A Course Material on

Principles of Compiler Design

By

Mrs.K.UMA MAHESWARI

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SASURIE COLLEGE OF ENGINEERING

VIJAYAMANGALAM – 638 056

QUALITY CERTIFICATE

This is to certify that the e-course material

Subject Code : CS2352

Subject : Principles of Compiler Design

Class : III Year CSE

being prepared by me and it meets the knowledge requirement of the university curriculum.

Signature of the Author

Name: K.Uma Maheswari

Designation: Assistant Professor

This is to certify that the course material being prepared by Mrs.K.Uma Maheswari is of adequate quality. He has referred more than five books among them minimum one is from abroad author.

Signature of HD

Name: P.Murugapriya

CS2352 PRINCIPLES OF COMPILER DESIGN L T P C 3 0 2 4

UNIT I LEXICAL ANALYSIS 9

Introduction to Compiling- Compilers-Analysis of the source program-The phases- Cousins-The grouping of phases-Compiler construction tools. The role of the lexical analyzer- Input buffering-Specification of tokens-Recognition of tokens-A language for specifying lexical analyzer.

UNIT II SYNTAX ANALYSIS and RUN-TIME ENVIRONMENTS 9

Syntax Analysis- The role of the parser-Context-free grammars-Writing a grammar-Top- down parsing-Bottom-up Parsing-LR parsers-Constructing an SLR(1) parsing table.

Type Checking- Type Systems-Specification of a simple type checker. Run-Time Environments-Source language issues-Storage organization-Storage-allocation strategies.

UNIT III INTERMEDIATE CODE GENERATION 9

Intermediate languages-Declarations-Assignment statements - Boolean expressions- Case statements- Backpatching-Procedure calls

UNIT IV CODE GENERATION 9

Issues in the design of a code generator- The target machine-Run-time storage management-Basic blocks and flow graphs- Next-use information-A simple code generator-Register allocation and assignment-The dag representation of basic blocks - Generating code from dags.

UNIT V CODE OPTIMIZATION 9

Introduction-The principle sources of optimization-Peephole optimization- Optimization of basic blocks-Loops in flow graphs- Introduction to global data-flow analysis-Code improving transformations.

TEXT BOOK:

1. Alfred V. Aho, Ravi Sethi Jeffrey D. Ullman, “Compilers- Principles, Techniques, and Tools”, Pearson Education Asia, 2007.

REFERENCES:

1. David Galles, “Modern Compiler Design”, Pearson Education Asia, 2007.
2. Steven S. Muchnick, “Advanced Compiler Design & Implementation”, Morgan Kaufmann Pulishers, 2000.
3. C. N. Fisher and R. J. LeBlanc “Crafting a Compiler with C”, Pearson Education, 2000.

UNIT I - LEXICAL ANALYSIS
INTRODUCTION TO COMPILING

Objectives:

- To describe the concept of a compiler
- To know the environment where compilers do their job
- To know the software tools that make it easier to build compilers

1.1 COMPILERS

A compiler is a program that reads a program written in one language-the source language-and translates it into an equivalent program in another language-the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

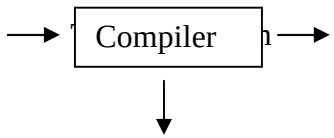


Fig. 1.1 A Compiler

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same.

The Analysis – Synthesis Model of Compilation

There are two parts of compilation.

- Analysis part
- Synthesis Part

The analysis part breaks up the source program into constant piece and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

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.

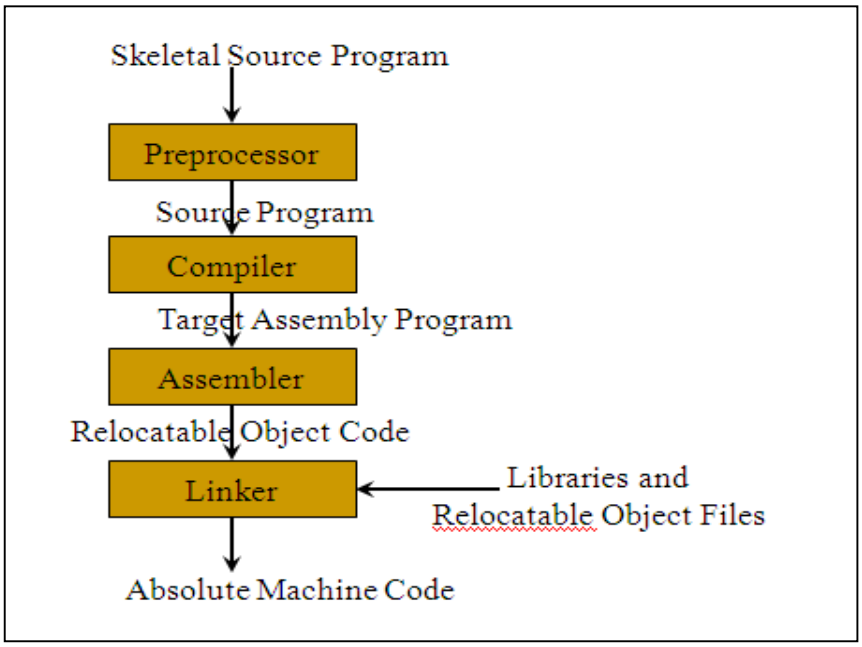


Fig. 1.2 A language processing system

1.2 ANALYSIS OF THE SOURCE PROGRAM

In Compiling, analysis consists of three phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

1.Lexical analysis:

In a compiler linear analysis is called lexical analysis or scanning. The lexical analysis phase reads the characters in the source program and grouped into them tokens that are sequence of characters having a collective meaning.

Example :

position : = initial + rate * 60

Identifiers – position, initial, rate.

Assignment symbol - : =

Operators - + , *
Number - 60
Blanks – eliminated.

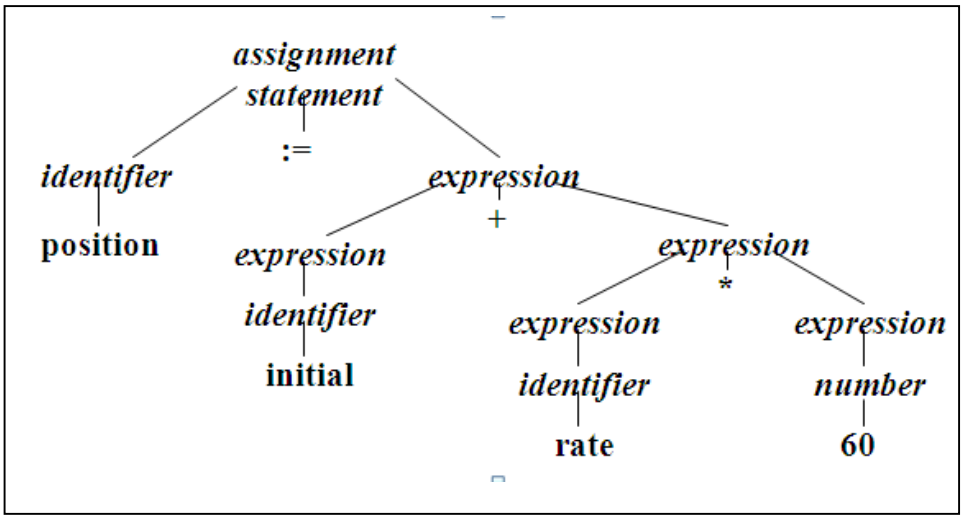


Fig. 1.3 Parse tree for position:=initial+rate*60

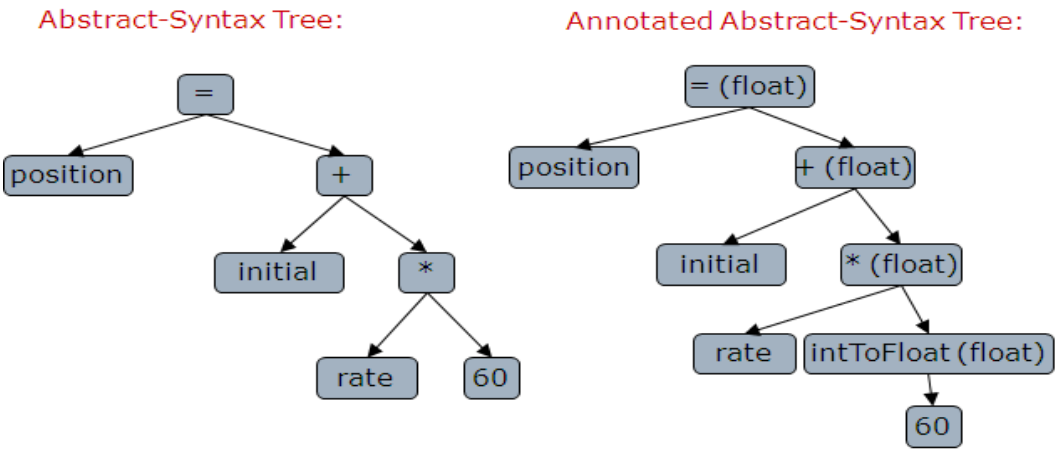


Fig. 1.4 Semantic analysis inserts a conversion from integer to real

2.Syntax analysis:

Hierarchical Analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output. They are represented using a syntax tree as shown in Fig. 1.3

- 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.

Example :
position : = initial + rate * 60
3.Semantic analysis :

This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase. 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 THE PHASES OF A COMPILER

1. Lexical analysis ("scanning")
   Reads in program, groups characters into "tokens"
2. Syntax analysis ("parsing")
   Structures token sequence according to grammar rules of the language.
3. Semantic analysis
   Checks semantic constraints of the language.
4. Intermediate code generation
   Translates to "lower level" representation.
5. Program analysis and code optimization
   Improves code quality.
6. Final code generation.

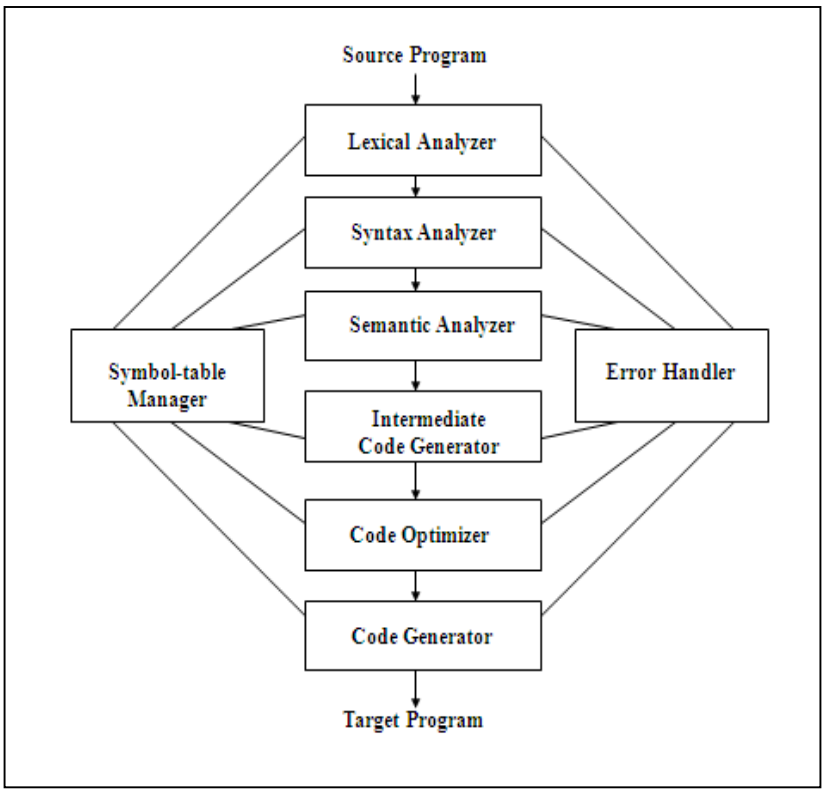


Fig. 1.5 Phases of Compiler

Conceptually, a compiler operates in *phases*, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in Fig 1.5 The first three phases, forms the bulk of the analysis portion of a compiler. Two other activities, Symbol table management and error handling, are shown interacting with the six phases.

Symbol table management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lex analyzer, the identifier is entered into the symbol table.

Error Detection and Reporting

Each phase can encounter errors. A compiler that stops when it finds the first error.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors when the token stream violates the syntax of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

The Analysis phases

As translation progresses, the compiler‟s internal representation of the source program changes. Consider the statement,

position := initial + rate * 10

The lexical analysis phase reads the characters in the source pgm and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword etc. The character sequence forming a token is called the *lexeme* for the token. Certain tokens will be augmented by a „lexical value‟. For example, for any identifier the lex analyzer generates not only the token id but also enter s the lexeme into the symbol table, if it is not already present there. The lexical value associated this occurrence of id points to the symbol table entry for this lexeme. The representation of the statement given above after the lexical analysis would be: id1: = id2 + id3 * 10

Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax trees (fig 3).

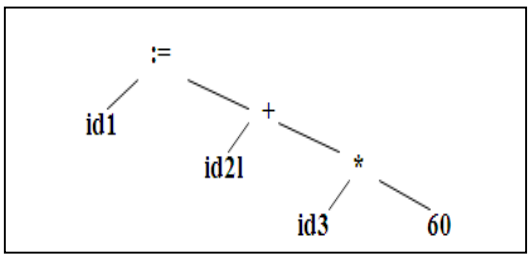


Fig. 1.6 Syntax tree

Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can have a variety of forms. In three-address code, the source pgm might look like this,

temp1: = inttoreal (10)
temp2: = id3 * temp1
temp3: = id2 + temp2
id1: = temp3

Code Optimization

The code optimization phase attempts to improve the intermediate code, so that faster running machine codes will result. Some optimizations are trivial. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called "optimising compilers", a significant fraction of the time of the compiler is spent on this phase.

Code Generation

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

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.

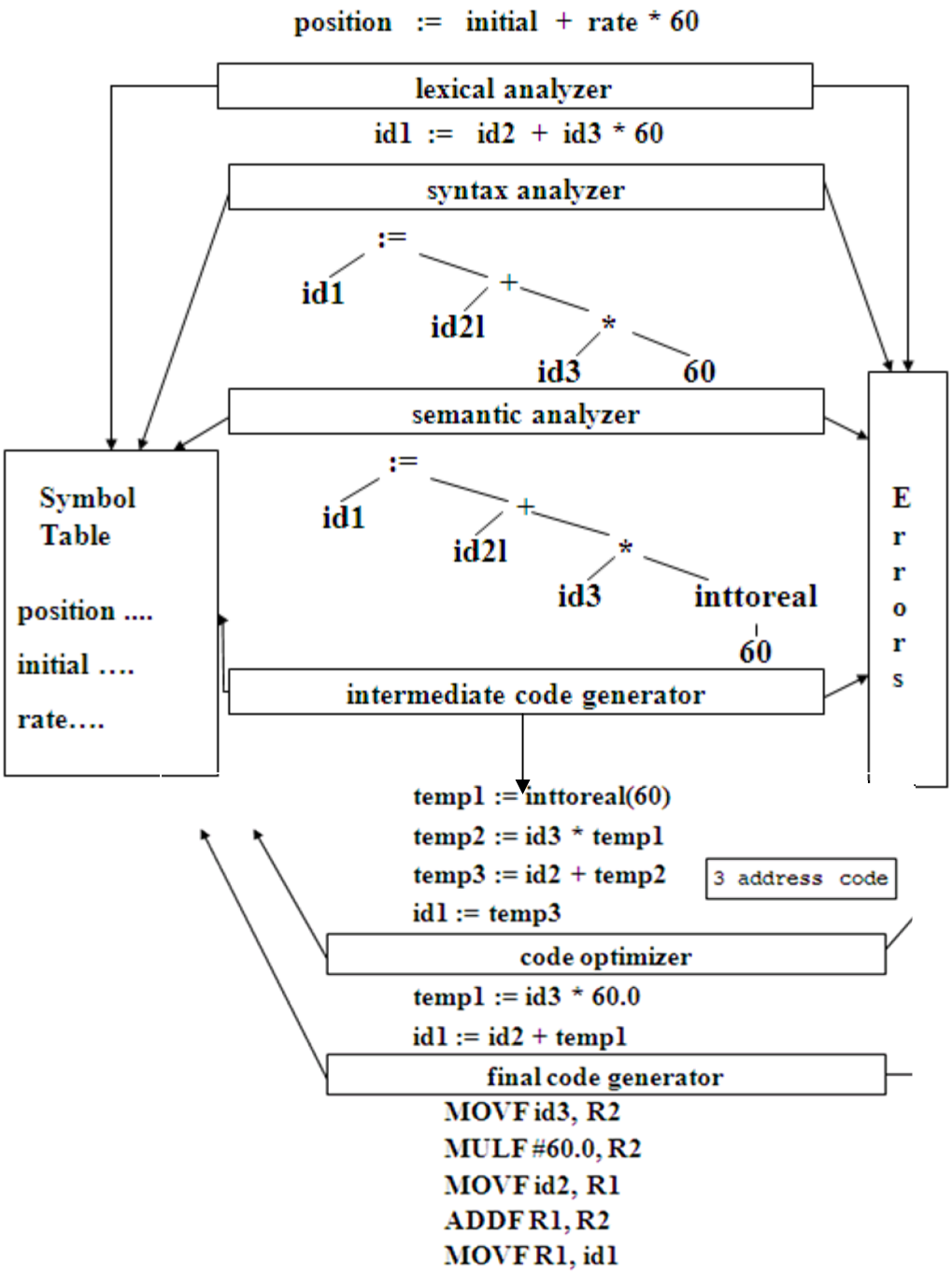


Fig. 1.7 Translation of a statement

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

A simple way to build lexical analyzer is to construct a diagram that illustrates the structure of the tokens of the source language, and then to hand-translate the diagram into a program for finding tokens. Efficient lexical analysers can be produced in this manner.

Role of Lexical Analyser

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis. As in the figure, upon receiving a "get next token" command from the parser the lexical analyzer reads input characters until it can identify the next token.

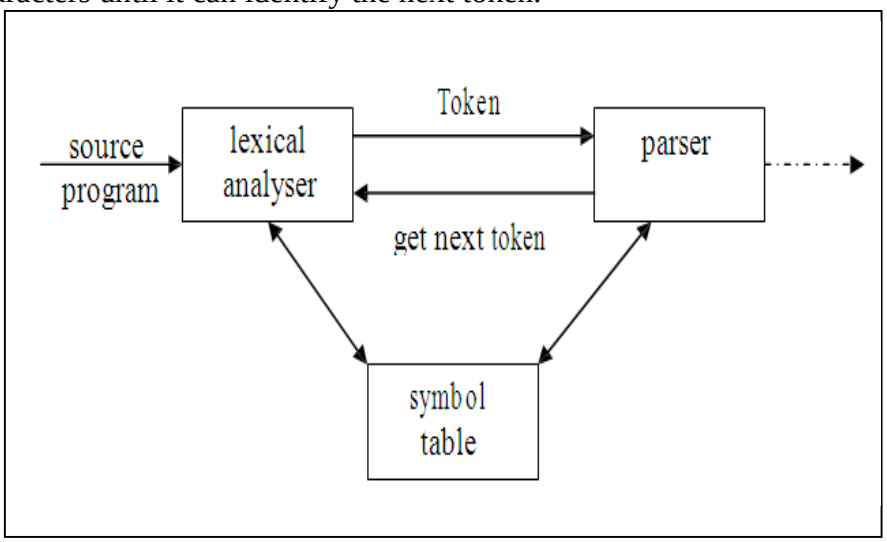


Fig. 1.8 Interaction of lexical analyzer with parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, tab, and new line character. Another is correlating error messages from the compiler with the source program.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing

1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
2) Compiler efficiency is improved.
3) Compiler portability is enhanced.

Tokens Patterns and Lexemes._

There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. The pattern is set to match each string in the set.

In most programming languages, the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.

Lexeme

Collection or group of characters forming tokens is called Lexeme. A lexeme is a sequence of characters in the source program that is matched by the pattern for the token. For example in the Pascal's statement const pi = 3.1416; the substring pi is a lexeme for the token identifier.

Patterns

A pattern is a rule describing a set of lexemes that can represent a particular token in source program. The pattern for the token const in the above table is just the single string const that spells out the keyword.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except"

Certain language conventions impact the difficulty of lexical analysis. Languages such as FORTRAN require a certain constructs in fixed positions on the input line. Thus the alignment of a lexeme may be important in determining the correctness of a source program.

