COMPILER DESIGN (S.Code: P16CS32)

COMPILER DESIGN

Material Prepared by

Dr.S.Malathi,

Assistant Professor in Computer Science,

Swami Dayananda College of Arts & Science -Manjakkudi

CORE COURSE VIII

COMPILER DESIGN

UNIT I

Introduction to compilers – Analysis of source program – Phase of compiler – Cousins of compilers – Grouping of phases – Simple one pass compiler: overview – Syntax definition Lexical analysis: removal of white space and comments – Constants – Recognizing identifiers and keywords – Lexical analysis – Role of a lexical analyzer – Input buffering –Specification of tokens – Recognition tokens.

UNIT II

Symbol tables: Symbol table entries – List data structures for symbol table – - Hash tables – Representation of scope information – Syntax Analysis: Role of parser – Context free grammar – Writing a grammar – Top down parsing – Simple bottom up parsing – Shift reducing parsing.

UNIT III

Syntax directed definition: Construction of syntax trees – Bottom up evaluation of S-Attributed definition – L-Attributed definitions – Top down translation - Type checking: Type systems – Specifications of simple type checker.

UNIT IV

Run-time environment: Source language issues – Storage organizations – Storage allocation strategies - Intermediate code generation: Intermediate languages – Declarations – Assignment statements.

UNIT V

Code generation: Issue in design of code generator – The target machine – Runtime storage management – Basic clocks and flow graphs – Code optimization: Introduction – Principle source of code optimization – Optimization of basic blocks

Text Books:

1. AHO, ULLMAN, "COMPILERS, PRINCIPLES AND TECHNIQUES AND TOOLS", PEARSON EDUCATION – 2001 6TH EDITION.

UNIT – I

Introduction to compilers:

Compiler Design is the structure and set of principles that guide the translation, analysis, and optimization process of a compiler.

A Compiler is computer software that transforms program source code which is written in a high-level language into low-level machine code. It essentially translates the code written in one programming language to another language without changing the logic of the code.

The Compiler also makes the code output efficient and optimized for execution time and memory space. The compiling process has basic translation mechanisms and error detection; it can't compile code if there is an error. The compiler process runs through syntax, lexical, and semantic analysis in the front end and generates optimized code in the back end.

When executing, the compiler first analyzes the entire language statements one after the other syntactically and then, if it's successful, builds the output code, making sure that statements that refer to other statements are referred to appropriately, traditionally; the output code is called **Object Code**.

Types of Compiler

1. **Cross Compiler:** This enables the creation of code for a platform other than the one on which the compiler is running. For instance, it runs on a machine 'A' and produces code for another machine 'B'.

2. **Source-to-source Compiler:** This can be referred to as a transcompiler or transpiler and it is a compiler that translates source code written in one programming language into source code of another programming language.

3. **Single Pass Compiler:** This directly transforms source code into machine code. For instance, Pascal programming language.

4. **Two-Pass Compiler:** This goes through the code to be translated twice; on the first pass it checks the syntax of statements and constructs a table of symbols, while on the second pass it actually translates program statements into machine language.

5. **Multi-Pass Compiler:** This is a type of compiler that processes the source code or abstract syntax tree of a program multiple times before translating it to machine language.

Language Processing Systems Steps

1. **High-Level Language:** These are programs that contain #define or #include directives such as #include or #define.

2. They are closer to human's language but far from machines. The (#) tags are referred to as preprocessor directives. They tell the pre-processor about what to do.

3. **Pre-Processor:** This produces input for the compiler and also deals with file inclusion, augmentation, macro-processing, language extension, etc. It removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion.

4. **Assembler:** This translates assembly language code into machine understandable language. Each platform (OS + Hardware) has its own assembler. The output of an assembler is known as an object file which is a combination of machine instruction along with the data required to store these instructions in memory.

5. **Interpreter:** An interpreter converts high-level language into low-level machine language almost similar to what Compiler does. The major difference between both is that the interpreter reads and transforms code line by line while Compiler reads the entire code at once and outputs the machine code directly. Another difference is, Interpreted programs are usually slower with respect to compiled ones.

6. **Reloadable Machine Code:** This can be loaded at any point in time and can be run. This enables the movement of a program using its unique address identifier.

7. **Linker:** It links and merges a variety of object files into a single file to make it executable. The linker searches for defined modules in a program and finds out the memory location where all modules are stored.

8. **Loader:** It loads the output from the Linker in memory and executes it. It basically loads executable files into memory and runs them

Features of a Compiler

Correctness: A major feature of a compiler is its correctness, and accuracy to compile the given code input into its exact logic in the output object code due to its being

developed using rigorous testing techniques (often called compiler validation) on an existing compiler.

Recognize legal and illegal program constructs: Compilers are designed in such a way that they can identify which part of the program formed from one or more lexical tokens using the appropriate rules of the language is syntactically allowable and which is not.

Good Error reporting/handling: A compiler is designed to know how to parse the error encountered from lack be it a syntactical error, insufficient memory errors, or logic errors are meticulously handled and displayed to the user.

The Speed of the target code: Compilers make sure that the target code is fast because in huge size code its a serious limitation if the code is slow, some compilers do so by translating the byte code into target code to run in the specific processor using classical compiling methods.

Preserve the correct meaning of the code: A compiler makes sure that the code logic is preserved to the tiniest detail because a single loss in the code logic can change the whole code logic and output the wrong result, so during the design process, the compiler goes through a whole lot of testing to make sure that no code logic is lost during the compiling process.

Code debugging help: Compilers make help the debugging process easier by pointing out the error line to the programmer and telling them the type of error that is encountered so they would know how to start fixing it.

Benefits of Using a Compiler:

Improved performance: Using a compiler increases your program performance, by making the program optimized, portable, and easily run on the specific hardware.

Reduced system load: Compilers make your program run faster than interpreted programs because it compiles the program only once, hence reducing system load and response time when next you run the program.

Protection for source code and programs: Compilers protect your program source by discouraging other users from making unauthorized changes to your programs, you as the author can distribute your programs in object code.

Portability of compiled programs: Compiled programs are always portable meaning that you can transfer it from one machine to another without worrying about dependencies as it is all compiled together.

Analysis of source program:

In Compiling, analysis consists of three phases:

Linear Analysis: In which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning.

Hierarchical Analysis: In which characters or tokens are grouped hierarchically in to nested collections with collective meaning.

Semantic Analysis: In which certain checks are performed to ensure that the components of a program fit together meaningfully.

Phases of compiler

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.
There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code. The phases of a compiler are:

Lexical analysis

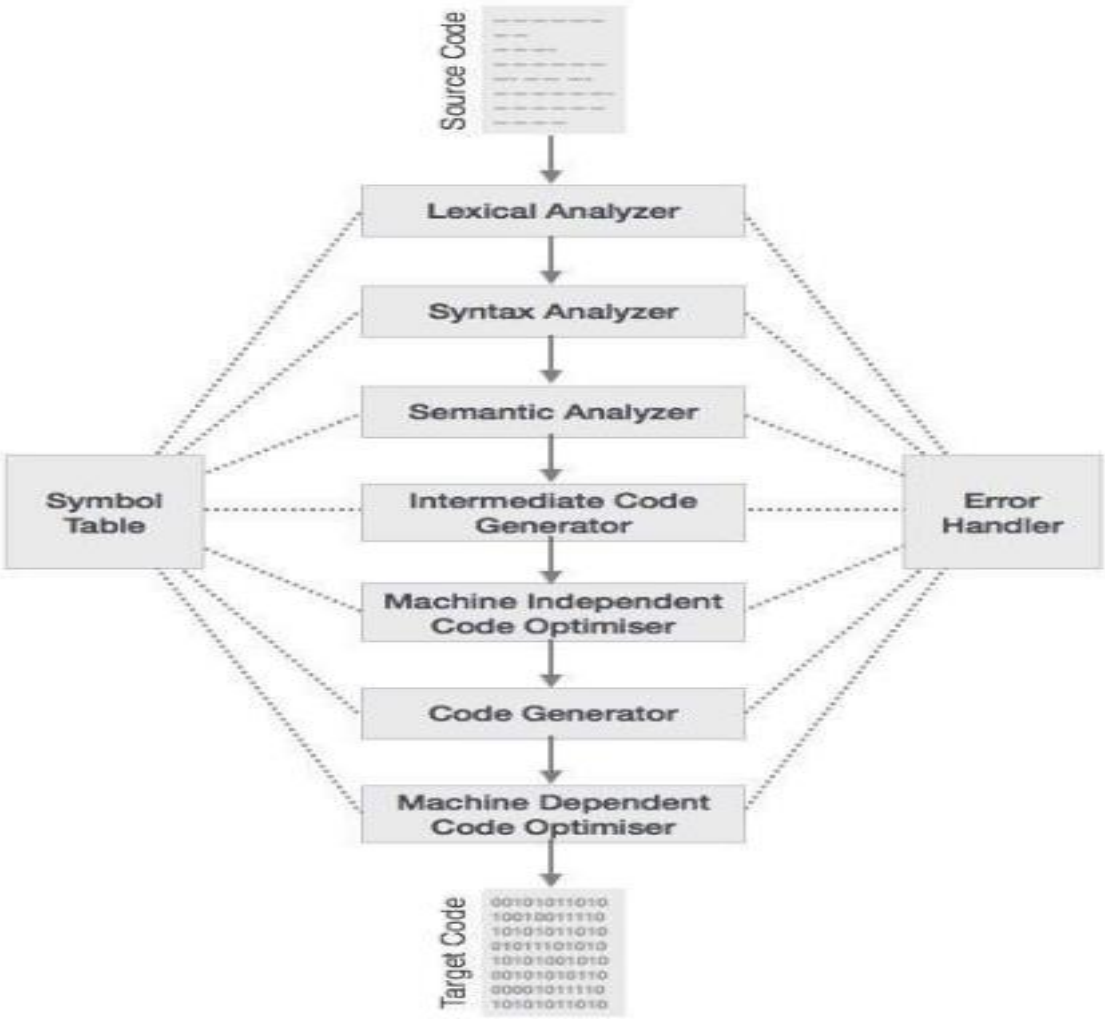
Syntax analysis

Semantic analysis

Intermediate code generator

Code optimizer

Code generator



Phase 1: Lexical Analysis

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.

Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Example:

x = y + 10

Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

Phase 2: Syntax Analysis

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

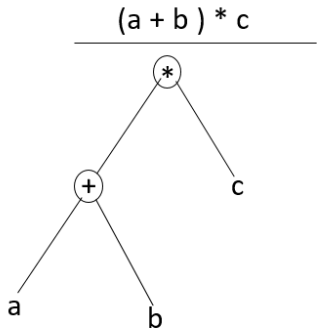
List of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

Example

Any identifier/number is an expression
If x is an identifier and y+10 is an expression, then x= y+10 is a statement.
Consider parse tree for the following example

(a+b)*c



In Parse Tree

- Interior node: record with an operator filed and two files for children
- Leaf: records with 2/more fields; one for token and other information about the token
- Ensure that the components of the program fit together meaningfully
- Gathers type information and checks for type compatibility
- Checks operands are permitted by the source language

Phase 3: Semantic Analysis

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.
Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.

Functions of Semantic analyses phase are:

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

Example

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication.

Phase 4: Intermediate Code Generation

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine. Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction

Example

For example,

total = count + rate * 5
Intermediate code with the help of address code method is:

t1 := int (5)
t2 := rate * t1
t3 := count + t2
total := t3

Phase 5: Code Optimization

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

The primary functions of this phase are:

- It helps you to establish a trade-off between execution and compilation speed
- Improves the running time of the target program
- Generates streamlined code still in intermediate representation
- Removing unreachable code and getting rid of unused variables

- Removing statements which are not altered from the loop

Example:
Consider the following code

a = intofloat(10)
b = c * a
d = e + b
f = d
Can become

b =c * 10.0
f = e+b

Phase 6: Code Generation

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate reloadable machine code. It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language.

The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

Example:

a = b + 60.0

Would be possibly translated to registers.

MOVF a, R1
MULF #60.0, R2
ADDF R1, R2

Symbol Table Management

A symbol table contains a record for each identifier with fields for the attributes of the identifier. This component makes it easier for the compiler to search the identifier record and retrieve it quickly. The symbol table also helps you for the scope management. The symbol table and error handler interact with all the phases and symbol table update correspondingly.

Error Handling Routine:

In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: When the memory is full or proper registers are not allocated
- Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis.
The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process. These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

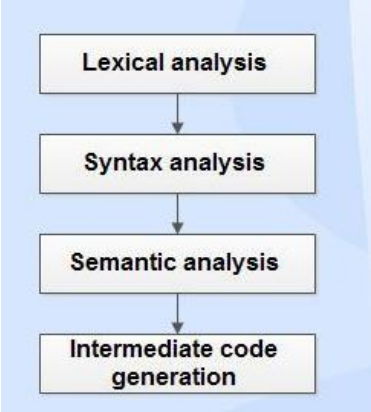
GROUPING OF PHASES

The phases of a compiler can be grouped as Front end and Back end.

Front end comprises of phases which are dependent on the input (source language) and independent on the target machine (target language). It includes lexical and syntactic analysis, symbol table management, semantic analysis and the generation of intermediate code. Code optimization can also be done by the front end. • It also includes error handling at the phases concerned.

Front end of a compiler consists of the phases

• Lexical analysis.
• Syntax analysis.
• Semantic analysis.
• Intermediate code generation.

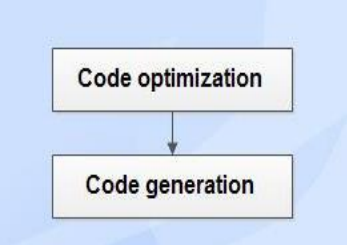


Back end

Back end comprises of those phases of the compiler that are dependent on the target machine and independent on the source language. This includes code optimization, code generation. In addition to this, it also encompasses error handling and symbol table management operations.

Back end of a compiler contains

• Code optimization.
• Code generation.



Passes

• The phases of compiler can be implemented in a single pass by marking the primary actions viz. reading of input file and writing to the output file.

• Several phases of compiler are grouped into one pass in such a way that the operations in each and every phase are incorporated during the pass.

• (eg.) Lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.

Reducing the Number of Passes

• Minimizing the number of passes improves the time efficiency as reading from and writing to intermediate files can be reduced.

• When grouping phases into one pass, the entire program has to be kept in memory to

ensure proper information flow to each phase because one phase may need information in a different order than the information produced in previous phase. The source program or target program differs from its internal representation. So, the memory for internal form may be larger than that of input and output.

COUSINS OF COMPILER

Cousins of compiler contains

1. Preprocessor
2. Compiler
3. Assembler
4. Linker
5. Loader
6. Memory

1) Preprocessor

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers. They may perform the following functions.

1. Macro processing
2. File Inclusion
3. Rational Preprocessors
4. Language extension

1. Macro processing: A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

2. File Inclusion: Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

3. Rational Preprocessors: These processors change older languages with more modern flow-of-control and data-structuring facilities.

4. Language extension: These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

2) Compiler

It takes pure high level language as a input and convert into assembly code.

3) Assembler

It takes assembly code as an input and converts it into assembly code. Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers. **One-pass assemblers** go through the source code once and assume that all symbols will be defined before any instruction that references them. **Two-pass assemblers** create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code.

4) Linker

It has the following functions

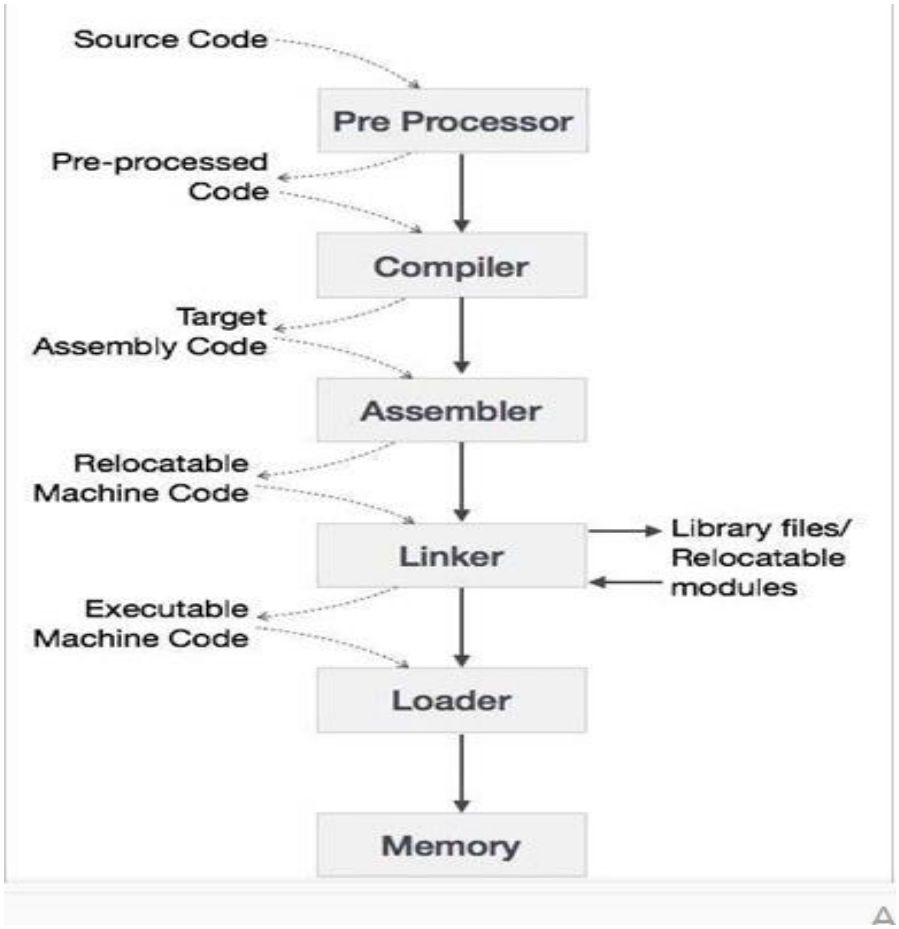
1. **Allocation:** It means get the memory portions from operating system and storing the object data.
2. **Relocation:** It maps the relative address to the physical address and relocating the object code.
3. **Linker:** It combines the entire executable object module to pre single executable file.

5) Loader

A loader is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

6) Memory

The output of an assembler is known as an object file which is a combination of machine instruction along with the data required to store these instructions in memory.



Simple one pass compiler: overview – Syntax definition

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

Overview Syntax Definition

Language Definition

•Appearance of programming language

Vocabulary : Regular expression

Syntax : Backus-Naur Form(BNF) or Context Free Form(CFG)

• Semantics : Informal language or some examples

Syntax definition

To specify the syntax of a language : CFG and BNF

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

An alphabet of a language is a set of symbols.

- Examples: {0,1} for a binary number system(language)={0,1,100,101,...}• {a,b,c} for language={a,b,c, ac,abcc..}
- {if,(,),else ...} for a if statements={if(a==1)goto10, if--}

A string over an alphabet

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

Language

It is a subset of all the strings over a given alphabet.

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

Example :Grammar for expressions consisting of digits and plus and minus signs.

