Sardar Patel University,Balaghat(M.P)

School of Engineering & Technology

Department of CSE

Subject Name: Compiler Design

Subject Code: CSE702

Semester: 7th

CSE702 Compiler Design

UNIT- 1:

1. INTRODUCTION OF COMPILER

A Compiler is a translator from one language, the input or source language, to another language, the output or target language. Often, but not always, the target language is an assembler language or the machine language for a computer processor.

Note that using a compiler requires a two step process to run a program.

1. Execute the compiler (and possibly an assembler) to translate the source program into a machine language program.
2. Execute the resulting machine language program, supplying appropriate input.

Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

1.1 Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessors

Preprocessors are normally fairly simple as in the C language, providing primarily the ability to include files and expand macros. There are exceptions, however. IBM's PL/I, another Algol-like language had quite an extensive preprocessor, which made available at preprocessor time, much of the PL/I language itself (e.g., loops and I believe procedure calls).

Assemblers

Assembly code is a mnemonic version of machine code in which names, rather than binary values, are used for machine instructions, and memory addresses.

Some processors have fairly regular operations and as a result assembly code for them can be fairly natural and not-too-hard to understand. Other processors, in particular Intel's x86 line, have let us charitably say more interesting instructions with certain registers used for certain things.

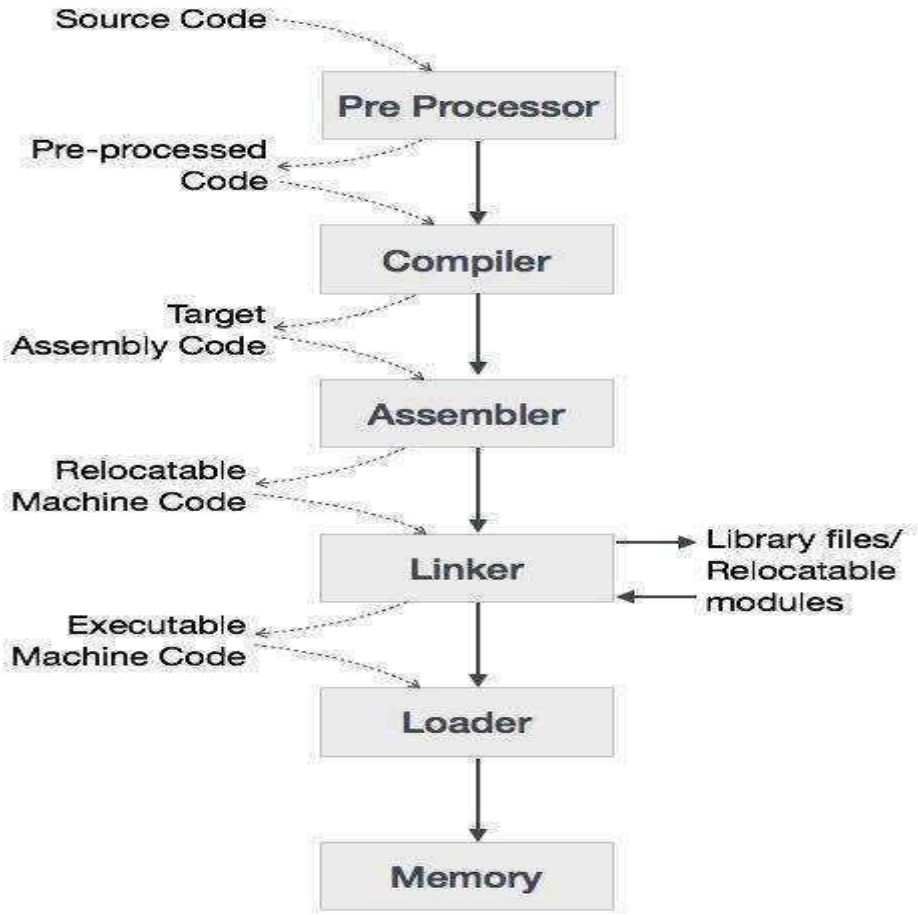


Figure 1.1: Language Processing System

Linkers

The linker has another input, namely libraries, but to the linker the libraries look like other programs compiled and assembled. The two primary tasks of the linker are

- Relocating relative addresses.
- Resolving external references (such as the procedure xor() above).

Loaders

After the linker has done its work, the resulting "executable file" can be loaded by the operating system into central memory. The details are OS dependent. With early single-user operating systems all programs would be loaded into a fixed address (say 0) and the loader simply copies the file to memory. Today it is much more complicated since (parts of) many programs reside in memory at the same time. Hence the compiler/assembler/linker cannot know the real location for an identifier. Indeed, this real location can

change.

Compiler

A Compiler is a translator from one language, the input or *source* language, to another language, the output or *target* language. Often, but not always, the target language is an assembler language or the machine language for a computer processor.

Executing a program written n HLL programming language is basically of two parts. The source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.

Assembler

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

Interpreter

An interpreter is a program that appears to execute a source program as if it were machine language. Execution in Interpreter Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter.

The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.
Type of object that denotes various may change dynamically.

Debugging a program and finding errors is simplified task for a program used for interpretation.
The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is slower.
Memory consumption is more.

2. MAJOR DATA STRUCTURE IN COMPILERSymbol Tables

Symbol Tables are organized for fast lookup. Items are typically entered once and then looked up several times. Hash Tables and Balanced Binary Search Trees are commonly used. Each record contains a “name” (symbol) and information describing it.

Simple Hash Table

Hasher translates “name” into an integer in a fixed range- the hash value. Hash Value indexes into an array of lists.

Entry with that symbol is in that list or is not stored at all. Items with same hash value = bucket.

Balanced Binary Search Tree

Binary search trees work if they are kept balanced. Can achieve logarithmic lookup time. Algorithms are somewhat complex. Red-black trees and AVL trees are used. No leaf is much farther from root than any other.

Parse Tree

The structure of a modern computer language is tree-like. Trees represent recursion well. A grammatical structure is a node with its parts as child nodes. Interior nodes are non terminals. The tokens of the language are leaves.

3. BOOTSTRAPPING & PORTING

Bootstrapping is a technique that is widely used in compiler development. It has four main uses:

- It enables new programming languages and compilers to be developed starting from existing ones.
- It enables new features to be added to a programming language and its compiler.
- It also allows new optimizations to be added to compilers.
- It allows languages and compilers to be transferred between processors with different instruction sets

A compiler is characterized by three languages:

- Source Language
- Target Language
- Implementation Language

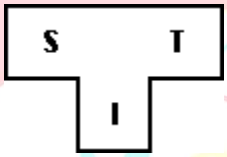


Figure 1.2: T-Diagram

Notation:

${}^{S}C_{I}^{T}$ represents a compiler for Source *S*, Target *T*, implemented in *I*. The *T-diagram* shown above is also used to depict the same compiler.

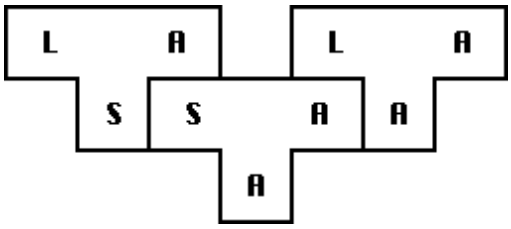
To create a new language, L, for machine A:

1. Create ${}^{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)

2. Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.

3. Compile ${}^{L}C_{S}^{A}$ using ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$, a compiler for language L, which runs on machine A and produces code for machine A.

$${}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A}$$


L A L A
S S A A
A

SARDAR PATEL
UNIVERSITY
SPU
।। ज्ञानेन प्रकाशते जगत् ।।

Figure 1.3: Bootstrapping of Compiler

The process illustrated by the T-diagrams is called *bootstrapping* and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

To produce a compiler for a different machine B:

1. Convert ${}^{L}C_{S}^{A}$ into ${}^{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset of language L.
2. Compile ${}^{L}C_{L}^{B}$ to produce ${}^{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A and produces code for machine B.
3. Compile ${}^{L}C_{L}^{B}$ with the cross-compiler to produce ${}^{L}C_{B}^{B}$, a compiler for language L which runs on machine B.

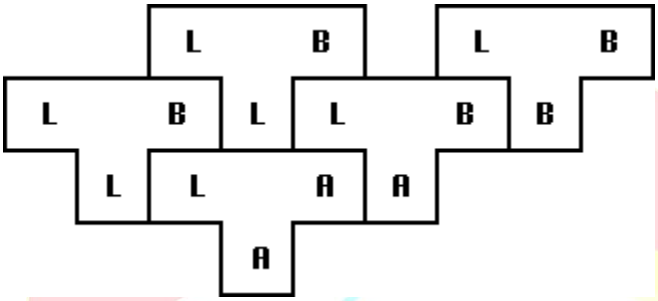


Figure 1.4: Porting of Compiler

4. STRUCTURE OF THE COMPILER: ANALYSIS AND SYNTHESIS MODEL OF COMPILATION

A compiler can broadly be divided into two phases based on the way they compile.

Analysis Model

Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

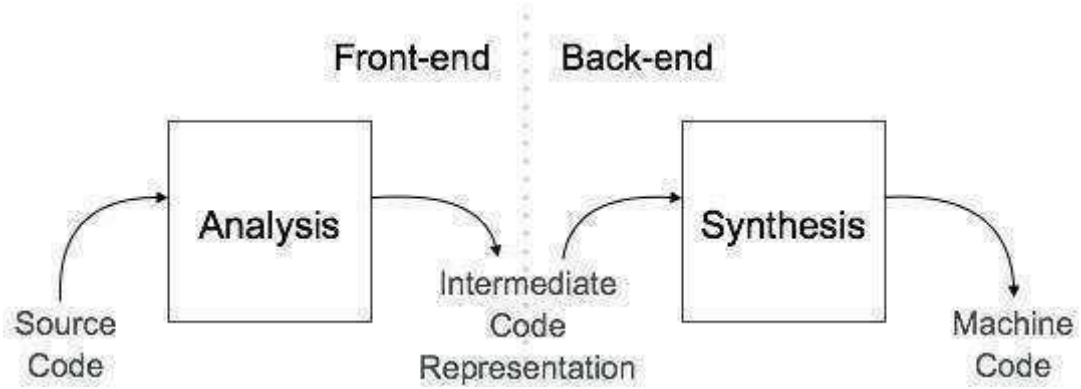


Figure 1.5: Analysis and Synthesis phase of Compiler

Synthesis Model

Known as the back-end of the compiler, the synthesis phase generates the target program with the help of

intermediate source code representation and symbol table.

A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.

- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

4.1 Various Phase of Compiler

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. Compilation process is partitioned into no-of-sub processes called 'phases'. The phases of a compiler are shown in below.

Lexical Analysis:-

Lexical Analysis or Scanners reads the source program one character at a time, carving the source program into a sequence of character units called tokens.

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

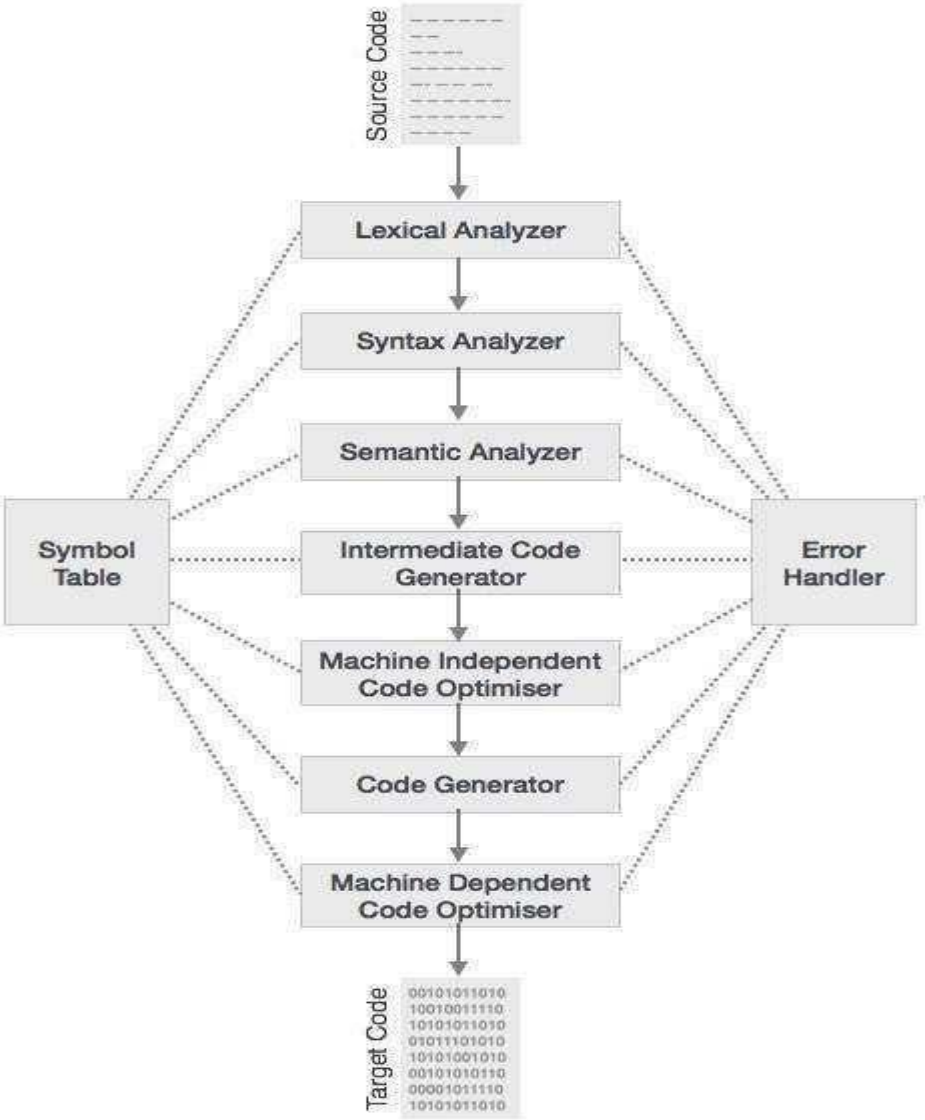


Figure 1.6: Phases of Compiler

Semantic Analysis

There is more to a front end than simply syntax. The compiler needs semantic information, e.g., the types (integer, real, pointer to array of integers, etc) of the objects involved. This enables checking for semantic errors and inserting type conversion where necessary.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Symbol-Table Management

The symbol table stores information about program variables that will be used across phases. Typically, this includes type information and storage location.

A possible point of confusion: the storage location does **not** give the location where the compiler has stored the variable. Instead, it gives the location where the compiled program will store the variable.

Error Handlers

It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens,which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

5. LEXICAL ANALYSIS

To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language. o Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

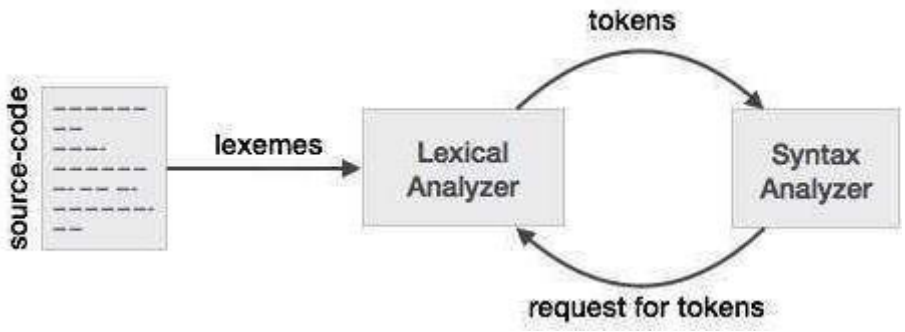


Figure 1.7: Lexical Analysis phase

ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon. LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

Token

Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern

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.

Lexeme

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

5.1 LEXICAL ERRORS

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognize a lexeme as a valid token for you lexer? Syntax errors, on the other side, will be thrown by your scanner when a given set of already recognized valid tokens don't match any of the right sides of your grammar rules. Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

a. Delete one character from the remaining input.
b. Insert a missing character in to the remaining input.
c. Replace a character by another character.
d. Transpose two adjacent characters.

