**16BECS603 COMPILER DESIGN**

**COURSE OBJECTIVES:**

- To understand and list the different stages in the process of compilation.
- To understand and design Lexical analyzers and parsers
- To Develop algorithms to generate code for a target machine
- To learn and develop techniques for optimization of code.

**COURSE OUTCOMES:**

Upon completing the course the students will be able to

- Understand the complete process of compilation from source code to target code
- develop the lexical analyzer and parsers
- Develop algorithms to generate code for a target machine
- Optimize the generated code

**UNIT 1:Introduction (9)**

Introduction - What is a Compiler? - Cousins of a Compiler- Assembler - Interpreter - Phases of compilation and overview, Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate code Generation, Code Optimization, Code Generation - Specification of Tokens.

**UNIT 2:Lexical Analysis(scanner) (9)**

Regular languages, finite automata, regular expressions, from regular expressions to finite automata, scanner generator (lex, flex).

Syntax Analysis (Parser): Context-free languages and grammars, push-down automata,LL(1)gram-mars and top-down parsing,operator grammars,LR(O),SLR(1),LR(1),LALR(1) grammars and bottom-up parsing, ambiguity and LR parsing,LALR(1) parser generator (yacc, bison)

**UNIT 3:Semantic Analysis (9)**

Attribute grammars, syntax directed definition,evaluation and flow of attribute in a syntax tree.

Symbol Table:Its structure, symbol attributes and management. Run-time environment: Procedure activation,parameter passing, value return, memory allocation, and scope.Intermediate Code Generation:Translation of different language features, different types of intermediate forms.

**UNIT 4 : Code Improvement(optimization) (9)**

Analysis: control-flow, data-flow dependence etc.; Code improvement local optimization, global optimization, loop optimization, peep-hole optimization etc.Architecture dependent code improvement: instruction scheduling(for pipeline), loop optimization (for cache memory) etc. Register allocation and target code generation

**UNIT 5 : Advanced topics (9)**

Type systems, data abstraction,compilation of Object Oriented features and non-imperativeprogramminglanguages.

**Total Hours: 45**

**TEXT BOOKS:**

1. Alfred Aho, Ravi Sethi, Jeffrey D Ullman, Compilers Principles, Techniques and Tools, Pearson Education Asia, 2nd Edition, 2017.

2. Allen I Holub, Compiler Design in C, Prentice Hall of india, 2016.

**REFERENCES:**

1. Keith Cooper and lindaTorczon, Engineering a compiler, 2nd edition, 2016.
2. Bennet.J.P, Introduction to Compiler Techniques, Tata McGraw-Hill, 2015.
3. R.Levine, Tony Mason, Doug Brown John, Lex &Yacc, 2nd Edition (October 2012) O'Reilly & Associates.
4. Kenneth c.Louden,Compiler Construction: Principles and Pratice, Thomson Learning, 2018.

**WEBSITES:**

1. http://www.tenouk.com/ModuleW.html/
2. http://www.mactech.com/articles/mactech/Vol.06/06.04/Lexical Analysis/index.html

# KARPAGAM ACADEMY OF HIGHER EDUCATION

**Coimbatore-21.**

**FACULTY OF ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

## 16BECS603- COMPILER DESIGN

## LECTURE PLAN

| S.NO | DESCRIPTION OF PORTION TO BE COVERED | HOURS | Reference Book & Page Nos. Used for teaching | TEACHING AIDS |
|---|---|---|---|---|
| 1 | Discussion on the Fundamentals of Compilers | 1 | R[1] Page no 17-33 | PPT |
| 2 | Introduction to Types of Compilers-uses of compilers | 1 | R[1] Page no 12-14 | PPT |
| | **UNIT- I INTRODUCTION TO COMPILING** | | | |
| 3 | Compilers<br>Analysis of the source program | 1 | R[2]-Page no 1.1-1.3<br>R[1]-Page no1-3 | BB |
| 4 | Phases of compiler | 1 | R[1]-Page no 4-11<br>R[2]-Page no1.6-1.10 | BB |
| 5 | Tutorial:<br>Compilers<br>Phases of compiler | 1 | R[2]-Page no 1.1-1.3<br>R[1]-Page no1-3 | PPT |
| 6 | Cousins of the compiler<br>Grouping of phases | 1 | R[2]-Page no 1.14-1.17 | BB |
| 7 | Compiler construction tools<br>Lexical Analysis | 1 | R[2]-Page no 1.19 | PPT |
| 8 | The role of the lexical analyzer | 1 | R[2]-Page no 1.25-1.28 | PPT |

| | | | | |
|---|---|---|---|---|
| 9 | Tutorial: Lexical Analysis<br>The role of the lexical analyzer | 1 | R[2]-Page no 1.25-1.28 | PPT |
| 10 | Input buffering-Tokens Specification | 1 | R[2]-Page no 1.28-1.29 | PPT |
| **TOTAL HOURS FOR UNIT-I** | | **10** | | |
| **UNIT- II SYNTAX ANALYSIS** | | | | |
| 11 | The role of the parser<br>writing a grammar | 1 | R[1]-Page no 191-195<br>R[2]-Page no 2.9-2..16 | BB |
| 12 | Context-free grammars | 1 | R[2]-Page no 197-206 | PPT |
| 13 | Tutorial hour –<br>Context-free grammars | 1 | R[2]-Page no 197-206 | PPT |
| 14 | Top-down parsing<br>Recursive-descent parser<br>Predictive parser | 1 | R[1]-Page no 217-231<br>R[2]-Page no 2.16-2.19 | BB |
| 15 | Constructing an SLR(1) parsing table | 1 | R[1]-Page no 252-248<br>R[2]-Page no 2.43 | BB |
| 16 | Bottom-up Parsing<br>Shift reduce parsing<br>Operator-precedence parsing | 1 | R[2]-Page no 2.26-2.37<br>R[1] Page no 233-240 | PPT |
| 17 | Tutorial :<br>Bottom-up Parsing<br>Shift reduce parsing | 1 | R[2]-Page no 2.26-2.37<br>R[1] Page no 233-240 | BB |
| 18 | LR Parsers<br>SLR Parser | 1 | R[1]-Page no 241-248<br>R[2]-Page no 2.39-2.42 | BB |
| 19 | Canonical LR Parser | 1 | R[1]-Page no 259-261 | PPT |

| | | | | |
|---|---|---|---|---|
| 20 | LALR Parser | 1 | R[1]-Page no 266-277<br>R[2]-Page no 2.62 | PPT |
| **TOTAL HOURS FOR UNIT-II** | | **10** | | |
| **UNIT -III INTERMEDIATE CODE GENERATION** | | | | |
| 21 | Intermediate languages | 1 | R[2]-Page no 3.1-3.6 | BB |
| 22 | Declarations<br>Assignment statements | 1 | R[1]-Page no 370-379<br>R[2] Page no 3.13-3.14 | BB |
| 23 | Boolean expressions | 1 | R[1]-Page no 399-409<br>R[2] Page no 3.24-3.26 | BB |
| 24 | Tutorial:<br>Boolean expressions | 1 | R[1]-Page no 399-409<br>R[2] Page no 3.24-3.26 | BB |
| 25 | Case statements | 1 | R[1]-Page no 418-421<br>R[2] Page no 33.31-3.32 | PPT |
| 26 | Backpatching | 1 | R[2] Page no 3.40 | BB |
| 27 | Tutorial: Case statements | 1 | R[1]-Page no 418-421<br>R[2] Page no 33.31-3.32 | PPT |
| 28 | Procedure calls | 1 | R[2] Page no 3.41-3.45 | PPT |
| 29 | Symbol table | 1 | R[2] Page no 3.41-3.45 | PPT |
| **TOTAL HOURS FOR UNIT-III** | | **9** | | |
| **UNIT- IV CODE GENERATION** | | | | |
| 30 | Issues in the design of a code generator | 1 | R[1]-Page no 501-505<br>R[2] Page no 4.2-4.3 | BB |
| 31 | The target machine | 1 | R[1]-Page no 512-516<br>R[2] Page no 4.6 | BB |

| | | | | |
|---|---|---|---|---|
| 32 | Run-time storage management | 1 | R[1]-Page no 427-440<br>R[2] Page no 4.6-4.10 | BB |
| 33 | Tutorial :<br>Run-time storage management | 1 | R[1]-Page no 427-440<br>R[2] Page no 4.6-4.10 | BB |
| 34 | Basic blocks and flow graphs<br>Next use information, a simple code generator | 1 | R[2] Page no 4.10-4.14 | BB |
| 35 | The dag representation of basic blocks | 1 | R[2] Page no 4.10-4.14 | PPT |
| 36 | Tutorial:<br>The dag representation of basic blocks | 1 | R[2] Page no 4.10-4.14 | PPT |
| 37 | Peephole optimization | 1 | R[1]-Page no 549-553<br>R[2]-Page no 4.22-4.24 | BB |
| **TOTAL HOURS FOR UNIT-IV** | | **8** | | |
| **UNIT- V CODE OPTIMIZATION AND RUN TIME ENVIRONMENTS** | | | | |
| 38 | Introduction to code optimization | 1 | R[1]-Page no 583-596<br>R[2]-Page no 5.1-5.2 | BB |
| 39 | The principle sources of optimization | 1 | R[1]-Page no 583-596<br>R[2]-Page no 5.3-5.11 | BB |
| 40 | Optimization of basic blocks<br>Global data flow analysis | 1 | R[2]-Page no 5.3-5.11 | BB |
| 41 | Tutorial<br>Code optimization | 1 | R[2]-Page no 5.3-5.11 | BB |
| 42 | Run time environment | 1 | R[2]-Page no 5.20 | PPT |
| 43 | Source Language issues | 1 | R[2]-Page no 5.20 | BB |
| 44 | Storage Organization<br>Storage Allocation strategies | 1 | R[2]-Page no 5.27-5.31 | BB |
| 45 | Tutorial :<br>Storage Organization | 1 | R[2]-Page no 5.27-5.31 | BB |

| | | | | |
|---|---|---|---|---|
| 46 | Access to non-local names, parameter missing | | R[2]-page no 5.32 | BB |
| 47 | **Discussion on Past Five Year End Semester Question Paper** 1 | | | |
| **TOTAL HOURS FOR UNIT-V** | | **10** | | |
| **TOTAL LECTURE HOURS** | | **37** | | |
| **TOTAL TUTORIAL HOURS** | | **10** | | |
| **TOTAL HOURS** | | **47** | | |

**REFERENCES**


| | |
|---|---|
| 1 | Alfred Aho, Ravi Sethi, Jeffrey D Ullman, 2006, Compilers Principles, Techniques and Tools, 4th Edition, Pearson Education Asia |
| 2 | Author: P.Kalaiselvi, Principles Of Compiler Design A.A.R.Senthikumaar, 2008, 3rd edition, charulatha publication, India |
| 3 | Allen I. Holub, 2003, Compiler Design in C, 4th Edition, Prentice Hall of India. |
| 4 | Fischer.C.N and R.J.LeBlanc, 2003, Crafting a compiler with C, 3rd Edition, Benjamin Cummings. |
| 5 | Bennet.J.P, 2003, Introduction to Compiler Techniques, 2nd Edition, Tata McGraw-Hill |


**Faculty In charge** **HOD**

# LECTURE NOTES

## CHAPTER I- LEXICAL ANALYSIS

### 1.1 INRODUCTION TO COMPILING

**Translator:**

It is a program that translates one language to another.


Figure1.1:translator

**Types of Translator:**

1.Interpreter
2.Compiler
3.Assembler

**1.Interpreter:**

It is one of the translators that translate high level language to low level language.

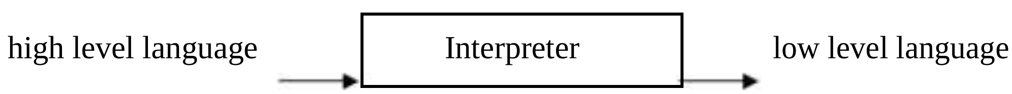



Figure 1.2: Interpreter

During execution, it checks line by line for errors.
Example: Basic, Lower version of Pascal.

**2.Assembler:**

It translates assembly level language to machine code.

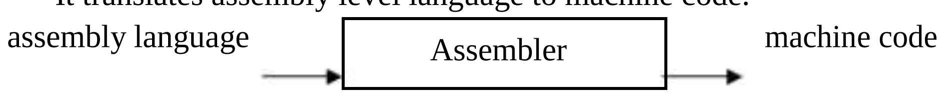



Figure 1.3:Assembler

Example: Microprocessor 8085, 8086.

**3.Compiler:**

It is a program that translates one language(source code) to another language (target code).


Figure 1.4:Compiler

It executes the whole program and then displays the errors.
Example: C, C++, COBOL, higher version of Pascal.

**Difference between compiler and interpreter:**

| Compiler | Interpreter |
|---|---|
| It is a translator that translates high level to low level language | It is a translator that translates high level to low level language |
| It displays the errors after the whole program is executed. | It checks line by line for errors. |
| Examples: Basic, lower version of Pascal. | Examples: C, C++, Cobol, higher version of Pascal. |

### 1.1.1 PARTS OF COMPILATION

There are 2 parts to compilation:
1. Analysis
2. Synthesis

**Analysis** part breaks down the source program into constituent pieces and creates an intermediate representation of the source program.

**Synthesis** part constructs the desired target program from the intermediate representation.

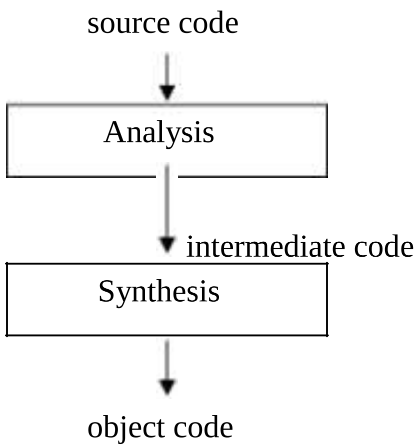



Figure 1.5:Parts of Compilation

**Software tools used in Analysis part:**

1) **Structure editor:**

Takes as input a sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. For example , it can supply key words automatically - while …. do and begin….. end.

**2) Pretty printers :**

A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font.

**3) Static checkers :**

A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program. For example, a static checker may detect that parts of the source program can never be executed.

**4) Interpreters :**

Translates from high level language ( BASIC, FORTRAN, etc..) into machine language.
An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree. Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

## 1.2 ANALYSIS OF THE SOURCE PROGRAM

Analysis consists of 3 phases:

**Linear/Lexical Analysis** :

It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning. For example, in the assignment statement a=b+c*2, the characters would be grouped into the following tokens:

i) The identifier1 ‘a’
ii) The assignment symbol (=)
iii) The identifier2 ‘b’
iv) The plus sign (+)
v) The identifier3 ‘c’
vi) The multiplication sign (*)
vii) The constant ‘2’

**Syntax Analysis** :

It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. They are represented using a syntax tree as shown below:

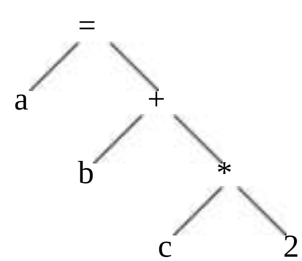


Figure 1.6:Syntax Analysis

A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands. This analysis shows an error when the syntax is incorrect.

**Semantic Analysis** :

It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase. It uses the syntax tree to identify the operators and operands of statements. An important component of semantic analysis is **type checking**. Here the compiler checks that each operator has operands that are permitted by the source language specification.

## 1.3 PHASES OF COMPILER

A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

**Main phases**:
1) Lexical analysis
2)Syntax analysis
3)Semantic analysis
4) Intermediate code generation
5)Code optimization
6)Code generation

**Sub-Phases**:
1)Symbol table management
2)Error handling

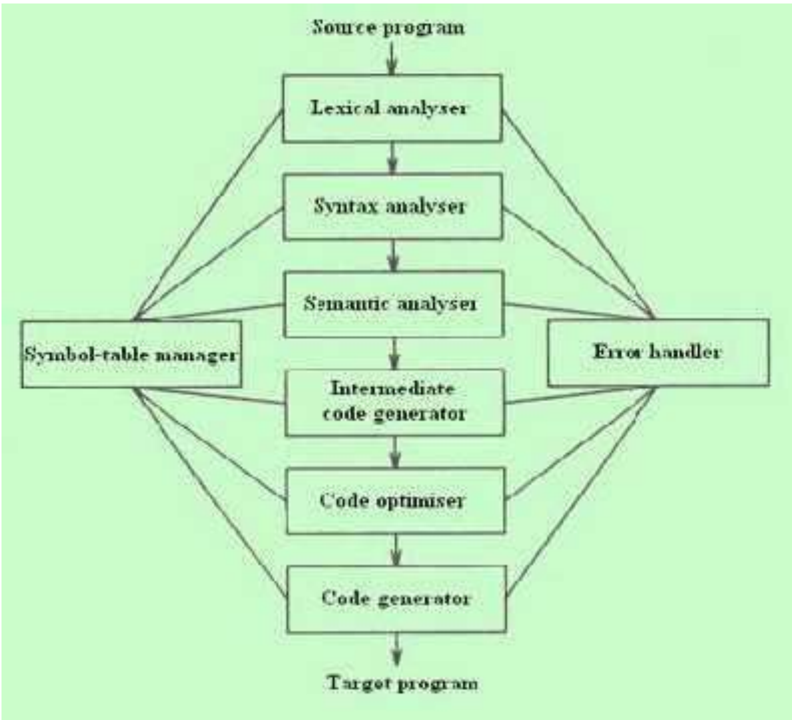



Figure 1.7:Phases of Compiler

**LEXICAL ANALYSIS:**

It is the first phase of the compiler. It gets input from the source program and produces tokens as output. It reads the characters one by one, starting from left to right and forms the tokens.

**Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: a +b =20

Here, a,b,+,=,20 are all separate tokens.

Group of characters forming a token is called the **Lexeme**.

The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

**SYNTAX ANALYSIS:**

It is the second phase of the compiler. It is also known as parser. It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.

Syntax tree: It is a tree in which interior nodes are operators and exterior nodes are operands. Example: For a=b+c*2, syntax tree is

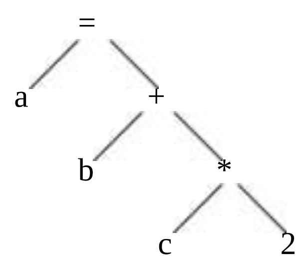



Figure 1.8: Syntax Tree

**SEMANTIC ANALYSIS:**

It is the third phase of the compiler. It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not. It performs type conversion of all the data types into real data types.

**INTERMEDIATE CODE GENERATION:**

It is the fourth phase of the compiler. It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code. The three-address code consists of a sequence of instructions, each of which has atmost three operands.

Example: t1=t2+t3

**CODE OPTIMIZATION:**

It is the fifth phase of the compiler. It gets the intermediate code as input and produces optimized intermediate code as output. This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result. During the code optimization, the result of the program is not affected. To improve the code generation, the optimization involves,

- deduction and removal of dead code (unreachable code).
- calculation of constants in expressions and terms.
- collapsing of repeated expression into temporary string.
- loop unrolling.
- moving code outside the loop.
- removal of unwanted temporary variables.

**CODE GENERATION:**

It is the final phase of the compiler. It gets input from code optimization phase and produces the target code or object code as result.Intermediate instructions are translated into a sequence of machine instructions that perform the same task. The code generation involves

- allocation of register and memory
- generation of correct references
- generation of correct data types
- generation of missing code

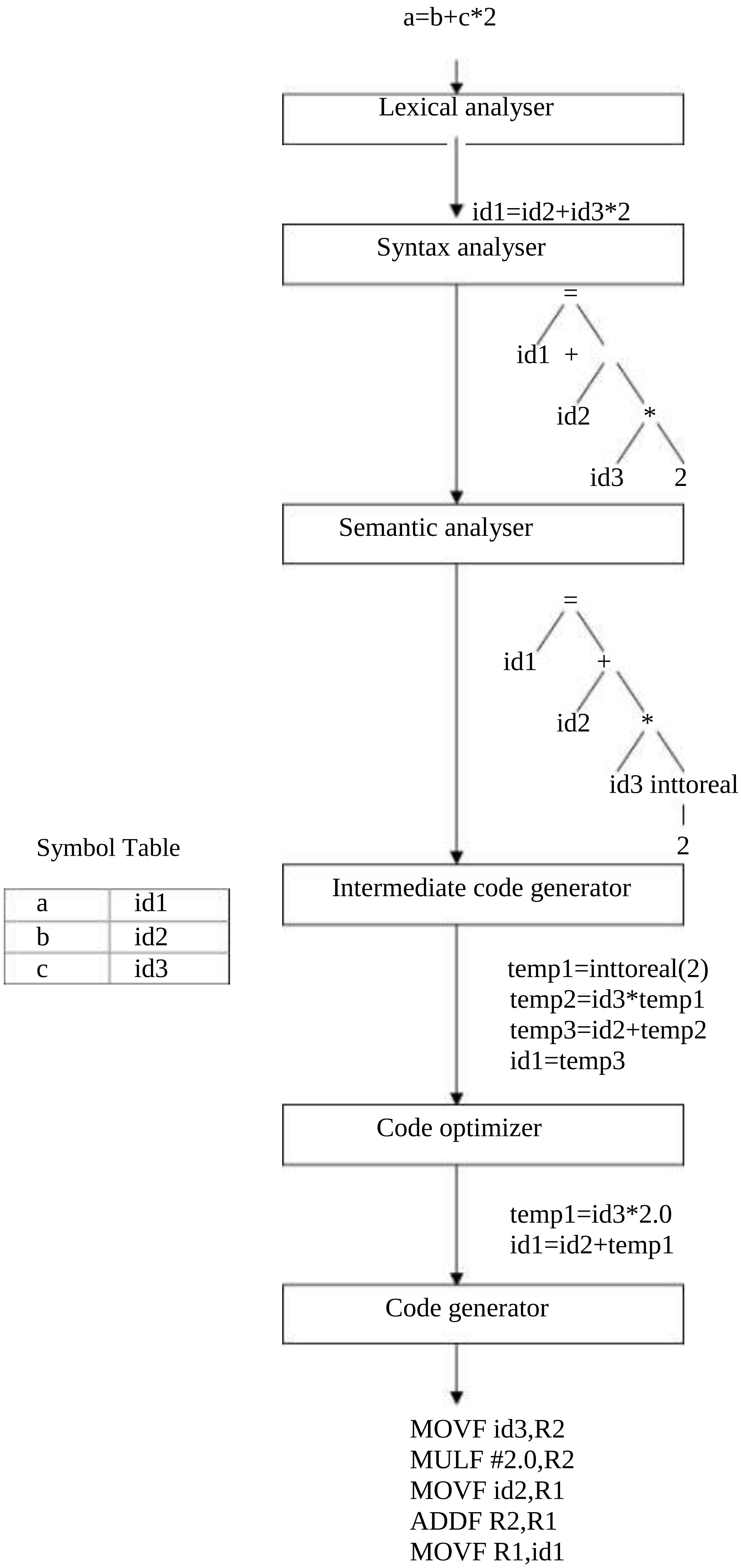
**SYMBOL TABLE MANAGEMENT:**

Symbol table is used to store all the information about identifiers used in the program. It is a data structure containing a record for each identifier, with fields for the attributes of the identifier. It allows to find the record for each identifier quickly and to store or retrieve data from that record. Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

**ERROR HANDLING:**

Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed. In lexical analysis, errors occur in separation of tokens. In syntax analysis, errors occur during construction of syntax tree. In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and duringtype conversion. In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2. The figure shows the representation of this statement after each phase:


a=b+c*2
Lexical analyser
id1=id2+id3*2
Syntax analyser
=
id1 +
id2 *
id3 2
Semantic analyser
=
id1 +
id2 *
id3 inttoreal
2
Symbol Table
a id1
b id2
c id3
Intermediate code generator
temp1=inttoreal(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
Code optimizer
temp1=id3*2.0
id1=id2+temp1
Code generator
MOVF id3,R2
MULF #2.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1

## 1.4 COUSINS OF COMPILER

1. Preprocessor
2. Assembler
3. Loader and Link-editor

### 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.

### ASSEMBLER

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.

### LINKER AND LOADER

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are :

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

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.

## 1.5 GROUPING OF THE PHASES

Compiler can be grouped into front and back ends:

- **Front end**: analysis (machine independent)

These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

- **Back end**: synthesis (machine dependent)

It includes code optimization phase and code generation along with the necessary error handling and symbol table operations.

### Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

## 1.6 COMPILER CONSTRUCTION TOOLS

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler construction tools:

1) **Parser Generators:**

-These produce syntax analyzers, normally from input that is based on a context-free grammar.

-It consumes a large fraction of the running time of a compiler. -

Example-YACC (Yet Another Compiler-Compiler).

2) **Scanner Generator:**

-These generate lexical analyzers, normally from a specification based on regular expressions. -The basic organization of lexical analyzers is based on finite automation.

3) **Syntax-Directed Translation:**

-These produce routines that walk the parse tree and as a result generate intermediate code. -Each translation is defined in terms of translations at its neighbor nodes in the tree.