- Language of expressions L={9-5+2, 3-1, ...}
- The productions of grammar for this language L are

list → list + digit

list → list – digit

list → digit

digit → 0|1|2|3|4|5|6|7|8|9

list, digit : Grammar variables, Grammar symbols.

0,1,2,3,4,5,6,7,8,9,-,+ : Tokens, Terminal symbols

Convention specifying grammar

- Terminal symbols : bold face string if, num, id
- Nonterminal symbol, grammar symbol : italicized names, list, digit ,A,B•

Grammar G=(N,T,P,S)

- N : a set of nonterminal symbols
- T : a set of terminal symbols, tokens
- P : a set of production rules
- S : a start symbol, S ∈ N

Grammar G for a language L = { 9-5+2, 3-1,….}

- G=(N,T,P,S)
- N={list,digit}
- T={0,1,2,3,4,5,6,7,8,9,-,+}
- P= list → list + digit
- list → list - digit
- list → digit
- digit → 0|1|2|3|4|5|6|7|8|9

Some definitions for a language L and its grammar G

Derivation :

A sequence of replacements S⇒α1⇒α2⇒…⇒αn is a derivation of αn. Example, A derivation 1+9 from the grammar G

• left most derivation list⇒list + digit ⇒digit + digit ⇒1 + digit ⇒1 + 9

• right most derivation list⇒list + digit ⇒list + 9 ⇒digit + 9 ⇒1 + 9

Language of grammar L(G)

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

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

Parse Tree

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

- The root is labeled by the start symbol.
- Each leaf is labeled by a token orε.

- Each interior none is labeled by a nonterminal symbol.
- When a production A→x1… xn is derived, nodes labeled by x1… xn are made aschildren nodes of node labeled by A
- nodes of node labeled by A.

  • root : the start symbol

  • internal nodes : nonterminal

  • leaf nodes : terminal

  Example : list -> list + digit | list - digit | digit

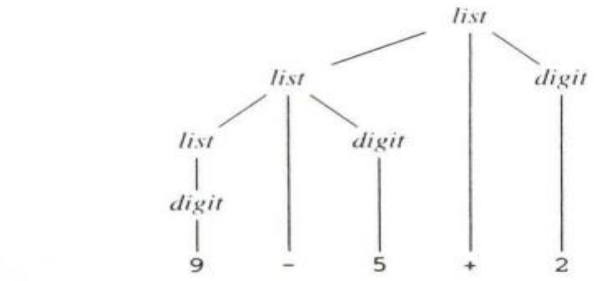
  digit -> 0|1|2|3|4|5|6|7|8|9

left most derivation for 9-5+2,

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

right most derivation for 9-5+2,

list⇒list+digit⇒list+2⇒list-digit+2 ⇒list-5+2⇒digit-5+2 ⇒9-5+2 parse tree for 9-5+2



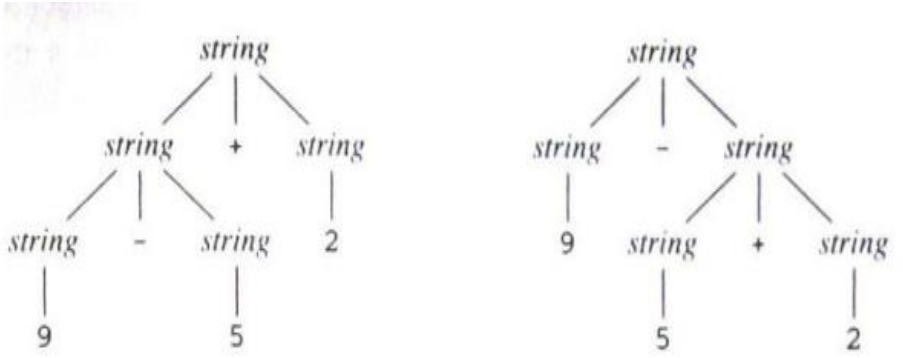
Parse tree for 9-5+2 according to the grammar in Example

Ambiguity

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

Example . Suppose a grammar G that cannot distinguish between lists and ϖ digits as in above example

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



Two Parse tree for 9-5+2

Associativity of operator

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

Example: 9+5+2≡(9+5)+2, a=b=c≡a=(b=c)

Left Associative Grammar:

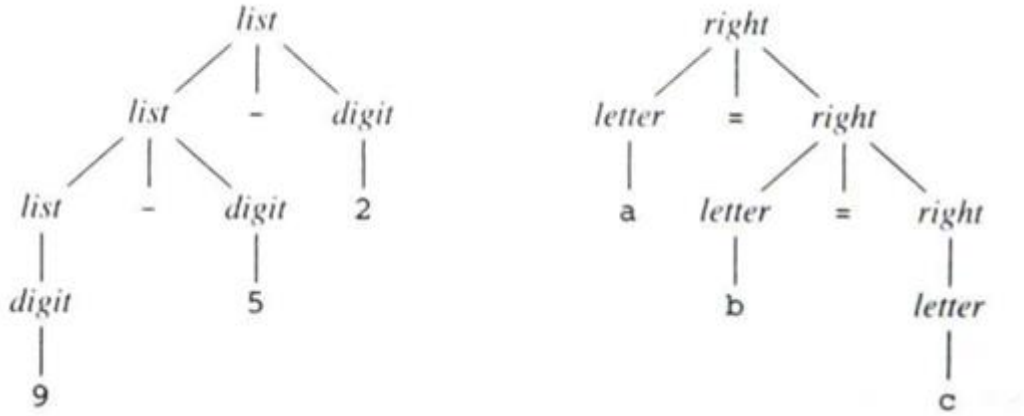
list → list + digit | list – digit

digit →0|1|…|9

Right Associative Grammar :

right → letter = right | letter

letter → a|b|…|z



Parse tree left- and right-associative operators

Precedence of operators

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

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

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

expr → expr + term | expr - term | term

term → term * factor | term / factor | factor

factor → digit | (expr)

digit → 0 | 1 | … | 9

Syntax of statements

stmt → id = expr ;

| if (expr) stmt ;

| if (expr) stmt else stmt ;

| while (expr) stmt ;

expr → expr + term | expr - term | term

term → term * factor | term / factor | factor

factor → digit | (expr)

digit → 0 | 1 | … | 9

LEXICAL ANALYSIS

Lexical Analysis:

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

Removal of White Space and Comments

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

Contsants

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

  output: <num, 31><+,

  ><num, 28>num + :token

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

Recognizing

- Identifiers
  - Identifiers are names of variables, arrays, functions...
  - A grammar treats an identifier as a token.
  - eg) input : count = count + increment; output : <id,1><=, ><id,1><+, ><id, 2>;

Symbol table

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

- Keywords are reserved, i.e., they cannot be used as identifiers. Then a character string forms an identifier only if it is not a keyword.
- punctuation symbols

  operators : + - * / := <> …

Role of a lexical analyzer

Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.
Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the

token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

Example

How Pleasant Is The Weather?
See this Lexical Analysis example; Here, we can easily recognize that there are five words How Pleasant, The, Weather, Is. This is very natural for us as we can recognize the separators, blanks, and the punctuation symbol.

HowPl easantIs Th ewe ather?
Now, check this example, we can also read this. However, it will take some time because separators are put in the Odd Places. It is not something which comes to you immediately.

Basic Terminologies

What's a lexeme?

A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

What's a token?

Tokens in compiler design are the sequence of characters which represents a unit of information in the source program.

What is Pattern?

A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

Lexical Analyzer Architecture: How tokens are recognized

The main task of lexical analysis is to read input characters in the code and produce tokens.

Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how recognition of tokens in compiler design works-

1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

Roles of the Lexical analyzer

Lexical analyzer performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

Example of Lexical Analysis, Tokens, Non-Tokens

Consider the following code that is fed to Lexical Analyzer

```
#include <stdio.h>
   int maximum(int x, int y) {
      // This will compare 2 numbers
      if (x > y)
         return x;
      else {
         return y;
      }
   }
```

Examples of Tokens created

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

Examples of Non tokens

Type	Examples

Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

Lexical Errors

A character sequence which is not possible to scan into any valid token is a lexical error. Important facts about the lexical error:

- Lexical errors are not very common, but it should be managed by a scanner
- Misspelling of identifiers, operators, keyword are considered as lexical errors
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery in Lexical Analyzer

Here, are a few most common error recovery techniques:

- Removes one character from the remaining input
- In the panic mode, the successive characters are always ignored until we reach a well-formed token
- By inserting the missing character into the remaining input
- Replace a character with another character
- Transpose two serial characters

Lexical Analyzer vs. Parser

Lexical Analyser	Parser
Scan Input program	Perform syntax analysis
Identify Tokens	Create an abstract representation of the code
Insert tokens into Symbol Table	Update symbol table entries
It generates lexical errors	It generates a parse tree of the source code

Why separate Lexical and Parser?

- The simplicity of design: It eases the process of lexical analysis and the syntax analysis by eliminating unwanted tokens
- To improve compiler efficiency: Helps you to improve compiler efficiency

- Specialization: specialized techniques can be applied to improves the lexical analysis process
- Portability: only the scanner requires to communicate with the outside world
- Higher portability: input-device-specific peculiarities restricted to the lexer

Advantages of Lexical analysis

- Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer's code to create a compiled binary executable code
- It is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
- A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task

Disadvantage of Lexical analysis

- You need to spend significant time reading the source program and partitioning it in the form of tokens
- Some regular expressions are quite difficult to understand compared to PEG or EBNF rules
- More effort is needed to develop and debug the lexer and its token descriptions
- Additional runtime overhead is required to generate the lexer tables and construct the tokens

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string. Component of regularexpression..

X - the character x

. any character, usually accept a new line[x y z] any of the characters x, y, z,

R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R1|R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters

or digits.In regular expression notation we would write.
Identifier = letter (letter | digit)*

re are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each ‘a’ in Σ, is a regular expression denoting { a }, the language with only one string
  consisting of the single symbol ‘a’ .
- If R and S are regular expressions, then

(R) | (S) means L(r) U L(s)
R.S means L(r).L(s)R* denotes L(r*)

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to defineregular expressions using these names as if they were symbols.
Identifiers are the set or string of letters and digits beginning with a letter. The following regulardefinition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))Pascal identifier
Letter - A | B | ……| Z | a | b |……| z|Digits - 0 | 1 | 2 | …. | 9
Id - letter (letter / digit)*

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string andfinds a prefix that is a lexeme matching one of the patterns.
Stmt →if expr then stmt
| If expr then else stmt
| ϵ
Expr →term relop term
| termTerm →id
|number
For relop ,we use the comparison operations of languages like Pascal or SQL where = is “equals”and < > is “not equals” because it presents an interesting structure of lexemes.
The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are

described using regular definitions.
digit → [0,9] digits →digit+
number →digit(.digit)?(e.[+-]?digits)?letter → [A-Z,a-z]
id →letter(letter/digit)*if → if
then →then
else →else
relop →< | > |<= | >= | = = | < >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:
WS → (blank/tab/newline)+
Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	**Token Name**	**Attribute Value**
Any WS	-	-
if	if	-
then	then	-
else	else	-
Any id	Id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
==	relop	EQ
<>	relop	NE

TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transitiondiagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shalladditionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.

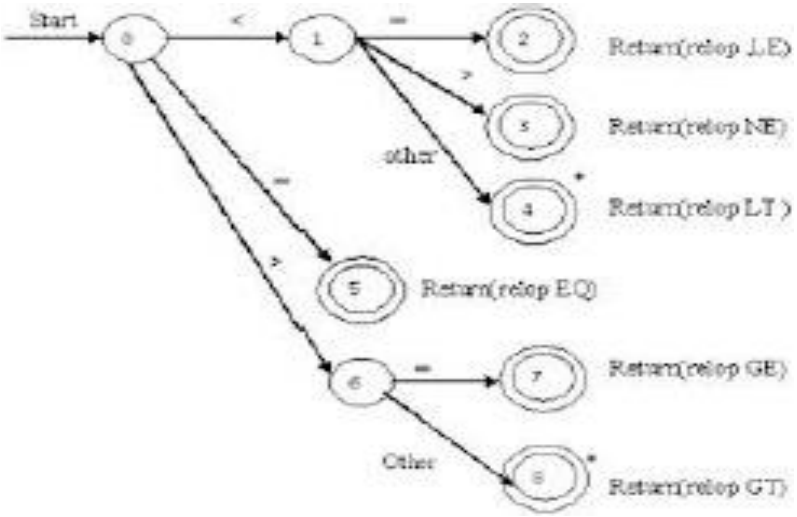


Fig. 3.3: Transition diagram of Relational operators

As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.

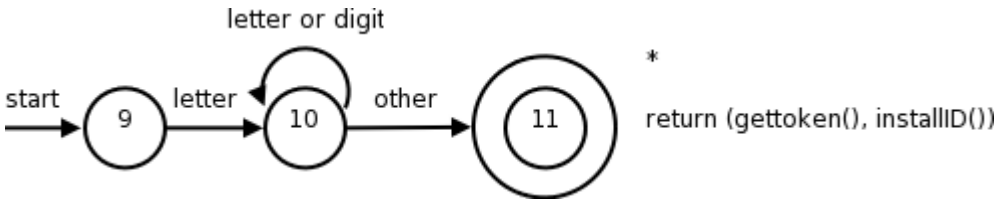


Fig. 3.4: Transition diagram of Identifier

The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specifiedby the diagrams. Each state gets a segment of code.

FINITE AUTOMATON

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is asentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexicalanalyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?

– deterministic – faster recognizer, but it may take more space
– non-deterministic – slower, but it may take less space
– Deterministic automatons are widely used lexical analyzers.

- First, we define regular expressions for tokens; Then we convert them into a DFA to get alexical analyzer for our tokens.

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:

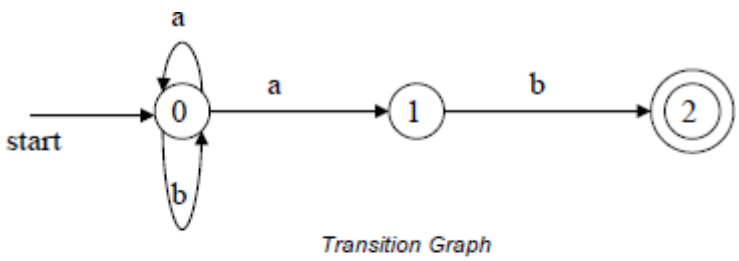
○ S - a set of states
○ Σ - a set of input symbols (alphabet)

omove - a transition function move to map state-symbol pairs to sets of states.
os0 - a start (initial) state

○ F- a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to

one ofaccepting states such that edge labels along this path spell out x.
Example:



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

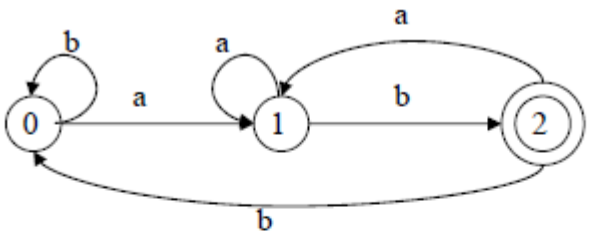
The language recognized by this NFA is (a|b)*ab

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has ε- transition
- For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transitionfunction is from pair of state-symbol to state (not set of states)

Example:

The DFA to recognize the language (a|b)* ab is as follows.



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

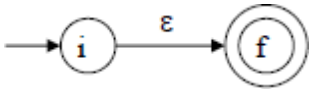
Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

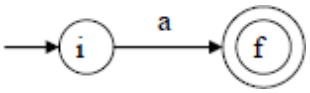
Note that the entries in this function are single value and not set of values (unlike NFA).

Converting RE to NFA

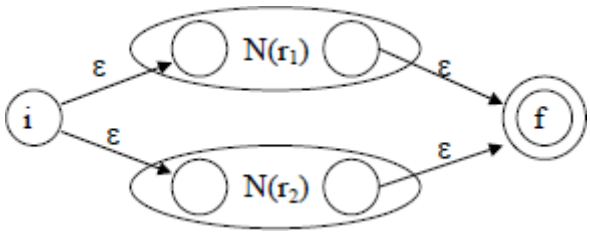
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions arecombined to create its NFA.
- To recognize an empty string ε:



- To recognize a symbol a in the alphabet Σ:



- For regular expression r1 | r2:

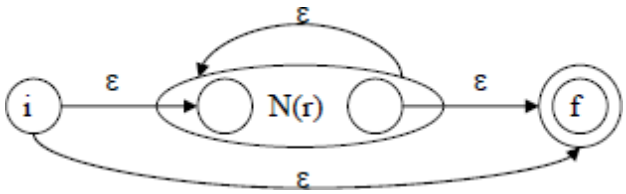


N(r1) and N(r2) are NFAs for regular expressions r1 and r2.

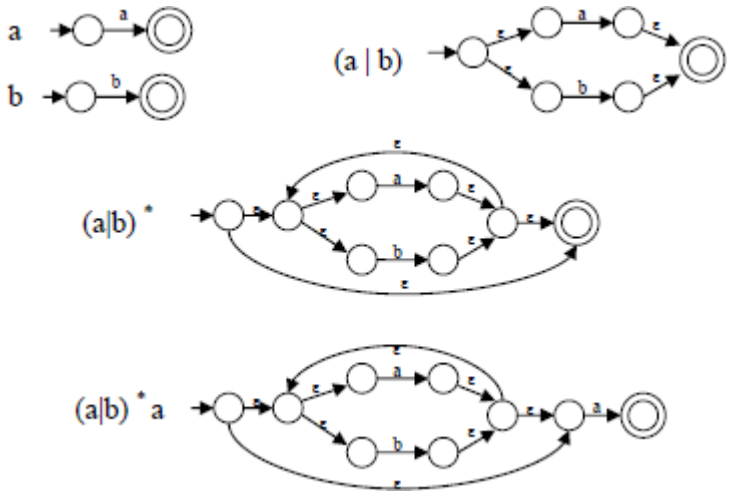
- For regular expression r1 r2

Here, final state of N(r1) becomes the final state of N(r1r2).

- For regular expression r*



Example:
For a RE (a|b) * a, the NFA construction is shown below.



Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an – transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an -transition will be represented bythe same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard

a transition on a symbol as moving from a state to a set of states (ie. the union of all

those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.
To perform this operation, let us define two functions:

- The **-closure** function takes a state and returns the set of states reachable from it based on (one or more) -transitions. Note that this will always include the state itself. We should beable to get from a state to any state in its -closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable byone transition on this character.

Wecan generalise both these functions to apply to sets of states by taking the union of the application to individual states.