5.2 LEXICAL ANALYSIS: INPUT BUFFER

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. 1.8 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 look ahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see:

DECLA‘E (A‘G1, A‘G2… A‘G *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 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

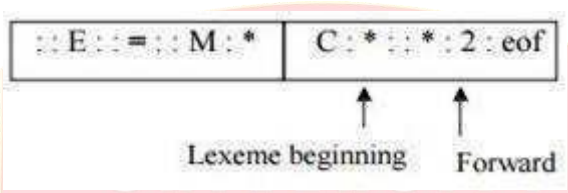


Figure 1.8: An input buffer in two halves

Each buffer is of the same size N, and N is usually the number of characters on one 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:

- Pointer lexeme beginning marks the beginning of the current lexeme, whose extent we are attempting to determine.
- 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

Figure. 1.9: Sentinels at end of each buffer half

lexeme beginning forward

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

5.3. SPECIFICATION & RECOGNITION OF TOKEN

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of

alphabets, e.g., the length of the string Compiler is 8.

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)

Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifies	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Table1.1: Lex Symbol Description

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Regular Expressions in Compiler design

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

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

languages are easy to understand and have efficient implementation.

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

Operations

The various operations on languages are:

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

Notations

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

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r) Precedence and Associativity
- *, concatenation (.), and | (pipe sign) are left associative
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression
If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate { e, x, xx, xxx, xxxx, … }
- x+ means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.
[A-Z] is all upper-case alphabets of English language.
[0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions
Letter = [a – z] or [A – Z]

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

Representing language tokens using regular expressions
Decimal = $(sign)^{?}(digit)^{+}$

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Compiler-construction tools

Originally, compilers were written “from scratch”, but now the situation is quite different. A

are available to ease the burden.

We will study tools that generate scanners and parsers. This will involve us in some theory, regular expressions for scanners and various grammars for parsers. These techniques are fairly successful. One drawback can be that they do not execute as fast as “hand-crafted” scanners and parsers.

We will also see tools for syntax-directed translation and automatic code generation. The automation in these cases is not as complete.

Finally, there is the large area of optimization. This is not automated; however, a basic component of optimization is “data-flow analysis” (how values are transmitted between parts of a program) and there are tools to help with this task.

5.4.LEX

Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
Lex tool itself is a lex compiler.

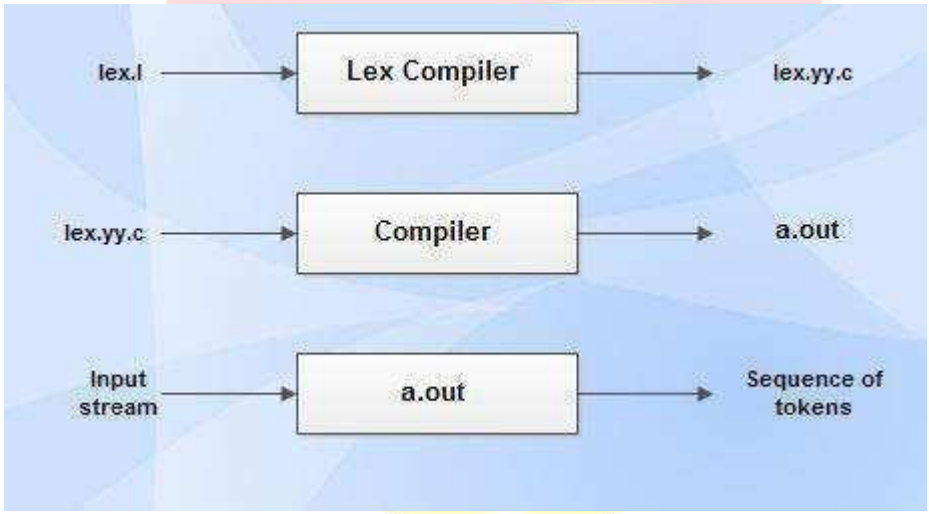


Figure 1.10 LEX Diagram

• ***lex.l*** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c.*

• ***lex.yy.c*** is compiled by the C compiler to a file called *a.out.*

• The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.

• ***yylval*** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

• The attribute value can be numeric code, pointer to symbol table or nothing.

• Another tool for lexical analyzer generation is Flex.

Structure of Lex Programs

Lex program will be in following form
declarations

%%

translation rules

%%

auxiliary functions

Declarations This section includes declaration of variables, constants and regular definitions.

Translation rules It contains regular expressions and code segments.
Form : Pattern {Action}

Pattern is a regular expression or regular definition.
Action refers to segments of code.

Auxiliary functions This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.

Lexical analyzer produced by Lex starts its process by reading one character at a time until a valid match for a pattern is found.

Once a match is found, the associated action takes place to produce token.
The token is then given to parser for further processing.

Conflict Resolution in Lex

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

- Always prefer a longer prefix than a shorter prefix.
- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

Lookahead Operator

- Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
- Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
- At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.

(eg.) In some languages keywords are not reserved. So the statements
IF (I, J) = 5 and IF(condition) THEN

results in conflict whether to produce IF as an array name or a keyword. To resolve this the lex rule for keyword IF can be written as,

IF/\ (.* \) {letter}

Design of Lexical Analyzer

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

Structure of Generated Analyzer

Architecture of lexical analyzer generated by lex is given in Figure:

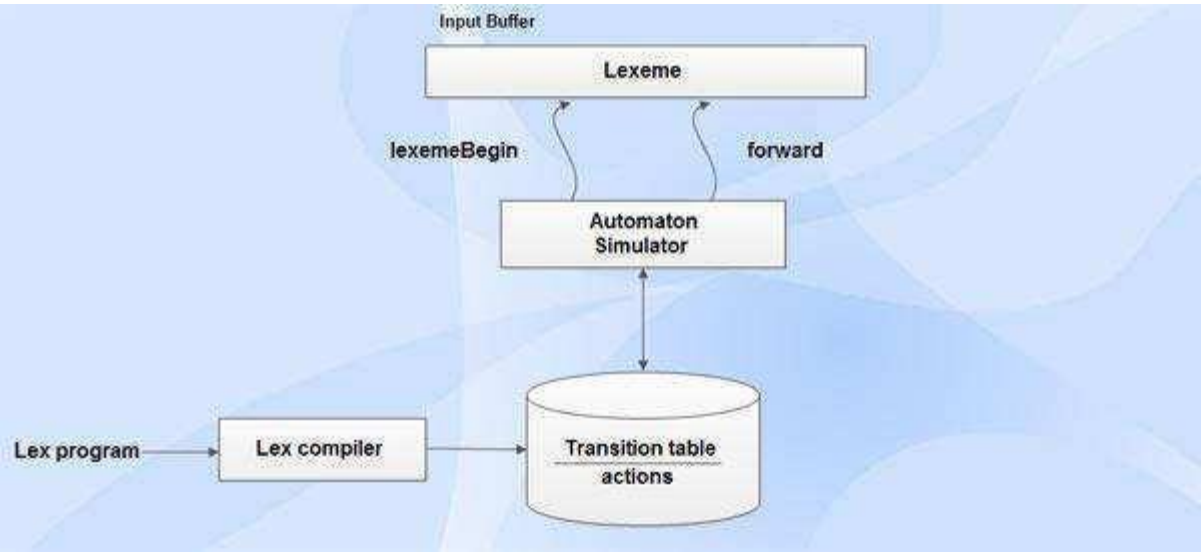


Figure 1.11 Architecture of lexical analyzer

Lexical analyzer program includes:

- Program to simulate automata.
- Components created from lex program by lex itself which are listed as follows:
    - A transition table for automaton.
    - Functions that are passed directly through lex to the output.
    - Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

CS7002 Compiler Design

Unit-II

1 INTRODUCTION OF SYNTAX ANALYSIS

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labeled. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called parser generation.

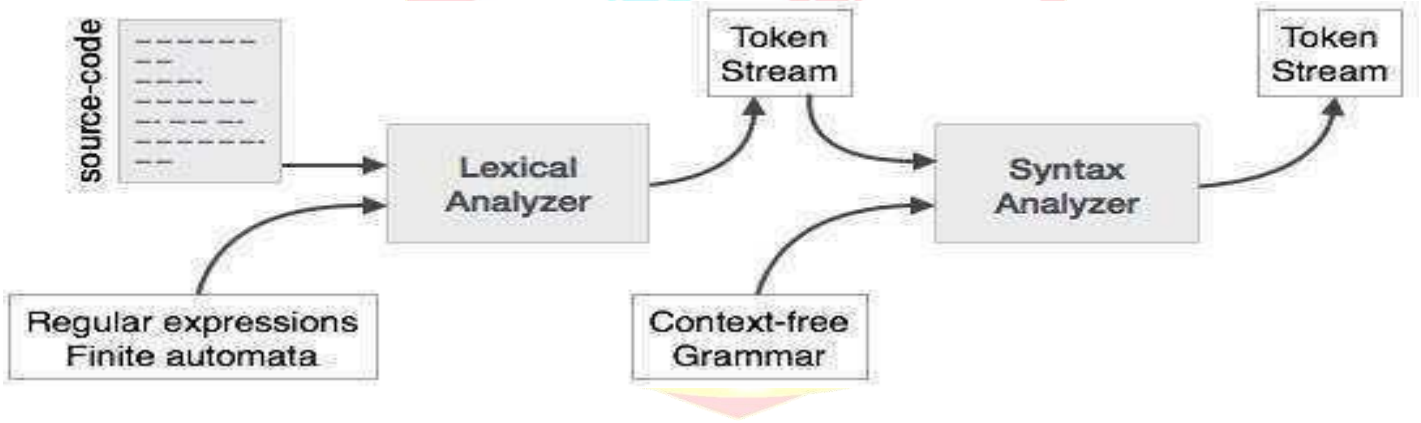


Figure 1: Syntax Analyzer

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.

This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

1.1 CONTEXT FREE GRAMMAR

Definition – A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P,S),** where

N is a set of non-terminal symbols.

T is a set of terminals where **N ∩ T = NULL.**

P is a set of rules, **P: N → (N** U **T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.

S is the start symbol.

Example

The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.

The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε

Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

Root vertex − Must be labeled by the start symbol.

Vertex − Labeled by a non-terminal symbol.

Leaves − Labeled by a terminal symbol or ε.

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows −

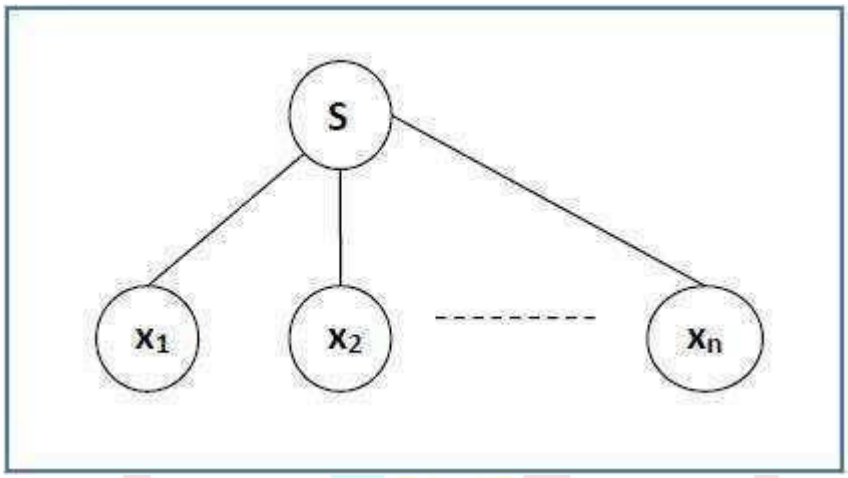


Figure 2: Derivation tree

There are two different approaches to draw a derivation tree −

Top-down Approach –

Starts with the starting symbol **S**

Goes down to tree leaves using productions

Bottom-up Approach –

Starts from tree leaves

Proceeds upward to the root which is the starting symbol **S**

Leftmost and Rightmost Derivation of a String

Leftmost derivation − A leftmost derivation is obtained by applying production to the leftmost variable in each step.

Rightmost derivation − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** may be −

X → X+X → a+X → a + X*X → a+a*X → a+a*a

The stepwise derivation of the above string is shown as below −

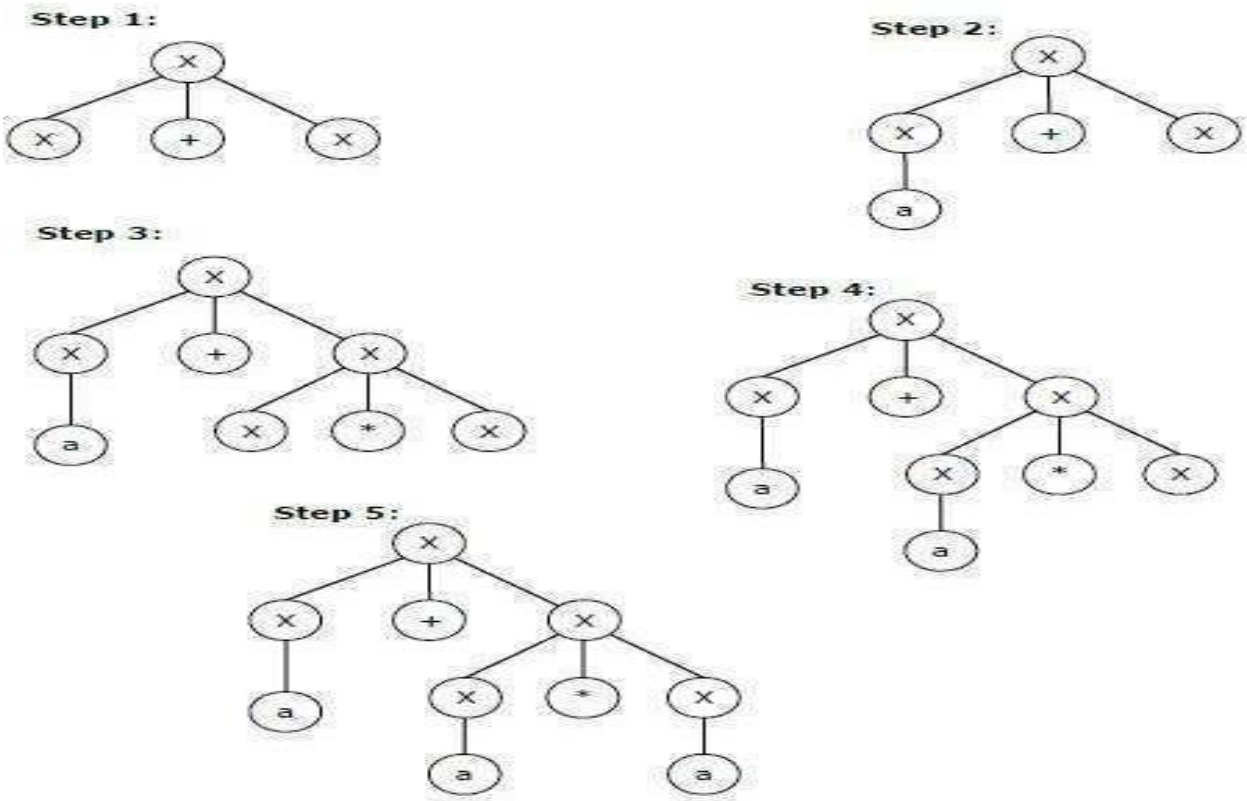


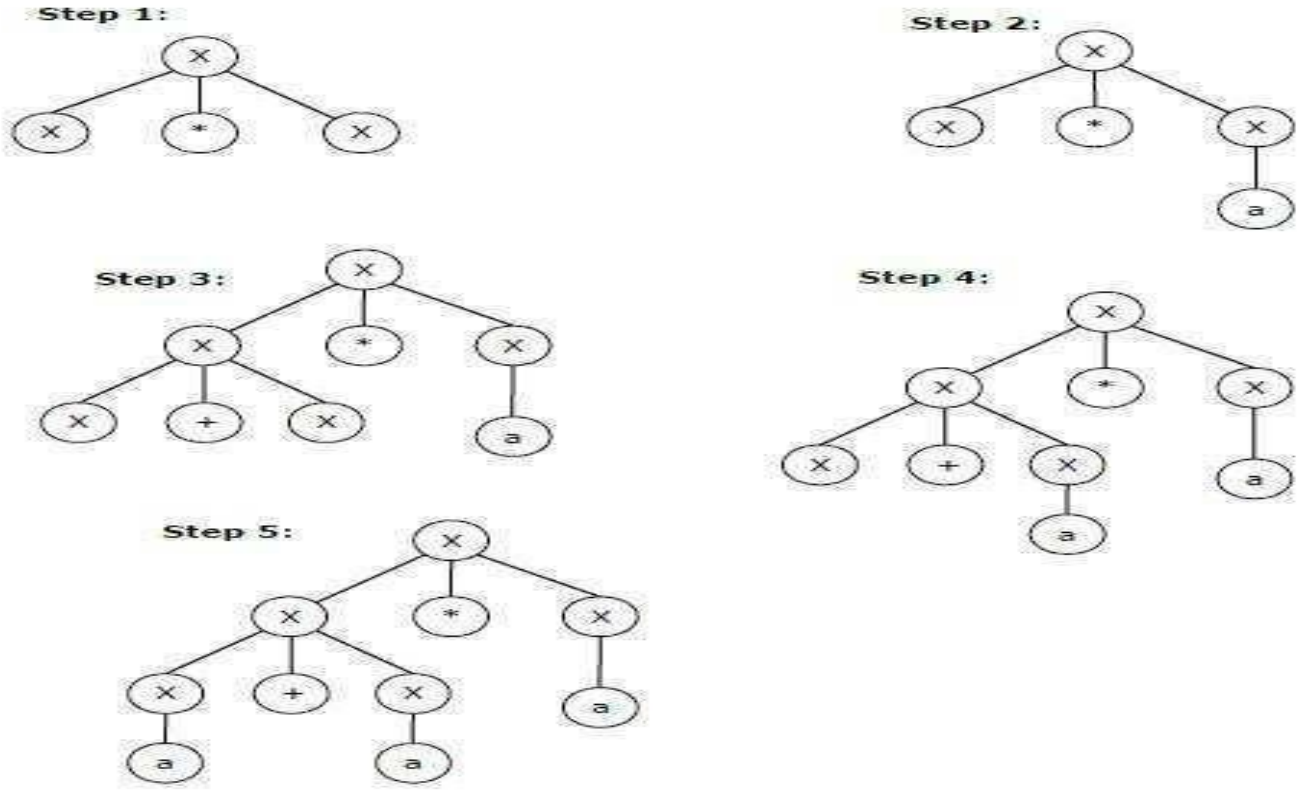
Figure 1:Solution of Bottom up Approach

Figure 2:Solution of Bottom up Approach

The rightmost derivation for the above string **"a+a*a"** may be –

X → X*X → X*a → X+X*a → X+a*a → a+a*a

.



