions.

In an **AG,** the **v**alues of attributes at a parse tree node are computed by semantic rules. There are two different specifications of **AGs** used by the **Semantic** Analyzer in evaluating the semantics of the program constructs. They are,

- **Syntax directed definition(SDD)s**
  - High level specifications
  - Hides implementation details
  - Explicit order of evaluation is not specified
- **Syntax directed Translation schemes(SDT)s**
  - ∑ Nothing but an SDD, which indicates order in which semantic rules are to be evaluated and
  - ∑ Allow some implementation details to be shown.

An **attribute grammar** is the formal expression of the syntax-derived semantic checks associated with a grammar. It represents the rules of a language not explicitly imparted by the syntax. In a practical way, it defines the information that is needed in the abstract syntax tree in order to successfully perform semantic analysis. This information is stored as attributes of the nodes of the abstract syntax tree. The values of those attributes are calculated by semantic rule.

There are two ways for writing attributes:

1) **Syntax Directed Definition(SDD):** Is a context free grammar in which a set of semantic actions are embedded (associated) with each production of G.

It is a high level specification in which implementation details are hidden, e.g., S.sys = A.sys + B.sys;

/* does not give any implementation details. It just tells us. This kind of attribute equation we will be using, Details like at what point of time is it evaluated and in what manner are hidden from the programmer.*/

E ⟶E1 + T	{ E.val = E1 .val+ E2.val }
E⟶ T	{ E.val = T.val }
T ⟶T 1 * F	{ T.val = T1 .val+ F.val)
T ⟶F	{ T.val = F.val }
F ⟶(E)	{ F.val = E.val }
F ⟶id	{ F.val = id.lexval }
F⟶ num	{ F.val = num.lexval }

2) **Syntax directed Translation(SDT) scheme**: Sometimes we want to control the way the attributes are evaluated, the order and place where they are evaluated. This is of a slightly lower level.

An SDT is an SDD in which semantic actions can be placed at any position in the body of the production.

For example , following SDT prints the prefix equivalent of an arithmetic expression consisting a + and * operators.

L⟶ En{ printf(„E.val‟) }
E ⟶ { printf(„+‟) }E1 + T
E⟶ T
T ⟶ { printf(„*‟) }T 1 * F
T ⟶ F
F ⟶ (E)
F ⟶ { printf(„id.lexval‟) }id
F⟶ { printf(„num.lexval‟) } num

This action in an SDT, is executed as soon as its node in the parse tree is visited in a preorder traversal of the tree.

Conceptually both the SDD and SDT schemes will:

∑ Parse input token stream
∑ Build parse tree
∑ Traverse the parse tree to evaluate the semantic rules at the parse tree nodes

Evaluation may:

∑ Generate code
∑ Save information in the symbol table
∑ Issue error messages
∑ Perform any other activity

To avoid repeated traversal of the parse tree, actions are taken simultaneously when a token is found. So calculation of attributes goes along with the construction of the parse tree.

Along with the evaluation of the semantic rules the compiler may simultaneously generate code, save the information in the symbol table, and/or issue error messages etc. at the same time while building the parse tree.

This saves multiple passes of the parse tree.

Example

Number ⟶ sign list
sign ⟶ + | -
list ⟶ list bit | bit
bit ⟶ 0 | 1

Build attribute grammar that annotates Number with the value it represents

. Associate attributes with grammar symbols

symbol	**attributes**
Number	value
sign	negative
list	position, value
bit	position, value

production Attribute rule number $\rightarrow$sign list
list.position | 0

if sign.negative

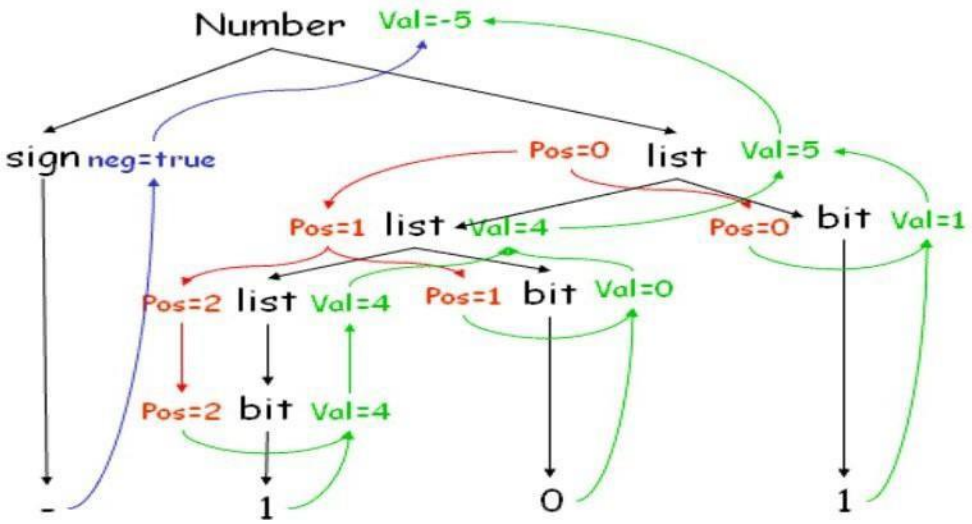
then number.value | - list.value
else number.value | list.value
sign $\rightarrow$+ sign.negative | false sign $\rightarrow$- sign.negative | true list $\rightarrow$bit
bit.position | list.position
list.value | bit.value
$list_0 \rightarrow list_1$ bit
$list_1$ .position | $list_0$ .position + 1
bit.position | $list_0$ .position
$list_0$ .value | $list_1$ .value + bit.value
bit $\rightarrow$0 bit.value | 0 bit $\rightarrow$1 bit.value | $2^{bit.position}$

Explanation of attribute rules

Num -> sign list	/*since list is the rightmost so it is assigned position 0
	*Sign determines whether the value of the number would be
	same or the negative of the value of list/
Sign -> + | -	/*Set the Boolean attribute (negative) for sign*/
List -> bit	/*bit position is the same as list position because this bit is the rightmost
	value of the list is same as bit./
List0 -> List1 bit	/*position and value calculations*/
Bit -> 0 | 1	/*set the corresponding value*/

Attributes of RHS can be computed from attributes of LHS and vice versa.

The Parse Tree and the Dependence graph are as under



.

Dependence graph shows the dependence of attributes on other attributes, along with the syntax tree. Top down traversal is followed by a bottom up traversal to resolve the dependencies. Number, val and neg are synthesized attributes. Pos is an inherited attribute.

Attributes : . Attributes fall into two classes namely *synthesized attributes* and *inherited attributes*. Value of a synthesized attribute is computed from the values of its children nodes. Value of an inherited attribute is computed from the sibling and parent nodes.

The attributes are divided into two groups, called synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute evaluation rules also using the values of the inherited attributes. The values of the inherited attributes are inherited from parent nodes and siblings.

Each grammar production A ⟶a has associated with it a set of semantic rules of the form

b = f (c_1 , c_2 , ..., c_k) , Where f is a function, and either ,b is a synthesized attribute of A Or

- b is an inherited attribute of one of the grammar symbols on the right

. attribute b depends on attributes c_1 , c_2 , ..., c_k

Dependence relation tells us what attributes we need to know before hand to calculate a particular attribute.

Here the value of the attribute b depends on the values of the attributes $\mathbf{c_1}$ to $\mathbf{c_k}$. If c_1 to c_k belong to the children nodes and b to A then b will be called a synthesized attribute. And if b belongs to one among a (child nodes) then it is an inherited attribute of one of the grammar symbols on the right.

Synthesized Attributes : A syntax directed definition that uses only synthesized attributes is said to be an S- attributed definition

. A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammars but characterized by having no inherited attributes at all. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis, pose a problem for bottom-up parsing because in bottom-up parsing, the parent nodesof the abstract syntax tree are created *after* creation of all of their children. Attribute evaluationin S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing .

Syntax Directed Definitions for a desk calculator program

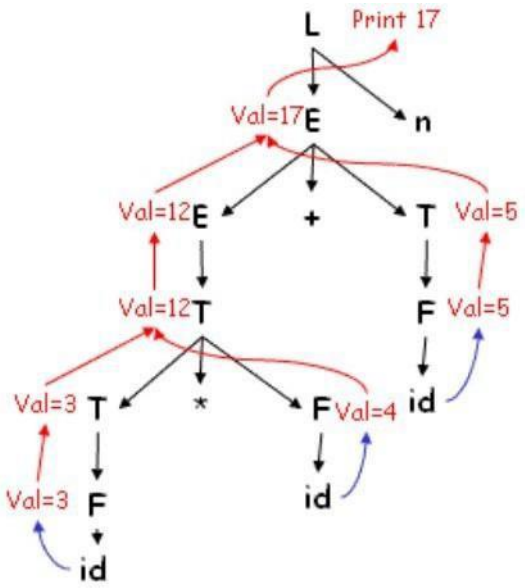
L ⟶E n	Print (E.val)
E ⟶E + T	E.val = E.val + T.val
E ⟶T	E.val = T.val
T ⟶T * F	T.val = T.val * F.val
T ⟶F	T.val = F.val
F ⟶(E)	F.val = E.val
F ⟶digit	F.val = digit.lexval

. terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer

. start symbol does not have any inherited attribute

This is a grammar which uses only synthesized attributes. Start symbol has no parents, hence no inherited attributes.

Parse tree for 3 * 4 + 5 n



Using the previous attribute grammar calculations have been worked out here for 3 * 4 + 5 n. Bottom up parsing has been done.

Inherited Attributes: An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings

. Used for finding out the context in which it appears

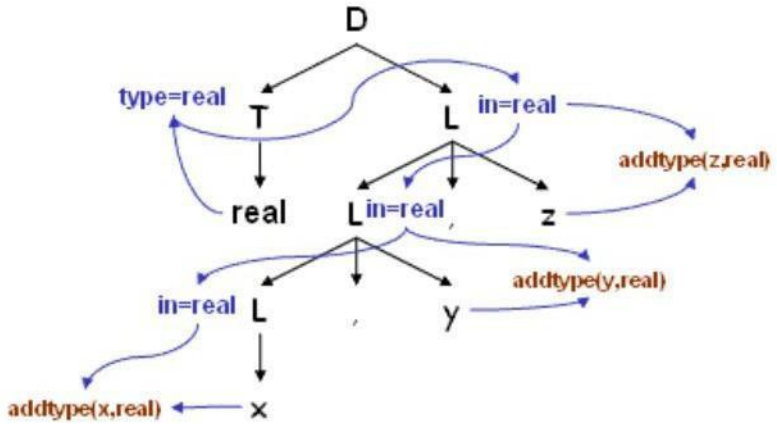
. possible to use only S-attributes but more natural to use inherited attributes

D ⟶ T L	L.in = T.type
T ⟶ real	T.type = real
T ⟶ int	T.type = int
L ⟶ L_1 , id	L_1 .in = L.in; addtype(id.entry, L.in)
L ⟶ id	addtype (id.entry,L.in)

Inherited attributes help to find the context (type, scope etc.) of a token e.g., the type of a token or scope when the same variable name is used multiple times in a program in different functions. An inherited attribute system may be replaced by an S -attribute system but it is more natural to use inherited attributes in some cases like the example given above.

Here addtype(a, b) functions adds a symbol table entry for the id a and attaches to it the type of b .

Parse tree for real x, y, z



Dependence of attributes in an inherited attribute system. The value of in (an inherited attribute) at the three L nodes gives the type of the three identifiers x , y and z . These are determined by computing the value of the attribute T.type at the left child of the root and then valuating L.in top down at the three L nodes in the right subtree of the root. At each L node the procedure addtype is called which inserts the type of the identifier to its entry in the symbol table. The figure also shows the dependence graph which is introduced later.

Dependence Graph : . If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c

. The dependencies among the nodes can be depicted by a directed graph called dependency graph

***Dependency Graph* :** Directed graph indicating interdependencies among the synthesized and inherited attributes of various nodes in a parse tree.

Algorithm to construct dependency graph

for each node **n** in the parse tree do

for each attribute **a** of the grammar symbol do

construct a node in the dependency graph

for **a**

for each node n in the parse tree do

for each semantic rule $b = f(c_1, c_2, ..., c_k)$ do

{ associated with production at n }

for i = 1 to k do

Construct an edge from c_i to b

An algorithm to construct the dependency graph. After making one node for everyattribute of all the nodes of the parse tree, make one edge from each of the other attributes on which it depends.

For example ,

- **Suppose A.a = f(X.x , Y.y) is a semantic rule for A → X Y**

- **If production A → X Y has the semantic rule X.x = g(A.a, Y.y)**

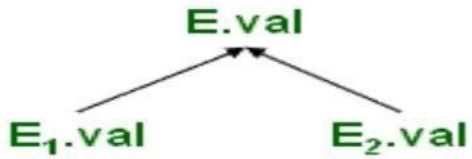
The semantic rule A.a = f(X.x , Y.y) for the production A -> XY defines the synthesized attribute a of A to be dependent on the attribute x of X and the attribute y of Y . Thus the dependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the two dependencies. Similarly for the semantic rule X.x = g(A.a , Y.y) for the same production there will be an edge from A.a to X.x and an edg e from Y.y to X.x.

Example

. Whenever following production is used in a parse tree

E ⟶E 1 + E 2 E.val = E 1 .val + E 2 .val

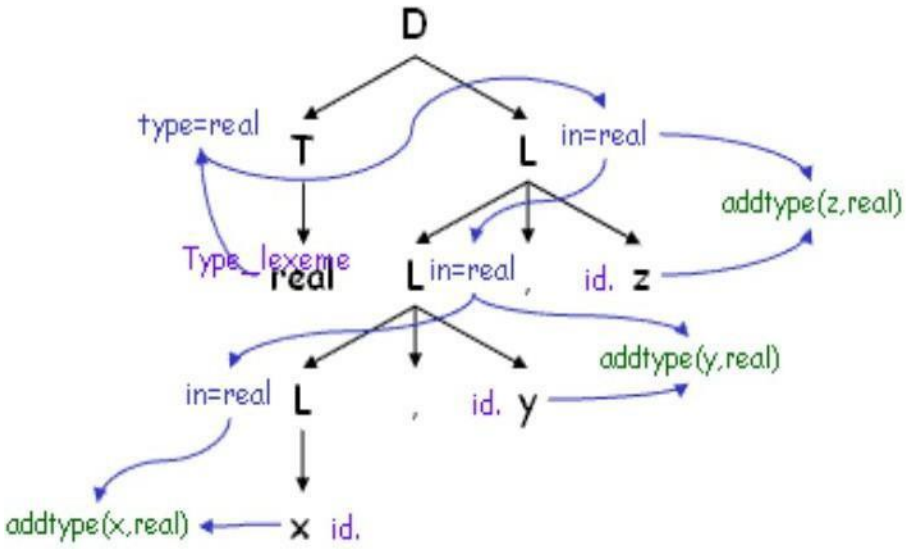
we create a dependency graph



The synthesized attribute E.val depends on E1.val and E2.val hence the two edges one each from E 1 .val & E 2 .val

For example, the dependency graph for the sting **real** id1, id2, id3

. Put a dummy synthesized attribute b for a semantic rule that consists of a procedure call



The figure shows the dependency graph for the statement real id1, id2, id3 along with the parse tree. Procedure calls can be thought of as rules defining the values of dummy synthesized attributes of the nonterminal on the left side of the associated production. Blue arrows constitute the dependency graph and black lines, the parse tree. Each of the semantic rules addtype(id.entry, L.in) associated with the L productions leads to the creation of the dummy attribute**.**

Evaluation Order :

Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

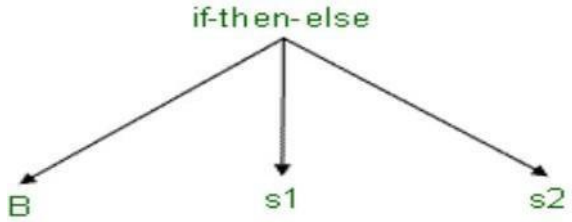
a4 = real
a5 = a4
addtype(id3.entry, a5)
a7 = a5
addtype(id2.entry, a7)

a9 := a7 addtype(id1.entry, a9)

A topological sort of a directed acyclic graph is any ordering m1, m2, m3 .....mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes. Thus if mi -> mj is an edge from mi to mj then mi appears before mj in the ordering. The order of the statements shown in the slide is obtained from the topological sort of the dependency graph in the previous slide. 'an' stands for the attribute associated with the node numbered n in the dependency graph. The numbering is as shown in the previous slide.