4) **Automatic Code Generators:**

-It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

5) **Data-Flow Engines:**

-It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

## 1.7 LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

### 1.7.1 THE ROLE OF THE LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

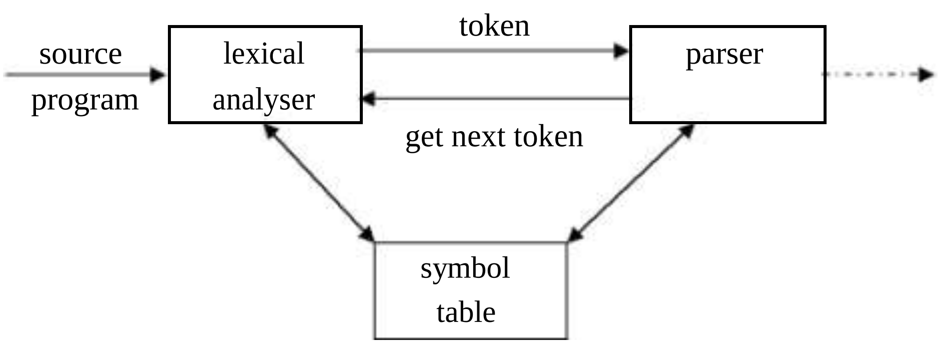



Figure 1.10:Role of Lexical Analyzer

Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

### 1.7.2 ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

1. To make the design simpler.
2. To improve the efficiency of the compiler.
3. To enhance the computer portability.

### 1.7.3 TOKENS

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming 10language: sum=3+2;

Table 1.1:Tokens

| Lexeme | Token type |
|---|---|
| sum | Identifier |
| = | Assignment operator |
| 3 | Number |
| + | Addition operator |
| 2 | Number |
| | End of statement |

**LEXEME:**

Collection or group of characters forming tokens is called Lexeme.

**PATTERN:**

A pattern is a description of the form that the lexemes of a token may take.In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

### 1.7.4 Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant
ii) Identifiers: pointer to the corresponding symbol table entry.

### 1.7.5 ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

The following are the error-recovery actions in lexical analysis:

1)Deleting an extraneous character.

2) Inserting a missing character.

3)Replacing an incorrect character by a correct character.

4)Transforming two adjacent characters.

**5)Panic mode recovery**: Deletion of successive characters from the token until

error is resolved.

## 1.8 INPUT BUFFERING

We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:

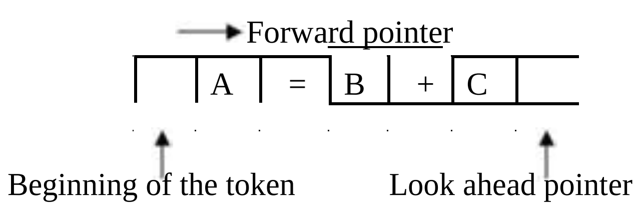



Figure1.11:Input Buffering

We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

### 1.8.1 BUFFER PAIRS

A buffer is divided into two N-character halves, as shown below

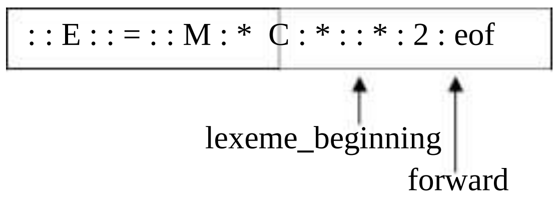



Figure 1.12: Buffer Pair

Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes. Using one system read command we can read N characters into a buffer. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file. Two pointers to the input are maintained:

1. Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found. Once the next lexeme is determined, forward is set to the character at its right end.

The string of characters between the two pointers is the current lexeme. After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found.

**Advancing forward pointer:**

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

***Code to advance forward pointer:***

```
if forward at end of first half then begin
    reload second half;
    forward := forward + 1
end
else if forward at end of second half then
    begin reload second half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

**SENTINELS**

For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.

The sentinel arrangement is as shown below:

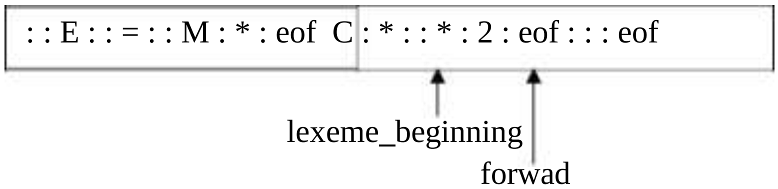



Figure 1.13:Sentinels

Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

***Code to advance forward pointer:***

```
forward : = forward + 1; if
forward ↑ = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1
   end
   else if forward at end of second half then
       begin reload first half;
       move forward to beginning of first
   half end
else /* eof within a buffer signifying end of input
       */ terminate lexical analysis
end
```

## 1.9 SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1) Strings
2) Language
3)Regular expression

**Strings and Languages**

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

**Operations on strings**

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.For example, ban is a prefix of banana.
2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.
3. A **substring** of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.
4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.

**Operations on languages:**

The following are the operations that can be applied to languages:

1.Union
2.Concatenation
3.Kleene closure
4.Positive closure

The following example shows the operations on strings: Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : $L^*$ ={ ε,0,1,00….}
4. Positive closure : $L^+$={0,1,00….}

**Regular Expressions**

Each regular expression r denotes a language L(r).Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If'a'is a symbol in Σ, then 'a'is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a'in its one position.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

   a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
   b) (r)(s) is a regular expression denoting the language L(r)L(s).
   c) (r)* is a regular expression denoting (L(r))*.
   d) (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. It| has lowest precedence and is left associative.

**Regular set**

A language that can be defined by a regular expression is called a regular set.
If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.
For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

**Regular Definitions**

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_l \rightarrow r_1$ $d_2$
$\rightarrow r_2$
………
$d_n \rightarrow r_n$

1. Each $d_i$ is a distinct name.
2. Each $r_i$ is a regular expression over the alphabet Σ U {$d_l$, $d_2$,. . . , $d_{i-l}$}.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:
letter → A | B | …. | Z | a | b | …. | z |
digit → 0 | 1 | …. | 9
id → letter ( letter | digit ) *

**Shorthands**

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

**1. *One or more instances (+)*:**

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $( r )^+$ is a regular expression that denotes the language $(L ( r ))^+$
- Thus the regular expression $a^+$ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator $^*$.

**2. *Zero or one instance ( ?)*:**

- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then ( r )? is a regular expression that denotes the language L( r ) U { ε }.

**3. *Character Classes*:**

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.

- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.

- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*

**Non-regular Set**

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

**<u>RECOGNITION OF TOKENS</u>**

Consider the following grammar fragment:

stmt → if expr then stmt
|if expr then stmt else stmt
|ε

expr → term relop term
|term

term → id
|num

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

If → if
Then → then
Else → else
Relop → <|<=|=|<>|>|>=
Id → letter(letter|digit)
Num → digit$^+$ (.digit$^+$)?(E(+|-)?digit$^+$)?

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

**Transition diagrams**

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

**Transition diagram for relational operators**

Figure 1.14:Transition Diagram

## 1.10 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- Lex
- YACC

**LEX**

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

**Creating a lexical analyzer**

First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c. Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

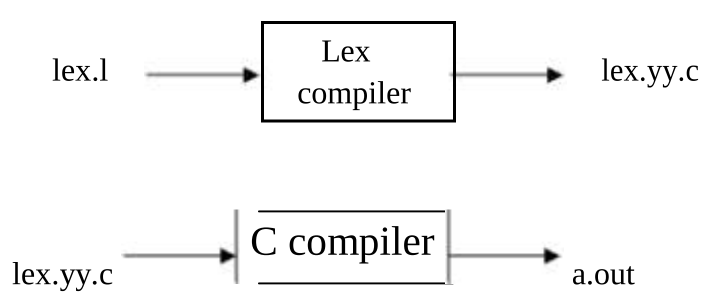



Figure 1.15: Creating Lexical Analyzer

**Lex Specification**

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions

- **Rules** are statements of the form
  $p_1$ {$action_1$}
  $p_2$ {$action_2$} …
  $p_n$ {$action_n$}
  where $p_i$ is regular expression and $action_i$ describes what action the lexical analyzer should take when pattern $p_i$ matches a lexeme. Actions are written in C code.

- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

**YACC- YET ANOTHER COMPILER-COMPILER**

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

## 1.11 FINITE AUTOMATA

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

**Types of Finite Automata**

There are tow types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

### 1.11.1 Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by
$M = \{Q_n, \Sigma, \delta, q_0, f_n\}$
$Q_n$ – finite set of states
$\Sigma$ – finite set of input symbols
$\delta$ – transition function that maps state-symbol pairs to set of states

$q_0$ – starting state

$f_n$ – final state

### 1.11.2 Deterministic Finite Automata

DFA is a special case of a NFA in which

i) no state has an ε-transition.

ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

$M = \{Q_d, \Sigma, \delta, q_0, f_d\}$

$Q_d$ – finite set of states

Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states

$q_0$ – starting state

$f_d$ – final state

### 1.11.3 Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:

i) Convert RE to NFA using Thomson's rules

ii) Convert NFA to DFA

iii) Construct minimized DFA

**UNIT-II**

# CHAPTER-II

# SYNTAX ANALYSIS AND RUNTIME ENVIRONMENT

## 2.1 SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

**Advantages of grammar for syntactic specification :**

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

### 2.1.1 THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

***Position of parser in compiler model***

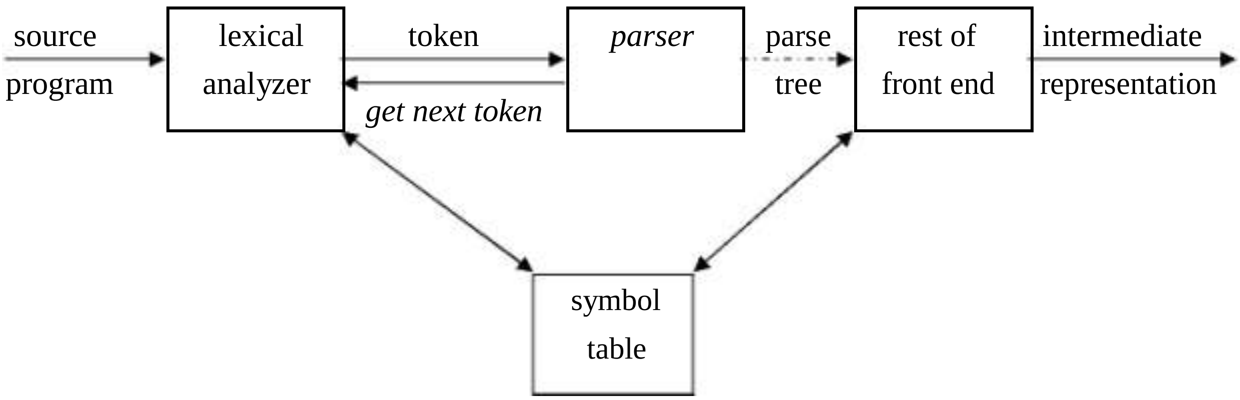



Figure 2.1:Role of Parser

**Functions of the parser:**

5) It verifies the structure generated by the tokens based on the grammar.
6) It constructs the parse tree.
7) It reports the errors.
8) It performs error recovery.

**Issues :**

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

**Syntax error handling :**

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

**2.1.2 Error recovery strategies:**

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

**Panic mode recovery:**

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

**Phrase level recovery:**

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

**Error productions:**

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

**Global correction:**

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

## 2.2 CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

**Terminals :** These are the basic symbols from which strings are formed.

**Non-Terminals :** These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

**Start Symbol :** One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

**Productions :** It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

**Example of context-free grammar:** The following grammar defines simple arithmetic expressions:

*expr* → *expr op expr*
*expr* → (*expr*)
*expr* → - *expr*
*expr* → **id** *op*
→ + *op* → -

*op* → *
*op* → /
*op* → ↑

In this grammar,

- **id** + - * / ↑( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

### 2.2.1 Derivations:

Two basic requirements for a grammar are :
 To generate a valid string.
 To recognize a valid string.

**Derivation** is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

**Example :** Consider the following grammar for arithmetic expressions :

E → E+E |E*E |( E ) | - E | id

To generate a valid string - (id+id ) from the grammar the steps are

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

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

**Types of derivations:**

The two types of derivation are:

Left most derivation
Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

**Example:**

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

Sentence to be derived : – (id+id)

| LEFTMOST DERIVATION | RIGHTMOST DERIVATION |
|---|---|
| E → - 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. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

**Ambiguity:**

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

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 :


**2.3 WRITING A GRAMMAR**

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
- Eliminating ambiguous grammar.
- Eliminating left-recursion
- Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

### 2.3.1 Regular Expressions vs. Context-Free Grammars:

| REGULAR EXPRESSION | CONTEXT-FREE GRAMMAR |
|---|---|
| It is used to describe the tokens of programming languages. | It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal. |
| It is used to check whether the given input is valid or not using **transition diagram.** | It is used to check whether the given input is valid or not using **derivation**. |
| The transition diagram has set of states and edges. | The context-free grammar has set of productions. |
| It has no start symbol. | It has start symbol. |
| It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth. | It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on. |

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

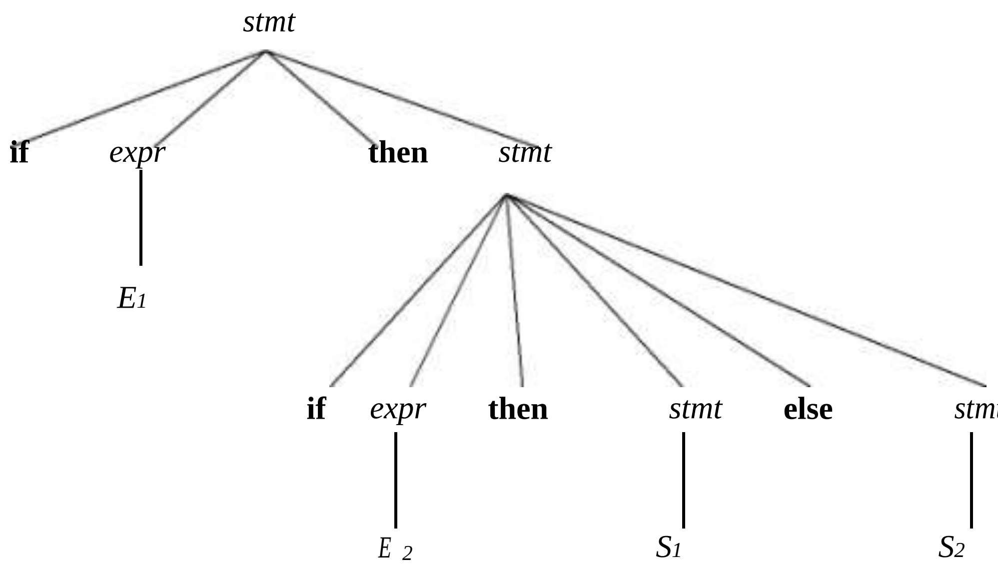
**Eliminating ambiguity:**

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

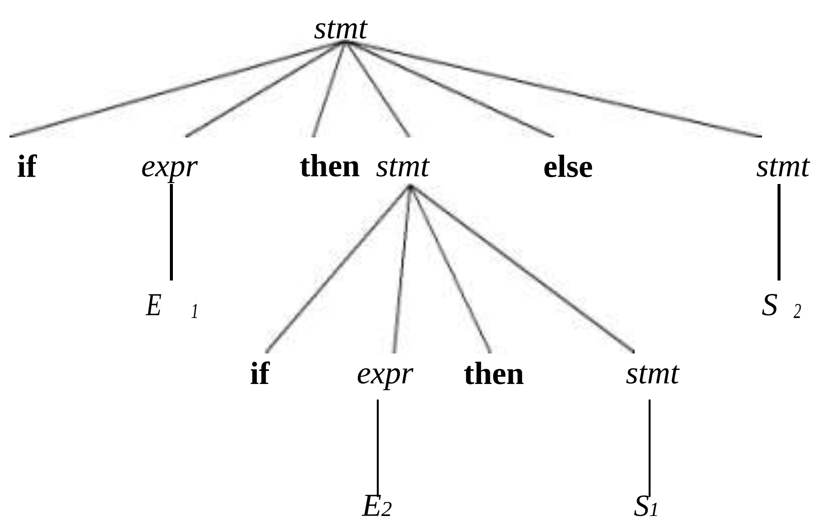
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if** $E_1$ **then if** $E_2$ **then** $S_1$ **else** $S_2$ 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*

### 2.3.2 Eliminating Left Recursion:

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

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

**A → βA'**

**A'→ αA' | ε** without

changing the set of strings derivable from A.

**Example** : Consider the following grammar for arithmetic expressions:

E → E+T |T

T → T*F |F

F→ (E) |id

First eliminate the left recursion for E

as E → TE'

E' → +TE' |ε

Then eliminate for T

as T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion

is E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) |id

**Algorithm to eliminate left recursion:**

1. Arrange the non-terminals in some order $A_1$, $A_2$ . . . $A_n$.
2. **for** *i* := 1 **to** *n* **do begin**
    **for** *j* := 1 **to** *i*-1 **do begin**
        replace each production of the form $A_i \rightarrow A_j \gamma$ by
            the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current $A_j$-productions;
    **end**
    eliminate the immediate left recursion among the $A_i$-productions
  **end**

### 2.3.3 Left factoring:

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

**If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ , it can be rewritten as**

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar , G : $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

Left factored, this grammar becomes

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

### PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

**Parse tree:**

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

**Types of parsing:**

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.
  Example : LL Parsers.
- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
  Example : LR Parsers.

### 2.4 TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt 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

### 2.4.1 RECURSIVE DESCENT PARSING

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

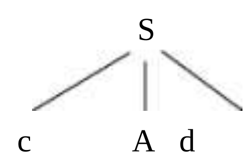
**Example for backtracking** :

Consider the grammar G : S → cAd
A → ab |a
and the input string w=cad.

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

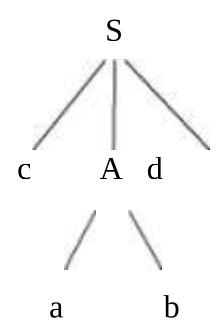
**Step1:**

Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol 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.



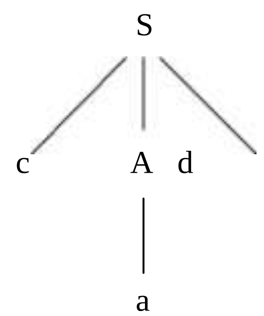

**Step3:**

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.

**<u>Example for recursive decent parsing</u>:**

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 expressions

E → E+T |T

T → T*F |F

F→ (E) |id

After eliminating the left-recursion the grammar

becomes, E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) |id

Now we can write the procedure for grammar as follows:

**<u>Recursive procedure</u>:**

```
Procedure E()
begin
        T( );
        EPRIME( );
end
```

```
Procedure EPRIME(
) begin
        If input_symbol='+'
        then ADVANCE( );
        T( );
        EPRIME( );
end

Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME(
) begin
        If input_symbol='*'
        then ADVANCE( );
        F( );
        TPRIME( );
end

Procedure F( )
begin
        If input-symbol='id' then
        ADVANCE( );
        else if input-symbol='('
        then ADVANCE( );
        E( );
        else if input-symbol=')'
        then ADVANCE( );
end

else ERROR( );
```

**Stack implementation:**

To recognize input **id+id*id :**

Table 2.1: Stack implementation

| PROCEDURE | INPUT STRING |
|---|---|
| E( ) | **id**+id*id |
| T( ) | **id**+id*id |
| F( ) | **id**+id*id |
| ADVANCE( ) | id**+**id*id |

| | |
|---|---|
| | 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( ) | |

### 2.4.2 PREDICTIVE PARSING

2.5 Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

2.6 The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

**Non-recursive predictive parser**

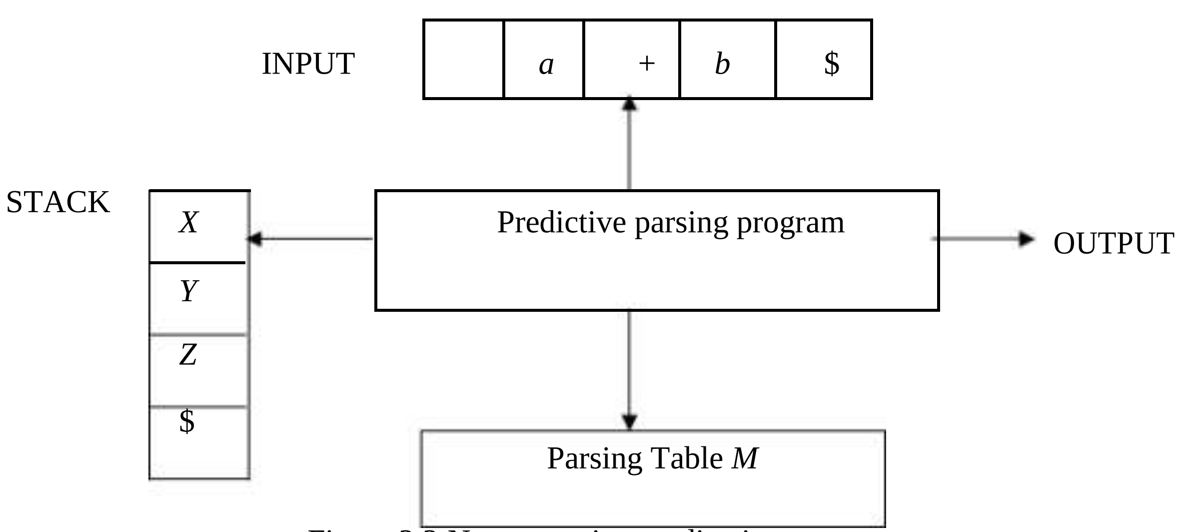



Figure 2.2:Non-recursive predicative parser

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

**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 anon-terminal and **'*a*'** is aterminal.

**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 to the next input symbol.
3. If *X* is a non-terminal , the program consults entry *M*[*X*, *a*] of the parsing table *M*. This entry will either be an *X*-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces *X* on top of the stack by *WVU*. 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 parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
      let X be the top stack symbol and a the symbol pointed to by ip;
      if X is a terminal or $ then
            if X = a then
                  pop X from the stack and advance ip
            else error()
      else          /* X is a non-terminal */
            if M[X, a] = X →Y1Y2 … Yk then begin
```

```
                pop X from the stack;
                push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
                output the production X → Y1 Y2 . . . Yk
            end
            else error()
    until X = $            /* stack is empty */
```

**Predictive parsing table construction:**

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

1. FIRST

2. FOLLOW

**Rules for first( ):**

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

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST($X$).

3. If $X$ is non-terminal and $X \rightarrow a\alpha$ is a production then add $a$ to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then place $a$ in FIRST($X$) if for some $i$, $a$ is in FIRST($Y_i$), and ε is in all of FIRST($Y_1$),…,FIRST($Y_{i-1}$); that is, $Y_1, \ldots . Y_{i-1} \Rightarrow \varepsilon$. 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 in follow($B$).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW($A$) is in FOLLOW($B$).

**Algorithm for construction of predictive parsing table:**

**Input** : Grammar *G*

**Output** : Parsing table *M*

**Method** :

1. For each production $A \rightarrow \alpha$ 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 $A \rightarrow \alpha$ to $M[A, b]$ for each terminal $b$ in FOLLOW($A$). If ε is in FIRST(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 is

E → TE'
E' → +TE' |ε
T → FT'
T' → *FT' | ε
F → (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) ={+, * , $ , ) }

**Table 2.2: Predictive parsEr**

| 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) | | |

**Table 2.3:Stack implem**

| 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 have

S → iEtSS' |a
S'→ eS | ε
E → b

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

FIRST(S) ={ i, a }

FIRST(S') = {e, ε }

FIRST(E) ={ b}

FOLLOW(S) ={ $ ,e }

FOLLOW(S’) = { $ ,e }

FOLLOW(E) = {t}

**Table 2.4:Parsing table**

| NON-TERMINAL | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | S → a | | | S → iEtSS’ | | |
| S’ | | | S’→ eS<br>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.

## 2.5 BOTTOM-UP PARSING

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

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

### 2.5.1 SHIFT-REDUCE PARSING

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

**Example:**

Consider the grammar:

S → aABe

A → Abc | b

B→ d

The 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 $w = \gamma_n$, where $\gamma_n$ is the $n^{th}$ right-sentinel form of some rightmost derivation.

**Table 2.5: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 recovery routine.

**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 by E→id |
| $ E*id | $ | Reduce by E→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. <u>Reduce-reduce conflict:</u>**

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 by R→c | $ c | +c $ | Reduce by R→c |
| $ R | +c $ | Shift | $ R | +c $ | Shift |
| $ R+ | c $ | Shift | $ R+ | c $ | Shift |
| $ R+c | $ | Reduce by R→c | $ R+c | $ | Reduce by M→R+c |
| $ R+R | $ | Reduce by M→R+R | $ M | $ | |
| $ M | $ | | | | |

**Viable prefixes:**

- $\alpha$ is a viable prefix of the grammar if there is $w$ such that $\alpha w$ 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.