2. PARSING

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.

2.1 TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

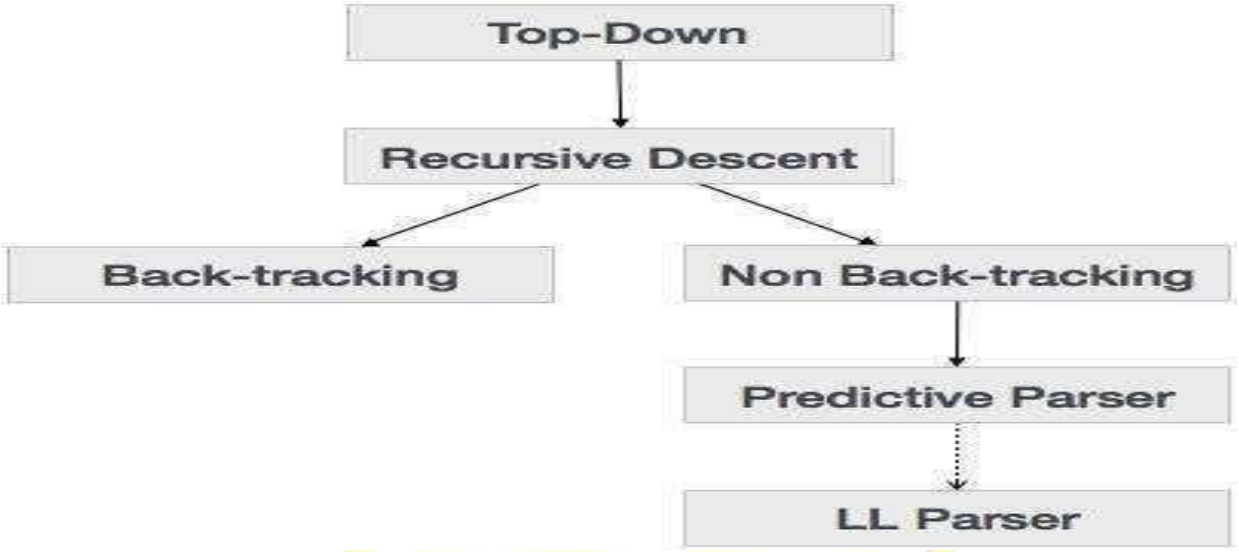


Figure 3:Top down parsing

2.2 BRUTE-FORCE APPROACH

A top-down parse moves from the goal symbol to a string of terminal symbols. In the terminology of trees, this is moving from the root of the tree to a set of the leaves in the syntax tree for a program. In using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached. In the worst possible case, that of trying to parse a string which is not in the language, all possible combinations are attempted before the failure to parse is recognized.

Top-down parsing with full backup is a " brute-force" method of parsing. In general terms, this method operates as follows:

1. Given a particular non-terminal that is to be expanded, the first production for this non-terminal is applied.

2. Then, within this newly expanded string, the next (leftmost) non-terminal is selected for expansion and its first production is applied.

3. This process (step 2) of applying productions is repeated for all subsequent non-terminals that are selected until such time as the process cannot or should not be continued. This termination (if it ever occurs) may be due to two causes. First, no more non-terminals may be present, in which case the string has been successfully parsed. Second, it may result from an incorrect expansion which would be indicated by the production of a substring of terminals which does not match the appropriate segment of the source string. In the case of such

an incorrect expansion, the process is "backed up" by undoing the most recently applied production. Instead of using the particular expansion that caused the error, the next production of this non-terminal is used as the next expansion, and then the process of production application continues as before.

If, on the other hand, no further productions are available to replace the production that caused the error, this error-causing expansion is replaced by the non-terminal itself, and the process is backed up again to undo the next most recently applied production. This backing up continues either until we are able to resume normal application of productions to selected non-terminals or until we have backed up to the goal symbol and there are no further productions to be tried. In the latter case, the given string must be unappeasable because it is not part of the language determined by this particular grammar.

As an example of this brute-force parsing technique, let us consider the simple grammar

$$S \rightarrow aAd|aB \qquad A \rightarrow b|c \qquad B \rightarrow ccd|ddc$$

where S is the goal or start symbol. Figure 6-1 illustrates the working of this brute-force parsing technique by showing the sequence of syntax trees generated during the parse of the string 'accd'.

2.3 RECURSIVE DECENT 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 look ahead. 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.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form . A syntax of the form defines zero or more occurrences of the form .

A usual implementation of an LL(1) parser is.