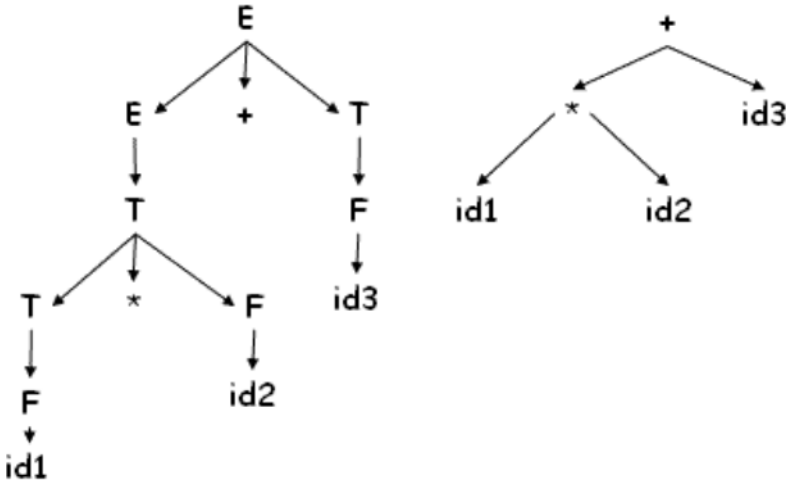
Abstract Syntax Tree is the condensed form of the parse tree, which is

. Useful for representing language constructs.
. The production : **S** ⟶**if** B **then** s1 else s2 may appear as



In the next few slides we will see how abstract syntax trees can be constructed from syntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normally operators and keywords appear as leaves but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the parse tree. This is clearly indicated by the examples in these slides.

. Chain of single productions may be collapsed, and operators move to the parent nodes



Chain of single production are collapsed into one node with the operators moving up to become the node.

For Constructing the Abstract Syntax tree for expressions,

. Each node can be represented as a record

. *operators* : one field for operator, remaining fields ptrs to operands mknode(op,left,right)

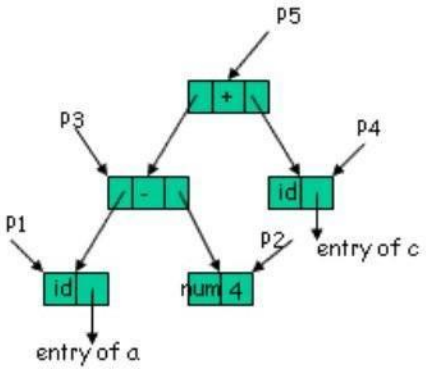
. *identifier* : one field with label id and another ptr to symbol table mkleaf(id,entry)

. *number* : one field with label num and another to keep the value of the number mkleaf(num,val)

Each node in an abstract syntax tree can be implemented as a record with several fields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

Example : **The following sequence of function calls creates a parse tree for a- 4 + c**

P 1 = mkleaf(id, entry.a)
P 2 = mkleaf(num, 4)
P 3 = mknode(-, P 1 , P 2)
P 4 = mkleaf(id, entry.c)
P 5 = mknode(+, P 3 , P 4)



An example showing the formation of an abstract syntax tree by the given function calls for the expression a-4+c.The call sequence can be explained as:

1. P1 = mkleaf(id,entry.a) : A leaf node made for the identifier Qa R and an entry for Qa R is made in the symbol table.
2. P2 = mkleaf(num,4) : A leaf node made for the number Q4 R.
3. P3 = mknode(-,P1,P2) : An internal node for the Q- Q.I takes the previously made nodes as arguments and represents the expression Qa-4 R.
4. P4 = mkleaf(id,entry.c) : A leaf node made for the identifier Qc R and an entry for Qc R is made in the symbol table.
5. P5 = mknode(+,P3,P4) : An internal node for the Q+ Q.I takes the previously made nodes as arguments and represents the expression Qa- 4+c R.

A syntax directed definition for constructing syntax tree

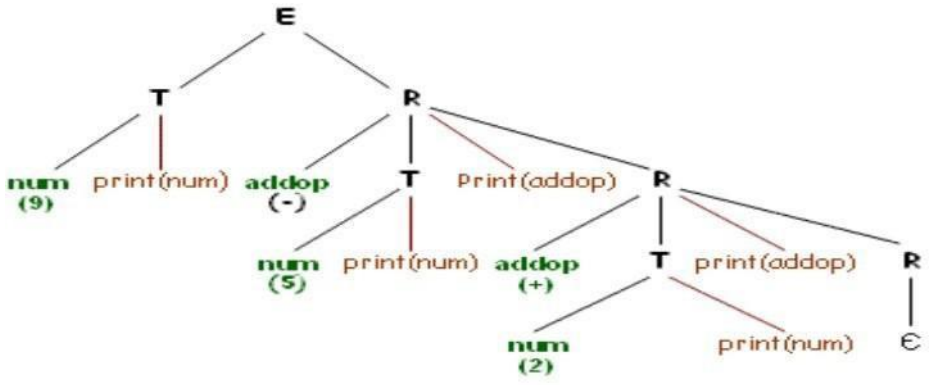
E ⟶E 1 + T	E.ptr = mknode(+, E 1 .ptr, T.ptr)
E⟶ T	E.ptr = T.ptr
T ⟶T 1 * F	T.ptr := mknode(*, T 1 .ptr, F.ptr)
T ⟶F	T.ptr := F.ptr
F ⟶(E)	F.ptr := E.ptr
F ⟶id	F.ptr := mkleaf(id, entry.id)
F⟶ num	F.ptr := mkleaf(num,val)

Now we have the syntax directed definitions to construct the parse tree for a given grammar. All the rules mentioned in slide 29 are taken care of and an abstract syntax tree is formed.

Translation schemes : A CFG where semantic actions occur within the right hand side of production, A translation scheme to map infix to postfix.

E ⟶ T R
R⟶ addop T { print(addop)} R | e
T ⟶ num {print(num)}

Parse tree for 9 - 5 + 2



We assume that the actions are terminal symbols and Perform depth first order traversal to obtain 9 5 - 2 +.

- ∑ When designing translation scheme, ensure attribute value is available when referred to
- ∑ In case of synthesized attribute it is trivial (why?)

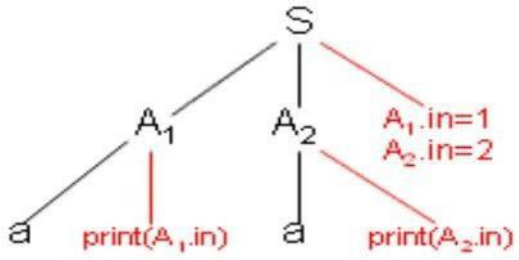
In a translation scheme, as we are dealing with implementation, we have to explicitly worry about the order of traversal. We can now put in between the rules some actions as part of the RHS. We put this rules in order to control the order of traversals. In the given example, we have two terminals (num and addop). It can generally be seen as a number followed by R (which

necessarily has to begin with an addop). The given grammar is in infix notation and we need to convert it into postfix notation. If we ignore all the actions, the parse tree is in black, without the red edges. If we include the red edges we get a parse tree with actions. The actions are so far treated as a terminal. Now, if we do a depth first traversal, and whenever we encounter a action we execute it, we get a post-fix notation. In translation scheme, we have to take care of the evaluation order; otherwise some of the parts may be left undefined. For different actions, different result will be obtained. Actions are something we write and we have to control it. Please note that translation scheme is different from a syntax driven definition. In the latter, we do not have any evaluation order; in this case we have an explicit evaluation order. By explicit evaluation order we have to set correct action at correct places, in order to get the desired output. Place of each action is very important. We have to find appropriate places, and that is that translation scheme is all about. If we talk of only synthesized attribute, the translation scheme is very trivial. This is because, when we reach we know that all the children must have been evaluated and all their attributes must have also been dealt with. This is because finding the placefor evaluation is very simple, it is the rightmost place.

In case of both inherited and synthesized attributes

. An inherited attribute for a symbol on rhs of a production must be computed in an action before that symbol

S →A 1 A 2 {A 1 .in = 1,A 2 .in = 2}
A →a {print(A.in)}



Depth first order traversal gives error *undefined*

. A synthesized attribute for non terminal on the lhs can be computed after all attributes it references, have been computed. The action normally should be placed at the end of rhs

We have a problem when we have both synthesized as well as inherited attributes. For the given example, if we place the actions as shown, we cannot evaluate it. This is because, when doing a depth first traversal, we cannot print anything for A1. This is because A1 has not yet been initialized. We, therefore have to find the correct places for the actions. This can be that the inherited attribute of A must be calculated on its left. This can be seen logically from the definition of L-attribute definition, which says that when we reach a node, then everything on its left must have been computed. If we do this, we will always have the attribute evaluated at the

correct place. For such specific cases (like the given example) calculating anywhere on the left will work, but generally it must be calculated immediately at the left.

Example: Translation scheme for EQN

S →B	B.pts = 10
	S.ht = B.ht
B →B1 B2	B_1 .pts = B.pts
	B_2 .pts = B.pts
	B.ht = max(B_1 .ht,B_2 .ht)
B →B1 sub B 2	B_1 .pts = B.pts;
	B_2 .pts = shrink(B.pts)
	B.ht = disp(B_1 .ht,B_2 .ht)
B →text	B.ht = text.h * B.pts

We now look at another example. This is the grammar for finding out how do I compose text. EQN was equation setting system which was used as an early type setting system for UNIX. It was earlier used as an latex equivalent for equations. We say that start symbol is a block: S - >B We can also have a subscript and superscript. Here, we look at subscript. A Block is composedof several blocks: B -> B1B2 and B2 is a subscript of B1. We have to determine what is the pointsize (inherited) and height Size (synthesized). We have the relevant function for height and point size given along side. After putting actions in the right place

$S \longrightarrow$ {B.pts = 10} B
{S.ht = B.ht}

$B \longrightarrow$ {B_1.pts = B.pts} B_1
{B_2.pts = B.pts} B_2
{B.ht = max(B_1.ht,B_2.ht)}

$B \longrightarrow$ {B_1.pts = B.pts} B_1 sub
{B_2.pts = shrink(B.pts)} B_2
{B.ht = disp(B_1.ht,B_2.ht)}

$B \longrightarrow$ text {B.ht = text.h * B.pts}

We have put all the actions at the correct places as per the rules stated. Read it from left to right, and top to bottom. We note that all inherited attribute are calculated on the left of B symbols and synthesized attributes are on the right.

Top down Translation: Use predictive parsing to implement L-attributed definitions
E →E1 + T E.val := E 1 .val + T.val

E →E1 - T E.val := E 1 .val - T.val
E →T E.val := T.val
T →(E) T.val := E.val
T →num T.val := num.lexval

We now come to implementation. We decide how we use parse tree and L-attribute definitions to construct the parse tree with a one-to-one correspondence. We first look at the top-down translation scheme. The first major problem is left recursion. If we remove left recursion by our standard mechanism, we introduce new symbols, and new symbols will not work with the existing actions. Also, we have to do the parsing in a single pass.

TYPE SYSTEM AND TYPE CHECKING:

. If both the operands of arithmetic operators +, -, x are integers then the result is of type integer
. The result of unary & operator is a pointer to the object referred to by the operand.
- If the type of operand is ***X*** then type of result is ***pointer to X***

In Pascal, types are classified under:

1. *Basic* types: These are atomic types with no internal structure. They include the types boolean, character, integer and real.

2. *Sub-range* types: A sub-range type defines a range of values within the range of another type. For example, type A = 1..10; B = 100..1000; U = 'A'..'Z';

3. *Enumerated* types: An enumerated type is defined by listing all of the possible values for the type. For example: type Colour = (Red, Yellow, Green); Country = (NZ, Aus, SL, WI, Pak, Ind, SA, Ken, Zim, Eng); Both the sub-range and enumerated types can be treated as basic types.

4. *Constructed* types: A constructed type is constructed from basic types and other basic types. Examples of constructed types are arrays, records and sets. Additionally, pointers and functions can also be treated as constructed types.

TYPE EXPRESSION:

It is an expression that denotes the type of an expression. The type of a language construct is denoted by a type expression

- ∑ It is either a basic type or it is formed by applying operators called *type constructor* to other type expressions
- ∑ A type constructor applied to a type expression is a type expression
- ∑ A basic type is type expression

- *type error* : error during type checking
- *void* : no type value

The type of a language construct is denoted by a type expression. A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. Formally, a type expression is recursively defined as:

1. A basic type is a type expression. Among the basic types are *boolean* , *char* , *integer* , and *real* .A special basic type, *type_error* , is used to signal an error during type checking. Another special basic type is *void* which denotes "the absence of a value" and is used to check statements.
2. Since type expressions may be named, a type name is a type expression.
3. The result of applying a type constructor to a type expression is a type expression.
4. Type expressions may contain variables whose values are type expressions themselves.

TYPE CONSTRUCTORS: are used to define or construct the type of user defined types based on their dependent types.
Arrays : If T is a type expression and I is a range of integers, then *array* (*I* , *T*) is the type expression denoting the type of array with elements of type T and index set I.

For example, the Pascal declaration, var A: array[1 .. 10] of integer; associates the type expression ***array* (1..10, *integer*)** with A.

Products *:* If *T1* and *T2* are type expressions, then their Cartesian product ***T1 X T2*** is also a type expression.

Records *:* A record type constructor is applied to a tuple formed from field names and field types. For example, the declaration

Consider the declaration

```
type row = record
addr : integer;
lexeme : array [1 .. 15] of char
end;
var table: array [1 .. 10] of row;
```

The type row has type expression : **record ((addr x integer) x (lexeme x array(1 .. 15, char)))** and type expression of table is **array(1 .. 10, row)**

Note: Including the field names in the type expression allows us to define another record type with the same fields but with different names without being forced to equate the two.

Pointers: If T is a type expression, then ***pointer*** (***T***) is a type expression denoting the type "pointer to an object of type T".
For example, in Pascal, the declaration
var p: row declares variable p to have type ***pointer(* row *)***.

Functions : Analogous to mathematical functions, functions in programming languages may be defined as mapping a domain type D to a range type R. The type of such a function is denoted by the type expression D R. For example, the built-in function mod of Pascal has domain type int X int, and range type *int* . Thus we say mod has the type: **int x int -> int**

As another example, according to the Pascal declaration

function f(a, b: char) : integer;

Here the type of f is denoted by the type expression is **char x char pointer(integer)**

SPECIFICATIONS OF A TYPE CHECKER: Consider a language which consists of a sequence of declarations followed by a single expression

P ⟶ D ; E

D ⟶ D ; D | id : T

T ⟶ char | integer | array [num] of T | ^ T

E ⟶ literal | num | E mod E | E [E] | E ^

A **type checker** is a translation scheme that synthesizes the type of each expression from the types of its sub-expressions. Consider the above given grammar that generates programs consisting of a sequence of declarations D followed by a single expression E.

Specifications of a type checker for the language of the above grammar: A program generated by this grammar is

key : integer;
key mod 1999

Assumptions:

1. The language has three basic types: *char , int* and *type-error*

2. For simplicity, all arrays start at 1. For example, the declaration array[256] of char leads to the type expression *array* (1.. 256, char).

Rules for Symbol Table entry

D ⟶ id : T	addtype(id.entry, T.type)
T ⟶ char	T.type = char
T ⟶ integer	T.type = int
T ⟶ ^T₁	T.type = pointer(T_1 .type)
T ⟶ array [num] of T $_1$	T.type = array(1..num, T_1 .type)

TYPE CHECKING OF FUNCTIONS :

Consider the Syntax Directed Definition,

E →E$_1$ (E$_2$) E. type = if E_2 .type == s and

E_1 .type == s →t

then t

else type-error

The rules for the symbol table entry are specified above. These are basically the way in which the symbol table entries corresponding to the productions are done.

Type checking of functions

The production E -> E (E) where an expression is the application of one expression to another can be used to represent the application of a function to an argument. 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* **}**

This rule says that in an expression formed by applying E1 to E2, the type of E1 must be a function *s* -> *t* from the type *s* of E2 to some range type *t* ; the type of E1 (E2) is *t* . The above rule can be generalized to functions with more than one argument byconstructing a product type consisting of the arguments. Thus n arguments of type *T1 , T2*

... *Tn* can be viewed as a single argument of the type *T1 X T2 ... X Tn* . For example,

root : (real real) X real real

declares a function root that takes a function from reals to reals and a real as arguments and returns a real. The Pascal-like syntax for this declaration is

function root (function f (real) : real; x: real) : real

TYPE CHECKING FOR EXPRESSIONS: consider the following SDD for expressions

E →literal	E.type = char
E →num	E.type = integer
E →id	E.type = lookup(id.entry)
E →E$_1$ mod E$_2$	E.type = if E_1 .type == integer and
	E_2 .type==integer
	then integer

	else type_error
E ⟶E₁ [E₂]	E.type = if E_2 .type==integer and E_1 .type==array(s,t) then t
	else type_error
E ⟶E₁ ^	E.type = if E_1 .type==pointer(t) then t
	else type_error

To perform type checking of expressions, following rules are used. Where the synthesized attribute type for E gives the type expression assigned by the type system to the expression generated by E.