Attributes of Token

The lexical analyzer returns to the parser a representation for the token it has found. The representation is an integer code if the token is a simple construct such as a left parenthesis, comma, or colon. The representation is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant.

The integer code gives the token type, the pointer points to the value of that token. Pairs are also retuned whenever we wish to distinguish between instances of a token.

The attributes influence the translation of tokens.

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

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

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Fig. 1.9 shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

The distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see:

DECLARE (ARG1, ARG2… ARG *n*)

Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, if the look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

BUFFER PAIRS

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

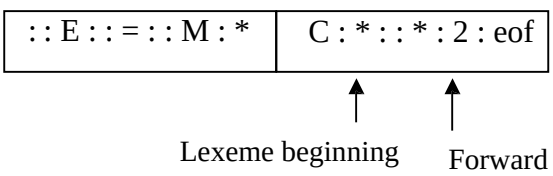


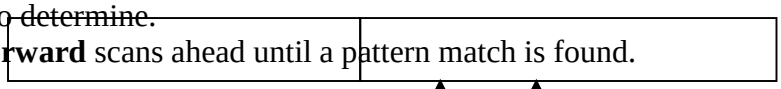
Fig. 1.9 An input buffer in two halves

- 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:

: : E : : = : : M : * : eof C : * : : * : 2 : eof : : : eof

lexeme_beginning

forward

Fig. 1.10 Sentinels at end of each buffer half

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 string s. 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. | 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

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
……
$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

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 short hands 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

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.

1.10 RECOGNITION OF TOKENS

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.

1.11 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

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

- Lex
- YACC

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 progra m a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

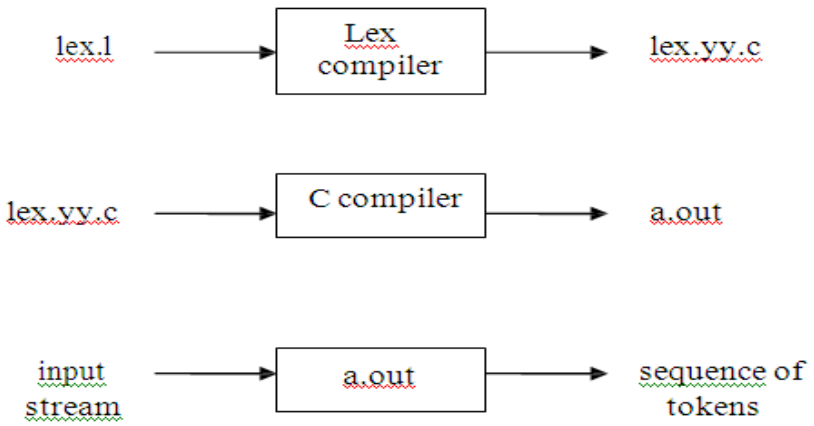


Fig1.11 Creating a lexical analyzer with lex

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.

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.
There are tow types of Finite Automata :

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

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

Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states q_0 – starting state

f_n – final state

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

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
SYNTAX ANALYSIS AND RUN-TIME ENVIRONMENTS

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.

2.2 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.

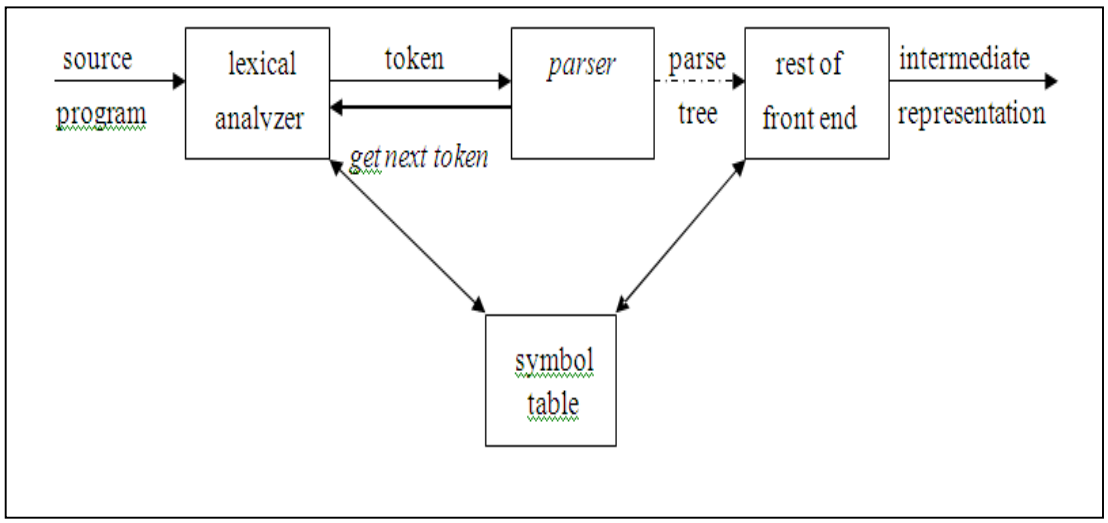


Fig. 2.1 Position of parser in compiler model

Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. 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 an identifier, keyword or operator.
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.

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.

2.3 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.

Derivations:
Two basic requirements for a grammar are :
1. To generate a valid string.
2. 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 :

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

To generate a valid string - (id+id) from the grammar the steps are
1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,
- E is the start symbol.

- -(id+id) is therequiredsentence(only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:
The two types of derivation are:
1. Left most derivation
2. 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)

Left Most Derivation	**Right Most 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 trees are,

Fig. 2.2 Two parse trees for id+id*id

2.4 WRITING A GRAMMAR

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

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,	It is useful in describing nested structures such as balanced

constants, keywords, and so forth.	parentheses, matching begin-end's and so on.

There are four categories in writing a grammar :
1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. 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.

Reasons for using the regular expression to define the lexical syntax of a language

- 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 (Fig. 2.3)

To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*
matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**
unmatched_stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

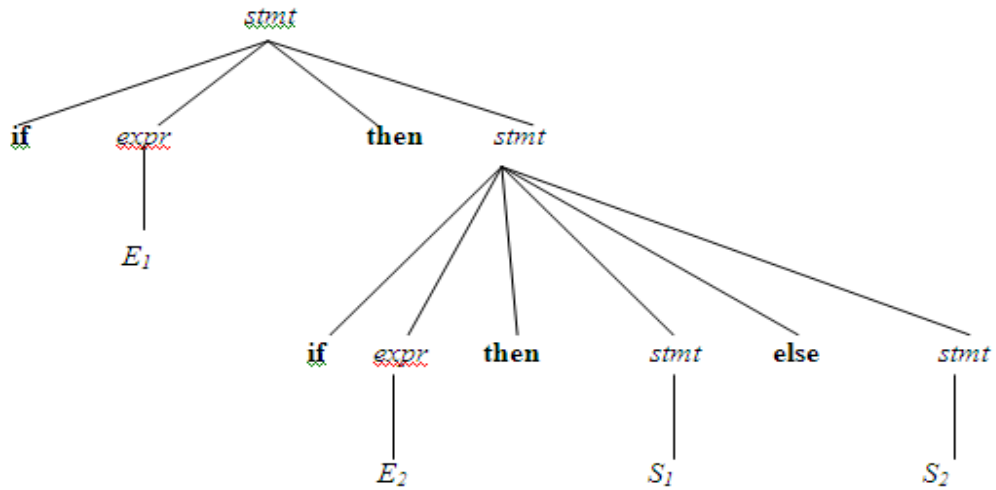
Eliminating Left Recursion:

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

If there is a production A → Aα | β it can be replaced with a sequence
A →βA'
A' → αA' | ε
without changing the set of strings derivable from A.
Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
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

1.



2.

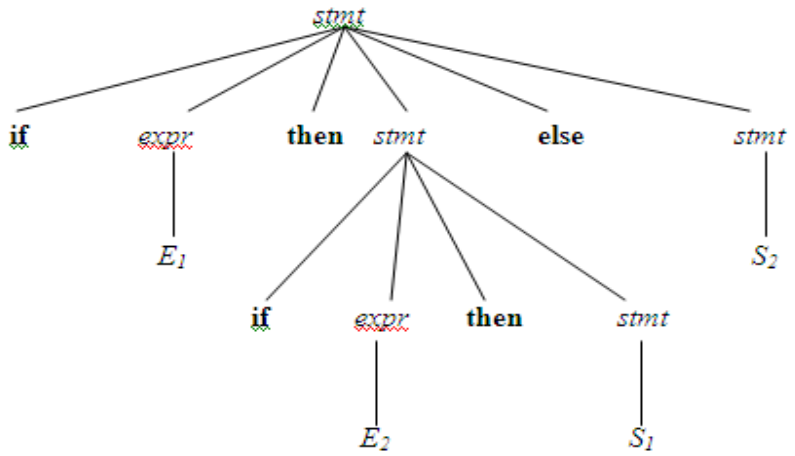


Fig. 2.3 Two parse trees for an ambiguous sentence

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 → iEtS | iEtSeS | a

E→b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS |ε

E → b

2.5 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.

2.5 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

RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead.

Therefore, a parser using the single-symbol look-ahead method and top-down parsing

without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated. This parsing method may involve backtracking.

Example for backtracking :

Consider the grammar G : S → cAd

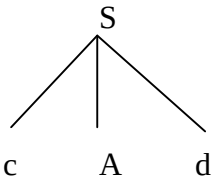
A → ab|a

and the input string w=cad.

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

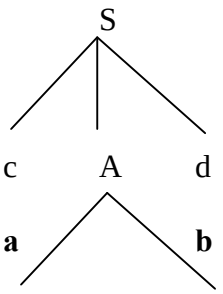
Step1:

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.

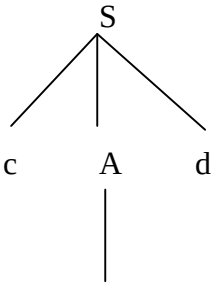


Step3:

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 **backtracking.**

Step4:

Now try the second alternative for A.



a

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

Predictive parsing

It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal . The nonrecursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

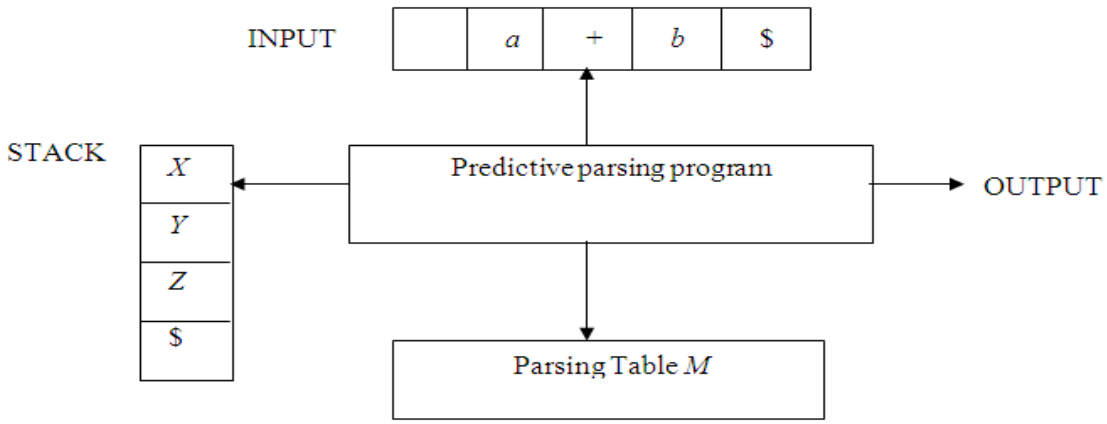


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.

2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.

3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X- >UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

Algorithm for Nonrecursive predictive arsing.
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 is in a configuration in which it 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 shown in Fig.

```
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 of $ then
     if X=a then


        pop X from the stack and advance ip
     else error()
   else
     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-> Y1Y2...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 → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1);that is, Y1,….Yi-1=> ε. If ε is in FIRST(Yj) 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 → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ 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 → α of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A, a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α 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) = {+, * , $,) }

Predictive parsing Table

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

Stack Implementation

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

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.
Consider this following grammar:
S → iEtS | iEtSeS| a
E → b

After eliminating left factoring, we 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}

Parsing table:

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

Since there are more than one production, the grammar is not LL(1) grammar.
Actions performed in predictive parsing:
1. Shift
2. Reduce
3. Accept
4. Error

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

2.6 BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

SHIFT-REDUCE PARSING

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

Example:
Consider the grammar:

S → aABe
A → Abc | b
B → d

The sentence to be recognized is abbcde.

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

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 id1+id2*id3

The rightmost derivation is :
E → E+E
→ E+E*E
→ E+E*id3
→ E+id2*id3
→ id1+id2*id3

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 = γn, where γn is the nth rightsentinel form of some rightmost derivation.

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-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.

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

1. Shift-reduce conflict:

Example:

Consider the grammar:

E → E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	
$ 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. Reduce-reduce conflict:
Consider the grammar:
M → R+R|R+c|R
R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$c	+c $	Reduce 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:

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

- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing. Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non-terminals.

Example:
Consider the grammar:
E → EAE|(E)|-E|id
A → +|-|*|/|↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows: E → E+E|E-E|E*E|E/E|E↑E |-E|id

Operator precedence relations:
There are three disjoint precedence relations namely

<. - less than
= - equal to
.> - greater than

The relations give the following meaning:

a<.b - a yields precedence to b
a=b - a has the same precedence as b
a.>b - a takes precedence over b

Rules for binary operations:
1. If operator θ1 has higher precedence than operator θ2, then make
θ1 . > θ2 and θ2 < . θ1
2. If operators θ1 and θ2, are of equal precedence, then make
θ1 . > θ2 and θ2 . > θ1 if operators are left associative
θ1 < . θ2 and θ2 < . θ1 if right associative
3. Make the following for all operators θ:
θ <. id ,id.>θ
θ <.(, (<.θ
).>θ, θ.>)
θ.>$, $<. θ

Also make
(=) , (<. (,) .>) , (<. id, id .>) , $ <. id , id .> $, $
Example:
Operator-precedence relations for the grammar
E → E+E|E-E|E*E|E/E|E↑E |(E)|-E|idis 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
(5) let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by ip;
(6) if a <. b or a = b then begin
(7) push b onto the stack;
(8) advance ip to the next input symbol;
end;
(9) else if a . > b then /*reduce*/
(10) repeat
(11) pop the stack
(12) until the top stack terminal is related by <.to the terminal most recently popped
(13) else error( )
end

Stack implementation of operator precedence parsing:

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

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w$

where w is the input string to be parsed.

Example:
Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is id+id*id .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:
1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar, the grammar can be ignored. The grammar is not referred anymore during implementation.

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

2.7 LR PARSERS

An efficient bottom-up syntax analysis technique that can be used 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:
1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.

4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

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

Types of LR parsing method:

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

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

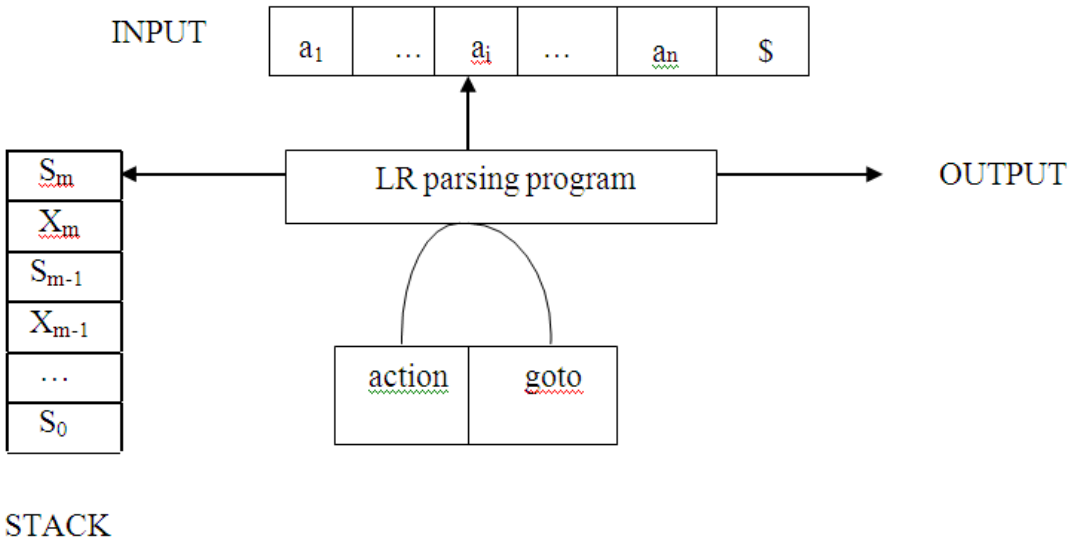


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa 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 s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
- The parsing table consists of two parts : action and goto functions.

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

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

4. error.

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

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G.
Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 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
```

2.8 CONSTRUCTING SLR(1) PARSING TABLE:

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

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

LR(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:

1. Initially, every item in I is added to closure(I).
2. 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.

Algorithm for construction of SLR parsing table:
Input : An augmented grammar G'
Output : The SLR parsing table functions action and goto for G'
Method :
1. Construct C = {I0, I1, …. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii.. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in Ii , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).
   (c) If [S'→S.] is in Ii, 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-term
   If goto(Ii,A) = Ij, then goto[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the [S'→.S].

2.9 TYPE CHECKING

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

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

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

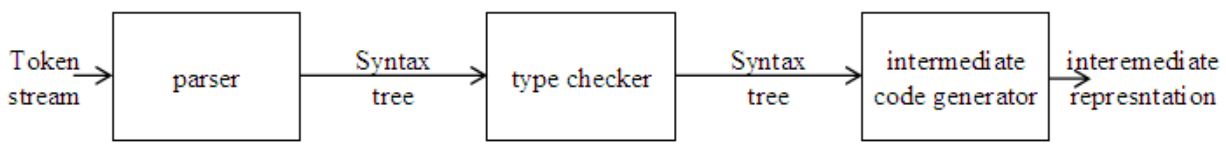


Fig. 2.6 Position of type checker

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

Type Systems

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

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

Type Expressions

The type of a language construct will be denoted by a "type expression." A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.
   A special basic type, type_error , will signal an error during type checking; void denoting "the absence of a value" allows statements to be checked.
2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

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

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

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

   For example:

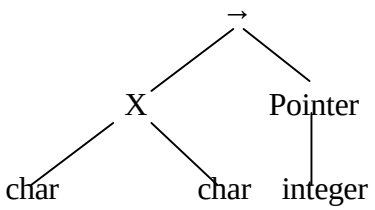
```
type row = record
        address: integer;
        lexeme: array[1..15] of char
        end;
var table: array[1...101] of row;
```

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

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

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

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



Fg. 5.7 Tree representation for char x char → pointer (integer)

Type systems

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

Static and Dynamic Checking of Types

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

Sound type system

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

Strongly typed language

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

Error Recovery

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

2.10 SPECIFICATION OF A SIMPLE TYPE CHECKER

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

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:
P → D ; E
D → D ; D
D → id : T { addtype (id.entry , T.type) }
T → char { T.type : = char }
T → integer { T.type : = integer }
T → ↑ T1 { T.type : = pointer(T1.type) }
T → array [num] of T1 { T.type : = array (1… num.val , T1.type) }

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

Type checking of expressions

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

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

2. E → id { E.type : = lookup (id.entry) }
lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 mod E2 { E.type : = if E1. type = integer and
E2. type = integer then integer
else type_error }
The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type_error.

4. E → E1 [E2] { E.type : = if E2.type = integer and
E1.type = array(s,t) then t
else type_error }
In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

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

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

Type checking of statements

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

Translation scheme for checking the type of statements:

1. Assignment statement:

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

2. Conditional statement:

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

3. While statement:

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

4. Sequence of statements:

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

Type checking of functions

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

E → E1 (E2) { E.type : = if E2.type = s and
E1.type = s → t then t
else type_error }

2.11 RUN-TIME ENVIRONMENTS - 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 n 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.