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```
proc syntax Analysis()
begin
initialize(); // initialize global data and structures
nextToken(); // get the lookahead token
program(); // parser routine that implements the start symbol
end;
```

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 position. This is called backtracking.

Step4:

Now try the second alternative for A.

Now we can halt and announce the successful completion of parsing.
Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
Hence, elimination of left-recursion must be done before parsing.

2.4 PREDICTIVE PARSING

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

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

Non-recursive predictive parser

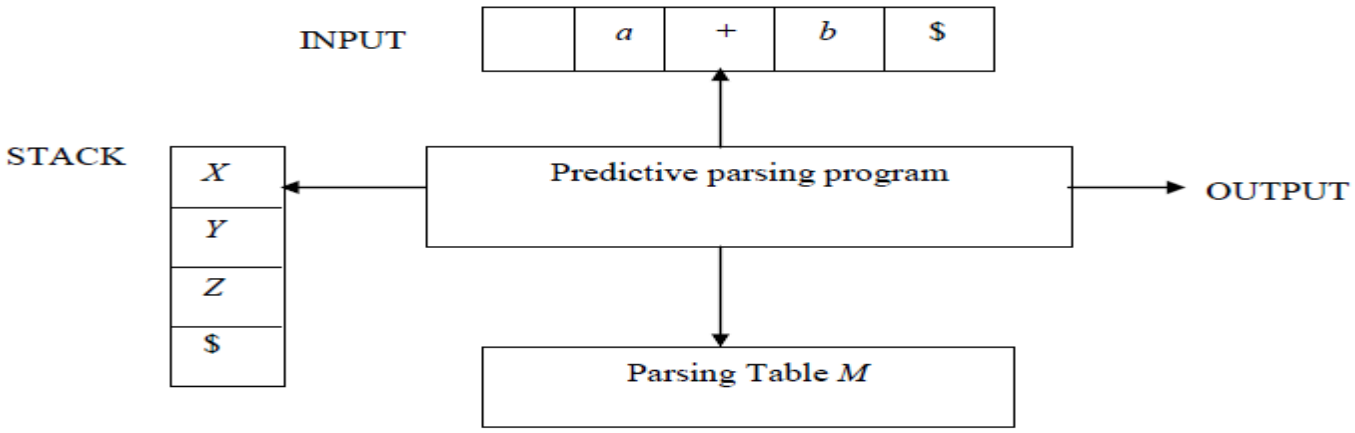


Figure 4: Predictive parsing

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

Input buffer:

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

Stack:

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

Parsing table:

It is a two-dimensional array M[A, a], where ‘A’ is a non-terminal and ‘a’ is a terminal. Predictive parsing program. The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

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

- If X = a ≠ $, the parser pops X off the stack and advances the input pointer tothe next input symbol.
- If X is a non-terminal , the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
- If M[X, a] = {X → UVW},the parser replaces X on top of the stack by UVW
- If M[X, a] = error, the parser calls an error recovery routine.

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 FI'ST(X) if for some i, a is in FIRST(Yi), and ε is in all of FI'ST(Y1),…,FI'ST(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).

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}

FI'ST(E') ={+ , ε }

FIRST(T) = { (, id}

FI'ST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

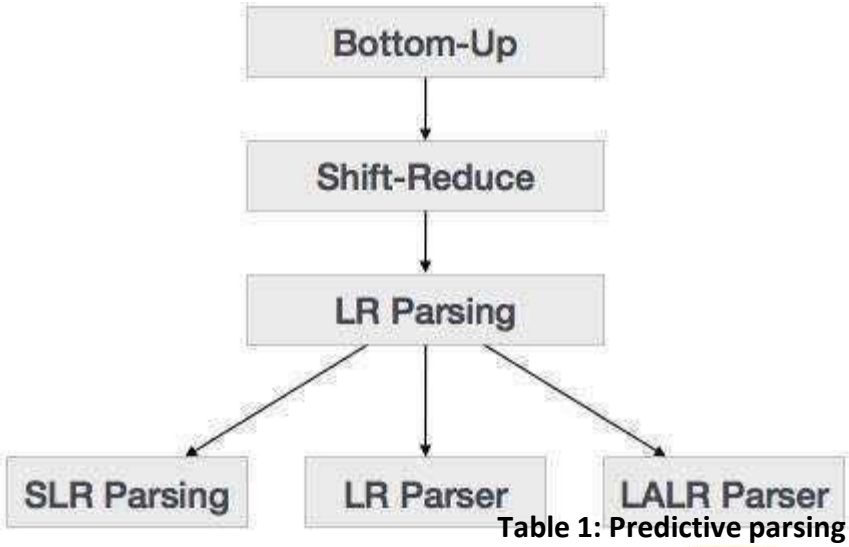


Table 1: Predictive parsing table

3. 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.

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)		

Figure 5: Bottum up parsing

3.1 SHIFT-REDUCE PARSING

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The sentence to be recognized is abbcde

REDUCTION (LEFT MOST)	RIGHTMOST DERIVATION
abcde (A->b)	S->aABe
aAcde (A->Avc)	->aAde
aAde(B->d)	->aAbcde
aABe(S->aABe)	->abbcde

Table 2:Shift Reduce Parser

3.2 OPERATOR PRECEDENCE PARSING

Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars. Operator grammars have the property that no production right side is empty or has two adjacent non-terminals. This property enables the implementation of efficient operator-precedence parsers. These parser rely on the following three precedence relations:

Relation Meaning

a <· b a yields precedence to b

a =· b a has the same precedence as b

a ·> b a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

Example: The input string:

id1 + id2 * id3

after inserting precedence relations becomes

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows to identify handles as follows:

scan the string from left until seeing ·>

scan backwards the string from right to left until seeing <·

everything between the two relations <· and ·> forms the handle

	id	+	*	$
Id		.>	.>	.>
+	<.	.>	<.	.>
*****	<.	.>	.>	.>
$	<.	<.	<.	.>

Table 3: Operator precedence parsing

3.3 LR PARSING INTRODUCTION

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in

reverse.

WHY LR PARSING:

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 non-backtracking 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 subset of the class of grammars that can be parsed with predictive parsers.

An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

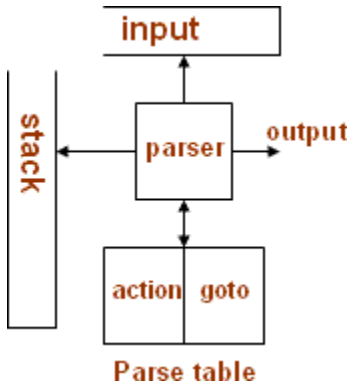


Figure 6 :LR Parser

The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

There are three widely used algorithms available for constructing an LR parser:

SLR(1) – Simple LR Parser:

- Works on smallest class of grammar
- Few number of states, hence very small table
- Simple and fast construction

LR(1) – LR Parser:

- Works on complete set of LR(1) Grammar
- Generates large table and large number of states
- Slow construction

LALR(1) – Look-Ahead LR Parser:

- Works on intermediate size of grammar
- Number of states are same as in SLR(1)

3.4 SLR(1) – SIMPLE LR PARSER:

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of "reducing" (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is LR (scanning from Left to right and using Right-most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. The symbol $ is used to mark the bottom of the stack and also the right-end of the input.

Notation ally, the top of the stack is identified through a separator symbol |, and the input string to be parsed

appears on the right side of |. The stack content appears on the left of |.

For example, an intermediate stage of parsing can be shown as follows:

$id1 | + id2 * id3$ …. (1)

Here “$id1” is in the stack, while the input yet to be seen is “+ id2 * id3$*

In shift-reduce parser, there are two fundamental operations: shift and reduce.

Shift operation: The next input symbol is shifted onto the top of the stack.

After shifting + into the stack, the above state captured in (1) would change into:

$id1 + | id2 * id3$

Reduce operation: Replaces a set of grammar symbols on the top of the stack with the LHS of a production rule.

After reducing id1 using E → id, the state (1) would change into:

$E | + id2 * id3$

In every example, we introduce a new start symbol (S'), and define a new production from this new start symbol to the original start symbol of the grammar.

Consider the following grammar (putting an explicit end-marker $ at the end of the first production:

(1) S' → S$

(2) S → Sa

(3) S → b

For this example, the NFA for the stack can be shown as follows:

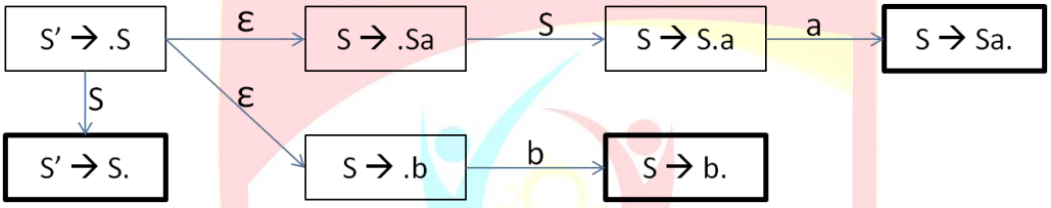


Figure 7: Shift Operation

After doing ε-closure, the resulting DFA is as follows:

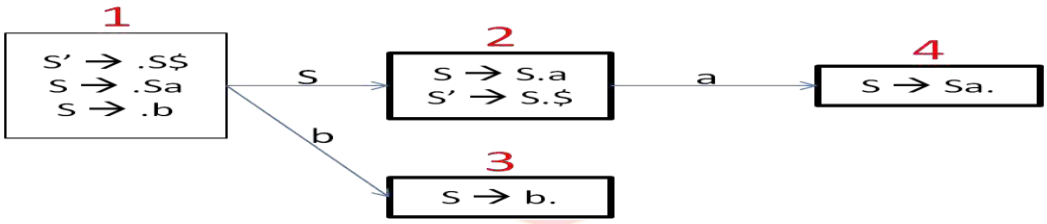


Figure 8: Reduce Operation

The states of DFA are also called "Canonical Collection of Items". Using the above notation, the ACTION-GOTO table can be shown as follows:

State	a	b	$	S
1		s3		g2
2	s4, r1	r1	r1, a	
3	r3	r3	r3	
4	r2	r2	r2	

Table 4: Simple LR Parser

3.5. CANONICAL LR PARSING

- Carry extra information in the state so that wrong reductions by A $\longrightarrow \alpha$ will be ruled out .
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol).
- The general form of the item becomes [A $\longrightarrow \alpha$. □, a] which is called LR(1) item.
- Item [A $\longrightarrow \alpha$., a] calls for reduction only if next input is a. The set of symbols

Canonical LR parsers solve this problem by storing extra information in the state itself. The problem we have with SLR parsers is because it does reduction even for those symbols of follow(A) for which it is invalid. So LR items are redefined to store 1 terminal (look ahead symbol) along with state and thus, the items now are LR(1) items.

An LR(1) item has the form : [A → a . □, a] and reduction is done using this rule only if input is 'a'. Clearly the symbols a's form a subset of follow(A).

Closure(I)

repeat

for each item [A → α .B □, a] in I
for each production B → γ in G'
and for each terminal b in First(□a)
add item [B → . γ , b] to I

until no more additions to I

To find closure for Canonical LR parsers:

Repeat

for each item [A → α .B □, a] in I
for each production B → γ in G'
and for each terminal b in First(□a)
add item [B → . γ , b] to I

until no more items can be added to I

Example

Consider the following grammar
S' → S

S → CC

C → cC | d

Compute closure(I) where I={[S' → .S, $]}

S' → .S,	$
S → .CC,	$
C .cC,	c
C .cC,	d
C → .d,	c
C → .d,	d

For the given grammar:
S ' → S

S → CC

C → cC | d

I : closure([S ' ⟶S, $])

S ' .S	$	as first(e $) = {$}
S .CC	$	as first(C$) = first(C) = {c, d}
C .cC	c	as first(Cc) = first(C) = {c, d}

C	**.cC**	d	as first(Cd) = first(C) = {c, d}
C	**.d**	c	as first(e c) = {c}
C	**.d**	d	as first(e d) = {d}

Table 5: Canonical LR Parsing

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0 :	S ' → .S,	$
	S → .CC,	$
	C → .cC,	c /d
	C → .d,	c /d
I_1 :	goto(I_0 ,S)	
	S ' S.,	$
I_2 :	goto(I 0 ,C)	
	S C.C,	$
	C .cC,	$
	C .d,	$
I_3 :	goto(I 0 ,c)	
	C c.C,	c/d
	C .cC,	c/d
	C .d,	c/d
	goto(I 0 ,d)	
I_4 :	C d.,	c/d
	goto(I 2 ,C)	
I_5 :		
	S CC.,	$
I_6 :	goto(I 2 ,c)	
	C c.C,	$
	C .cC,	$
	C .d,	$
I_7 :	goto(I 2 ,d)	
	C d.,	$

To construct sets of LR(1) items for the grammar given in previous slide we will begin by computing closure of {[S ´ → .S, $]}.

To compute closure we use the function given previously.

In this case α = ε , B = S, ß =ε and a=$. So add item [S ⟶ .CC, $].
Now first(C$) contains c and d so we add following items

we have A=S, α = ε , B = C, ß=C and a=$

Now first(C$) = first(C) contains c and d

so we add the items [C ⟶ .cC, c], [C ⟶ .cC, d], [C ⟶ .dC, c], [C ⟶ .dC, d].
Similarly we use this function and construct all sets of LR(1) items.

Construction of Canonical LR parse table

. Construct C={I_0 , . , I_n } the sets of LR(1) items.

. If [A ⟶ α .a □, b] is in I_i and goto(I_i , a)=I_j then action[i,a]=shift j

. If [A ⟶ α ., a] is in I_i then action[i,a] reduce A ⟶ α

. If [S ' ⟶ S., $] is in I_i then action[i,$] = accept

. If goto(I i , A) = I_j then goto[i,A] = j for all non

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ~~S.,~~$] ε I_1 .

Parse table

Table 6: Parser Table

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ~~S.,~~$] ε I_1

LALR Parse table

Look Ahead LR parsers

Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
I_4 : C ⟶ d. , c/d I_7 : C ⟶ d., $

Replace I_4 and I_7 by a new state I_{47} consisting of (C ⟶ d., c/d/$)
Similarly I_3 & I_6 and I_8 & I_9 form pairs

Merge LR(1) items having the same core

We will combine Ii and Ij to construct new Iij if Ii and Ij have the same core and the difference is only in look ahead symbols. After merging the sets of LR(1) items for previous example will be as follows:

I_0 : S' ⟶ S $
S ⟶ .CC $

C ⟶ .cC c/d

C ⟶ .d c/d

I_1 : goto(I 0 ,S)
S' ⟶ S. $

I_2 : goto(I 0 ,C)

S ⟶C.C $

C⟶.cC $

C ⟶.d $

I_{36} : goto(I 2 ,c)
C ⟶c.C c/d/$

C ⟶.cC c/d/$

C ⟶ .d c/d/$

I_4 : goto(I 0 ,d)
C d. €/d

I_5 : goto(I 2 ,C)
S CC. $

I_7 : goto(I 2 ,d)
C ⟶d. $

I_{89} : goto(I 36 ,C)

C ⟶cC. c/d/$

Construct LALR parse table

Construct C={I_0 , .. ,I_n } set of LR(1) items

For each core present in LR(1) items find all sets having the same core and replace these sets by their union
Let C' = {J_0 , .. .,J_m } be the resulting set of items

Construct action table as was done earlier
Let J = I_1 U I_2 .. .U I_k

since I_1 , I_2 .. ., I_k have same core, goto(J,X) will have he same core
Let K=goto(I_1 ,X) U goto(I_2 ,X) .. goto(I_k ,X) the goto(J,X)=K

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

LALR parse table

State	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Table 7: LALR parse table .

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

Notes on LALR parse table

Modified parser behaves as original except that it will reduce C ⟶d on inputs like ccd. The error will

eventually be caught before any more symbols are shifted.

In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.

4 PARSER GENERATORS

Some common parser generators
YACC: **Y**et **A**nother **C**ompiler **C**ompiler

Bison: GNU Software

ANTLR: **AN** other **T**ool for **L**anguage **R**ecognition

Yacc/Bison source program specification (accept LALR grammars)

declaration

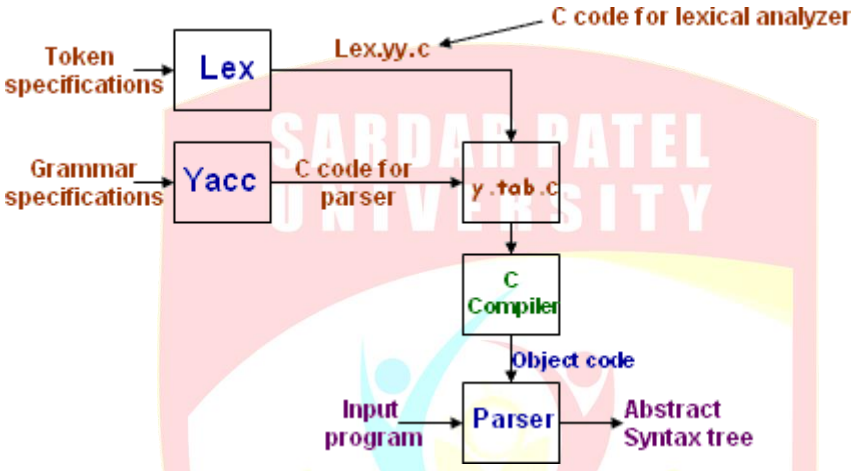
%%

translation rules

%%

supporting C routines

Yacc and Lex schema



YACC Manual

Figure 9: YACC

5. SYNTAX DIRECTED DEFINITION

Specifies the values of attributes by associating semantic rules with the grammar productions.

It is a context free grammar with attributes and rules together which are associated with grammar symbols and productions respectively.

The process of syntax directed translation is two-fold:

• Construction of syntax tree and

• Computing values of attributes at each node by visiting the nodes of syntax tree.

Semantic actions

Semantic actions are fragments of code which are embedded within production bodies by syntax directed translation.

They are usually enclosed within curly braces ({ }).

It can occur anywhere in a production but usually at the end of production.

E---> E_1 + *T {print* '+'}

Types of translation

- **L-attributed translation**

It performs translation during parsing itself.
No need of explicit tree construction.

L represents 'left to right'.

- **S-attributed translation**

It is performed in connection with bottom up parsing.
'S' represents synthesized.

Types of attributes

- **Inherited attributes**

It is defined by the semantic rule associated with the production at the parent of node.
Attributes values are confined to the parent of node, its siblings and by itself.

The non-terminal concerned must be in the body of the production.

- **Synthesized attributes**

It is defined by the semantic rule associated with the production at the node.
Attributes values are confined to the children of node and by itself.

The non terminal concerned must be in the head of production.

Terminals have synthesized attributes which are the lexical values (denoted by lex val) generated by the lexical analyzer.

Syntax directed definition of simple desk calculator

Production	Semantic rules
L ---> E_n	*L.val = E.val*
E ---> E_1+ T	*E.val = E_1.val+ T.val*
E ---> T	*E.val = T.val*
T---> T_1*F	*T.val = T_i.val x F.val*
T ---> F	*T.val = F.val*
F ---> (E)	*F.val = E.val*
F ---> digit	*F.val = digit.lexval*

Table 8: Syntax directed

Syntax directed definition inherited attributes

Production	Semantic Rules
D --->TL	*L.inh = T.type*
T ---> *int*	*T.type* =integer
T ---> *float*	*T.type = float*
L ---> L_1, **id**	L_1.inh = *L.inh* *addType (id.entry, Linh)*
L ---> **id**	*addType (id.entry, L.inh)*

Table 9: Syntax directed

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

Types of Syntax Directed Definitions

5.1 S-ATTRIBUTED DEFINITIONS

Syntax directed definition that involves only synthesized attributes is called S-attributed. Attribute values for the non-terminal at the head is computed from the attribute values of the symbols at the body of the production.

The attributes of a S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree. i.e., by performing post order traversal of the parse tree and evaluating the attributes at a node when the traversal leaves that node for the last time.

Production	Semantic rules
L ---> E_n	*L.val = E.val*
E ---> E_1+ T	*E.val = E_1.val+ T.val*
E ---> T	*E.val = T.val*
T---> T_1*F	*T.val = T_i.val x F.val*
T ---> F	*T.val = F.val*
F ---> (E)	*F.val = E.val*
F ---> digit	*F.val = digit.lexval*

Table 10: S-attributed Definitions

L-ATTRIBUTED DEFINITIONS

The syntax directed definition in which the edges of dependency graph for the attributes in production body, can go from left to right and not from right to left is called L-attributed definitions. Attributes of L-attributed definitions may either be synthesized or inherited.

If the attributes are inherited, it must be computed from:

• Inherited attribute associated with the production head.

• Either by inherited or synthesized attribute associated with the production located to the left of the attribute which is being computed.

• Either by inherited or synthesized attribute associated with the attribute under consideration in such a way that no cycles can be formed by it in dependency graph.

Production	Semantic Rules
T ---> FT'	*T '.inh = F.val*
T ' ---> *FT_1'	*T'$_1$.inh =T'.inh x F.val*

Table 11: L-attributed Definitions

In production 1**,** the inherited attribute T' is computed from the value of F which is to its left. In production 2, the inherited attributed Tl' is computed from T'. *inh* associated with its head and the value of F which appears to its left in the production. i.e., for computing inherited attribute it must either use *from the above* or *from the left* information of SDD

Construction of Syntax Trees

SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.

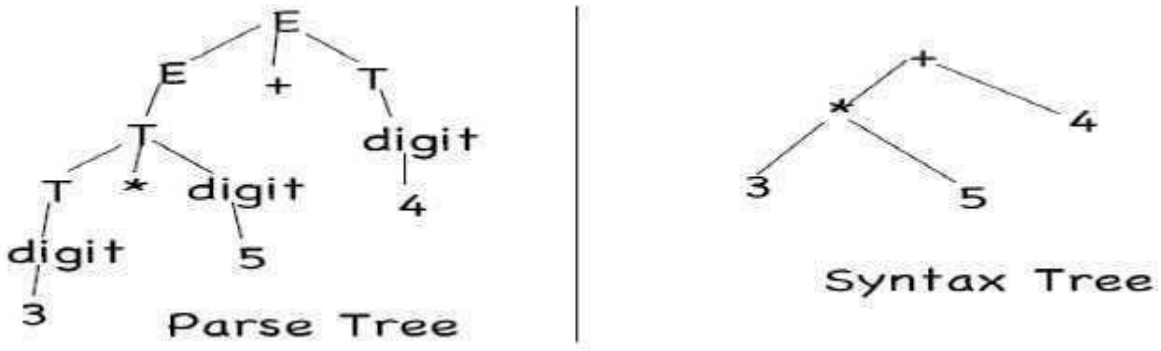


Figure 10: Syntax Trees

• Syntax trees are useful for representing programming language constructs like expressions and statements.

•They help compiler design by decoupling parsing from translation.

• Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

• e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2

• Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:

——————————— If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf (op, val)

——————————— If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node (op, c1, c2,...,ck) .

Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All non-terminals have one synthesized attribute node, which represents a node of the syntax

tree.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('–', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, **id**.*entry*)
6) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, **num**.*val*)

Syntax tree for a-4+c using the above SDD is shown below.

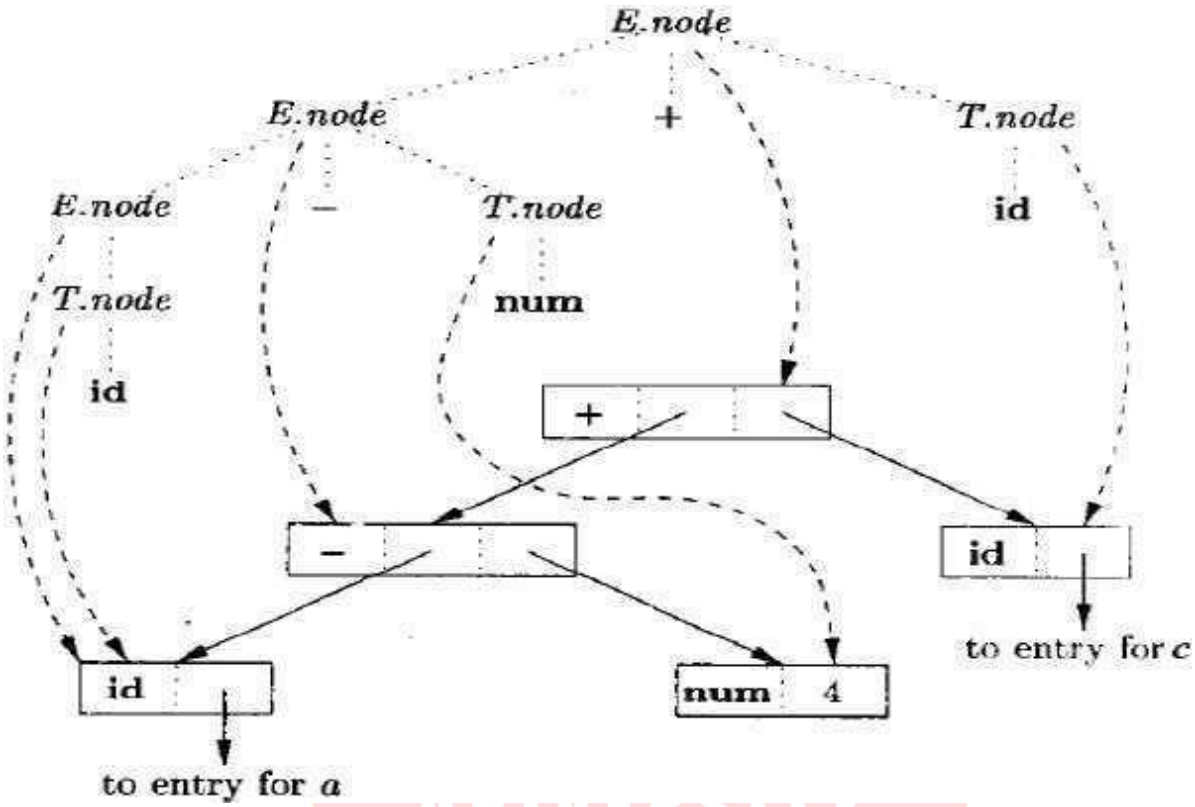


Figure 11:L-Attribute

5.2 BOTTOM-UP EVALUATION OF SDT

Given an SDT, we can evaluate attributes even during bottom-up parsing. To carry out the semantic actions, parser stack is extended with semantic stack. The set of actions performed on semantic stack are mirror reflections of parser stack. Maintaining semantic stack is very easy.

During shift action, the parser pushes grammar symbols on the parser stack, whereas attributes are pushed on to semantic stack.

During reduce action, parser reduces handle, whereas in semantic stack, attributes are evaluated by the corresponding semantic action and are replaced by the result.

For example, consider the SDT

A → X Y)	{A · a := f(X · x, Y · y, Z · z);}

Strictly speaking, attributes are evaluated as follows

A → X Y)	{val[ntop] := f(val[top - 2], val[top - 1], val[top]);}

Evaluation of Synthesized Attributes

- Whenever a token is shifted onto the stack, then it is shifted along with its attribute value placed in val[top].
- Just before a reduction takes place the semantic rules are executed.
- If there is a synthesized attribute with the left-hand side non-terminal, then carrying out semantic rules will place the value of the synthesized attribute in val[ntop].

Let us understand this with an example:

	E → E_1 "+" T	{val[ntop] := val[top-2] + val[top];}
	E → T	{val[top] := val[top];}
	T → T_1 "*" F	{val[ntop] := val[top-2] * val[top];}
	T → F	{val[top] := val[top];}
	F → "(" E ")"	{val[ntop] := val[top-1];}
	F → **num**	{val[top] := num.lvalue;}

Table 12:

Figure shows the result of shift action. Now after performing reduce action by E → E * T resulting stack is as shown in figure.

Along with bottom-up parsing, this is how attributes can be evaluated using shift action/reduce action.

5.3 L-ATTRIBUTED DEFINITION

It allows both types, that is, synthesized as well as inherited. But if at all an inherited attribute is present, there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.

For example, consider the rule

A → XY)PQ assume that there is an inherited attribute, “i” is present with each non-terminal.

)•I = f(A•i|X•i|Y•i) but)•i = f(P•i|Q•i) is wrong as they are right siblings.

Semantic actions can be placed anywhere on the right hand side.

Attributes are generally evaluated by traversing the parse tree depth first and left to right. It is possible to rewrite any L-attributes to S-attributed definition.

L-attributed definition for converting infix to post fix form.

E → TE”

E” → +T #1 E” | ε

T → F T”

T” → * F #2 T” | ε

F → id #3

where #1 corresponds to printing “+” operator, #2 corresponds to printing “*,” and # 3 corresponds to printing id.val.

Look at the above SDT; there are no attributes, it is L-attributed definition as the semantic actions are in between grammar symbols. This is a simple example of L-attributed definition. Let us analyze this L-attributed definition and understand how to evaluate attributes with depth first left to right traversal. Take the parse tree for the input string “a + b*c” and perform *Depth first left to right traversal, i.e. at each node traverse the left sub tree depth wise completely then right sub tree completely.*

Follow the traversal in . During the traversal whenever any dummy non-terminal is seen, carry out the translation.

Converting L-Attributed to S-Attributed Definition

Now that we understand that S-attributed is simple compared to L-attributed definition, let us see how to convert an L-attributed to an equivalent S-attributed.

Consider an L-attributed with semantic actions in between the grammar symbols. Suppose we have an L-attributed as follows:

S → A {} B

How to convert it to an equivalent S-attributed definition? It is very simple!!
Replace actions by non-terminal as follows:

S → A M B

M → ε {}

Convert the following L-attributed definition to equivalent S-attributed definition.

	E → TE”	
	E” → +T	#1 E” | ε

	T → F T”	
	T” → *F	#2 T” | ε
	F → id	#3

Table 13:Solution

Solution:

Replace dummy non-terminals that is, actions by non-terminals.

	E → TE”	
	E” → +T A E” | ε	
	A →	{ print(“+”);}
	T → F T”	
	T” → *F **B** T” |ε	
	B →	{ print(“*”);}
	F → id	{ print(“id”);}

Table 14:Solution

6. YACC

YACC—Yet Another Compiler Compiler—is a tool for construction of automatic LALR parser generator.
Using Yacc

Yacc specifications are prepared in a file with extension “.y” For example, “test.y.” Then run this file with the Yacc command as “$yacc test.y.” This translates yacc specifications into C-specifications under the default file mane “y.tab.c,” where all the translations are under a function name called yyparse(); Now compile “y.tab.c” with C-compiler and test the program. The steps to be performed are given below:

Commands to execute

$yacc test.y

This gives an output “y.tab.c,” which is a parser in c under a function name yyparse().

With –v option ($yacc -v test.y), produces file y.output, which gives complete information about the LALR parser like DFA states, conflicts, number of terminals used, etc.

$cc y.tab.c

$./a.out

Preparing the Yacc specification file

Every yacc specification file consists of three sections: the *declarations, grammar rules*, and *supporting* subroutines. The sections are separated by double percent “%%” marks.

declarations

% %

Translation rules

% %

supporting *subroutines*

The declaration section is optional. In case if there are no supporting subroutines, then the second %% can also be skipped; thus, the smallest legal Yacc specification is

% %

Translation rules

Declarations section

Declaration part contains two types of declarations—Yacc declarations or C-declarations. To distinguish between the two, C-declarations are enclosed within %{ and %}. Here we can have C-declarations like global variable declarations (int x=l;), header files (#include....), and macro definitions(#define...). This may be used for defining subroutines in the last section or action part in grammar rules.

Yacc declarations are nothing but tokens or terminals. We can define tokens by %token in the declaration part.

For example, “num” is a terminal in grammar, then we define

% token num in the declaration part. In grammar rules, symbols within single quotes are also taken as terminals.

We can define the precedence and associativity of the tokens in the declarations section. This is done using %left, %right, followed by a list of tokens. The tokens defined on the same line will have the same precedence and associativity; the lines are listed in the order of increasing precedence. Thus,

%left ‘+’ ‘-’

%left ‘*’ ‘/’

are used to define the associativity and precedence of the four basic arithmetic operators ‘+’,‘-’,‘/’,‘*’. Operators ‘*’ and ‘/’ have higher precedence than ‘+’ and both are left associative. The keyword %left is used to define left associativity and %right is used to define right associativity.

Translation rules section

This section is the heart of the yacc specification file. Here we write grammar. With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values when a token is matched. An action is defined with a set of C statements. Action can do input and output, call subprograms, and alter external vectors and variables. The action normally sets the pseudo variable “$$” to some value to return a value. For example, an action that does nothing but return the value 1 is { $$ = 1;}

To obtain the values returned by previous actions, we use the pseudo-variables $1, $2, . . ., which refer to the values returned by the grammar symbol on the right side of a rule, reading from left to right.

7. ANALYSIS SYNTAX DIRECTED DEFINATION

Are a generalizations of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;

2. Productions are associated with Semantic Rules for computing the values of attributes.

• Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

• The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

• We distinguish between two kinds of attributes:

1. **Synthesized Attributes:** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes:** They are computed from the values of the attributes of both the parent nodes.

Form of Syntax Directed Definitions

Each production, A ! α, is associated with a set of semantic rules: b := f(c1, c2, . . . , ck), where f is a function and either.

b is a synthesized attribute of A, and c1, c2, . . . , ck are attributes of the grammar symbols of the production, or

b is an inherited attribute of a grammar symbol in α, and c1, c2, . . . , ck are attributes of grammar symbols in α or attributes of A.

Unit-III

Type Checking & Run Time Environment

Type checking: type system, specification of simple type checker, equivalence of expression, types, type conversion, overloading of functions and operations, polymorphic functions. Run time Environment: storage organization, Storage allocation strategies, parameter passing, dynamic storage allocation , Symbol table

1. TYPE CHECKING & RUN TIME ENVIRONMENT TYPE CHECKING

Parsing cannot detect some errors. Some errors are captured during compile time called static checking (e.g., type compatibility). Languages like C, C++, C#, Java, and Haskell uses static checking. Static checking is even called early binding. During static checking programming errors are caught early. This causes program execution to be efficient. Static checking not only increases the efficiency and reliability of the compiled program, but also makes execution faster.

Type checking is not only limited to compile time, it is even performed at execution time. This is done with the help of information gathered by a compiler; the information is gathered during compilation of the source program.

Errors that are captured during run time are called dynamic checks (e.g., array bounds check or null pointers dereference check). Languages like Perl, python, and Lisp use dynamic checking. Dynamic checking is also called late binding. Dynamic checking allows some constructs that are rejected during static checking. A sound type system eliminates run-time type checking for type errors. A programming language is strongly-typed, if every program its compiler accepts will execute without type errors. In practice, some of the type checking operations is done at run-time (so, most of the programming languages are not strongly typed).

For example, int x[100]; … x[i] → most of the compilers cannot guarantee that i will be between 0 and 99

A semantic analyzer mainly performs static checking. Static checks can be any one of the following type of checks:

Uniqueness checks: This ensures uniqueness of variables/objects in situations where it is required. For example, in most of the languages no identifier can be used for two different definitions in the same scope.

Flow of control checks: Statements that cause flow of control to leave a construct should have a place to transfer flow of control. If this place is missing, it is confusion. For example, in C language, “break” causes flow of control to exit from the innermost loop. If it is used without a loop, it confuses where to leave the flow of control.

Type checks: A compiler should report an error if an operator is applied to incompatible operands. For example, for binary addition, operands are array and a function is incompatible. In a function, the number of arguments should match with the number of formals and the corresponding types.

Name-related checks: Sometimes, the same name must appear two or more times. For example, in ADA, a loop or a block may have a name that appears at the beginning and end of the construct. The compiler must check whether the same name is used at both places.

What does semantic analysis do? It performs checks of many kinds which may include

- All identifiers are declared before being used.
- Type compatibility.
- Inheritance relationships.
- Classes defined only once.
- Methods in a class defined only once.

- Reserved words are not misused.

In this chapter we focus on type checking. The above examples indicate that most of the other static checks are routine and can be implemented using the techniques of SDT discussed in the previous chapter. Some of

them can be combined with other activities. For example, for uniqueness check, while entering the identifier into the symbol table, we can ensure that it is entered only once. Now let us see how to design a type checker. A type checker verifies that the type of a construct matches with that expected by its context. For example, in C language, the type checker must verify that the operator "%" should have two integer operands dereferencing is applied through a pointer, indexing is done only on an array, a user-defined function is applied with correct number and type of arguments. The goal of a type checker is to ensure that operations are applied to the correct type of operands. Type information collected by a type checker is used later by code generator.

1.1 TYPE SYSTEMS

Consider the assembly language program fragment. Add R1, R2, R3. What are the types of operands R1, R2, R3? Based on the possible type of operands and its values, operations are legal. It doesn't make sense to add a character and a function pointer in C language. It does make sense to add two float or int values. Irrespective of the type, the assembly language implementation remains the same for add. A language's type system specifies which operations are valid for which types. A type system is a collection of rules for assigning types to the various parts of a program. A type checker implements a type system. Types are represented by type expressions. Type system has a set of rules defined that take care of extracting the data types of each variables and check for the compatibility during the operation.

1.2 TYPE EXPRESSIONS

The type expressions are used to represent the type of a programming language construct. Type expression can be a basic type or formed by recursively applying an operator called a type constructor to other type expressions. The basic types and constructors depend on the source language to be verified. Let us define type expression as follows:

- A basic type is a type expression
- Boolean, char, integer, real, void, type_error
- A type constructor applied to type expressions is a type expression
- Array: array(I, T)
- Array(I,T) is a type expression denoting the type of an array with elements of type T and index set I, where T is a type expression. Index set I often represents a range of integers. For example, the Pascal declaration

var C: array[1..20] of integer;

associates the type expression array(1..20, integer) with C.

- Product: T1 × T2
- If T1 and T2 are two type expressions, then their Cartesian product T1 × T2 is a type expression. We assume that × associates to the left.
- Record: record((N1 × T1) × (N2 × T2))

A record differs from a product. The fields of a record have names. The record type constructor will be applied to a tuple formed from field types and field names. For example, the Pascal program fragment