The following semantic rules say that constants represented by the tokens literal and num have type *char* and *integer* , respectively:

E -> literal { *E.type* := *char* }

E -> num { *E.type* := *integer* }

. The function *lookup* (*e*) is used to fetch the type saved in the symbol-table entry pointed to by e. When an identifier appears in an expression, its declared type is fetched and assigned to the attribute type:

E -> id { *E.type* := *lookup* (id . *entry*) }

. According to the following rule, the expression formed by applying the mod operator to two sub-expressions of type *integer* has type *integer* ; otherwise, its type is *type_error* .

E -> E1 mod E2 { *E.type* := if *E1.type* == *integer* and *E2.type* == *integer* then *integer* else *type_error* }

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

E -> E1 [E2] { *E.type* := if *E2.type* == *integer* and *E1.type* == *array* (*s, t*) then *t* else *type_error* }

Within expressions, 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:

E E1 { *E.type* := if *E1.type* == *pointer* (*t*) then *t* else *type_error* }

TYPE CHECKING OF STATEMENTS: Statements typically do not have values. Special basic type *void* can be assigned to them. Consider the SDD for the grammar below which generates Assignment statements conditional, and looping statements.

S ⟶id := E	S.Type = if id.type == E.type then void else type_error
S ⟶if E then S1	S.Type = if E.type == boolean then S1.type else type_error
S ⟶while E do S1	S.Type = if E.type == boolean then S1.type else type_error
S ⟶S1 ; S2	S.Type = if S1.type == void and S2.type == void then void else type_error

Since statements do not have values, the special basic type *void* is assigned to them, but if an error is detected within a statement, the type assigned to the statement is *type_error* .

The statements considered below are assignment, conditional, and while statements. Sequences of statements are separated by semi-colons. The productions given below can be combined with those given before if we change the production for a complete program to P -> D; S. The program now consists of declarations followed by statements.

Rules for type checking the statements are given below.

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

This rule checks that the left and right sides of an assignment statement have the same type.

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

This rule specifies that the expressions in an if -then statement must have the type *boolean* .

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

This rule specifies that the expression in a while statement must have the type *boolean* .

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

Errors are propagated by this last rule because a sequence of statements has type *void* only if each sub-statement has type *void.*

IMPORTANT & EXPECTED QUESTIONS

1. What do you mean by THREE ADDRESS CODE? Generate the three-address code for the following code.

```
begin
        PROD: = 0;
        I: =1;
do
begin
        PROD:=PROD + A[I] B[I];
        I:=I+1;
End
```

```
while I <=20
end
```

2. Write a short note on Attributed grammar & Annotated parse tree.
3. Define an intermediate code form. Explain various intermediate code forms?
4. What is Syntax Directed Translation? Construct Syntax Directed Translation scheme to convert a given arithmetic expression into three address code.
5. What are Synthesized and Inherited attributes? Explain with examples?
6. Explain SDT for Simple Type checker?
7. Define and construct triples, quadruples and indirect triple notations of an expression: a * - (b + c).

ASSIGNMENT QUESTIONS:

1. Write Three address code for the below example

```
While( i<10)
{
a= b+c*-d;
i++;
}
```

2. What is a Syntax Directed Definition? Write Syntax Directed definition to convert binary value in to decimal?

SYMBOL TABLE

Symbol Table(ST) : Is a data structure used by the **c**ompiler to keep track of scope and binding information about names

- Symbol table is changed every time a name is encountered in the source;

Changes to table occur when ever a new name is discovered; new information about an existing name is discovered

As we know the compiler uses a symbol table to keep track of scope and binding information about names. It is filled after the AST is made by walking through the tree, discovering and assimilating information about the names. There should be two basic operations - to insert a new name or information into the symbol table as and when discovered and to efficiently lookup a name in the symbol table to retrieve its information.

Two common data structures used for the symbol table organization are -

1. Linear lists:- Simple to implement, Poor performance.
2. Hash tables:- Greater programming / space overhead, but, Good performance.

Ideally a compiler should be able to grow the symbol table dynamically, i.e., insert new entries or information as and when needed.

But if the size of the table is fixed in advance then (an array implementation for example), then the size must be big enough in advance to accommodate the largest possible program.

For each entry in declaration of a name

- The format need not be uniform because information depends upon the usage of the name
- Each entry is a record consisting of consecutive words
- To keep records uniform some entries may be outside the symbol table

Information is entered into symbol table at various times. For example,

- keywords are entered initially,
- identifier lexemes are entered by the lexical analyzer.

. Symbol table entry may be set up when role of name becomes clear , attribute values are filled in as information is available during the translation process.

For each declaration of a name, there is an entry in the symbol table. Different entries need to store different information because of the different contexts in which a name can occur. An entry corresponding to a particular name can be inserted into the symbol table at different stages depending on when the role of the name becomes clear. The various attributes that anentry in the symbol table can have are lexeme, type of name, size of storage and in case of functions - the parameter list etc.

A name may denote several objects in the same block

- int x; struct x { float y, z; }

The lexical analyzer returns the name itself and not pointer to symbol table entry. A record in the symbol table is created when role of the name becomes clear. In this case two symbol table entries are created.

∑ Aattributes of a name are entered in response to declarations

- ∑ Labels are often identified by colon

The syntax of procedure / function specifies that certain identifiers are formals, characters in a name. There is a distinction between token id, lexeme and attributes of the names.
It is difficult to work with lexemes

- ∑ if there is modest upper bound on length then lexemes can be stored in symbol table
- ∑ if limit is large store lexemes separately

There might be multiple entries in the symbol table for the same name, all of them having different roles. It is quite intuitive that the symbol table entries have to be made only when the role of a particular name becomes clear. The lexical analyzer therefore just returns the name and not the symbol table entry as it cannot determine the context of that name. Attributes corresponding to the symbol table are entered for a name in response to the corresponding declaration. There has to be an upper limit for the length of the lexemes for them to be stored in the symbol table.

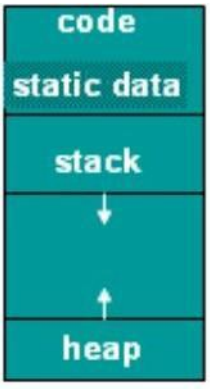
STORAGE ALLOCATION INFORMATION: Information about storage locations is kept in the symbol table.

If target code is assembly code, then assembler can take care of storage for various names and the compiler needs to generate data definitions to be appended to assembly code

If target code is machine code, then compiler does the allocation. No storage allocation is done for names whose storage is allocated at runtime

Information about the storage locations that will be bound to names at run time is kept in the symbol table. If the target is assembly code, the assembler can take care of storage for various names. All the compiler has to do is to scan the symbol table, after generating assembly code, and generate assembly language data definitions to be appended to the assembly language program for each name. If machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin must be ascertained. The compiler has to do the allocation in this case. In the case of names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all, it plans out the activation record for each procedure.

STORAGE ORGANIZATION: The runtime storage might be subdivided into :



- ∑ Target code ,
- ∑ Data objects,
- ∑ Stack to keep track of procedure activation, and
- ∑ Heap to keep all other information

This kind of organization of run-time storage is used for languages such as Fortran, Pascal and C. The size of the generated target code, as well as that of some of the data objects, is known at compile time. Thus, these can be stored

in statically determined areas in the memory.

STORAGE ALLOCATION PROCEDURE CALLS: Pascal and C use the stack for procedure activations. Whenever a procedure is called, execution of activation gets interrupted, and information about the machine state (like register values) is stored on the stack.

When the called procedure returns, the interrupted activation can be restarted after restoring the saved machine state. The heap may be used to store dynamically allocated data objects, and also other stuff such as activation information (in the case of languages where an activation tree cannot be used to represent lifetimes). Both the stack and the heap change in size during program execution, so they cannot be allocated a fixed amount of space. Generally they start from opposite ends of the memory and can grow as required, towards each other, until the space available has filled up.

ACTIVATION RECORD: An Activation Record is a data structure that is activated/ created when a procedure / function are invoked and it contains the following information about the function.

- ∑ Temporaries: used in expression evaluation
- ∑ Local data: field for local data
- ∑ Saved machine status: holds info about machine status before procedure call
- ∑ Access link : to access non local data
- ∑ Control link : points to activation record of caller
- ∑ Actual parameters: field to hold actual parameters
- ∑ Returned value : field for holding value to be returned

The activation record is used to store the information required by a single procedure call. Not all the fields shown in the figure may be needed for all languages. The record structure can be modified as per the language/compiler requirements.

For Pascal and C, the activation record is generally stored on the run-time stack during the period when the procedure is executing.

Of the fields shown in the figure, access link and control link are optional (e.g. FORTRAN doesn't need access links). Also, actual parameters and return values are often stored in registers instead of the activation record, for greater efficiency.

- ∑ The activation record for a procedure call is generated by the compiler. Generally, all field sizes can be determined at compile time.

However, this is not possible in the case of a procedure which has a local array whose size depends on a parameter. The strategies used for storage allocation in such cases will be discussedin forth coming lines.

STORAGE ALLOCATION STRATEGIES: The storage is allocated basically in the following THREE ways,

- ∑ Static allocation: lays out storage at compile time for all data objects
- ∑ Stack allocation: manages the runtime storage as a stack
- ∑ Heap allocation :allocates and de-allocates storage as needed at runtime from heap

These represent the different storage-allocation strategies used in the distinct parts of the run-time memory organization (as shown in slide 8). We will now look at the possibility of using these strategies to allocate memory for activation records. Different languages use different strategies for this purpose. For example, old FORTRAN used static allocation, Algol type languages use stack allocation, and LISP type languages use heap allocation.

STATIC ALLOCATION: In this approach memory is allocated statically. So,Names are bound to storage as the program is compiled

- ∑ No runtime support is required
- ∑ Bindings do not change at run time
- ∑ On every invocation of procedure names are bound to the same storage
- ∑ Values of local names are retained across activations of a procedure

These are the fundamental characteristics of static allocation. Since name binding occurs during compilation, there is no need for a run-time support package. The retention of local name values across procedure activations means that when control returns to a procedure, the values of the locals are the same as they were when control last left. For example, suppose we had the following code, written in a language using static allocation:

```
function F( )
 {
        int a;
        print(a);
        a = 10;
 }
```

After calling F( ) once, if it was called a second time, the value of a would initially be 10, and this is what would get printed.

The type of a name determines its storage requirement. The address for this storage is an offset from the procedure's activation record, and the compiler positions the records relative to the target code and to one another (on some computers, it may be possible to leave this relative

position unspecified, and let the link editor link the activation records to the executable code). After this position has been decided, the addresses of the activation records, and hence of the storage for each name in the records, are fixed. Thus, at compile time, the addresses at which the target code can find the data it operates upon can be filled in. The addresses at which information is to be saved when a procedure call takes place are also known at compile time. Static allocation does have some limitations.

- Size of data objects, as well as any constraints on their positions in memory, must be available at compile time.
- No recursion, because all activations of a given procedure use the same bindings for local names.
- No dynamic data structures, since no mechanism is provided for run time storage allocation.

STACK ALLOCATION: Figure shows the activation records that are pushed onto and popped for the run time stack as the control flows through the given activation tree.

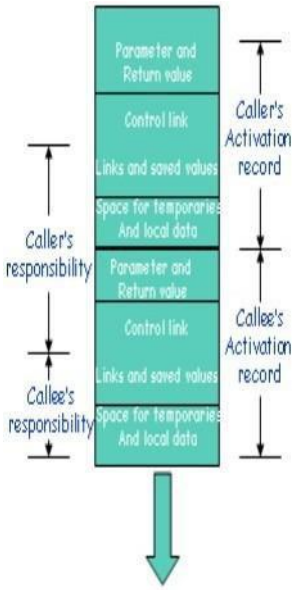
First the procedure is activated. Procedure readarray 's activation is pushed onto the stack, when the control reaches the first line in the procedure sort . After the control returns from the activation of the readarray , its activation is popped. In the activation of sort , the control then reaches a call of qsort with actuals 1 and 9 and an activation of qsort is pushed onto the top of thestack. In the last stage the activations for partition (1,3) and qsort (1,0) have begun and ended during the life time of qsort (1,3), so their activation records have come and gone from the stack, leaving the activation record for qsort (1,3) on top.

CALLING SEQUENCES: A call sequence allocates an activation record and enters information into its field. A return sequence restores the state of the machine so that calling procedure can continue execution.

Calling sequence and activation records differ, even for the same language. The code in the calling sequence is often divided between the calling procedure and the procedure it calls.

There is no exact division of runtime tasks between the caller and the colleen.

As shown in the figure, the register stack top points to the end of the machine status field in the activation record.



This position is known to the caller, so it can be made responsible for setting up stack top before control flows to the called procedure.

The code for the Callee can access its temporaries and the local data using offsets from stack top.

Call Sequence: In a call sequence, following sequence of operations is performed.

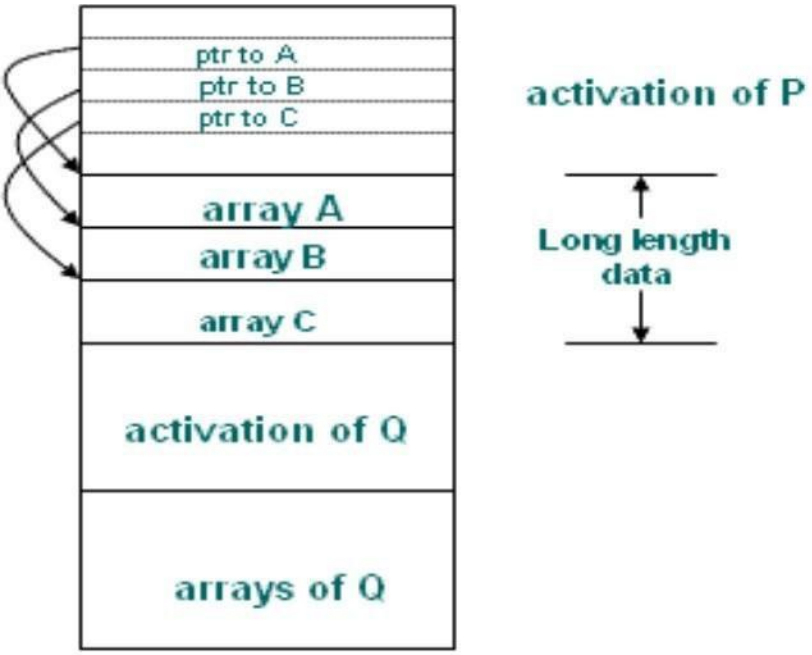
- ∑ Caller evaluates the actual parameters
- ∑ Caller stores return address and other values (control link) into callee‘s activation record
- ∑ Callee saves register values and other status information
- ∑ Callee initializes its local data and begins execution

The fields whose sizes are fixed early are placed in the middle. The decision of whether or not to use the control and access links is part of the design of the compiler, so these fields can be fixed at compiler construction time. If exactly the same amount of machine-status information is saved for each activation, then the same code can do the saving and restoring for all activations. The size of temporaries may not be known to the front end. Temporaries needed by the procedure may be reduced by careful code generation or optimization. This field is shown after that for the local data. The caller usually evaluates the parameters and communicates them to the activation record of the callee. In the runtime stack, the activation record of the caller is just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller. The caller can then access these fields using offsets from the end of its own activation record. In particular, there is no reason for the caller to know about the local data or temporaries of the callee.

Return Sequence : In a return sequence, following sequence of operations are performed.

- ∑ Callee places a return value next to activation record of caller
- ∑ Restores registers using information in status field
- ∑ Branch to return address
- ∑ Caller copies return value into its own activation record

As described earlier, in the runtime stack, the activation record of the caller is just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller. The caller can then access these fields using offsets from the end of its own activation record. The caller copies the return value into its own activation record. In particular, there is no reason for the caller to know about the local data or temporaries of the callee. The given calling sequence allows the number of arguments of the called procedure to depend on the call. At compile time, the target code of the caller knows the number of arguments it is supplying to the callee. The caller knows the size of the parameter field. The target code of the called must be prepared to handle other calls as well, so it waits until it is called, then examines the parameter field. Information describing the parameters must be placed next to the status field so the callee can find it.