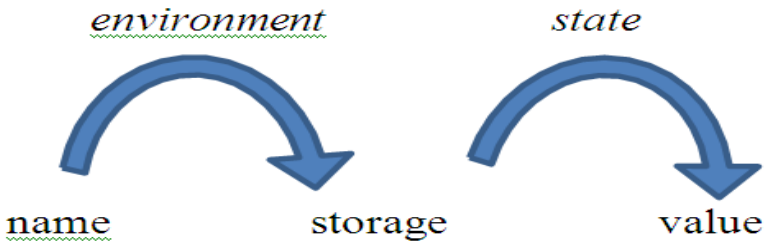


Fig. 2.8 Two-stage mapping from names to values

2.12 STORAGE ORGANIZATION

The executing target program runs in its own logical address space in which each program value has a location. The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

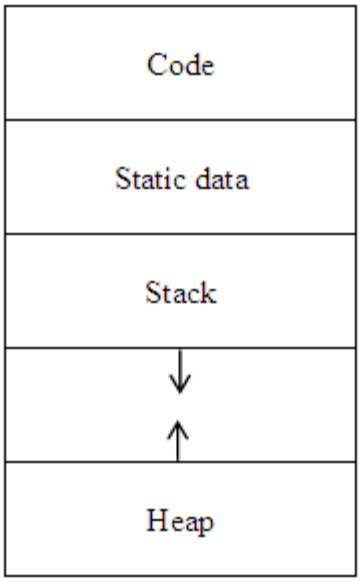


Fig. 2.9 Typical subdivision of run-time memory into code and data areas

- Run-time storage comes in blocks, where a byte is the smallest unit of memory. Four bytes form a machine word. Multibyte objects are 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.

- 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, n return a value, and if one does, we may prefer to place that value i 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.

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.

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 i 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.

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.

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.

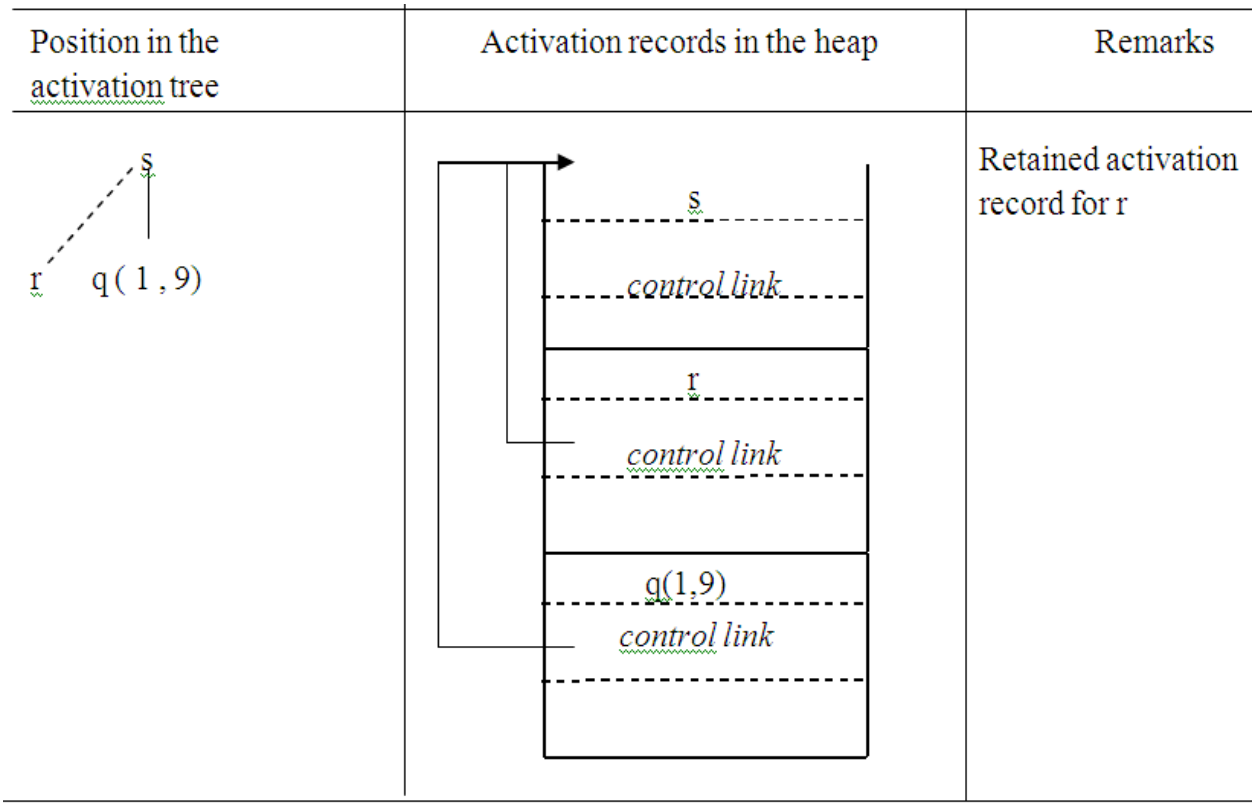


Fig. 2.10 Records for live activations need not be adjacent in heap

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

UNIT III
INTERMEDIATE CODE GENERATION

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:

1. 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.

2. A machine-independent code optimizer can be applied to the intermediate representation.

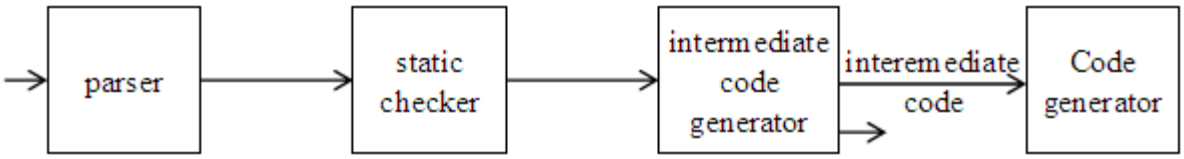


Fig. 3.1 Intermediate code generator

3.1 INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

* Syntax tree
* Postfix notation
* Three address code

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

Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement a : = b * - c + b * - c are shown in Fig.3.2:

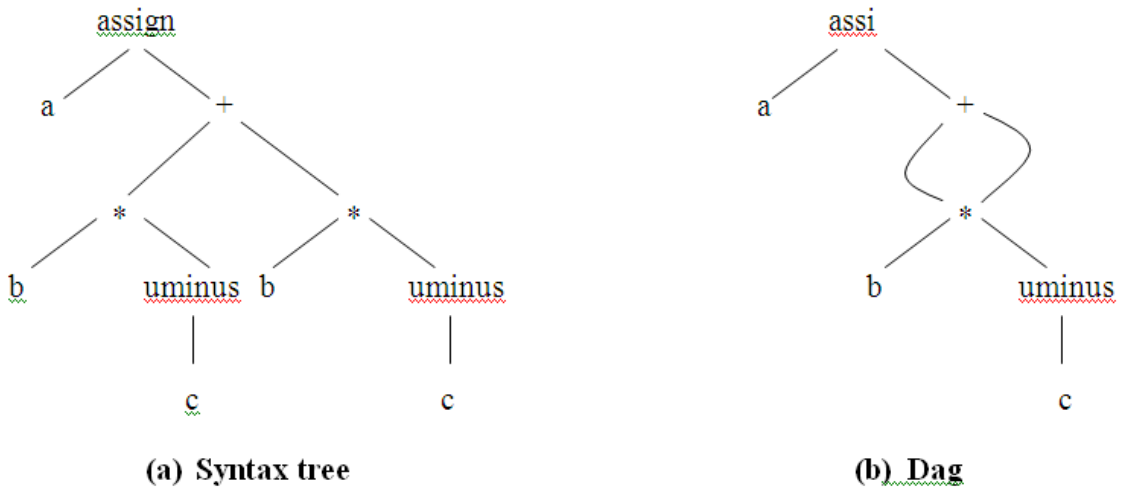
Postfix notation:

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

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are

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



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

Fig. 3.2 Graphical representation of a:=b*- c+b*- c

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

Fig. 3.3 Syntax-directed definition to produce syntax trees for assignment statements

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

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

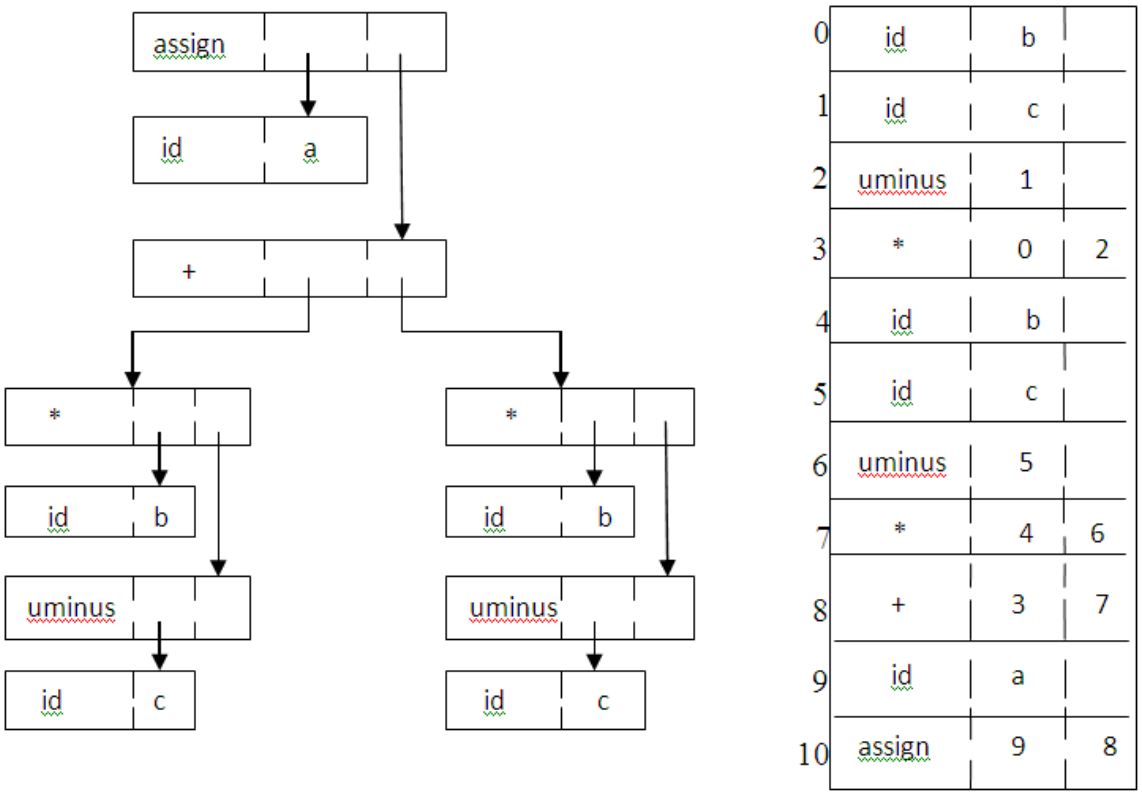


Fig. 3.4 Two representations of the syntax tree

Three-address code

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

x : = y op z

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

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

where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

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

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

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

(a) Code for the syntax tree

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

(b) Code for the dag

Fig.3.5 Three-address code corresponding to the syntax tree and dag

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

Types of Three-Address Statements:

The common three-address statements are:

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

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

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

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

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

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

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

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

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

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

Syntax-Directed Translation into Three-Address Code:

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

Given input a : = b * - c + b * - c, the three-address code is as shown in Fig. 8.3a. The synthesized attribute S.code represents the three-address code for the assignment S.

The nonterminal E has two attributes :
1. E.place, the name that will hold the value of E , and
2. E.code, the sequence of three-address statements evaluating E.

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

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

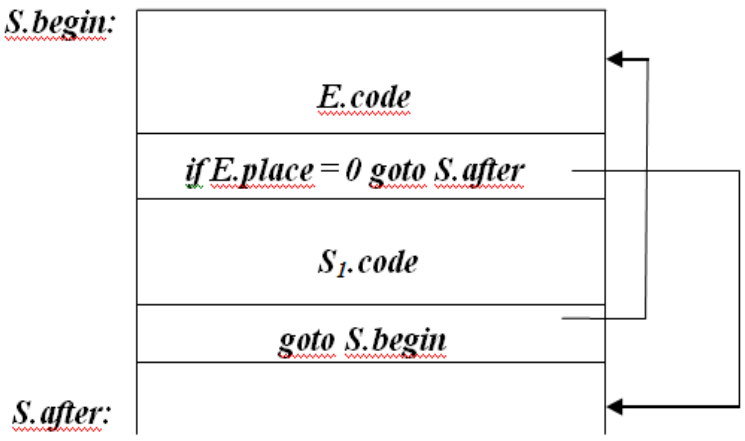


Fig.3.6 Semantic rules generating code for a while statement

PRODUCTION	SEMANTIC RULES
S→whileEdoS1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ':') || E.code || gen ('if' E.place '=' '0' 'goto' S.after)|| S1.code || gen ('goto' S.begin) || gen (S.after ':')

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

- 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 S1 is generated using new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code for S, respectively.
- The function newlabel returns a new label every time it is called.
  We assume that a non-zero expression represents true; that is when the value of E becomes zero, control leaves the while statement.

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

	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

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

Fig.3.7 (a) Quadruples (b) 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.

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

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

Fig. 3.8 (a) x[i] : = y

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

(b) x : = y[i]

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)

Fig. 3.9 Indirect triples representation of three-address statements

3.2 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.

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 T1       { T.type : = array(num.val, T1.type);
                              T.width : = num.val X T1.width }
T→↑T1                       { T.type : = pointer ( T1.type);
                              T.width : = 4 }
```

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→D

D→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→ proc id D1;S is seen, and entries for the declarations in D1 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.

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→D1 ; D2	
D→proc id ; N D1; 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{actionA}

are done before actionA 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 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; ND1; S occurs, the width of all declarations generated by D1 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.

3.3 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 tran slation 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 → E1 + E2 has the general form:

evaluate E1 into t1
evaluate E2 into t2
t : = t1 + t2

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

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

Fig. 3.10 Three-address code with stack temporaries

Addressing Array Elements:

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

base + (i - low) x w

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

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

ixw+ (base-low x w)

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

Address calculation of multi-dimensional arrays:

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

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

In the case of row-major form, the relative address of A[i1 , i2] can be calculated by the formula

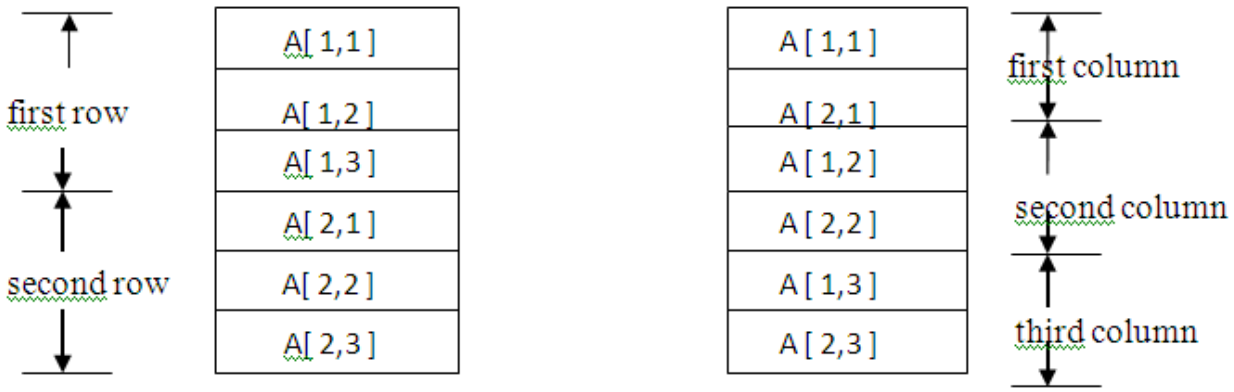
base + ((i1 - low1) x n2 + i2 - low2) x w

where, low1 and low2 are the lower bounds on the values of i1 and i2 and n2 is the number of values that i2 can take. That is, if high2 is the upper bound on the value of i2, then n2 = high2 -

low2 + 1.

Assuming that i1 and i2 are the only values that are known at compile time, we can rewrite the above expression as

((i1 x n2) + i2) x w + (base - ((low1 x n2) + low2) x w)



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

Fig. 3.11 Layouts for a 2 x 3 array

Generalized formula:

The expression generalizes to the following expression for the relative address of A[i1,i2,…,ik]

((. . . ((i1n2 + i2) n3 + i3) . . .) nk + ik) x w + base - ((. . .((low1n2 + low2)n3 + low3) . . .) nk + lowk) x w

for all j, nj = highj - lowj + 1

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) S→ L : = E
(2) Ẹ→ E + E
(3) Ẹ→ (E)
(4) Ẹ→ L
(5) L→ Elist]
(6) Ḷ→ id
(7) Elịst→ Elist , E
(8) Elịst→ 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) Ṣ→ 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) Ẹ→ E1 + E2      { E.place : = newtemp;
                      emit ( E.place ‘: =’ E1.place ‘ +’ E2.place ) }
(3) Ẹ→ ( E1 )       { E.place : = E1.place }
```

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

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

(5) L→ Elist ]    { L.place : = newtemp;
                  L.offset : = newtemp;
                  emit (L.place ‘: =’ c( Elist.array ));
                  emit (L.offset ‘: =’ Elist.place ‘*’ width (Elist.array)) }
(6) L→ id         { L.place := id.place;
                  L.offset := null }
(7) Elist→ Elist1 , E  { t := newtemp;
                  m : = Elist1.ndim + 1;
            emit ( t ‘: =’ Elist1.place ‘*’ limit (Elist1.array,m)); emit ( t ‘: =’ t ‘+’ E.place);
                  Elist.array : = Elist1.array;
                  Elist.place : = t;
                  Elist.ndim : = m }
(8) Elist→ id [ E  { Elist.array : = id.place;
                  Elist.place : = E.place; Elist.ndim : = 1 }
```

Type conversion within Assignments :

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

E→ E + E is :

```
E→E + E     { E.type : =
            if E1.type = integer and
            E2.type = integer then integer
            else real       }
```

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

Semantic action for E E1 + E2

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
        emit( E.place ‘: =’ E1.place ‘int +’ E2.place);
        E.type : = integer
end
else if E1.type = real and E2.type = real then begin
```

```
        emit( E.place ‘: =’ E1.place ‘real +’ E2.place);
        E.type : = real
end
else if E1.type = integer and E2.type = real then begin
        u : = newtemp;
emit( u ‘: =’ ‘inttoreal’ E1.place);
emit( E.place ‘: =’ u ‘ real +’ E2.place); E.type : = real
end
else if E1.type = real and E2.type =integer then begin
        u : = newtemp;
emit( u ‘: =’ ‘inttoreal’ E2.place);
emit( E.place ‘: =’ E1.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

```
        t1 : =  i  int* j
        t3 : =  inttoreal  t1
        t2 : = y real+ t3
        x : = t2
```

3.4 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 E1 relop E2, where E1 and E2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E→EorE | EandE |notE | (E) |id relop id | true | false

Methods of Translating Boolean Expressions:

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

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

Numerical Representation

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

For example :
* The translation for a or b and not c is the three-address sequence
t1 : = not c
t2 : = b and t1
t3 : = a or t2
* 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 (aga 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→E1 orE2 { E.place : = newtemp;
emit(E.place ‘: =’ E1.place ‘or’ E2.place) }
E→E1 andE2 { E.place : = newtemp;
emit(E.place ‘: =’ E1.place ‘and’ E2.place) }
E→notE1 { E.place : = newtemp;
emit(E.place ‘: =’ ‘not’ E1.place) }
E→ (E1) { E.place : = E1.place }

E→id1 relop id2 { E.place : = newtemp;
emit(‘if’ id1.place relop.op id2.place ‘goto’ nextstat + 3); emit(E.place ‘: =’ ‘0’);
emit(‘goto’ nextstat +2);
emit(E.place ‘: =’ ‘1’) }
E→true { E.place : = newtemp;
emit(E.place ‘: =’ ‘1’) }
E→false { E.place : = newtemp;
emit(E.place ‘: =’ ‘0’) }

Short-Circuit Code:

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

Translation of a < b or c < d and e < f
100 : if a < b goto 103 101 : t1 : = 0
102 : goto 104 103 : t1 : = 1
104 : if c < d goto 107 105 : t2 : = 0
106 : goto 108
107 : t2 : = 1

108 : if e < f goto 111 109 : t3 : = 0
110 : goto 112
111 : t3 : = 1
112 : t4 : = t2 and t3
113 : t5 : = t1 or t4

Flow-of-Control Statements

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

S→ifEthenS1
| if E then S1 else S2
| while E do S1

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

* 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.

3.5 CASE STATEMENTS

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

Switch-statement syntax

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

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

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

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

* By a sequence of conditional goto statements, if the number of cases is small.
* By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
* If the number of cases s large, it is efficient to construct a hash table.
* There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say imin to imax, and the number of different values is a reasonable fraction of imax - imin, 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 - imin 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 va transfer to the table entry at offset j-imin .

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```
switch E
begin
case V1 :   S1
case V2 :   S2

case Vn-1 :  Sn-1
default :        Sn
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
L1 :     code for S1
goto next
L2 :     code for S2
goto next

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

if  t = Vn-1 goto Ln-1
goto Ln
next :
```

To translate into above form :

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

* As expression E is parsed, the code to evaluate E into t is generated. After processing E , the jump goto test is generated.
* As each case keyword occurs, a new label Li is created and entered into the symbol table. A pointer to this symbol-table entry and the value Vi of case constant are placed on a stack (used only to store cases).
* Each statement case Vi : Si is processed by emitting the newly c by the code for Si , 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 Ln is the label for the default statement.