```
type node = record
    address : integer ;
    data : array [1..15] of char
end;
```

var node table : array [1..10] of node ;

declares the type name “node” representing the type expression

record((address×integer) × (data × array(1..15,char)))

and the variable “node_table” to be an array of records of this type.

Pointer: pointer (T)

Pointer(T) is a type expression denoting the type “pointer to an object of type T where T is a type expression.

For example, in Pascal, the declaration

var ptr: *row

declares variable “ptr” to have type pointer(row).

Function: D → R

Mathematically, a function is a mapping from elements of one set called domain to another set called range. We may treat functions in programming languages as mapping a domain type “Dom” to a range type “Rg.”. The type of such a function will be denoted by the type expression Dom → Rg. For example, the built-in function mod, i.e. modulus of Pascal has type expression int × int → int.

As another example, the Pascal declaration

function fun(a, b: char) * integer;

says that the domain type of function “fun” is denoted by “char × char” and range type by “pointer(integer).” The type expression of function “fun” is thus denoted as follows:

char × char → pointer(integer)

However, there are some languages like Lisp that allow functions to return objects of arbitrary types. For example, we can define a function “g” of type (integer → integer) → (integer → integer).

That is, function “g” takes as input a function that maps an integer to an integer and “g” produces another function of the same type as output.

1.3 DESIGN OF SIMPLE TYPE CHECKER

Different type systems are designed for different languages. The type checking can be done in two ways. The checking done at compile time is called static checking and the checking done at run time is called dynamic checking. A system is said to be a Sound System if it completely eliminates the dynamic check. In such systems, if the type checker assigns any type other than type error for some fragment of code, then there is no need to check for errors when it is run. Practically this is not always true; for example, if an array X is declared to hold 100 elements. Usually the index would be from 0 to 99 or from 1 to 100 depending on the language support. And there is a statement in the program referred to as X[i]; during compilation this would not guarantee error free at runtime as there is possibility that if the value of i is 120 at run time then there will be an error. Therefore, it is essential that there is a need even for the dynamic check to be done.

Let us consider a simple language that has declaration statements followed by statements, where these statements are simple arithmetic statements, conditional statements, iterative statements, and functional statements. The program block of code can be generated by defining the rules as follows:

Type Declarations

	P → D “;” E	
	D → D “;” D	
	| id “:” T	{add_type(id.entry, T.type) }
	T → char	{T.type := char }

	T → integer	{T.type := int }
	:.....	:.....

	T → "*" T_1	{T.type := pointer(T_1.type) }
	T → array "["num "]" of T_1	{T.type := array(num.value, T_1.type) }

Table 1: Type Declarations

These rules are defined to write the declaration statements followed by expression statements. The semantic rule { add_type(id.entry, T.type) } indicates to associate type in T with the identifier and add this type info into the symbol table during parsing. A semantic rule of the form {T.type := int } associates the type of T to integer. So the above SDT collects type information and stores in symbol table.

1.4 TYPE CHECKING OF EXPRESSIONS

Let us see how to type check expressions. The expressions like 3 mod 5, A[10], *p can be generated by the following rules. The semantic rules are defined as follows to extract the type information and to check for compatibility.

	E → literal	{E.type := char}
	E → num	{E.type := int}
	E → id	{E.type := lookup(id.entry)}
	E → E1 mod E2	{E.type := if E1.type = int and E2.type = int
		then int
		else type_error}
	E → E1 "[" E2 "]"	{E.type := if E1.type = array(s, t) and E2.type = int
		Th en t
		else type_error}
	E → "*" E1	{E.type := if E1.type = pointer(t)
		then t
		else type_error}

Table 2: Type Checking Of Expressions

When we write a statement as i mod 10**,** then while parsing the element i, it uses the rule as E → id and performs the action of getting the data type for the id from the symbol table using the lookup method. When it parses the lexeme 10, it uses the rule E → num and assigns the type as int. While parsing the complete statement i mod 10**,** it uses the rule E → E_1 mod E_2, which checks the data types in both E_1 and E_2 and if they are the same it returns int otherwise type_error.

1.5 TYPE CHECKING OF STATEMENTS

The statements are simple of the form “a = b + c” or “a = b.” It can be a combination of statements followed by another statement or a conditional statement or iterative. To generate either a simple or a complex group of statements, the rules can be framed as follows: To validate the statement a special data type **void** is defined, which is assigned to a statement only when it is valid at expression level, otherwise type_error is assigned to indicate that it is invalid. If there is an error at expression level, then it is propagated to the statement, from the statement it is propagated to a set of statements and then to the entire block of program.

P → D “;” S	
S → id “:=” E	{S.type := if lookup(id.entry)= E.type
S → S_1 “;” S_2	then void
	else type_error}
	{S.type := if S_1.type = void and S_2.type
	= void
	then void
	else type_error}
S → if E then S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}
S → while E do S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}

Table 3: Type Checking Of Statements

1.6 TYPE CONVERSION

In an expression, if there are two operands of different type, then it may be required to convert one type to another in order to perform the operation. For example, the expression “a + b,” if a is of integer and b is real, then to perform the addition a may be converted to real. The type checker can be designed to do this conversion. The conversion done automatically by the compiler is implicit conversion and this process is known as coercion. If the compiler insists the programmer to specify this conversion, then it is said to be explicit. For instance, all conversions in Ada are said to be explicit. The semantic rules for type conversion are listed below.

	E → num	{E.type := int}

	E → num.num	{E.type := real}
	E → id	{E.type := lookup(id.entry)}
	E → E1 op E2	{E.type := if E1.type = int and E2.type = int
		then int
		else if E1.type = int and E2.type = real
		then real
		else if E1.type = real and E2.type = int
		then real
		else if E1.type = real and E2.type = real
		then real
		else type_error}

Table 4:Type Conversion

2. OVERLOADING OF FUNCTIONS AND OPERATORS

expression a + b, the addition operator "+" is overloaded because it performs different operations, when a and b are of different types like integer, real, complex, and so on. Another example of operator overloading is overloaded parenthesis in ada, that i, the expression A(i) has different meanings. It can be the i^{th} element of an array, or a call to function A with argument I, and so on. Operator overloading is resolved when the unique definition for an overloaded operator is determined. The process of resolving overloading is called operator identification because it specifies what operation an operator performs. The overloading of arithmetic operators can be easily resolved by processing only the operands of an operator.

Like operator overloading, the function can also be overloaded. In function overloading, the functions have the same name but different numbers and arguments of different types. In Ada, the operator "*" has the standard meaning that it takes a pair of integers and returns an integer. The function of "*" can be overloaded by adding the following declarations:

Function "*"(a,b: integer) return integer.
Function "*"(a,b: complex) return integer.
Function "*"(a,b: complex) return complex.

By addition of the above declarations, now the operator "*" can take the following possible types:

- It takes a pair of integers and returns an integer
- It takes a pair of integers and returns a complex number
- It takes a pair of complex numbers and returns a complex number

Function overloading can be resolved by the type checker based on the number and types of arguments. The type checking rule for function by assuming that each expression has a unique type is given as

E → E1(E2)

{

E.type : = t

E2.type : = t → u then

E.type : = u

else E.type : = type_error

}

	E' → E	{E'.type := E.type}
	E → id	{E.type := lookup(id.entry)}
	E → $E_1(E_2)$	{E.type := { u | there exists an s in E_2.type Such that s → u is in E_1.type }

Table 5: Overloading Of Functions And Operators

3. POLYMORPHIC FUNCTIONS

A piece of code is said to be polymorphic if the statements in the body can be executed with different types. A function that takes the arguments of different types and executes the same code is a polymorphic function. The type checker designed for a language like Ada that supports polymorphic functions, the type expressions are extended to include the expressions that vary with type variables. The same operation performed on different types is called overloading and are often found in object-oriented programming. For example, let us consider the function that takes two arguments and returns the result.

int add(int, int)
int add(real, real)
real add(real, real)

The type expression for the function add is given as

int × int → int
real × real → int
real × real → real

Write type expression for an array of pointer to real, where the array index ranges from 1 to 100.

Solution: The type expression is array[1..100,pointer(real)]

Write a type expression for a two-dimensional array of integers (that is, an array of arrays) whose rows are indexed from 0 to 9 and whose columns are indexed from –10 to 10.

Solution: Type expression is array[0..9, array[-10..10,integer]]

Write a type expression for a function whose domains are functions from integers to pointers to integers and whose ranges are records consisting of an integer and a character.

Solution: Type expression is

Domain type expression is integer → pointer(integer)

Let range has two fields a and b of type integer and character respectively.
Range type expression is record((a × integer)(b × character))

The final type expression is (integer → pointer(integer)) → record((a × integer) (b × character))

Consider the following program in C and the write the type expression for abc.