Long Length Data:

The procedure P has three local arrays. The storage for these arrays is not part of the activation record for P; only a pointer to the beginning of each array appears in the activation record. The relative addresses of these pointers are known at the compile time, so the target code can access array elements through the pointers. Also shown is the procedure Q called by P . The activation record for Q begins after the arrays of P. Access to data on the stack is through two pointers, top and stack top. The first of these marks the actual top of the stack; it points to the

position at which the next activation record begins. The second is used to find the local data. For consistency with the organization of the figure in slide 16, suppose the stack top points to the end of the machine status field. In this figure the stack top points to the end of this field in the activation record for Q. Within the field is a control link to the previous value of stack top when control was in calling activation of P. The code that repositions top and stack top can be generated at compile time, using the sizes of the fields in the activation record. When q returns, the new value of top is stack top minus the length of the machine status and the parameter fields in Q's activation record. This length is known at the compile time, at least to the caller. After adjusting top, the new value of stack top can be copied from the control link of Q.

Dangling References: Referring to locations which have been de-allocated.

```
void main()
{
   int *p;
   p = dangle(); /* dangling reference */
}

int *dangle();
{
   int i=23;
   return &i;
}
```

The problem of dangling references arises, whenever storage is de-allocated. A dangling reference occurs when there is a reference to storage that has been de-allocated. It is a logical error to use dangling references, since the value of de-allocated storage is undefined according tothe semantics of most languages. Since that storage may later be allocated to another datum, mysterious bugs can appear in the programs with dangling references.

HEAP ALLOCATION: If a procedure wants to put a value that is to be used after its activation is over then we cannot use stack for that purpose. That is language like Pascal allows data to be allocated under program control. Also in certain language a called activation may outlive the caller procedure. In such a case last-in-first-out queue will not work and we will require a data structure like heap to store the activation. The last case is not true for those languages whose activation trees correctly depict the flow of control between procedures.

Limitations of Stack allocation: It cannot be used if,

- The values of the local variables must be retained when an activation ends
- A called activation outlives the caller

∑ In such a case de-allocation of activation record cannot occur in last-in first-out fashion
∑ Heap allocation gives out pieces of contiguous storage for activation records

There are two aspects of dynamic allocation -:

- Runtime allocation and de-allocation of data structures.
- Languages like Algol have dynamic data structures and it reserves some part of memory for it**.**

Initializing data-structures may require allocating memory but where to allocate this memory. After doing type inference we have to do storage allocation. It will allocate some chunk of bytes. But in language like LISP, it will try to give continuous chunk. The allocation in continuous bytes may lead to problem of fragmentation i.e. you may develop hole in process of allocation and de-allocation. Thus storage allocation of heap may lead us with many holes and fragmented memory which will make it hard to allocate continuous chunk of memory to requesting program. So,we have heap mangers which manage the free space and allocation and de-allocation of memory. It would be efficient to handle small activations and activations of predictable size as a special case as described in the next slide. The various allocation and de- allocation techniques used will be discussed later.

Fill a request of size s with block of size s ' where s ' is the smallest size greater than or equal to s

- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory so that time taken by the manager may be negligible compared to the computation time

As mentioned earlier, for efficiency reasons we can handle small activations and activations of predictable size as a special case as follows:

1. For each size of interest, keep a linked list if free blocks of that size

2. If possible, fill a request for size s with a block of size s', where s' is the smallest size greater than or equal to s. When the block is eventually de-allocated, it is returned to the linked list it came from.

3. For large blocks of storage use the heap manger.

Heap manger will dynamically allocate memory. This will come with a runtime overhead. As heap manager will have to take care of defragmentation and garbage collection. But since heap manger saves space otherwise we will have to fix size of activation at compile time, runtime overhead is the price worth it.

ACCESS TO NON-LOCAL NAMES :

The scope rules of a language decide how to reference the non-local variables. There are two methods that are commonly used:

1. Static or Lexical scoping: It determines the declaration that applies to a name by examining the program text alone. E.g., Pascal, C and ADA.
2. Dynamic Scoping: It determines the declaration applicable to a name at run time, by considering the current activations. E.g., Lisp

ORGANIZATION FOR BLOCK STRUCTURES:

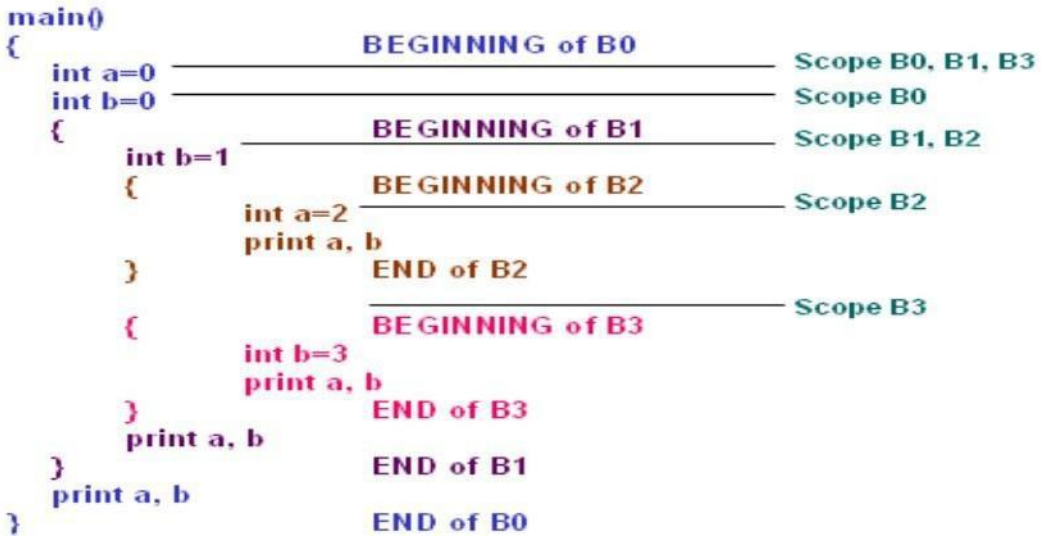
A block is a any sequence of operations or instructions that are used to perform a [sub] task. In any programming language,

- ∑ Blocks contain its own local data structure.
- ∑ Blocks can be nested and their starting and ends are marked by a delimiter.
- ∑ They ensure that either block is independent of other or nested in another block. That is, it is not possible for two blocks B1 and B2 to overlap in such a way that first block B1 begins, then B2, but B1 end before B2.
- ∑ This nesting property is called block structure. The scope of a declaration in a block-structured language is given by the most closely nested rule:

  1. The scope of a declaration in a block B includes B.

  2. If a name X is not declared in a block B, then an occurrence of X in B is in the scope of a declaration of X in an enclosing block B ' such that. B ' has a declaration of X, and. B ' is more closely nested around B then any other block with a declaration of X.

For example, consider the following code fragment.



For the example, in the above figure, the scope of declaration of b in B0 does not include B1 because b is re-declared in B1. We assume that variables are declared before the first statement in which they are accessed. The scope of the variables will be as follows:

DECLARATION	SCOPE
int a=0	B0 not including B2
int b=0	B0 not including B1
int b=1	B1 not including B3
int a =2	B2 only
int b =3	B3 only

The outcome of the print statement will be, therefore:

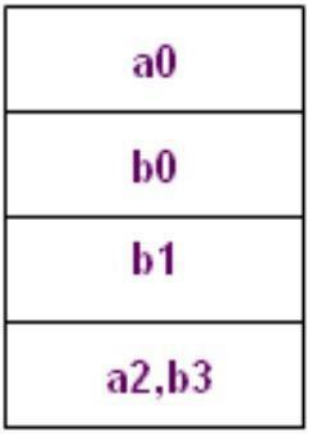
2 1
0 3
0 1
0 0

Blocks : . Blocks are simpler to handle than procedures

. Blocks can be treated as parameter less procedures

. Use stack for memory allocation

. Allocate space for complete procedure body at one time

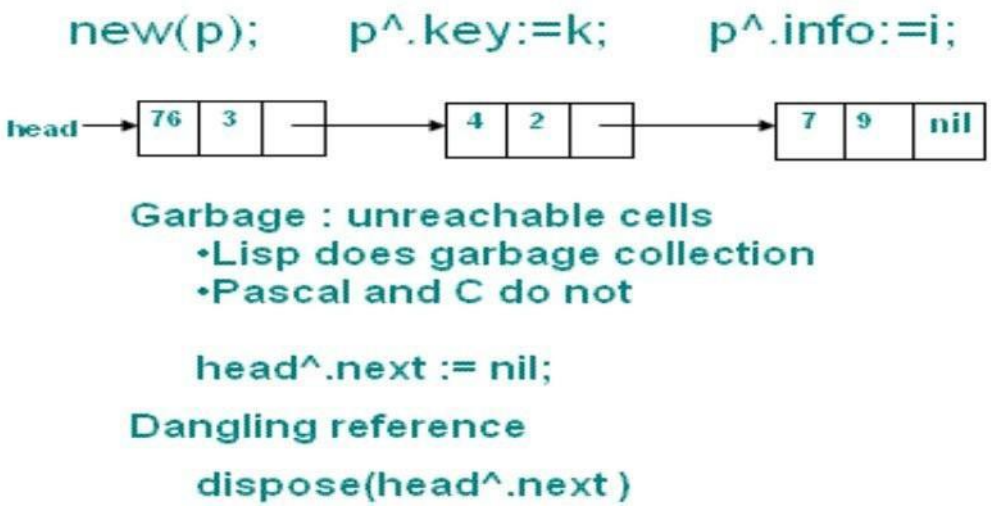


There are two methods of implementing block structure in compiler construction:

1. **STACK ALLOCATION:** This is based on the observation that scope of a declaration does not extend outside the block in which it appears, the space for declared name can be allocated when the block is entered and de-allocated when controls leave the block. The view treat blockas a "parameter less procedure" called only from the point just before the block and returning only to the point just before the block.

2. **COMPLETE ALLOCATION:** Here you allocate the complete memory at one time. If there are blocks within the procedure, then allowance is made for the storage needed for declarations within the books. If two variables are never alive at the same time and are at same depth they can be assigned same storage.

DYNAMIC STORAGE ALLOCATION:

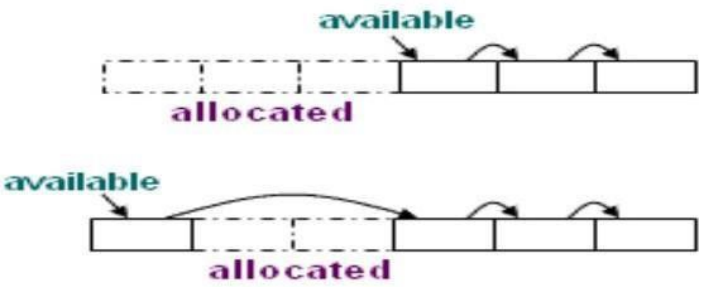


Generally languages like Lisp and ML which do not allow for explicit de-allocation of memory do garbage collection. A reference to a pointer that is no longer valid is called a 'dangling reference'. For example, consider this C code:

```
int main (void)
 {
        int* a=fun();
 }
int* fun()
 {
        int a=3;
        int* b=&a;
        return b;
 }
```

Here, the pointer returned by fun() no longer points to a valid address in memory as the activation of fun() has ended. This kind of situation is called a 'dangling reference'. In case of explicit allocation it is more likely to happen as the user can de-allocate any part of memory, even something that has to a pointer pointing to a valid piece of memory.

In Explicit Allocation of Fixed Sized Blocks , Link the blocks in a list , and Allocation and de-allocation can be done with very little overhead.



The simplest form of dynamic allocation involves blocks of a fixed size. By linking the blocks in a list, as shown in the figure, allocation and de-allocation can be done quickly with little or no storage overhead.

Explicit Allocation of Fixed Sized Blocks: In this approach, blocks are drawn from contiguous area of storage, and an area of each block is used as pointer to the next block

- ∑ The pointer available points to the first block
- ∑ Allocation means removing a block from the available list
- ∑ De-allocation means putting the block in the available list
- ∑ Compiler routines need not know the type of objects to be held in the blocks
- ∑ Each block is treated as a variant record

Suppose that blocks are to be drawn from a contiguous area of storage. Initialization of the area is done by using a portion of each block for a link to the next block. A pointer available points to the first block. Generally a list of free nodes and a list of allocated nodes is maintained, and whenever a new block has to be allocated, the block at the head of the free list is taken off and allocated (added to the list of allocated nodes). When a node has to be de-allocated, it is removed from the list of allocated nodes by changing the pointer to it in the list to point to the block previously pointed to by it, and then the removed block is added to the head of the list of free blocks. The compiler routines that manage blocks do not need to know the type of object that will be held in the block by the user program. These blocks can contain any type of data (i.e., they are used as generic memory locations by the compiler). We can treat each block as a variant record, with the compiler routines viewing the block as consisting of some other type. Thus, there is no space overhead because the user program can use the entire block for its own purposes. When the block is returned, then the compiler routines use some of the space from the block itself to link it into the list of available blocks, as shown in the figure in the last slide.

Explicit Allocation of Variable Size Blocks :

Limitations of Fixed sized block allocation: In explicit allocation of fixed size blocks, internal fragmentation can occur, that is, the heap may consist of alternate blocks that are free and in use, as shown in the figure.

The situation shown can occur if a program allocates five blocks and then de-allocates the second and the fourth, for example.

Fragmentation is of no consequence if blocks are of fixed size, but if they are of variable size, a situation like this is a problem, because we could not allocate a block larger than any one of the free blocks, even though the space is available in principle.

So, if variable- sized blocks are allocated, then internal fragmentation can be avoided, as we only allocate as much space as we need in a block. But this creates the problem of external fragmentation, where enough space is available in total for our requirements, but not enough

space is available in continuous memory locations, as needed for a block of allocated memory. For example, consider another case where we need to allocate 400 bytes of data for the next request, and the available continuous regions of memory that we have are of sizes 300, 200 and 100 bytes. So we have a total of 600 bytes, which is more than what we need. But still we are unable to allocate the memory as we do not have enough contiguous storage.

The amount of external fragmentation while allocating variable-sized blocks can become very high on using certain strategies for memory allocation.

So we try to use certain strategies for memory allocation, so that we can minimize memory wastage due to external fragmentation. These strategies are discussed in the next few lines.

. Storage can become fragmented, Situation may arise, If program allocates five blocks
. then de-allocates second and fourth block

IMPORTANT QUESTIONS:

1. What are calling sequence, and Return sequences? Explain briefly.
2. What is the main difference between Static & Dynamic storage allocation? Explain the problems associated with dynamic storage allocation schemes.
3. What is the need of a display associated with a procedure? Discuss the procedures for maintaining the display when the procedures are not passed as parameters.
4. Write notes on the static storage allocation strategy with example and discuss its limitations?
5. Discuss about the stack allocation strategy of runtime environment with an example?
6. Explain the concept of implicit de allocation of memory.
7. Give an example of creating dangling references and explain how garbage is created.

ASSIGNMENT QUESTIONS:

1. What is a calling sequence? Explain briefly.
2. Explain the problems associated with dynamic storage allocation schemes.
3. List and explain the entries of Activation Record.
4. Explain about parameter passing mechanisms.

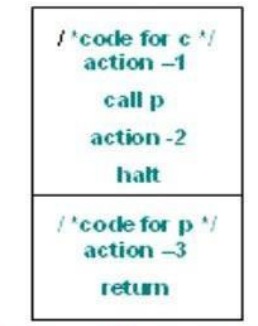
UNIT-IV

RUN TIME STORAGE MANAGEMENT:

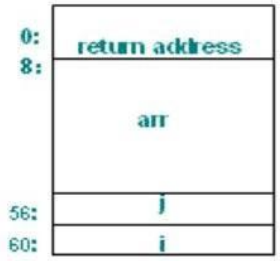
To study the run-time storage management system it is sufficient to focus on the statements: action, call, return and halt, because they by themselves give us sufficient insight into the behavior shown by functions in calling each other and returning.

And the run-time allocation and de-allocation of activations occur on the call of functions and when they return.

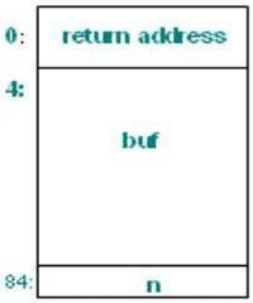
There are mainly two kinds of run-time allocation systems: **Static allocation** and **Stack Allocation**. While static allocation is used by the FORTRAN class of languages, stack allocation is used by the Ada class of languages.



Three address code



Activation record for c (64 bytes)



Activation record for p (88 bytes)

STATIC ALLOCATION: In this, A call statement is implemented by a sequence of two instructions.

- ∑ A move instruction saves the return address
- ∑ A goto transfers control to the target code.

The instruction sequence is

MOV #here+20, callee.static-area

GOTO callee.code-area

callee.static-area and callee.code-area are constants referring to address of the activation record and the first address of called procedure respectively.