3.6 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, 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 properlabel can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

1. makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
2. merge(p1,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.
3. backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions:

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

(1) E → E1 or M E2
(2) | E1 and M E2
(3) | not E1
(4) | (E1)
(5) | id1 relop id2
(6) | true
(7) | false
(8) M → ε

Synthesized attributes truelist and falselist of nonterminal E are used to g for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by E.truelist and E.falselist. Consider production E E1 and M E2. If E1 is false, then E is also false, so the statements on E1.falselist become part of E.falselist. If E1 is true, then we must next test E2, so the target for the statements E1.truelist must be the beginning of the code generated for E2. This target is obtained using marker nonterminal M.

Attribute M.quad records the number of the first statement of E2.code. With the production M→ε we associate the semantic action { M.quad : = nextquad }. The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E1.truelist when we have seen the remainder of the production E E1 and M E2. The translation scheme is as follows:

(1) E→E1 orM E2 { backpatch (E1.falselist, M.quad);
E.truelist : = merge(E1.truelist, E2.truelist);
E.falselist : = E2.falselist }
(2) E→E1 andM E2
{ backpatch (E1.truelist, M.quad);
E.truelist : = E2.truelist;
E.falselist : = merge(E1.falselist, E2.falselist) }
(3) E→notE1 { E.truelist : = E1.falselist;
E.falselist : = E1.truelist; }
(4) E→ (E1) { E.truelist : = E1.truelist;
E.falselist : = E1.falselist; }
(5) E→id1 relop id2
{ E.truelist : = makelist (nextquad);
E.falselist : = makelist(nextquad + 1);
emit(‘if’ id1.place relop.op id2.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 follow

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

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in

execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

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

The semantic rules for the revised grammar are as follows:

(1) S→ifEthenM1S1NelseM2 S2
{ backpatch (E.truelist, M1.quad);
backpatch (E.falselist, M2.quad);
S.nextlist : = merge (S1.nextlist, merge (N.nextlist, S2.nextlist)) }

We backpatch the jumps when E is true to the quadruple M1.quad, which is the beginning of the code for S1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S2. The list S.nextlist includes all jumps out of S1 and S2, as well as the jump generated by N.

(2) N→ε { N.nextlist : = makelist(nextquad);
emit('goto _') }
(3) M→ε { M.quad : = nextquad }
(4) S→ifEthenMS1 { backpatch(E.truelist, M.quad);
S.nextlist : = merge(E.falselist, S1.nextlist) }

(5) S→whileM1 EdoM2 S1
{ backpatch(S1.nextlist, M1.quad);
backpatch(E.truelist, M2.quad);
S.nextlist : = E.falselist
emit('goto' M1.quad) }
(6) S→beginLend { S.nextlist : = L.nextlist }
(7) S→A { S.nextlist : = nil }

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

(8) L→L1;M S { backpatch(L1.nextlist, M.quad);
L.nextlist : = S.nextlist }

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

(9) L→S { L.nextlist : = S.nextlist }

3.7 PROCEDURE CALLS

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

Let us consider a grammar for a simple procedure call statement

(1) S→call id(Elist)
(2) Elist→ Elist , E
(3) Elist→ E

Calling Sequences:

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

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

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

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.

(2) Elist→ Elist , E
{ append E.place to the end of queue }
(3) Elist→ E
{ initialize queue to contain only E.place }

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

UNIT 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.

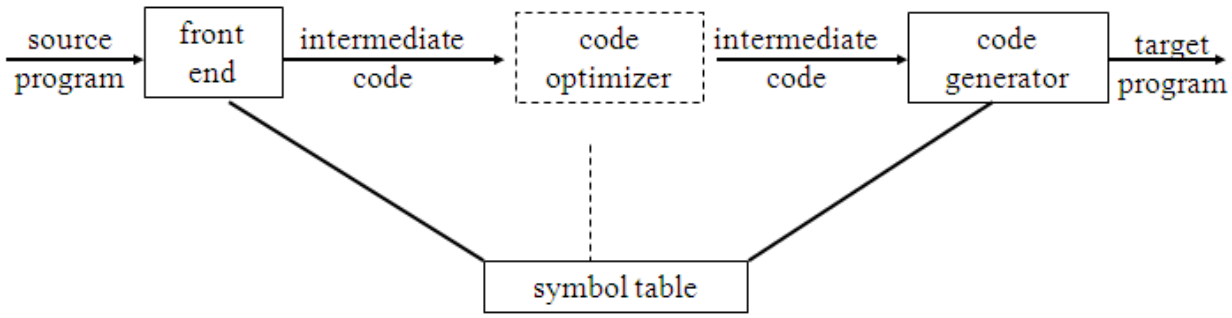


Fig. 4.1 Position of code generator

4.1 ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. 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 :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. 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.

2. Target program:

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

a. Absolute machine language

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

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. 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:gotoigenerates 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:

```
a:=b+c
d:=a+e          (a)

MOV b,R0
ADD c,R0
MOV R0,a        (b)
MOV a,R0
ADD e,R0
MOV R0,d
```

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 :

1. Register allocation - the set of variables that will reside in registers at a point in the program is selected.

2. Register assignment - the specific register that a value 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* are data 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.

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

For example : MOV R_0, M stores contents of Register R_0 into memory location 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, R0
ADD c, R0 cost = 6
MOV R0, a

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

iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :
MOV *R1, *R0
ADD *R2, *R0 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:

1. Static allocation 2. Stack allocation

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

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

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

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

1. Code
2. Static data
3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:
MOV #here + 20, callee.static_area /*It saves return address*/
GOTO callee.code_area /*It transfers control to the target code for the called procedure */
where,

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

Implementation of return statement:

A return from procedure callee is implemented by :

GOTO *callee.static_area

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

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

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

Stack allocation

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

The codes needed to implement stack allocation are as follows:

Initialization of stack:

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

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

Basic Blocks

• Abasic blockis 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:

t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5

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:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statementis a leader.
2. 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.

Consider the following source code for dot product of two vectors:

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

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

```
(1)     prod := 0
(2)     i := 1
(3)     t1 := 4* i
(4)     t2 := a[t1]      /*compute a[i] */
(5)     t3 := 4* i
(6)     t4 :=  b[t3]     /*compute b[i] */
(7)     t5 := t2*t4
(8)     t6 := prod+t5
(9)     prod := t6
(10)    t7 := i+1
(11)    i := t7
(12)    if i<=20 goto (3)
```

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:
A number of transformations can be applied to a basic block without expressions computed by the block. Two important classes of transformation are :
- Structure-preserving transformations
- 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 t1 and neither b nor c is t2.

2. 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.

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:

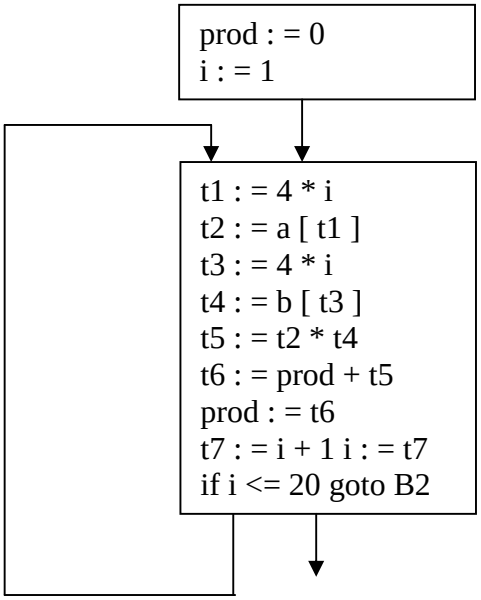


Fig. 4.2 Flow graph for program

- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.

Loops

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

1. All nodes in the collection are strongly connected.
2. 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.

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

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:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. 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.

3. 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.

4. 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.
Code sequence for the example is:

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

d : = v + u	ADD R1, R0
	
	MOV R0, d
		R0 contains d	d in R0
			
			d in R0 and memory

Generating Code for Indexed Assignments
The table shows the code sequences generated for the indexed assignmen a:= b[i] and a[i]:= b

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

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

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p : = a	MOV a, *Rp	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z	MOV y, R0

```
if x < 0 goto z        ADD z, R0
                       MOV R0,x
                       CJ<   z
```

4.7 REGISTER ALLOCATION AND ASSIGNMENT

Local register allocation

- Register allocation is only within a basic block. It follows top-down approach.
- Assign registers to the most heavily used variables
  - Traverse the block
  - Count uses
  - Use count as a priority function
  - Assign registers to higher priority variables first
- Advantage
  - Heavily used values reside in registers
- Disadvantage
  Does not consider non-uniform distribution of uses

Need of global register allocation

Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.

To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Register allocation by graph coloring

Global register allocation can be seen as a graph coloring problem.
Basic idea:

1. Identify the live range of each variable
2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
3. Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors.

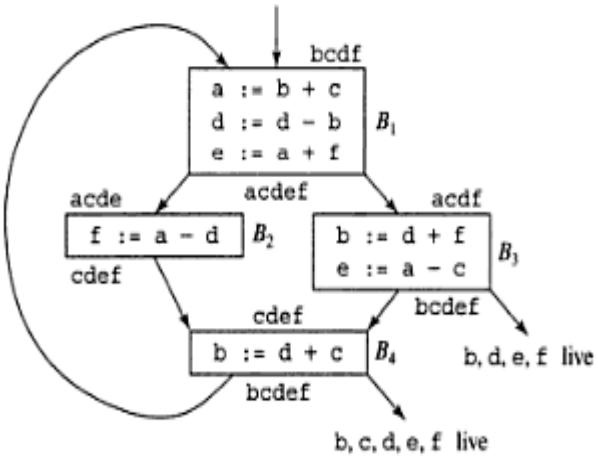


Fig 4.3 Flow graph of an inner loop

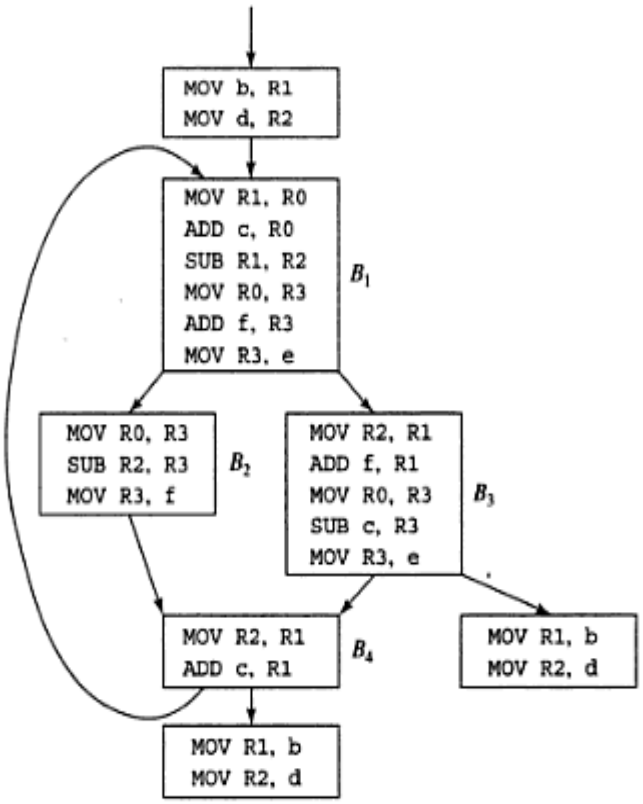


Fig 4.4 Code sequence using global register assignment

4.8 THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
    1. Leaves are labeled by unique identifiers, either variable names or constants.
    2. Interior nodes are labeled by an operator symbol.
    3. 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). (Checking for 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).

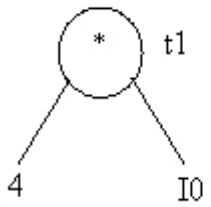
Step 3:

Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements in Fig 4.6

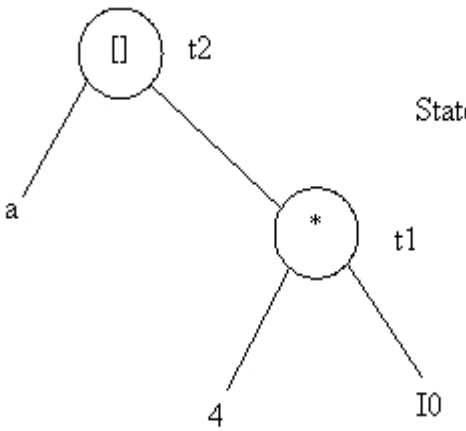
Stages in DAG Construction

(a)



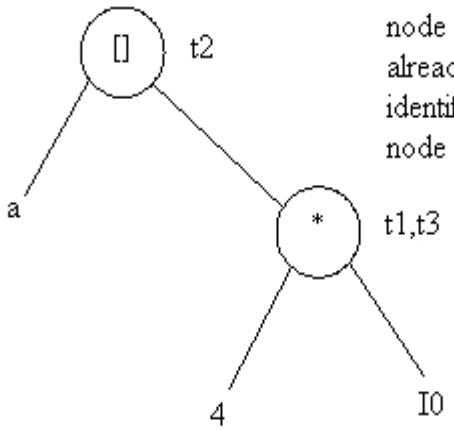
Statement (1)

(b)

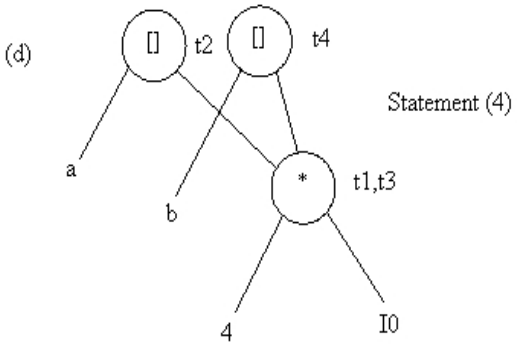


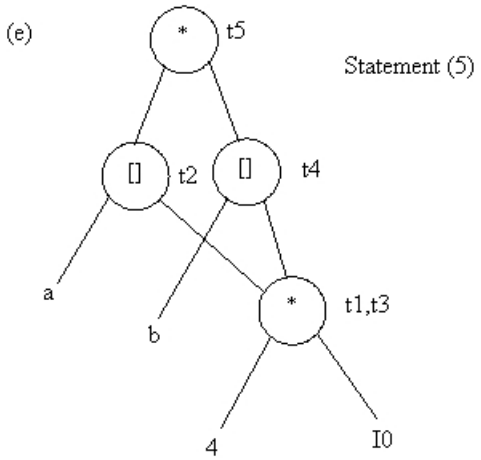
Statement (2)

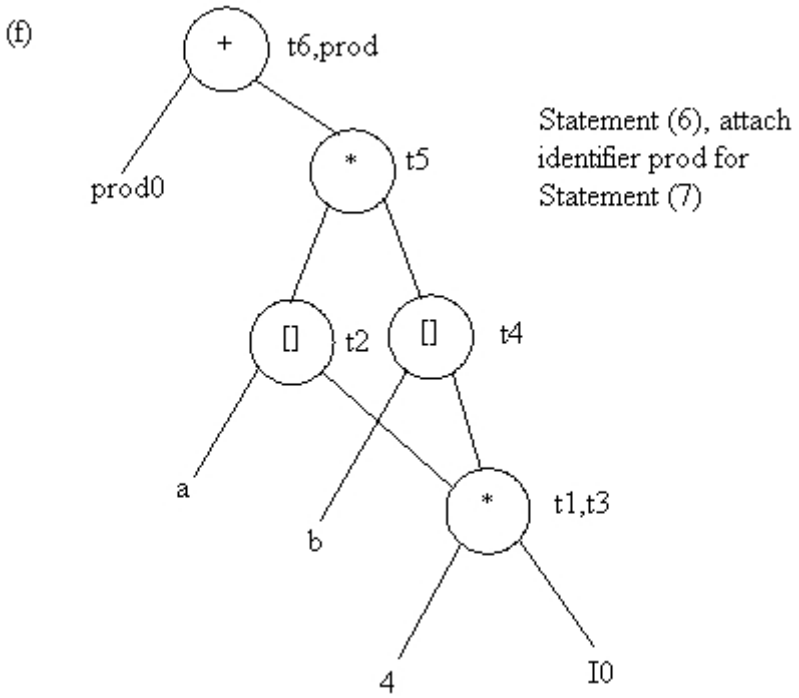
(c)



node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)


(d)
[]
t2
[]
t4
Statement (4)
a
*
t1,t3
b
4
I0


(e)
*
t5
Statement (5)
[]
t2
[]
t4
a
*
t1,t3
b
4
I0


(f)
+
t6,prod
Statement (6), attach
identifier prod for
Statement (7)
prod0
*
t5
[]
t2
[]
t4
a
*
t1,t3
b
4
I0

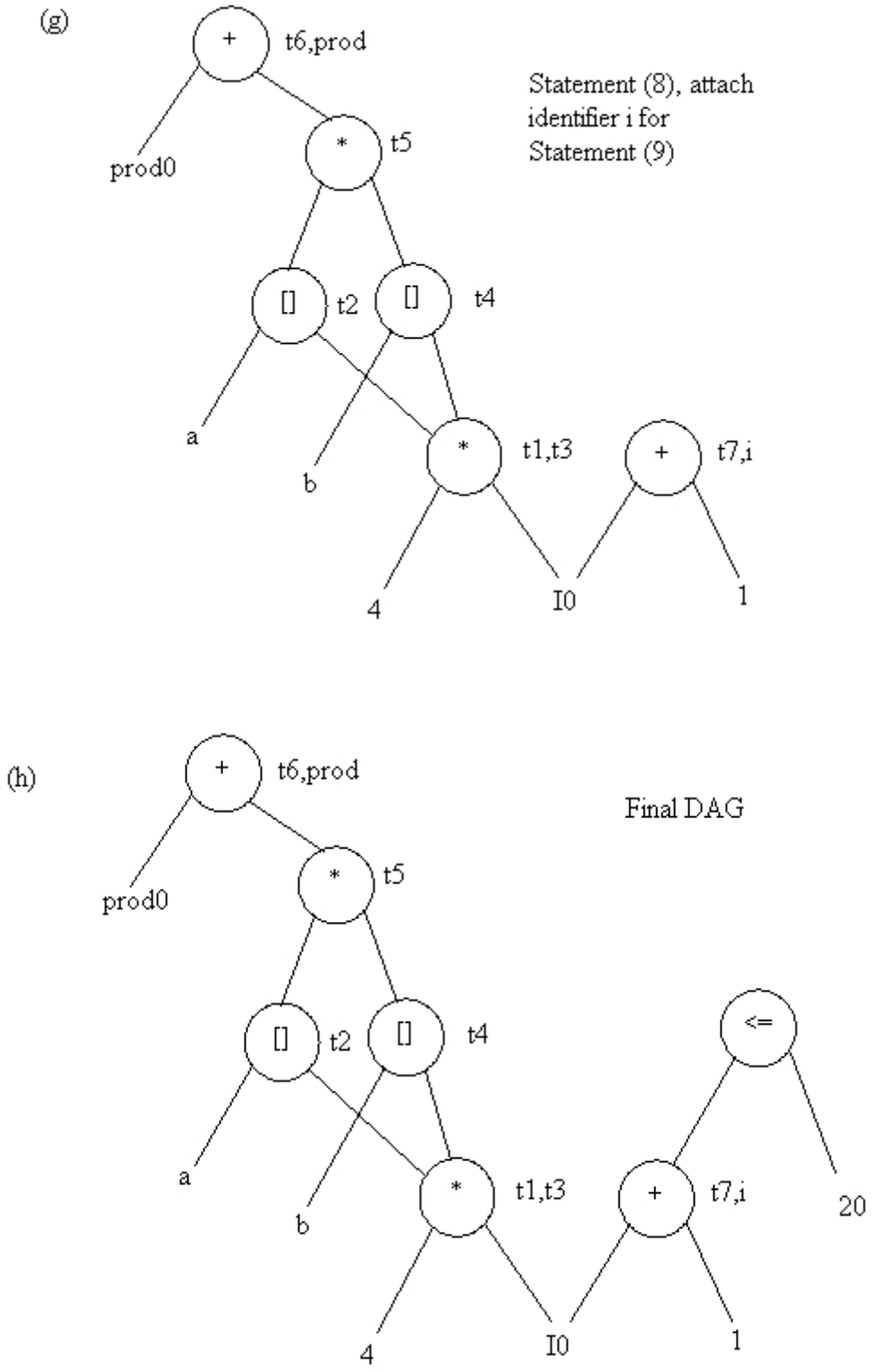


Fig. 4.5 Steps in DAG construction process

```
1.  t1 := 4* i
2.  t2 := a[t1]
3.  t3 := 4* i
4.  t4 :=  b[t3]
5.  t5 := t2*t4
6.  t6 := prod+t5
7.  prod := t6
8.  t7 := i+1
9.  i := t7
10. if i<=20 goto (1)
```

Fig. 4.6 Code Block

Application of DAGs:

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

4.9 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 start 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:

```
t1 : = a + b
t2 : = c + d
t3 : = e - t2
t4 : = t1 - t3
```

Generated code sequence for basic block:

```
MOV a , R0
ADD b , R0
MOV c , R1
```

```
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

```
t2 : = c + d
t3 : = e - t2
t1 : = a + b
t4 : = t1 - t3
```

Revised code sequence:

```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

In this order, two instructions **MOV R0 , t1 and MOV t1 , R1** have been saved.