```
typedef struct
{
    int a,b;
 }  NODE;
    NODE abc[100];
```

Solution: The type expression for NODE is record((a × integer) × (b × integer)) abc is an array of NODE; hence, its type expression is

array[0..99, record((a × integer) × (b × integer))]

Consider the following declarations.

	type cell=record		
		info: integer;	
		next: pointer(cell)	
	type	link = ↑ cell;	
	var	next = link;	
		last = link;	
		p = ↑ cell;	
		q,r = ↑ cell;	

Table 6: Solution

Among the following, which expressions are structurally equivalent? Which are name equivalent? Justify your answer.

1. link
2. Pointer(cell)
3. Pointer(link)
4. Pointer (record ((info × integer) × (next × pointer (cell))).

Solution: Let	A = link
	B = pointer (cell)

	C = pointer (link)
	D = Pointer (record ((info × integer) × (next ×pointer(cell))).

Table 7: Solution

To get structural equivalence we need to substitute each type name by its type expression.

We know that, link is a type name. If we substitute pointer (cell) for each appearance of link we get,

A = pointer (cell)
B = pointer (cell)

C = pointer (pointer (cell))

D = Pointer (record ((info × integer) × (next × pointer (cell))).

We know that, cell is also type name given by
type cell=record

info: integer;
next: pointer(cell)

Substituting type expression for cell in A and B, we get

A = pointer (record ((info × integer) × (next × pointer (cell)))
B = pointer (record ((info × integer) × (next × pointer (cell)))
C = pointer(pointer(cell))

D = Pointer (record ((info × integer) × (next × pointer (cell))).

We have not substituted for the type expression of cell in "C" as it is anyway different from A, B, and D. That is, even if we substitute in C, the type expression will not be the same for A, B, C, and D.

We can say that A, B, and D are structurally equivalent.

For name equivalence, we will not do any substitutions. Rather we look at type expressions directly. If they are the same then we say they are name equivalent.

None of A, B, C, D are name equivalent.

4 RUN TIME ENVIRONMENT

4.1 STORAGE ALLOCATION INFORMATION

- Information about storage locations is kept in the symbol table
- If target is assembly code then assembler can take care of storage for various names
- Compiler needs to generate data definitions to be appended to assembly code
- If target is machine code then compiler does the allocation
- For names whose storage is allocated at runtime no storage allocation is done

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

4.2 STORAGE ORGANIZATION:

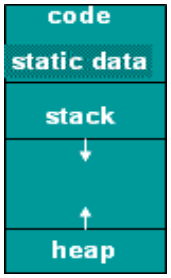


Figure 1:Storage stack

This kind of organization of run-time storage is used for languages such as FORTRAN, Pascal and C. The size of the generated target code, as well as that of some of the data objects, is known at compile time. Thus, these can be stored in statically determined areas in the memory. Pascal and C use the stack for procedure activations. Whenever a procedure is called, execution of activation gets interrupted, and information about the machine state (like register values) is stored on the stack. When the called procedure returns, the interrupted activation can be restarted after restoring the saved machine state. The heap may be used to store dynamically allocated data objects, and also other stuff such as activation information (in the case of languages where an activation tree cannot be used to represent lifetimes). Both the stack and the heap change in size during program execution, so they cannot be allocated a fixed amount of space. Generally they start from opposite ends of the memory and can grow as required, towards each other, until the space available has filled up.

Activation Record :

. temporaries: used in expression evaluation

. local data: field for local data

. saved machine status: holds info about machine status before procedure call

. access link : to access non local data

. control link : points to activation record of caller

. actual parameters: field to hold actual parameters

. returned value : field for holding value to be returned

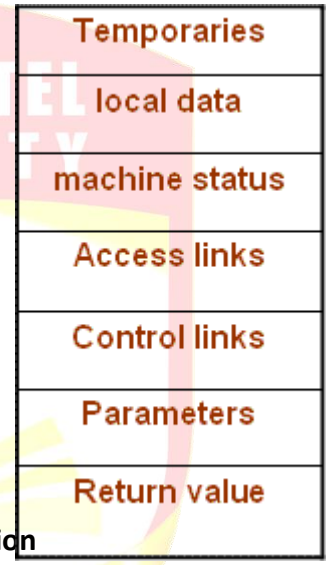


Figure 2: Storage Organization

The activation record is used to store the information required by a single procedure call. Not all the fields shown in the figure may be needed for all languages. The record structure can be modified as per the language/compiler requirements. For Pascal and C, the activation record is generally stored on the run-time stack during the period when the procedure is executing. Of the fields shown in the figure, access link and control link are optional (e.g. Fortran doesn't need access links). Also, actual parameters and return values are often stored in registers instead of the activation record, for greater efficiency. The activation record for a procedure call is generated by the compiler. Generally, all field sizes can be determined at compile time. However, this is not possible in the case of a procedure which has a local array whose size depends on a parameter. The strategies used for storage allocation in such cases will be discussed in the coming slides.

4.3 STORAGE ALLOCATION STRATEGIES

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

Static allocation:

. Names are bound to storage as the program is compiled

. No runtime support is required

. Bindings do not change at run time

. On every invocation of procedure names are bound to the same storage

. Values of local names are retained across activations of a procedure

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

{

int a;
print(a);
a = 10;

}

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

Type of a name determines the amount of storage to be set aside

. Address of a storage consists of an offset from the end of an activation record

. Compiler decides location of each activation

. All the addresses can be filled at compile time

. Constraints

- Size of all data objects must be known at compile time

- Recursive procedures are not allowed

- Data structures cannot be created dynamically

The type of a name determines its storage requirement, as outlined in slide 11. The address for this storage is an offset from the procedure's activation record, and the compiler positions the records relative to the target code and to one another (on some computers, it may be possible to leave this relative position unspecified, and let the link editor link the activation records to the executable code). After this position has been decided, the addresses of the activation records, and hence of the storage for each name in the records, are fixed. Thus, at compile time, the addresses at which the target code can find the data it operates upon can be filled in. The addresses at which information is to be saved when a procedure call takes place are also known at compile time. Static allocation does have some limitations:

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

Stack Allocation

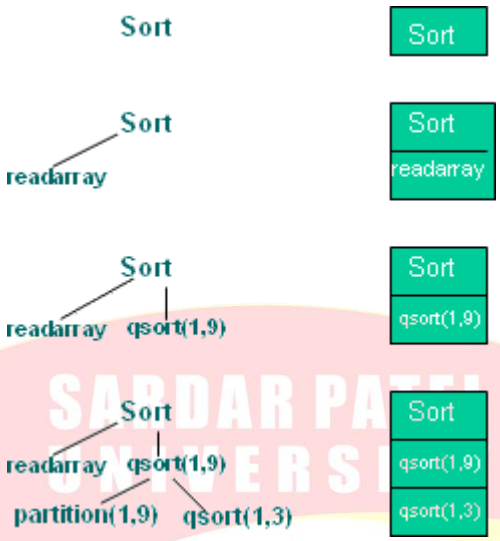


Figure 3: Stack Record

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

Calling Sequence :

A call sequence allocates an activation record and enters information into its field

A return sequence restores the state of the machine so that calling procedure can continue execution

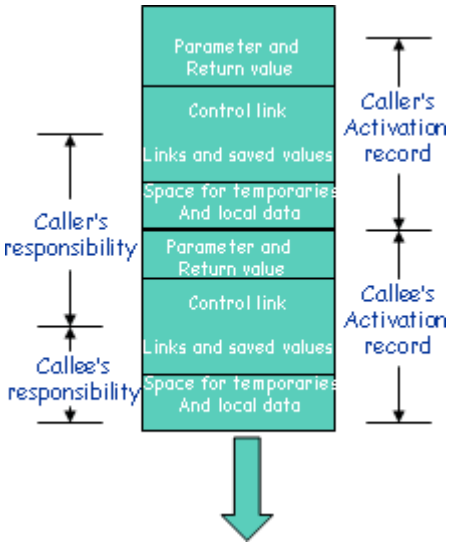


Figure 4: Calling Sequence in Stack

A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine so that the calling sequence can continue execution. Calling sequence and activation records differ, even for the same language. The code in the calling sequence is often divided between the calling procedure and the procedure it calls. There is no exact division of runtime tasks between the caller and the callee. As shown in the figure, the register stack top points to the end of the machine status field in the activation record. This position is known to the caller, so it can be made responsible for setting up stack top before control flows to the called procedure. The code for the callee can access its temporaries and the local data using offsets from stack top.

Call Sequence :

Caller evaluates the actual parameters

Caller stores return address and other values (control link) into callee's activation record
Callee saves register values and other status information

Callee initializes its local data and begins execution

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

Return Sequence :

Callee places a return value next to activation record of caller
Restores registers using information in status field

Branch to return address

Caller copies return value into its own activation record

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

Long Length Data

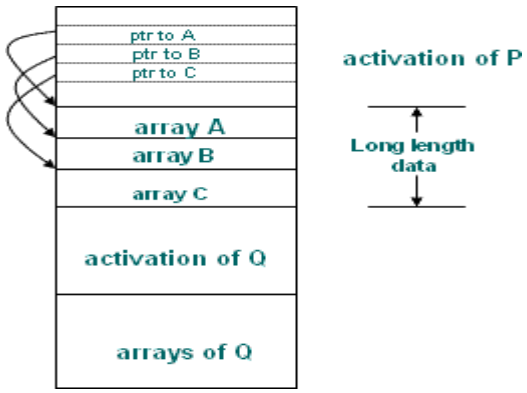


Figure 5: Data In Stack

The procedure P has three local arrays. The storage for these arrays is not part of the activation record for P; only a pointer to the beginning of each array appears in the activation record. The relative addresses of these pointers are known at the compile time, so the target code can access array elements through the pointers. Also shown is the procedure Q called by P . The activation record for Q begins after the arrays of P. Access to data on the stack is through two pointers, top and stack top. The first of these marks the actual top of the stack; it points to the position at which the next activation record begins. The second is used to find the local data. For consistency with the organization of the figure in slide 16, suppose the stack top points to the end of the machine status field. In this figure the stack top points to the end of this field in the activation record for Q. Within the field is a control link to the previous value of stack top when control was in calling activation of P. The code that repositions top and stack top can be generated at compile time, using the sizes of the fields in the activation record. When q returns, the new value of top is stack top minus the length of the machine status and the parameter fields in Q's activation record. This length is known at the compile time, at least to the caller. After adjusting top, the new value of stack top can be copied from the control link of Q.

Dangling references :

Referring to locations which have been deallocated