. #here+20 in the move instruction is the return address; the address of the instruction following the goto instruction

. A return from procedure callee is implemented by

GOTO *callee.static-area

For the call statement, we need to save the return address somewhere and then jump to the location of the callee function. And to return from a function, we have to access the return address as stored by its caller, and then jump to it. So for call, we first say: MOV #here+20, callee.static-area. Here, #here refers to the location of the current MOV instruction, and callee.static- area is a fixed location in memory. 20 is added to #here here, as the code corresponding to the call instruction takes 20 bytes (at 4 bytes for each parameter: 4*3 for this instruction, and 8 for the next). Then we say GOTO callee. code-area, to take us to the code of the callee, as callee.codearea is merely the address where the code of the callee starts. Then a return from the callee is implemented by: GOTO *callee.static area. Note that this works only because callee.static-area is a constant.

Example:

. Assume each action block takes 20 bytes of space .Start address of code for c and p is 100 and 200

100: ACTION-l
120: MOV 140, 364
132: GOTO 200
140: ACTION-2
160: HALT
:
200: ACTION-3
220: GOTO *364

. The activation	:
Records	300:
arestatically	304:
allocated starting	:
at addresses	364:
300 and 364.	368:

This example corresponds to the code shown in slide 57. Statically we say that the code for c starts at 100 and that for p starts at 200. At some point, c calls p. Using the strategy discussed earlier, and assuming that callee.staticarea is at the memory location 364, we get the code as given. Here we assume that a call to 'action' corresponds to a single machine instruction which takes 20 bytes.

STACK ALLOCATION : . Position of the activation record is not known until run time

- ∑ . Position is stored in a register at run time, and words in the record are accessed with an offset from the register
- ∑ . The code for the first procedure initializes the stack by setting up SP to the start of the stack area

MOV #Stackstart, SP

code for the first procedure

HALT

In stack allocation we do not need to know the position of the activation record until run-time. This gives us an advantage over static allocation, as we can have recursion. So this is used in many modern programming languages like C, Ada, etc. The positions of the activations are stored in the stack area, and the position for the most recent activation is pointed to by the stack pointer. Words in a record are accessed with an offset from the register. The code for the first procedure initializes the stack by setting up SP to the stack area by the following command: MOV #Stackstart, SP. Here, #Stackstart is the location in memory where the stack starts.

A procedure call sequence increments SP, saves the return address and transfers control to the called procedure

ADD #caller.recordsize, SP

MOVE #here+ 16, *SP

GOTO callee.code_area

Consider the situation when a function (caller) calls the another function(callee), then procedure call sequence increments SP by the caller record size, saves the return address and transfers control to the callee by jumping to its code area. In the MOV instruction here, we only need to add 16, as SP is a register, and so no space is needed to store *SP. The activations keep getting pushed on the stack, so #caller.recordsize needs to be added to SP, to update the value of SP to its new value. This works as #caller.recordsize is a constant for a function, regardless of the particular activation being referred to.

DATA STRUCTURES: Following data structures are used to implement symbol tables

LIST DATA STRUCTURE : Could be an array based or pointer based list. But this implementation is

- Simplest to implement
- Use a single array to store names and information
- Search for a name is linear
- Entry and lookup are independent operations
- Cost of entry and search operations are very high and lot of time goes into book keeping

Hash table: Hash table is a data structure which gives O(1) performance in accessing any element of it. It uses the features of both array and pointer based lists.

- The advantages are obvious

REPRESENTING SCOPE INFORMATION

The entries in the symbol table are for declaration of names. When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration, according to the scoping rules of the language, must be returned. A simple approach is to maintain a separate symbol table for each scope.

Most closely nested scope rules can be implemented by adapting the data structures discussed in the previous section. Each procedure is assigned a unique number. If the language isblock-structured, the blocks must also be assigned unique numbers. The name is represented as a pair of a number and a name. This new name is added to the symbol table. Most scope rules can be implemented in terms of following operations:

a) Lookup - find the most recently created entry.
b) Insert - make a new entry.
c) Delete - remove the most recently created entry.
d) Symbol table structure
e) . Assign variables to storage classes that prescribe scope, visibility, and lifetime

f) - scope rules prescribe the symbol table structure
g) - scope: unit of static program structure with one or more variable declarations
h) - scope may be nested
i) . Pascal: procedures are scoping units
j) . C: blocks, functions, files are scoping units
k) . Visibility, lifetimes, global variables
l) . Common (in Fortran)
m) . Automatic or stack storage
n) . Static variables
o) **storage class :** A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. Ex. static, extern etc.
p) Scope: The scope of a variable is simply the part of the program where it may be accessed or written. It is the part of the program where the variable's name may be used.If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,
q)
```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```
Visibility: The visibility of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program.

r) **Local and Global variables:** A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables. On the other hand, a variable declared outside of any function is a global variable , and it is potentially visible anywhere within the program.

s) **Automatic Vs Static duration:** How long do variables last? By default, local variables (those declared within a function) have automatic duration: they spring into existence when the function is called, and they (and their values) disappear when the function

returns. Global variables, on the other hand, have static duration: they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) By default, local variables have automatic duration. To give them static duration (so that, instead ofcoming and going as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default, a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword extern: extern int j; Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword: static int k; Notice that the static keyword can do two different things: it adjuststhe duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

t) Symbol attributes and symbol table entries
u) Symbols have associated attributes
v) Typical attributes are name, type, scope, size, addressing mode etc.
w) A symbol table entry collects together attributes such that they can be easily set and retrieved
x) Example of typical names in symbol table

Name	Type
name	character string
class	enumeration
size	integer
type	enumeration

LOCAL SYMBOL TABLE MANAGEMENT :

Following are prototypes of typical function declarations used for managing local symbol table. The right hand side of the arrows is the output of the procedure and the left side has the input.

NewSymTab : SymTab ⟶SymTab
DestSymTab : SymTab ⟶SymTab
InsertSym : SymTab X Symbol ⟶boolean
LocateSym : SymTab X Symbol ⟶boolean
GetSymAttr : SymTab X Symbol X Attr ⟶boolean
SetSymAttr : SymTab X Symbol X Attr X value ⟶boolean
NextSym : SymTab X Symbol ⟶Symbol
MoreSyms : SymTab X Symbol ⟶boolean

A major consideration in designing a symbol table is that insertion and retrieval should be as fast as possible
. One dimensional table: search is very slow

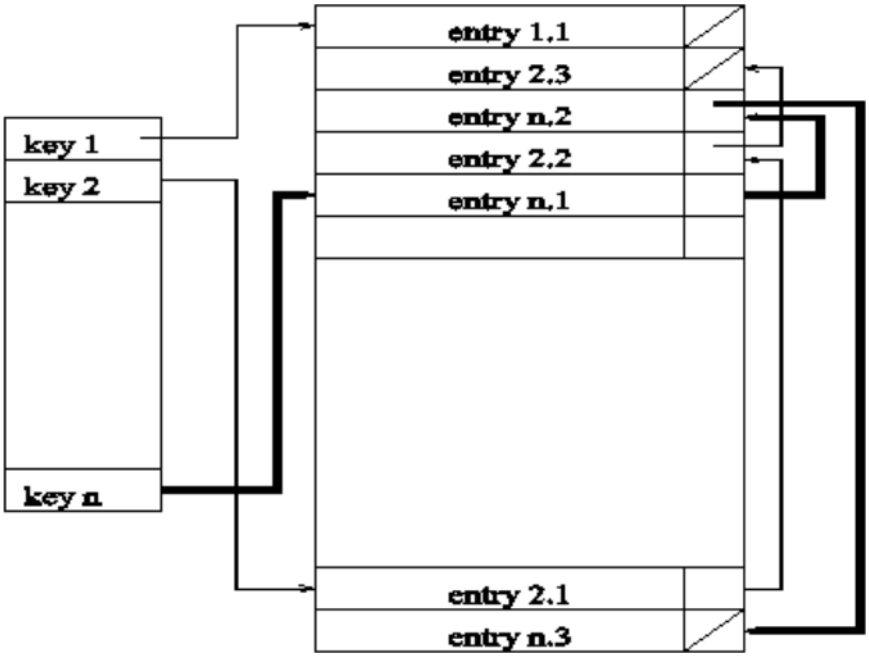
. Balanced binary tree: quick insertion, searching and retrieval; extra work required to keep the tree balanced

. Hash tables: quick insertion, searching and retrieval; extra work to compute hash keys

. Hashing with a chain of entries is generally a good approach

A major consideration in designing a symbol table is that insertion and retrieval should be as fast as possible. We talked about the one dimensional and hash tables a few slides back. Apart from these balanced binary trees can be used too. Hashing is the most common approach.

HASHED LOCAL SYMBOL TABLE



Hash tables can clearly implement 'lookup' and 'insert' operations. For implementing the 'delete', we do not want to scan the entire hash table looking for lists containing entries to be deleted. Each entry should have two links:

a) A hash link that chains the entry to other entries whose names hash to the same value - the usual link in the hash table.

b) A scope link that chains all entries in the same scope - an extra link. If the scope link is left undisturbed when an entry is deleted from the hash table, then the chain formed by the scope links will constitute an inactive symbol table for the scope in question.

Nesting structure of an example Pascal program

```
program e;
  var a, b, c: integer;

  procedure f;
    var a, b, c: integer;
    begin
       a := b+c
    end;

  procedure g;
    var a, b: integer;

    procedure h;
      var c, d: integer;
      begin
          c := a+d
      end;

    procedure i;
      var b, d: integer;
      begin
         b:= a+c
      end;

  procedure j;
    var b, d: integer;
    begin
       b := a+d
    end;

  begin
    a := b+c
  end.
```

Look at the nesting structure of this program. Variables a, b and c appear in global as well as local scopes. Local scope of a variable overrides the global scope of the other variable with the same name within its own scope. The next slide will show the global as well as the localsymbol tables for this structure. Here procedure I and h lie within the scope of g (are nested within g).

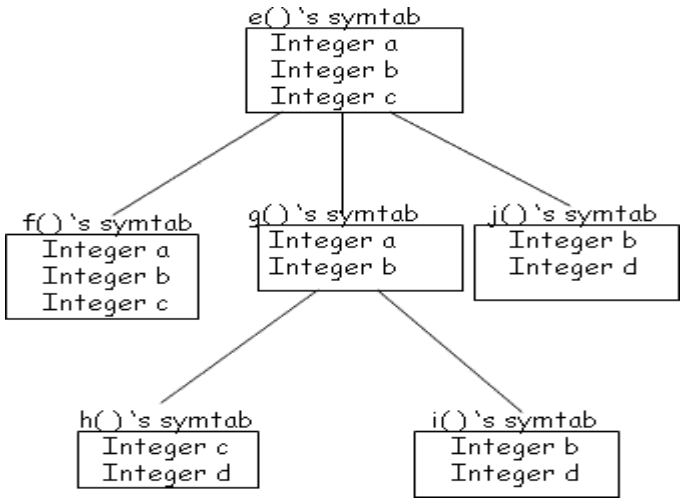
GLOBAL SYMBOL TABLE STRUCTURE The global symbol table will be a collection of symbol tables connected with pointers.

. Scope and visibility rules determine the structure of global symbol table

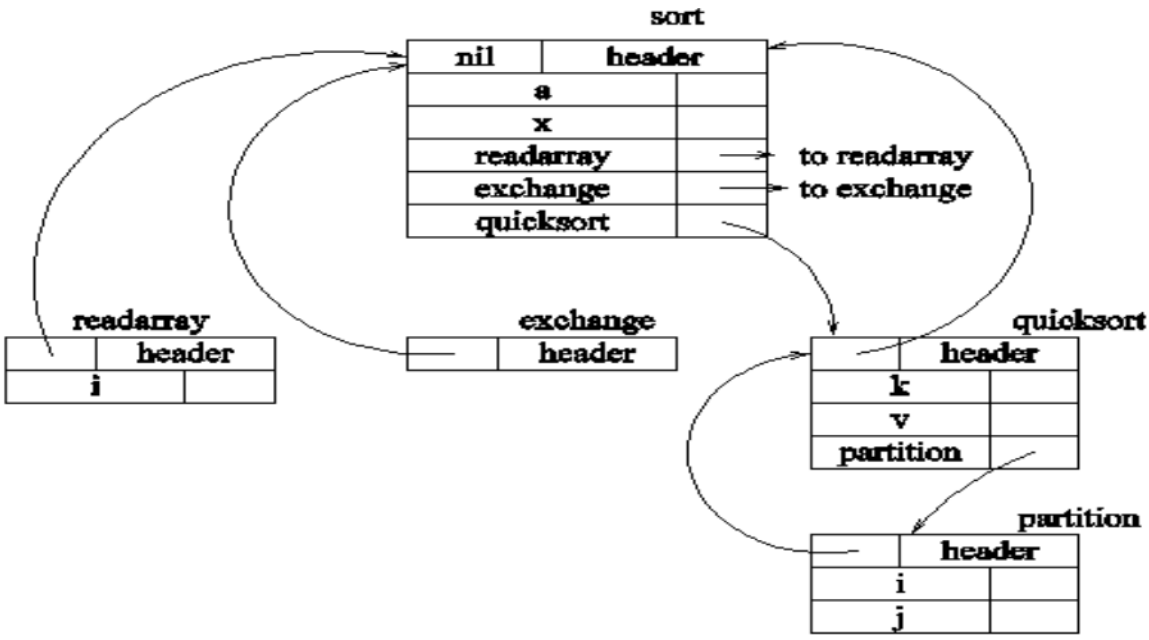
. For ALGOL class of languages scoping rules structure the symbol table as tree of local tables

- Global scope as root

- Tables for nested scope as children of the table for the scope they are nested in



The exact structure will be determined by the scope and visibility rules of the language.The global symbol table will be a collection of symbol tables connected with pointers. The exact structure will be determined by the scope and visibility rules of the language. Whenever a new scope is encountered a new symbol table is created. This new table contains a pointer back to the enclosing scope's symbol table and the enclosing one also contains a pointerto this new symbol table. Any variable used inside the new scope should either be present in its own symbol table or inside the enclosing scope's symbol table and all the way up to the root symbol table. A sample global symbol table is shown in the below figure.



BLOCK STRUCTURES AND NON BLOCK STRUCTURE STORAGE ALLOCATION

Storage binding and symbolic registers : Translates variable names into addresses and the process must occur before or during code generation

- . Each variable is assigned an address or addressing method
- . Each variable is assigned an offset with respect to base which changes with every invocation
- . Variables fall in four classes: global, global static, stack, local (non-stack) static
- The variable names have to be translated into addresses before or during code generation.

There is a base address and every name is given an offset with respect to this base which changes with every invocation. The variables can be divided into four categories**:**

a) Global Variables : fixed relocatable address or offset with respect to base as global pointer

b) Global Static Variables : .Global variables, on the other hand, have static duration (hence also called static variables): they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) Therefore they have fixed relocatable address or offset with respect to base as global pointer.

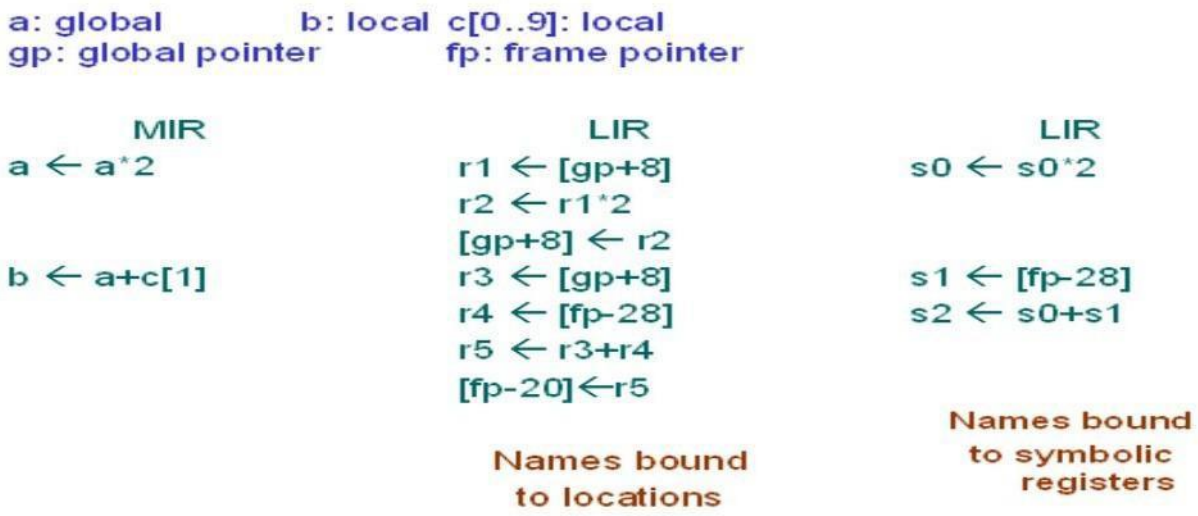
c) Stack Variables : allocate stack/global in registers and registers are not indexable, therefore, arrays cannot be in registers

. Assign symbolic registers to scalar variables

. Used for graph coloring for global register allocation

d) Stack Static Variables : By default, local variables (stack variables) (those declared within a function) have automatic duration: they spring into existence when the function is called, and they (and their values) disappear when the function returns. This is why they are stored in stacks and have offset from stack/frame pointer.

Register allocation is usually done for global variables. Since registers are not indexable, therefore, arrays cannot be in registers as they are indexed data structures. Graph coloring is a simple technique for allocating register and minimizing register spills that works well in practice.Register spills occur when a register is needed for a computation but all available registers are in use. The contents of one of the registers must be stored in memory to free it up for immediate use. We assign symbolic registers to scalar variables which are used in the graph coloring.



Local Variables in Frame

- ∑ Assign to consecutive locations; allow enough space for each
- ∑ May put word size object in half word boundaries
- ∑ Requires two half word loads
- ∑ Requires shift, or, and
- ∑ Align on double word boundaries
- ∑ Wastes space
- ∑ And Machine may allow small offsets

word boundaries - the most significant byte of the object must be located at an address whose two least significant bits are zero relative to the frame pointer

half-word boundaries - the most significant byte of the object being located at an address whose least significant bit is zero relative to the frame pointer **.**

Sort variables by the alignment they need

- Store largest variables first
- Utomatically aligns all the variables
- Does not require padding
- Store smallest variables first
- Requires more space (padding)
- For large stack frame makes more variables accessible with small offsets

While allocating memory to the variables, sort variables by the alignment they need. You may:

Store largest variables first: It automatically aligns all the variables and does not require padding since the next variable's memory allocation starts at the end of that of the earlier variable

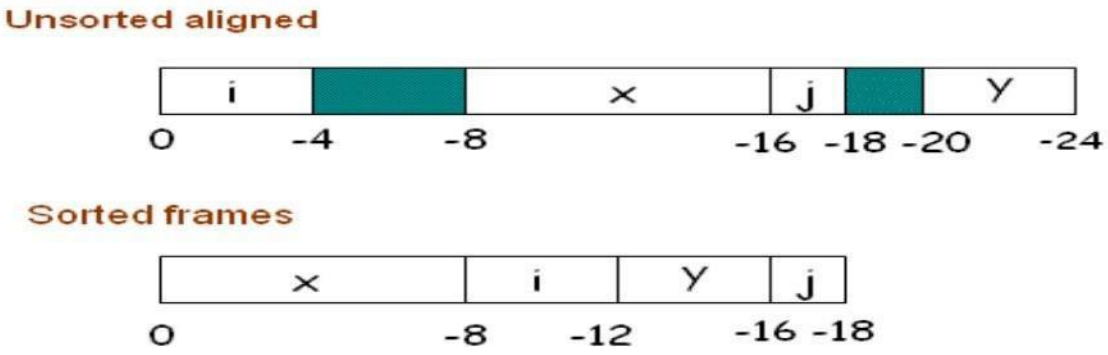
. Store smallest variables first: It requires more space (padding) since you have to accommodate for the biggest possible length of any variable data structure. The advantage is that for large stack frame, more variables become accessible within small offsets

How to store large local data structures? Because they Requires large space in local frames and therefore large offsets

- If large object is put near the boundary other objects require large offset either from fp (if put near beginning) or sp (if put near end)
- Allocate another base register to access large objects
- Allocate space in the middle or elsewhere; store pointer to these locations from at a small offset from fp
- Requires extra loads

Large local data structures require large space in local frames and therefore large offsets. As told in the previous slide's notes, if large objects are put near the boundary then the other objects require large offset. You can either allocate another base register to access large objectsor you can allocate space in the middle or elsewhere and then store pointers to these locations starting from at a small offset from the frame pointer, fp.