For Example, if A, B and C are states, move({A,B,C},`a') = move(A,`a') move(B,`a') move(C,`a').
The Subset Construction Algorithm is a follows:

put ε-closure({s0}) as an unmarked state into the set of DFA (DS) while (there is one unmarked S1 in DS) do
begin
mark S1
for each input symbol a dobegin

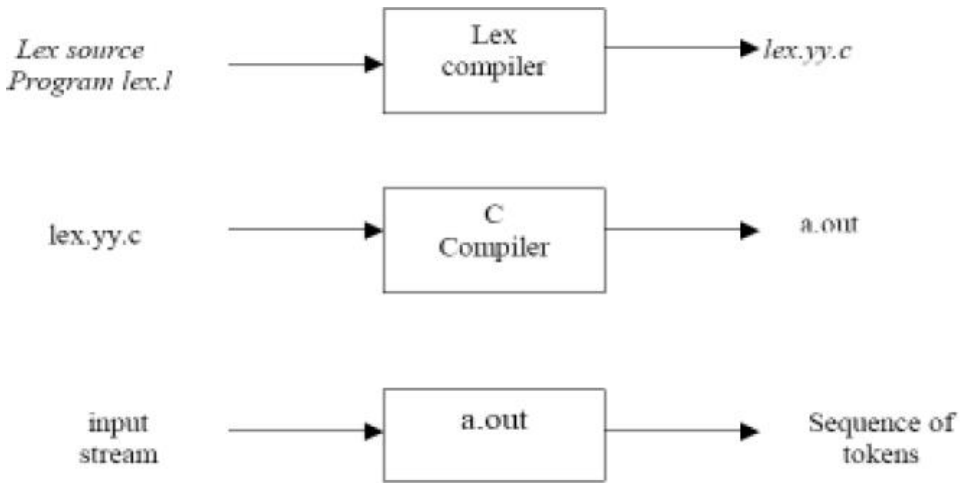
end

end

S2 ← ε-closure(move(S1,a))if (S2 is not in DS) then
add S2 into DS as an unmarked statetransfunc[S1,a] ← S2

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure({s0})

Lexical Analyzer Generator



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

p1 {action 1}

p2 {action 2}

p3 {action 3}

… …

… …

Where, each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are

needed by the *actions*.Alternatively these procedures can be compiled separately and loaded with thelexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book:

Compilers: Principles, Techniques, and Tools by Aho, Sethi & Ullman for more clarity.

Input buffering

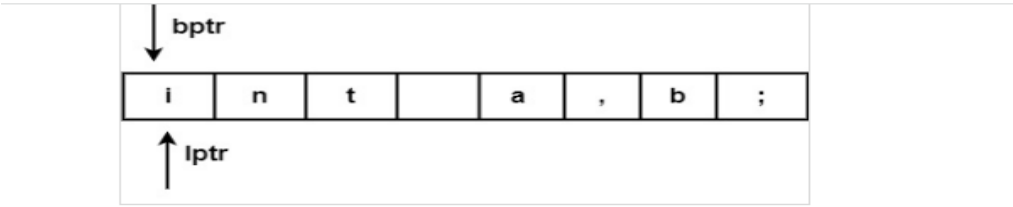
Lexical Analysis has to access secondary memory each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis.

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

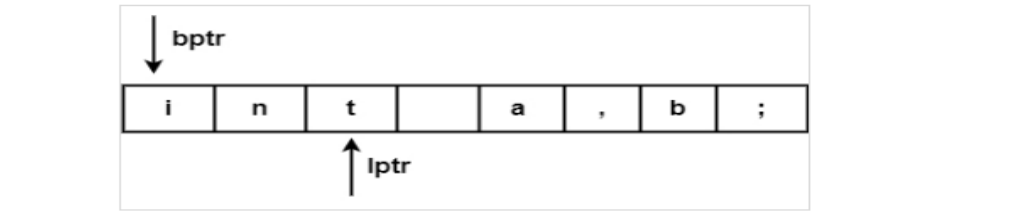
- **Begin Pointer (bptr)** − It points to the beginning of the string to be read.
- **Look Ahead Pointer (lptr)** − It moves ahead to search for the end of the token.

Example − For statement int a, b;

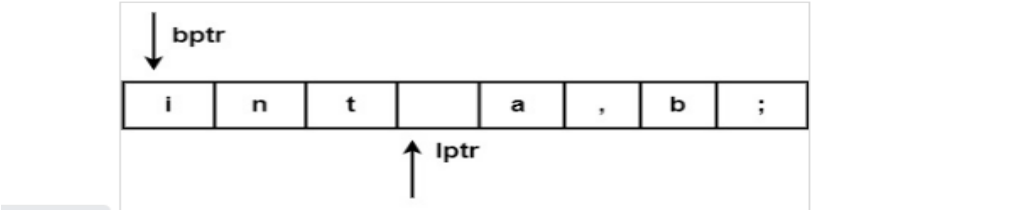
1. Both pointers start at the beginning of the string, which is stored in the buffer.



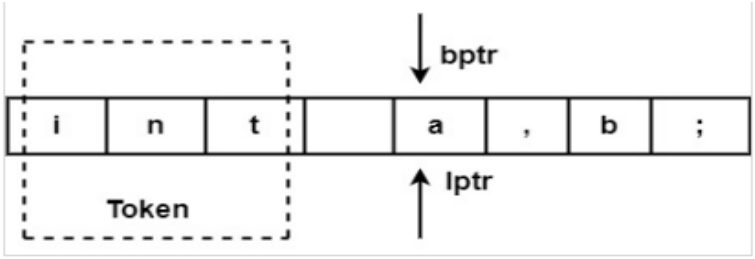
- Look Ahead Pointer scans buffer until the token is found.



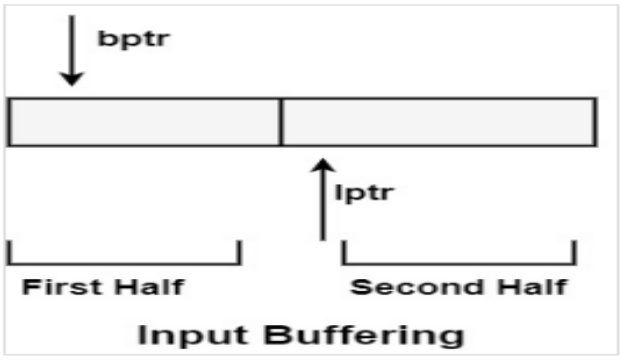
- The character ("blank space") beyond the token ("int") have to be examined before the token ("int") will be determined.



- After processing token ("int") both pointers will set to the next token ('a'), & this process will be repeated for the whole program.



A buffer can be divided into two halves. If the look Ahead pointer moves towards halfway in First Half, the second half is filled with new characters to be read. If the look Ahead pointer moves towards the right end of the buffer of the second half, the first half will be filled with new characters, and it goes on.



Sentinels − Sentinels are used to making a check, each time when the forward pointer is converted, a check is completed to provide that one half of the buffer has not converted off. If it is completed, then the other half should be reloaded.

Buffer Pairs − A specialized buffering technique can decrease the amount of overhead, which is needed to process an input character in transferring characters. It includes two buffers, each includes N-character size which is reloaded alternatively.

There are two pointers such as the lexeme Begin and forward are supported. Lexeme Begin points to the starting of the current lexeme which is discovered. Forward scans ahead before a match for a pattern are discovered. Before a lexeme is initiated, lexeme begin is set to the character directly after the lexeme which is only constructed, and forward is set to the character at its right end.

Preliminary Scanning − Certain processes are best performed as characters are moved from the source file to the buffer. For example, it can delete comments. Languages like

FORTRAN which ignores blank can delete them from the character stream. It can also collapse strings of several blanks into one blank. Pre-processing the character stream being subjected to lexical analysis saves the trouble of moving the look ahead pointer back and forth over a string of blanks.

UNIT - II

UNIT II

Symbol tables: Symbol table entries – List data structures for symbol table – - Hash tables – Representation of scope information – Syntax Analysis: Role of parser – Context free grammar – Writing a grammar – Top down parsing – Simple bottom up parsing – Shift reducing parsing.

Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations

A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

int a;

should be processed by the compiler as:

insert(a, int);

lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.

- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

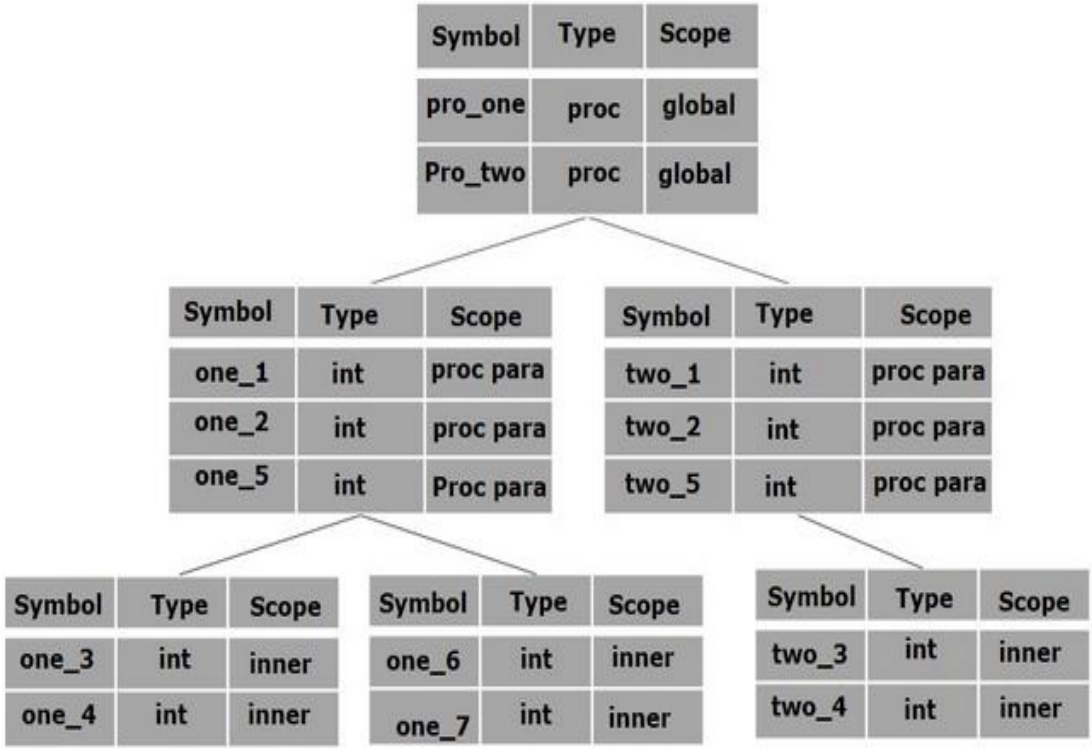
A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
  {
  int one_1;
  int one_2;

    {           \
    int one_3;    |_  inner scope 1
    int one_4;    |
    }           /

  int one_5;

    {           \
    int one_6;    |_  inner scope 2
    int one_7;    |
    }           /
  }

void pro_two()
  {
  int two_1;
```

```
   int two_2;

      {            \
      int two_3;     |_  inner scope 3
      int two_4;     |
      }            /

   int two_5;
   }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.

- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

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

Implementation of Symbol table

Following are commonly used data structures for implementing symbol table:-

1. **List**
    - In this method, an array is used to store names and associated information.
    - A pointer **"available"** is maintained at end of all stored records and new names are added in the order as they arrive
    - To search for a name we start from the beginning of the list till available pointer and if not found we get an error **"use of the undeclared name"**
    - While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e. **"Multiple defined names"**
    - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
    - The advantage is that it takes a minimum amount of space.
2. **Linked List**
    - This implementation is using a linked list. A link field is added to each record.
    - Searching of names is done in order pointed by the link of the link field.
    - A pointer **"First"** is maintained to point to the first record of the symbol table.
    - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
3. **Hash Table**
    - In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
    - A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
    - To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
    - Insertion and lookup can be made very fast – O(1).
    - The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.
4. **Binary Search Tree**
    - Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.

- All names are created as child of the root node that always follows the property of the binary search tree.
- Insertion and lookup are $O(\log_2 n)$ on average.

SYNTAX ANALYSIS

ROLE OF THE PARSER:

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementry sutree:

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

The two types of parsers employed are:

a. Top down parser: which build parse trees from top(root) tobottom(leaves)
b. Bottom up parser: which build parse trees from leaves and work up theroot.

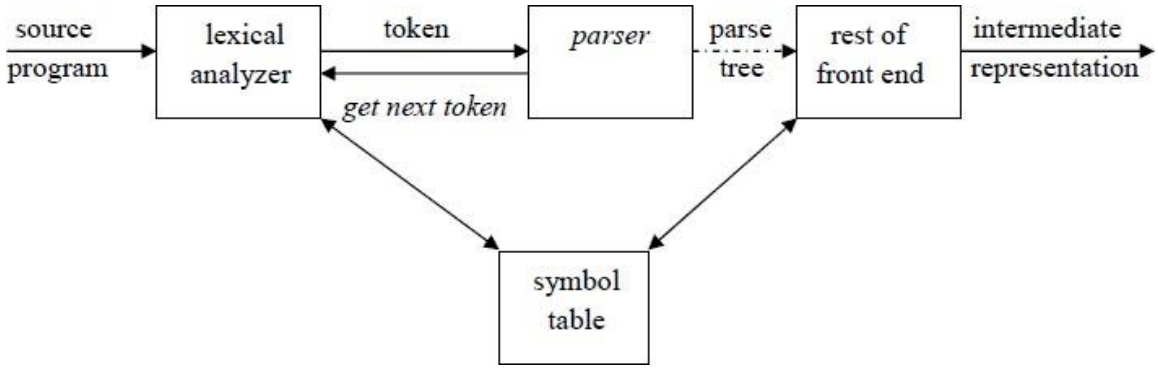


Fig . 4.1: position of parser in compiler model.

CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S).

Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals(including the empty string)

S is a start symbol (one of the non-terminal symbol)

L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff $S \Rightarrow \omega$ whereω is a string of terminals of G. If G is a context- free grammar, L(G) is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

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

Derivations

In general a derivation step is

$\alpha A\beta \Rightarrow \beta$ is sentential form and if there is a production rule $A \rightarrow \gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha 1 \Rightarrow \alpha 2 \Rightarrow ... \Rightarrow \alpha n$ (αn derives from $\alpha 1$ or $\alpha 1$ derives αn). There are two types of derivaion

1At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

2If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:

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

Leftmost derivation :

E → E + E

→E * E+E →id* E+E→id*id+E→id*id+id

The string is derive from the grammar w= id*id+id, which is consists of all terminalsymbols

Rightmost

derivation E → E

+ E

→E+E * E→E+

E*id→E+id*id→id+id*id Given grammar

G : E → E+E | E*E | (E) | - E | idSentence

to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST
DERIVATIONE → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)

E → - (id+id) E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.Sentinels:**
- Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

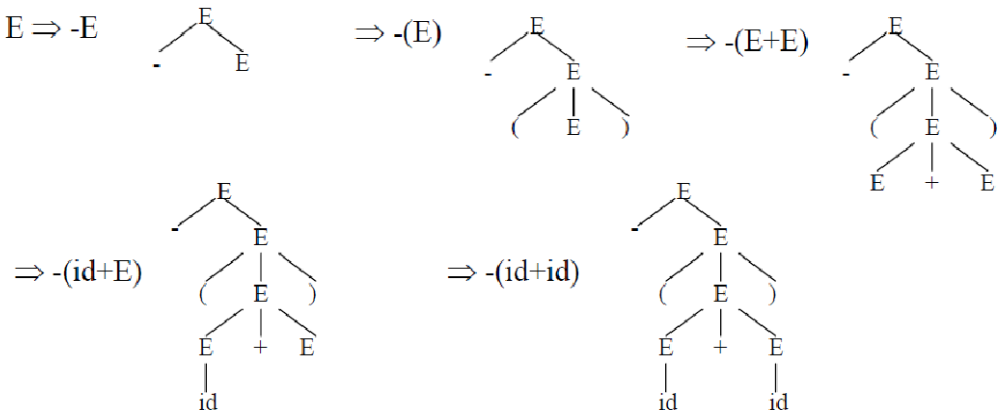
Yield or frontier of tree:

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

PARSE TREE

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

Example:



Ambiguity:

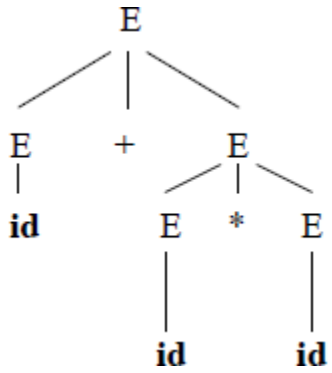
A grammar that produces more than one parse for some sentence is said to e **ambiguousgrammar**.

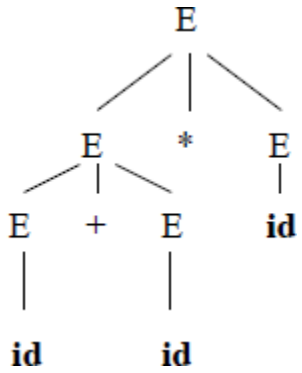
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

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

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :





Example:

To disambiguate the grammar E → E+E | E*E | E^E | id | (E), we can use precedence ofoperators as follows:

^ (right to left)

/,* (left to right)

-,+ (left to right) We get the following unambiguous grammar:

E → E+T | T

T → T*F |

FF → G^F

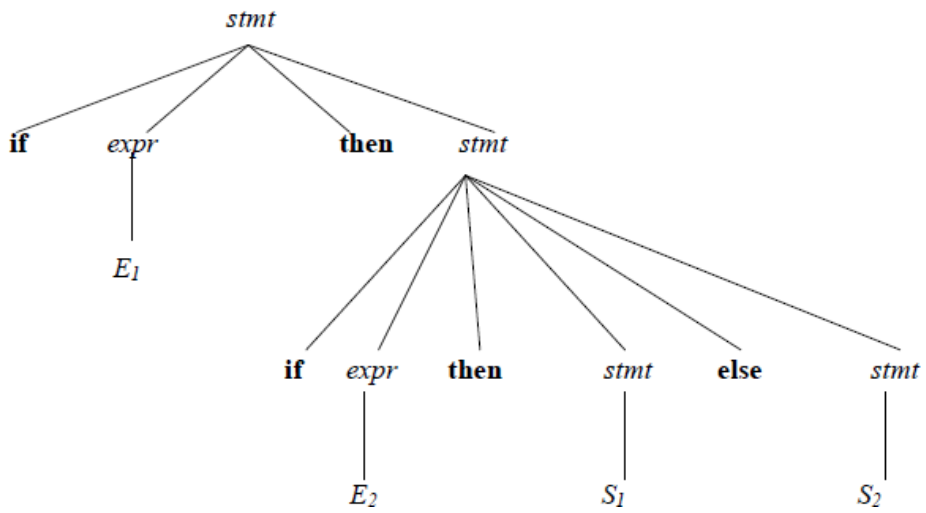
| GG → id

| (E)

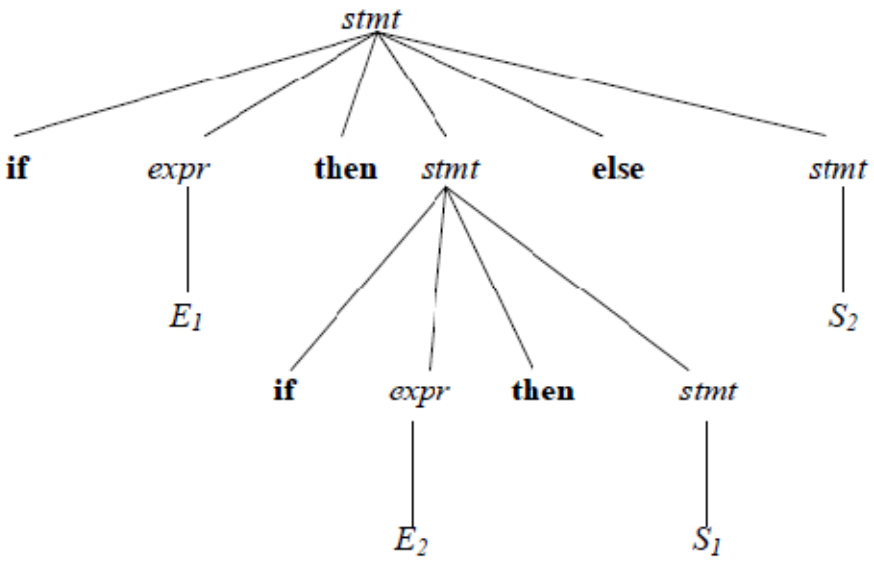
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* |**if** *expr* **then** *stmt* **else***stmt* | **other** This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following

Two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other** *unmatched_stmt* → **if** *expr* **then** *stmt*| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt***Eliminating Left Recursion:**

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

f there is a production A → Aα | β it can be replaced with a sequence of twoproductions

A → βA'

A' → αA' | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:E → E+T | T

T → T*F |

F F → (E)

| id

First eliminate the left recursion for E asE → TE'

E' → +TE' |ε

Then eliminate for T as

T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion isE → TE'

E' → +TE'

|εT → FT'

T' → *FT'

| εF → (E) |

id

Algorithm to eliminate left recursion:

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

. . . An.2.**for** *i* := 1 **to** *n* **do begin**

for *j* := 1 **to** *i*-1 **do begin**

replace each production of the form Ai → Aj γ

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

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

end

end

Factoring:

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

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

rewritten asA → αA'

A' → β1 | β2

Consider the grammar , G : S→iEtS | iEtSeS | a

E → b

Left factored, this grammar

becomesS → iEtSS' | a

S' → eS |

εE → b

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or anattempt to construct a parse tree for the input starting from

the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set ofrecursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of theinput.