## 2.5.2 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 $\varepsilon$ or has two adjacent non-terminals.

**Example:**

Consider the grammar:

$E \rightarrow EAE \mid (E) \mid -E \mid id$
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

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

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid -E \mid id$

**Operator precedence relations:**

There are three disjoint precedence relations

namely $<\cdot$ - less than
$=$ - equal to
$\cdot>$ - greater than

The relations give the followingmeaning:

$a <\cdot b$ – a yields precedence to b
$a=b$ – a has the same precedence as b
$a \cdot> b$ – a takes precedence over b

**Rules for binary operations:**

- If operator $\theta_1$ has higher precedence than operator $\theta_2$, then

  make $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

- If operators $\theta_1$ and $\theta_2$, 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

- Make the following for all operators $\theta$:

  $: \ <\cdot id \ , \ id \cdot> \theta$
  $: \ <\cdot ( , \ ( <\cdot \theta$
  $) \cdot> \theta \ , \ \theta \cdot> )$
  $\theta \cdot> \$ \ , \ \$ <\cdot \theta$

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 all interior nodes; otherwise, an error indication.
**Method :** Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute the 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**
4. let *a* be the topmost terminal symbol on the stack
and let *b* be the symbol pointed to by *ip;*
5. **if** *a* <· *b* or *a* = *b* **then begin**
6. push *b* onto the stack;
7. advance *ip* to the next input symbol;
**end;**

```
else if a ·> b then      /*reduce*/
    repeat
        pop the stack
    until the top stack terminal is related by <·
        to the terminal most recently popped
  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** .The implementation 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:**

3. It is easy to implement.
4. 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:**

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

## 2.6 LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. 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 be written.
- ✓ 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 that can be parsed with predictive parser.
- ✓ It detects asyntactic error as soon as possible.

**Drawbacks of LR method:**

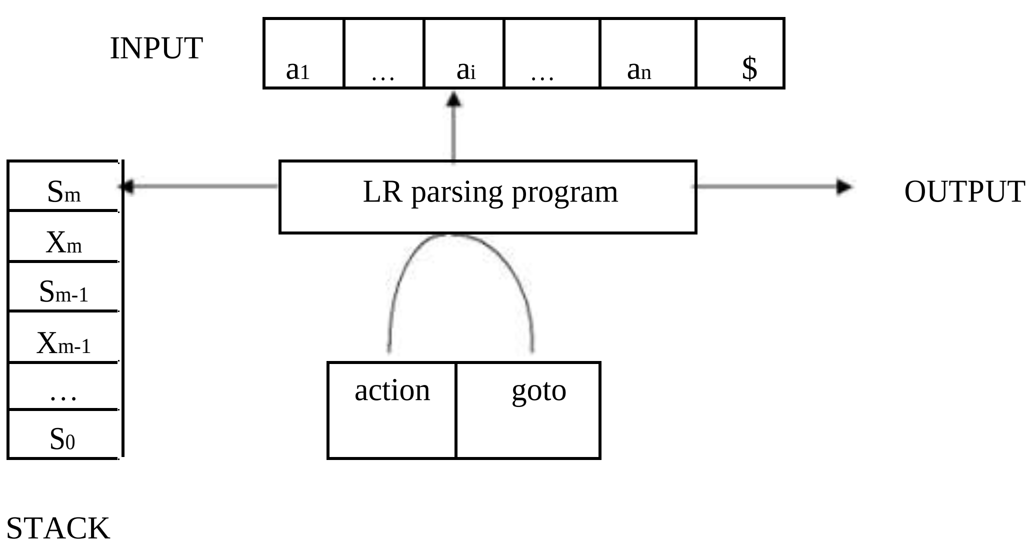
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

**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:



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

- ❖ The driver program is the same for all LR parser.
- ❖ The parsing 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 on top. 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 $a_i$, the current input symbol. It then consults *action*[$s_m$,$a_i$] in the action table which can have one of four values :

- shift s, where s is a state,
- reduce by a grammar production A → β,
- accept, and
- 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;
           let s' be the state now on top of the stack;
           push A then goto[s', A] on top of the
           stack; output the production A→ β
      end
      else if action[s, a]=accept then
           return
      else error( )
end
```

## 2.7 CONSTRUCTING SLR(1) PARSING TABLE:

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

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

**LR(0) items:**

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

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

**Closure operation:**

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

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

**Goto operation:**

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

Steps to construct SLR parsing table for grammar G are:

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

**Algorithmfor 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 ={$I_0$, $I_1$, .... $I_n$}, 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→α·*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→α·] is in $I_i$ , then set *action*[*i*,*a*] to "reduce A→α" 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*.

- Σ All entries not defined by rules (2) and (3) are made “error”
- Σ The initial state of the parser is the one constructed from the set of items containing [S’→.S].

**<u>Example for SLR parsing:</u>**

Construct SLR parsing for the following grammar :

G : E → E + T | T
 T → T * F | F
 F→ (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 * F
T → F
F→ (E)
F→ id

**Step 2 :** Find LR (0) items.

$I_0$ : E’ → . E
 δ → . E + T
 iv) → . T
 T → . T * F
 T → . F
 F → . (E)
 F → . id

<u>GOTO ( $I_0$ , E)</u>

$I_1$ : E’ → E .
 E → E . + T

<u>GOTO ( $I_4$ , id )</u>

$I_5$ : F→ id .

GOTO ( $I_0$ , T)

$I_2 : E \rightarrow T .$

$T \rightarrow T . * F$

GOTO ( $I_0$ , F)

$I_3 : T \rightarrow F .$

GOTO ( $I_0$ , ( )

$I_4 : F \rightarrow ( . E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO ( $I_0$ , id )

$I_5 : F \rightarrow id .$

GOTO ( $I_1$ , + )

$I_6 : E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO ( $I_2$ , * )

$I_7 : T \rightarrow T * . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO ( $I_4$ , E )

$I_8 : F \rightarrow ( E . ) E$

$\rightarrow E . + T$

GOTO ( $I_4$ , T)

$I_2 : E \rightarrow T .$

$T \rightarrow T . * F$

GOTO ( $I_4$ , F)

$I_3 : T \rightarrow F .$

GOTO ( $I_6$ , T )

$I_9 : E \rightarrow E + T .$

$T \rightarrow T . * F$

GOTO ( $I_6$ , F )

$I_3 : T \rightarrow F .$

GOTO ( $I_6$ , ( )

$I_4 : F \rightarrow ( . E )$

GOTO ( $I_6$ , id)

$I_5 : F \rightarrow id .$

GOTO ( $I_7$ , F )

$I_{10} : T \rightarrow T * F .$

GOTO ( $I_7$ , ( )

$I_4 : F \rightarrow ( . E )$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO ( $I_7$ , id )

$I_5 : F \rightarrow id .$

GOTO ( $I_8$ , ) )

$I_{11} : F \rightarrow ( E ) .$

GOTO ( $I_8$ , + )

$I_6 : E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . ( E )$

$F \rightarrow . id$

GOTO ( $I_9$ , *)

$I_7 : T \rightarrow T * . F$

$F \rightarrow . ( E )$

$F \rightarrow . id$

GOTO ( $I_4$ , ()

$I_4$ : F → ( . E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

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

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

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

**SLR parsing table:**

| | ACTION | | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** | **E** | **T** | **F** |
| **$I_0$** | s5 | | | s4 | | | 1 | 2 | 3 |
| **$I_1$** | | s6 | | | | ACC | | | |
| **$I_2$** | | r2 | s7 | | r2 | r2 | | | |
| **$I_3$** | | r4 | r4 | | r4 | r4 | | | |
| **$I_4$** | s5 | | | s4 | | | 8 | 2 | 3 |
| **$I_5$** | | r6 | r6 | | r6 | r6 | | | |
| **$I_6$** | s5 | | | s4 | | | | 9 | 3 |
| **$I_7$** | s5 | | | s4 | | | | | 10 |
| **$I_8$** | | s6 | | | s11 | | | | |
| **$I_9$** | | r1 | s7 | | r1 | r1 | | | |
| **$I_{10}$** | | r3 | r3 | | r3 | r3 | | | |
| **$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<br>GOTO ( $I_3$ , + ) = r4 ; **reduce** by T → F |
| 0 T 2 | + id * id $ | GOTO ( $I_0$ , T ) = 2<br>GOTO ( $I_2$ , + ) = r2 ; **reduce** by E → T |
| 0 E 1 | + id * id $ | GOTO ( $I_0$ , E ) = 1<br>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<br>GOTO ( $I_3$ , * ) = r4 ; **reduce** by T → F |
| 0 E 1 + 6 T 9 | * id $ | GOTO ( $I_6$ , T ) = 9<br>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<br>GOTO ( $I_{10}$ , $ ) = r3 ; **reduce** by T → T * F |
| 0 E 1 + 6 T 9 | $ | GOTO ( $I_6$ , T ) = 9<br>GOTO ( $I_9$ , $ ) = r1 ; **reduce** by E → E + T |
| 0 E 1 | $ | GOTO ( $I_0$ , E ) = 1<br>GOTO ( $I_1$ , $ ) = **accept** |

## 2.8 TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language.

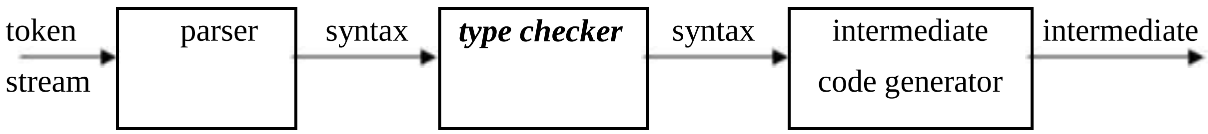
This checking, called *static checking*, detects and reports programming errors.

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 error occurs when an enclosing statement, such as break, does not exist in switch statement.

***Position of type checker***




- 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.

**2.9 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 $T_1$ and $T_2$ are type expressions, then their Cartesian product $T_1$ X $T_2$ is a type expression.

***Records*** : The difference between a record and a product is that the fields of a record have 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;***

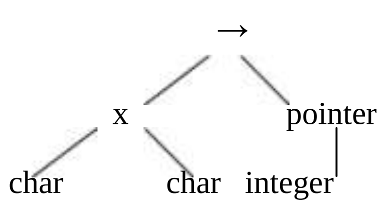
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

***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**

- Checkingdone 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 for type errors because it 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.

## 2.10 SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. 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.

<u>**A Simple Language**</u>

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($T_1$.*type*) }
T → array [ num ] of T1 { T.*type* : = array ( 1… num.val , $T_1$.*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* forE 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 → $E_1$ **mod** $E_2$ { *E.type* : = **if** *$E_1$. type* = *integer* **and**
   *$E_2$. 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 → $E_1$ [ $E_2$ ] { *E.type* : = if *$E_2$.type* = *integer* **and**
   *$E_1$.type* = *array(s,t)* **then** *t*
   **else** *type_error* }

   In an array reference $E_1$ [ $E_2$ ] , the index expression $E_2$ must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of $E_1$.

5. E → $E_1$ ↑ { *E.type* : = **if** *$E_1$.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 { S.*type* : = **if** **id**.*type* = E.*type* **then** *void* **else**
*type_error* }

**2. Conditional statement:**

**S** → **if** E **then** $S_1$ { S.*type* : = **if** E.*type* = *boolean* **then** $S_1$.*type*
**else** *type_error* }

**3. While statement:**

**S** → while E do $S_1$ { S.*type* : = **if** E.*type* = *boolean* **then** $S_1$.*type*
**else** *type_error* }

**4. Sequence of statements:**

**S** → $S_1$ ; $S_2$ { S.*type* : = **if** $S_1$.*type* = *void* and $S_1$.*type* = *void*
**then** *void*
**else** *type_error* }

**Type checking of functions**

The rule for checking the type of a function application is :

E → $E_1$ ( $E_2$ ) { E.*type* : = **if** $E_2$.*type* **=** *s* **and**
$E_1$.*type* **=** *s* → *t* **then**
*t* **else** *type_error* }

## 2.11 SOURCE LANGUAGE ISSUES

**Procedures:**

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.
For example, the following is the 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 within an executable statement, the procedure is said to be *called* at that point.

**Activation trees:**

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

**Control stack:**

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

- 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 control stack, the stack contains the nodes along the path from *n* to the root.

**The Scope of a Declaration:**
A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

var i : integer ;

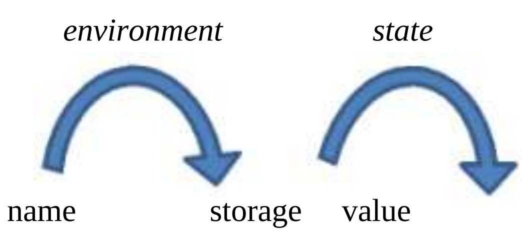
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the 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 objects at run time. "Data object" corresponds to a storage location that holds 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.

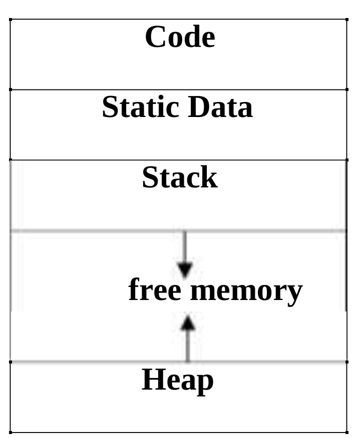



When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

**2.12 STORAGE ORGANISATION**

- 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 complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

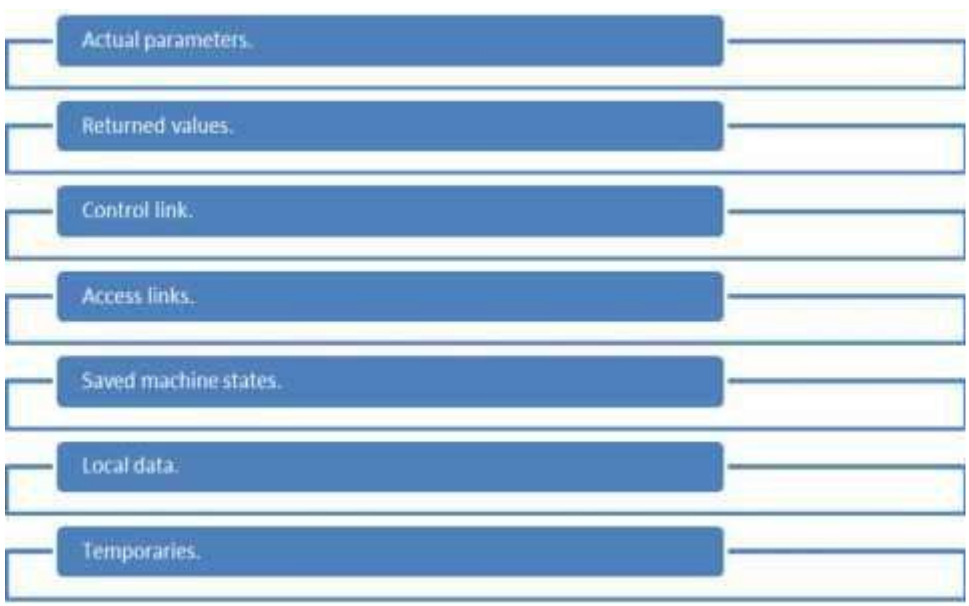
**Typical subdivision of run-time memory:**



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

**Activation records:**

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the 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 activation record.




- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

## 2.13 STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

### 2.13.1 STATIC ALLOCATION

- In static allocation, names are 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 run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

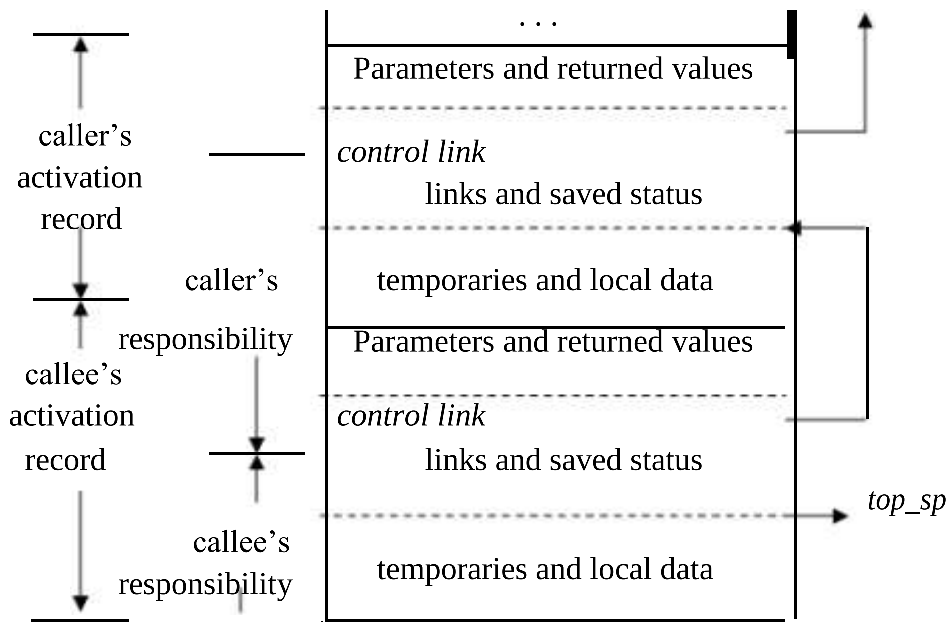
### 2.13.2 STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

**Calling sequences:**

- Procedures called are 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 machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.

- ➢ Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- ➢ Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- ➢ We must locate the top-of-stack pointer judiciously. A common approach is to have it point to 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-stack pointer.




**Division of tasks between caller and callee**

- The calling sequence and its division between caller and callee are as follows.

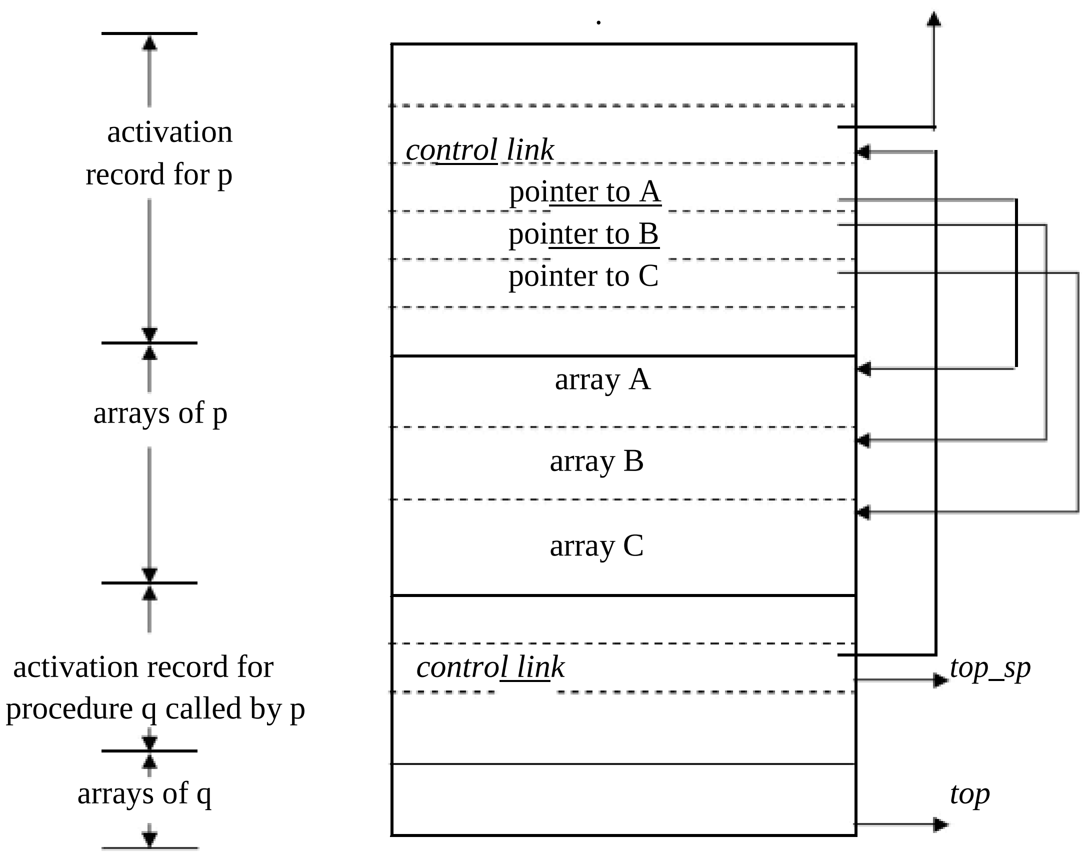
  - ➢ The caller evaluates the actual parameters.
  - ➢ The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - ➢ The callee saves the register values and other status information.
  - ➢ The callee initializes its local data and begins execution.

- A suitable, corresponding return sequence is:

  - ➢ The callee places the return value next to the parameters.
  - ➢ 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.
  - ➢ 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 stack:**

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.




**Access to dynamically allocated arrays**

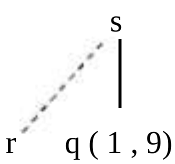
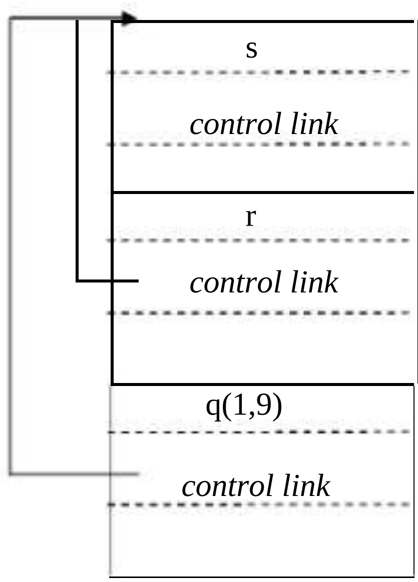
- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

### 2.13.3 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.

| Position in the activation tree | Activation records in the heap | Remarks |
|---|---|---|
|   |   | Retained activation record for r |

- 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.

## 3.1 INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

**Benefits of using a machine-independent intermediate form are:**

5. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
6. A machine-independent code optimizer can be applied to the intermediate representation.

***Position of intermediate code generator***

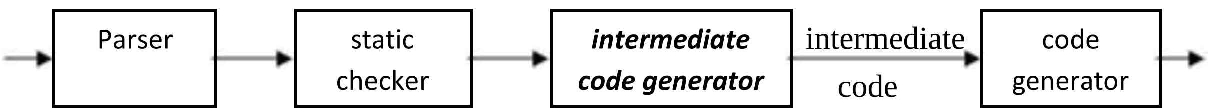



Figure 3.1:Intermediate code generator

## 3.2 INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

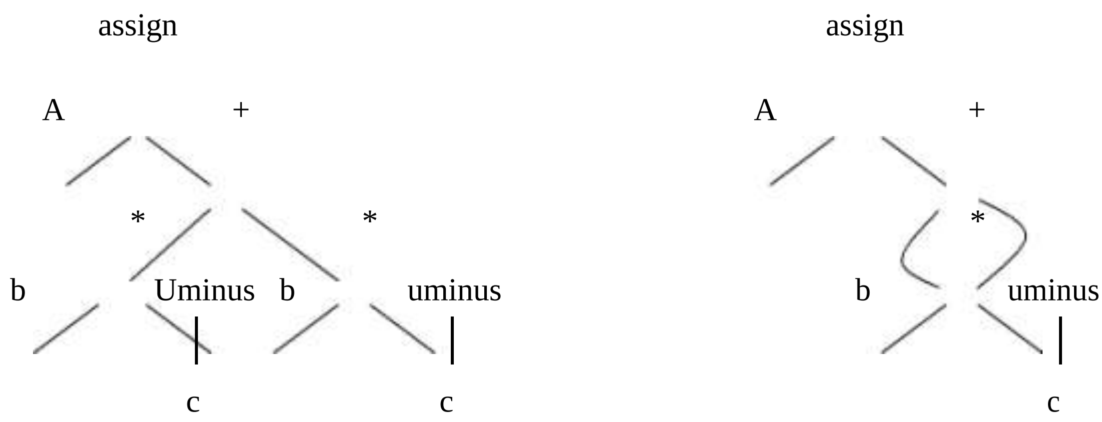
1. Syntax tree
2. Postfix notation
3. Three address code

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

### 3.2.1 Graphical Representations:

**Syntax tree:**

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:




**(a) Syntax tree** **(b) Dag**

Figure 3.2:Syntax Tree and DAG

**Postfix notation:**

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

a b c uminus * b c uminus * + assign

**Syntax-directed definition:**

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

Table 3.1:Syntax directed definition

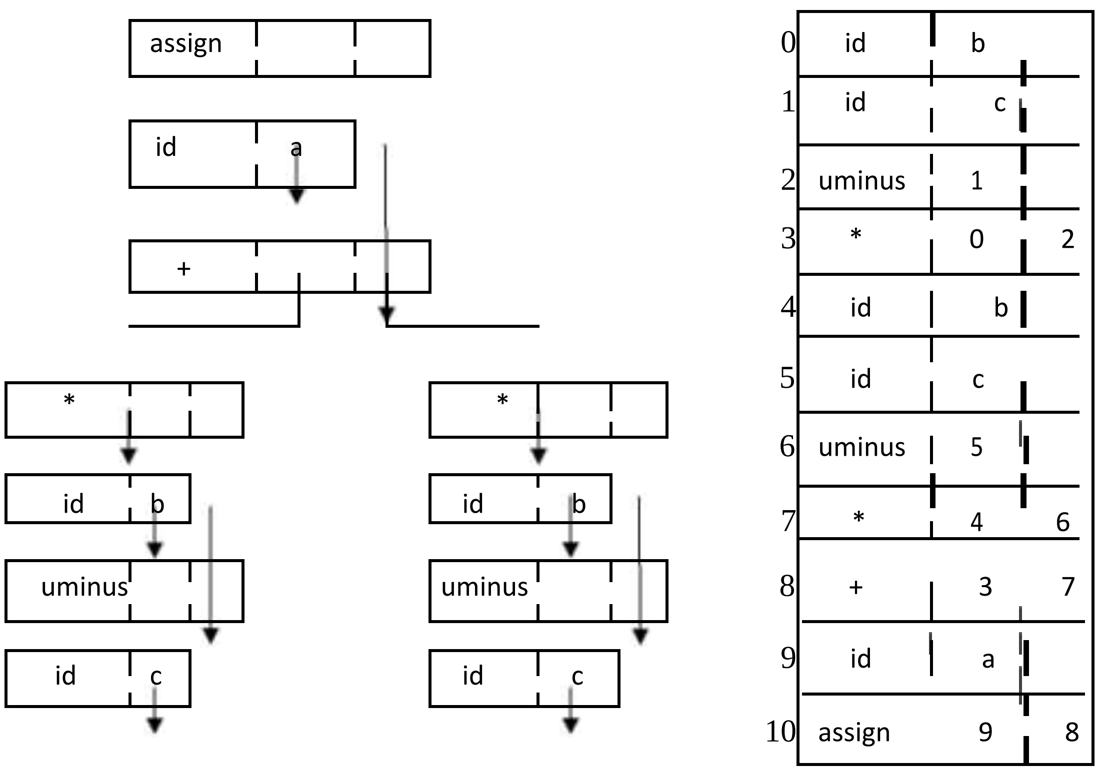
| PRODUCTION | SEMANTIC RULE |
|---|---|
| S → id : = E | S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr) |
| E → $E_1$ +$E_2$ | E.nptr : = mknode('+', $E_1$.nptr, $E_2$.nptr ) |
| E → $E_1$ * $E_2$ | E.nptr : = mknode('*', $E_1$.nptr, $E_2$.nptr ) |
| E → - $E_1$ | E.nptr : = mknode('uminus',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

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

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), 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 pointers, starting from the root at position 10.