A Heuristic ordering for Dags

The heuristic ordering algorithm attempts to make the evaluation of a nod the evaluation of its leftmost argument. The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1)  while unlisted interior nodes remain do begin
2)      select an unlisted node n, all of whose parents have been listed;
3)      list n;
4)      while the leftmost child m of n has no unlisted parents and is not a leaf do
begin
5)      list m;
6)      n : = m
end
end
```

Example: Consider the DAG shown below

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 a new 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:

t8 : = d + e
t6 : = a + b
t5 : = t6 - c
t4 : = t5 * t8
t3 : = t4 - e
t2 : = t6 + t4
t1 : = t2 * t3

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

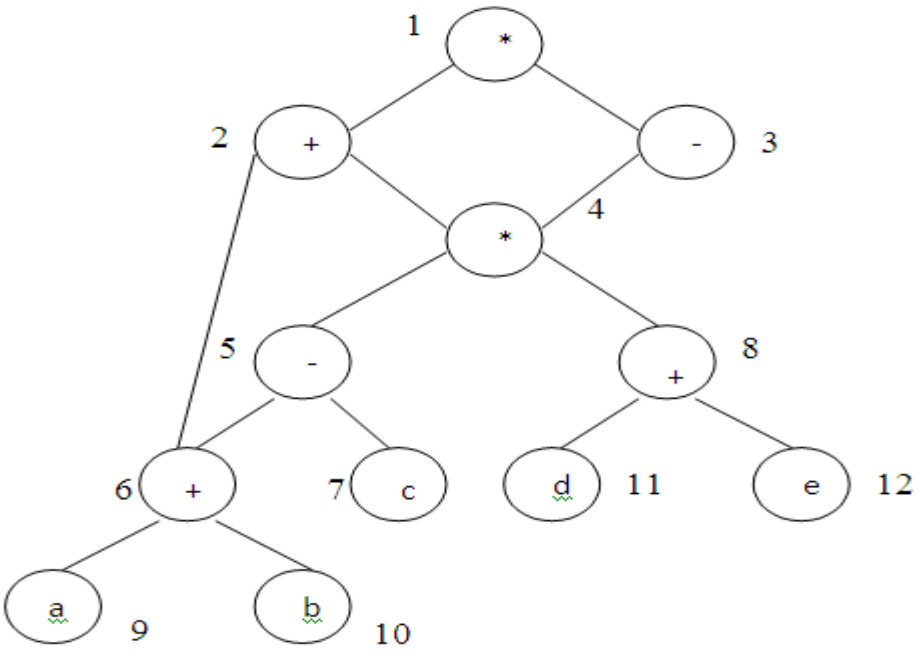


Fig. 4.7 A DAG

UNIT 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 transformations provided by an optimizing compiler should have several properties. They are:

1. 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.

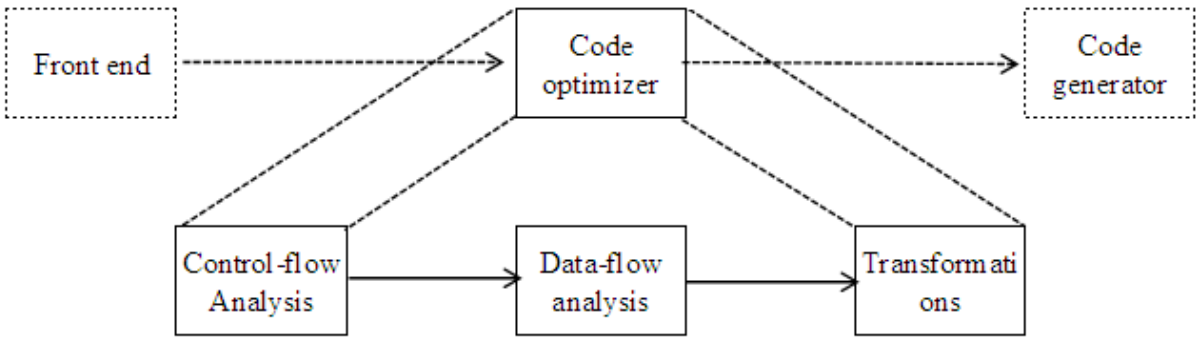


Fig. 5.1 Organization of the code optimizer

2. A transformation must, on the average, speedup 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.

3. 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.

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 control flow graph, Call graph. Data flow analysis (DFA) is the process of asserting 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 performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

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

Function preserving transformations examples:

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

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

***

Common Sub expressions elimination:

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

• For example

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

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

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

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

Copy Propagation:

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

• For example:
x=Pi;
……
A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

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

Example:

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

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

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.
For example,

a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- 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.

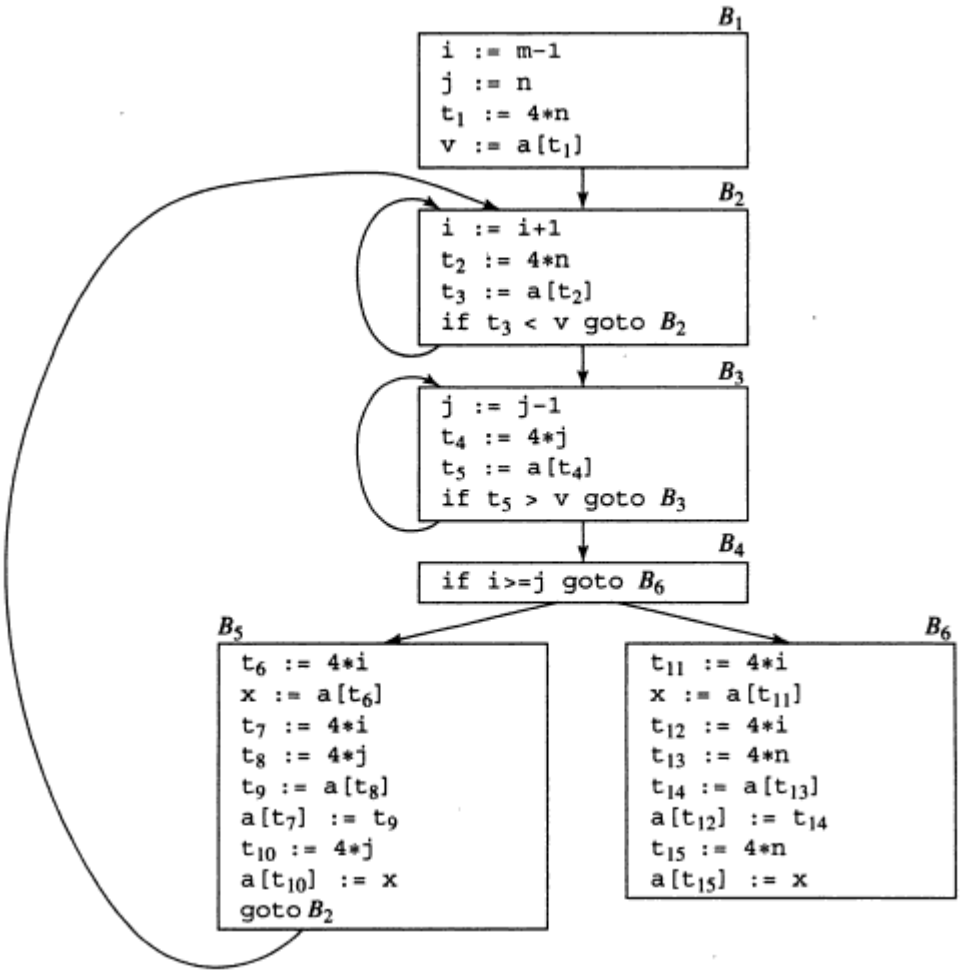


Fig. 5.2 Flow graph

Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2)     /* statement does not change limit*/
```

Code motion will result in the equivalent of

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

Induction Variables :

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

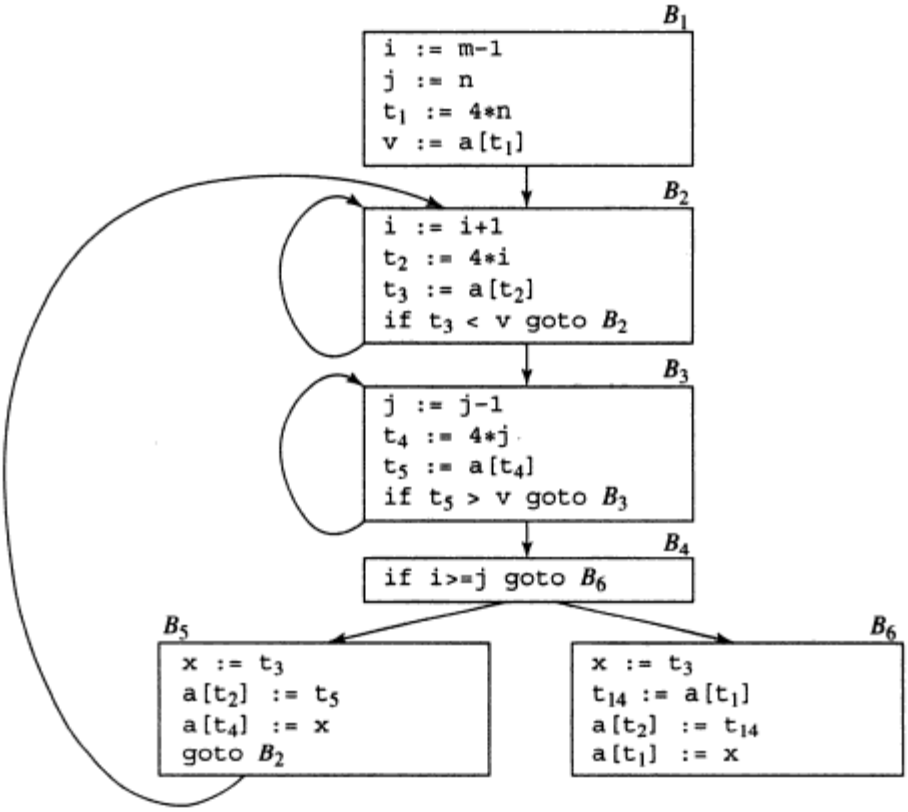


Fig. 5.3 B5 and B6 after common subexpression elimination

5.3 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produces 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 be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics 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 R0,a

(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.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 L1
        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)

        If debug ≠0 goto L2
        Print debugging information
L2:     ……                          (c)
```

- As the argument of the 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                  (d)
```

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             (e)
```

can be replaced by

```
        If a < 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        (f)
L3:
```

may be replaced by

```
        If a < b goto L2
        goto L3
        …….
L3:
```

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) 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

```
x := 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.