Example for backtracking :

Consider the grammar G : S→cAd

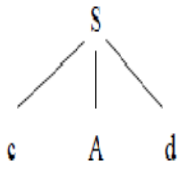
A → ab |

aand the input string w=cad.

The parse tree can be constructed using the following top-down approach :

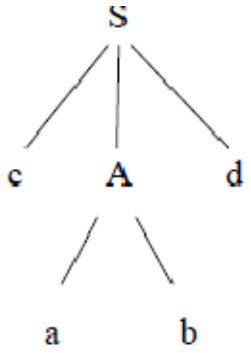
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the firstsymbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



Step3:

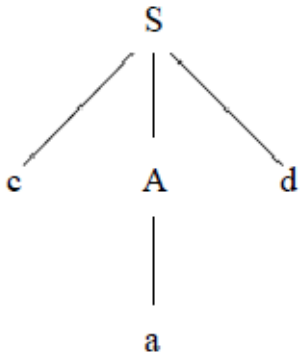
The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called

backtracking.

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.Hence, **elimination of left-recursion** must be done before parsing.
Consider the grammar for arithmetic
expressionsE → E+T | T
T → T*F |

F F → (E)

| id

After eliminating the left-recursion the grammar becomes, E → TE'

E' → +TE'

|εT → FT'

T' → *FT'

| εF → (E) |

id

Now we can write the procedure for grammar as follows:

Recursive procedure:

Procedure E()

begin

T( );

EPRIME( );

End

Procedure EPRIME( )

begin

end

put_symbol='+' thenADVANCE( );

T( );

EPRIME( );

Procedure T( )

begin

End

F( );

TPRIME( );

Procedure TPRIME( )

begin

end input_symbol=’*’ thenADVANCE( );

F( );

TPRIME( );

ocedure F( )

begin

end

If input-symbol=’id’ then

ADVANCE( );

else if input-symbol=’(‘ then

ADVANCE( );

E( );

else if input-symbol=’)’ then

ADVANCE( );

else ERROR( );

Stack implementation:

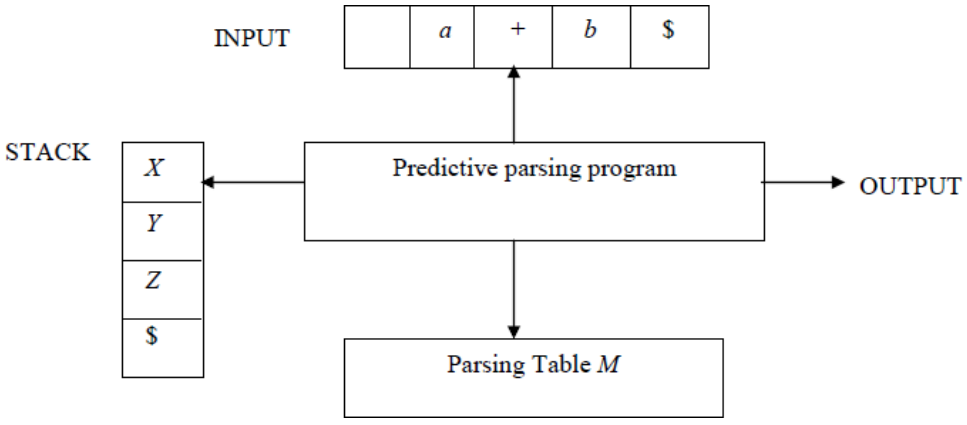
PROCEDURE	**INPUT STRING**
E( )	**id**+id*id
T( )	**id**+id*id

F( )	**id**+id*id
ADVANCE( )	id id*id
TPRIME( )	id id*id
EPRIME( )	id id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id*id
TPRIME( )	id+id***id
ADVANCE( )	id+id***id
F( )	id+id***id
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

PREDICTIVE PARSING

- ✓ Predictive parsing is a special case of recursive descent parsing where nobacktracking is required.
- ✓ The key problem of predictive parsing is to determine the production to be appliedfor a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an outputstream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack.Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array *M*[*A*, *a*], where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

Predictive parsing program:

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

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops *X* off the stack and advances the input pointer tothe next input symbol.
3. If *X* is a non-terminal , the program consults entry *M*[*X*, *a*] of the parsing table
4. *M*. This entry will either be an *X*-production of the grammar or an error entry.
5. If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces *X* on top of the stack by *UVW*
6. If *M*[*X*, *a*] =**error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

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

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

Method : Initially, the parser has $\$S$ on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parsefor the input is as follows:

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

repeat

letX be the top stack symbol and*a*the symbol pointed to by *ip*;

if *X* is a terminal or

$**then if** *X* = *a*

then

pop*X* from the stack and advance *ip*

else e*rror*()

else/* *X* is a non-terminal */

if *M*[*X*, *a*] = *X* →*Y1Y2* … *Yk* **then begin**

pop *X* from the stack;

push *Yk*, *Yk-1*, … ,*Y1* onto the stack, with *Y1*

on top;output the production *X* → *Y1 Y2* . . .

Yk

end

else*error*

()

until *X* = $

Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

Rules for first( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If *X* → ε is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and X → *a*α is a production then add *a* to FIRST(X).
4. If X is non-terminal and *X* → $Y_1\ Y_2 \ldots Y_k$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*),…,FIRST(*Yi-1*); that is, *Y1*,….*Yi-1*
   => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(*X*).

Rules for follow( ):

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed infollow(*B*).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, theneverything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input : Grammar*G*

Output : Parsing table *M***Method** :

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

Example:

Consider the following

grammar :E → E+T | T

T→T*F |

F F → (E)

| id

After eliminating left-recursion the

grammar isE → TE’

E’ → +TE’

|εT → FT’

T’ → *FT’

| εF → (E) |

id **First( ) :**

FIRST(E) = { (, id}

FIRST(E’) ={+ ,ε}

FIRST(T) = { (, id}

FIRST(T’) = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E’) = { $,) }

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

FOLLOW(T') = { +, $,) }

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

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry.This type of grammar is called LL(1) grammar.
Consider this following

grammar:S → iEtS | iEtSeS |
a
E → b
After eliminating left factoring,
we haveS→iEtSS' | a
S'→eS
|εE→b
To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.FIRST(S) = { i, a }
FIRST(S') =
{e,ε} FIRST(E)
= { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $
,e } FOLLOW(E) =
{t}

Parsing table:

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

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

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

Example:

Consider the grammar:

S → aABe

A → Abc |

bB → d

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)	**RIGHTMOST DERIVATION**
a**b**bcde (A → b)	**S**→ aA**B**e
a**Abc**de (A → Abc)	→ a**A**de
aA**d**e (B → d)	→ a**A**bcde
aABe(S → aABe)	→ abbcde
S	

The reductions trace out the right-most derivation in reverse.

Handles:

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

Example:

Consider the grammar:

E → E+E

E → E*E

E → (E)

E → id

And the input string $id_1+id_2*id_3$ The rightmost derivation is :

E →**E**+**E**

→ E+**E*E**

→ E+E***id**$_3$

→ E+**id**$_2$***id**$_3$

→**id** $_1$+id_2***id**$_3$

In the above derivation the underlined substrings are called **handles.** **Handle pruning:**

A rightmost derivation in reverse can be obtained by "**handle pruning**".

(i.e.) if *w* is a sentence or string of the grammar at hand, then *w* = y_n, where y_n is the *n* th right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	$id_1+id_2*id_3$ $	shift

$ id_1	$+id_2*id_3$ $	reduce by E→id
$ E	$+id_2*id_3$ $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	$*id_3$ $	reduce by E→id
$ E+E	$*id_3$ $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recoveryroutine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. **Shift-reduce conflict**: The parser cannot decide whether to shift or to reduce.

2. **Reduce-reduce conflict**: The parser cannot decide which of several reductions to make.

1. **Shift-reduce conflict: Example:**

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift

$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce byE→id
$ E*id	$	Reduce byE→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. **Reduce-reduce conflict:**

Consider the grammar: M → R+R | R+c | R

R → c

and input c+c

Stack	**Input**	**Action**	**Stack**	**Input**	**Action**
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce byR→c	$ c	+c $	Reduce byR→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce byR→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+ R	$ M	$	
$ M	$				

Viable prefixes:

- a is a viable prefix of the grammar if there is *w* such that a*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non- terminals.

Example:

Consider the grammar:

E → EAE | (E) | -
E | idA → + | - | * |
/ | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

E → E+E | E-E | E*E | E/E | E↑E | -E | id

Operator precedence relations:

There are three disjoint precedence relations namely

<· - less than

=- equal to

·>- greater than

The relations give the following meaning: a< · b – a yields precedence to b

a = b – a has the same precedence as b a · >b – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2, then make $\theta_1 \cdot > \theta_2$ and $\theta_2 < \cdot \theta_1$
2. If operators θ_1 and θ_2, are of equal precedence, then make $\theta_1 \cdot > \theta_2$ and $\theta_2 \cdot > \theta_1$ if operators are left associative $\theta_1 < \cdot \theta_2$ and $\theta_2 < \cdot \theta_1$ if right associative
3. Make the following for all operators θ: θ< · id , id · >θ
   θ< ·(, (< ·θ
) ·>θ , θ ·>)
   θ ·>$, $< · θ

Also make

(=) , (< · (,) ·>) , (< · id , id ·>) , $<· id , id · >$, $< · (,) ·>$

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and

3. + and - are of lowest precedence and left-associative Note that the**blanks**in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Operator precedence parsing algorithm:

Input : An input string*w*and a table of precedence relations.

Output : If*w*is well formed, a*skeletal*parse tree**,**with a placeholder non-terminal E labeling allinterior nodes; otherwise, an error indication.

Method : Initially the stack contains $ and the input buffer the string*w*$. To parse, we executethe following program :

(1) Set*ip*to point to the first symbol of*w*$;(2)**repeat forever**

(3)**if**$ is on top of the stack and*ip*points to $**then**(4)**return**

else begin

(5) let*a*be the topmost terminal symbol on the stack and let*b*be the symbol pointed to by*ip;*

(6)if*a*< · *b*or*a=b***then begin**

(7) push*b*onto the stack;

(8) advance*ip*to the next input symbol;

end;

(9)**else if***a* ·>*b***then** **/***reduce***/**

(10)**repeat**

(11) pop the stack

(12)**until**the top stack terminal is related by < · to the terminal most recently popped(13)**else**error( )

end

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is**id+id*id**.Theimplementation is as follows:

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

$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

LR PARSERS

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

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can bewritten.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar thatcan be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

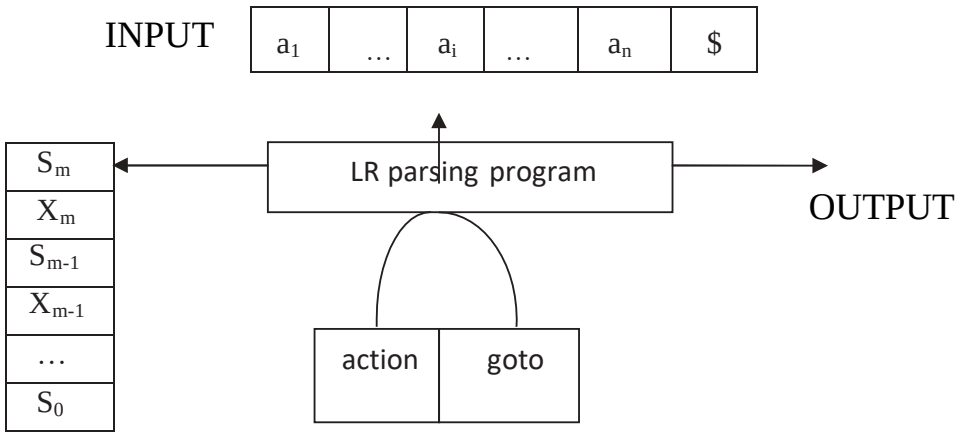
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



STACK

It consists of : an input, an output, a stack, a driver program, and a parsing table that has twoparts (*action*and*goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is ontop. Each X_i is a grammar symbol and each s_i is a state.

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

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

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

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

LR Parsing algorithm:

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

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

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and*w*$ in the input buffer. The parser then executes the following program :

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

repeat forever begin

let*s*be the state on top of the stack and

*a*the symbol pointed to by*ip*;

if*action*[*s*,*a*] = shift*s*'**then begin** push*a*then*s*' on top of the stack; advance*ip*to the next input symbol

end

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

pop 2* | β | symbols off the stack;

lets' be the state now on top of the stack; push A then*goto*[*s*', A] on top of the stack; output the production A→ β

end

else if*action*[*s,a*] = accept**then return**

else*error*( )

end

CONSTRUCTING SLR(1) PARSING TABLE:

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

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

LR(O) items:

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

A →**.**XYZ
A → X**.**YZ A → XY**.**Z
A → XYZ**.**

Closure operation:

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

1. Initially, every item in I is added to closure(I).
2. If A → a . Bβ is in closure(I) and B → y is a production, then add the item

B → . y to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

3.

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ aX . β] such that [A→ a . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

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

Algorithm for construction of SLR parsing table:

Input: An augmented grammar G'

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

Method:

1. Construct C = {$I_0, I_1, \ldots I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→a·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→a·] is in I_i , then set *action*[*i*,*a*] to "reduce A→a" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

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

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] = *j*.

4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing[S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :G : E → E + T | T

T → T * F
| FF → (E)
| id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E

E → E + T

E → T

T → T *
FT → F

F → (E)

F → id

Step 2 : Find LR (0)

items. I_0 : E' →.E

E →.E + T

E →.T

T →.T *

FT →.F

F →.(E)

F →.id

GOTO (I_0 , E)I_1 : E' → E.

E → E.+ T

GOTO (I_4 , id)

I_5 : F → id.

GOTO (I_6 , T)

GOTO (I_0 , T)

I_2 : E → T.

T → T.* F

GOTO (I_0 , F) I_3 : T → F.

I_9 : E → E + T.

T → T.* F

GOTO (I_6 , F) I_3 : T → F.

GOTO (I_6 , ( )

I_4 : F → (.E)

GOTO (I_0 , ( )

I_4 : F → (.E)

E →.E +
TE →.T

T →.T *
FT →.F

F →.(E)

F →.id

GOTO (I_0 , id)

I_5 : F → id.

GOTO (I_1 , +)I_6 : E → E +.T

T →.T *
FT →.F

F →.(E)

F →.id

GOTO (I_2 , *) I_7 : T → T *.F

F →.(E)

F →.id

GOTO (I_4 , E)I_8 : F → (E.)

E → E.+ T

GOTO (I_6 , id)

I_5 : F → id.

GOTO (I_7 , F)I_{10} : T → T * F.

GOTO (I_7 , ( )

I_4 : F → (.E)

E →.E +
TE →.T

T →.T *
FT →.F

F →.(E)

F →.id

GOTO (I_7 , id)

I_5 : F → id.

GOTO (I_8 ,))

I_{11} : F → (E).

GOTO (I_8 , +) I_6 : E → E +.T

T →.T *
FT →.F

F →.(E)

F →.id

GOTO (I_4 , T)I_2 : E →T**.**

T → T**.*** F

GOTO (I_4 , F)I_3 : T → F**.**

GOTO (I_9 , *)I_7 : T → T ***.**F

F →**.**(E)

F →**.**id

GOTO (I_4 , ( )

I_4 : F → (.E)

E →.E + T

E →.T

T →.T * F

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

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

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

SLR parsing table:

	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_O	s 5			s 4			1	2	3
I_1		s 6				AC C			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s 5			s 4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s 5			s 4				9	3
I_7	s 5			s 4					10
I_8		s 6			s11				
I_9		r1	s7		r1	r1			
I_{1O}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input**id + id * id**is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ;**shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ;**reduce**by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ;**reduce**by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ;**reduce**by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ;**shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ;**shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ;**reduce**by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ;**reduce**by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ;**shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ;**shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ;**reduce**by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ;**reduce**by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ;**reduce**by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) =**accept**

UNIT – III

Syntax-Directed Translation -Definition

The translation techniques in this chapter will be applied to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks; this chapter includes an example from typesetting.

We associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION SEMANTIC RULE

PRODUCTION	SEMANTIC RULE
E -> Ei + T	E.code = Ei.code || T.code || '+'

This production has two nonterminals, E and T; the subscript in E1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute code. The semantic rule specifies that the string E.code is formed by concatenating Ei.code, T.code, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

E -» Ei + T { print '+' }

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in ' { ' and '}'.) The

position of a semantic action in a production body determines the order in which the action is executed. In production (5.2), the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "L-attributed translations" (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "S-attributed translations" (S for synthesized), which can be performed easily in connection with a bottom-up parse.