**Two representations of the syntax tree**




(a) (b)

Figure 3.3:Two representations of syntax tree

### 3.2.2 Three-Address Code:

Three-address code is a sequence of statements of the general form

x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into asequence

$t_1$ : = y * z $t_2$ :
= x + $t_1$

where$t_1$ and $t_2$ 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. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

**Three-address code corresponding to the syntax tree and dag given above**

| | |
|---|---|
| $t_1$ : = - c | $t_1$ : = -c |
| $t_2$ : = b * $t_1$ | $t_2$ : = b * $t_1$ |
| $t_3$ : = - c | $t_5$ : = $t_2$ + $t_2$ |
| $t_4$ : = b * $t_3$ | a : = $t_5$ |
| $t_5$ : = $t_2$ + $t_4$ | |
| a : = $t_5$ | |
| **(a) Code for the syntax tree** | **(b) Code for the dag** |

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

**Types of Three-Address Statements:**

The common three-address statements are:

5. Assignment statements of the form **x : = y** ***op*** **z**, where ***op*** is a binary arithmetic or logical operation.

6. Assignment instructions of the form **x : =** ***op*** **y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

7. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.

8. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

9. Conditional jumps such as **if** ***x relop y*** **goto L**. This instruction applies a relational operator ( <, =, >=, etc. ) to *x* and *y*, and executes the statement with label L next if *x* stands in relation *relop to y*. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

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

   param $x_1$
   param $x_2$
   . . .
   param $x_n$
   call p,n

   generated as part of a call of the procedure p($x_1$, $x_2$, …. ,$x_n$ ).

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

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

**Syntax-Directed Translation into Three-Address Code:**

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

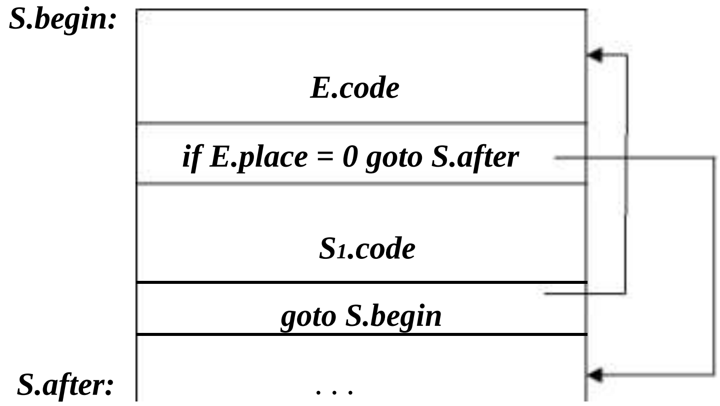
Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute *S.code* represents the three-address code for the assignment *S.* The nonterminal *E* has two attributes :

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

Table 3.2: Syntax-directed definition to produce three-address code for assignments

| PRODUCTION | SEMANTIC RULES |
|---|---|
| ***S* → *id : = E*** | ***S.code : = E.code \|\| gen(id.place ':=' E.place)*** |
| ***E* → $E_1$ + $E_2$** | ***E.place := newtemp;*** <br> ***E.code := $E_1$.code \|\| $E_2$.code \|\| gen(E.place ':=' $E_1$.place '+' $E_2$.place)*** |
| ***E* → $E_1$ * $E_2$** | ***E.place := newtemp;*** <br> ***E.code := $E_1$.code \|\| $E_2$.code \|\| gen(E.place ':=' $E_1$.place '*' $E_2$.place)*** |
| ***E* → - $E_1$** | ***E.place := newtemp;*** <br> ***E.code := $E_1$.code \|\| gen(E.place ':=' 'uminus' $E_1$.place)*** |
| ***E* → ($E_1$ )** | ***E.place : = $E_1$.place;*** <br> ***E.code : = $E_1$.code*** |
| ***E* → id** | ***E.place : = id.place;*** <br> ***E.code : = ' '*** |

**Semantic rules generating code for a while statement**




| **PRODUCTION** | **SEMANTIC RULES** |
|---|---|
| S → **while** *E* **do** $S_1$ | ***S.begin := newlabel;***<br>***S.after := newlabel;***<br>***S.code := gen(S.begin ':') \|\|***<br>***E.code \|\|***<br>***gen ( 'if' E.place '=' '0' 'goto' S.after)\|\|***<br>***$S_1$.code \|\|***<br>***gen ( 'goto' S.begin) \|\|***<br>***gen (S.after ':')*** |

Figure 3.4:semantic rule generating code for while statement

- The function *newtemp* returns a sequence of distinct names $t_1$,$t_2$,….. in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x*, *y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S*** → **while** ***E*** **do** ***$S_1$*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

**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, arg1, 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 fields 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.

***Triples:***

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2.*
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure ( for temporary values ).
- Since three fields are used, this intermediate code format is known as *triples*.

| | *op* | *arg1* | *arg2* | *result* |
|---|---|---|---|---|
| (0) | uminus | c | | $t_1$ |
| (1) | * | b | $t_1$ | $t_2$ |
| (2) | uminus | c | | $t_3$ |
| (3) | * | b | $t_3$ | $t_4$ |
| (4) | + | $t_2$ | $t_4$ | $t_5$ |
| (5) | : = | $t_3$ | | a |

**(a) Quadruples**

| | *op* | *arg1* | |
|---|---|---|---|
| (0) | uminus | c | |
| (1) | * | b | (0) |
| (2) | uminus | c | |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | assign | a | (4) |

**(b) Triples**

**Figure 3.4: Quadruple and triple representation of three-address statements given above**

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

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

**(a) x[i] : = y**

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

**(b) x : = y[i]**

Figure 3.5:triple structure

***Indirect Triples:***

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

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

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (14) | uminus | c | |
| (15) | * | b | (14) |
| (16) | uminus | c | |
| (17) | * | b | (16) |
| (18) | + | (15) | (17) |
| (19) | assign | a | (18) |

Figure 3.6:Indirect triples representation of three-address statements

## 3.3 DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

### 3.3.1 Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id : *T.***
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter( name, type, offset* ) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors *pointer* and *array*. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

**Computing the types and relative addresses of declared names**

| | |
|---|---|
| ***P → D*** | ***{ offset : = 0 }*** |
| ***D → D ; D*** | |
| ***D → id : T*** | ***{ enter(id.name, T.type, offset); offset : = offset + T.width }*** |
| ***T → integer*** | ***{ T.type : = integer; T.width : = 4 }*** |
| ***T → real*** | ***{ T.type : = real; T.width : = 8 }*** |
| ***T → array [ num ] of*** $T_1$ | ***{ T.type : = array(num.val,*** $T_1$***.type); T.width : = num.val X*** $T_1$***.width }*** |
| ***T → ↑*** $T_1$ | ***{ T.type : = pointer (*** $T_1$***.type); T.width : = 4 }*** |

### 3.3.2 Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

$D \rightarrow D\ ;\ D$ | **id** : $T$ | **proc id** ; $D$ ; $S$

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow$ ***proc id*** $D_1;S$ is seen, and entries for the declarations in $D_1$ are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray*, *exchange*, and *quicksort* pointing back to that for the containing procedure *sort*, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

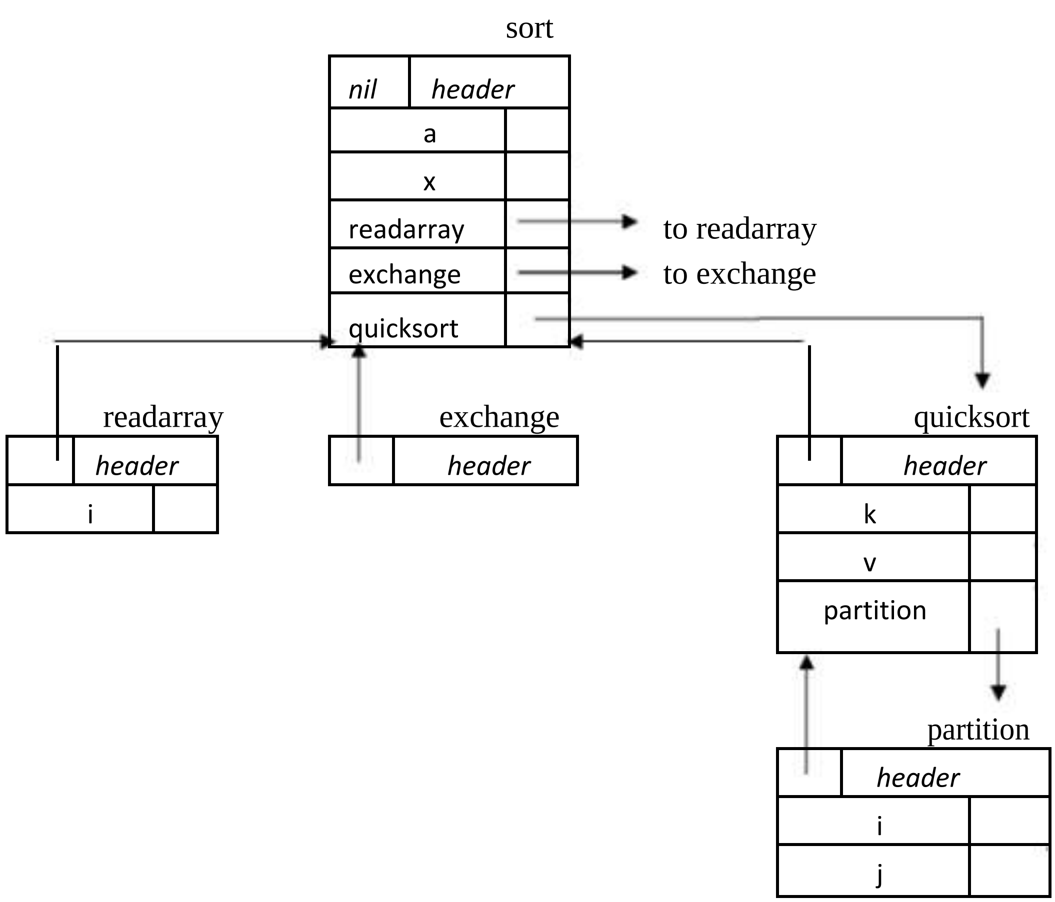



Figure 3.7: Symbol tables for nested procedures

The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

**Syntax directed translation scheme for nested procedures**

***P → M D*** ***{ addwidth ( top( tblptr) , top (offset)); pop (tblptr); pop (offset) }***

***M → ε*** ***{ t : = mktable (nil); push (t,tblptr); push (0,offset) }***

***D → $D_1$ ; $D_2$***

***D → proc id ; N $D_1$ ; S*** ***{ t : = top (tblptr); addwidth ( t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) }***

***D → id : T*** ***{ enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width }***

***N → ε*** ***{ t := mktable (top (tblptr)); push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.
- The top element of stack *offset* is the next available relative address for a local of the current procedure.
- All semantic actions in the subtrees for B and C in

  A → BC {$action_A$}

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil).* The action also pushes relative address 0 onto stack offset.
- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.
- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.
- When the action on the right side of *D → proc id; ND*$_1$*; S* occurs, the width of all declarations generated by $D_1$ is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

### 3.4 ASSIGNMENT STATEMENTS

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

P → M D

M → ε

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

N → ε

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

**Translation scheme to produce three-address code for assignments**

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

E → E1 + E2    { E.place : = newtemp;
                 emit( E.place ': =' E1.place ' + ' E2.place ) }

E → E1 * E2    { E.place : = newtemp;
                 emit( E.place ': =' E1.place ' * ' E2.place ) }

E → - 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 }

**Reusing Temporary Names**

- The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
- Temporaries can be reused by changing *newtemp*. The code generated by the rules for E → $E_1$ + $E_2$ has the general form:

  evaluate $E_1$ into $t_1$
  evaluate $E_2$ into $t_2$
  t : = $t_1$ + $t_2$

- The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1.
- For example, consider the assignment x := a * b + c * d – e * f

**Table 3.3:Three-address code with stack temporaries**

| *statement* | *value of c* |
|---|---|
| | 0 |
| $0 := a * b | 1 |
| $1 := c * d | 2 |
| $0 := $0 + $1 | 1 |
| $1 := e * f | 2 |
| $0 := $0 - $1 | 1 |
| x := $0 | 0 |

**Addressing Array Elements:**

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

***base* + ( *i* – *low* ) x *w***

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

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

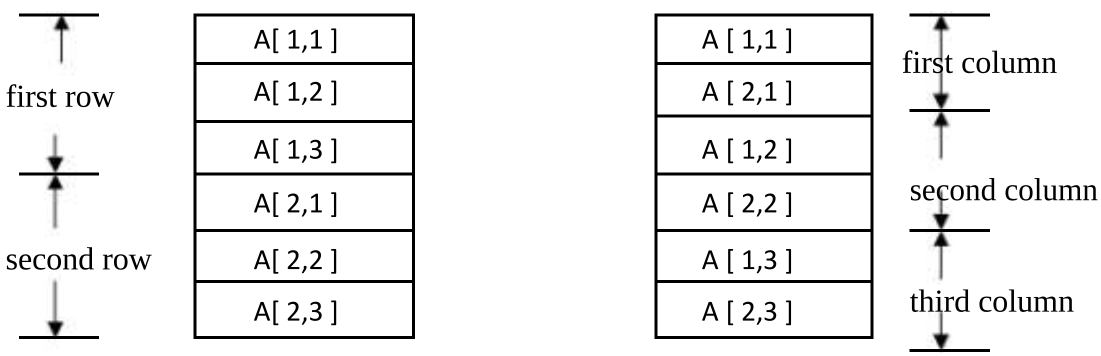
$$i \times w + (base - low \times w)$$

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

**Address calculation of multi-dimensional arrays:**
A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

**Layouts for a 2 x 3 array**




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

Figure 3.8:Address calculation of multi-dimensional arrays

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

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

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

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

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

**Generalized formula:**

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

$$((\ldots((i_1 n_2 + i_2)\, n_3 + i_3) \ldots)\, n_k + i_k) \times w + base - ((\ldots((low_1 n_2 + low_2) n_3 + low_3) \ldots)\, n_k + low_k) \times w$$

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

**The Translation Scheme for Addressing Array Elements :**
Semantic actions will be added to the grammar :

(1) *S* → *L* : = *E*
(2) *E* → *E* + *E*
(3) *E* → ( *E* )
(4) *E* → *L*
(5) *L* → *Elist* ]
(6) *L* → **id**
(7) *Elist* → *Elist* , *E*
(8) *Elist* → **id** [ *E*

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

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

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

(3) E → ( E1 )         { E.place : = E1.place }
```

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

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

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

(6) $L \rightarrow$ **id** { *L.place* := **id**.*place;*
*L.offset* := **null** }

(7) *Elist* $\rightarrow$ $Elist_1$ , *E* { *t := newtemp;*
*m : =* $Elist_1$*.ndim* + 1*;*
*emit* ( *t '': ='' Elist* $_1$*.place* '*' *limit*
($Elist_1$*.array,m*)); *emit* ( *t '': ='' t* '+' *E.place*);
*Elist.array : =* $Elist_1$*.array;*
*Elist.place : = t;*
*Elist.ndim : = m* }

(8) *Elist* $\rightarrow$ **id** [ *E* { *Elist.array : =* **id**.*place;*
*Elist.place : = E.place;*
*Elist.ndim : =* 1 }

**Type conversion within Assignments :**

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

$E \rightarrow E + E$ { *E.type* : =
**if** $E_1$*.type = integer* **and**
$E_2$*.type = integer* **then** *integer*
**else** *real* }

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

**Semantic action for** $E \rightarrow E_1 + E_2$

*E.place := newtemp;*
**if** $E_1$*.type = integer* **and** $E_2$*.type = integer* **then**
**begin** *emit( E.place ': =' $E_1$.place* 'int +'
$E_2$*.place); E.type : = integer*
**end**
**else if** $E_1$*.type = real* **and** $E_2$*.type = real* **then begin**
*emit( E.place ': =' $E_1$.place* 'real +'
$E_2$*.place); E.type : = real*
**end**

**else if** *$E_1$ .type = integer* **and** *$E_2$.type = real* **then**

**begin** *u : = newtemp;*

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

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

*E.type : = real*

**end**

**else if** *$E_1$.type = real* **and** *$E_2$.type =integer* **then**

**begin** *u : = newtemp;*

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

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

*E.type : = real*

**end**

**else**

*E.type : = type_error;*

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

$t_1$ : = i int* j $t_3$ :
= inttoreal $t_1$ $t_2$ :
= y real+ $t_3$ x : =
$t_2$

## 3.5 BOOLEAN EXPRESSIONS

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

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

Here we consider boolean expressions generated by the following grammar :

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

**Methods of Translating Boolean Expressions:**

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

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

### 3.5 .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.
For example :

- The translation for
  a **or** b **and not** c is
  the three-address sequence
  $t_1$ : = **not** c $t_2$
  : = b **and** $t_1$ $t_3$
  : = a **or** $t_2$
- A relational expression such as a < b is equivalent to the conditional
  statement if a < b then 1 else 0
  which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

  100 : if a < b goto 103
  101 : t : = 0
  102 : goto 104
  103 : t : = 1
  104 :

**Translation scheme using a numerical representation for booleans**

| | |
|---|---|
| *E* → *E*$_1$ **or** *E*$_2$ | *{ E.place : = newtemp;*<br>*emit( E.place ': =' E*$_1$*.place '***or***'E*$_2$*.place )}* |
| *E* → *E*$_1$ **and** *E*$_2$ | *{ E.place : = newtemp;*<br>*emit( E.place ': =' E*$_1$*.place '***and***'E*$_2$*.place )}* |
| *E* → **not** *E*$_1$ | *{ E.place : = newtemp;*<br>*emit( E.place ': =' '***not***' E*$_1$*.place )}* |
| *E* → *( E*$_1$ *)* | *{ E.place : = E*$_1$*.place }* |
| *E* → **id$_1$ relop id$_2$** | *{ E.place : = newtemp;*<br>*emit( 'if'* **id$_1$***.place* **relop***.op* **id$_2$***.place '***goto***' nextstat +* **3***);*<br>*emit( E.place ': =' '***0***' );*<br>*emit('***goto***' nextstat +***2***);*<br>*emit( E.place ': =' '***1***') }* |
| *E* → **true** | *{ E.place : = newtemp;*<br>*emit( E.place ': =' '***1***') }* |
| *E* → **false** | *{ E.place : = newtemp;*<br>*emit( E.place ': =' '***0***') }* |

## Short-Circuit Code:

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

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

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

**Flows-of-Control Statements**

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

S → **if** E **then** $S_1$

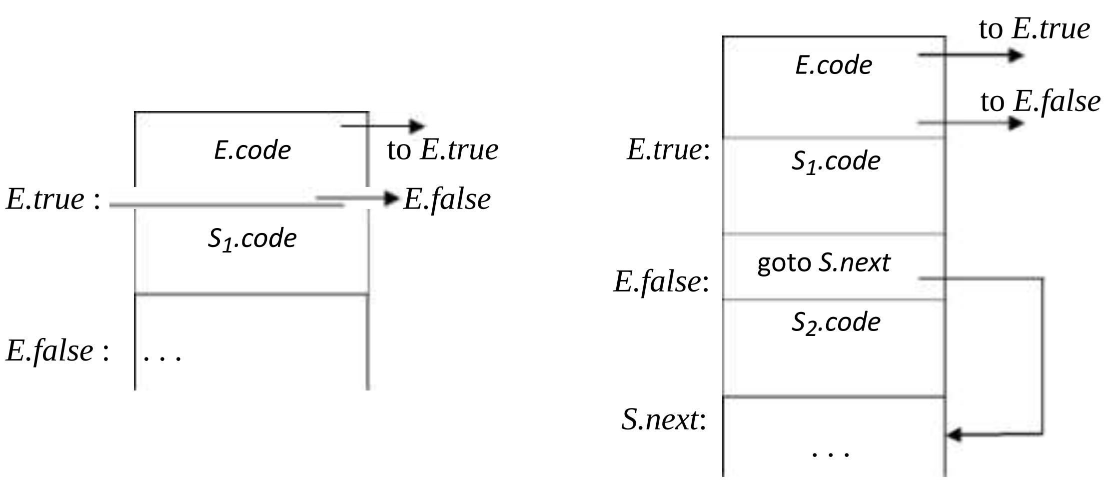
| **if** E **then** $S_1$ **else** $S_2$

| **while** E **do** $S_1$

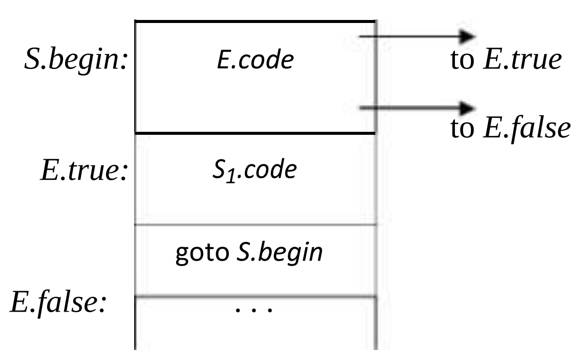
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

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




**(a) if-then** **(b) if-then-else**




**(c) while-do**

Figure3.9: code for if-then,if-then-else,while-do statements

Table 3.4:Syntax-directed definition for flow-of-control statements

| PRODUCTION | SEMANTIC RULES |
| --- | --- |
| $S \rightarrow$ **if** $E$ **then** $S_1$ | *E.true : = newlabel;*<br>*E.false : = S.next;*<br>*$S_1$.next : = S.next;*<br>*S.code : = E.code \|\| gen(E.true ':') \|\| $S_1$.code* |
| $S \rightarrow$ **if** $E$ **then** $S_1$ **else** $S_2$ | *E.true : = newlabel;*<br>*E.false : = newlabel;*<br>*$S_1$.next : = S.next;*<br>*$S_2$.next : = S.next;*<br>*S.code : = E.code \|\| gen(E.true ':') \|\| $S_1$.code \|\|*<br>*gen('***goto***' S.next) \|\|*<br>*gen( E.false ':') \|\| $S_2$.code* |
| $S \rightarrow$ **while** $E$ **do** $S_1$ | *S.begin : = newlabel;*<br>*E.true : = newlabel;*<br>*E.false : = S.next;*<br>*$S_1$.next : = S.begin;*<br>*S.code : = gen(S.begin ':')\|\| E.code \|\|*<br>*gen(E.true ':') \|\| $S_1$.code \|\|*<br>*gen('***goto***' S.begin)* |

### 3.5.2 Control-Flow Translation of Boolean Expressions:

Table 3.5:Syntax-directed definition to produce three-address code for booleans

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $E \rightarrow E_1$ **or** $E_2$ | $E_1.true := E.true;$<br>$E_1.false := newlabel;$<br>$E_2.true := E.true;$<br>$E_2.false := E.false;$<br>$E.code := E_1.code \mid\mid gen(E_1.false\ ':') \mid\mid E_2.code$ |
| $E \rightarrow E_1$ **and** $E_2$ | $E.true := newlabel;$<br>$E_1.false := E.false;$<br>$E_2.true := E.true;$<br>$E_2.false := E.false;$<br>$E.code := E_1.code \mid\mid gen(E_1.true\ ':') \mid\mid E_2.code$ |
| $E \rightarrow$ **not** $E_1$ | $E_1.true := E.false;$<br>$E_1.false := E.true;$<br>$E.code := E_1.code$ |
| $E \rightarrow ( E1 )$ | $E_1.true := E.true;$<br><br>$E.code := E_1.code$ |
| $E \rightarrow$ **id**$_1$ **relop id**$_2$ | $E.code := gen($'**if**' **id**$_1.place$ **relop.**$op$ **id**$_2.place$ '**goto**' $E.true) \mid\mid gen($'**goto**' $E.false)$ |
| $E \rightarrow$ **true** | $E.code := gen($'**goto**' $E.true)$ |
| $E \rightarrow$ **false** | $E.code := gen($'**goto**' $E.false)$ |

### 3.6 CASE STATEMENTS

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

**Switch-statement syntax**

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

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

4. Evaluate the expression.
5. Find which value in the list of cases is the same as the value of the expression.
6. Execute the statement associated with the value found.

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

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say $i_{min}$ to $i_{max}$, and the number of different values is a reasonable fraction of $i_{max}$ - $i_{min}$, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - $i_{min}$ and the label for the default in entries not filled otherwise. To perform switch, evaluate the expression to obtain the value of *j* , check the value is within range and transfer to the table entry at offset j-$i_{min}$ .

**Syntax-Directed Translation of Case Statements:**

Consider the following switch statement:

**switch** *E* **begin**
    **case** $V_1$ : $S_1$ **case**
    $V_2$ : $S_2$
        . . .
    **case** $V_{n-1}$ : $S_{n-1}$
    **default :** $S_n$
**end**

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

**Translation of a case statement**

| | |
|---|---|
| | code to evaluate *E* into t |
| | goto test |
| $L_1$ : | code for $S_1$ |
| | goto next |
| $L_2$ : | code for $S_2$ |
| | goto next |
| | . . . |
| $L_{n-1}$ : | code for $S_{n-1}$ |

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

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression *E* is parsed, the code to evaluate *E* into **t** is generated. After processing *E* , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label $L_i$ is created and entered into the symbol table. A pointer to this symbol-table entry and the value $V_i$ of case constant are placed on a stack (used only to store cases).
- Each statement **case** $V_i$ : $S_i$ is processed by emitting the newly created label $L_i$, followed by the code for $S_i$ , followed by the jump **goto next**.
- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