```
main()
{int *p;
p = dangle(); /* dangling reference */
}
int *dangle();
{
int i=23;
return &i;
}
```

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

Heap Allocation :

Stack allocation cannot be used if:

The values of the local variables must be retained when an activation ends

A called activation outlives the caller

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

There are two aspects of dynamic allocation -:

Runtime allocation and de-allocation of data structures.

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

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

Pieces may be de-allocated in any order

Over time the heap will consist of alternate areas that are free and in use
Heap manager is supposed to make use of the free space

For efficiency reasons it may be helpful to handle small activations as a special case
For each size of interest keep a linked list of free blocks of that size

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

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

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

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

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

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

For large blocks of storage use the heap manger.

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

Access to non-local names :

Scope rules determine the treatment of non-local names

A common rule is lexical scoping or static scoping (most languages use lexical scoping)

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

1. Static or Lexical scoping: It determines the declaration that applies to a name by examining the program text alone. E.g., Pascal, C and ADA.

2. Dynamic Scoping: It determines the declaration applicable to a name at run time, by considering the current activations. E.g., Lisp

5. DYNAMIC STORAGE ALLOCATION:

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

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

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

Explicit Allocation of Fixed Sized Blocks

Link the blocks in a list

Allocation and de-allocation can be done with very little overhead

Figure 6

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

Explicit Allocation of Fixed Sized Blocks

blocks are drawn from contiguous area of storage

An area of each block is used as pointer to the next block
A pointer available points to the first block

Allocation means removing a block from the available list
De-allocation means putting the block in the available list

Compiler routines need not know the type of objects to be held in the blocks
Each block is treated as a variant record

Suppose that blocks are to be drawn from a contiguous area of storage. Initialization of the area is done by using a portion of each block for a link to the next block. A pointer available points to the first block. Generally a list of free nodes and a list of allocated nodes is maintained, and whenever a new block has to be allocated, the block at the head of the free list is taken off and allocated (added to the list of allocated nodes). When a node has to be de-allocated, it is removed from the list of allocated nodes by changing the pointer to it in the list to point to the block previously pointed to by it, and then the removed block is added to the head of the list of free blocks. The compiler routines that manage blocks do not need to know the type of object that will be held in the block by the user program. These blocks can contain any type of data (i.e., they are used as generic memory locations by the compiler). We can treat each block as a variant record, with the compiler

routines viewing the block as consisting of some other type. Thus, there is no space overhead because the user program can use the entire block for its own purposes. When the block is returned, then the compiler routines use some of the space from the block itself to link it into the list of available blocks, as shown in the figure in the last slide.

Explicit Allocation of Variable Size Blocks
Storage can become fragmented
Situation may arise

If program allocates five blocks

then de-allocates second and fourth block

Fragmentation is of no consequence if blocks are of fixed size
Blocks cannot be allocated even if space is available

In explicit allocation of fixed size blocks, internal fragmentation can occur, that is, the heap may consist of alternate blocks that are free and in use, as shown in the figure. The situation shown can occur if a program allocates five blocks and then de-allocates the second and the fourth, for example. Fragmentation is of no consequence if blocks are of fixed size, but if they are of variable size, a situation like this is a problem, because we could not allocate a block larger than any one of the free blocks, even though the space is available in principle. So, if variable- sized blocks are allocated, then internal fragmentation can be avoided, as we only allocate as much space as we need in a block. But this creates the problem of external fragmentation, where enough space is available in total for our requirements, but not enough space is available in continuous memory locations, as needed for a block of allocated memory. For example, consider another case where we need to allocate 400 bytes of data for the next request, and the available continuous regions of memory that we have are of sizes 300, 200 and 100 bytes. So we have a total of 600 bytes, which is more than what we need. But still we are unable to allocate the memory as we do not have enough contiguous storage. The amount of external fragmentation while allocating variable-sized blocks can become very high on using certain strategies for memory allocation. So we try to use certain strategies for memory allocation, so that we can minimize memory wastage due to external fragmentation. These strategies are discussed in the next few slides.

6 SYMBOL TABLE

Compiler uses symbol table to keep track of scope and binding information about names
Symbol table is changed every time a name is encountered in the source; changes to table occur

- if a new name is discovered
- if new information about an existing name is discovered

Symbol table must have mechanism to:

- add new entries
- find existing information efficiently

Two common mechanisms:

- linear lists, simple to implement, poor performance
- hash tables, greater programming/space overhead, good performance

Compiler should be able to grow symbol table dynamically

if size is fixed, it must be large enough for the largest program

A compiler uses a symbol table to keep track of scope and binding information about names. It is filled after the AST is made by walking through the tree, discovering and assimilating information about the names. There

should be two basic operations - to insert a new name or information into the symbol table as and when discovered and to efficiently lookup a name in the symbol table to retrieve its information.

Variable	Information(type)	Space (byte)
A	integer	2
B	float	4
C	float	8
D	character	1
..	..	.....

Table 1: Symbol Table

Two common data structures used for the symbol table are -

- Linear lists:- simple to implement, poor performance.
- Hash tables:- greater programming/space overhead, good performance.

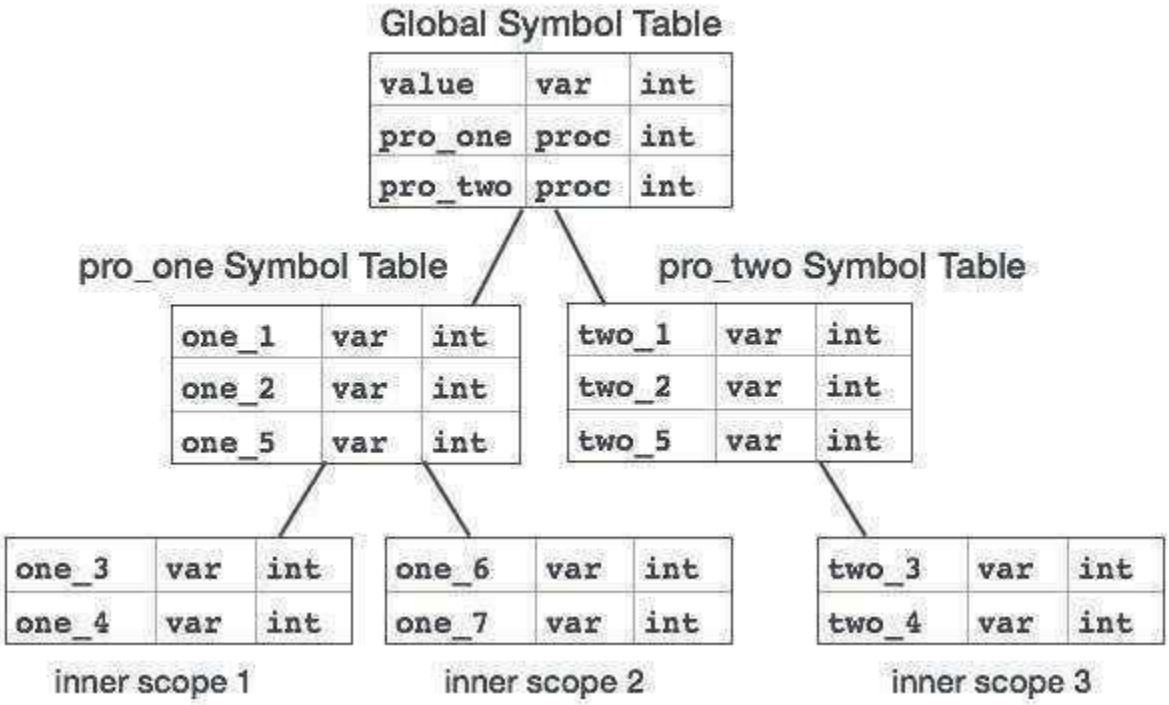


Figure 7 :Symbol table

Ideally a compiler should be able to grow the symbol table dynamically, i.e., insert new entries or information as and when needed. But if the size of the table is fixed in advance then (an array implementation for example), then the size must be big enough in advance to accommodate the largest possible program.

each entry for a declaration of a name

format need not be uniform because information depends upon the usage of the name
each entry is a record consisting of consecutive words

to keep records uniform some entries may be outside the symbol table
information is entered into symbol table at various times

keywords are entered initially

identifier lexemes are entered by lexical analyzer

symbol table entry may be set up when role of name becomes clear
attribute values are filled in as information is available

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

a name may denote several objects in the same block
int x; struct x {float y, z; }

lexical analyzer returns the name itself and not pointer to symbol table entry
record in the symbol table is created when role of the name becomes clear
in this case two symbol table entries will be created

attributes of a name are entered in response to declarations
labels are often identified by colon

syntax of procedure/function specifies that certain identifiers are formals
characters in a name

there is a distinction between token id, lexeme and attributes of the names
it is difficult to work with lexemes

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

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

Code Generation Intermediate code generation: Declarations, Assignment statements, Boolean expressions, Case statements, Back patching, Procedure calls Code Generation: Issues in the design of code generator, Basic block and flow graphs, Register allocation and assignment, DAG representation of basic blocks, peephole optimization, generating code from DAG.

1. INTERMEDIATE CODE GENERATION

The intermediate code is useful representation when compilers are designed as two pass system, i.e. as front end and back end. The source program is made source language independent by representing it in intermediate form, so that the back end is filtered from source language dependence. The intermediate code can be generated by modifying the syntax-directed translation rules to represent the program in intermediate form. This phase of intermediate code generation comes after semantic analysis and before code optimization.

Benefits of intermediate code

- Intermediate code makes target code generation easier
- It helps in retargeting, that is, creating more and more compilers for the same source language but for different machines.
- As intermediate code is machine independent, it helps in machine-independent code optimization.

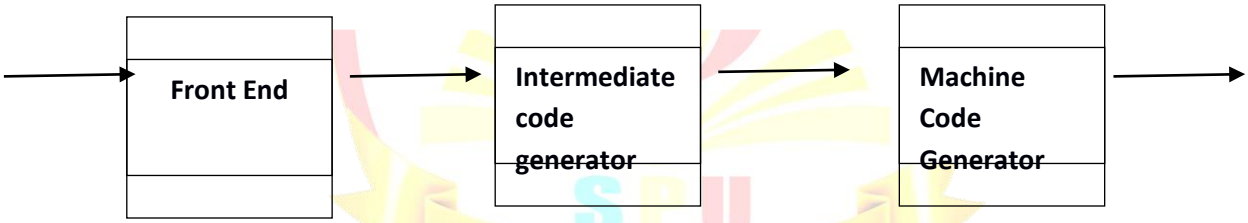


Figure 1:Intermediate code generation phase

A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

1.1 Intermediate code can be represented in the following four ways.

- Syntax trees
- Directed acyclic graph(DAG)
- Postfix notation
- Three address code

1.1.1 Syntax Trees

A syntax tree is a graphical representation of the source program. Here the node represents an operator and children of the node represent operands. It is a hierarchical structure that can be constructed by syntax rules. The target code can be generated by traversing the tree in post order form. For instance, consider an assignment statement a = b* – (c – d) + b* – (c – d) when represented using the syntax tree

The tree for the statement a = b* – (c – d) + b* – (c – d) is constructed by creating the nodes in the following order.

p_1 = mkleaf(id, c)
p_2 = mkleaf(id, d)
p_3 = mknode('–', p_1, p_2)
p_4 = mknode('U', p_3, NULL)
p_5 = mkleaf(id, b)

Production	Semantic Rule
S → id := E	S.nptr := mknode('assign', mkleaf(id, id.place), E.nptr)
E → E1 + E2	E.nptr := mknode('+', E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode('* ', E1.nptr ,E2.nptr)
E → − E1	E.nptr := mkunode('uminus', E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → id	E.nptr := mkleaf(id, id.place)

Table 1: Production in Syntax tree

1.1.2 Directed Acyclic Graph (DAG)

The tree that shows the same information with identified common sub-expression is called Directed Acyclic Graph (DAG). On examining the above example, it is observed that there are some nodes that are unnecessarily created. To avoid extra nodes these functions can be modified to check the existence of similar node before creating it. If a node exists then the pointer to it is returned instead of creating a new node. This creates a DAG, which reduces the space and time requirement.

1.1.3 Postfix Notation

Postfix notation is a linear representation of a syntax tree. This can be written by traversing the tree in the post order form. The edges in a syntax tree do not appear explicitly in postfix notation; only the nodes are listed. The order is followed by listing the parent node immediately after listing its left sub tree and its right sub tree. In postfix notation, the operators are placed after the operands.

1.1.4 Three Address Code

Three address code is a linear representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. Three address code is a sequence of statements of the form A = B *OP* C where A, B and C are the names of variables, constants or the temporary variables generated by the compiler. *OP* is any arithmetic operation or logical operation applied on the operands B and C. The name reflects that there are at most three variables where two are operands and one is for the result. In three address statement, only one operator is permitted; if the expression is large, then break it into a sequence of sub expressions using the BODMAS rules of arithmetic and store the intermediate results in newly created temporary variables. For example, consider the expression a + b * c;

this expression is expressed as follows:

T_1 = b * c

T_2 = a + T_1

Here T_1 and T_2 are compiler-generated temporary names. This simple representation of a complex expression in three address code makes the task of optimizer and code generator simple. It is also easy to rearrange the sequence for efficient code generation. Three address code for the statement a = b* – (c – d) + b* – (c – d) is

as follows:

$T_1 = c - d$

$T_2 = -T_1$

$T_3 = b * T_2$

$T_4 = c - d$

$T_5 = -T_4$

$T_6 = b * T_5$

$T_7 = T_3 + T_6$

$a = T_7$

The code can also be written for DAG as follows:

$T_1 = c - d$

$T_2 = -T_1$

$T_3 = b * T_2$

$T_4 = T_3 + T_3$

$a = T_4$

Types of Three Address Statements

For expressing the different programming constructs, the three address statements can be written in different standard formats and these formats are used based on the expression. Some of them are as follows:

- *Assignment statements with binary operator.* They are of the form A := B op C where op is a binary arithmetic or logical operation.
- *Assignment statements with unary operator.* They are of the form A: = op B where op is a unary operation like unary plus, unary minus, shift, etc.
- *Copy statements.* They are of the form A: = B where the value of B is assigned to variable A.
- *Unconditional Jumps* such as goto L**:** The label **L** with three address statement is the next statement number to be executed.
- *Conditional Jumps* such as if X relop Y goto L**.** If the condition is satisfied, then this instruction applies a relational operator (<=,>=,<,>) to X and Y and executes the statement with label L else the statement following if X relop Y goto L is executed.
- *Functional calls:* The functional calls are written as a sequence of param A, call fun,n, and returnB statements, where A indicates one of the input argument in *n* arguments to be passed to the function fun that returns B. The return statement is optional.

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:

- Non-terminal 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 itstype 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.

$T \rightarrow$ array [num] of T_1

T.type = array(num.val, T_1 .type)
T.width = num.val x T_1 .width

$T \rightarrow \uparrow T_1$

T.type = pointer(T_1 .type)
T.width = 4

This is the continuation of the example in the previous slide

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.

Whenever a procedure declaration Dproc id ; D1 ; S is processed, a new symbol table with a pointer to the symbol table of the enclosing procedure in its header is created and the entries for declarations in D1 are created in the new symbol table. The name represented by id is local to the enclosing procedure and is hence entered into the symbol table of the enclosing procedure.

Example

Program sort:

```
var a: array[1..n] of integer;
X:integer;
Procedure readarray;
var i: integer
…….
Procedure exchange(I,j:integers);
…….
Procedure quicksort(m,n : integer);
Var k,v:integer;