Construction of syntax trees

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

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

Example

E -> E+T | T

T -> T*F | F

F -> INTLIT

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

E -> E+T { E.val = E.val + T.val } PR#1

E -> T { E.val = T.val } PR#2

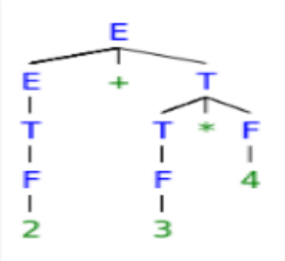
T -> T*F { T.val = T.val * F.val } PR#3

T -> F { T.val = F.val } PR#4

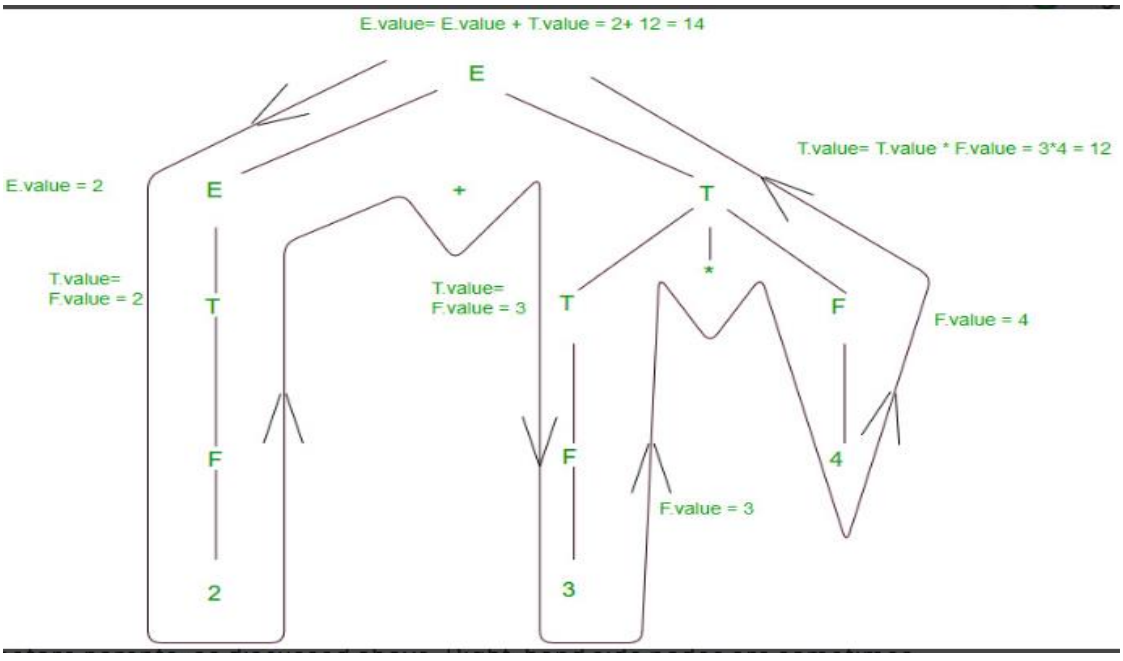
F -> INTLIT { F.val = INTLIT.lexval } PR#5

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

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



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

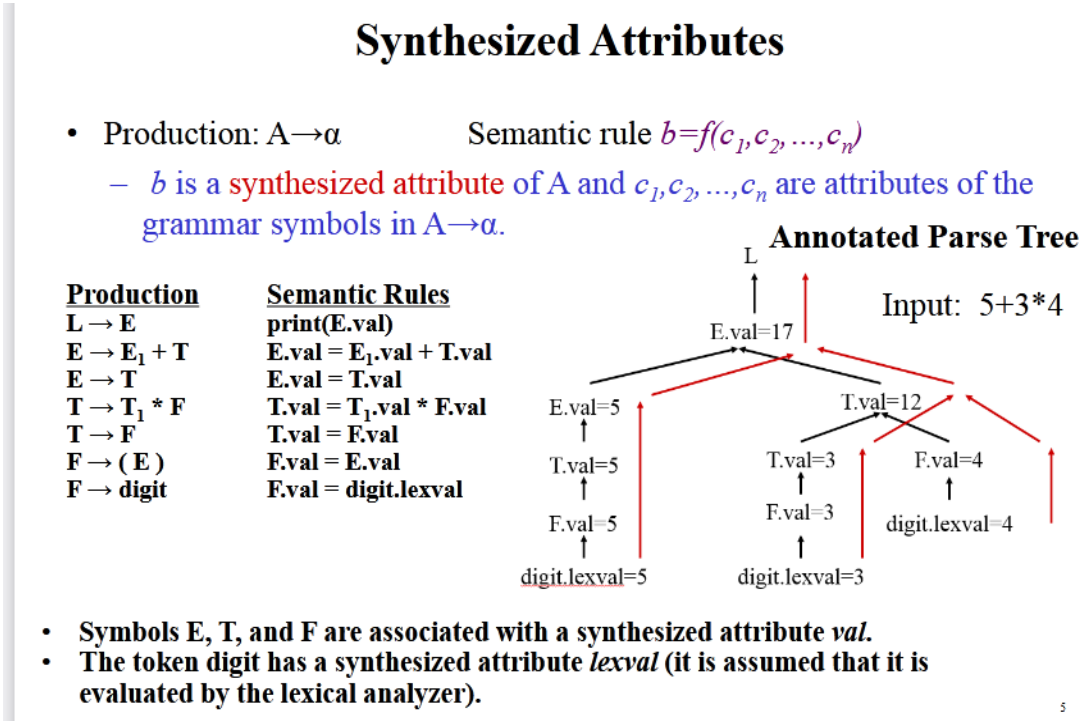


The above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children's attributes are computed before parents, as discussed above. Right-hand side nodes are sometimes annotated with

subscript 1 to distinguish between children and parents. Additional Information

Synthesized Attributes are such attributes that depend only on the attribute values of children nodes.

Thus [E -> E+T { E.val = E.val + T.val }] has a synthesized attribute val corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth-first search traversal in any order is sufficient for the semantic analysis phase.



Inherited Attributes are such attributes that depend on parent and/or sibling's attributes.

Thus [Ep -> E+T { Ep.val = E.val + T.val, T.val = Ep.val }], where E & Ep are same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

Inherited Attributes

Production: A→α **Semantic rule $b=f(c_1,c_2,\ldots,c_n)$**

- ***b* is an inherited attribute of one of the grammar symbols in α and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in A→α.**

Production	Semantic Rules
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 , **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

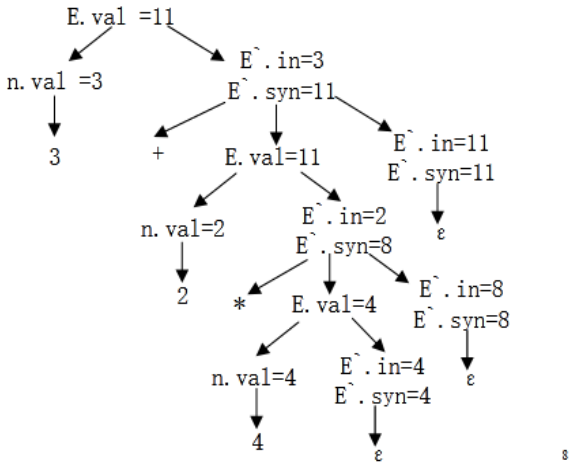
Synthesized and Inherited Attributes

- Sometimes both synthesized and inherited attributes are required to evaluate necessary information

E:= n E'
E':= +EE' | *EE' | ε

Production	Semantic rule
E:= n E'	E'.in= n.val; E.val = E'.syn
E':= + EE'$_1$	E'$_1$.in = E'.in + E.val; E'.syn = E'$_1$.syn
E' :=* EE'$_1$	E'$_1$.in = E'.in * E.val; E'.syn = E'$_1$.syn
E': = ε	E'.syn = E'.in

3+2*4



Difference between Synthesized and Inherited Attributes

r. No.	Key	Synthesized Attribute	Inherited Attribute
1	Definition	Synthesized attribute is an attribute whose parse tree node value is determined by the attribute value at child nodes. To illustrate, assume the following production S → ABC if S is taking values from its child nodes (A, B,	On other hand an attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node. In case of S → ABC if A can get values from S, B and C. B can take values from

		C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.	S, A, and C. Likewise, C can take values from S, A, and B then S is said to be Inherited Attribute.
2	Design	As mentioned above in case of Synthesized attribute the production must have non-terminal as its head.	On other hand in case of Inherited attribute the production must have non-terminal as a symbol in its body.
3	Evaluation	Synthesized attribute can be evaluated during a single bottom-up traversal of parse tree.	While on other hand Inherited attribute can be evaluated during a single top-down and sideways traversal of parse tree.
4	Terminal	Both terminal and Non terminals can contain the Synthesized attribute.	On other hand only Non terminals can contain the Inherited attribute.
5	Usage	Synthesized attribute is used by both S-attributed SDT and L-attributed STD.	On other hand Inherited attribute is used by only L-attributed SDT.

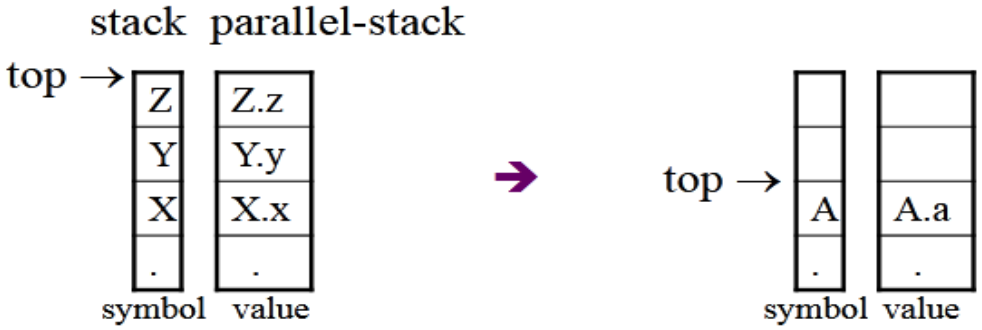
Bottom-Up Evaluation of S-Attributed Definitions

S-attributed Definition: Syntax-Directed Definition using only synthesized attributes. Stack of a LR parser contains states.

Recall that each state corresponds to some grammar symbol and many different states might correspond to the same grammar symbol.

Keep attribute values of grammar symbols in tack. Evaluate attribute values at each reduction.

$A \rightarrow XYZ$ $A.a=f(X.x,Y.y,Z.z)$ where all attributes are synthesized.



In a bottom-up evaluation of a syntax directed definition, inherited attributes can

1. always be evaluated
2. be evaluated only if the definition is L-attributed .
3. be evaluated only if the definition has synthesized attributes.
4. never be evaluated.

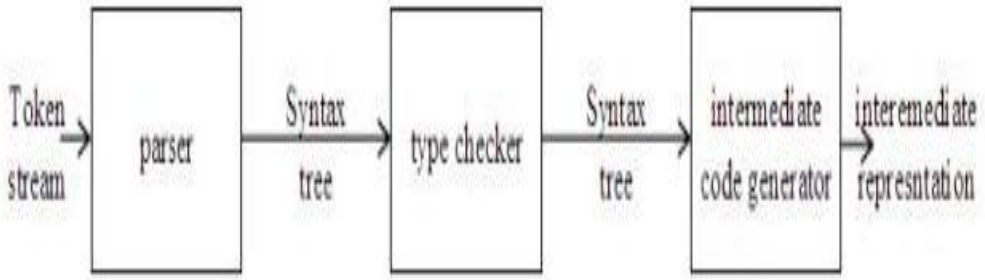
TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Some examples of static checks:

1. Type checks - A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. Flow-of-control checks - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switch statement.



A type checker verifies that the type of a construct matches that expected by its context. For example: arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to

language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

The type of a language construct will be 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. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.
   A special basic type, type_error , will signal an error during type checking; void denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

Constructors include:

Arrays : If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

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

Records : The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.

For example:

type row = record
address: integer;

lexeme: array[1..15] of char

end;

var table: array[1...101] of row;

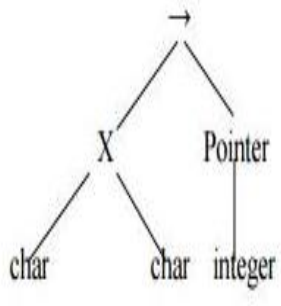
declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

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, var p: ↑ row declares variable p to have type pointer(row).

Functions : A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.



Tree representation for char x char → pointer (integer)

Type systems

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. It is specified in a syntax-directed manner. Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A sound type system eliminates the need for dynamic checking fo allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input. Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

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

A Simple Language

Consider the following grammar:

P → D ; E

D → D ; D | id : T

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

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

Translation scheme:

P → D ; E

D → D ; D

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

T → char { T.type : = char }

T → integer { T.type : = integer }

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

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

In the above language,
→ There are two basic types : char and integer ; → type_error is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression

pointer (integer). **Type checking of expressions**

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

1. E → literal { E.type : = char } E→num { E.type : = integer }

Here, constants represented by the tokens literal and num have type char and integer.

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

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

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

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

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

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

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

The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t of the object pointed to by the pointer E.

Type checking of statements

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

Translation scheme for checking the type of statements:

1. Assignment statement: S→id: = E
2. Conditional statement: S→if E then S1

3 While statement:

S → while E do S1

4. Sequence of statements:

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

Type checking of functions

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

UNIT –IV

Run-Time Environment

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for

procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Activation Trees

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

To understand this concept, we take a piece of code as an example:

```
. . .
printf(“Enter Your Name: “);
scanf(“%s”, username);
show_data(username);
printf(“Press any key to continue…”);
. . .
int show_data(char *user)
  {
  printf(“Your name is %s”, username);
  return 0;
  }
. . .
```

Below is the activation tree of the code given.

Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.

As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

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

is an l-value error, as the constant 7 does not represent any memory location.

Formal Parameters

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
  int actual_parameter = 10;
  call fun_two(int actual_parameter);
}
  fun_two(int formal_parameter)
{
```

```
  print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
  y = 10;
  copy_restore(y); //l-value of y is passed
  printf y; //prints 99
```

```
}
copy_restore(int x)
{
  x = 99; // y still has value 10 (unaffected)
  y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

SOURCE LANGUAGE ISSUES

Procedure

A *procedure* definition is a declaration that associates an identifier with a statement. The identifier is *procedure* name, and statement is the *procedure* body.

For example, the following definition of procedure named *readarray*

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears with in an executable statement, the procedure is

said to be
called at that point.

Activation Tree

Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure.

If 'a' and 'b' be two procedures, then their activations will be non-overlapping (when one is called after other) or nested (nested procedures).

A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. An activation tree shows the way control enters and leaves, activations.

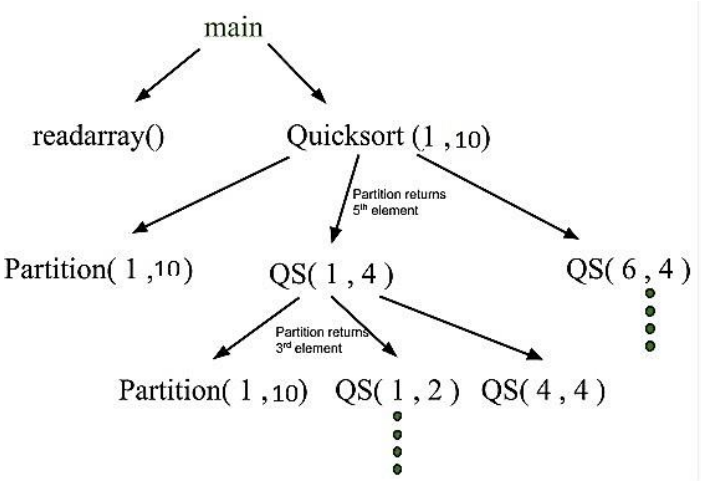
Properties of activation trees are :-

- Each node represents an activation of a procedure.
- The root shows the activation of the main function.
- The node for procedure 'x' is the parent of node for procedure 'y' if and only if the control flows from procedure x to procedure y.

EXAMPLE

Consider the following program of quicksort

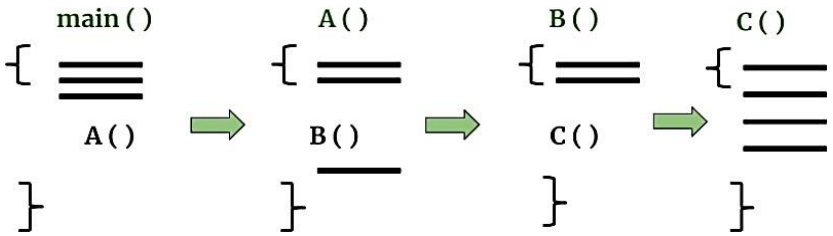
```
main()
{
readarray();
quicksort(1,10);
}

quicksort(int m, int n)
{

int i= partition(m,n);
quicksort(m,i-1);
quicksort(i+1,n);
```



}

First main function as root then main calls readarray and quicksort.

Quicksort in turn calls partition and quicksort again. The flow of control in a program corresponds to the depth first traversal of activation tree which starts at the root.



Before going to A() activation record of main is created when A() is called activation record of A() is created.

Before going to B() activation record of main and A() in stack.

Before going to C() activation record of main and B() is pushed in stack.

Control Stack

Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed.

A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends).

Information needed by a single execution of a procedure is managed using

an activation record.

When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.

Then the contents of the control stack are related to paths to the root of the activation tree. When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

Consider the above activation tree, when quicksort(4,4) gets executed, the contents of control stack were main() quicksort(1,10) quicksort(1,4), quicksort(4,4)

quicksort(4,4)
Quicksort(1,4)
Quicksort(1.10)
Main()

The Scope of Declaration

A declaration is a syntactic construct that associates information with a name. Declaration may be explicit such as

var i : integer;

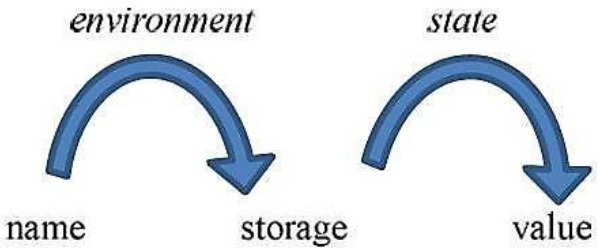
or may be explicit. The portion of program to which a declaration applies is called the

scope of that declaration.

Binding Of Names

Even if each name is declared once in a program, the same name may denote different data object at run time. "Data objects" corresponds to a storage location that hold values.

The term *environment* refers to a function that maps a name to a storage location. The term *state* refers to a function that maps a storage location to the value held there.



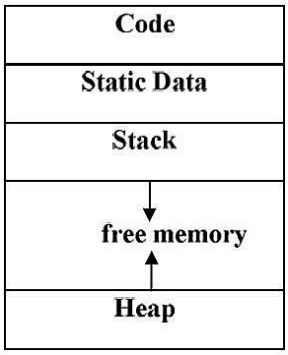
When an environment associates storage location *s* with a name *x*, we say that *x* is bounds to *s*. This association is referred to as a binding of *x*.

STORAGE ORGANIZATION

The executing target program runs in its own logical address space in which each program value has a location

The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread through memory.

Typical subdivision of run time memory.



Code area: used to store the generated executable instructions, memory locations for the code are determined at compile time

Static Data Area: Is the locations of data that can be determined at compile time

Stack Area: Used to store the data object allocated at runtime. eg. Activation records

Heap: Used to store other dynamically allocated data objects at runtime (for ex: malloac)

This runtime storage can be subdivided to hold the different components of an existing system

1. Generated executable code
2. Static data objects
3. Dynamic data objects-heap
4. Automatic data objects-stack

Activation Records

It is LIFO structure used to hold information about each instantiation.

Procedure calls and returns are usually managed by a run time stack called control stack.