```
case V1 L1 case
V2 L2
    . . .
case Vn-1 Ln-1
case t Ln label
next
```

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

## 3.7 BACKPATCHING

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

To manipulate lists of labels, we use three functions :

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

**Boolean Expressions:**

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

5. $E \rightarrow E_1$ **or** $M\ E_2$
6. | $E_1$ **and** $M\ E_2$
7. | **not** $E_1$
8. | ( $E_1$)
9. | **id**$_1$ **relop id**$_2$
10. | **true**
11. | **false**
12. $M \rightarrow \varepsilon$

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

Consider production $E \rightarrow E_1$ **and** $M\ E_2$. If $E_1$ is false, then *E* is also false, so the statements on $E_1$.*falselist* become part of *E.falselist*. If $E_1$ is true, then we must next test $E_2$, so the target for the statements $E_1$.*truelist* must be the beginning of the code generated for $E_2$. This target is obtained using marker nonterminal *M*.

Attribute *M.quad* records the number of the first statement of $E_2$.*code*. With the production M $\rightarrow \varepsilon$ we associate the semantic action

{ *M.quad : = nextquad* }

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the $E_1$.*truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** $ME_2$. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* ( $E_1$.*falselist*, *M.quad*);
*E.truelist* : = *merge*( $E_1$.*truelist*, $E_2$.*truelist*);
*E.falselist* : = $E_2$.*falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* ( $E_1$.*truelist*, *M.quad*);

*E.truelist* : = $E_2$.*truelist*;
*E.falselist* : = *merge*($E_1$.*falselist*, $E_2$.*falselist*)}

(3) *E* → **not** $E_1$ { *E.truelist* : = $E_1$.*falselist*;
*E.falselist* : = $E_1$.*truelist*; }

(4) *E* → ( $E_1$ ) { *E.truelist* : = $E_1$.*truelist*;
*E.falselist* : = $E_1$.*falselist*; }

(5) *E* → **id**$_1$ **relop id**$_2$ { *E.truelist* : = *makelist* (*nextquad*);
*E.falselist* : = *makelist*(*nextquad + 1*);
*emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.place '**goto_**')
*emit*('**goto_**') }

(6) *E* → **true** { *E.truelist* : = *makelist*(*nextquad*);
*emit*('**goto_**') }

(7) *E* → **false** { *E.falselist* : = *makelist*(*nextquad*);
*emit*('**goto_**') }

(8) *M* → ε { *M.quad* : = *nextquad* }

**Flow-of-Control Statements:**

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

5. *S* → **if** *E* **then** *S*
6. | **if** *E* **then** *S* **else** *S*
7. | **while** *E* **do** *S*
8. | **begin** *L* **end**
9. | *A*
10. *L* → *L ; S*
11. | *S*

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

**Scheme to implement the Translation:**

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

The semantic rules for the revised grammar are as follows:

F $S \rightarrow$ **if** $E$ **then** $M_1$ $S_1$ $N$ **else** $M_2$ $S_2$

{ *backpatch* ($E.truelist$, $M_1.quad$);
*backpatch* ($E.falselist$, $M_2.quad$);
$S.nextlist$ : = *merge* ($S_1.nextlist$, *merge* ($N.nextlist$, $S_2.nextlist$)) }

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

(2) $N \rightarrow \varepsilon$ { *N.nextlist* : = *makelist*( *nextquad* );
*emit*('**goto** _') }

(3) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

(4) $S \rightarrow$ **if** $E$ **then** $MS_1$ { *backpatch*( *E.truelist*, *M.quad*);
$S.nextlist$ : = *merge*( *E.falselist*, $S_1.nextlist$) }

(5) $S \rightarrow$ **while** $M_1$ $E$ **do** $M_2$ $S_1$ { *backpatch*( $S_1.nextlist$, $M_1.quad$);
*backpatch*( *E.truelist*, $M_2.quad$);
*S.nextlist* : = *E.falselist*
*emit*( '**goto**' $M_1.quad$ ) }

(6) $S \rightarrow$ **begin** $L$ **end** { *S.nextlist* : = *L.nextlist* }

(7) $S \rightarrow A$ { *S.nextlist* : = **nil** }

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

(8) $L \rightarrow L1$ ; $M$ $S$ { *backpatch*( $L_1.nextlist$, *M.quad*);
*L.nextlist* : = *S.nextlist* }

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

(9) $L \rightarrow S$ { *L.nextlist* : = *S.nextlist* }

## 3.8 PROCEDURE CALLS

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

Let us consider a grammar for a simple procedure call statement

- *S* → **call id** ( *Elist* )
- *Elist* → *Elist* , *E*
- *Elist* → *E*

**Calling Sequences:**

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

- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be

generated. For example, consider the following syntax-directed translation

*S* → **call id** ( *Elist* )
{ **for** each item *p* on *queue* **do**
*emit* (' **param** ' *p* );
*emit* ('call' **id.***place*) }

3. *Elist* → *Elist* , *E*

{ append *E.place* to the end of *queue* }

4. *Elist* → *E*

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

- Here, the code for S is the code for *Elist*, which evaluates the arguments, followed by a **param** *p* statement for each argument, followed by a **call** statement.
- *queue* is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

# CHAPTER IV - CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

**Position of code generator**

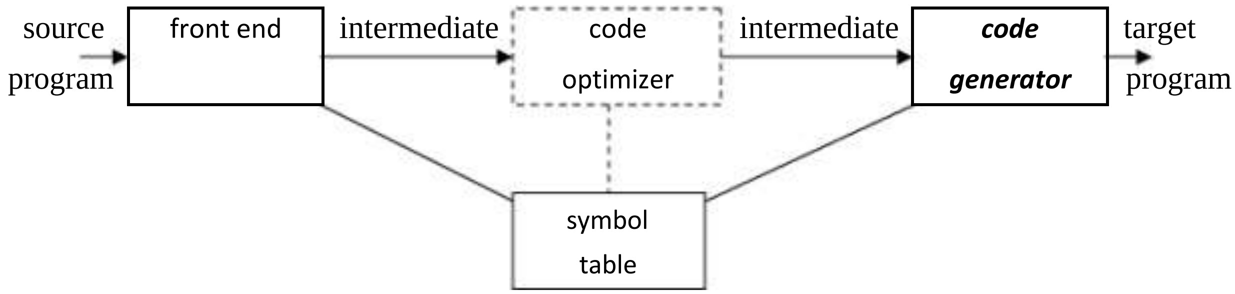



Figure 4.1:Code Generator

## 4.1 ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

7. Input to code generator
8. Target program
9. Memory management
10. Instruction selection
11. Register allocation
12. Evaluation order

**9) Input to code generator:**

The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation. Intermediate representation can be :

Linear representation such as postfix notation
Three address representation such as quadruples
Virtual machine representation such as stack machine code
Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

**10) Target program:**

The output of the code generator is the target program. The output may be :

Absolute machine language

It can be placed in a fixed memory location and can be executed immediately.

10. Relocatable machine language
    It allows subprograms to be compiled separately.

11. Assembly language
    - Code generation is made easier.

**3. Memory management:**

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name. Labels in three-address statements have to be converted to addresses of instructions. For example,

*j* : **goto** *i* generates jump instruction as follows :

- if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
- if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

**4. Instruction selection:**

The instructions of target machine should be complete and uniform. Instruction speeds and machine idioms are important factors when efficiency of target program is considered. The quality of the generated code is determined by its speed and size. The former statement can be translated into the latter statement as shown below:

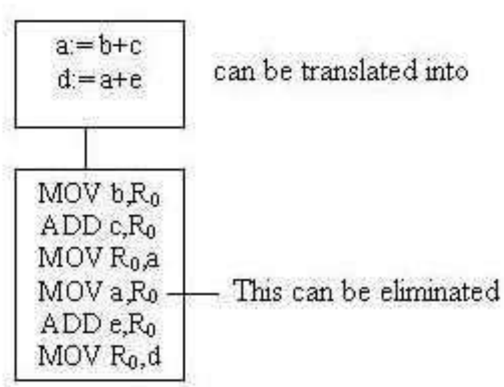



Figure 4.2:Instruction selection

**5. Register allocation**

Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two subproblems :

- ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected
- ***Register assignment*** – the specific register that a variable will reside in is picked.

Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair
y – divisor
even register holds the remainder
odd register holds the quotient

**6. Evaluation order**

The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

## 4.2 TARGET MACHINE

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. The target computer is a byte-addressable machine with 4 bytes to a word.It has *n* general-purpose registers, $R_0$, $R_1$, . . . , $R_{n-1}$.

It has two-address instructions of the form:

*op source, destination*

where, *op* is an op-code, and *source* and *destination* aredata fields.

It has the following op-codes :

MOV (move *source* to *destination*)
ADD (add *source* to *destination*)
SUB (subtract *source* from *destination*)

The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

**Table 4.1:Address modes with their assembly-language forms**

| MODE | FORM | *ADDRESS* | ADDED COST |
|---|---|---|---|
| *Absolute* | M | M | 1 |
| *Register* | R | R | 0 |
| *Indexed* | *c*(R) | *c*+*contents*(R) | 1 |
| *indirect register* | *R | *contents* (R) | 0 |
| *indirect indexed* | **c*(R) | *contents*(*c*+ *contents*(R)) | 1 |
| *Literal* | #*c* | *c* | 1 |

For example : MOV $R_0$, M stores contents of Register $R_0$ into memory location M ;
MOV 4($R_0$), M stores the value *contents*(4+*contents*($R_0$)) into M.

**Instruction costs :**

Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction. Address modes involving registers have cost zero. Address modes involving memory location or literal have cost one. Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.

The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

i) MOV b, $R_0$
ADD c, $R_0$ cost = 6
MOV $R_0$, a

ii) MOV b, a
ADD c, a cost = 6

iii) Assuming $R_0$, $R_1$ and $R_2$ contain the addresses of a, b, and c :
MOV *$R_1$, *$R_0$
ADD *$R_2$, *$R_0$ cost = 2

In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

## 4.3 RUN-TIME STORAGE MANAGEMENT

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

Static allocation
Stack allocation

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

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

Call,
Return,
Halt, and
Action, a placeholder for other statements.

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

Code
Static data
Stack

## 4.3.1 Static allocation

**Implementation of call statement:**

The codes needed to implement static allocation are as follows:

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

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

where,
*callee.static_area* – Address of the activation record *callee.code_area*
– Address of the first instruction for called procedure
*#here* +20 – Literal return address which is the address of the instruction following GOTO.

**Implementation of return statement:**

A return from procedure *callee* is implemented by :

**GOTO** **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

**Implementation of action statement:**

The instruction ACTION is used to implement action statement.

**Implementation of halt statement:**

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

## 4.3.2 Stack allocation

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

The codes needed to implement stack allocation are as follows:
**Initialization of stack:**

**MOV** *#stackstart* , SP /* initializes stack */

Code for the first procedure

**HALT** /* terminate execution */
**Implementation of Call statement:**

**ADD** *#caller.recordsize*, SP /* increment stack pointer */

**MOV** *#here* +16, *SP /*Save return address */

**GOTO** *callee.code_area*

where,
*caller.recordsize* – size of the activation record
*#here* +16 – address of the instruction following the **GOTO**

**Implementation of Return statement:**

**GOTO** *0 (SP ) /*return to the caller */

**SUB** *#caller.recordsize*, SP /* decrement SP and restore to previous value */

## 4.4 BASIC BLOCKS AND FLOW GRAPHS

### 4.4.1 Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block: $t_1$:=a*a

$t_2$ : = a * b
$t_3$ : = 2 * $t_2$
$t_4$ : = $t_1$ + $t_3$
$t_5$ : = b * b
$t_6$ : = $t_4$ + $t_5$

**Basic Block Construction:**

**Algorithm:** Partition into basic blocks

**Input:** A sequence of three-address statements

**Output:** A list of basic blocks with each three-address statement in exactly one block

**Method:**

(9) We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
The first statement is a leader.
Any statement that is the target of a conditional or unconditional goto is a leader.
Any statement that immediately follows a goto or conditional goto statement is a leader.
(10) For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Figure 4.3:Basic block construction

Consider the following source code for dot product of two vectors a and b of length 20

```
begin
        prod :=0;
        i:=1; do
        begin
                prod :=prod+ a[i]* b[i];
                i :=i+1;
        end
        while i <= 20
end
```

Figure 4.4:Source code for dot product

The three-address code for the above source program is given as :

| | | |
|---|---|---|
| (1) | prod := 0 | |
| (2) | i := 1 | |
| (3) | $t_1$ := 4* i | |
| (4) | $t_2$ := a[$t_1$] | /*compute a[i] */ |
| (5) | $t_3$ := 4*i | |
| (6) | $t_4$ := b[$t_3$] | /*compute b[i] */ |
| (7) | $t_5$ := $t_2$*$t_4$ | |
| (8) | $t_6$ := prod+$t_5$ | |
| (9) | prod := $t_6$ | |
| (10) | $t_7$ := i+1 | |
| (11) | i := $t_7$ | |
| (12) | if i<=20 goto (3) | |

Figure 4.5:Three address code for above fig4.4

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

### 4.4.2 Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

1. Structure-preserving transformations
2. Algebraic transformations

1. **Structure preserving transformations:**

**a. Common subexpression elimination:**

| | |
|---|---|
| a : = b + c | a : = b + c |
| b : = a – d | b : = a - d |
| c : = b + c | c : = b + c |
| d : = a – d | d : = b |

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

**b) Dead-code elimination:**

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

**c) Renaming temporary variables:**

A statement **t : = b + c** ( t is a temporary ) can be changed to **u : = b + c** (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block.
Such a block is called a *normal-form block.*

**d) Interchange of statements:**

Suppose a block has the following two adjacent statements:

t1 : = b + c
t2 : = x + y

We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is $\mathbf{t_1}$ and neither **b** nor **c** is $\mathbf{t_2}$.

**8. Algebraic transformations:**

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

## 4.4.2 Flow Graphs

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks. It has a distinguished initial node. E.g.: Flow graph for the vector dot product is given as follows:

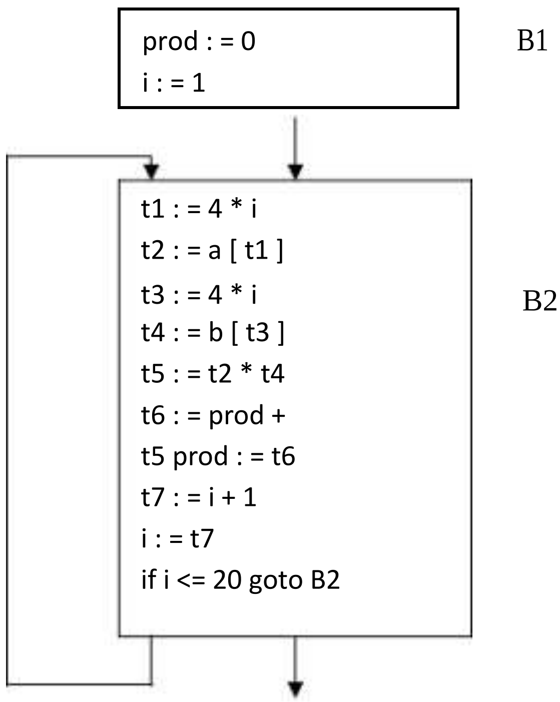



Figure 4.6:Flow graph for vector dot product

$B_1$ is the *initial* node. $B_2$ immediately follows $B_1$, so there is an edge from $B_1$ to $B_2$. The target of jump from last statement of $B_1$ is the first statement $B_2$, so there is an edge from $B_1$ (last statement) to $B_2$ (first statement). $B_1$ is the *predecessor* of $B_2$, and $B_2$ is a *successor* of $B_1$.

## 4.4.4 Loops

A loop is a collection of nodes in a flow graph such that

All nodes in the collection are *strongly connected*.

The collection of nodes has a unique *entry*.

A loop that contains no other loops is called an inner loop.

## 4.5 NEXT-USE INFORMATION

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

**Input:** Basic block B of three-address statements

**Output:** At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

**Method:** We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live",and next-uses of y and z to i.

Figure 4.7:Next-Use Information

**Table 4.2: Symbol Table:**

| **Names** | **Liveliness** | **Next-use** |
|---|---|---|
| x | not live | no next-use |
| y | Live | i |
| z | Live | i |

## 4.6 A SIMPLE CODE GENERATOR

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements. For example: consider the three-address statement **a := b+c** It can have the following sequence of codes:

ADD $R_j$, $R_i$ Cost = 1 // if $R_i$ contains b and $R_j$ contains c

(or)

ADD c, $R_i$ Cost = 2 // if c is in a memory location

(or)

MOV c, $R_j$ Cost = 3 // move c from memory to Rj and add

ADD $R_j$, $R_i$

**Register and Address Descriptors:**

A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty. An address descriptor stores the location where the current value of the name can be found at run time.

**A code-generation algorithm:**

The algorithm takes as input a sequence of three -address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

-  Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.
-  Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.
-  Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
-  If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

**Generating Code for Assignment Statements:**

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

t : = a – b
u : = a – c
v : = t + u
d : = v + u

with d live at the end.

Table 4.3: Code sequence for the example:

| **Statements** | **Code Generated** | **Register descriptor** | **Address descriptor** |
|---|---|---|---|
| | | Register empty | |
| t : = a - b | MOV a, $R_0$<br>SUB b, R0 | $R_0$ contains t | t in $R_0$ |
| u : = a - c | MOV a , R1<br>SUB c , R1 | $R_0$ contains t<br>R1 contains u | t in $R_0$<br>u in R1 |
| v : =t + u | ADD $R_1$, $R_0$ | $R_0$ contains v<br>$R_1$ contains u | u in $R_1$<br>v in $R_0$ |
| d : = v + u | ADD $R_1$, $R_0$<br>MOV $R_0$, d | $R_0$ contains d | d in $R_0$<br>d in $R_0$ and memory |

## Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements **a : = b [ i ]** and **a [ i ] : = b**

Table 4.4: Indexed assignment

| **Statements** | **Code Generated** | **Cost** |
|---|---|---|
| a : = b[i] | MOV b($R_i$), R | 2 |
| a[i] : = b | MOV b, a($R_i$) | 3 |

## Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments **a : = *p** and ***p : = a**

Table 4.5:Pointer assignment

| **Statements** | **Code Generated** | **Cost** |
|---|---|---|
| a : = *p | MOV *$R_p$, a | 2 |
| *p : = a | MOV a, *$R_p$ | 2 |

Table 4.6:Generating Code for Conditional Statements

| **Statement** | **Code** |
|---|---|
| if x < y goto z | CMP x, y<br>CJ< z /* jump to z if condition code is negative */ |
| x : = y +z if x <0 goto z | MOV y, $R_0$<br>ADD z, $R_0$<br>MOV $R_0$,x<br>CJ< z |

## 4.7 THE DAG REPRESENTATION FOR BASIC BLOCKS

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

Leaves are labeled by unique identifiers, either variable names or constants.
Interior nodes are labeled by an operator symbol.
Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks.

- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

### Algorithm for construction of DAG

**Input:** A basic block

**Output:** A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i)x := y OP z

Case (ii)x := OP y

Case (iii)x := y

**Method:**

**Step 1:** If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

**Step 2:** For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checkingfor common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

**Step 3:** Delete x from the list of identifiers for node(x). Append x to the list of attached

identifiers for the noden found in step 2 and set node(x) to n.

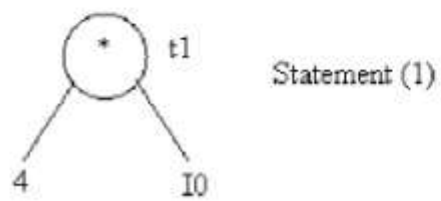
**Figure 4.8:Algorithm for construction of DAG**

**Example:** Consider the block of three- address statements:

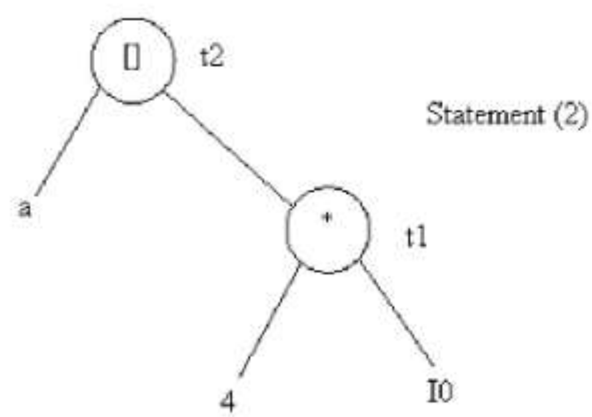
4. $t_1$ := 4* i
5. $t_2$ := a[$t_1$]
6. $t_3$ := 4* i
7. $t_4$ := b[$t_3$]
8. $t_5$ := $t_2$*$t_4$
9. $t_6$ := prod+$t_5$
10. prod := $t_6$
11. $t_7$ := i+1
12. i := $t_7$
13. if i<=20 goto (1)

**Stages in DAG Construction**

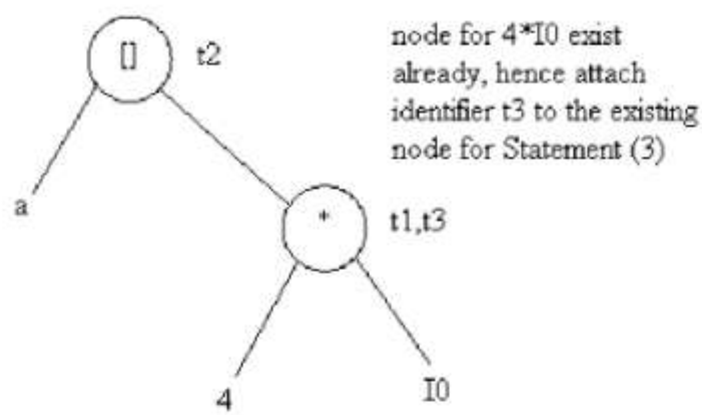
(a)



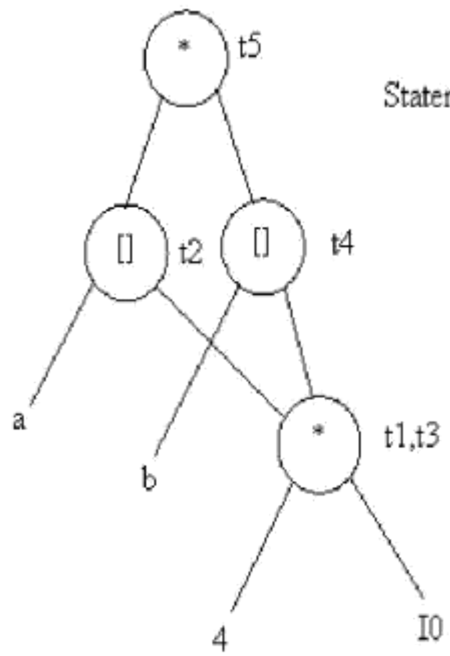

(b)




(c)

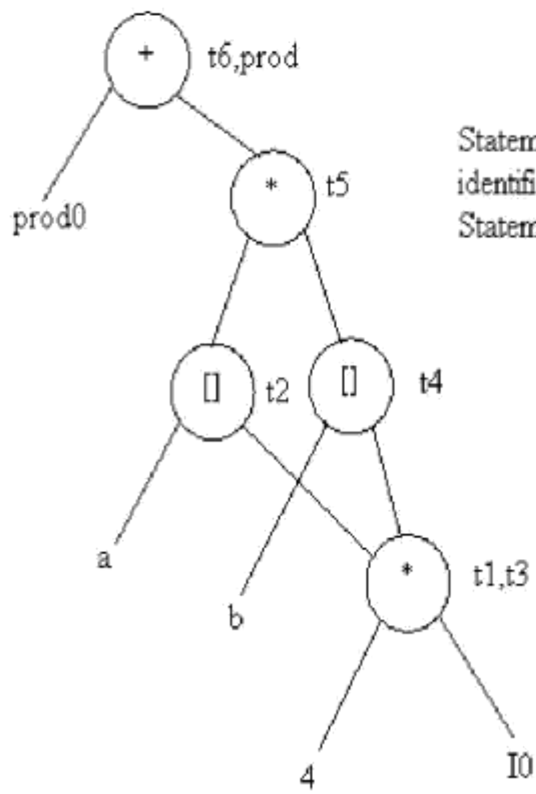


(e)



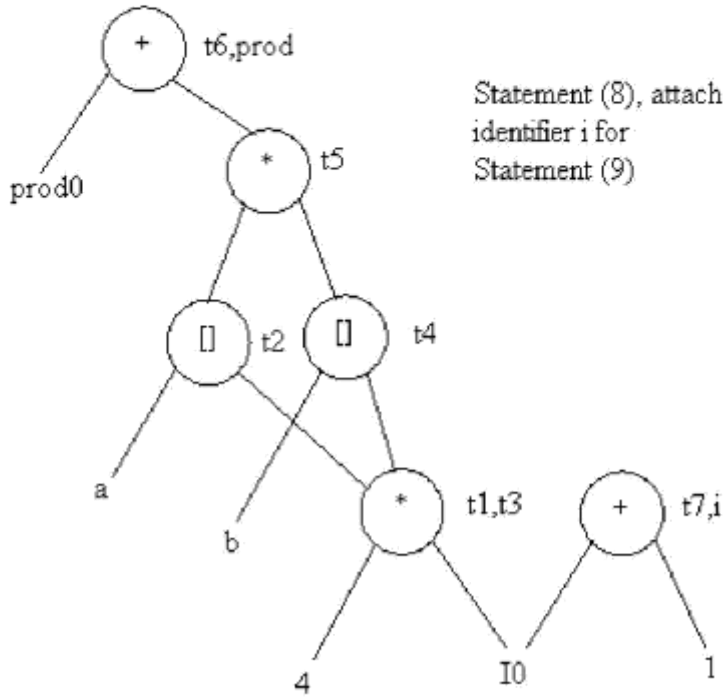

Statement (5)

(f)



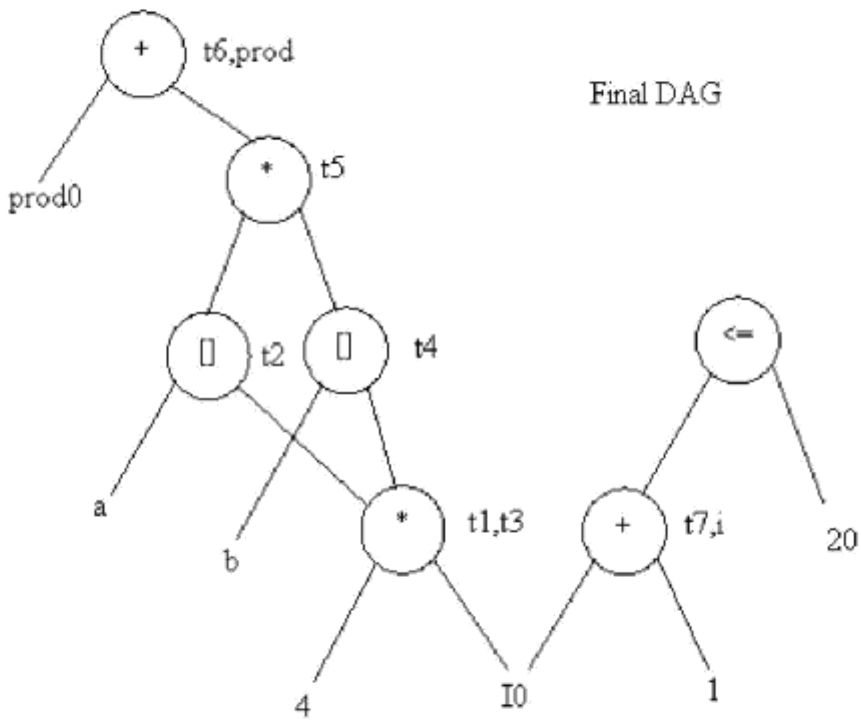

Statement (6), attach identifier prod for Statement (7)

(g)




Statement (8), attach identifier i for Statement (9)

(h)




Final DAG

**Application of DAGs:**

5. We can automatically detect common sub expressions.
6. We can determine which identifiers have their values used in the block.
7. We can determine which statements compute values that could be used outside the block.

## GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

**Rearranging the order**

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1$ : = a + b
$t_2$ : = c + d
$t_3$ : = e – $t_2$
$t_4$ : = $t_1$ – $t_3$

**Generated code sequence for basic block:**

MOV a , $R_0$
ADD b , $R_0$
MOV c , $R_1$
ADD d , $R_1$
MOV $R_0$ , $t_1$
MOV e , $R_0$
SUB $R_1$ , $R_0$
MOV $t_1$ , $R_1$
SUB $R_0$ , $R_1$
MOV $R_1$ , $t_4$

**Rearranged basic block:**

Now t1 occurs immediately before t4.

$t_2$ : = c + d
$t_3$ : = e – $t_2$
$t_1$ : = a + b
$t_4$ : = $t_1$ – $t_3$

**Revised code sequence:**

MOV c , $R_0$
ADD d , $R_0$
MOV a , $R_0$
SUB $R_0$ , $R_1$
MOV a , $R_0$
ADD b , $R_0$
SUB $R_1$ , $R_0$
MOV $R_0$ , $t_4$

In this order, two instructions **MOV $R_0$ , $t_1$** and **MOV $t_1$ , $R_1$** have been saved.

## A Heuristic ordering for Dags

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

**Algorithm:**