```
int i;
double float x;
short int j;
float y;
```



In the unsorted allocation you can see the waste of space in green. In sorted frame there is no waste of space.

STORAGE ALLOCATION FOR ARRAYS

Elements of an array are stored in a block of consecutive locations. For a single dimensional array, if low is the lower bound of the index and base is the relative address of the storage allocated to the array i.e., the relative address of A[low], then the i th Elements of an array are stored in a block of consecutive locations

For a single dimensional array, if low is the lower bound of the index and base is the relative address of the storage allocated to the array i.e., the relative address of A[low], then the i th elements begins at the location: *base + (I - low)* w* . This expression can be reorganized as*i*w + (base -low*w)* . The sub-expression *base-low*w* is calculated and stored in the symbol table at compile time when the array declaration is processed, so that the relative address of A[i] can be obtained by just adding *i*w* to it.

- Addressing Array Elements
- Arrays are stored in a block of consecutive locations
- Assume width of each element is w
- ith element of array A begins in location base + (i - low) x w where base is relative address of A[low]
- The expression is equivalent to
- i x w + (base-low x w)

⟶i x w + const

2-DIMENSIONAL ARRAY: For a row major two dimensional array the address of A[i][j] can be calculated by the formula :

base + ((i-low_i)*n2 +j - low_j)*w where low_i and low_j are lower values of I and j and n2 is number of values j can take i.e. n2 = high2 - low2 + 1.

This can again be written as :

((i * n2) + j) *w + (base - ((low_i *n2) + low_j) * w) and the second term can be calculated at compile time.

In the same manner, the expression for the location of an element in column major two-dimensional array can be obtained. This addressing can be generalized to multidimensional arrays. **S**torage can be either row major or column major approach.

Example: Let A be a 10x20 array therefore, n 1 = 10 and n 2 = 20 and assume w = 4

The Three address code to access A[y,z] is

$t_1 = y * 20$

$t_1 = t_1 + z$

$t_2 = 4 * t_1$

t_3 =A-84 {((low_1 Xn_2)+low_2)Xw)=(1*20+1)*4=84}

$t_4 = t_2 + t_3$

x = t 4

Let A be a 10x20 array

n1 = 10 and n2 = 20

Assume width of the type stored in the array is 4. The three address code to access A[y,z] is

t1 = y * 20
t1 = t1 + z
t2 = 4 *t1
t3 =base A -84 {((low 1 *n2)+low 2)*w)=(1*20+1)*4=84}
t4 =t2 +t3
x = t4

The following operations are designed :1. mktable(previous): creates a new symbol table and returns a pointer to this table. Previous is pointer to the symbol table of parent procedure.

2. entire(table,name,type,offset): creates a new entry for *name* in the symbol table pointed to by *table* .

3. addwidth(table,width): records cumulative width of entries of a table in its header.

4. enterproc(table,name ,newtable): creates an entry for procedure *name* in the symbol table pointed to by *table* . *newtable* is a pointer to symbol table for *name* .

P ⟶ {t=mktable(nil);
push(t,tblptr);
push(0,offset)}

D
{addwidth(top(tblptr),top(offset));
pop(tblptr);
pop(offset)}

D ⟶ **D ;** D

The symbol tables are created using two stacks: *tblptr* to hold pointers to symbol tables of the enclosing procedures and *offset* whose top element is the next available relative address for a local of the current procedure. Declarations in nested procedures can be processed by the syntax directed definitions given below. Note that they are basically same as those given above but we have separately dealt with the epsilon productions. Go to the next page for the explanation**.**

```
P -> MD     {     addwidth(top(tblptr) ,top(offset));
                  pop(tblptr); pop(offset);
            }
M ->        {     t= mktable(nil);
                  push(t,tblptr);
                  push(0,offset);
            }
D -> D1 ; D2
D -> proc id ; ND1 ; S        {     t = top(tblptr);
                                    addwidth(t, top(offset));
                                    pop(tblptr); pop(offset);|
                                    enterproc(top(tblptr), id.name , t)
                              }
D -> id:T                     { enter(top(tblptr), id.name, T.type , top(offset));
                              top(offset) = top(offset) + T.width
                              }
N ->                          { t = mktable ( top(tblptr));
                              push(t,tblptr); push(0,offset);
                              }
```

D → proc id;
{ t = mktable(top(tblptr));
push(t, tblptr); push(0, offset)}
D 1 ; S
{ t = top(tblptr);
addwidth(t, top(offset));
pop(tblptr); pop(offset);;
enterproc(top(tblptr), id.name, t)}
D → id: T
{ enter(top(tblptr), id.name, T.type, top(offset));
top(offset) = top (offset) + T.width }

The action for M creates a symbol table for the outermost scope and hence a nil pointer is passed in place of previous. When the declaration, D proc id ; ND1 ; S is processed, the action corresponding to N causes the creation of a symbol table for the procedure; the pointer to symbol table of enclosing procedure is given by top(tblptr). The pointer to the new table is pushed on to the stack tblptr and 0 is pushed as the initial offset on the offset stack. When the actions corresponding to the subtrees of N, D1 and S have been executed, the offset corresponding to the current procedure i.e., top(offset) contains the total width of entries in it. Hence top(offset) is added to the header of symbol table of the current procedure. The top entries of *tblptr* and *offset* are popped so that the pointer and offset of the enclosing procedure are now on top of these stacks. The entry for id is added to the symbol table of the enclosing procedure. When the declaration D -> id :T is processed entry for id is created in the symbol table of currentprocedure. Pointer to the symbol table of current procedure is again obtained from top(tblptr).

Offset corresponding to the current procedure i.e. top(offset) is incremented by the width required by type T to point to the next available location.

STORAGE ALLOCATION FOR RECORDS

Field names in records

T → record

{t = mktable(nil);

push(t, tblptr); push(0, offset)}

D end

{T.type = record(top(tblptr));

T.width = top(offset);

pop(tblptr); pop(offset)}

```
T -> record LD end      { t = mktable(nil);
                          push(t, tblptr); push(0, offset)
                          }
L ->                    { T.type = record(top(tblptr));
                          T.width = top(offset);
                          pop(tblptr); pop(offset)
                          }
```

The processing done corresponding to records is similar to that done for procedures. After the keyword record is seen the marker L creates a new symbol table. Pointer to this table and offset 0 are pushed on the respective stacks. The action for the declaration D -> id :T push the information about the field names on the table created. At the end the top of the offset stack contains the total width of the data objects within the record. This is stored in the attribute T.width. The constructor *record* is applied to the pointer to the symbol table to obtain T.type.

Names in the Symbol table :

S → id := E
{p = lookup(id.place);
if p <> nil then emit(p := E.place)
else error}
E → id
{p = lookup(id.name);
if p <> nil then E.place = p

else error}

The operation lookup in the translation scheme above checks if there is an entry for this occurrence of the name in the symbol table. If an entry is found, pointer to the entry is returned else nil is returned. Look up first checks whether the name appears in the current symbol table. If not then it looks for the name in the symbol table of the enclosing procedure and so on. The pointer to the symbol table of the enclosing procedure is obtained from the header of the symbol table.

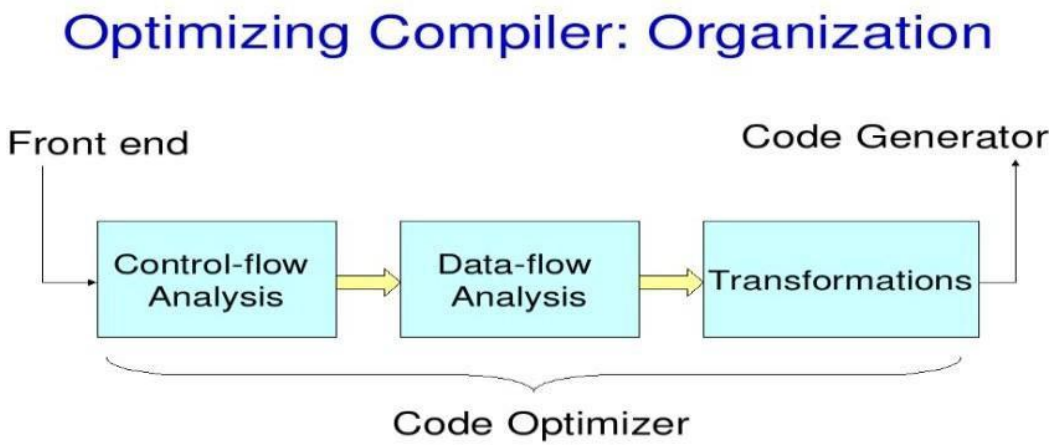
CODE OPTIMIZATION

Considerations for optimization : The code produced by the straight forward compiling algorithms can often be made to run faster or take less space,or both. This improvement is achieved by program transformations that are traditionally called optimizations. Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine. Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

Criteria for code improvement transformations

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- First, the transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error.
- Second, a transformation must, on the average, speed up programs by a measurable amount.
- Third, the transformation must be worth the effort.

Some transformations can only be applied after detailed, often time-consuming analysis of the source program, so there is little point in applying them to programs that will be run only a few times.



OBJECTIVES OF OPTIMIZATION: The main objectives of the optimization techniques are as follows

1. Exploit the fast path in case of multiple paths fro a given situation.
2. Reduce redundant instructions.
3. Produce minimum code for maximum work.
4. Trade off between the size of the code and the speed with which it gets executed.
5. Place code and data together whenever it is required to avoid unnecessary searching of data/code

During code transformation in the process of optimization, the basic requirements are as follows:

1. Retain the semantics of the source code.
2. Reduce time and/ or space.
3. Reduce the overhead involved in the optimization process.

Scope of Optimization: Control-Flow Analysis

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation. The compiler has done an enormous amount of analysis, but it still doesn‘t really know how the program does what it does. In control-flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

Control-flow analysis begins by constructing a control-flow graph, which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into basic blocks. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it.

A basic block begins in one of several ways:

• The entry point into the function

- The target of a branch (in our example, any label)
- The instruction immediately following a branch or a return

A basic block ends in any of the following ways:

- A jump statement
- A conditional or unconditional branch
- A return statement

Now we can construct the control-flow graph between the blocks. Each basic block is a node in the graph, and the possible different routes a program might take are the connections, i.e. if a block ends with a branch, there will be a path leading from that block to the branch target. The blocks that can follow a block are called its successors. There may be multiple successors or just one. Similarly the block may have many, one, or no predecessors. Connect up the flow graphfor Fibonacci basic blocks given above. What does an if then-else look like in a flow graph?What about a loop? You probably have all seen the gcc warning or javac error about: "Unreachable code at line XXX." How can the compiler tell when code is unreachable?

LOCAL OPTIMIZATIONS

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information; we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. We'll consider some of the more common local optimizations as examples.

FUNCTION PRESERVING TRANSFORMATIONS

- ∑ Common sub expression elimination
- ∑ Constant folding
- ∑ Variable propagation
- ∑ Dead Code Elimination
- ∑ Code motion
- ∑ Strength Reduction

1. Common Sub Expression Elimination:

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An

expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

Example :

```
a=b*c;
d=b*c+x-y;
```

We can eliminate the second evaluation of b*c from this code if none of the intervening statements has changed its value. We can thus rewrite the code as

```
t1=b*c;
a=t1;
d=t1+x-y;
```

Let us consider the following code

```
a=b*c;
b=x;
d=b*c+ x-y;
```

in this code, we can not eliminate the second evaluation of b*c because the value of b is changed due to the assignment b=x before it is used in calculating d.

We can say the two expressions are common if

- ∑ They lexically equivalent i.e., they consist of identical operands connected to each other by identical operator.
- ∑ They evaluate the identical values i.e., no assignment statements for any of their operands exist between the evaluations of these expressions.
- ∑ The value of any of the operands use in the expression should not be changed even due to the procedure call.

Example :

```
c=a*b;
x=a;
d=x*b;
```

We may note that even though expressions a*b and x*b are common in the above code, they can not be treated as common sub expressions.

2. Variable Propagation:

Let us consider the above code once again

```
c=a*b;
x=a;
d=x*b+4;
```

if we replace x by a in the last statement, we can identify a*b and x*b as common sub expressions. This technique is called variable propagation where the use of one variable is replaced by another variable if it has been assigned the value of same

Compile Time evaluation

The execution efficiency of the program can be improved by shifting execution time actions to compile time so that they are not performed repeatedly during the program execution. We can evaluate an expression with constants operands at compile time and replace that expression by a single value. This is called folding. Consider the following statement:

```
a= 2*(22.0/7.0)*r;
```

Here, we can perform the computation 2*(22.0/7.0) at compile time itself.

3. Dead Code Elimination:

If the value contained in the variable at a point is not used anywhere in the program subsequently, the variable is said to be dead at that place. If an assignment is made to a dead variable, then that assignment is a dead assignment and it can be safely removed from the program. Similarly, a piece of code is said to be dead, which computes value that are never used anywhere in the program.