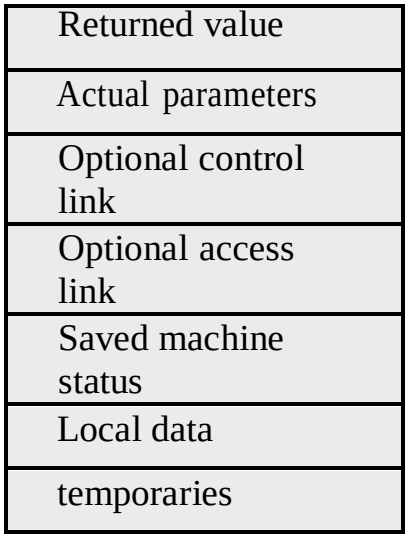
Each live activation has an activation record on control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack

The contents of the activation record vary with the language being implemented. The diagram below shows the contents of an activation record.

The purpose of the fields of an activation record is as follows, starting from the field for temporaries.

1. Temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.
2. The field for local data holds data that is local to an execution of a procedure.
3. The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to be restored when control returns from the procedure.
4. The optional access link is used to refer to nonlocal data held in other activation records.
5. The optional control /ink paints to the activation record of the caller

6. The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
7. The field for the returned value is used by the called procedure to return a value to the calling procedure, Again, in practice this value is often returned in a register for greater efficiency.



General Activation Record

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are:

Static allocation - lays out storage for all data objects at compile time

Stack allocation - manages the run-time storage as a stack.

Heap allocation - allocates and deallocates storage as needed at run time from a data area known as heap.

Static Allocation

In static allocation, names bound to storage as the program is compiled, so there is no need for a run-time support package.

Since the bindings do not change at runtime, every time a procedure activated, its run- time, names bounded to the same storage location.

Therefore, values of local names retained across activations of a procedure. That is when control returns to a procedure the value of the local are the

same as they were when control left the last time.

From the type of a name, the compiler decides amount of storage for the name and decides where the activation records go. At compile time, we can fill in the address at which the target code can find the data it operates on.

Stack Allocation

All compilers for languages that use procedures, functions or methods as units of user functions define actions manage at least part of their runtime memory as a stack run-time stack.

Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off from the stack

Calling Sequences

Procedures called implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.

A return sequence is similar to code to restore the state of a machine so the calling procedure can continue its execution after the call.

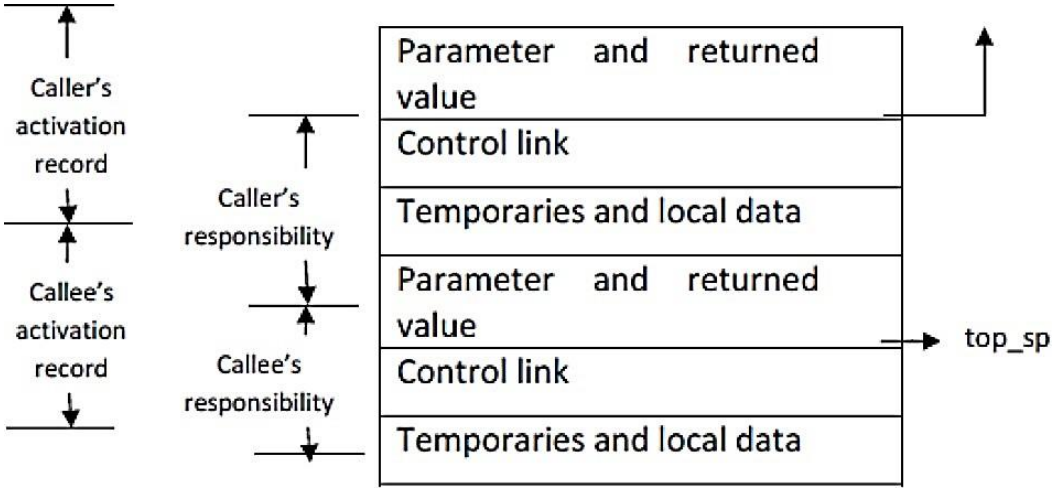
The code is calling sequence of often divided between the calling procedure (caller) and a procedure is calls (callee)(callee).

When designing calling sequences and the layout of activation record, the following principles are helpful:

1. Value communicated between caller and callee generally placed at the caller beginning of the callee's activation record, so they as close as possible to the caller's activation record.
2. Fixed length items generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.
3. Items whose size may not be known early enough placed at the end of the activation record.
4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation is to have it point to fix the end of fixed

length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to the top of the stack pointer.

- The calling sequence and its division between caller and callee are as follows:
  1. The caller evaluates the actual parameters.
  2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
  3. The callee-saves the register values and other status information.
  4. The callee initializes its local data and begins execution.



- A suitable, corresponding return sequence is:
  1. The callee places the return value next to the parameters.
  2. Using the information in the machine status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
  3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller, therefore, may use that value.

Variable length data on the stack

- The run-time memory-management system must deal frequently with the allocation of space for objects the sizesof which are not known at

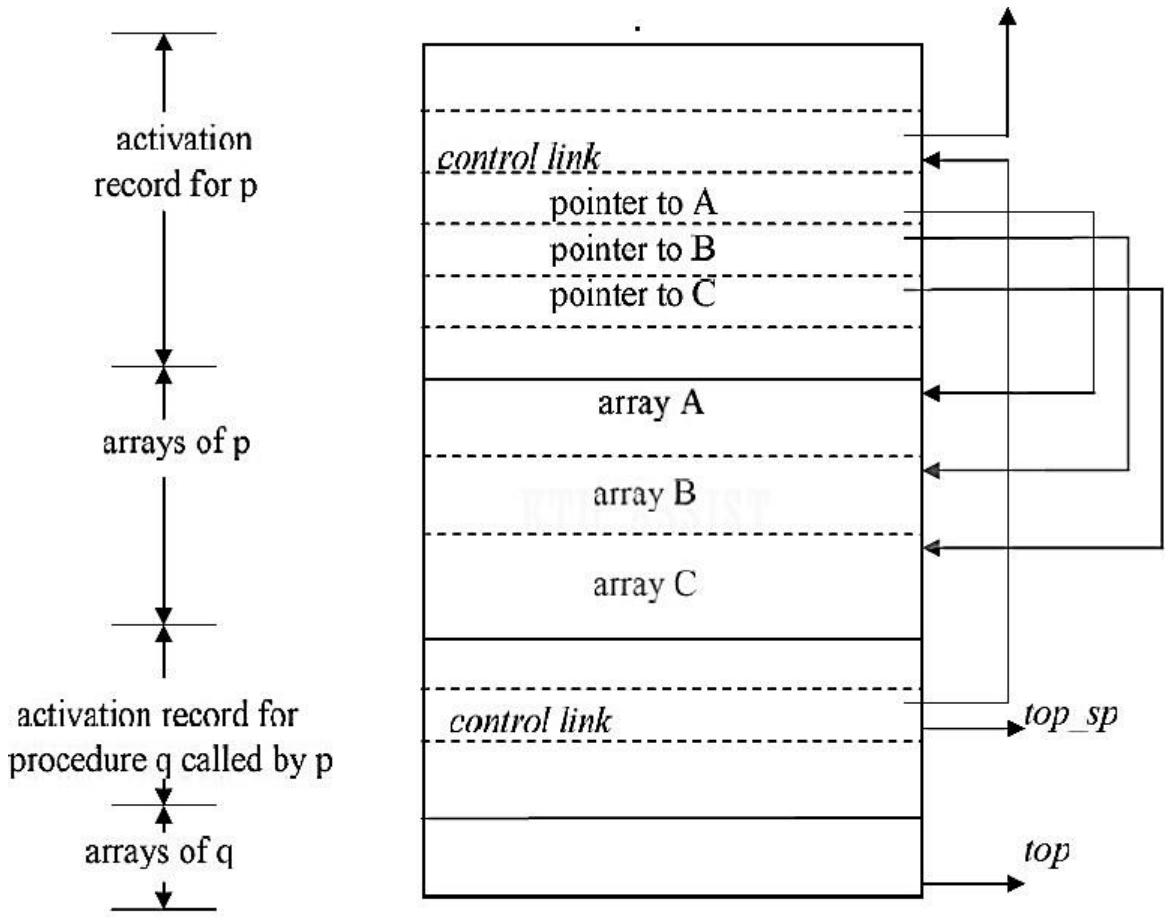
compile time, but which are local to a procedure and thus may be allocated on the stack.

- In modern languages, objects whose size cannot be determined at compile time are

  allocated space in the heap

- However, it is also possible to allocate objects, arrays, or other structures of unknown size on the stack.
- We avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

A common strategy for allocating variable-length arrays is shown in following figure



Access to dynamically allocated arrays

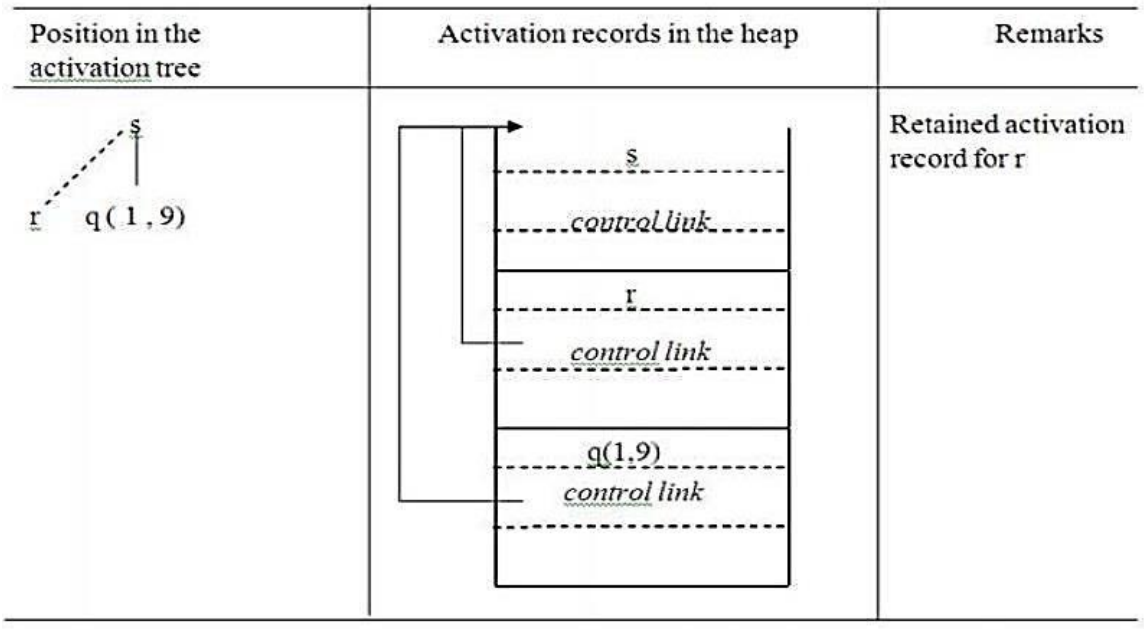
Heap Allocation

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.

Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.



Records for live activations need not be adjacent in heap

The record for an activation of procedure r is retained when the activation ends.

Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.

If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

INTERMEDIATE CODEGENERATION (ICG)

In compiler, the front-end translates a source program into an intermediate representation from which the back end generates target code.

Need For ICG

1. If a compiler translates the source language to its target machine language without generating IC, then for each new machine, a full native compiler is required.
2. IC eliminates the need of a new full compiler for every machine by keeping the analysis portion for all the compilers.
3. Synthesis part of back end depends on the target

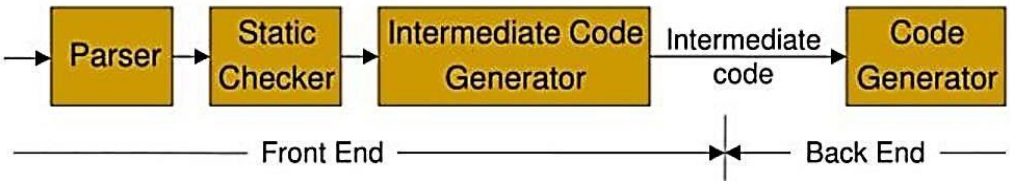
machine. 2 important things:

- IC Generation process should not be very complex
- It shouldn't be difficult to produce the target program from the intermediate code.

A source program can be translated directly into the target language, but some benefits of using intermediate form are:

- Retargeting is facilitated: a compiler for a different machine can be created by attaching a Back-end (which generate Target Code) for the new machine to an existing Front-end (which generate Intermediate Code).
- A machine Independent Code-Optimizer can be applied to the Intermediate Representation.

Logical Structure of a Compiler Front End



INTERMEDIATE LANGUAGES

The most commonly used intermediate representations were:-

- Syntax Tree
- DAG (Direct Acyclic Graph)
- Postfix Notation
- 3 Address Code

GRAPHICAL REPRESENTATION

Includes both

- Syntax Tree
- DAG (Direct Acyclic Graph)

Syntax Tree Or Abstract Syntax Tree (AST)

Graphical Intermediate Representation

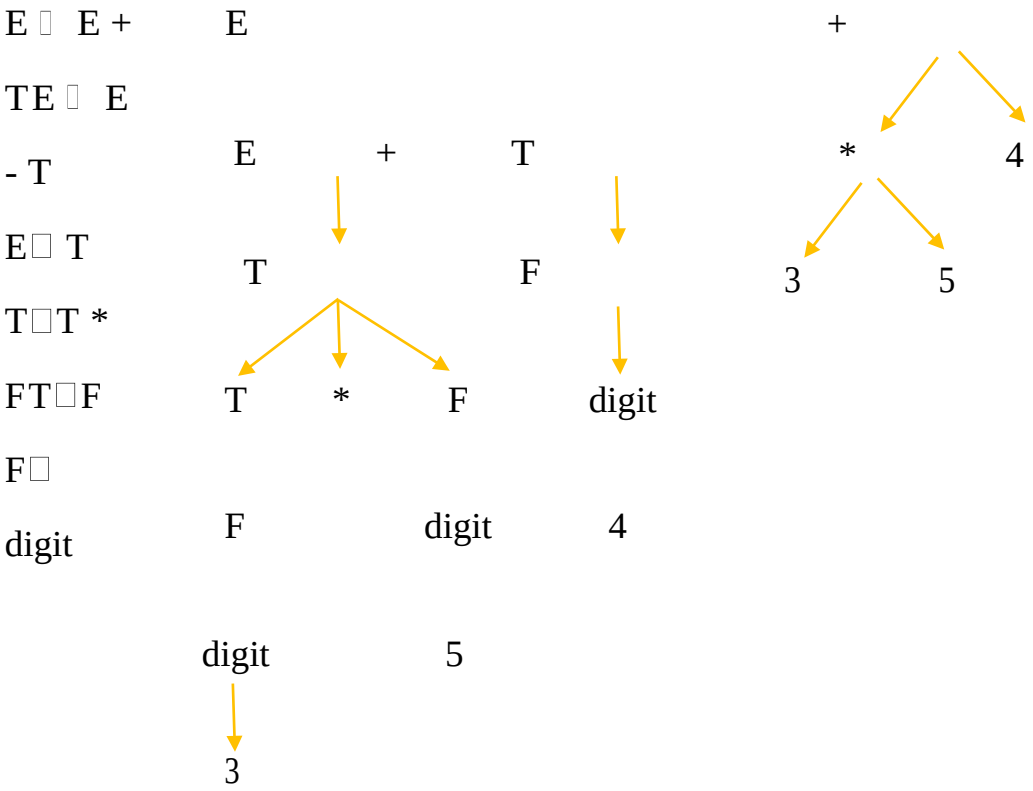
Syntax Tree depicts the hierarchical structure of a source program.

Syntax tree (AST) is a condensed form of parse tree useful for representing language constructs.

EXAMPLE

Parse tree and syntax tree for 3 * 5 + 4 as follows.

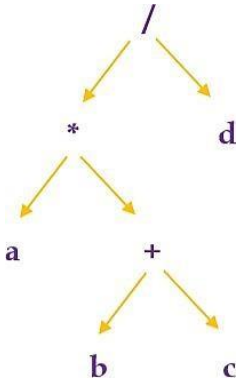
Grammar Parse Tree Syntax Tree



Parse Tree VS Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of a replacement process in a derivation	A syntax tree (AST) is a condensed form of parse tree
Each interior node represents a grammar rule	Each interior node represents an operator
Each leaf node represents a terminal	Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax tree does not represent every detail from the real syntax Eg : No parenthesis

Syntax tree for a * (b + c) /d



Constructing Syntax Tree For Expression

- Each node in a syntax tree can be implemented in a record with several fields.
- In the node of an operator, one field contains operator and remaining field contains pointer to the nodes for the operands.
- When used for translation, the nodes in a syntax tree may contain addition of fields to hold the values of attributes attached to the node.
- Following functions are used to create syntax tree

1. mknode(op,left,right): creates an operator node with label op and two fields containing pointers to left and right.
2. mkleaf(id,entry): creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for identifier
3. mkleaf(num,val): creates a number node with label num and a field containing val, the value of the number.

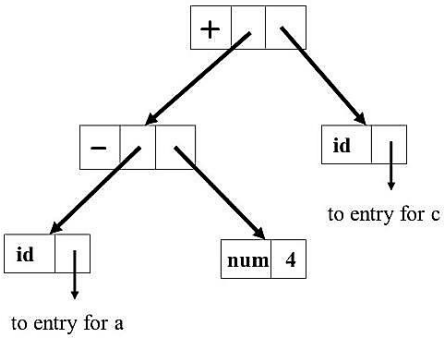
- Such functions return a pointer to a newly created node.

EXAMPLE

a – 4 + c

The tree is constructed bottom up

P1 = mkleaf(id,entry a)
P2 = mkleaf(num, 4) P3 = mknode(-, P1, P2) P4 = mkleaf(id,entry c)
P5 = mknode(+, P3, P4)



Syntax Tree

Syntax directed definition

- Syntax trees for assignment statements are produced by the syntax-directed

definition.

Non-terminal S generates an assignment statement.