```
8.   while unlisted interior nodes remain do begin
9.        select an unlisted node n, all of whose parents have been listed;
10.       list n;
11.       while the leftmost child m of n has no unlisted parents and is not a leaf do
              begin
12.               list m;
13.               n : = m
              end
          end
```

**Example:** Consider the DAG shown below:

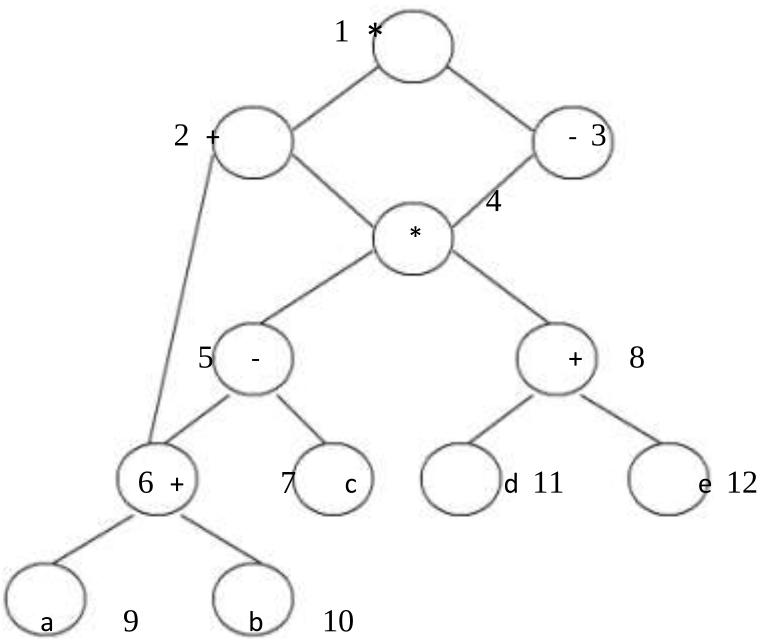



Figure 4.9:Example DAG

Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select anew n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

**Code sequence:**

$t_8$ : = d + e $t_6$ :
= a + b $t_5$ : = $t_6$
– c $t_4$ : = $t_5$ * $t_8$
$t_3$ : = t4 – e $t_2$ :
= $t_6$ + $t_4$ $t_1$ : =
$t_2$ * $t_3$

This will yield an optimal code for the DAG on machine whatever be the number of registers.

# CHAPTER V - CODE OPTIMIZATION

## 5.1 INTRODUCTION

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are
Machine independent optimizations:
Machine dependant optimizations:

**Machine independent optimizations:**

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

**Machine dependant optimizations:**

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

**The criteria for code improvement transformations:**

- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.

- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

**Organization for an Optimizing Compiler:**

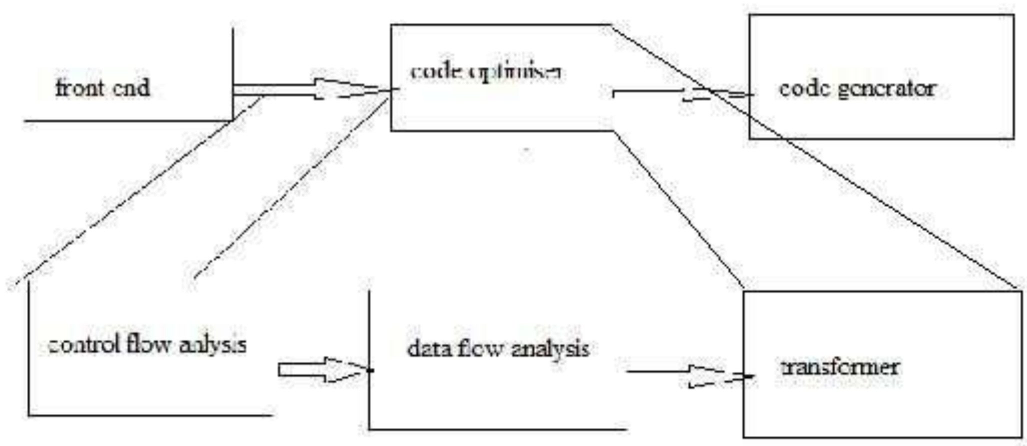



Figure 5.1:Organization for an Optimizing Compiler

Flow analysis is a fundamental prerequisite for many important types of code improvement. Generally control flow analysis precedes data flow analysis.Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as

1. control flow graph
2. Call graph

Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

## 5.2 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 perormed at both the local and global levels. Local transformations are usually performed first.

### 5.2.1 Function-Preserving Transformations

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

- ✓ Common sub expression elimination,
- ✓ Copy propagation,
- ✓ Dead-code elimination, and
- ✓ Constant folding

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

Frequently, a program will include several calculations of the same value, such as an 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.

**1.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

$t_1$: =4*i
$t_2$: =a [t1]
$t_3$: =4*j
$t_4$:=4*i
$t_5$: =n
$t_6$: =b [$t_4$] +$t_5$

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

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

The common sub expression t $_4$: =4*i is eliminated as its computation is already in $t_1$. And value of i is not been changed from definition to use.

**2.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

**3.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 the 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. An optimization can be done byeliminating dead code.

Example:

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

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

**4.Constant folding**:

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

### 5.2.2 Loop Optimizations:

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization:

code motion, which moves code outside a loop;

Induction-variable elimination, which we apply to replace variables from inner loop.

Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

**1.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*/
```

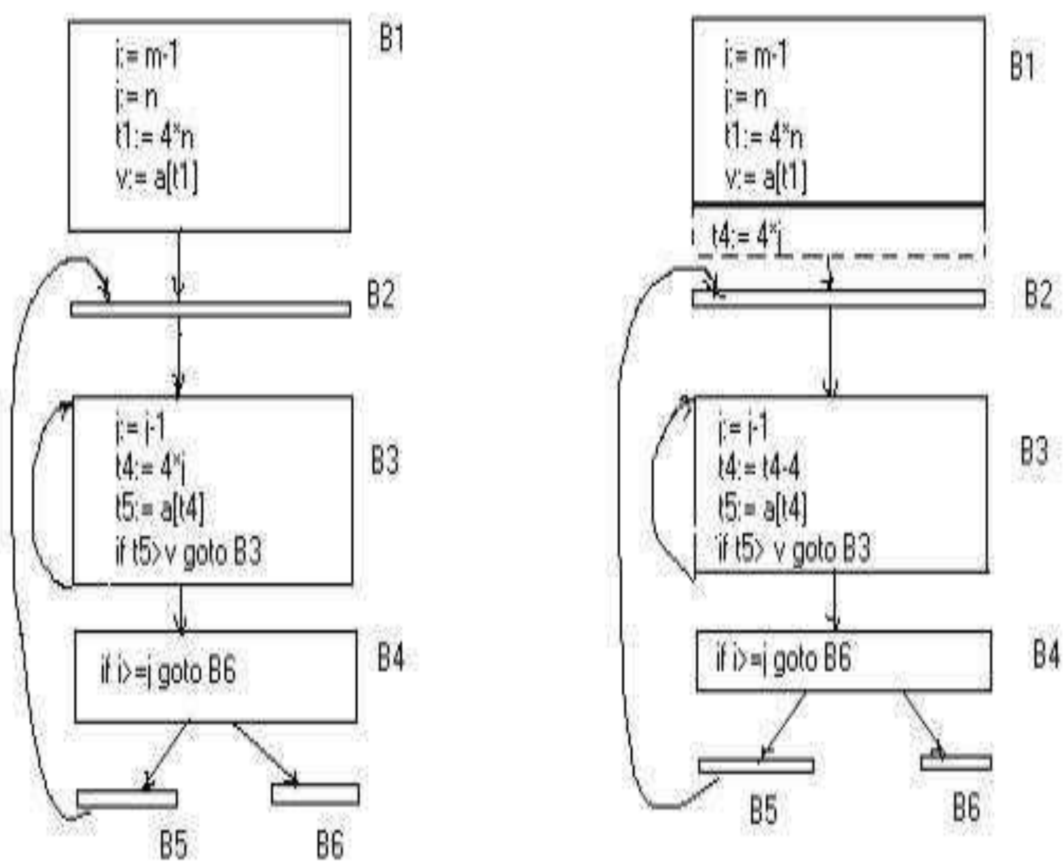
Code motion will result in the equivalent of

t= limit-2;
while (i<=t) /* statement does not change limit or t */

**2.Induction Variables :**

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and $t_4$ remain in lock-step; every time the value of j decreases by 1, that of $t_4$ decreases by 4 because 4*j is assigned to $t_4$. 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. we cannot get rid of either j or $t_4$ completely; $t_4$ 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 $t_4$:=4*j surely holds after such an assignment to $t_4$ in Fig. and $t_4$ is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j -1 the relationship $t_4$:= 4*j-4 must hold. We may therefore replace the assignment $t_4$:= 4*j by $t_4$:= $t_4$-4. The only problem is that $t_4$ does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t_4$=4*j on entry to the block B3, we place an initializations of $t_4$ at the end of the block where j itself is




**before** **after**

Figure 5.2:Induction variable example

initialized, shown by the dashed addition to block B1 in second Fig.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.

**3.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.

## 5.2.3 OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- Structure-Preserving Transformations
- Algebraic Transformations

**Structure-Preserving Transformations:**

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expressionelimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

**1.Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it’s referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

 =b+c
 =a-d
 =b+c
 =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to
 =b+c
 =a-d
 =a
 =b

2.**Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error -correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

**3.Renaming of temporary variables:**

A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u. In this we can transform a basic block to its equivalent block called normal-form block.

**4.Interchange of two independent adjacent statements:**

Two statements

$t_1$:=b+c

$t_2$:=x+y

can be interchanged or reordered in its computation in the basic block when value of $t_1$ does not affect the value of $t_2$.

**Algebraic Transformations:**

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c e

:=c+d+b

the following intermediate code may be generated:

a :=b+c
t :=c+d
:=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

## 5.3 LOOPS IN FLOW GRAPH

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

**Dominators:**

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

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

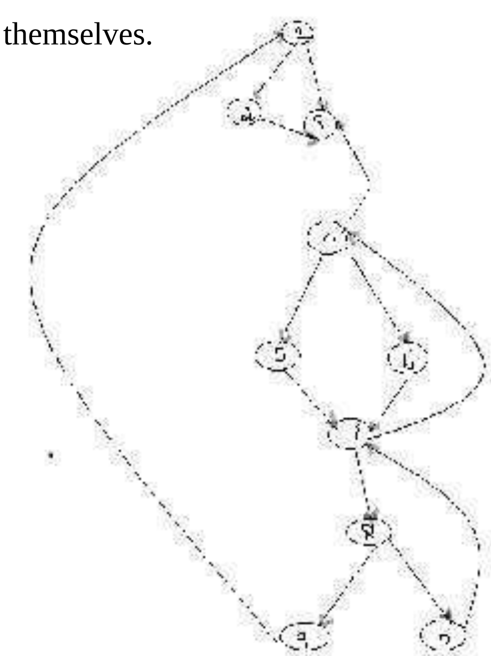


Figure 5.3:Flow graph

The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root. The parent of each other node is its immediate dominator. Each node d dominates only its descendents in the tree. The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.

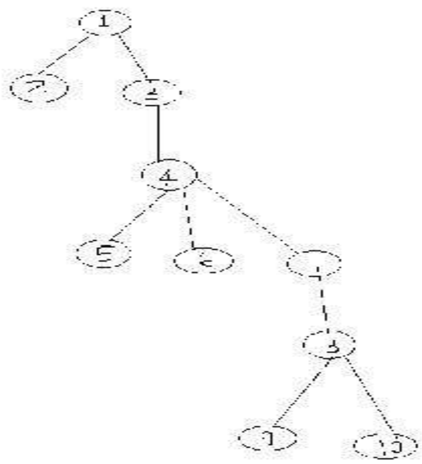


Figure 5.4:Dominator Tree

D(1)={1}

D(2)={1,2}

D(3)={1,3}

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

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

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

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

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

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

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

**Natural Loop:**

One application of dominator information is in determining the loops of a flow graph suitable for improvement. The properties of loops are

A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
There must be at least one wayto iterate the loop(i.e.)at least one path back to the header.

One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.
Example:

In the above graph,

7 → 4 4 DOM 7

10 →7 7 DOM 10

4 → 3

8 → 3

9 →1

The above edges will form loop in flow graph. Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

**Algorithm:** Constructing the natural loop of a back edge.

**Input:** A flow graph G and a back edge n→d.

**Output:** The set loop consisting of all nodes in the natural loop n→d.

**Method:** Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

**Procedure** insert(m);
**if** m is not in *loop* **then**
 **begin** *loop* := *loop* U
 {m}; push *m* onto *stack*
**end;**

*stack* : =empty;

*loop* :
={*d*};
*insert*(*n*);
**while** *stack* is not empty **do begin**
pop *m,* the first element of *stack*, off *stack*;
**for** each predecessor *p* of *m* **do** *insert(p)*
**end Inner**

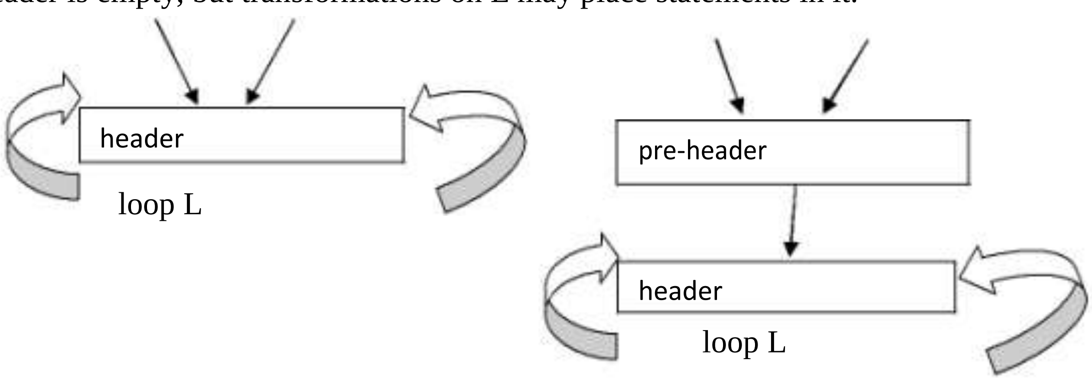
**loop:**

If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

**Pre-Headers:**

Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater. The pre-header has only the header as successor, and all edges which formerly entered the header of Lfrom outside L instead enter the pre-header. Edges from inside loop L to the header are not changed. Initially the pre-header is empty, but transformations on L may place statements in it.




**(a) Before** **(b) After**

Figure 5.5:Pre-Header

**Reducible flow graphs:**

Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently. Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

***Definition***:

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

- ✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.
- ✓ The back edges consist only of edges where heads dominate theirs tails.
- ✓ Example: The above flow graph is reducible.

If we know the relation DOM for a flow graph, we can find and remove all the back edges. The remaining edges are forward edges. If the forward edges form an acyclic graph, then we can say the flow graph reducible. In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic. The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

## 5.4 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program. A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible. The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

We shall give the following examples of program transformations that are characteristic of peephole optimizations:

Redundant-instructions elimination
Flow-of-control optimizations
Algebraic simplifications
Use of machine idioms
Unreachable Code

**Redundant Loads And Stores:**

If we see the instructions sequence

(1) MOV $R_0$,a

(2) MOV a,$R_0$

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register $R_0$.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

**Unreachable Code:**

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug

0 ….

If ( debug ) {

Print debugging information

}
```

In the intermediate representations the if-statement may be translated as: If

```
debug =1 goto L2

  goto L2

L1: print debugging information

L2:                              ……………………………(a)
```

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
  If debug ≠1 goto L2
  Print debugging information

L2:                              ………………………………(b)
```

As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: ……………………………(c)

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

**Flows-Of-Control Optimizations:**

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

….

L1: gotoL2

by the sequence
goto L2
….
L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1: goto L2

can be replaced by

Ifa < b goto L2

….

L1: goto L2
Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence
goto L1

……..

L1: if a <b goto L2

L3: …………………………………..(1)

Maybe replaced by

Ifa<b goto L2

goto L3

…….

L3: ………………………………….(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

**Algebraic Simplification:**

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

(d) := x+0

Or

x := x * 1

Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

**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.

$X^2 \rightarrow X*X$

**Useof Machine Idioms:**

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++

i:=i-1 → i--

## 5.5 INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

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

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

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

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

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

- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

**Points and Paths:**

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

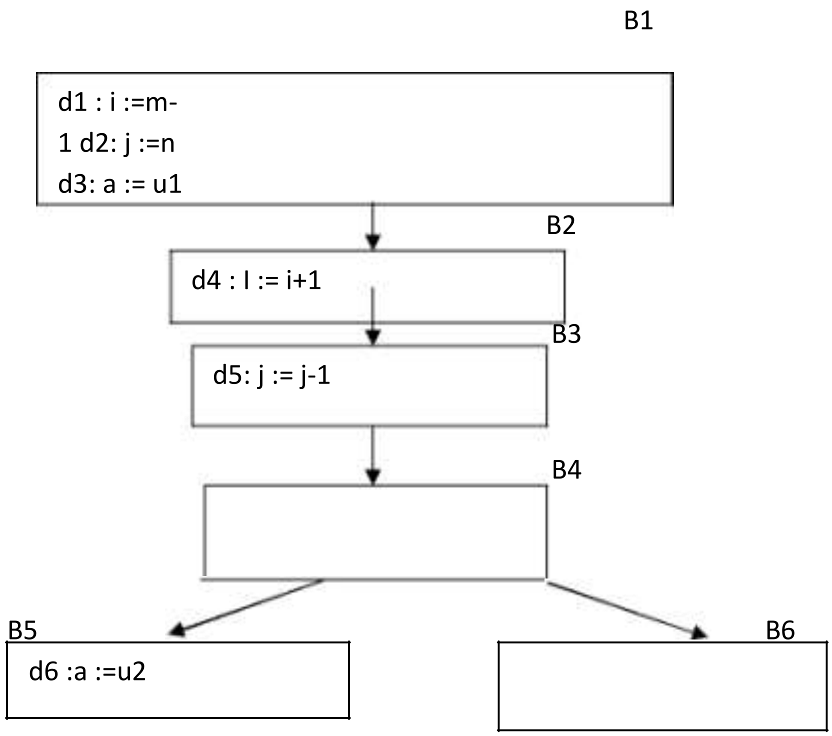



Figure 5.6:Points and paths

Now let us take a global view and consider all the points in all the blocks. A path from $p_1$ to $p_n$ is a sequence of points $p_1$, $p_2$,….,$p_n$ such that for each i between 1 and n-1, either

- $P_i$ is the point immediately preceding a statement and $p_{i+1}$ is the point immediately following that statement in the same block, or
- $P_i$ is the end of some block and $p_{i+1}$ is the beginning of a successor block.

**Reaching definitions:**

A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x. These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

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

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

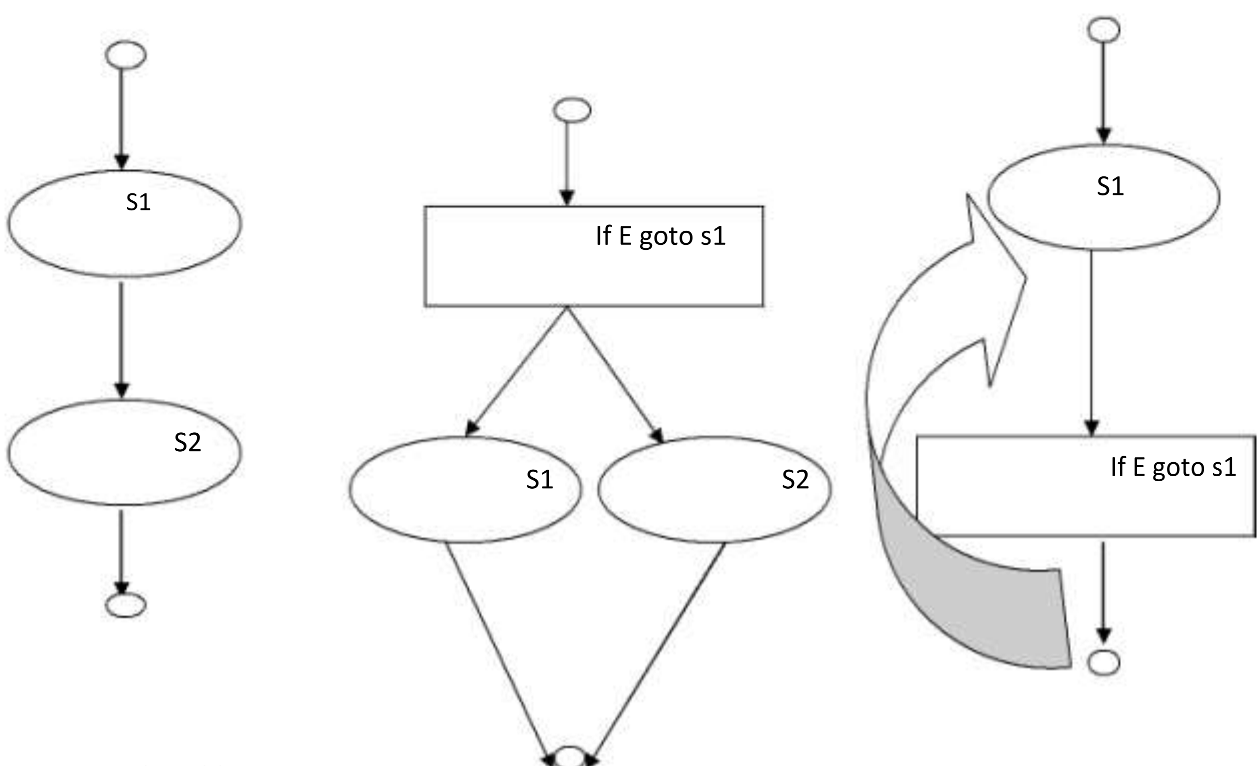
**Data-flow analysis of structured programs:**

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

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

E E →id + id| id

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




S1 ; S2

IF E then S1 else S2 do S1 while E

Figure 5.7:Data flow analysis of structured programs

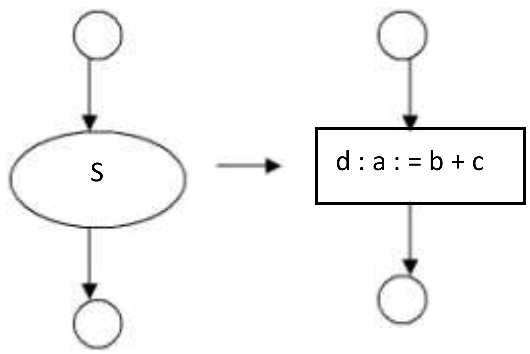
We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header. The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

we say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.

**gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**

Consider the following data-flow equations for reaching definitions :

i )




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

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

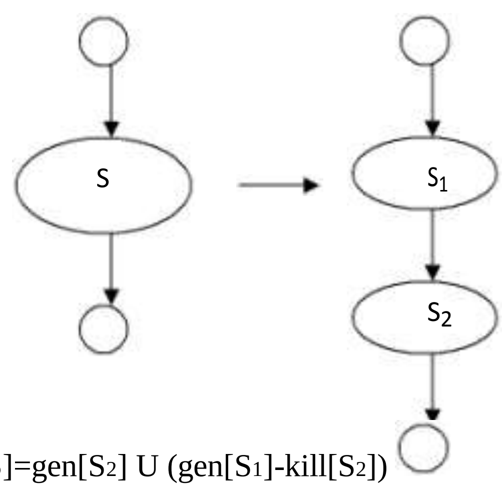
Gen[S]={d}

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

Kill[S] = $D_a$ – {d}

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

ii )




gen[S]=gen[$S_2$] U (gen[$S_1$]-kill[$S_2$])

Kill[S] = kill[$S_2$] U (kill[$S_1$] – gen[$S_2$]

in [$S_1$] = in [S]
in [$S_2$] = out [$S_1$]
out [S] = out [$S_2$]

Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by $S_2$, then it is surely generated by S. if d is generated by $S_1$, it will reach the end of S provided it is not killed by $S_2$. Thus, we write

$$gen[S]=gen[S_2] \cup (gen[S_1]-kill[S_2])$$

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

$$Kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$

**Conservative estimation of data-flow information:**

There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way. We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.

When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill. These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth. Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

**Computation of in and out:**

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

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

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

Considering cascade of two statements $S_1$; $S_2$, as in the second case. We start by observingin[$S_1$]=in[S]. Then, we recursively compute out[$S_1$], which gives us in[$S_2$], since a definition reaches the beginning of $S_2$ if and only if it reaches the end of $S_1$. Now we can compute out[$S_2$], and this set is equal to out[S].

Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of $S_1$ or $S_2$ exactly when it reaches the beginning of S.

In[$S_1$] = in[$S_2$] = in[S]

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

Out[S]=out[$S_1$] U out[$S_2$]

**Representation of sets:**

Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set. The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

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

**Local reaching definitions:**

Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks. Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

**Use-definition chains:**

It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions ofa.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

**Evaluation order:**

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

**General control flow:**

Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner. When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account. Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods. However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

## 5.6 CODE IMPROVIG TRANSFORMATIONS

Algorithms for performing the code improving transformations rely on data-flow information. Here we consider common sub-expression elimination, copy propagation and transformations for moving loop invariant computations out of loops and for eliminating induction variables. Global transformations are not substitute for local transformations; both must be performed.

**Elimination of global common sub expressions:**

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

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

**INPUT: A flow graph with available expression information.**

**OUTPUT: A revised flow graph.**

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

**To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. Thelast evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.**

**Create new variable u.**

**Replace each statement w: =y+z found in (1) by**
**u : = y +**
**z w : = u**

**Replace statement s by x:=u.**

Some remarks about this algorithm are in order.

The search in step(1) of the algorithm for the evaluations of y+z that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.

- ✓ Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.
- ✓ Algorithm will miss the fact that a*z and c*z must have the same value in

a :=x+y          c :=x+y

vs

b :=a*z          d :=c*z

- ✓ Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

**Copy propagation:**

Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x. Statement s must be the only definition of x reaching u. On every path from s to including paths that go through u several times, there are no assignments to y.

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

- **ALGORITHM: Copy propagation.**

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

  **OUTPUT: A revised flow graph.**

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

- ✓ **Determine those uses of x that are reached by this definition of namely, s: x: =y.**
- ✓ **Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B]then s is the only definition of x that reaches B.**
- ✓ **If s meets the conditions of (2), then remove s and replace all uses of x found in (1) by y.**

**Detection of loop-invariant computations:**

Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.

If an assignment x := y+z is at a position in the loop where all possible definitions of y and z are outside the loop, then y+z is loop-invariant because its value will be the same each time x:=y+z is encountered. Having recognized that value of x will not change, consider v := x+w, where w could only have been defined outside the loop, then x+w is also loop-invariant.

- ❖ **ALGORITHM: Detection of loop-invariant computations.**

**INPUT: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.**

**OUTPUT: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.**

**METHOD: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.**

- ✓ **Mark "invariant" those statements whose operands are all either constant or have all their reaching definitions outside L.**
- ✓ **Repeat step (3) until at some repetition no new statements are marked "invariant".**
- ✓ **Mark "invariant" all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.**

**Performing code motion:**

Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.

- ✓ The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
- ✓ There is no other statement in the loop that assigns to x. Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

- ❖ **ALGORITHM: Code motion.**

**INPUT: A loop L with ud-chaining information and dominator information.**

**OUTPUT: A revised version of the loop with a pre-header and some statements moved to the pre-header.**

**METHOD:**

- ✓ **Use loop-invariant computation algorithm to find loop-invariant statements.**

- ✓ **For each statement s defining x found in step(1), check:**
  - **i) That it is in a block that dominates all exits of L,**
  - **ii) That x is not defined elsewhere in L, and**
  - **iii) That all uses in L of x can only be reached by the definition of x in statement s.**
- ✓ **Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii) , to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.**

To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L. When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L. Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s.

**Alternative code motion strategies:**

The condition (1) can be relaxed if we are willing to take the risk that we may actually increase the running time of the program a bit; of course, we never change what the program computes. The relaxed version of code motion condition (1) is that we may move a statement s assigning x only if:

1'. The block containing s either dominates all exists of the loop, or x is not used outside the loop. For example, if x is a temporary variable, we can be sure that the value will be used only in its own block.

If code motion algorithm is modified to use condition (1'), occasionally the running time will increase, but we can expect to do reasonably well on the average. The modified algorithm may move to pre-header certain computations that may not be executed in the loop. Not only does this risk slowing down the program significantly, it may also cause an error in certain circumstances.

Even if none of the conditions of (2i), (2ii), (2iii) of code motion algorithm are met by an assignment x: =y+z, we can still take the computation y+z outside a loop. Create a new temporary t, and set t: =y+z in the pre-header. Then replace x: =y+z by x: =t in the loop. In many cases we can propagate out the copy statement x: = t.

**Maintaining data-flow information after code motion:**

The transformations of code motion algorithm do not change ud-chaining information, since by condition (2i), (2ii), and (2iii), all uses of the variable assigned by a moved statement s that were reached by s are still reached by s from its new position. Definitions of variables used by s are either outside L, in which case they reach the pre-header, or they are inside L, in which case by step (3) they were moved to pre-header ahead of s. If the ud-chains are represented by lists of pointers to pointers to statements, we can maintain ud-chains when we move statement s by simply changing the pointer to s when we move it. That is, we create for each statement s pointer $p_{s,}$ which always points to s. We put the pointer on each ud-chain containing s. Then, no matter where we move s, we have only to change $p_s$ , regardless of how many ud-chains s is on.

The dominator information is changed slightly by code motion. The pre-header is now the immediate dominator of the header, and the immediate dominator of the pre-header is the node that formerly was the immediate dominator of the header. That is, the pre-header is inserted into the dominator tree as the parent of the header.

**Elimination of induction variable:**

A variable x is called an induction variable of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as in a loop headed by for i := 1 to 10. However, our methods deal with variables that are incremented or decremented zero, one, two, or more times as we go around a loop. The number of changes to an induction variable may even differ at different iterations.

A common situation is one in which an induction variable, say i, indexes an array, and some other induction variable, say t, whose value is a linear function of i, is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t. We shall look for basic induction variables, which are those variables i whose only assignments within loop L are of the form i := i+c or i-c, where c is a constant.

- **ALGORITHM: Elimination of induction variable**

**INPUT: A loop L with reaching definition information, loop-invariant computation information and live variable information.**

**OUTPUT: A revised loop.**

**METHOD:**

- ✓ Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i's family, preferably one such that c and d in its triple are as simple as possible and modify each test that i appears in to use j instead. We assume in the following tat c is positive. A test of the form 'if i relop x goto B', where x is not an induction variable, is replaced by

  **r := c*x /* r := x if c is 1. */**

  **r := r+d /* omit if d is 0 */**

  **if j relop r goto B**

where, r is a new temporary. The case 'if x relop i goto B'is handled analogously. If there are two induction variables $i_1$ and $i_2$ in the test if $i_1$ relop $i_2$ goto B, then we check if both $i_1$ and $i_2$ can be replaced. The easy case is when we have $j_1$ with triple and $j_2$ with triple, and $c_1=c_2$ and $d_1=d_2$. Then, $i_1$ relop $i_2$ is equivalent to $j_1$ relop $j_2$.

Now, consider each induction variable j for which a statement j: =s was introduced. First check that there can be no assignment to s between the introduced statement j :=s and any use of j. In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the check. Then replace all uses of j by uses of s and delete statement j:

# ONLINE QUESTIONS

## UNIT-I

| Questions | opt1 | opt2 | opt3 | opt4 | opt 5 | opt 6 | answer |
|---|---|---|---|---|---|---|---|
| ________ translates assembly level language in to an equivalent machine level language | Compiler | Assembler | Loader | Preprocessor | | | Assembler |
| ________ translates high level language in to an equivalent low level language | Compiler | Assembler | Loader | Preprocessor | | | Compiler |
| File inclusion is performed by ________ | Compiler | Assembler | Loader | Preprocessor | | | Preprocessor |
| ________ performs type checking | Lexical analysis | Semantic analysis | Linear analysis | Syntax analysis | | | Semantic analysis |
| Grouping of characters is called ________ | String | Stream | Token | Record | | | Token |
| _______ groups tokens in to grammatical phrases | Parser | Scanner | Analyzer | Processor | | | Parser |
| Example for Token | Syntax | Character | Symbol | Keyword | | | Keyword |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The Idem potent law in regular expression is | R *** = r * | R ** = r *** | R ** = r * | R *** = r ** | | | R ** = r * |
| ________ breaks up the source program into pieces & creates intermediate code representation | Linear phase | Analysis phase | Syntax phase | Synthesis phase | | | Analysis phase |
| ________ constructs the target program from intermediate code representation | Linear phase | Analysis phase | Syntax phase | Synthesis phase | | | Synthesis phase |
| Grouping of tokens into syntactic structure is performed by | Linear analysis | Parser | Scanner | Code optimization | | | Parser |
| _______ transforming parse tree in to intermediate language representation | Three address code | Code generation | Intermediate code generation | Post fix notation | | | Intermediate code generation |
| _______ converts intermediate code in to low level language | Intermediate code generation | Code generation | Assembler | Loader | | | Code generation |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| _______ is the input of structure editors | Sequence of commands | Sequence of characters | Sequence of tokens | String | | | Sequence of commands |
| Pretty printers performs | Printing only | Analyzing and printing | Debugging and printing | Debugging only | | | Analyzing and printing |
| Static checker work is _______ | Debugging and printing | Analyzing and printing | Analyzing and debugging | Debugging only | | | Analyzing and debugging |
| _______ translates high level language in to an equivalent low level Language | Interpreter | Assemble r | Loader | Preprocesso r | | | Interpreter |
| _______ is the input of text formatters | Sequence of commands | Stream of characters | Stream of tokens | Lexeme | | | Stream of characters |
| Query interpreters translates a predicate contains_____ | Boolean operators only | Relational operators only | Relational and Boolean operators | Arithmetic, Relational and Boolean operators | | | Relational and Boolean operators |
| Loader performs ______ | Loading program in to cache memory | Loading program in to main memory | Loading program in to secondary memory | Inking | | | Loading program in to main memory |
| _____ is the linking-editor job | Linking preprocessor directives | Linking library functions | Linking machine code | Linking object modules | | | Linking object modules |
| Parser generators produce _____ | Scanners | Lexical analyzers | Syntax analyzers | Little languages | | | Syntax analyzers |
| ____ uses Scanner generators | Semantic Analyzers | Lexical analyzers | Syntax analyzers | Little languages | | | Lexical analyzers |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Intermediate code generation using ________ tool | Syntax direction engine | Syntax directed trlation engine | Syntax trlation scheme | Syntax directed scheme | | | Syntax directed trlation engine |
| Code Optimization phase using ________ tool | Data flow engine | Automatic code generator | Code generator | Code optimizer | | | Data flow engine |
| Code Generation phase using ________ tool | Data flow engine | Automatic code generator | Code generator | Code optimizer | | | Automatic code generator |
| Lexeme is a ________ | Sequence of characters | Sequence of command s | Set of strings | Pattern | | | Sequence of characters |
| Relational operators is a ________ | Lexeme | Pattern | Token | Character | | | Token |
| Deleting an extraneous character is a action of ______ phase | Lexical | Syntax | Semantic | Synthesis | | | Lexical |
| Trposing two adjacent characters is a action of ______ phase | Semantic | Syntax | Lexical | Synthesis | | | Lexical |
| __________ is an error recovery action in Lexical analysis | Inserting a missing character | Function call return | Semicolon missing | Misspelled keyword | | | Inserting a missing character |
| Sentinel is an __________ | Foe | Foef | Feof | Eof | | | Eof |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ____ is an one way of implementing lexical analyzer | Using Lex | Using Lexeme | Using Yacc | Using Operating System | | | Using Lex |
| The pointer used in buffer pair scheme is | Backward | Forward | Lexeme-End | Lexeme-Start | | | Forward |
| _______ is an example of Computer Alphabets | ASC | EBDCIC | ASCI | EBCDIC | | | EBCDIC |
| Finite sequence of symbols called _______ | String | Character | Sequence | Group | | | String |
| Any set of strings over some fixed alphabet is a _____ | Abstract | Alphabets | Language | Sequence | | | Language |
| Set of letters and digits is represented by | LD | LUD | (LU* | (L* | | | LUD |
| Set of all four letter strings is represented in a language as | L4 | LLLL | L* | (LLLL)* | | | L4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Set of strings including empty string is represented is a language as | L+ | L* | D+ | D* | | | L* |
| _____ is an representation of one or more digits | L+ | L* | D+ | D* | | | D+ |
| If X = class , Y = room then XY is | Class | Room | Class Room | Classroom | | | Classroom |
| Prefix of Banana is | Ban | Ana | Na | Banana | | | Ban |
| _____ is the subsequence of banana | Can | Baaa | Nand | Nanan | | | Baaa |
| Suffix of Banana is | Ban | Baaa | Nana | Banana | | | Nana |
| Substring of banana is | Nan | Baaa | Aaa | Bnn | | | Nan |
| Definition of LUM is | { s\| s is in L or s is in M } | { s\| s is in L and s is in M } | { s\| s is in L nor s is in M } | { s\| s is in L nand s is in M } | | | { s\| s is in L or s is in M } |
| _____ is a notation for Regular Expression | Letter(letterdigit) + | Digit (letterdigit) + | Digit (letter \| digit) * | Letter(letter \| digit) * | | | Letter(letter \| digit) * |
| Definition of LM is | { st \| s is in L or s is in M } | { st \| s is in L and t is in M } | { st \| s is in L or t is in M } | { st \| s is in L and s is in M } | | | { st \| s is in L and t is in M } |
| L* is an representation of _____ | Negative closure | Positive closure | Kleene closure | Line closure | | | Kleene closure |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Positive closure of L is written as | L+ | L* | D+ | D* | | | L+ |
| (r) \| (s) is a regular expression denoting ______ | L(r) \| L(s) | L(r) L(s) | L(r)* L(s) | L(r) U L(s) | | | L(r) U L(s) |
| (r) (s) is a regular expression denoting ______ | L(r) \| L(s) | L(r)L(s) | L(r)* L(s) | L(r) U L(s) | | | L(r)L(s) |
| (r) * is a regular expression denoting ______ | (L(r))* | Lr* | L(r) | L*(r) | | | (L(r))* |
| The regular expression a* denotes | { a } | { ? , a} | { a , aa , aaa, ……} | { ? , a , aa , aaa, ……} | | | { ? , a , aa , aaa, ……} |
| Identifier is represented in character class as | [ A-Z] [ A-Z0-9]* | [ A-Za-z] [ A-Za-z0-9]* | [ A-Za-z] [ A-Za-z]* | [ A-Z] [ A-Za-z0-9]* | | | [ A-Za-z] [ A-Za-z0-9]* |
| ________ cannot be described by a regular expression | ???{ wcw \| w is a string of a's and b's } | { w \| w is a string of a's and b's } | ??{ w* \| w is a string of a's and b's } | ??{ w+ \| w is a string of a's and b's } | | | ???{ wcw \| w is a string of a's and b's } |
| ________ is an associative property of a regular expression | R(s\|t) = (r\|s)t | R\|(s\|t) = (r\|s)t | R\|(s\|t) = (r\|s)\|t | (s\|t)r= t(r\|s) | | | R\|(s\|t) = (r\|s)\|t |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A replacement according to a production is called_______ | Reduction | Production | Derivation | Parse tree | | | Derivation |

## UNIT-II

| Questions | opt1 | opt2 | opt3 | opt4 | opt 5 | opt 6 | answer |
|---|---|---|---|---|---|---|---|
| Process of replacing a string by an NT according to a production_______ | Reduction | Production | Derivation | Parse tree | | | Reduction |
| Which of the following is not a true statement as a derivation tree? | all the leaf nodes are terminals | root node is start symbol | interior node is terminal | interior node is non - terminal | | | interior node is terminal |
| Method of converting regular expression to a recognizer is _______ | Lexical analyzer | Finite automata | Lex | Yacc | | | Finite automata |
| Demerits of using transition table is _______ | tough to implement | slower | takes up less space | takes up lot of space | | | takes up lot of space |
| In which Finite Automata no states has an ? - transition | NFA | DFA | NDFA | DNFA | | | DFA |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ________ has atmost one edge labeled “a” leaving state S | NFA | DFA | NDFA | DNFA | | | DFA |
| Tool used to design the lexical analysis is________ | Yacc | Lexeme | Lex | Yaccer | | | Lex |
| ________ language is used to create Lex program | C | Lex | Yacc | Linux | | | Lex |
| ________ is a c program produced by Lex compiler | Lex.yy.c | Lex.c | a.out | tokens | | | Lex.yy.c |
| ________ is the output produced by C compiler in Lex | Lex.yy.c | Lex.c | a.out | tokens | | | a.out |
| ________ is the input taken by C compiler in Lex | Lex.yy.c | Lex.c | a.out | tokens | | | Lex.yy.c |
| _______ is one of the field in the Lex specification | Transition rules | Translatio n rules | Definition | main function | | | Translation rules |
| ________ is not a part of Lex specification | manifest constants | auxiliary procedure s | declaration s | main function | | | main function |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Build parse trees from root to leaves is called ________ | Top down parsing | Bottom up parsing | LR parsing | Root leaf parsing | | | Top down parsing |
| Input to the top down parsing is scanned from ________ | root to right | left to right | top to left | right to left | | | left to right |
| A ? Aa is called as | Left factoring | Ambiguou s | Left Recursion | Left refactoring | | | Left Recursion |
| Elimination of left recursion for the A ? Aa / ß are ________ | A ? A'a , A' ? ß 'a / ? | A ? ßA' , A' ? aA' / ? | A ? Aa , A' ? A'a / ? | A ? Aa / ß , A ? ß'a / ? | | | A ? ßA' , A' ? aA' / ? |
| Elimination of left recursion for the E ? E+T / T is ______ | E? TE' , E' ? +TE' / ? | E? T'E , E' ? +E / ? | E? T'E' , E' ? +T'E/ ? | E ? TE , E' ? +T'E' / ?: | | | E? TE' , E' ? +TE' / ? |
| Elimination of left recursion for the T ? T * F / F is ______ | T ? FT , T' ? *F'T' / ? | T? F'T , T' ? *E / ? | T? F'T' , T' ? *F'T/ ? | T? FT' , T' ? *FT' / ?\ | | | T? FT' , T' ? *FT' / ?\ |
| Process of factoring out the common prefixes is _____ | Left factoring | Ambiguou s | Left Recursion | Left refactoring | | | Left factoring |
| Elimination of left factoring for the A ? a ß1 / a ß2 are ________ | A ? aßA' , A' ? ß1 / ß2 | A ? a'A , A' ? aß2 / ß1 | A ? aA' , A' ? ß1 / ß2 | A ? a'A' , A' ? a ß1 / a ß2 | | | A ? aA' , A' ? ß1 / ß2 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ____ is the left factored grammar for S? iEtS \| iEtSeS \| a | S? iEtSS' \| a , S'? eS \| ? | S? iEtS' \| a , S'? iEtSS' \| ? | S? iEtSS' \| a , S'? iEtSeS \| ? | S? iEtSeS \| a , S'? eS \| ? | | | S? iEtSS' \| a , S'? eS \| ? |
| Top down parsing is creating the tree in ________ order | post | pre | in | reverse polish | | | pre |
| Top down parsing is used to find________ | post order | Right most derivation | in order | Left most derivation | | | Left most derivation |
| left recursive grammar can cause top down parser to go ________ | elimination | error condition | infinite loop | finite loop | | | infinite loop |
| Difficulty of top down parsing is ________ | Forward loop | For tracking | Backward loop | Back tracking | | | Back tracking |
| ________ is the Top down parsing technique | Recursive descent | Recursion descent | Predicate logic | periodic parsing | | | Recursive descent |
| Demerits of recursive descent parsing is ________ | Backtracking | Left factoring | Recursive | Recursion | | | Recursive |
| Predictive parsing consists of | input , parsing program, parsing table , output | input , stack, parsing table , output | input , parsing program, stack , output | input , parsing program,stack, parsing table | | | input , stack, parsing table , output |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| In predictive parsing program let X be the top stack symbol and a be the next input symbol , if X is terminal and if X= a then ________ | push X on to the stack | push a on to the stack | pop a from the stack | pop X from the stack | | | pop X from the stack |
| In predictive parsing program let X be the top stack symbol and a be the next input symbol , if X is Non - terminal and if M[ X,a] =X?Y1 , Y2 , …. Yk then ________ | push X from the stack , pop Yk , Yk-1 , …. Y1 on to the stack | push X from the stack , pop Y1 , Y2 , …. Yk on to the stack | pop X from the stack , push Yk , Yk-1 , …. Y1 on to the stack | pop X from the stack , push Y1 , Y2 , …. Yk on to the stack | | | pop X from the stack , push Yk , Yk-1 , …. Y1 on to the stack |
| In predictive parsing if X is a terminal , then FIRST(X) is | { ? } | { X } | terminal | nonterminal | | | { X } |
| Two functions used in predictive parsing is ______ | FIRST , LAST | FIRST , FOLLOW | FOLLOW , LAST | FIRST, PREDICT | | | FIRST , FOLLOW |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| In predictive parsing if X ? ?, then FIRST(X) is | { ? } | { X } | terminal | nonterminal | | | { ? } |
| In predictive parsing A? aBß , then everything in _____ is in _____ | FIRST(a) , FOLLOW (B) | FIRST(ß) , FOLLOW (ß) | FIRST(ß) , FOLLOW (B) | FIRST(a) , FOLLOW (ß) | | | FIRST(ß) , FOLLOW (B) |
| In predictive parsing A? aB , then everything in _____ is in _____ | FOLLOW (a), FOLLOW (B) | FOLLOW (A) , FOLLOW (B) | FOLLOW (a) , FOLLOW (A) | FOLLOW (A) , FOLLOW (a) | | | FOLLOW (A) , FOLLOW (B) |
| In predictive parsing A? aBß, where FIRST(ß) contains ? , then everything in ______ is in _____ | FOLLOW (A) , FOLLOW (B) | FOLLOW (a), FOLLOW (B) | FOLLOW (a) , FOLLOW (A) | FOLLOW (A) , FOLLOW (a) | | | FOLLOW (A) , FOLLOW (B) |
| _____ is included in FOLLOW function set of predictive parsing | % | # | $ | @ | | | $ |
| If a grammar S ? (L) \| a , then FIRST(S) is _____ | { ( , a } | { ? } | { L , a } | { $ } | | | { ( , a } |
| Elimination of left recursion for the L ? L,S \| S is ______ | L? S,L' , L'? LL' / ? | L? S',L , L'? S,L / ? | L? SL' , L'? SL' / ? | L? S',L' , L'? LL / ? | | | L? SL' , L'? SL' / ? |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| LL(1) grammar has the property of ______ | Ambiguity | No ambiguity | No recursion | No backtracking | | | No ambiguity |
| LL(1) grammar has ______ property | Ambiguity | Left recursive | No recursion | No Left recursive | | | No Left recursive |
| In LL(1) grammar the first L stands for ______ | left to right scanning | left most derivation | left to right derivation | left subtree | | | left to right scanning |
| In LL(1) grammar the second L stands for ______ | left to right scanning | left most derivation | left to right derivation | left subtree | | | left most derivation |
| In LL(1) grammar the 1 stands for ______ | one output symbol | one time scanning | one sub tree | one input symbol | | | one input symbol |
| In predictive parsing , If X = a = $ | error report | successful completio n | advances pointer | pop X off the stack | | | successful completion |
| In predictive parsing , If X = a ? $ | error report | successful completio n | advances pointer & pop X off the stack | pop X off the stack & parser halts | | | advances pointer & pop X off the stack |
| In predictive parsing , If X = $ | Stack is empty | Stack is full | error report | pop X from Stack | | | Stack is empty |
| Ambiguity means ______ | produces more than two parse tree | produces null parse tree | produces more than one parse tree | produces finite parse tree | | | produces more than one parse tree |
| Ambiguous means ______ | produces more than two derivation | produces only left most derivation | produces more than one derivation | produces only right most derivation | | | produces more than one derivation |
| The output of Lex compiler is ______ | Transition table | Transition diagram | Lex specificatio n | action | | | Transition table |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The input of Lex compiler is ______ | Transition table | Transition diagram | Lex specification | action | | | Lex specification |
| The lexical analyzer design has ______ | Lex compiler | Transition table | Lex specification | Transition diagram | | | Transition table |
| Recognizer is a ______ | tool | program | string | grammar | | | program |
| NFA representation in a directed graph is called________ | NFA graph | NFA direction graph | Transition edge | Transition graph | | | Transition graph |
| ________ is an easiest implementation of NFA in a computer. | Transition diagram | Transition table | Transition graph | Finite automata | | | Transition table |
| ________ is an input for the syntax analysis | token | source program | parse tree | syntax | | | token |
| ________ is an output of the parser | expression | token | parse tree | intermediate representation | | | parse tree |
| Parser is a ________ | Lexical analyzer | Front end tool | Scanner | Back end tool | | | Front end tool |
| ________ is syntax error | misspelled identifier | misspelled keyword | misspelled operator | unbalanced parenthesis | | | unbalanced parenthesis |