```
c=a*b;
x=a;
d=x*b+4;
```

Using variable propagation, the code can be written as follows:

```
c=a*b;
x=a;
d=a*b+4;
```

Using Common Sub expression elimination, the code can be written as follows:

```
t1= a*b;
c=t1;
x=a;
d=t1+4;
```

Here, x=a will considered as dead code. Hence it is eliminated.

```
t1= a*b;
c=t1;
d=t1+4;
```

4. Code Movement:

The motivation for performing code movement in a program is to improve the execution time of the program by reducing the evaluation frequency of expressions. This can be done by moving the evaluation of an expression to other parts of the program. Let us consider the bellow code:

```
If(a<10)
{
b=x^2-y^2;
}
else
{
b=5;
a=( x^2-y^2)*10;
}
```

At the time of execution of the condition a<10, x^2-y^2 is evaluated twice. So, we can optimize the code by moving the out side to the block as follows:

```
t= x^2-y^2;
If(a<10)
{
b=t;
}
else
{
b=5;
a=t*10;
}
```

5. Strength Reduction:

In the frequency reduction transformation we tried to reduce the execution frequency of the expressions by moving the code. There is other class of transformations which perform equivalent actions indicated in the source program by reducing the strength of operators. By strength reduction, we mean replacing the high strength operator with low strength operator with out affecting the program meaning. Let us consider the bellow example:

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

The above can written as follows:

```
i=1;
t=4;
```

```
while (i<10)
{
y=t;
t=t+4;
}
```

Here the high strength operator * is replaced with +.

GLOBAL OPTIMIZATIONS, DATA-FLOW ANALYSIS:

So far we were only considering making changes within one basic block. With some Additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It‘s worth pointing out that global in this case does not mean across the entire program. We usually optimize only one function at a time. Inter procedural analysis is an even larger task, one not even attempted by some compilers.

The additional analysis the optimizer does to perform optimizations across basic blocks is called **data-flow analysis**. Data-flow analysis is much more complicated than control-flow analysis, and we can only scratch the surface here.

Let‘s consider a global common sub expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be **available at that point**. Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis. Each block is a node in the flow graph of a program. The **successor** set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value. An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed. Lets define such useful functions in DF analysis in following lines.

avail[B] = set of expressions available on entry to block B
exit[B] = set of expressions available on exit from B
avail[B] = ∩ **exit[x]: x** ∈ **pred[B]** (i.e. B has available the intersection of the exit of its predecessors)
killed[B] = set of the expressions killed in B
defined[B] = set of expressions defined in B
exit[B] = **avail[B] - killed[B]** + **defined[B]**

avail[B] = ∩ **(avail[x] - killed[x] + defined[x])** : **x** ∈ **pred[B]**

Here is an **Algorithm for Global Common Sub-expression Elimination**:

1) First, compute defined and killed sets for each basic block (this does not involve any of its predecessors or successors).

2) Iteratively compute the avail and exit sets for each block by running the following algorithm until you hit a stable fixed point:

a) Identify each statement **s** of the form **a = b op c** in some block B such that **b op c** is available at the entry to B and neither **b** nor **c** is redefined in B prior to **s**.

b) Follow flow of control backward in the graph passing back to but not through each block that defines **b op c**. The last computation of **b op c** in such a block reaches **s**.

c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to that block where t is a new temp.

d) Replace **s** by **a = t**.

Try an example to make things clearer:

```
main:
   BeginFunc 28;
        b = a + 2 ;
        c = 4 * b ;
        tmp1 = b < c;
        ifNZ tmp1 goto L1 ;
        b = 1 ;
        L1:
        d = a + 2 ;
EndFunc ;
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block. Then find an expression available in a block and perform step 2c above. What common sub-expression can you share between the two blocks? What if the above code were:

```
main:
   BeginFunc 28;
        b = a + 2 ;
        c = 4 * b ;
        tmp1 = b < c ;
        IfNZ tmp1 Goto L1 ;
        b = 1 ;
        z = a + 2 ; <========= an additional line here
   L1:
        d = a + 2 ;
EndFunc ;
```

MACHINE OPTIMIZATIONS

In final code generation, there is a lot of opportunity for cleverness in generatingefficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

REGISTER ALLOCATION:

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind of memory available, but as a resource, they can be scarce.

The problem is how to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches. Your Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them before performing an operation and spills them right afterwards.

A much more effective strategy would be to consider which variables are more heavily in demand and keep those in registers and spill those that are no longer needed or won't be needed until much later.

One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem. If we have 8 registers, then we try to color a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs are determined by calculating interference between the webs. A web represents a variable‘s definitions, places where it is assigned a value (as in **x = …**), and the possible different uses of those definitions (as in **y = x + 2**). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. If two portions of a variable‘s definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i = 10;
j = 20;
x = i + j;
y = j + k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of **j**, thus, **i** and **j** are "alive" at the same time. A variable is alive between the time it has been defined and that definition‘s last use, after which the variable is dead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupy the same register. Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register).

We may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimal algorithm. Here is a simplified version of something that might be used:

1. Find the node with the least neighbors. (Break ties arbitrarily.)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty.
4. Now, rebuild the graph as follows:
    a. Take the top node off the stack and reinsert it into the graph
    b. Choose a color for it based on the color of any of its neighbors presently in the graph, rotating colors in case there is more than one choice.
    c. Repeat a , and b until the graph is either completely rebuilt, or there is no color available to color the node.

If we get stuck, then the graph may not be r-colorable, we could try again with a different heuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable to memory.

INSTRUCTION SCHEDULING:

Another extremely important optimization of the final code generator is instruction scheduling. Because many machines, including most RISC architectures, have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions.

In MIPS, each instruction is issued in one cycle, but some take multiple cycles tocomplete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reach its destination, but an instruction can be placed in the "delay slot" after a branch and executed in that slack time. On the left is one arrangement of a set of instructions that requires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the second and third slots while the load completes and has a Dead cycle after the branch because the delay slot holds a noop. On the right, a more favorable rearrangement of the same instructions will execute in 5 cycles with no dead Cycles.

```
lw $t2, 4($fp)
lw $t3, 8($fp)
noop
add $t4, $t2, $t3
subi $t5, $t5, 1
goto L1
noop
```

```
lw $t2, 4($fp)
lw $t3, 8($fp)
subi $t5, $t5, 1
goto L1
add $t4, $t2, $t3
```

PEEPHOLE OPTIMIZATIONS:

Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent

code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination of load of a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations. Much of the work done in peephole optimization can be though of as find-replace activity, looking for certain idiomatic patterns in a single or sequence of two to threeInstructions than can be replaced by more efficient alternatives.

For example, MIPS has instructions that can add a small integer constant to the value in a register without loading the constant into a register first, so the sequence on the left can be replaced with that on the right:

```
li $t0, 10
lw $t1, -8($fp)
add $t2, $t1, $t0
sw $t1, -8($fp)
lw $t1, -8($fp)
addi $t2, $t1, 10
sw $t1, -8($fp)
```

What would you replace the following sequence with?

```
lw $t0, -8($fp)
sw $t0, -8($fp)
```

What about this one?

```
mul $t1, $t0, 2
```

Abstract Syntax Tree/DAG : Is nothing but the condensed form of a parse tree and is

- ∑ . Useful for representing language constructs
- ∑ . Depicts the natural hierarchical structure of the source program

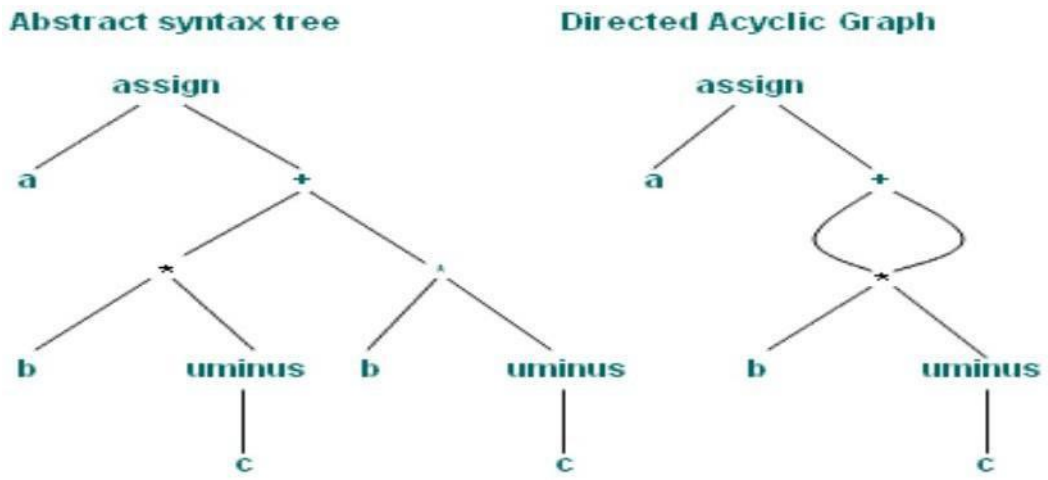
- Each internal node represents an operator
- Children of the nodes represent operands
- Leaf nodes represent operands

.DAG is more compact than abstract syntax tree because common sub expressions are eliminated A syntax tree depicts the natural hierarchical structure of a source program. Its structure has already been discussed in earlier lectures. DAGs are generated as a combination of trees:operands that are being reused are linked together, and nodes may be annotated with variable names (to denote assignments). This way, DAGs are highly compact, since they eliminate local common sub-expressions. On the other hand, they are not so easy to optimize, since they are more specific tree forms. However, it can be seen that proper building of DAG for a given

sequence of instructions can compactly represent the outcome of the calculation.

An example of a syntax tree and DAG has been given in the next slide .

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



You can see that the node " * " comes only once in the DAG as well as the leaf " b ", but the meaning conveyed by both the representations (AST as well as the DAG) remains the same.

IMPORTANT QUESTIONS:

1. What is Code optimization? Explain the objectives of it. Also discuss Function preserving transformations with your own examples?
2. Explain the following optimization techniques
   (a) Copy Propagation
   (b) Dead-Code Elimination
   (c) Code Motion
   (d) Reduction in Strength.
4. Explain the principle sources of code-improving transformations.
5. What do you mean by machine dependent and machine independent code optimization? Explain about machine dependent code optimization with examples.

ASSIGNMENT QUESTIONS:

1. Explain Local Optimization techniques with your own Examples?
2. Explain in detail the procedure that eliminating global common sub expression?
3. What is the need of code optimization? Justify your answer?

UNIT-V

CONTROL /DATA FLOW ANALYSIS:

FLOW GRAPHS :

We can add flow control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of a flow graph are the basic nodes. One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if

- There is conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 , or
- B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump. We say that B_1 is the predecessor of B_2,and B_2 is a successor of B_1.

For register and temporary allocation

- Remove variables from registers if not used
- Statement X = Y op Z defines X and uses Y and Z
- Scan each basic blocks backwards
- Assume all temporaries are dead on exit and all user variables are live on exit

The use of a name in a three-address statement is defined as follows. Suppose three- address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statementj uses the value of x computed at i.

We wish to determine for each three-address statement x := y op z, what the next uses of x, y and z are. We collect next-use information about names in basic blocks. If the name in a register is no longer needed, then the register can be assigned to some other name. This idea of keeping a name in storage only if it will be used subsequently can be applied in a number of contexts. It is used to assign space for attribute values.

The simple code generator applies it to register assignment. Our algorithm is to determine next uses makes a backward pass over each basic block, recording (in the symbol table) for each name x whether x has a next use in the block and if not, whether it is live on exit from that block. We can assume that all non-temporary variables are live on exit and all temporary variables are dead on exit.

Algorithm to compute next use information

- Suppose we are scanning i : X := Y op Z in backward scan

- Attach to i, information in symbol table about X, Y, Z
- Set X to not live and no next use in symbol table
- Set Y and Z to be live and next use in i in symbol table

As an application, we consider the assignment of storage for temporary names. Suppose we reach three-address statement i: x := y op z in our backward scan. We then do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and live ness of x, y and z.

2. In the symbol table, set x to "not live" and "no next use".

3. In the symbol table, set y and z to "live" and the next uses of y and z to i. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z.

If three-address statement i is of the form x := y or x := op y, the steps are the same as above, ignoring z. consider the below example:

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

Example :

STATEMENT

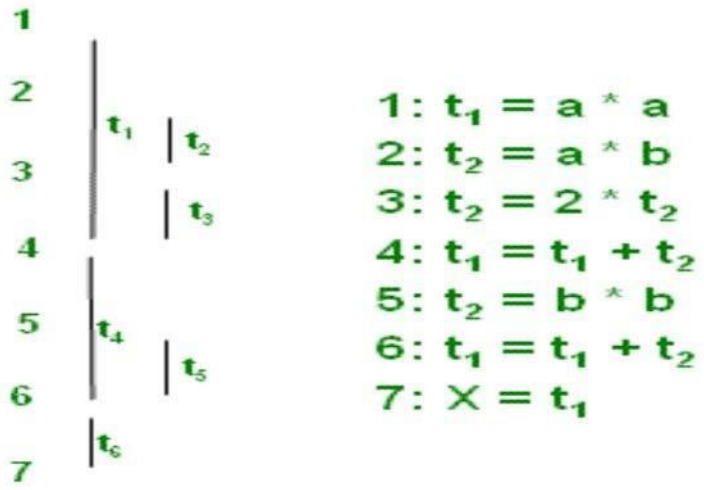
7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

We can allocate storage locations for temporaries by examining each in turn and assigning a temporary to the first location in the field for temporaries that does not contain a live temporary. If a temporary cannot be assigned to any previously created location, add a new location to the data area for the current procedure. In many cases, temporaries can be packed intoregisters rather than memory locations, as in the next section.

Example .



The six temporaries in the basic block can be packed into two locations. These locations correspond to t 1 and t 2 in:

1: t_1 = a * a ,2: t_2 = a * b,3: t_2 = 2 * t_2 ,4: t_1 = t_1 + t_2 ,5: t_2 = b * b

6: t_1 = t_1 + t_2 ,7: X = t_1

DATA FLOW EQUATIONS:

Data analysis is needed for global code optimization, e.g.: Is a variable live on exit from a block? Does a definition reach a certain point in the code? Data flow equations are used to collect dataflow information A typical dataflow equation has the form

Out[s]=Gen[s]U(in[s]-kill[s])

The notion of generation and killing depends on the dataflow analysis problem to be solved Let's first consider Reaching Definitions analysis for structured programs A definition ofa variable x is a statement that assigns or may assign a value to x An assignment to x is an unambiguous definition of x An ambiguous assignment to x can be an assignment to a pointer or a function call where x is passed by reference When x is defined, we say the definition is generated An unambiguous definition of x kills all other definitions of x When all definitions ofx are the same at a certain point, we can use this information to do some optimizations Example: all definitions of x define x to be 1. Now, by performing constant folding, we can do strength reduction if x is used in z=x*y.

GLOBAL OPTIMIZATIONS, DATA-FLOW ANALYSIS

So far we were only considering making changes within one basic block. With some additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It's worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Interprocedural analysis is an even larger task, one not even attempted by some compilers. The additional analysis the optimizer must do to perform optimizations across basic blocks is called data-flow analysis. Data-flow analysis is much more complicated than control-flow analysis.

Let's consider a global common sub-expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be available at that point.

Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis. Each block is a node in the flow graph of a program. The successor set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value. An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not
subsequently killed.

avail[B] = set of expressions available on entry to block B
exit[B] = set of expressions available on exit from B
avail[B] = ∩ **exit[x]: x** ∈ **pred[B]** (i.e. B has available the intersection of the exit of its predecessors)
killed[B] = set of the expressions killed in B
defined[B] = set of expressions defined in B
exit[B] = **avail[B] - killed[B]** + **defined[B]**
avail[B] = ∩ **(avail[x] - killed[x] + defined[x])** : **x** ∈ **pred[B]**

Here is an algorithm for global common sub-expression elimination:
1) First, compute defined and killed sets for each basic block (this does not involve any of its redecessors or successors).
2) Iteratively compute the avail and exit sets for each block by running the following algorithm until you hit a stable fixed point:

a) Identify each statement **s** of the form **a = b op c** in some block B such that **b op c** is available at the entry to B and neither **b** nor **c** is redefined in B prior to **s**.
b) Follow flow of control backward in the graph passing back to but not through each block that defines **b op c**. The last computation of **b op c** in such a block reaches **s**.
c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to that block where t is a new temp.
d) Replace **s** by **a = t**.