The two binary operators + and * are examples of the full operator set in a typical language. Operator associates and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a:=b* -c + b* -c

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

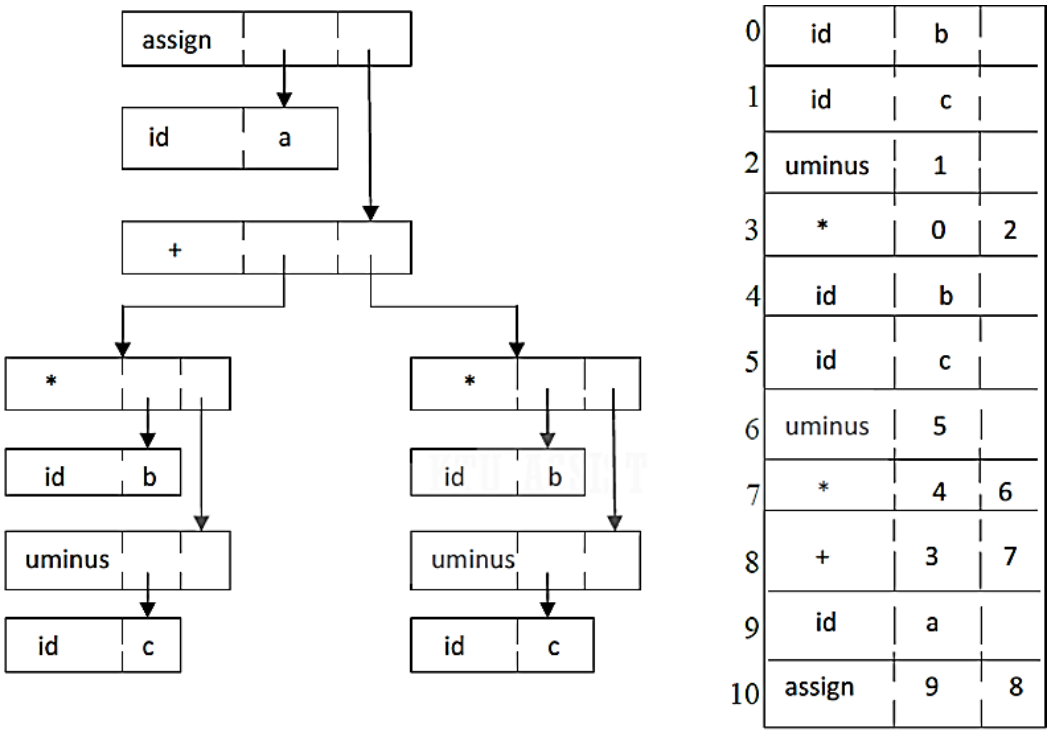
Syntax-directed definition to produce syntax trees for assignment statements

The token id has an attribute *place* that points to the symbol-table entry for the identifier.

A symbol-table entry can be found from an attribute id.*name*, representing the lexeme associated with that occurrence of id.

If the lexical analyser holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows.



(a) (b)

In (a), each node is represented as a record with a field for its operator and additional fields for pointers to its children.

In Fig (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.

All the nodes in the syntax tree can be visited by following winters, starting from the root at position 10.

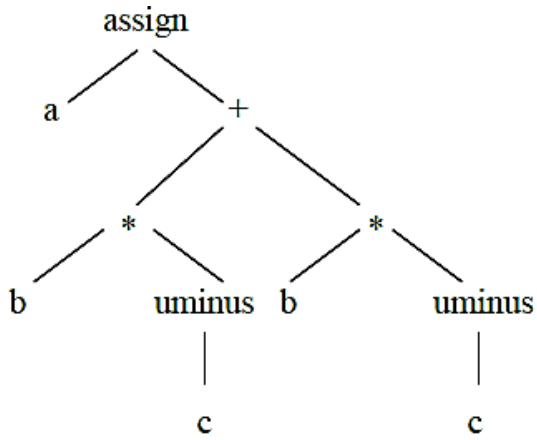
Direct Acyclic Graph (DAG)

Graphical Intermediate Representation

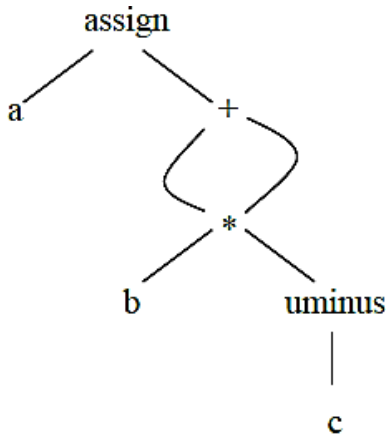
Dag also gives the hierarchical structure of source program but in a more compact way because common sub expressions are identified.

EXAMPLE

a=b-c + b*-c*



(a) Syntax tree



(b) Dag

Postfix Notation

Linearized representation of syntax tree

In postfix notation, each operator appears immediately after its last operand.

Operators can be evaluated in the order in which they appear in the string

EXAMPLE

Source String : a := b * -c + b * -c

Postfix String: a b c uminus * b c uminus * + assign

Postfix Rules

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E1 op E2 then postfix notation for E is E1' E2' op, here E1' and E2' are the postfix notations for E1and E2, respectively
3. If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation for E.
4. For unary operation –E the

   postfix is E- Ex: postfix notation

   for 9- (5+2) is 952+-

Postfix notation of an infix expression can be obtained using stack

THREE-ADDRESS CODE

In Three address statement, at most 3 addresses are used to represent any statement.

The reason for the term "three address code" is that each statement contains 3 addresses at most. Two for the operands and one for the result.

General Form Of 3 Address Code

a = b op c

where,

a, b, c are the operands that can be names, constants or compiler generated temporaries.

op represents operator, such as fixed or floating point arithmetic operator or a logical operator on Boolean valued data. Thus a source language expression like x + y * z might be translated into a sequence

t1 := y*z

t2 := x+t1 where, t1 and t2 are compiler generated temporary names.

Advantages Of Three Address Code

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

Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

Three Address Code corresponding to the syntax tree and DAG given above (page no:)

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

(a) Code for the syntax tree

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

(b) Code for the dag

Types of Three-Address Statements

1. Assignment statements

x := y op z, where op is a binary arithmetic or logical operation.

2. Assignment instructions

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

x : = y where the value of y is assigned to x.

4. Unconditional jump

goto L The three-address statement with label L is the next to be executed

5. Conditional jump

   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. Procedural call and return

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 x_n

call p,n

generated as part of the call procedure p(xl , *x2*, . . . , x_n) . The integer n indicating the number of actual-parameters in "call p , *n"* is not redundant because calls can be

nested.

7. Indexed Assignments

   Indexed assignments of the form x = y[i] or x[i] = y

8. Address and pointer assignments

   Address and pointer operator of the form x := &y, x := *y and *x := y

Syntax-Directed Translation Into Three-Address Code

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

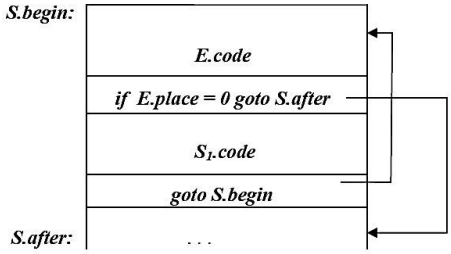
Given input a:= b * - c + b + - c, it produces the three address code in given above (page no:) The synthesized attribute S.code represents the three address code for the assignment S. The nonterminal E has two attributes:

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

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

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

Semantic rule generating code for a while statement



PRODUCTION	SEMANTIC RULES
S → while E do S_1	$S.begin := newlabel;$ $S.after := newlabel;$ $S.code := gen(S.begin ':') \parallel$ $E.code \parallel$ $gen ('if' E.place '=' '0' 'goto' S.after) \parallel$ $S_1.code \parallel$ $gen ('goto' S.begin) \parallel$ $gen (S.after ':')$

The function *newtemp* returns a sequence of distinct names t1, t2,…… in respose of successive calls. Notation *gen*(x ':= 'y '+' z is used 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 operand, like '+' are taken literally.

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

code for S, respectively.

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

Implementation 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
- Indirect triples

QUADRUPLES

- A quadruple is a record structure with four fields, which are *op, ag1, arg2* 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 *arg1*, z in *arg2* and x in *result.*

The contents of *arg1, arg2*, 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.

EXAMPLE 1

Translate the following expression to quadruple triple and indirect triple

a + b * c | e ^ f + b * a

For the first construct the three address code for the expression

t1 = e ^ f
t2 = b *
c t3 = t2 /
t1t4 = b
* a t5 =
a + t3 t6
= t5 + t4

Location	OP	arg 1	arg 2	Result
(0)	^	e	f	t1
(1)	*	b	c	t2
(2)	/	t2	t1	t3
(3)	*	b	a	t4
(4)	+	a	t3	t5
(5)	+	t3	t4	t6

Exceptions

⇨ The statement x := op y, where op is a unary operator is represented by placing op in the operator field, y in the argument field & n in the result field. The *arg2* is not used

⇨ A statement like param t1 is represented by placing param in the operator field and t1 in the arg1 field. Neither *arg2* not result field is used

⇨ Unconditional & Conditional jump statements are represented by placing the target in the result field.

TRIPLES

In triples representation, the use of temporary variables is avoided & instead reference to instructions are made

So three address statements can be represented by records with only there fields OP, arg1 & arg2.

Since, there fields are used this intermediated code formal is known as triples

Advantages

- No need to use temporary variable which saves memory as well as time

Disadvantages

- Triple representation is difficult to use for optimizing compilers
- Because for optimization statements need to be suffled.
- for e.g. statement 1 can be come down or statement 2 can go up ect.
- So the reference we used in their representation will change.

EXAMPLE 1

a + b * c | e ^ f + b * a

t1 = e ^ f t2
*= b * c t3 =*
t2 / t1t4 = b
** a t5 = a*
+ t3 t6 = t5
+ t4

Location	OP	arg 1	arg 2
(0)	^	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

EXAMPLE 2

A ternary operation like x[i] : = y requires two entries in the triple structure while x : = y[i] isnaturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

x[i] := y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

x := y[i]

INDIRECT TRIPLES

This representation is an enhancement over triple representation.

It uses an additional instruction array to led the pointer to the triples in the desired order.

- Since, it uses pointers instead of position to stage reposition the

expression to produce an optimized code.

EXAMPLE 1

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

Location	op	arg1	arg2
(0)	^	E	f
(1)	*	B	c
(2)	/	(1)	(0)
(3)	*	B	a
(4)	+	A	(2)
(5)	+	(4)	(3)

Comparison

When we ultimately produce the target code each temporary and programmer defined name will assign runtime memory location

This location will be entered into symbol table entry of that data.

Using the quadruple notation, a three address statement containing a temporary can immediately access the location for that temporary via symbol table.

But this is not possible with triples notation.

With quadruple notation, statements can often move around which makes optimization easier.

This is achieved because using quadruple notation the symbol table interposes high degree of indirection between computation of a value and its use.

With quadruple notation, if we move a statement computing x, the statement using x requires no change.

But with triples, moving a statement that defines a temporary value requires us to change all references to that statement in arg1 and arg2 arrays. This makes triples difficult to use in optimizing compiler

With indirect triples also, there is no such problem.

A statement can be moved by reordering the statement list.

Space Utilization

Quadruples and indirect triples requires same amount of space for storage

(normal case).

But if same temporary value is used more than once indirect triples can save some space. This is bcz, 2 or more entries in statement array can point to the same line of op-arg1-arg2 structure.

Triples requires less space for storage compared to above 2.

Quadruples

- direct access of the location for temporaries
- easier for optimization

Triples

- space efficiency

Indirect Triples

- easier for optimization
- space efficiency

PROBLEM 1

Translate the following expression to quadruple tuples & indirect tuples

$$a = b * - c + b * - c$$

Sol : - Three address code for given expression is

TAC

t1 = uniminus ct2 =
b* t1
t3 = uniminus
ct4 = b* t3

t5 = t2 + t4Q = t5

QUADRUPLES

Location	OP	arg1	arg2	result
(0)	uniminus	c		t1
(1)	*	b	t1	t2
(3)	uniminus	c		t3

(4)	*	b	t3	t4
(5)	+	t2	t4	t5
(6)	=	t5		a

TRIPLES

Location	OP	arg 1	arg 2
(1)	uniminus	c	
(2)	*	b	(1)
(3)	uniminus	c	
(4)	*	b	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

INDIRECT TRIPLES

	Statements
35	(1)
36	(2)
37	(3)
38	(4)
39	(5)
40	(6)

Location	**OP**	**arg1**	**arg2**
(1)	uniminus	C	
(2)	*	B	(1)
(3)	uniminus	C	
(4)	*	B	(3)
(5)	+	(2)	(4)
(6)	=	A	(5)

ASSIGNMENT STATEMENTS

Translation Scheme (SDT) To Produce Three-Address Code ForAssignments

Production Semantic action

S->id : = E	{ p : = lookup (id.name); if p ≠ nil then emit(p ‘ : =’ E.place) else error }
E->E1 + E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘ + ‘ E2.place) }
E->E1 * E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘ * ‘ E2.place) }
E->-E1	{ E.place : = newtemp; emit (E.place ‘: =’ ‘uminus’ E1.place) }
E-> (E1)	{ E.place : = E1.place }
E->id	{ p : = lookup (id.name); if p ≠ nil then E.place : = p else error }

emite generate the three address code to the output file.

newtemp return a new temporary variable.

lookup identifier check if the id is in symbol table

EXAMPLE : Annotated Parse Tree For Generation Of TAC For Assignment Statements

$x = a*b+c+d$

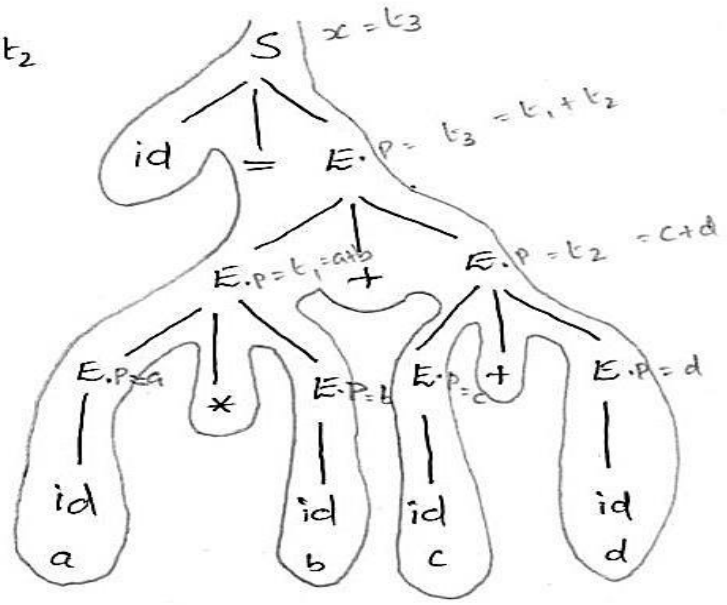
TAC

$t_1 = a * b$

$t_2 = c+d$

$t_3 = t_1 + t_2$

$x = t_3$



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

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

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes.

- They are used to compute logical values.
- But more often they are used as conditional expressions in statements that alterthe flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the Boolean operators (and, or, and not) appliedto elements that are Boolean variables or relational expressions.

Relational expressions are of the form E1 relop E2, where E1 and E2 are arithmeticexpressions and relop can be <, <=, =!, =, > or >=

Here we consider Boolean expressions generated by the following grammar :

E->E or E | E and E | note | (E) |id relop id | true
| false

Methods Of Translating Boolean Expressions

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

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

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

Method 1: Numerical Representation

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

EXAMPLE

The translation for a or b and not c will result following three-

address sequence t1 : = not c

t2 : = b

and t1 t3

: = a or

t2

Translation Scheme Using A Numerical Representation For Boolean Expression

$E \rightarrow E_1$ **or** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place* **'or'** *E_2.place) }*
$E \rightarrow E_1$ **and** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place* **'and'** *E_2.place) }*
$E \rightarrow$ **not** E_1	*{ E.place : = newtemp;* *emit(E.place ': ='* **'not'** *E_1.place) }*
$E \rightarrow (E_1)$	*{ E.place : = E_1.place }*
$E \rightarrow$ **id_1 relop id_2**	*{ E.place : = newtemp;* *emit('if'* **id_1**.*place* **relop**.*op* **id_2**.*place* **'goto'** *nextstat + 3);* *emit(E.place ': ='* **'0'** *);* *emit(***'goto'** *nextstat +2);* *emit(E.place ': ='* **'1'** *') }*
$E \rightarrow$ **true**	*{ E.place : = newtemp;* *emit(E.place ': ='* **'1'** *') }*
$E \rightarrow$ **false**	*{ E.place : = newtemp;* *emit(E.place ': ='* **'0'** *') }*

where the function emit( ) output the three address statement into the output file and nextstat( ) gives the index of the next three address statement in the output sequence and emit increments nextstat after producing each three address statement.

A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0 which can be translated into the three-address code sequence (letstatement numbers start at 100)

100 if a < b

goto 103101 t :

= 0

102 goto 104

103 t : = 1

104

Method 2: Jumping or Short-Circuit Code

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short- circuit” or “jumping” code.

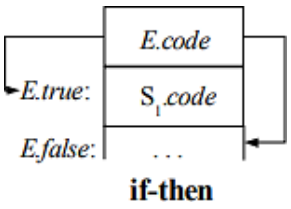
This is normally used for flow-of-control statements, such as the if-then, if-then-else and while-do statements those generated by the following grammar:

S → if E then S1

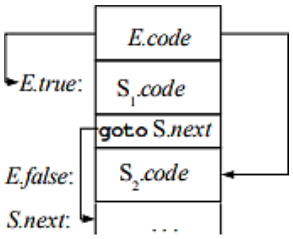
| if E then S1 else S2

| while E do S1

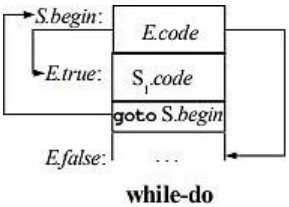
Code for if-then, if-then-else and while-do is given below:



if-then



if-then-else



while-do

Consider the grammar

S → if E then S1

| if E then S1 else S2

| while E do S1

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

With each E we associate two labels E.true and E.false. E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.