$X^2 \rightarrow X*X$

Use of 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- -
```

5.4 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-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

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.

Example:

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

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

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

Dead code elimination:

It is 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.

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 a basic block is transformed to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

- Two statements

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

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

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
e :=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 specification 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.

5.5 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. Similarly every node 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.

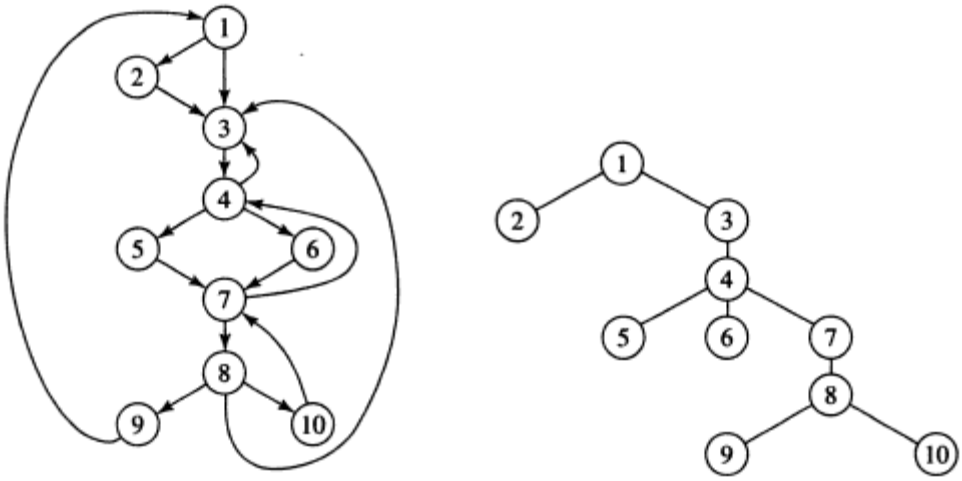


Fig. 5.3(a) Flow graph (b) Dominator tree

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.

***

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 Loops:

One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:

- A loop must have a single entrypoint, 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 way to 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 loops:

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 preheader. The pre-header has only the header as successor, and all edges which formerly entered the header of L from 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.

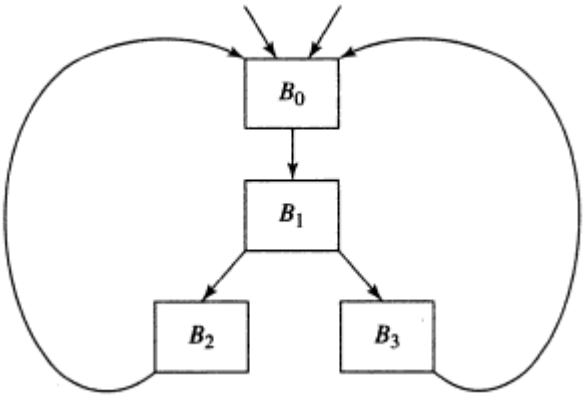


Fig. 5.4 Two loops with the same header

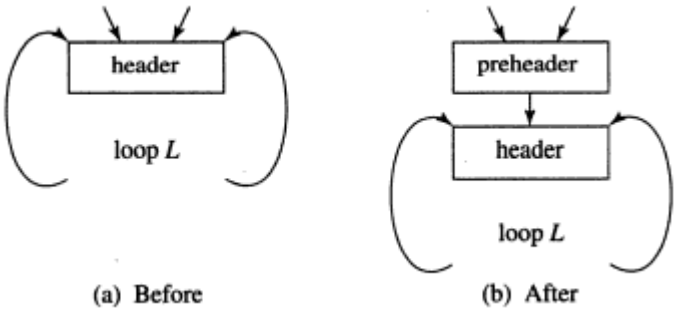


Fig. 5.5 Introduction of the preheader

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

1. There are no umps into the middle of loops from outside;
2. 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.

1. The forward edges from an acyclic graph in which every node can be reached from initial node of G.
2. 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 $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ and $10 \rightarrow 7$ whose heads dominate their tails, the remaining graph is acyclic.

5.6 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] U (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." Such equations are called data-flow equation.

1. 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].

2. 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.

3. 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.

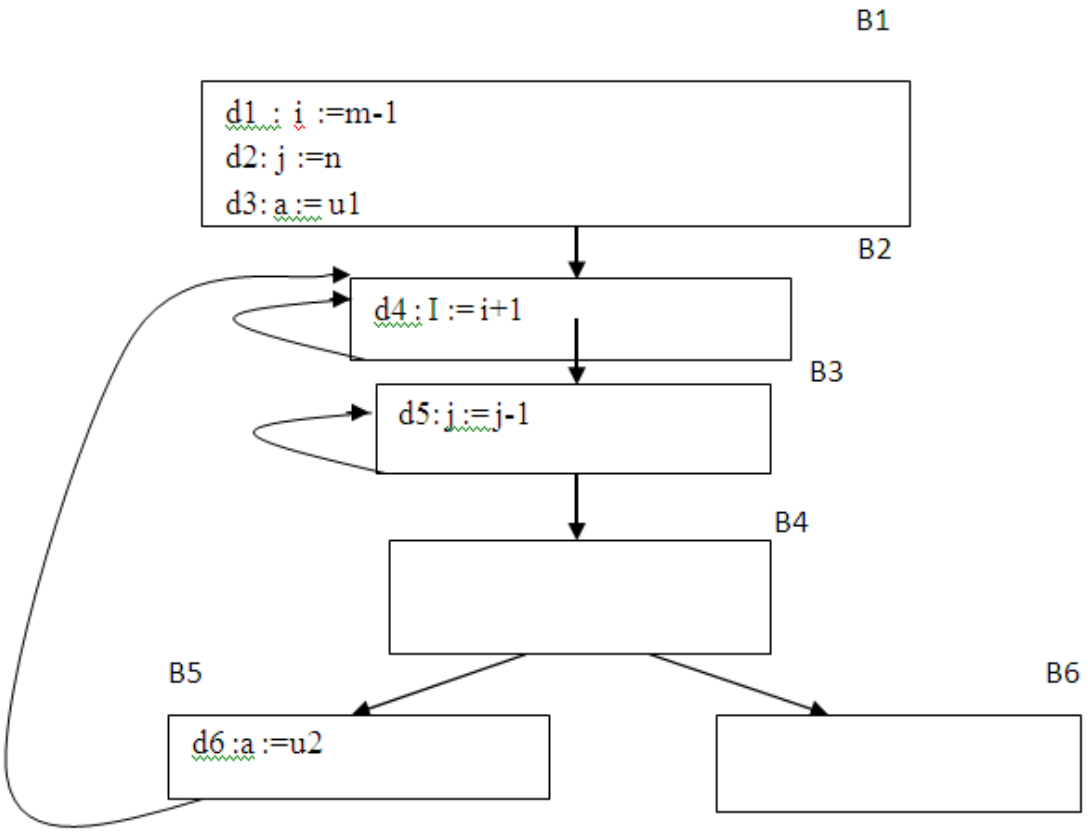


Fig. 5.6 A flow graph

Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

1. Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and pi+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:

1. 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.
2. 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 appearing later along one path.

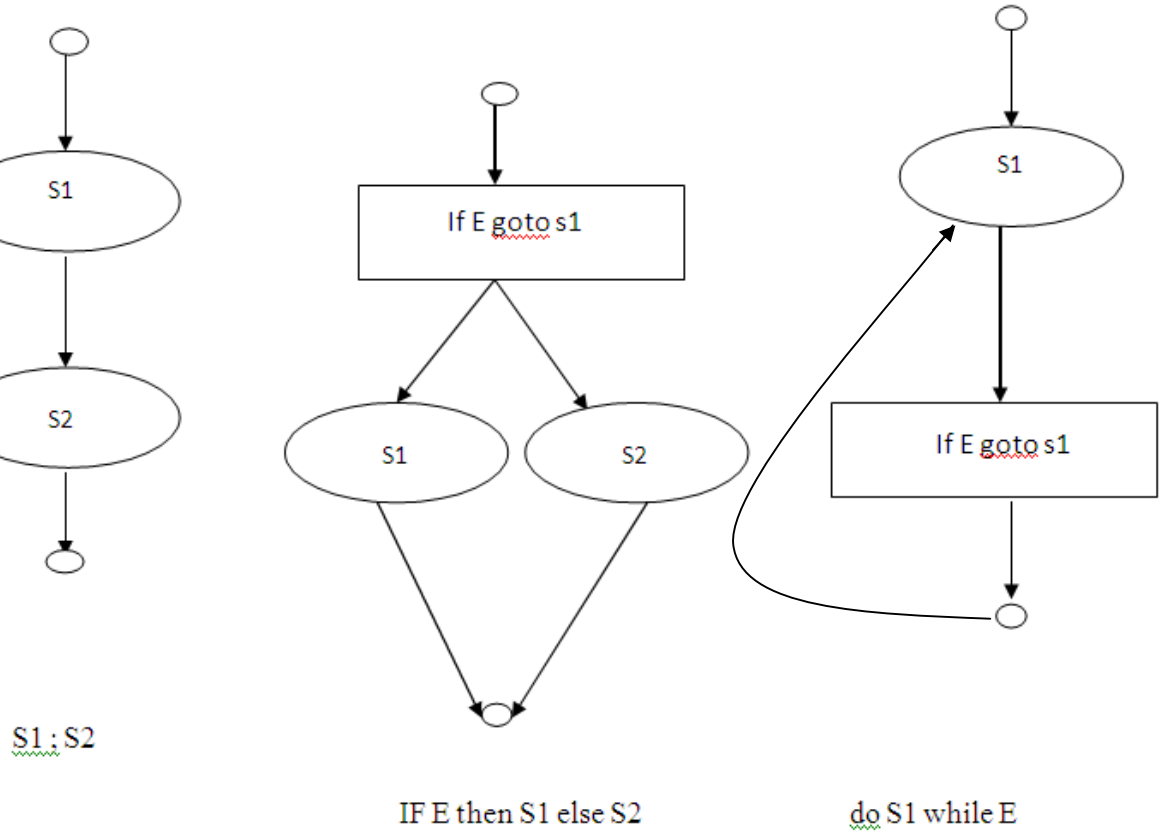


Fig. 5.7 Some structured control constructs

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

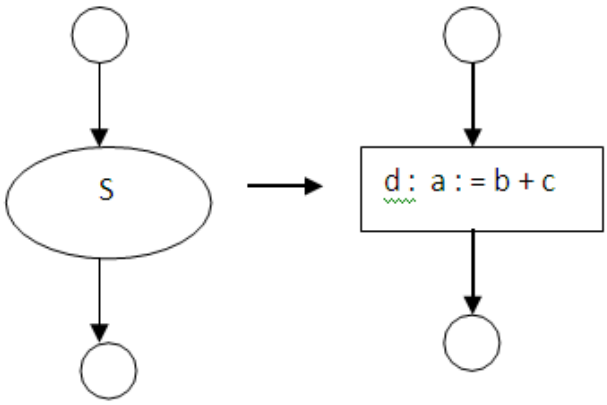
Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

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 statement's region are the beginning and end points, respective 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] = Da - { d }
out [S] = gen [S] U (in[S] - kill[S])

Fig. 5.8 (a) Data flow equations for reaching definitions

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

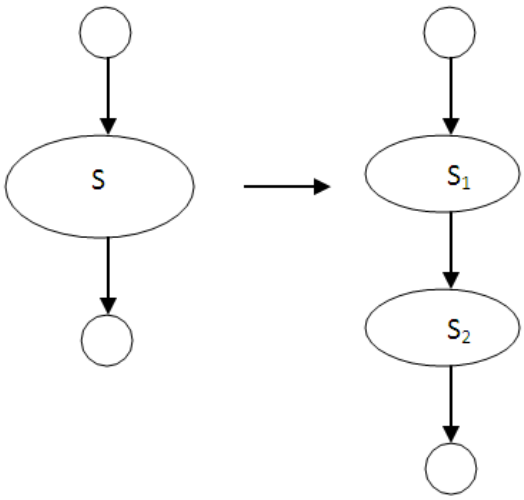
gen[S]={d}

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

Kill[S] = Da - {d}

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

ii)



gen[S]=gen[S2] U (gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] - gen[S2])

in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]

Fig. 5.8 (b) Data flow equations for reaching definitions

Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write

gen[S]=gen[S2] U (gen[S1]-kill[S2])

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

Kill[S] = kill[S2] U (kill[S1] - gen[S2])

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:

Many data-flow problems can be solved by synthesized translation to compute gen and kill. It can be used, for example, to determine computations. However, there are other kinds of data-flow information, such as the reaching-definitions 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 S1; S2, as in the second case. We start by observing in[S1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].

Consider the if-statement. we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S. That is,

in[S1] = in[S2] = in[S]

If a definition reaches the end of S if and only if it reaches the end of one or both substatements; i.e,

out[S]=out[S1] U out[S2]

Representation of sets:

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 programming languages. The difference A-B of sets A and B can be implement 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 of a.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.

5.7 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+z6 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.

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

Some remarks about this algorithm are in order:

1. 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.
2. Not all changes made by algorithm are improvements. We might number of different evaluations reaching s found in step (1), probably to one.
3. 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*zd :=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.

1. Statement s must be the only definition of x reaching u.
2. 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:

1. Determine those uses of x that are reached by this definition of namely, s: x: =y.
2. 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.
3. If s meets the conditions of (2), then remove s and replace all u 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.

1. Mark "invariant" those statements whose operands are all either constant or have all their reaching definitions outside L.
2. Repeat step (3) until at some repetition no new statements are marked "invariant".
3. 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.

1. 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.
2. 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.
3. No use of x in the loop is reached by any definition of x other than 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:

1. Use loop-invariant computation algorithm to find loop-invariant statements.
2. 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.

3. 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 significan 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 ps, 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 ps , 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-in information and live variable information.
OUTPUT: A revised loop.
METHOD:

1. 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 that c is positive. A test of the form 'if i relop x goto B', where x is not an induction variable, is replaced by

   a. r := c*x /* r := x if c is 1. */
   b. r := r+d /* omit if d is 0 */
   c. 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 i1 and i2 in the test if i1 relop i2 goto B, then we check if both i1 and i2 can be replaced. The easy case is when we have j1 with triple and j2 with triple, and c1=c2 and d1=d2. Then, i1 relop i2 is equivalent to j1 relop j2.

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: =s.

GLOSSARY

1. Compiler - a program that reads a program written in one language and translates it in to an equivalent program in another language.

2. Analysis part - breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

3. Synthesis part - constructs the desired target program from the intermediate representation.

4. Structure editor - takes as input a sequence of commands to build a source program.

5. Pretty printer - analyses a program and prints it in such a way that the structure of the program becomes clearly visible.

6. Static checker - reads a program, analyses it and attempts to discover potential bugs without running the program.

7. Linear analysis - This is the phase in which the stream of characters making up the source program is read from left to right and grouped in to tokens that are sequences of characters having collective meaning.

8. Hierarchical analysis - This is the phase in which characters or tokens are grouped hierarchically in to nested collections with collective meaning.

9. Semantic analysis - This is the phase in which certain checks are performed to ensure that the components of a program fit together meaningfully.

10. Loader - is a program that performs the two functions: Loading and Link editing

11. Loading - taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper locations.

12. Link editing - makes a single program from several files of relocatable machine code.

13. Preprocessor - produces input to compilers and expands macros into source language statements.

14. Symbol table - a data structure containing a record for each identifier, with fields for the attributes of the identifier. It allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

15. Assembler - a program, which converts the source language in to assembly language.

16. Lexeme - a sequence of characters in the source program that is matched by the pattern for a token.

17. Regular set - language denoted by a regular expression.

18. Sentinel - a special character that cannot be part of the source program. It speeds-up the lexical analyzer.

19. Regular expression - a method to describe regular language Rules:

20. Recognizers - machines which accept the strings belonging to certain language.

21. Parser - is the output of syntax analysis phase.

22. Error handler - report the presence of errors clearly and accurately and recover from each error quickly enough to be able to detect subsequent errors.

23. Context free grammar - consists of terminals, non-terminals, a start symbol, and productions.

24. Terminals - basic symbols from which strings are formed. It is a synonym for terminal.

25. Nonterminals - syntactic variables that denote sets of strings, which help define the language generated by the grammar.

26. Start symbol - one of the nonterminal in a grammar and the set of strings it denotes is the language defined by the grammar. Ex: S.

27. Context free language - a language that can be generated by a grammar.

28. Left most derivations – the leftmost nonterminal in any sentential form is replaced at each step.

29. Canonical derivations - rightmost nonterminal is replaced at each step are termed.

30. Parse tree - a graphical representation for a derivation that filters out the choice regarding replacement order.

31. Ambiguous grammar - A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

32. Left recursive - A grammar is a left recursive if it has a nonterminal A such that there is a derivation A□Aα for some string α.

33. Left factoring - a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

34. Parsing - the process of determining if a string of tokens can be generated by a grammar.

35. Top Down parsing - Starting with the root, labeled, does the top-down construction of a parse tree with the starting nonterminal.

36. Recursive Descent Parsing - top down method of syntax analysis in which we execute a set of recursive procedures to process the input.

37. Predictive parsing - A special form of Recursive Descent parsing, in which the look-ahead symbol unambiguously determines the procedure selected for each nonterminal, where no backtracking is required.

38. Bottom Up Parsing - Parsing method in which construction starts at the leaves and proceeds towards the root is called as Bottom Up Parsing.

39. Shift-Reduce parsing - A general style of bottom-up syntax analysis, which attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

40. Handle - a sub string that matches the right side of production and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

41. canonical derivations - process of obtaining rightmost derivation in reverse.

42. Viable prefixes - the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.

43. Operator grammar - A grammar is operator grammar if no production rule involves “ε” on the right side.

44. LR parsing method - most general nonbacktracking shift-reduce parsing method.

45. goto function - takes a state and grammar symbol as arguments and produces a state.

46. LR grammar - A grammar for which we can construct a parsing table is said to be an LR grammar.

47. Kernel - The set of items which include the initial item, S̃S, and all items whose dots are not at the left end are known as kernel items.

48. Non kernel items - The set of items, which have their dots at the left end, are known as non kernel items.

49. Postfix notation - is a linearized representation of a syntax tree.

50. Syntax directed definition - Syntax trees for assignment statement are produced by the syntax directed definition.

51. 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.

52. Quadruple - is a record structure with four fields, op, arg1, arg2 and result.

53. Abstract or syntax tree - A tree in which each leaf represents an operand and each interior node an operator is called as abstract or syntax tree.

54. Triples - is a record structure with three fields op, arg1, arg2

55. Declaration - The process of declaring keywords, procedures, functions, variables, and statements with proper syntax.

56. Boolean Expression - Expressions which are composed of the Boolean operators (and, or, and not) applied to elements that are Boolean variables or relational expressions.

57. Calling sequence - A sequence of actions taken on entry to and exit from each procedure.

58. Back patching - the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

59. Basic blocks - A sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch.

60. Flow graph - The basic block and their successor relationships shown by a directed graph is called a flow graph.

61. Virtual machine - An intermediate language as a model assembly language, optimized for a non-existent but ideal computer.

62. Back-end - Intermediate to binary translation is usually done by a separate compilation pass called back end.

63. Relocatable object module - The unpatched binary image is usually called a relocatable object module.

64. Multiregister operations – operations that requires more than one register to perform.

65. Cost of an instruction - one plus the costs associated with the source and destination address modes. This cost corresponds to the length of the instruction.

66. Recursive procedure - A procedure is recursive a new activation can begin before an earlier activation of the same procedure has ended.

67. DAG - a directed acyclic graph with the labels on nodes:

68. Memory management - Mapping names in the source program to addresses of data object in run time memory.

69. Backpatching - Process of leaving a blank slot for missing information and fill in the slot when the information becomes available.

70. Algebraic transformation - change the set of expressions computed by a basic blocks into an algebraically equivalent set.

71. Register descriptor - keeps track of what is currently in each register.

72. Address descriptor - keeps track of the location where the current value of the name can be found at run time.

73. Local optimization - The optimization performed within a block of code.

74. Constant folding - Deducing at compile time that the value of an expression is a constant and using the constant instead.

75. Common Subexpressions - An occurrence of an expression E is called a common subexpression, if E was previously computed, and the values of variables in E have not changed since the previous computation.

76. Dead Code - A variable is live at a point in a program if its value can be used subsequently otherwise, it is dead at that point. The statement that computes values that never get used is known Dead code or useless code.

77. Reduction in strength - replaces an expensive operation by a cheaper one such as a multiplication by an addition.

78. Loop invariant computation - An expression that yields the same result independent of the number of times the loop is executed.

79. Static allocation - the position of an activation record in memory is fixed at run time.

80. Activation tree - A tree which depicts the way of control enters and leaves activations.

81. Control stack - A stack which is used to keep track of live procedure actions.

82. Heap - A separate area of run-time memory which holds all other information.

83. Padding - Space left unused due to alignment consideration.

84. Call sequence - allocates an activation record and enters information into its fields

85. Return sequence - restores the state of the machine so that calling procedure can continue execution.

86. Dangling reference - occurs when there is storage that has been deallocated.

87. Lexical or static scope rule - determines the declaration that applies to a name by a examining the program text alone.

88. Dynamic scope rule - determines the declaration applicable to name at runtime, by considering the current activations.

89. Block - a statement containing its own data declaration.

90. Access link - a pointer to each activation record which obtains a direct implementation of lexical scope for nested procedure.

91. Environment - refers to a function that maps a name to a storage location.

92. State - refers to a function that maps a storage location to the value held there.

93. Peephole optimization - a technique used in many compliers, in connection with the optimization of either intermediate or object code.

TUTORIAL PROBLEMS AND WORKED OUT EXAMPLES

Problem 1

Table-based LL(1) Predictive Top-Down Parsing

Consider the following CFG G = (N={S, A, B, C, D}, T={a,b,c,d}, P, S) where the set of productions P is given below:

S → A
A → BC | DBC
B → Bb | ε
C → c | ε
D → a | d

a) Is this grammar suitable to be parsed using the recursive descendent parsing method? Justify and modify the grammar if needed.

b) Compute the FIRST and FOLLOW set of non-terminal symbols of the grammar resulting from your answer in a)

c) Show the stack contents, the input and the rules used during parsing for the input w = dbb

d) Construct the corresponding parsing table using the predictive parsing LL method.

Answers:

a) No because it is left-recursive. You can expand B using a production with B as the left-most symbol without consuming any of the input terminal symbols. To eliminate this left recursion we add another non-terminal symbol, B' and productions as follows:

S → A
A → BC | DBC
B → bB' | ε
B' → bB' | ε
C → c | ε
D → a | d

b) FIRST(S) = { a, b, c, d, ε } FOLLOW(S) = { $ }
FIRST(A) = { a, b, c, d, ε } FOLLOW(A) = { $ }
FIRST(B) = { b, ε } FOLLOW(B) = { c, $ }
FIRST(B') = { b, ε } FOLLOW(B') ={ c, $ }
FIRST(C) = { c, ε } FOLLOW(C) = { $ }
FIRST(D) = { a, d } FOLLOW(D) = { b, c, $ }
Non-terminals A, B, B', C and S are all nullable.

c) The stack and input are as shown below using the predictive, table-driven parsing algorithm:

STACK	INPUT	RULE/OUTPUT
$S	dbb$	
$ A	dbb$	S → A
$ CBD	dbb$	A → DBC
$ CBd	dbb$	D → d
$ CB	bb$	
$ CB' b	bb$	B → bB'
$ CB'	b$	

$ CB'b	b$	B → bB'
$ CB'	$	
$ C	$	B' → ε
$	$	C → ε
$	$	halt or accept

d) The parsing table is as shown below:

	a	b	c	d	$
S	S→A	S→A	S → A	S→A	S→A
A	A→DBC	A→BC	A→BC	A→DBC	A→BC
B		B→b B'	B→ε		B→ε
B'		B'→b B'	B'→ε		B'→ε
C			C → c		C → ε
D	D → a			D → d	

Problem 2

Perform left recursion and left factoring.

S → ( )
S → a
S → (A)
A → S
A → A , S

Answer

S → (S'
S → a
S' →)
S' → A)
A → SA'
A' → ,SA'
A' → ε

Problem 3

Eliminate immediate left recursion for the following grammar E->E+T | T, T->T * F | F, F-> (E) | id.

Answer

The rule to eliminate the left recursion is A->Aα|β can be converted as A-> βA' and A'-> αA'|ε. So, the grammar after eliminating left recursion is
E->TE'; E'->+TE'| ε; T->FT'; T'->*FT' | ε; F-> (E) | id.

Problem 4

Perform backpatching for the source code:

if a or b then
if c then
x= y+1

Translation:

if a go to L1
if b go to L1
go to L3
L1: if c goto L2
goto L3
L2: x= y+1
L3:

Answer

After Backpatching:

100: if a goto 103
101: if b goto 103
102: goto 106
103: if c goto 105
104: goto 106
105: x=y+1
106:

Problem 5

Find the SLR parsing table for the given grammar and parse the sentence (a+b)*c. E->E+E | E*E | (E) | id.

Answer

Given grammar:

1. E->E+E
2. E->E*E
3. E->(E)
4. E->id

Augmented grammar

E'->E
E->E+E
E->E*E
E->(E)
E->id

I0: E'->.E

E->.E+E
E->.E*E
E->.(E)
E->.id

I1: goto(I0, E)

E'->E.
E->E.+E
E->E.*E

I2: goto(I0, ()

E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: goto(I0, id)

E->id.
I4: goto(I1, +)
E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id
I5: goto(I1, *)
E->E*.E
E->.E+E
E->.E*E
E->.(E)
E->.id
I6: goto(I2, E)
E->(E.)
E->E.+E
E->E.*E
I7: goto(I4, E)
E->E+E.
E->E.+E
E->E.*E
I8: goto(I5, E)
E->E*E.
E->E.+E
E->E.*E
goto(I2, ()=I2
goto(I2, id)=I3

States	Action						Goto
	+	*	(	)	id	$	E
0			S2		S3		1
1	S4	S5				Acc	
2			S2		S3		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	S4, r1	S5, r1		r1		r1	
8	S4, r2	S5, r2		r2		r2	
9	r3	r3		r3		r3	

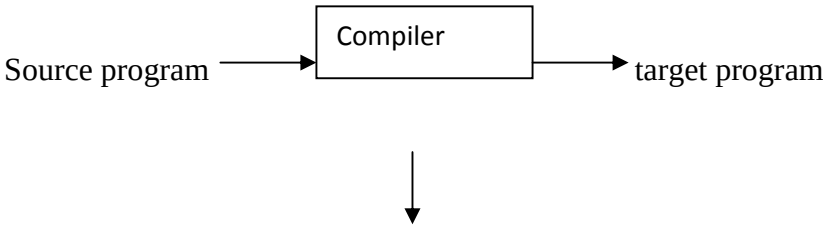
CS2352-PRINCIPLES OF COMPILER DESIGN

2 MARK QUESTIONS WITH ANSWERS

UNIT I

1. What is a Complier?

A Complier is a program that reads a program written in one language-the source language-and translates it in to an equivalent program in another language-the target language . As an important part of this translation process, the compiler reports to its user the presence of errors in the source program



2. State some software tools that manipulate source program?

i. Structure editors

ii. Pretty printers

iii. Static checkers

iv. Interpreters.

3. What are the cousins of compiler? April/May 2004, April/May 2005

The following are the cousins of compilers

i. Preprocessors

ii. Assemblers

iii. Loaders

iv. Link editors.

4. What are the main two parts of compilation? What are they performing?

The two main parts are

- **Analysis** part breaks up 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

5. What is a Structure editor?

A 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.

6. What are a Pretty Printer and Static Checker?

- A Pretty printer analyses a program and prints it in such a way that the structure of the program becomes clearly visible.
- A static checker reads a program, analyses it and attempts to discover potential bugs with out running the program.

7. How many phases does analysis consists?

Analysis consists of three phases

i .Linear analysis

ii .Hierarchical analysis

iii. Semantic analysis

8. What happens in linear analysis?

This is the phase in which the stream of characters making up the source program is read from left to right and grouped in to tokens that are sequences of characters having collective meaning.

9. What happens in Hierarchical analysis?

This is the phase in which characters or tokens are grouped hierarchically in to nested collections with collective meaning.

10. What happens in Semantic analysis?

This is the phase in which certain checks are performed to ensure that the components of a program fit together meaningfully.

11. State some compiler construction tools?Arpil /May 2008

i. Parse generator

ii. Scanner generators

iii. Syntax-directed translation engines

iv. Automatic code generator

v. Data flow engines.

12. What is a Loader? What does the loading process do?

A Loader is a program that performs the two functions

i. Loading

ii .Link editing

The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper locations.

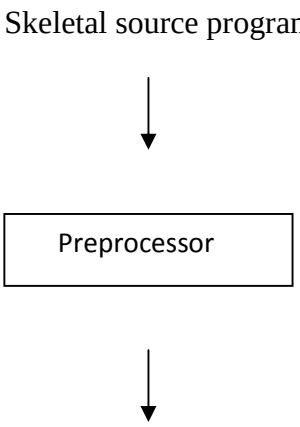
13. What does the Link Editing does?

Link editing: This allows us to make a single program from several files of relocatable machine code. These files may have been the result of several compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

14. What is a preprocessor? Nov/Dev 2004

A preprocessor is one, which produces input to compilers. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program called a preprocessor.

The preprocessor may also expand macros into source language statements.



Source program

15. State some functions of Preprocessors

i) Macro processing
ii) File inclusion
iii) Relational Preprocessors
iv) Language extensions

16. What is a Symbol table?

A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

17. State the general phases of a compiler

i) Lexical analysis
ii) Syntax analysis
iii) Semantic analysis
iv) Intermediate code generation
v) Code optimization
vi) Code generation

18. What is an assembler?

Assembler is a program, which converts the source language in to assembly language.

19. What is the need for separating the analysis phase into lexical analysis and parsing? (Or) What are the issues of lexical analyzer?

- Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

20. What is Lexical Analysis?

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

21. What is a lexeme? Define a regular set. Nov/Dec 2006

- A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A language denoted by a regular expression is said to be a regular set

22. What is a sentinel? What is its usage? April/May 2004

A Sentinel is a special character that cannot be part of the source program. Normally we use ‘eof’ as the sentinel. This is used for speeding-up the lexical analyzer.

23. What is a regular expression? State the rules, which define regular expression?

Regular expression is a method to describe regular language

Rules:

1) ε-is a regular expression that denotes {ε} that is the set containing the empty string
2) If a is a symbol in ∑,then a is a regular expression that denotes {a}
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 L(r) U L(s).
   b) (r)(s) is a regular expression denoting L(r)L(s)
   c) (r)* is a regular expression denoting L(r)*.
   d) (r) is a regular expression denoting L(r).

24. What are the Error-recovery actions in a lexical analyzer?

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

25. Construct Regular expression for the language

L= {w ε{a,b}/w ends in abb}

Ans: {a/b}*abb.

26. What is recognizer?