Lets try an example to make things clearer:
main:

```
BeginFunc 28;
b = a + 2 ;
c = 4 * b ;
tmp1 = b < c;
ifNZ tmp1 goto L1 ;
b = 1 ;
L1:
d = a + 2 ;
EndFunc ;
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block. Then find an expression available in a block and perform step 2c above. What common subexpression can you share between the two blocks? What if the above code were:

```
main:
BeginFunc 28;
b = a + 2 ;
c = 4 * b ;
tmp1 = b < c ;
IfNZ tmp1 Goto L1 ;
b = 1 ;
z = a + 2 ; <========= an additional line here
L1:
d = a + 2 ;
EndFunc ;
```

Common Sub expression Elimination

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

```
main()
{
int x, y, z;
x = (1+20)* -x;
y = x*x+(x/y);
y = z = (x/y)/(x*x);
}
straight translation:
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
```

```
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
```

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined? Here is an optimized version, after constant folding and propagation and elimination of common sub-expressions:

```
tmp2 = -x ;
x = 21 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
z = tmp5 / tmp3 ;
y = z ;
```

Induction Variable Elimination

Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant. In its simplest form, it involves determining that all of the operands in an expression are constant-valued, performing the evaluation of the expression at compile-time, and then replacing the expression by its value. If an expression such as **10 + 2 * 3** is encountered, the compiler can compute the result at compile-time (**16**) and emit code as if the input contained the result rather than the original expression. Similarly, constant conditions, suchas a conditional branch **if a < b goto L1 else goto L2** where **a** and **b** are constant can be replaced by a **Goto L1** or **Goto L2** depending on the truth of the expression evaluated at compile-time. The constant expression has to be evaluated at least once, but if the compiler does it, it meansyou don‘t have to do it again as needed during runtime. One thing to be careful about is that the compiler must obey the grammar and semantic rules from the source language that apply to expression evaluation, which may not necessarily match the language you are writing the compiler in. (For example, if you were writing an APL compiler, you would need to take carethat you were respecting its Iversonian precedence rules). It should also respect the expected treatment of any exceptional conditions (divide by zero, over/underflow). Consider the Decaf code on the far left and its un optimized TAC translation in the middle, which is then transformed by constant-folding on the far right:

```
a = 10 * 5 + 6 - b; _tmp0 = 10 ;
_tmp1 = 5 ;
_tmp2 = _tmp0 * _tmp1 ;
_tmp3 = 6 ;
_tmp4 = _tmp2 + _tmp3 ;
_tmp5 = _tmp4 – b;
a = _tmp5 ;
_tmp0 = 56 ; _tmp1 = _tmp0 – b ; a = _tmp1 ;
```

Constant-folding is what allows a language to accept constant expressions where a constant is required (such as a case label or array size) as in these C language examples:

```
int arr[20 * 4 + 3];
switch (i) {
case 10 * 5: ...
}
```

In both snippets shown above, the expression can be resolved to an integer constant at compile time and thus, we have the information needed to generate code. If either expression involved a variable, though, there would be an error. How could you rewrite the grammar to allow the grammar to do constant folding in case statements? This situation is a classic example of the gray area between syntactic and semantic analysis.

Live Variable Analysis

A variable is live at a certain point in the code if it holds a value that may be needed in the future.
Solve backwards:
Find use of a variable This variable is live between statements that have found use as next statement Recursive until you find a definition of the variable

Using the sets *use[B]*and *de f[B]*

de f[B] is the set of variables assigned values in *B* prior to any use of that variable in *B use* [*B]* is the set of variables whose values may be used in [*B]* prior to any definition of the variable.

A variable comes live into a block (in *in[B]*), if it is either used before redefinition of it is live coming out of the block and is not redefined in the block .A variable comes live out of a block (in *out[B]*) if and only if it is live coming into one of its successors

In[B]=use[B] U *(out[B]-de f[B])*

$$Out[B] = \bigcup_{S \in succ[B]} in[s]$$

Note the relation between reaching-definitions equations: the roles of *in* and *out* are interchanged

Copy Propagation

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment **a = b** in our instruction stream, we can replace later occurrences of **a** with **b** (assuming there are no changes to either variable in-between). Given the way we generate TAC code, this is a particularly valuable optimization since it is able to

eliminate a large number of instructions that only serve to copy values from one variable to another. The code on the left makes a copy of **tmp1** in **tmp2** and a copy of **tmp3** in **tmp4**. In the optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
tmp2 = tmp1 ;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3 ;
tmp5 = tmp3 * tmp2 ;
c = tmp5 + tmp4 ;
tmp3 = tmp1 * tmp1 ;
tmp5 = tmp3 * tmp1 ;
c = tmp5 + tmp3 ;
```

We can also drive this optimization "backwards", where we can recognize that the original assignment made to a temporary can be eliminated in favor of direct assignment to the final goal:

```
tmp1 = LCall _Binky ;
a = tmp1;
tmp2 = LCall _Winky ;
b = tmp2 ;
tmp3 = a * b ;
c = tmp3 ;
a = LCall _Binky;
b = LCall _Winky;
c = a * b ;
```

IMPORTANT QUESTIONS:

1. What is DAG? Explain the applications of DAG.
2. Explain briefly about code optimization and its scope in improving the code.
3. Construct the DAG for the following basic block:
   D := B*C
   E :=A+B
   B := B+C
   A := E-D.
3. Explain Detection of Loop Invariant Computation
4. Explain Code Motion.

ASSIGNMENT QUESTIONS:

1. What is loops? Explain about the following terms in loops:
   (a)Dominators
   (b) Natural loops
   (c) Inner loops
   (d) pre-headers.
2. Write short notes on Global optimization?

OBJECT CODE GENERATION

Machine dependent code optimization:

In final code generation, there is a lot of opportunity for cleverness in generating efficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

Register Allocation

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind of memory available, but as a resource, they can be scarce. The problem is how to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches. Your Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them before performing an operation and spills them right afterwards. A much more effective strategy would be to consider which variables are more heavily in demand and keep those in registers andspill those that are no longer needed or won't be needed until much later. One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem. If we have 8 registers, then we try to color a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs are determined by calculating interference between the webs. A web represents a variable‘s definitions, places where it is assigned a value (as in **x = …**), and the possible different uses of those definitions (as in **y = x + 2**). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i = 10;
j = 20;
x = i + j;
y = j + k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of **j**, thus, **i** and **j** are "alive" at the same time. A variable is alive between the time it has been defined and that definition‘s last use, after which the variable is dead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupy the same register.

Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register). You may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimal algorithm. Here is a simplified version of something that might be used:

1. Find the node with the least neighbors. (Break ties arbitrarily.)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty.
4. Now, rebuild the graph as follows:
    a. Take the top node off the stack and reinsert it into the graph
    b. Choose a color for it based on the color of any of its neighbors presently in the graph, rotating colors in case there is more than one choice.
    c. Repeat a and b until the graph is either completely rebuilt, or there is no color available to color the node.

If we get stuck, then the graph may not be r-colorable, we could try again with a different heuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable tomemory.

Instruction Scheduling:

Another extremely important optimization of the final code generator is instruction scheduling. Because many machines, including most RISC architectures, have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions. In MIPS, each instruction is issued in one cycle, but some take multiple cycles to complete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reach its destination, but an instruction can be placed in the "delay slot" after a branch and executed in that slack time. On the left is one arrangement of a set of instructions that requires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the second and third slots while the load completes and has a Dead cycle after the branch because the delay slot holds a noop. On the right, a more Favorable rearrangement of the same instructions will execute in 5 cycles with no dead Cycles.

```
lw $t2, 4($fp)
lw $t3, 8($fp)
noop
add $t4, $t2, $t3
subi $t5, $t5, 1
goto L1
noop
lw $t2, 4($fp)
lw $t3, 8($fp)
subi $t5, $t5, 1
goto L1
add $t4, $t2, $t3
```

Register Allocation

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind of memory available, but as a resource, they can be scarce. The problem is how to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches. Your Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them before performing an operation and spills them right afterwards. A much more effective strategy would be to consider which variables are more heavily in demand and keep those in registers andspill those that are no longer needed or won't be needed until much later. One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem. If we have 8 registers, then we try to color a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs are determined by calculating interference between the webs. A web represents a variable‘s definitions, places where it is assigned a value (as in **x = …**), and the possible different uses of those definitions (as in **y = x + 2**). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i = 10;
j = 20;
x = i + j;
y = j + k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of **j**, thus, **i** and **j** are "alive" at the same time. A variable is alive between the time it has been defined and that definition‘s last use, after which the variable is dead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupy the same register. Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register). You may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimal algorithm. Here is a simplified version of something that might be used:

1. Find the node with the least neighbors. (Break ties arbitrarily.)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty.
4. Now, rebuild the graph as follows:
    a. Take the top node off the stack and reinsert it into the graph

b. Choose a color for it based on the color of any of its neighbors presently in the graph, rotating colors in case there is more than one choice.

c. Repeat a and b until the graph is either completely rebuilt, or there is no color available to color the node.

If we get stuck, then the graph may not be r-colorable, we could try again with a different heuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable tomemory.

CODE GENERATION:

The code generator generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact, if possible. The code-generation uses descriptors to keep track of register contents and addresses for names.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case). As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.

2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method fora name.

CODE GENERATION ALGORITHM :

for each X = Y op Z do

- Invoke a function getreg to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

  Mov Y', L

- Generate

  op Z', L

Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.

. If current value of Y and/or Z has no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

The code generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x := y op z we perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.

2. Consult the address descriptor for u to determine y', (one of) the current location(s) of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of u is not already in L, generate the instruction MOV y', L to place a copy of y in L.

3. Generate the instruction OP z', L where z' is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

4. If the current values of y and/or y have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution ofx := y op z, those registers no longer will contain y and/or z, respectively.

FUNCTION getreg:

1. If Y is in register (that holds no other values) and Y is not live and has no next use after
   X = Y op Z
   then return register of Y for L.
2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
4. Else select memory location X as L

The function **getreg** returns the location L to hold the value of x for the assignment x := y op z.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables

simultaneously), and y is not live and has no next use after execution of x := y op z, then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor M, and return R. If R holds the value of several variables, a MOV instruction must be generated for eachvariable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

Example :

Stmt	code	reg desc	addr desc
t_1 =a-b	mov a,R_0	R_0 contains t_1	t_1 in R_0
	sub b,R_0		
t_2 =a-c	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
t_3 =t_1 +t_2	add R_1 ,R_0	R_0contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
d=t_3 +t_2	add R_1 ,R_0	R_0contains d	d in R_0
	mov R_0 ,d		d in R_0 and memory

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

t_1 = a - b

t_2 = a - c

t_3 = t_1 + t_2

d = t_3 + t_2

The code generation algorithm that we discussed would produce the code sequence as shown. Shown alongside are the values of the register and address descriptors as code generation progresses.

DAG for Register allocation:

DAG (Directed Acyclic Graphs) are useful data structures for implementing transformations on basic blocks. A DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block. Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used insidethe block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

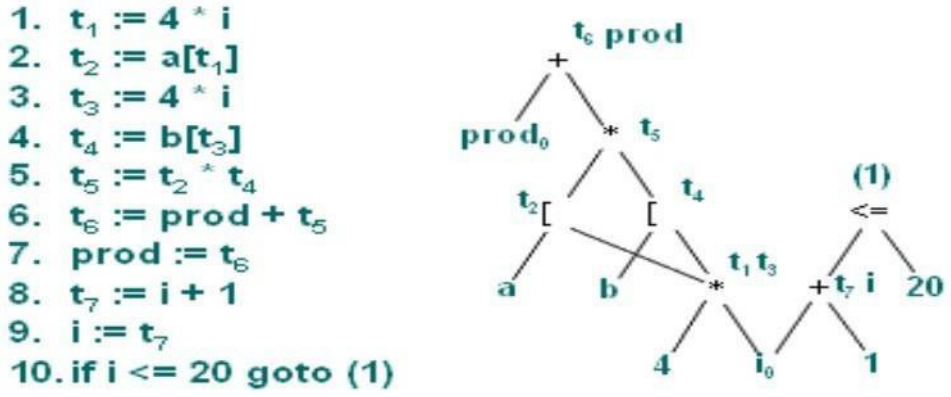
A DAG for a basic block is a directed cyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to a name we determine whether the l-value or r-value of a name is needed; most leaves represent r- values. The leaves represent initial values of names, and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.

2. Interior nodes are labeled by an operator symbol.

3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

DAG representation Example:



For example, the slide shows a three-address code. The corresponding DAG is shown. We observe that each node of the DAG represents a formula in terms of the leaves, that is, the values possessed by variables and constants upon entering the block. For example, the node labeled t 4 represents the formula

b[4 * i]

that is, the value of the word whose address is 4*i bytes offset from address b, which is the intended value of t 4 .

Code Generation from DAG

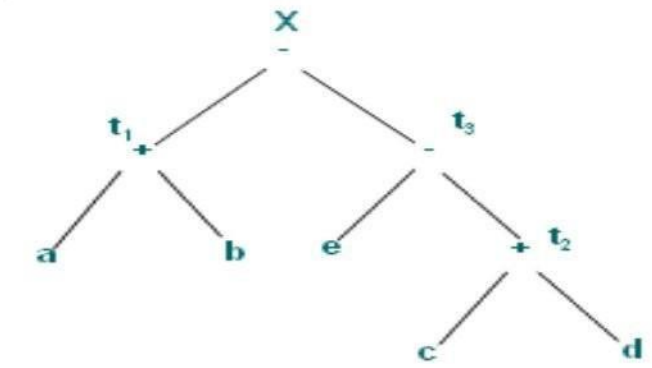
S_1= 4 * i	$S_1 = 4 * i$
S_2 = addr(A)-4	$S_2 = addr(A)-4$
$S_3 = S_2 [S_1]$	$S_3 = S_2 [S_1]$
$S_4 = 4 * i$	
S_5 = addr(B)-4	S_5= addr(B)-4
$S_6 = S_5 [S_4]$	$S_6 = S_5 [S_4]$
$S_7 = S_3 * S_6$	$S_7 = S_3 * S_6$
S_8 = prod+S_7	
prod = S_8	prod = prod + S_7
S_9 = I+1	
I = S_9	I = I + 1
If I <= 20 goto (1)	If I <= 20 goto (1)

We see how to generate code for a basic block from its DAG representation. Theadvantage of doing so is that from a DAG we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples. If the DAG is a tree, we can generate code that we can prove is optimalunder such criteria as program length or the fewest number of temporaries used. The algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.

Rearranging order of the code

Consider following basic block :

t 1 = a + b
t 2 = c + d
t 3 = e -t 2
X = t 1 -t 3



and its **DAG** given here.

Here, we briefly consider how the order in which computations are done can affect the cost of resulting object code. Consider the basic block and its corresponding DAG representationas shown in the slide.

Rearranging order .

		Rearranging the code as
Three adress code for the DAG (assuming only two registers are available)		$t_2 = c + d$
		$t_3 = e - t\ 2$
		$t_1 = a + b$
MOV a, R_0		$X = t_1 - t_3$
ADD b, R_0		gives
MOV c, R_1		MOV c, R_0
ADD d, R_1		ADD d, R_0
MOV R_0 , t_1	Register spilling	MOV e, R_1
MOV e, R_0		SUB R_0 , R_1
SUB R_1 , R_0		MOV a, R_0
MOV t_1 , R_1	Register reloading	ADD b, R_0
SUB R_0 , R_1		SUB R_1 , R_0
MOV R_1 , X		MOV R_1 , X

If we generate code for the three-address statements using the code generation algorithm described before, we get the code sequence as shown (assuming two registers R0 and R1 are available, and only X is live on exit). On the other hand suppose we rearranged the order of the statements so that the computation of t 1 occurs immediately before that of X as:

$t_2 = c + d$
$t_3 = e - t\ 2$
$t_1 = a + b$
$X = t_1 - t_3$

Then, using the code generation algorithm, we get the new code sequence as shown (again only R0 and R1 are available). By performing the computation in this order, we have been able to save two instructions; MOV R0, t 1 (which stores the value of R0 in memory location t 1) and MOV t 1 , R1 (which reloads the value of t 1 in the register R1).

IMPORTANT & EXPECTED QUESTIONS:

Construct the DAG for the following basic block:
D := B*C
E :=A+B
B := B+C
A := E-D.

1. What is Object code? Explain about the following object code forms:
   (a) Absolute machine-language
   (b) Relocatable machine-language
   (c) Assembly-language.
2. Explain about Generic code generation algorithm?
3. Write and explain about object code forms?
4. Explain Peephole Optimization

ASSIGNMENT QUESTIONS:

1. Explain about Generic code generation algorithm?
2. Explain about Data-Flow analysis of structured flow graphs.
3. What is DAG? Explain the applications of DAG.