The inherited attribute S.next is a label that is attached to the first three-address instruction to be executed after the code for S and another inherited attribute S.beginis the first instruction of S

Syntax Directed Definition for flow –of –control statements

S→if E then S1	{ E.true := newlabel;E.false := S.next; S1.next := S.next; S.code := E.code || gen (E.true ‘:’) || S1.code }
S→if E then S1 else S2	{ E.true := newlabel;E.false := newlabel; S1.next := S.next; S2.next := S.next; S.code := E.code || gen (E.true ‘:’) || S1.code ||gen(‘goto’ S.next) || gen(E.false ‘:’)|| S2.code }

UNIT - V

CODE GENERATION

Issues in the design of a code generator

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

The following issue arises during the code generation phase:

Input to code generator

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

Target Program

Target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

1. Absolute machine language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.
2. Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.
3. Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

Memory Management

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

Instruction selection

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

For example, the respective three-address statements would be translated into latter code sequence as shown below: P:=Q+R

S:=P+T

```
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction costis needed in order to design good sequences, but accurate cost information is difficult to predict.

- **Register allocation issues –**

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

- **During Register allocation –**

we select only those set of variables that will reside in the registers at each point in the program.

During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increase, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain

machine requires register pairs consist of an even and next odd-numbered register. For example

```
M a, b
```

These types of multiplicative instruction involve register pairs where a, the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

- **Evaluation order –**

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

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

- Correct
- Easily maintainable
- Testable
- Maintainable

❖ **Target Machine**

A Target machine is a byte-addressable machine. This machine has n general-purpose registers, R0, R1,…..Rn-1. A Simple Target Machine Model has three-address instruction. A full-edged assembly language would have a variety of instructions. The component of instruction is an operator, followed by a target, and then followed by a list of source operands.

Some of the instructions are as follows:

- Load operations: LD dst, addr instruction loads the value in location addr into location dst. It means that assignments dst = addr. L, r, x is the general form of this instruction. The role of this instruction is to load the value in location x into register r.
- Store operations: ST r, x instruction stores the value in the location x into register r.
- Computation operations: OP, dst, src1, src2 are the form of computation operations where OP are the add or sub operator and dst, src1, and src2 are locations. The locations may or may not be distinct.
- Unconditional Operations: The instruction BR L causes control to branch to the machine instruction with label L (BR stands for the branch).
- Conditional jumps: The general form of this operation is Bcond, r, L. Here R is the register, L is a label, and cond stands for any of the common tests on the value in register r.

The various addressing modes associated with the target machine are discussed below:

- In instruction, a variable name x means there is a location in memory reserved for x.
- An indexed address in the form a(r), where 'a' is a variable and r is a register, can also be a form of a location. By taking the l-value of 'a' and adding it with the value in the register, the value of memory location denoted by a(r) can be computed.
- An integer indexed by a register can be a memory location. For example, LD R1, 100(R2) has the effect of setting R1 = contents (100 + contents (R2)).
- There are two indirect addressing modes: *r and *100(r). *r has the address of contents(r), and *100(r) has the address for adding 100 to the contents(r).
- The immediate constant addressing mode is the last addressing mode, which is denoted by prefix #.

Program and Instruction Costs

The cost refers to compiling and running a program. There are some aspects of the program on which we optimize the program. The program's cost can be determined by the compilation time's length and the size, execution time, and power consumption of the target program. Finding the actual cost of the program is a tough task. Therefore, code generation use heuristic techniques to produce a good target program. Each target-machine instruction has an associated cost. The instruction cost is one plus the cost associated with the addressing modes of the operands.

Example

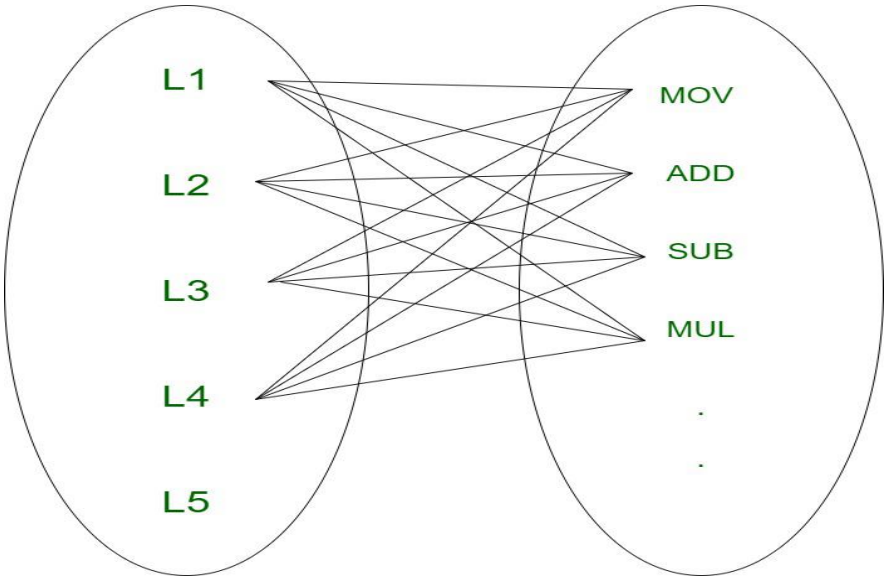
LD R0, R1: This instruction copies the contents of register R1 into register R0. The cost of this instruction is one because no additional memory is required.

LD R0, M: This instruction's role is to load the contents of memory location M into R0. So the cost will be two due to the address of memory location M is found in the word following the instruction.

LD R1, *100(R2): The role of this instruction is to load the value given by contents (contents (100 + contents (R2))) into register R1. This instruction's cost will be two due to the constant 100 is stored in the word following the instruction.

Target code generation is the final Phase of Compiler.

1. Input : Optimized Intermediate Representation.
2. Output : Target Code.
3. Task Performed : Register allocation methods and optimization, assembly level code.
4. Method : Three popular strategies for register allocation and optimization.



Computations are generally assumed to be performed on high speed memory locations, known as registers. Performing various operations on registers is efficient as registers are faster than cache memory. This feature is effectively used by compilers, However registers are not available in large amount and they are costly. Therefore we should try to use minimum number of registers to incur overall low cost.

Optimized code :

Example 1 :

L1: a = b + c * d

optimization :

t0 = c * d

a = b + t0

Example 2 :

L2: e = f - g / d

optimization :

t0 = g / d

e = f - t0

Register Allocation :

Register allocation is the process of assigning program variables to registers and reducing the number of swaps in and out of the registers. Movement of variables

across memory is time consuming and this is the main reason why registers are used as they available within the memory and they are the fastest accessible storage location.

Example 1:

R1<--- a

R2<--- b

R3<--- c

R4<--- d

MOV R3, c

MOV R4, d

MUL R3, R4

MOV R2, b

ADD R2, R3

MOV R1, R2

MOV a, R1

Example 2:

R1<--- e

R2<--- f

R3<--- g

R4<--- h

MOV R3, g

MOV R4, h

DIV R3, R4

MOV R2, f

SUB R2, R3

MOV R1, R2

MOV e, R1

Advantages :

- ✓ Fast accessible storage
- ✓ Allows computations to be performed on them
- ✓ Deterministic as it incurs no miss
- ✓ Reduce memory traffic
- ✓ Reduces overall computation time

Disadvantages :

- ✓ Registers are generally available in small amount (up to few hundred Kb)
- ✓ Register sizes are fixed and it varies from one processor to another
- ✓ Registers are complicated
- ✓ Need to save and restore changes during context switch and procedure calls.

RUNTIME STORAGE MANAGEMENT:

During the execution of a program, the same name in the source can denote different data objects in the computer. The allocation and deallocation of data objects is managed by the run-time support package . Terminologies:

• Name → storage space: the mapping of a name to a storage space is called environment .

• Storage space → value: the current value of a storage space is called its state.

• The association of a name to a storage location is called a binding. Each execution of a procedure is called an activation .

• If it is a recursive procedure, then several of its activations may exist at the same time.

• Life time: the time between the first and last steps in a procedure.

• A recursive procedure needs not to call itself directly.

General run time storage layout code static data stack heap dynamic space storage space that won't change: global data, constant, ... lower memory address higher memory address For activation records: local data, parameters, control info, ... For dynamic memory allocated by the program

Activation record

- ✓ returned value
- ✓ actual parameters
- ✓ optional control link
- ✓ optional access link
- ✓ saved machine status
- ✓ local data
- ✓ temporaries

Activation record:

Data about an execution of a procedure.

• Parameters:

. Formal parameters: the declaration of parameters.

. Actual parameters: the values of parameters for this activation.

• Links:

. Access (or static) link: a pointer to places of non-local data,

. Control (or dynamic) link: a pointer to the activation record of the caller.

Static storage allocation (1/3) There are two different approaches for run time storage allocation.

• Static allocation.

• Dynamic allocation.

• A.R. in static data area, one per procedure.

• Names bounds to locations at compiler time.

• Every time a procedure is called, its names refer to the same preassigned location.

• **Disadvantages**:

- ✓ No recursion.
- ✓ Waste lots of space when inactive.
- ✓ No dynamic allocation.

• **Advantage:**

- ✓ No stack manipulation or indirect access to names, i.e., faster in accessing variables.
- ✓ Values are retained from one procedure call to the next. For example: static variables in C.

On procedure calls:

• **The calling procedure:**

First evaluate arguments. Copies arguments into parameter space in the A.R. of called procedure. Convention: call that which is passed to a procedure arguments from the calling side, and parameters from the called side. May save some registers in its own A.R.Jump and link: jump to the first instruction of called procedure and put address of next instruction (return address) into register RA (the return address register).

• **The called procedure:**

Copies return address from RA into its A.R.'s return address field. May save some registers. May initialize local data.

On procedure returns,

• **The called procedure:**

Restores values of saved registers. Jump to address in the return address field.

• **The calling procedure:** May restore some registers. If the called procedure was actually a function, put return value in an appropriate place.

Basic Blocks and Flow Graphs

In this section, we are going to learn how to work with basic block and flow graphs in compiler design.

Basic Block

The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.

Here, the first task is to partition a set of three-address code into the basic block. The new basic block always starts from the first instruction and keep adding instructions until a jump or a label is met. If no jumps or labels are found, the control will flow in sequence from one instruction to another.

The algorithm for the construction of basic blocks is given below:

Algorithm: Partitioning three-address code into basic blocks.

Input: The input for the basic blocks will be a sequence of three-address code.

Output: The output is a list of basic blocks with each three address statements in exactly one block.

METHOD: First, we will identify the leaders in the intermediate code. There are some rules for finding leaders, which are given below:

1. The first instruction in the intermediate code will always be a leader.
2. The instructions that target a conditional or unconditional jump statement are termed as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement will be a leader.

Each leader's basic block will have all the instructions from the leader itself until the instruction, which is just before the starting of the next leader.

Example:

Consider the following source code for a 10 x 10 matrix to an identity matrix

- Instruction 2 is also a leader because this instruction is the target for instruction 11.
- Instruction 3 is also a leader because this instruction is the target for instruction 9.
- Instruction 10 is also a leader because it immediately follows the conditional goto statement.
- Similar to step 4, instruction 12 is also a leader.
- Instruction 13 is also a leader because this instruction is the target for instruction 17.

So there are six basic blocks for the above code, which are given below:

B1 for statement 1

B2 for statement 2

B3 for statement 3-9

B4 for statement 10-11

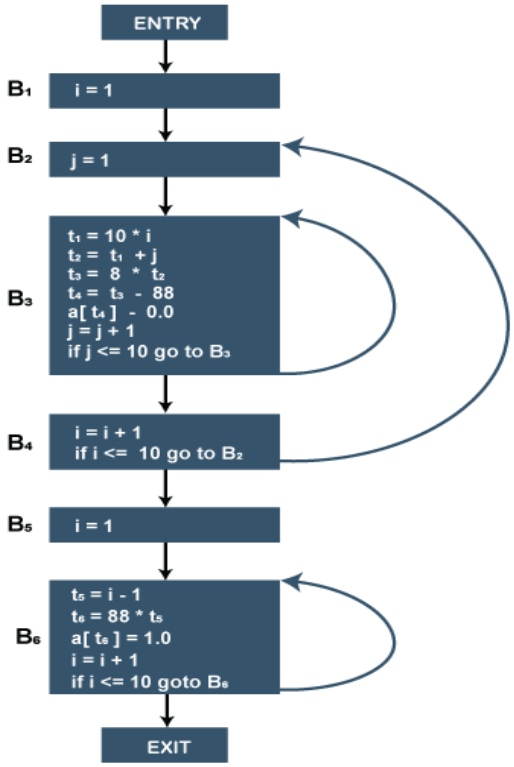
B5 for statement 12

B6 for statement 13-17.

Flow Graph

It is a directed graph. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph. An edge can flow from one block X to another block Y in such a case when the Y block's first instruction immediately follows the X block's last instruction. The following ways will describe the edge:

- There is a conditional or unconditional jump from the end of X to the starting of Y.
- Y immediately follows X in the original order of the three-address code, and X does not end in an unconditional jump.



- Flow graph for the 10 x 10 matrix to an identity matrix.
- Block B1 is the entry point for the flow graph because B1 contains starting instruction.
- B2 is the only successor of B1 because B1 doesn't end with unconditional jumps, and the B2 block's leader immediately follows the B1 block's leader.

- B3 block has two successors. One is a block B3 itself because the first instruction of the B3 block is the target for the conditional jump in the last instruction of block B3. Another successor is block B4 due to conditional jump at the end of B3 block.
- B6 block is the exit point of the flow graph.

Code Optimization in Compiler Design

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize?

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Why Optimize?

Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:

- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Types of Code Optimization –The optimization process can be broadly classified into two types :

1. **Machine Independent Optimization** – This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

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

Code Optimization is done in the following different ways :

Compile Time Evaluation :

(i) A = 2*(22.0/7.0)*r

Perform 2*(22.0/7.0)*r at compile time.

(ii) x = 12.4

y = x/2.3

Evaluate x/2.3 as 12.4/2.3 at compile time.

Variable Propagation :

//Before Optimization

c = a * b

x = a

till

d = x * b + 4

//After Optimization

c = a * b

x = a

till

d = a * b + 4

Hence, after variable propagation, a*b and x*b will be identified as common sub-expression.

Dead code elimination : Variable propagation often leads to making assignment statement into dead code

c = a * b

x = a

till

d = a * b + 4

//After elimination :

c = a * b

till

d = a * b + 4

Code Motion :

• Reduce the evaluation frequency of expression.

• Bring loop invariant statements out of the loop.

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

//This code can be further optimized as

```
a = 200;
```

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

Induction Variable and Strength Reduction :

• An induction variable is used in loop for the following kind of assignment i = i + constant.

• Strength reduction means replacing the high strength operator by the low strength.

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


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

Where to apply Optimization?

Now that we learned the need for optimization and its two types,now let's see where to apply these optimization.

Source program

Optimizing the source program involves making changes to the algorithm or changing the loop structures.User is the actor here.

Intermediate Code

Optimizing the intermediate code involves changing the address calculations and transforming the procedure calls involved. Here compiler is the actor.

Target Code

Optimizing the target code is done by the compiler. Usage of registers,select and move instructions is part of optimization involved in the target code.

Phases of Optimization

There are generally two phases of optimization:

Global Optimization:

Transformations are applied to large program segments that includes functions,procedures and loops.

Local Optimization:

Transformations are applied to small blocks of statements.The local optimization is done prior to global optimization.

PRINCIPAL SOURCES OF OPTIMISATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

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

Function preserving transformations examples:

Common sub expression elimination

Copy propagation,

Dead-code elimination

Constant folding

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

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

For example

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

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

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t5: = n

t6: = b [t1] +t5

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

Copy Propagation:

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

• For example:

x=Pi;

A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

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

Example:

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

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

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

Loop Optimizations:

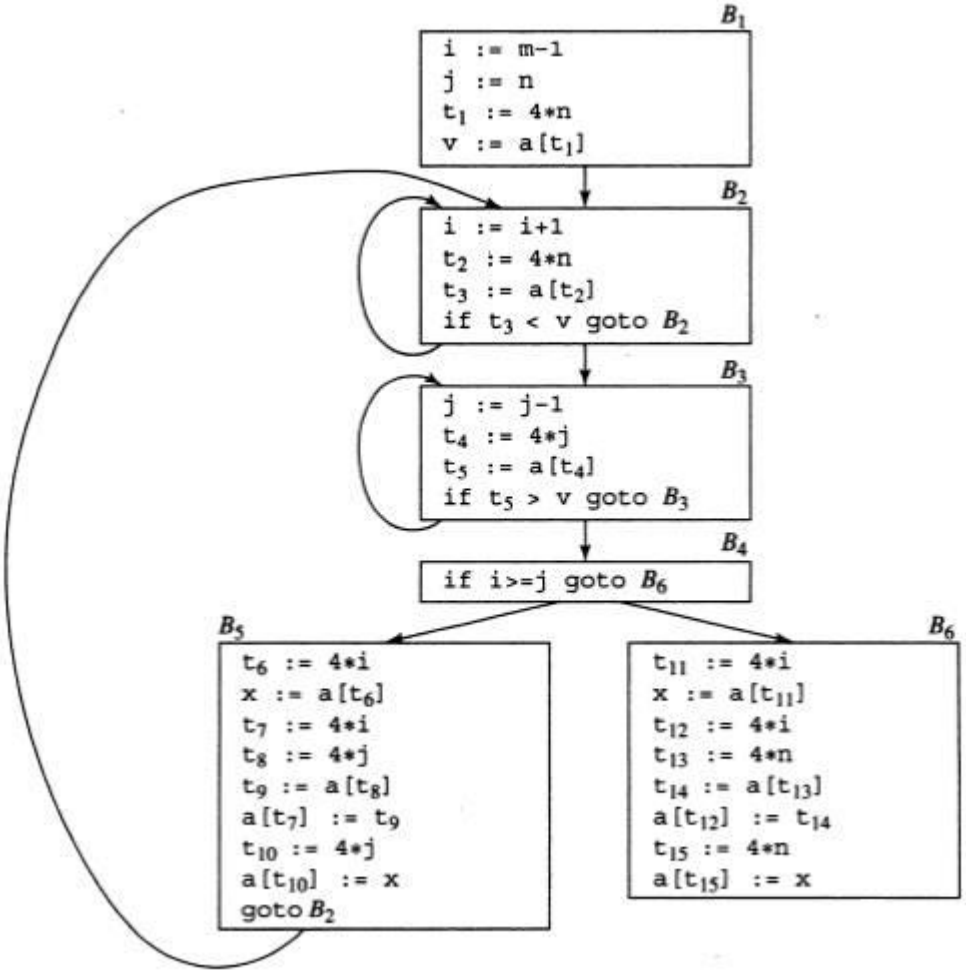
In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

Ø Code motion, which moves code outside a loop;

Ø Induction-variable elimination, which we apply to replace variables from inner loop.

Ø Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.



Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/

Code motion will result in the equivalent of

t= limit-2;

while (i<=t) /* statement does not change limit or t */

Induction Variables :

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

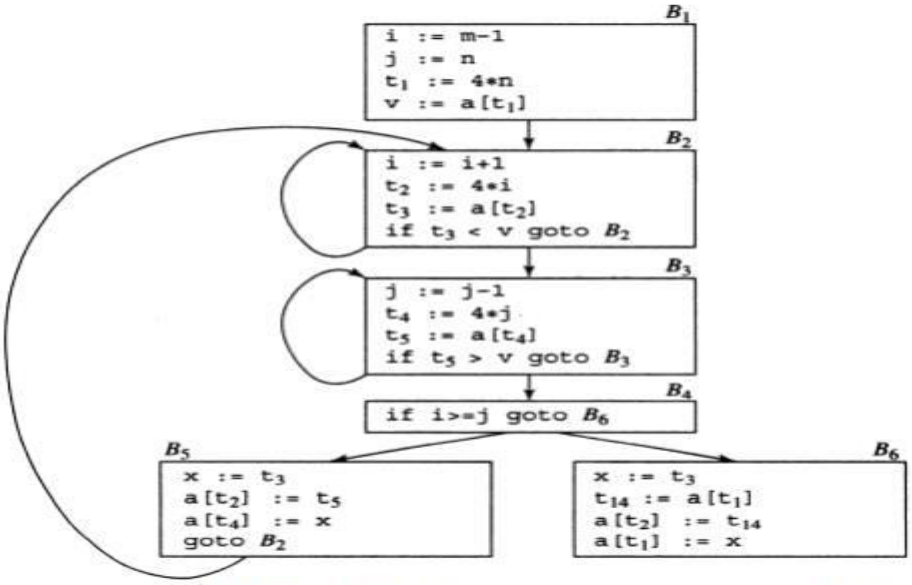


Fig. 5.3 B5 and B6 after common subexpression elimination

Optimization of Basic Blocks:

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

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

1. Structure-Preserving Transformations
2. Algebraic Transformations

1. Structure preserving transformations:

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

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements

(a) Common sub-expression elimination:

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

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

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

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

(b) Dead-code elimination

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

(c) Renaming temporary variables

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

(d) Interchange of statement

Suppose a block has the following two adjacent statements:

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

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

2. Algebraic transformations:

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

1. a:= b + c
2. e:= c +d +b

The following intermediate code may be generated:

1. a:= b + c
2. t:= c +d
3. e:= t + b