## UNIT-III

| Questions | opt1 | opt2 | opt3 | opt4 | opt 5 | opt 6 | answer |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| _______ is one of the method of parsing | left to right parsing | Top down parsing | Bottom down parsing | Top up parsing | | | Top down parsing |
| ______ constructs parse tree from leaf node to root | Bottom up parsing | Top down parsing | Bottom down parsing | Top up parsing | | | Bottom up parsing |
| one of the goal of error handler in parser is | avoid common error | slow down the process | report the presence of error | avoid specific error | | | report the presence of error |
| In which error recovery strategy parser discards one symbol at a time ? | Panic mode | phrase level | error production s | global corrections | | | Panic mode |
| In which error recovery strategy parser makes local corrections ? | Panic mode | phrase level | error production s | global corrections | | | phrase level |
| In which error recovery strategy parser generate error diagnostics ? | Panic mode | phrase level | error production s | global corrections | | | error productions |
| In which error recovery strategy parser using algorithmic approach ? | Panic mode | phrase level | error production s | global corrections | | | global corrections |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The syntactic structure of a programming language is described by | Context free grammar | CNF | CCNF | Regular language | | | Context free grammar |
| Context free grammar consists of ______ | T , NT, $ , P | T, NT, S , Production | Terminal , token , Non terminal | Terminal , Token , production | | | T, NT, S , Production |
| The keyword else is a _______ | Terminal | Non terminal | Start symbol | Production | | | Terminal |
| _______ are syntactic variable | Production | Non terminal | Terminal | string | | | Non terminal |
| Non terminal denotes _______ | sets of characters | sets of grammar | sets of production | sets of strings | | | sets of strings |
| Start symbol is a _______ | Production | Non terminal | Terminal | string | | | Non terminal |
| Upper case letters are _______ | Production | Non terminal | Terminal | string | | | Non terminal |
| Lower case letters are _______ | Production | Non terminal | Terminal | string | | | Terminal |
| Punctuation symbols are_____ | Production | Non terminal | Terminal | string | | | Terminal |
| Boldface strings are _____ | Terminal | Non terminal | Start symbol | Production | | | Terminal |
| operator symbols are _____ | Terminal | Non terminal | Start symbol | Production | | | Terminal |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Lower case italic names are ____ | Production | Non terminal | Terminal | string | | | Non terminal |
| The left side of the first production is called ________ | String | Terminal | Non terminal | Start symbol | | | Start symbol |
| Lower case Greek letters represents________ | Grammar symbols | Terminals | Non terminals | Start symbol | | | Grammar symbols |
| sequence of replacements is called ________ | Reduction | Derivation | parse tree | sentence | | | Derivation |
| Graphical representation of a derivation is a ________ | parse tree | syntactic tree | syntax graph | pattern | | | parse tree |
| substring that matches the right side of a production is called ________ | Handle pruning | Pattern | Handle | parsing | | | Handle |
| Right most derivation in reverse is called______ | Handle pruning | Pattern | Handle | parsing | | | Handle pruning |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| In______ action the next input symbol is shifted on to the top of the stack | accept | reduce | shift | error | | | shift |
| In______ action the parser announces the successful completion of parsing | accept | reduce | shift | error | | | accept |
| In______ action parser discovers the syntax error | accept | reduce | shift | error | | | error |
| In______ action parser replacing the handle with the non terminal | accept | reduce | shift | error | | | reduce |
| The set of prefixes that appear on the stack are called ______ | prefixes | viable prefixes | reduce conflict | viable suffixes | | | viable prefixes |
| Grammar that has no production right side is e is called _______ | operator grammar | shift reduce grammar | operator precedenc e grammar | precedenc e grammar | | | operator grammar |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Grammar that has no production right side is two adjacent non terminals is called ________ | operator precedence grammar | shift reduce grammar | operator grammar | precedence grammar | | | operator grammar |
| In precedence relation a < b means ______ | a has different precedence to b | a has same precedence as b | a takes precedence over b | a yields precedence to b | | | a yields precedence to b |
| In precedence relation a > b means ______ | a has different precedence to b | a has same precedence as b | a takes precedence over b | a yields precedence to b | | | a takes precedence over b |
| In precedence relation a = b means ______ | a has different precedence to b | a has same precedence as b | a takes precedence over b | a yields precedence to b | | | a has same precedence as b |
| In precedence relation * and + has the precedence of ______ | * > + | + > * | + = + | * = * | | | * > + |
| In precedence relation $ and id has the precedence of ______ | $ > id | $ =id | $ <id | $ ? id | | | $ <id |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| In operator precedence parsing if a < b then | pop a from the stack | push a on to the stack | pop b from the stack | push b on to the stack | | | push b on to the stack |
| In operator precedence parsing if a = b then | pop a from the stack | push b on to the stack | pop b from the stack | push a on to the stack | | | push b on to the stack |
| In operator precedence parsing if a > b then | pop the stack | push the stack | pop b from the stack | push a on to the stack | | | pop the stack |
| In LR(k) parsing , the L stands for ______ | left to right scanning | left most derivation | left to right derivation | left subtree | | | left to right scanning |
| In LR(k) parsing , the R stands for ______ | left to right scanning | Right most derivation | left to right derivation | Right most derivation in reverse | | | Right most derivation in reverse |
| In LR(k) parsing , the k stands for ______ | parsing symbol | number of input symbols | number of characters | look ahead symbol | | | number of input symbols |
| ______ LR technique is easy to implement | SLR | canonical LR | LALR | Look ahead LR | | | SLR |
| ______ LR technique is most powerful | SLR | canonical LR | LALR | Look ahead LR | | | canonical LR |
| ______ LR technique is most expensive | SLR | canonical LR | LALR | Look ahead LR | | | canonical LR |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ______ LR technique is least powerful | SLR | canonical LR | LALR | Look ahead LR | | | SLR |
| S' ? .S is included in ______ items | closure | non kernel | kernel | non closure | | | kernel |
| The functions performed in LR parsing are ______ | action and shift | action and goto | action and error | goto and shift | | | action and goto |
| Action function involved with ______ | Terminals | Non terminals | start symbol | production | | | Terminals |
| Goto function involved with ______ | Terminals | Non terminals | Start symbol | Production | | | Non terminals |
| LALR is abbreviated from ______ | Left and Right LR | Look ahead Simple LR | Look ahead LR | Left to Right simple LR | | | Look ahead LR |
| ________ is a combination of terminals and non terminals | ?productio n | token | regular expression | regular definition | | | ?production |
| ________ is an one of the bottom up parsing technique | Operator parsing | Shift reduce parsing | Recursive descent parsing | Predictive parsing | | | Shift reduce parsing |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| _______ translates intermediate representation in to an equivalent low level language | Analyzer | Front end | Back end | Synthesizer | | | Back end |
| The input for the intermediate code generator is _______ | Optimized code | Intermediate Code | Token | Meaningful expression | | | Meaningful expression |
| The output for the intermediate code generator is _______ | Optimized code | Intermediate Code | Token | Meaningful expression | | | Intermediate Code |
| In _______ tree the operators represented in the interior node | Postfix | Parse Tree | Syntax Tree | Prefix | | | Syntax Tree |
| ______ is a linearized representation of a syntax tree | Postfix Notation | Parse Tree | Two address code | Prefix notation | | | Postfix Notation |
| Postfix notation for the statement a = b * - c is ______ | * uminus a b c | uminus * a b c | a b c * uminus | a b c uminus * | | | a b c uminus * |

# UNIT-IV

| Questions | opt1 | opt2 | opt3 | opt4 | opt5 | opt6 | answer |
|---|---|---|---|---|---|---|---|
| Semantic rule to produce syntax tree for the production E ? id is ______ | E.nptr = mkleaf ( id.place) | E.nptr = mknode ( id.place) | E.nptr = mkleaf ( id , id.place) | E.nptr = mknode (id , id.place) | | | E.nptr = mkleaf ( id , id.place) |
| Semantic rule to produce syntax tree for the production E ? - E1 is ______ | E.nptr = mknode (‘uminus’ , E1.nptr ) | E.nptr = mkunode (‘uminus’ , E1.nptr ) | E.nptr = mkleaf (‘uminus’ , E1.nptr ) | E.nptr = mkuleaf (‘uminus’ , E1.nptr ) | | | E.nptr = mkunode (‘uminus’ , E1.nptr ) |
| Semantic rule to produce syntax tree for the production E ? E1 + E2 is ______ | E.nptr = mknode (‘+’ , E1.nptr ) | E.nptr = mkpnode (‘+’ , E1.nptr ) | E.nptr = mkpnode (‘+’ , E1.nptr , E2.nptr ) | E.nptr = mknode (‘+’ , E1.nptr , E2.nptr ) | | | E.nptr = mknode (‘+’ , E1.nptr , E2.nptr ) |
| _______ is an general form of three address code representation | x = y op z | x = z op | op x = z op | op x = op y z | | | x = y op z |
| _______ is an one type of three address code statements | if x y goto L | if x relop y goto L | goto L if x y | goto L if x relop y | | | if x relop y goto L |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The name that will hold the value of E is called ________ | E.place | E.code | place.E | code.E | | | E.place |
| The sequence of three address statements evaluating E is called______ | E.place | E.code | place.E | code.E | | | E.code |
| Record structure with four fields is called______ | Three address code | Quadruples | Triples | Indirect triples | | | Quadruples |
| Three fields of Record structure is called______ | Three address code | Quadruples | Triples | Indirect triples | | | Triples |
| emit is used to ______ | emit 3address statements to an output file | emit assignments to an output file | emit terminals to an output file | emit non terminals to an output file | | | emit 3address statements to an output file |
| Translation of E?(E1) is ________ | E1.place = E.place | E.place=E1.place | E.place = (E1.place) | (E1.place)= E.place | | | E.place=E1.place |
| Translation of E? id is ________ | E1.place = E.place | E.place=E1.place | E.place = id.place | E1.place= id.place | | | E.place = id.place |
| _______ are the 3 fields of triples | arg1,arg2,arg3 | arg1,arg2,result | arg1,op,result | arg1,arg2,op | | | arg1,arg2,op |
| _______ are the 4 fields of Quadruples | arg1,arg2,arg3 , arg4 | arg1,arg2,arg3,result | arg1,,arg2,op,result | arg1,arg2,arg3,op | | | arg1,,arg2,op,result |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Listing pointers to triples is called ______ | Indirect triples | triples | Quadruples | Indirect ruples | | | Indirect triples |
| offset represents_____ | relative address | three address | location | address | | | relative address |
| E1.type=integer ,E2.type = integer , E.type is _____ | integer | real | inttoreal | float | | | integer |
| E1.type=integer ,E2.type = real , E.type is _____ | integer | real | inttoreal | float | | | real |
| E1.type=real,E2 .type = real , E.type is _____ | integer | real | inttoreal | float | | | real |
| E1.type=real,E2 .type = integer, E.type is _____ | integer | real | inttoreal | float | | | real |
| E?E1 or E2 represents _____ | E1.place = E1.place * E2.place | E.code= E1.place or E2.place | E1.code= E1.place and E2.place | E.place= E1.place or E2.place | | | E.place= E1.place or E2.place |
| E? true represents _________ | E.code=0 | E.place = 0 | E.place = 1 | E.code=1 | | | E.place = 1 |
| E? false represents _________ | E.code=0 | E.place = 0 | E.place = 1 | E.code=1 | | | E.place = 0 |
| E?not E1 represents _____ | E1.place = E1.place not E2.place | E.code= E1.place not E2.place | E1.code= not E1.place | E.place= not E1.place | | | E.place= not E1.place |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| __________ is a one semantic rule of S?if E then S1 | E.false=newlabel | E.true=newlabel | S.next=S1.next | S.code=E.place B | | | E.true=newlabel |
| __________ is a one of the semantic rule of S?if E then S1 | E.false=newlabel | S1.next=S.next | S.next=S1.next | S.code=E.place | | | S1.next=S.next |
| __________ is a one of the semantic rule of S? while E do S1 | E.false=newlabel | S1.next=S.next | S.next=S1.next | E.false=S.next | | | E.false=S.next |
| __________ is a one of the semantic rule of S? while E do S1 | E.false=newlabel | S1.next=S.next | E.true=newlabel | E.false=S1.next | | | E.true=newlabel |
| __________ is a one of the semantic rule of S?if E then S1 else S2 | E.false=S1.next | S1.next=S.next | S.next=S1.next | S.code=E.code | | | S.code=E.code |
| __________ is a one of the semantic rule of S?if E then S1 else S2 | E.false=E.true | S1.next=S.next | S.next=S1.next | S.code=E.place | | | S1.next=S.next |
| __________ is a one of the semantic rule of S? while E do S1 | E.false=S.next | S1.next=S2.next | E.true=E.false | E.false=S1.next | | | E.false=S.next |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| __________ is a one of the semantic rule of E?E1 or E2 | E1.true=E.t rue | E1.false=E.fal se | E2.true=E.fal se | E2.fasle=E 1.true | | | E1.true=E.tr ue |
| __________ is a one of the semantic rule of E?E1 or E2 | E1.true=E.f alse | E1.false=E.fal se | E2.true=E.tr ue | E2.fasle=E 1.true | | | E2.true=E.tr ue |
| __________ is a one of the semantic rule of E?E1 or E2 | E1.true=E.f alse | E1.false=E.fal se | E2.true=E1.t rue | E2.fasle=E. false | | | E2.fasle=E.f alse |
| __________ is a one of the semantic rule of E?E1 and E2 | E1.true=E.f alse | E1.false=E.fal se | E2.true=E1.t rue | E2.fasle=E 1.false | | | E1.false=E.f alse |
| __________ is a one of the semantic rule of E?E1 and E2 | E1.true=E.f alse | E1.false=E.tr ue | E2.true=E1.t rue | E2.fasle=E. false | | | E2.fasle=E.f alse |
| __________ is a one of the semantic rule of E?E1 and E2 | E1.true=E.f alse | E1.false=E2.f alse | E2.true=E.tr ue | E2.fasle=E 1.false | | | E2.true=E.tr ue |
| __________ is a one of the semantic rule of E?E1 and E2 | E1.true=ne wlabel | E1.false=E2.f alse | E2.true=E1.t rue | E2.fasle=E. true | | | E1.true=new label |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| __________ is a one of the semantic rule of E? not E1 | E.true= E1.false | E1.true = E.false | E.false= E2.true | E.code=E.p lace | | | E1.true = E.false |
| __________ is a one of the semantic rule of E? not E1 | E1.true= E.false | E1.true = E2.false | E.false= E2.true | E.code=E.p lace | | | E1.true= E.false |
| The use of makelist(i) is | creates a new list containing quadruples | creates a new list containing only i | creates a new list by inserting i | creates a new list pointed to p1 | | | creates a new list containing only i |
| ____ is the use of merge(p1,p2) | concatenat es the lists pointed by p1 and p2 | merge the list containg only i | merge the list pointed by p1 | merge the list containing quadruples | | | concatenate s the lists pointed by p1 and p2 |
| The use of backpatch(p,i) is | concatenat es the lists pointed by p1 and p2 | merge the list containing only i | inserts I as the target label | merge the list containing quadruples | | | inserts I as the target label |
| __________ is a one of the semantic rule of E? not E1 | E.truelist= E1.falselist | E1.truelist = E2.falselist | E.falselist= E2.truelist | E.code=E.p lace | | | E.truelist= E1.falselist |
| __________ is a one of the semantic rule of E? not E1 | E1.truelist= E1.falselist | E1.truelist = E2.falselist | E.falselist= E1.truelist | E.code=E.p lace | | | E.falselist= E1.truelist |
| _________ is a one of the semantic rule of E? (E1) | E.truelist= E1.truelist | E1.truelist = E2.falselist | E.falselist= E2.truelist | E.code=E.p lace | | | E.truelist= E1.truelist |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| __________ is a one of the semantic rule of E? (E1) | E2.truelist= E1.truelist | E1.truelist = E2.falselist | E.falselist= E1.falselist | E.code=E.p lace | | | E.falselist= E1.falselist |
| translation of s?begin L end is ______ | S.list = L.nextlist | S.nextlist=L.n extlist | L.List = S.Listnext | L.nextlist=S .list | | | S.nextlist=L. nextlist |
| The translation of Elist?Elist,E is | append E.code to the end of the queue | append E.place to the beginning of the queue | append E.code to the beginning of the queue | append E.place to the end of queue | | | append E.place to the end of queue |
| The translation of Elist? E is | Initialize E.code to the end of the queue | Initialize E.place to the beginning of the queue | Initialize E.code to the beginning of the queue | Initialize queue to contain only E.place | | | Initialize queue to contain only E.place |
| The translation of M? ? with quadruple is | M.quad = nextQuad | M.nextquad = nextQuad | M.next = M.Quad | M.quad = M.nextQua d | | | M.quad = nextQuad |
| The translation of N? ? with quadruple is | N.nextlist = makelist(ne xtqua | N.nextlist = list(qua | N.nextlist = makelist(M.q ua | N.nextlist = make (nextqua | | | N.nextlist = makelist(nex tqua |
| ________ is an input of code generation phase | optimized code | target code | source program | object code | | | optimized code |
| The output for the code generator is ________ | Optimized code | Intermediate Code | Token | Assembly language | | | Assembly language |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The transformation performed only within a basic block is called________ | local | global | preserve | optimization | | | local |
| Eliminating the same sub expressions is called ________ | common elimination | common sub expression elimination | common expression deletion | common sub expression deletion | | | common sub expression elimination |
| _______ is a transformations of copy statements | copy propagatio n | copy transformatio n | copy for long | copy elimination | | | copy propagation |
| Useless code transformation is called _______ | usecode elimination | dead elimniation | useless code elimination | Deadcode elimination | | | Deadcode elimination |
| Using the constant and deducing during compile time is called _______ | dead code elimination | copy propagation | constant folding | constant propagation | | | constant folding |
| Optimizing inner loops named as _______ | Loop transformat ion | Loop optimization | Deadcode elimination | copy propagation | | | Loop optimization |

## UNIT-V

| Questions | opt1 | opt2 | opt3 | opt4 | opt 5 | opt 6 | answer |
|---|---|---|---|---|---|---|---|
| Decreasing the amount | Induction | Reduction | Loop motion | Code motion | | | Code motion |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| of code in a inner loop is called as _______ | | | | | | | |
| _______ is an one way of loop optimization | Induction variable elimination | Copy propagation | Deadcode elimination | constant folding | | | Induction variable elimination |
| _______ is an loop optimization technique | Reduction variable elimination | Reduction in strength | Deadcode elimination | constant folding | | | Reduction in strength |
| The Expansion for DAG is _______ | Directed Acyclic Graph | Directed Action Graph | Direction Asymmetric Graph | Direction Action Graph | | | Directed Acyclic Graph |
| The Algebraic transformation includes _______ | Algebraic Deduction | Algebraic Identities | constant folding | reduction in strength | | | Algebraic Identities |
| The output for the code generation phase is _______ | Optimized code | Intermediate Code | Machine level language | Token | | | Machine level language |
| _______ is the input of code generation phase | Optimized code | Intermediate Code | Machine level language | Token | | | Intermediate Code |
| The use of symbol table is | to determine the run time addresses of the data objects | to determine the run time value of the data | to determine the compile time value of the data | to determine the compile time addresses of the data objects | | | to determine the run time addresses of the data objects |
| _______ is a linear representations of intermediate code | Prefix notation | Infix notation | Postfix notation | RP notation | | | Postfix notation |
| _______ is an representation of three address code | Quadruples | Indirect Quadruples | Postfix notation | Linear | | | Quadruples |
| _______ is the virtual machine representation | Sequence of commands | Stack machine code | Machine code | Stack code | | | Stack machine code |
| _____ is a Graphical | DEG tree | Parsing | Syntax trees | Linear tree | | | Syntax trees |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| representation of three address code | | | | | | | |
| ______ is a Graphical representation of intermediate code | DAG | Parsing | Semantic tree | Linear tree | | | DAG |
| ______ is the output form of a target program | Intermediate code | linking library functions | linking machine code | Absolute machine code | | | Absolute machine code |
| ______ is an one of the output form of a target program | Intermediate code | linking library functions | Re locatable machine code | Absolute intermediate code | | | Re locatable machine code |
| Semantic checking done in _____ | Intermediate code generator | Lexical analyzers | Syntax analyzers | Code generator | | | Code generator |
| Mapping names to addresses of data objects is done by _________ | intermediate code generator | code generator | code optimizer | Lexical analysis | | | code generator |
| Deducing the number of jumping labels is done by _________ | Backpatching | Quadruple | Triple | Indirect Triple | | | Backpatching |
| The speed is increased based on instruction selection by using _________ | Assignment | Machine idioms | Structure | Register | | | Machine idioms |
| During Register Allocation | we select variables reside in the register | we pick specific register that a variable reside in | choose register pairs | Allocate constants to register | | | we select variables reside in the register |
| During Register assignment | We select variables reside in the register | We pick specific register that a variable reside in | Choose registers pairs | Allocate constants to register | | | We pick specific register that a variable reside in |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SRDA stands for _______ | Shift Right Double Arithmetic | Shift Round Direct Arithmetic | Scan Right Double Arithmetic | Scan Round Direct Arithmetic | | | Shift Right Double Arithmetic |
| _______ is used to improve the efficiency | Choice of Run | Syntax | Choice of Evaluation order | Semantic | | | Choice of Evaluation order |
| _________ is an two address instruction form | op source,destination | source op destination | source,destination op | destination source,op | | | op source,destination |
| ADD is an __________ | ADD to register | ADD destination to memory | ADD to memory | ADD source to destination | | | ADD source to destination |
| SUB is an __________ | SUB to register | SUB destination to memory | SUB to memory | SUB source from destination | | | SUB source from destination |
| For absolute mode the added cost is ________ | 1 | 0 | 2 | 3 | | | 1 |
| For Register mode the added cost is ________ | 1 | 0 | 2 | 3 | | | 0 |
| For Indexed mode the added cost is ________ | 1 | 0 | 2 | 3 | | | 1 |
| For Indirect indexed mode the added cost is ________ | 1 | 0 | 2 | 3 | | | 1 |
| The form for absolute mode is ________ | c(R) | *R | R | M | | | M |
| The form for Register mode is ________ | c(R) | *R | R | M | | | R |
| The form for Indexed mode is ________ | c(R) | *R | R | M | | | c(R) |
| The form for Indirect Register mode is ________ | c(R) | *R | R | M | | | *R |
| The form for Indirect | *c(R) | *R | R | M | | | *c(R) |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Indexed mode is ________ | | | | | | | |
| The address of Register mode is ________ | c(R) | *R | R | M | | | R |
| The address of Indexed mode is ________ | c + contents of (R) | R | contents of R | M | | | c + contents of (R) |
| The address of Indirect Register mode is ________ | c + contents of (R) | R | contents of (R) | M | | | contents of (R) |
| The address of Indirect Indexed mode is ________ | contents (c + contents of (R)) | R | contents of R | M | | | contents (c + contents of (R)) |
| The cost of MOV R0,R1 is ________ | 1 | 0 | 2 | 3 | | | 1 |
| The cost of MOV R5,M is ________ | 1 | 0 | 2 | 3 | | | 2 |
| The cost of ADD #1,R3 is ________ | 1 | 0 | 2 | 3 | | | 2 |
| The cost of SUB 4(R0), *12(R1) is ________ | 1 | 0 | 2 | 3 | | | 3 |
| The cost of MOV b,a and ADD c,a is ________ | 1 | 0 | 6 | 3 | | | 6 |
| The cost of ADD R2,R1 and MOV R1,a is ________ | 1 | 0 | 2 | 3 | | | 3 |
| The getreg denotes ________ | to determine the location L | to determine the value | to determine the Register | to determine the memory | | | to determine the location L |
| The function of register descriptor is ________ | to keep track of the location | to keep track of what is currently in each register | to keep track of the register | to keep track of the descriptor value | | | to keep track of what is currently in each register |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| The function of register descriptor is _______ | to keep track of the location | to keep track of what is currently in each register | to keep track of the register | to keep track of the descriptor value | | | to keep track of the location |
| For the statement t = a – b , the value of Address Descriptor is_______ | R0 contains t | u in R0 | t in R0 | R0 contains u | | | t in R0 |