Recognizers are machines. These are the machines which accept the strings belonging to certain language. If the valid strings of such language are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected.

UNIT II

1. **What is the output of syntax analysis phase? What are the three general types of parsers for grammars?**
   Parser (or) parse tree is the output of syntax analysis phase.

   General types of parsers:

   1) Universal parsing
   2) Top-down
   3) Bottom-up
2. **What are the different strategies that a parser can employ to recover from a syntactic error?**
   - Panic mode
   - Phrase level
   - Error productions
   - Global correction
3. **What are the goals of error handler in a parser?**
   The error handler in a parser has simple-to-state goals:

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

4. **What is phrase level error recovery?**

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. This is known as phrase level error recovery.

5. **How will you define a context free grammar?**

A context free grammar consists of terminals, non-terminals, a start symbol, and productions.

i. Terminals are the basic symbols from which strings are formed. "Token" is a synonym for terminal. Ex: **if, then, else.**
ii. Nonterminals are syntactic variables that denote sets of strings, which help define the language generated by the grammar. Ex: stmt, expr.
iii. Start symbol is one of the nonterminals in a grammar and the set of strings it denotes is the language defined by the grammar. Ex: S.
iv. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings Ex: expr **id**

6. **Define context free language. When will you say that two CFGs are equal?**
   - A language that can be generated by a grammar is said to be a context free language.
   - If two grammars generate the same language, the grammars are said to be equivalent.
7. **Differentiate sentence and sentential form.**

Sentence	**Sentential form**
• If $S \overset{*}{\Rightarrow} w$ then the string w is called Sentence of G. • Sentence is a string of terminals. Sentence is a sentential form with no nonterminals.	• If $S \Rightarrow \alpha$ then α is a sentential form of G. • Sentential form may contain non terminals.

8. **Give the definition for leftmost and canonical derivations.**
   - Derivations in which only the leftmost nonterminal in any sentential form is replaced at each step are termed leftmost derivations
   - Derivations in which the rightmost nonterminal is replaced at each step are termed canonical derivations.
9. **What is a parse tree?**

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some nonterminal A and that the children of the node are labeled from left to

right by symbols in the right side of the production by which this A was replaced in the derivation. The leaves of the parse tree are terminal symbols.

10. What is an ambiguous grammar? Give an example.

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous
- An ambiguous grammar is one that produces more than one leftmost or rightmost derivation for the same sentence.

Ex:

$E \longrightarrow E+E / E*E /$ **id**

11. Why do we use regular expressions to define the lexical syntax of a language?

i. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
ii. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.
iii. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.
iv. Separating the syntactic structure of a language into lexical and non lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

12. When will you call a grammar as the left recursive one?

A grammar is a left recursive if it has a nonterminal **A** such that there is a derivation $A \Rightarrow A\alpha$ for some stringα.

13. Define left factoring.

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal "**A**", we may be able to rewrite the "**A**" productions to refer the decision until we have seen enough of the input to make the right choice.

14. Left factor the following grammar:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b.$

Ans: The left factored grammar is,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

15. What is parsing?

Parsing is the process of determining if a string of tokens can be generated by a grammar.

16. What is Top Down parsing?

Starting with the root, labeled, does the top-down construction of a parse tree with the starting nonterminal, repeatedly performing the following steps.

i. At node n, labeled with non terminal "**A**", select one of the productions for "**A"** and construct children at n for the symbols on the right side of the production.
ii. Find the next node at which a sub tree is to be constructed.

17. What do you mean by Recursive Descent Parsing?

Recursive Descent Parsing is top down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each nonterminal of a grammar.

18. What is meant by Predictive parsing? Nov/Dec 2007

A special form of Recursive Descent parsing, in which the look-ahead symbol unambiguously determines the procedure selected for each nonterminal, where no backtracking is required.

19. Define Bottom Up Parsing.

Parsing method in which construction starts at the leaves and proceeds towards the root is called as Bottom Up Parsing.

20. What is Shift-Reduce parsing?

A general style of bottom-up syntax analysis, which attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

21. Define handle. What do you mean by handle pruning? Nov/Dec 2004, April/May 2005

- An Handle of a string is a sub string that matches the right side of production and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.
- The process of obtaining rightmost derivation in reverse is known as Handle Pruning.

22. Define 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. Thus the production A → XYZ yields the following four items,

A → **.**XYZ

A → X**.**YZ

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

23. What do you mean by viable prefixes?

- The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- A viable prefix is that it is a prefix of a right sentential form that does not continue the past the right end of the rightmost handle of that sentential form.

24. What is meant by an operator grammar? Give an example.

A grammar is operator grammar if,

i. No production rule involves "**ε"** on the right side.
ii. No production has two adjacent nonterminals on the right side..

Ex:

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

25. What are the disadvantages of operator precedence parsing? May/June 2007

i. It is hard to handle tokens like the minus sign, which has two different precedences.
ii. Since the relationship between a grammar for the language being parsed and the operator – precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.
iii. Only a small class of grammars can be parsed using operator precedence techniques.

26. State error recovery in operator-Precedence Parsing.

There are two points in the parsing process at which an operator-precedence parser can discover the syntactic errors:

i. If no precedence relation holds between the terminal on top of the stack and the current input.
ii. If a handle has been found, but there is no production with this handle as a right side.

27. LR (k) parsing stands for what?

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 of lookahead that are used in making parsing decisions.

28. Why LR parsing is attractive one?

- LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.
- The LR parsing method is the, most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift reduce methods.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

29. What is meant by *goto* function in LR parser? Give an example.

- The function *goto* takes a state and grammar symbol as arguments and produces a state.
- The *goto* function of a parsing table constructed from a grammar G is the transition function of a DFA that recognizes the viable prefixes of G.

Ex: *goto*(I,X)

Where I is a set of items and X is a grammar symbol to be the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I

30. Write the configuration of an LR parser?

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$

31. Define LR grammar.

A grammar for which we can construct a parsing table is said to be an LR grammar.

32. What are kernel and non kernel items?Nov/Dec 2005

i. The set of items which include the initial item, $S' \rightarrow .S$, and all items whose dots are not at the left end are known as kernel items.

ii. The set of items, which have their dots at the left end, are known as non kernel items.

33. Why SLR and LALR are more economical to construct than canonical LR?

For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like Pascal. The canonical LR table would typically have several thousand states for the same size language. Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

34. What is ambiguous grammer? Give an example. Nov/Dec 2005, Nov/Dec 2007

A grammer G is said to be ambiguous if it generates more than one parse trees for sentence of language L(G).

Example: E-> E+E|E*E|id

UNIT III

1. What are the benefits of using machine-independent intermediate form?

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

2. List the three kinds of intermediate representation.

The three kinds of intermediate representations are

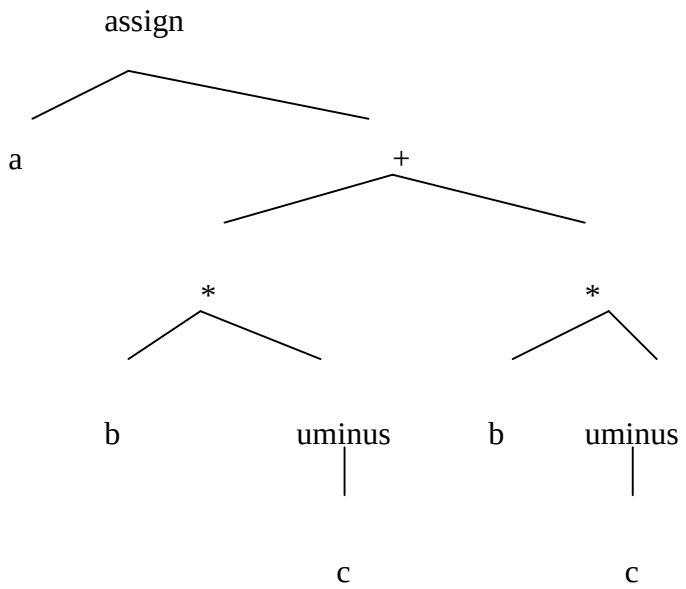
i. Syntax trees
ii. Postfix notation
iii. Three address code

3. How can you generate three-address code?

The three-address code is generated using semantic rules that are similar to those for constructing syntax trees for generating postfix notation.

4. What is a syntax tree? Draw the syntax tree for the assignment statement
a **:=** b * -c + b * -c.

- A syntax tree depicts the natural hierarchical structure of a source program.
- Syntax tree:



5. What is postfix notation?

A Postfix notation is a linearized representation of a syntax tree. It is a list of nodes of the tree in which a node appears immediately after its children.

6. What is the usage of syntax directed definition.

Syntax trees for assignment statement are produced by the syntax directed definition.

7. Why "Three address code" is named so?

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

8. Define 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 fixed or floating-point arithmetic operator, or a logical operator on boolean-valued data.

- 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.

9. State quadruple

A quadruple is a record structure with four fields, which we call op, arg1, arg2 and result.

10. What is called an abstract or syntax tree?

A tree in which each leaf represents an operand and each interior node an operator is called as abstract or syntax tree.

11. Construct Three address code for the following

position := initial + rate * 60

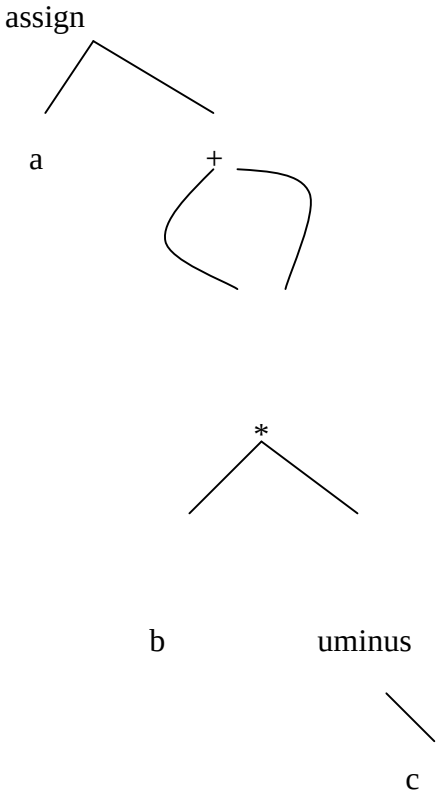
Ans:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

12. What are triples?

- The fields arg1,and arg2 for the arguments of op, are either pointers to the symbol table or pointers into the triple structure then the three fields used in the intermediate code format are called triples.
- In other words the intermediate code format is known as triples.

13. Draw the DAG for a := b * -c + b * -c



14. List the types of three address statements.

The types of three address statements are

a. Assignment statements
b. Assignment Instructions
c. Copy statements
d. Unconditional Jumps
e. Conditional jumps
f. Indexed assignments
g. Address and pointer assignments
h. Procedure calls and return

15. What are the various methods of implementing three-address statements?

i. Quadruples
ii. Triples
iii. Indirect triples

16. What is meant by declaration?

The process of declaring keywords, procedures, functions, variables, and statements with proper syntax is called declaration.

17. How semantic rules are defined?

The semantic rules are defined by the following ways

a. mktable(previous)
b. enter(table,name,type,offset)
c. addwith(table,width)
d. enterproc(table,name,newtable)

18. What are the two primary purposes of Boolean Expressions?

- They are used to compute logical values
- They are used as conditional expressions in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.

19. Define Boolean Expression.

Expressions which are composed of the Boolean operators (and, or, and not) applied to elements that are Boolean variables or relational expressions are known as Boolean expressions

20. What are the two methods to represent the value of a Boolean expression?

i. The first method is to encode true and false numerically and to evaluate a Boolean expression analogously to an arithmetic expression.
ii. The second principal method of implementing Boolean expression is by flow of control that is representing the value of a Boolean expression by a position reached in a program.

21. What do you mean by viable prefixes. Nov/Dec 2004

Viable prefixes are the set of prefixes of right sentinels forms that can appear on the stack of shift/reduce parser are called viable prefixes. It is always possible to add terminal symbols to the end of the viable prefix to obtain a right sentential form.

22. What is meant by Shot-Circuit or jumping 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.

23. What is known as calling sequence?

A sequence of actions taken on entry to and exit from each procedure is known as calling sequence.

24. What is the intermediate code representation for the expression a or b and not c? (Or) Translate a or b and not c into three address code.

Three-address sequence is

t_1 := not c

t_2 := b and t_1

t_3 := a or t_2

25. Translate the conditional statement if *a<b then 1 else 0* into three address code.

Three-address sequence is

100: if a < b goto 103

101: t := 0

102: goto 104

103: t := 1

104:

26. Explain the following functions:

i) makelist(*i*) ii) merge(p_1,p_2) iii) backpatch(*p*,*i*)

i. 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.
ii. merge(p_1,p_2) concatenates the lists pointed to by p_1 *and* p_2 , and returns a pointer to the concatenated list.
iii. backpatch(*p*,*i*) inserts *i* as the target label for each of the statements on the list pointed to by *p*.

27. Define back patching. May/June 2007 & Nov/Dec 2007

Back patching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

28. What are the methods of representing a syntax tree?

i. Each node is represented as a record with a field for its operator and additional fields for pointers to its children.
ii. Nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.

29. Give the syntax directed definition for if-else statement.

Ans:

Production	Semantic rule
$S \rightarrow$ if E then S_1 else S_2	E.true := newlabel; E.false := newlabel; S_1.next := S.next S_2.next := S.next S.code := E.code || gen(E.true ':') || S_1.code || gen('goto' S.next) || gen(E.false ':') || S_2.code

UNIT –IV

1. What are basic blocks?

A sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch , are called basic blocks.

2. What is a flow graph?

- The basic block and their successor relationships shown by a directed graph is called a flow graph.
- The nodes of a flow graph are the basic blocks.

3. Mention the applications of DAGs.

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

4. What are the advantages and disadvantages of register allocation and assignments?

- Advantages:
  i. It simplifies the design of a compiler
- Disadvantages:
  i. It is applied too strictly.
  ii. It uses registers in efficiently. Certain registers may go unused over substantial portions of the code, while unnecessary load and stores are generated.

5. What is meant by virtual machine?

An intermediate language as a model assembly language, optimized for a non-existent but ideal computer called a virtual machine.

6. Discuss back-end and front end?

- Back-end
  i. Intermediate to binary translation is usually done by a separate compilation pass called back end.
- Front end
  i. There are several back ends for different target machines, all of which use the same parser and code generator called front end.

7. Define relocatable object module.

The unpatched binary image is usually called a relocatable object module.

8. What is meant by multiregister operations?

We can modify our labeling algorithm to handle operations like multiplication, division, or function calls which normally requires more than one register to perform. Hence this operation is called multiregister operations.

9. What is meant by peephole optimization?

Peephole optimization is a technique used in many compliers, in connection with the optimization of either intermediate or object code. It is really an attempt to overcome the difficulties encountered in syntax directed generation of code.

10. List the types of addressing modes:-

i) Intermediate mode
ii) Direct mode
iii) Indirect mode
iv) Effective address mode

11. What is input to code generator?

The input to code generator consists of the intermediate representation of the source program produced by the front end together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

12. How the use of registers is subdivided into 2 sub-problems?

- During register allocation we select the set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

13. How would you calculate the cost of an instruction?

- The cost of an instruction to be one plus the costs associated with the source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero, while those with a memory location or literal in them have cost one.

14. What are the primary structure preserving transformations on basic blocks?

- Common sub-expression elimination
- Dead-code elimination
- Renaming of temporary variable
- Interchange of 2 independent adjacent statements.

15. Give some examples for 3 address statements.

- Call
- Return
- Halt
- Action

16. What are the characteristics of peephole optimization?

- Redundant –instruction elimination
- Flow-of control optimizations
- Algebraic simplifications
- Use of machine idioms

17. What is a recursive procedure?

A procedure is recursive a new activation can begin before an earlier activation of the same procedure has ended.

18. What are the common methods for associating actual and formal parameters?

- Call-by-value
- Call-by-reference

- Copy-restore
- Call-by-name
- Macro-expansion

19. Define DAG. Nov/Dec 2007

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

i) Leaves are labeled by unique identifiers, either variable names or constants.
ii) Interior nodes are labeled by an operator symbol.
iii) Nodes are also optionally given a sequence of identifiers for labels.

20. What are the issues in the design of code generators? Nov/Dec 2007

i) Input to the code generator
ii) Target programs
iii) Memory management
iv) Instruction selection
v) Register allocation
vi) Choice of evaluation order
vii) Approaches to code generation

21. What are the various forms of target programs?

i) Absolute machine language
ii) Relocatable machine language
iii) Assembly language

22. What is memory management?

Mapping names in the source program to addresses of data object in run time memory done comparatively by the front end and the code generator is called memory management

23. What is backpatching? April/May 2008

Process of leaving a blank slot for missing information and fill in the slot when the information becomes available is known as backpatching.

24. What are the rules to determine the leaders of basic blocks?

i) The first statement is a leader
ii) Any statement that is the target of a conditional or unconditional goto is a leader
iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

25. What is the use of algebraic transformation?

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

26. What is meant by loop?

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

i) All nodes in the collection are strongly connected i.e., from any node in the loop to any other, there is a path of length one or more, wholly within the loop

ii) The collection of nodes has a unique entry, i.e. a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

27. What is register descriptor and address descriptor?

- A register descriptor keeps track of what is currently in each register.
- An address descriptor keeps track of the location where the current value of the name can be found at run time.

28. What are the characteristics of peephole optimization?

i) Redundant – instruction elimination iii) Flow – of – control optimizations
ii) Algebraic simplifications iv) Use of machine idioms

UNIT –V

1. How the quality of object program is measured?

The quality of an object program is measured by its Size or its running time. For large computation running time is particularly important. For small computations size may be as important or even more.

2. What is the more accurate term for code optimization?

The more accurate term for code optimization would be "code improvement"

3. Explain the principle sources of optimization.

Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation. The techniques consist of detecting patterns in the program and replacing these patterns by equivalent and more efficient constructs.

4. What are the patterns used for code optimization?

The patterns may be local or global and replacement strategy may be a machine dependent or independent

5. What are the 3 areas of code optimization?

- Local optimization
- Loop optimization
- Data flow analysis

6. Define local optimization.

The optimization performed within a block of code is called a local optimization.

7. Define constant folding.

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

8. What do you mean by inner loops?

The most heavily traveled parts of a program, the inner loops, are an obvious target for optimization. Typical loop optimizations are the removal of loop invariant computations and the elimination of induction variables.

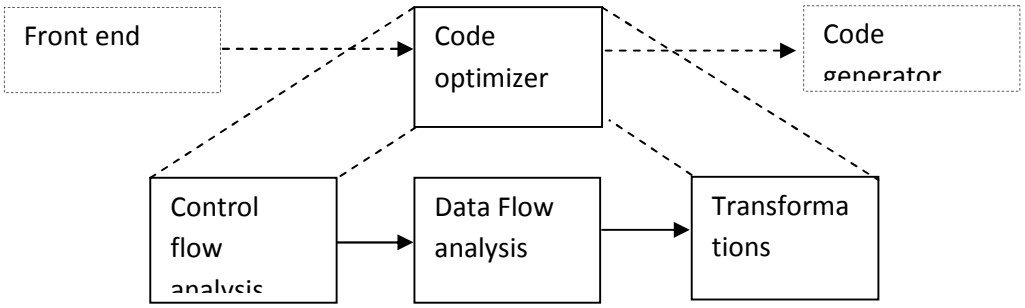
9. What is code motion? April/May 2004, May/June 2007, April/May-2008

Code motion is an important modification that decreases the amount of code in a loop.

10. What are the properties of optimizing compilers?

- Transformation must preserve the meaning of programs.
- Transformation must, on the average, speed up the programs by a measurable amount
- A Transformation must be worth the effort.

11. Give the block diagram of organization of code optimizer.



12. What are the advantages of the organization of code optimizer?

a. The operations needed to implement high level constructs are made explicit in the intermediate code, so it is possible to optimize them.
b. The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine

13. Define Local transformation & Global Transformation.

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.

14. Give examples for function preserving transformations.

- Common subexpression elimination
- Copy propagation
- Dead – code elimination
- Constant folding

15. What is meant by Common Subexpressions?

An occurrence of an expression E is called a common subexpression, if E was previously computed, and the values of variables in E have not changed since the previous computation.

16. What is meant by Dead Code?

A variable is live at a point in a program if its value can be used subsequently otherwise, it is dead at that point. The statement that computes values that never get used is known Dead code or useless code.

17. What are the techniques used for loop optimization?

i) Code motion
ii) Induction variable elimination
iii) Reduction in strength

18. What is meant by Reduction in strength?

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

19. What is meant by loop invariant computation?

An expression that yields the same result independent of the number of times the loop is executed is known as loop invariant computation.

20. Define data flow equations.

A typical equation has the form

Out[S] = gen[S] U (In[S] – kill[S])

and 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". Such equations are called data flow equations.

29. What are the two standard storage allocation strategies?

The two standard allocation strategies are

1. Static allocation.
2. Stack allocation

30. Discuss about static allocation.

In static allocation the position of an activation record in memory is fixed at run time.

31. Write short notes on activation tree. Nov/Dec 2007

- A tree which depicts the way of control enters and leaves activations.
- In an activation tree
  i. Each node represents an activation of an procedure
  ii. The root represents the activation of the main program.
  iii. The node for a is the parent of the node for b , if and only if control flows from activation a to b
  iv. 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.

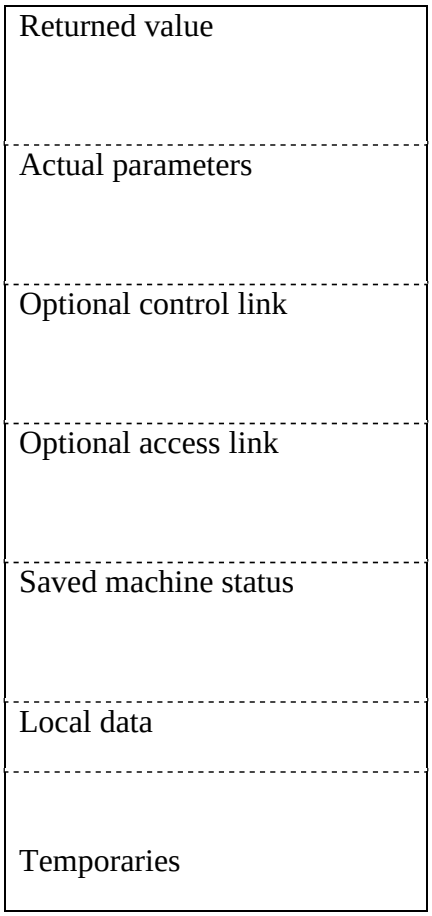
32. Define control stack.

A stack which is used to keep track of live procedure actions is known as control stack.

33. Define heap.

A separate area of run-time memory which holds all other information is called a heap.

34. Give the structure of general activation record



35. Discuss about stack allocation.

In stack allocation a new activation record is pushed on to the stack for each execution of a procedure. The record is popped when the activation ends.

36. What are the 2 approaches to implement dynamic scope?

- Deep access
- Shallow access

37. What is padding?

Space left unused due to alignment consideration is referred to as padding.

38. What are the 3 areas used by storage allocation strategies?

- Static allocation
- Stack allocation

- Heap allocation

39. What are the limitations of using static allocation?

- The size of a data object and constraints on its position in memory must be known at compile time.
- Recursive procedure are restricted, because all activations of a procedure use the same bindings for local name
- Data structures cannot be created dynamically since there is no mechanism for storage allocation at run time

40. Define calling sequence and return sequence.

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

41. When dangling reference occurs?

- A dangling reference occurs when there is storage that has been deallocated.
- It is logical error to use dangling references, since the value of deallocated storage is undefined according to the semantics of most languages.

42. Define static scope rule and dynamic rule

- Lexical or static scope rule determines the declaration that applies to a name by a examining the program text alone.
- Dynamic scope rule determines the declaration applicable to name at runtime, by considering the current activations.

43. What is block? Give its syntax.

- A block is a statement containing its own data declaration.
- Syntax:

```
{

Declaration statements

}
```

44. What is access link?

- An access link is a pointer to each activation record which obtains a direct implementation of lexical scope for nested procedure.

45. What is known as environment and state?

- 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.

46. How the run-time memory is sub-divided?

- Generated target code
- Data objects
- A counterpart of the control stack to keep track of procedure activations.

PART B QUESTIONS

UNIT I- INTRODUCTION TO COMPILING

1. Explain in detail about the role of Lexical analyzer with the possible error recovery actions.

2. (a)Describe the following software tools

i. Structure Editors ii. Pretty printers iii. Interpreters

(b)Write in detail about the cousins of the compiler.

3. Describe in detail about input buffering. What are the tools used for constructing a compiler?

4. (a)Explain the functions of the Lexical Analyzer with its implementation.

(b)Elaborate specification of tokens.

5. (a)What is a compiler? Explain the various phases of compiler in detail,with a neat sketch.

(b) Elaborate on grouping of phases in a compiler.

6. (a) Explain the various phases of a compiler in detail. Also write down the output for the following expression after each phase a:=b*c-d.

(b) What are the phases of the compiler? Explain the phases in detail. Write down the output of each phase for the expression a: = b + c *50.

UNIT II- SYNTAX ANALYSIS

1. What is FIRST and FOLLOW? Explain in detail with an example. Write down

the necessary algorithm.

2. Construct Predictive Parsing table for the following grammar:

S -> (L) / a

L -> L, S/S

and check whether the following sentences belong to that grammar or not.

(i) (a,a)

(ii) (a, (a , a))

(iii) (a, ((a , a), (a , a)))

3. (a) Construct the predictive parser for the following grammar:

S -> (L)|a

L -> L,S|S. (12)

(b) Construct the behaviour of the parser on sentence (a, a) using the grammar:

S -> (L)|a

L -> L,S|S. (4)

4. (a) Check whether the following grammar is SLR (1) or not. Explain your answer with reasons.

S-> L=R L->id

S->R R->L

L->*R

(b) For the grammar given below, calculate the operator precedence relation and the precedence functions.

E -> E + E | E – E | E * E | E / E | E ^ E | (E) | -E | id

5. Check whether the following grammar is a LL(1) grammar

S -> iEtS | iEtSeS | a

E -> b Also define the FIRST and FOLLOW procedures.

6. (a) Consider the grammar given below.

E->E+T T->F

E->T F->(E)

T->T*F F->id.

Construct an LR Parsing table for the above grammar. Give the moves of LR parser on id*id+id.

(7) What is a shift-reduce parser? Explain in detail the conflicts that may occur during shift-reduce parsing.

UNIT III-INTERMEDIATE LANGUAGES

1. How would you generate the intermediate code for the flow of control

statements?Explain with examples.

2. (a)What are the various ways of calling procedures? Explain in detail.

(b)What is a three-address code? Mention its types. How would you implement the three address statements? Explain with examples.

3. How would you generate intermediate code for the flow of control statements?

Explain with examples.

4. (a)Describe the method of generating syntax-directed definition for Control statements.

(b)Give the semantic rules for declarations in a procedure.

5. (a) How Back patching can be used the generate code for Boolean expressions and flow of control statements.

(b)Explain how the types and relative addresses of declared names are computed and how scope information is dealt with.

6. (a) Describe in detail the syntax-directed translation of case statements.

(b) Explain in detail the translation of assignment statements.

UNIT IV- CODE GENERATION

1. (a) Explain the issues in design of code generator.

(b) Explain peephole optimization.

2. (a)Discuss run time storage management of a code generator.

   (b)Explain DAG representation of the basic blocks with an example.

3. (a) Explain the simple code generator with a suitable example.

   (b) Describe about the stack allocation in memory management.

4. (a) What are the different storage allocation strategies?

   (b)What are steps needed to compute the next use information?

5. (a) Write detailed notes on Basic blocks and flow graphs.

   (b) How would you construct a DAG for a Basic block? Explain with an example.

UNIT V- CODE OPTIMIZATION

1. (a) Explain the principle sources of optimization in detail.

   (b)What are the various ways of calling procedures?

2. (a) Discuss about the following:

      i). Copy Propagation ii) Dead-code Elimination and iii) Code motion

   (b)Describe in detail about the stack allocation in memory management.

3. (a) Write about Data flow analysis of structural programs.

   (b)Describe the various storage allocation strategies.

4. (a) Describe in detail the source language issues.

   (b) Explain in detail access to nonlocal names.

5. (a) Elaborate storage organization.

   (b)Write detailed notes on parameter passing.

6. (a) Explain optimization of basic blocks.

   (b) Explain the various approaches to compiler development.

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011
Sixth Semester
Computer Science and Engineering
CS 2352 — PRINCIPLES OF COMPILER DESIGN
(Regulation 2008)

Time : Three hours Maximum : 100 marks

Answer ALL questions

PART A — (10 × 2 = 20 marks)

1. What is an interpreter?
2. Define token and lexeme.
3. What is handle pruning?
4. What are the limitations of static allocation?
5. List out the benefits of using machine-independent intermediate forms.
6. What is a syntax tree? Draw the syntax tree for the following statement: . : c b c b a − □ + − □ =
7. List out the primary structure preserving transformations on basic block.
8. What is the purpose of next-use information?
9. Define dead-code elimination.
10. What is loop optimization?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Describe the various phases of complier and trace the program segment 4 : □ + = c b a for all phases. (10)

(ii) Explain in detail about compiler construction tools. (6)

Or

(b) (i) Discuss the role of lexical analyzer in detail. (8)

(ii) Draw the transition diagram for relational operators and unsigned numbers in Pascal. (8)

12. (a) (i) Explain the error recovery strategies in syntax analysis. (6)

(ii) Construct a SLR construction table for the following grammar.

T E E + →
T E →
F T T □ →
F T →
( ) E F →
id F → (10)

Or

(b) (i) Distinguish between the source text of a procedure and its activation at run time. (8)

(ii) Discuss the various storage allocation strategies in detail. (8)

13. (a) (i) Define three-address code. Describe the various methods of implementing three-address statements with an example. (8)

(ii) Give the translation scheme for converting the assignments into three address code. (8)

Or

(b) (i) Discuss the various methods for translating Boolean expression. (8)

(ii) Explain the process of generating the code for a Boolean expression in a single pass using back patching. (8)

132 132 132
11267 3

14. (a) (i) Write in detail about the issues in the design of a code generator. (10)

(ii) Define basic block. Write an algorithm to partition a sequence of three-address statements into basic blocks. (6)

Or

(b) (i) How to generate a code for a basic block from its dag representation? Explain. (6)

(ii) Briefly explain about simple code generator. (10)

15. (a) (i) Write in detail about function-preserving transformations. (8)

(ii) Discuss briefly about Peephole Optimization. (8)

Or

(b) (i) Write an algorithm to construct the natural loop of a back edge. (6)

(ii) Explain in detail about code-improving transformations. (10)

B.E/B.Tech.DEGREE EXAMINATION,MAY/JUNE 2009
Sixth semester
Computer Science and Engineering
CS1352-PRINCIPLES OF COMPILER DESIGN
(Regulation 2004)

Time: Three hours Maximum:100 marks

Answer All questions
Part A --(10*2=20 marks)

1. What are the issues to be considered in the design of lexical analyzer?
2. Define concrete and abstract syntax with example.
3. Derive the string and construct a syntax tree for the input string ceaedbe using the grammar S->SaA|A,A->AbB|B,B->cSd|e
4. List the factors to be considered for top-down parsing.
5. Why is it necessary to generate intermediate code instead of generating target program itself?
6. Define back patching.
7. List the issues in code generation.
8. Write the steps for constructing leaders in basic blocks.
9. What are the issues in static allocation?
10. What is meant by copy-restor?

PART B--(5*16=80)

11. (a) (i) Explain the need for dividing the compilation process into various phases and explain its functions.
    (ii) Explain how abstract stack machine can be used as translators.

Or

(b) What is syntax directed translation? How it is used for translation of expresions? (16)

12. (a) Given the following grammar S->AS|b,A->SA|a Construct a SLR parsing table for the string baab

Or

(b) Consider the grammar E->E+T,T->T*F,F->(E)\id.Using predictive parsing the string id+id*id. (16)

13. (a) Explain in detail how three address code are generated and implemented.

Or

(b) Explain the role of declaration statements in intermediate code generation.

14. (a) Design a simple code generator and explain with example.

Or

(b) Write short notes on:
1: Peep hole optimization
2: Issues in code generation

15. (a) Explain with an example how basic blocks are optimized.

Or

(b) Explain the storage allocation strategies used in run time environments.

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2011.
Sixth Semester
Computer Science and Engineering
CS 2352 — PRINCIPLES OF COMPILER DESIGN
(Regulation 2008)

Time : Three hours Maximum : 100 marks

PART A — (10 × 2 = 20 marks)

1. What is the role of lexical analyzer?
2. Give the transition diagram for an identifier.
3. Define handle pruning.
4. Mention the two rules for type checking.
5. Construct the syntax tree for the following assignment statement: a:=b*- c+b*-c.
6. What are the types of three address statements?
7. Define basic blocks and flow graphs.
8. What is DAG?
9. List out the criterias for code improving transformations.
10. When does dangling reference occur?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Describe the various phases of compiler and trace it with the program segment (position: = initial+ rate * 60). (10)

(ii) State the complier construction tools. Explain them. (6)

Or

(b) (i) Explain briefly about input buffering in reading the source program for finding the tokens. (8)

(ii) Construct the minimized DFA for the regular expression (0+1)*(0+1) 10. (8)

12. (a) Construct a canonical parsing table for the grammar given below. Also explain the algorithm used. (16)

E → E + T F → (E) E → T F → id . T → T * F T → F

Or

(b) What are the different storage allocation strategies? Explain. (16)

13. (a) (i) Write down the translation scheme to generate code for assignment statement. Use the scheme for generating three address code for the assignment statement g: = a+b-c*d. (8)

(ii) Describe the various methods of implementing three-address statements. (8)

Or

(b) (i) How can Back patching be used to generate code for Boolean expressions and flow of control

statements? (10)
(ii) Write a short note on procedures calls. (6)
14. (a) (i) Discuss the issues in the design of code generator. (10)
(ii) Explain the structure-preserving transformations for basic blocks. (6)

Or

(b) (i) Explain in detail about the simple code generator. (8)
(ii) Discuss briefly about the Peephole optimization. (8)
15. (a) Describe in detail the principal sources of optimization. (16)

Or

(b) (i) Explain in detail optimization of basic blocks with example. (8)
(ii) Write about Data flow analysis of structural programs. (8)

B.E/B.Tech DEGREE EXAMINATION, MAY/JUNE 2012

Sixth semester

Computer science and Engineering

CS2352/Cs62/10144 Cs602-PRINCIPLES OF COMPILER DESIGN

(Regulation 2008)

Answer ALL questions

PART- A

1. Mention few cousins of compiler.
2. What are the possible error recovery actions in lexical analyzer?
3. Define an ambiguous grammar.
4. What is dangling reference?
5. Why are quadruples preferred over triples in an optimizing compiler?
6. List out the motivations for back patching.
7. Define flow graph.
8. How to perform register assignment for outer loops?
9. What is the use of algebraic identities in optimization of basic blocks?
10. List out two properties of reducible flow graph?

PART B

11. (a) (i) What are the various phases of the compiler? Explain each phase in detail.
(ii) Briefly explain the compiler construction tools.

OR

(b) (i) What are the issues in lexical analysis?
(ii)Elaborate in detail the recognition of tokens.
12. (a) (i) Construct the predictive parser for the following grammar. S->(L)/a L->L,S/S
(ii)Describe the conflicts that may occur during shift reduce parsing.

OR

(b) (i)Explain the detail about the specification of a simple type checker.
(ii)How to subdivide a run-time memory into code and data areas. Explain
13. (a) (i) Describe the various types of three address statements.
(ii)How names can be looked up in the symbol table? Discuss.

OR

(b) (i)Discuss the different methods for translating Boolean expressions in detail.
(ii) Explain the following grammar for a simple procedure call statement S->call id (enlist).
14. (a) (i) Explain in detail about the various issues in design of code generator.
(ii)Write an algorithm to partition a sequence of three address statements into basic blocks.

OR

(b) (i)Explain the code-generation algorithm in detail.
(ii)Construct the dag for the following basic block. d: =b*c e: = a +b b: =b*c a: =e-d

15. (a) (i) Explain the principal sources of optimization in detail.
(ii)Discuss the various peephole optimization techniques in detail.

OR

(b) (i) How to trace data-flow analysis of structured program?
(ii) Explain the common sub expression elimination, copy propagation, and transformation for moving loop invariant computations in detail

Computer Science andEngineering
CS2352-Principles of Computer Design
Sixth Semester
May/June 2014 Question Paper
PART-A(10*2=20)

1.State any two reasons as to why phases of compiler should be grouped.
2.Why is buffering used in loxical analysis? What are the commonly used buffering methods?
3.Define Lexeme.
4.Compare the features of DFA and NFA.
5.What is the significance of intermediate code?
6.Write the various three address code form of intermediate code.
7.Define symbol table.
8.Name the techniques in loop optimation.
9.What do you mean by Cross-Compiler?
10. How would you represent the dummy blocks with no statements indicated in global data flow analysis?

PART-B(5*16=80)

11.a)1)Define the following terms :compiler,interpretor,Translator and differentiatebetween them. (6)
2)Differentiate between lexeme,token and pattern.(6)
3)What are the issues in lexical analysis?(4)

Or

b)Explain in detail the process of compilation .Illustrate the output of each phase of compilation of the input “a=(b+c)*(b+c)*2”.. (16)

12.a)Consider the following grammar

S-AS|b
A-SA|a.
Construct the SLR parse table for the grammar .Shoe the actions of the parser for the input string “abab”.(16)

Or

b)1)What is an ambiguous grammar? Is the following grammar ambiguous?
Prove E-E+|E(E)|id. The grammar should be moved to the next line ,centered.
2)Draw NFA for the regular expression ab*/ab.

13.a) How would you convert the following into intermediate code?Give a suitable example.
1)Assignment Statements. 2)Case Statements

Or

b)1)Write notes on backpatching.
2)Explain the sequence of stack allocation process for a function call.
14.a) Discuss the various issues in code generation with examples. (16)

Or

b)Define a directed acyclic graph.Construct a DAG and write the sequence of instructions for the expression a+a*(b-c)+(b-c)*d.(16)
15.a)Discuss in detail the process of optimization of asic blocks.Give an example(16)

Or

b).What is data flow analysis?Explain data flow abstraction with examples.(16)