Function partition(x,y:integer):integer;
```

For the above procedures, entries for x, a and b *quicksort*are created in the symbol table of *sort***.** A pointer pointing to the symbol table of quicksort is also entered. Similarly, entries for k,v and partition are created in the symbol table of quicksort. The headers of the symbol tables of quicksort and partition have pointers pointing to sort and quicksort respectively

This structure follows from the example in the previous slide.

Creating symbol table
mktable (previous)

create a new symbol table and return a pointer to the new table. The argument previous points to the enclosing procedure

enter (table, name, type, offset)

creates a new entry

addwidth (table, width)

records cumulative width of all the entries in a table

enterproc (table, name, newtable)

creates a new entry for procedure name. newtable points to the symbol table of the new procedure

The following operations are designed :

- mktable(previous): creates a new symbol table and returns a pointer to this table. *previous* is pointer to the symbol table of parent procedure.
- enter(table,name,type,offset): creates a new entry for *name* in the symbol table pointed to by *table* .
- addwidth(table,width): records cumulative width of entries of a table in its header.
- enterproc(table,name ,newtable): creates an entry for procedure *name* in the symbol table pointed to by *table* . *newtable* is a pointer to symbol table for *name* .

Creating symbol table.

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

Table 1:Symbol table

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

productions.

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

Non-terminal P becomes the new start symbol when these productions are added to those in thetranslation 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.

3.1 Addressing Array Elements

Arrays are stored in a block of consecutive locations
assume width of each element is w

ith element of array A begins in location base + (i - low) x w where base is relative address of A[low]the

expression is equivalent to

i x w + (base-low x w)

⟶i x w + const

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

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

3.2 2-dimensional array

storage can be either row major or column major

in case of 2-D array stored in row major form address of $A[i_1, i_2]$ can be calculated as
base + $((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$

where $n_2 = high_2 - low_2 + 1$
rewriting the expression gives

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

this can be generalized for $A[i_1, i_2, .., i_k]$

Similarly, for a row major two dimensional array the address of A[i][j] can be calculated by the formula :

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

This can again be written as :

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

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

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 .E or E | E and E | not E | (E) | id relop id | true | false
Methods of Translating Boolean Expressions:

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

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

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

4.1 Numerical representation

a or b and not c
t_1 = not c

t_2 = b and t_1
t_3 = a or t_2

relational expression a < b is equivalent to if a < b then 1 else 0

1. if a < b goto 4.
2. t = 0
3. goto 5
4. t = 1

Consider the implementation of Boolean expressions using 1 to denote true and 0 to denote false. Expressions are evaluated in a manner similar to arithmetic expressions.

For example, the three address code for a or b and not c is:

t1 = not c

t2 = b and t1
t3 = a or t2

4.2 Syntax directed translation of boolean expressions

$E \longrightarrow E_1$ or E_2

E.place := newtmp

emit(E.place ':=' E_1 .place 'or' E_2 .place)
$E \longrightarrow E_1$ and E_2

E.place:= newtmp

emit(E.place ':=' E_1 .place 'and' E_2 .place)
$E \longrightarrow$ not E_1

E.place := newtmp
emit(E.place ':=' 'not' E_1 .place)
$E \longrightarrow (E_1)$ E.place = E_1 .place

The above written translation scheme produces three address code for Boolean expressions. It is continued to the next page**.**

4.3 Syntax directed translation of boolean expressions

$E \longrightarrow$ id1 relop id2
E.place := newtmp

emit(if id1.place relop id2.place goto nextstat+3)
emit(E.place = 0) emit(goto nextstat+2)
emit(E.place = 1)

$E \longrightarrow$ true

E.place := newtmp
emit(E.place = '1')
E false

E.place := newtmp
emit(E.place = '0')

In the above scheme, nextstat gives the index of the next three address code in the output sequence and emit increments nextstat after producing each three address statement.

Example:

Code for a < b or c < d and e < f

100: if a < b goto 103	if e < f goto 111
101: t_1 = 0	109: t_3 = 0
102: goto 104	110: goto 112

103: $t_1 = 1$	111: $t_3 = 1$
104:	112:
if c < d goto 107	$t_4 = t_2$ and t_3
105: $t_2 = 0$	113: $t_5 = t_1$ or t_4
106: goto 108	

107: $t_2 = 1$	
108:	

Table 2:Three address code for above example

A relational expression a < b is equivalent to the conditional statement if a < b then 1 else 0 and three address code for this expression is:

100: if a < b goto 103.
101: t = 0

102: goto 104

103: t = 1

104:

It is continued from 104 in the same manner as the above written block.

5. CASE STATEMENT

switch expression
begin

case value: statement
case value: statement

..

case value: statement
default: statement
end

evaluate the expression

find which value in the list of cases is the same as the value of the expression

Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression execute the statement associated with the value found.

There is a selector expression, which is to be evaluated, followed by ***n*** constant values that the expression can take. This may also include a *defaul*t value which always matches the expression if no other value does. The intended translation of a switch case code to:

- evaluate the expression
- find which value in the list of cases is the same as the value of the expression.
- Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression. executethe statement associated with the value found
- Most machines provide instruction in hardware such that case instruction can be implemented easily. So, case is treated differently and not as a combination of if-then statements.

Translation

code to evaluate E into t	code to evaluate E into t
if t <> V1 goto L1	goto test

Table 3: Translation table

Efficient for n-way branch

There are two ways of implementing switch-case statements, both given above. The above two implementations are equivalent except that in the first case all the jumps are short jumps while in the second case they are long jumps. However, many machines provide the n-way branch which is a hardware instruction. Exploiting this instruction is much easier in the second implementation while it is almost impossible in the first one. So, if hardware has this instruction the second method is much more efficient.

6. BACKPATCHING

code for S1	L1: code for S1
goto next	goto next
L1 if t <> V2 goto L2	L2: code for S2

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 proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions:

code for S2	goto next
goto next	..
L2: ..	Ln: code for Sn
Ln-2 if t <> Vn-l goto Ln-l	goto next
code for Sn-l	test: if t = V1 goto L1
goto next	if t = V2 goto L2
Ln-1: code for Sn	..
next:	if t = Vn-1 goto Ln-1
	goto Ln
	next:

- 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.
- merge(p1,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.
- backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions

E ⟶E_1 or M E_2

| E_1 and M E_2

| not E_1

| (E_1)

| id_1 relop id_2

| true

| false M ? ε

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

E.truelist and E.falselist.

Consider production E ⟶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 non-terminal 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 or M E2 { backpatch (E1.falselist, M.quad);
E.truelist : = merge(E1.truelist, E2.truelist);

E.falselist : = E2.falselist }

(2) E ⟶E1 and M E2 { backpatch (E1.truelist, M.quad);
E.truelist : = E2.truelist;

E.falselist : = merge(E1.falselist, E2.falselist) }

(3) E ⟶not E1 { 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 }

7. PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it isimperative for a compiler to generate good code for procedure calls and returns. The run-timeroutines 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 entryto 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 afterit 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) }

(2) Elist ⟶Elist , E

{ append E.place to the end of queue }

(3) Elist ⟶E

{ initialize queue to contain only E.place }

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

8. CODE GENERATION

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

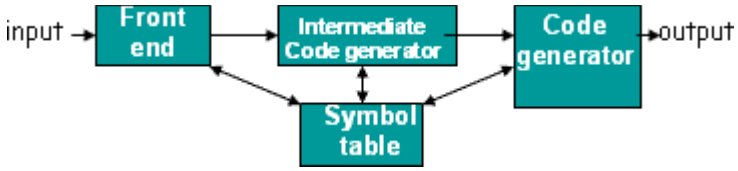


Figure 3: Code Generation in Compiler

8.1 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

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.

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. Reloadable machine language

It allows subprograms to be compiled separately.

c. Assembly language

Code generation is made easier.

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.

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.

Register allocation

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

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

Register assignment – the specific register that a variable will reside in is picked.

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

Where, x – dividend even register in even/odd register pair

y – Divisor even register holds the remainder odd register holds the quotient

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.

9. BASIC BLOCKS AND FLOW GRAPHS

9.1 Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:

```
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

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 a and b of length 20

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

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

Transformations on Basic Blocks:

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

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common sub expression 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 betransformed 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 changingthe 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 ofexpressions it computes.

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

9.2 FLOW GRAPHS

Flow graph is a directed graph containing the flow-of-control information for the set ofbasic blocks making up a program.

The nodes of the flow graph are basic blocks. It has a distinguished initial node.

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.

9.3 NEXT-USE INFORMATION

If the name in a register is no longer needed, then we remove the name from the registerand 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"

10. 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

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.

11. PEEPHOLE OPTIMIZATION

Target code often contains redundant instructions and suboptimal constructs.

Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence peephole is a small moving window on the target systems.

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examininga 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 codein the peep 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 yCase
(iii) x : = y

hole need not be contiguous, although some implementations do require this.

Input: A basic block

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

3. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.

4. For each node a list of attached identifiers to hold the computed values.Case (i)
x : = y OP z

Case (ii) x : = OP yCase
(iii) x : = y

Peephole optimization examples.

Redundant loads and stores

Consider the code sequence

Move R_0 , a Move a, R_0

Instruction 2 can always be removed if it does not have a label.

Now, we will give some examples of program transformations that are characteristic of peephole optimization: Redundant loads and stores: If we see the instruction sequence

Move R_0 , a

Move a, R_0

We can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0. Note that is (2) has a label, we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Peephole optimization example

Consider the following code

define debug 0If(debug)

{

Print debugging info

}

This may be translated asIf debug =1 goto L1

Goto L2

L1:print debugging infoL2:

Eliminate jump over iumpsIf

debug<> 1 goto L2

Print debugging informationL2:

Another opportunity for peephole optimization is the removal of unreachable instructions.

12. 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 thanwe can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

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

For example, consider the following basic block:

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 node immediately follow 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

Code generation phase generates the code based on the numbering assigned to each node T. All the registers available are arranged as a stack to maintain the order of the lower register at the top. This makes assumption that the required number of registers cannot exceed the number of available registers. In some cases, we may need to spill the intermediate result of a node to memory. This algorithm also does not take advantage of the commutative and associative properties of operators to rearrange the expression tree.

It first checks whether the node T is a leaf node; if yes, it generates a load instruction corresponding to it as load top (), T. If the node T is an internal node, then it checks the left *l* and right *r* sub tree for the number assigned. There are three possible values, the number on the right is 0 or greater than or less than the number on the left. If it is 0 then call the generate () function with left sub tree *l* and then generate instruction op top (),r. If the numbering on the left is greater than or equal to right, then call generate () with left sub tree, get new register by popping the top, call generate () with right sub tree, generate new instruction for OP R, top ()**,** and push back the used register on to the stack.

Unit –V

Code Optimization Introduction to Code optimization: sources of optimization of basic blocks, loops in flow graphs, dead code elimination, loop optimization, Introduction to global data flow analysis, Code Improving transformations ,Data flow analysis of structure flow graph Symbolic debugging of optimized code.

1. CODE OPTIMIZATION

Code optimization phase is an optional phase in the phases of a compiler, which is either before the code generation phase or after the code generation phase. This chapter focuses on the types of optimizer and the techniques available for optimizing.

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:

1.1 Machine independent optimizations:

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

1.1 Machine dependant optimizations:

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

1.1 The criteria for code improvement transformations:

Simply stated, the best program transformations are those that yield the most benefit for theleast effort.

The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

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

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

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.

2.1 Function-Preserving Transformations

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

The transformations

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

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

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

2.3 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 re-computing 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 value of i is not been changed from definition to use.

2.4 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

2.5 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. An optimization can be done by eliminating dead code.

Example:

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

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

2.6 Constant folding:

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

One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

2.7 Loop Optimizations:

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

Three techniques are important for loop optimization:

code motion, which moves code outside a loop;

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

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

2.8 Code Motion:

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

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

while (i <= limit-2) /* statement does not change limit*/
Code motion will result in the equivalent of

t= limit-2;

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

2.9 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.

3. OPTIMIZATION OF BASIC BLOCKS

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

Structure-Preserving Transformations
Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

Common sub-expression elimination
Dead code elimination

Renaming of temporary variables

Interchange of two independent adjacent statements.

3.1 Common sub-expression elimination:

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

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

3.2 Dead code elimination:

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

3.3 Renaming of temporary variables:

A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.

In this we can transform a basic block to its equivalent block called normal-form block.
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 t 1 does not affect the value of t2.

3.4 Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks.

This includes simplifying expressions or replacing expensive operation by cheaper onesi.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

4. 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.

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

Invariant code : A fragment of code that resides in the loop and computes the same• value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.

Induction analysis : A variable is called an induction variable if its value is altered• within the loop by a loop-invariant value.

Strength reduction : There are expressions that consume more CPU cycles, time,• and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- There must be at least one 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.

5.DEAD-CODEELIMINATION

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.
Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.

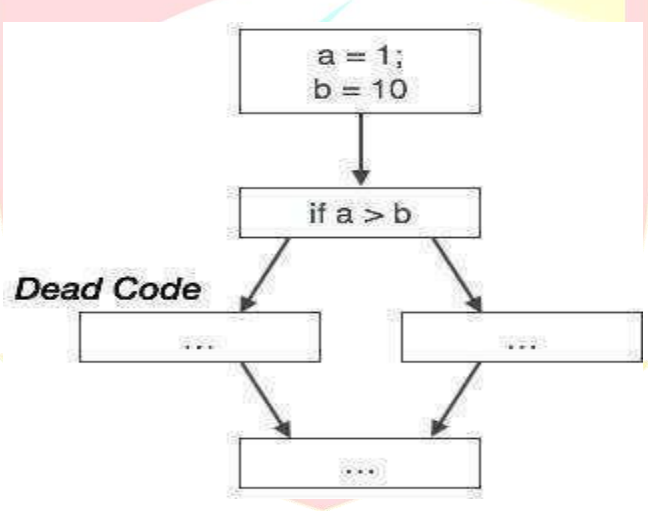


Figure 1: Dead–code elimination

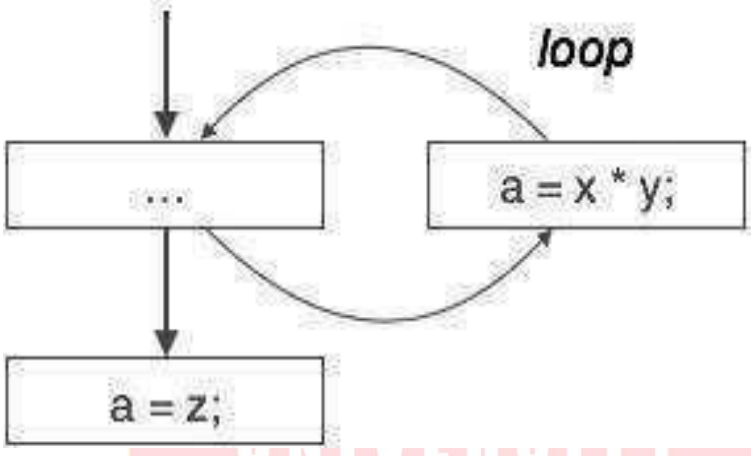
The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression 'x * y'. Let us assume that the value assigned to 'a' is never used inside the loop. Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.

Figure 2: Dead–code elimination

Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

5.1 Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands. Whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



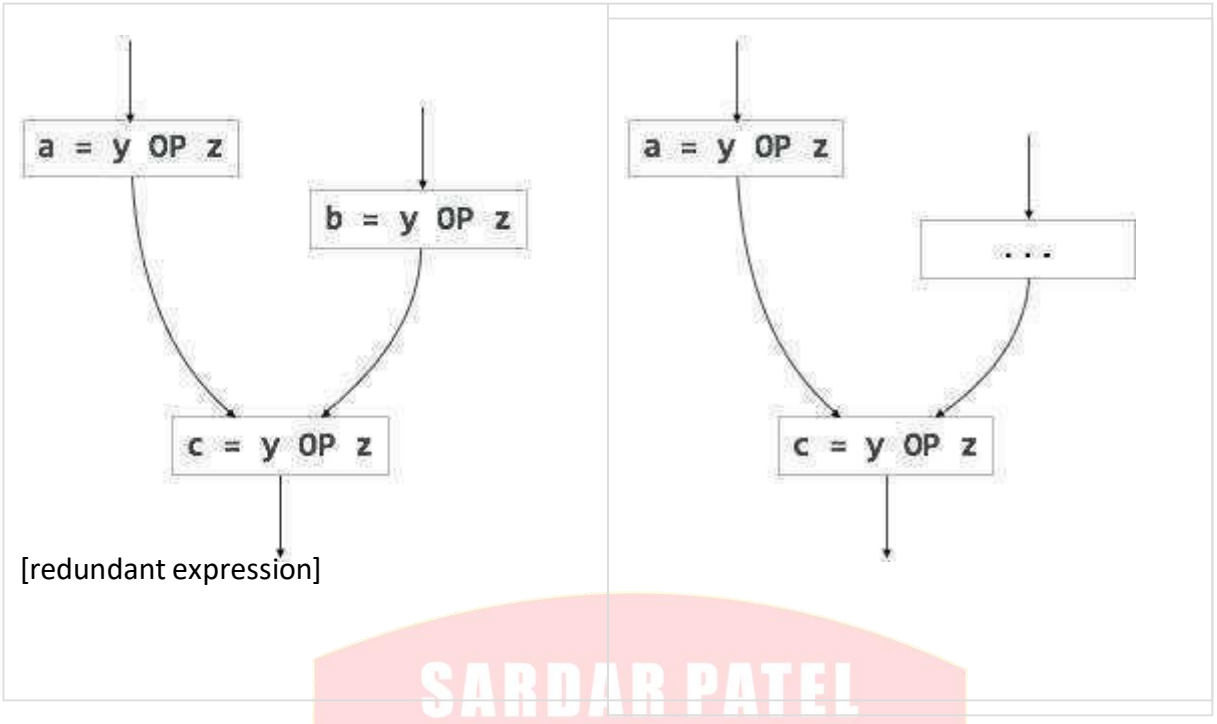


Figure 3: Partial Redundancy

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique. Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
```

```
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

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.”

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

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

6.1 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.

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.

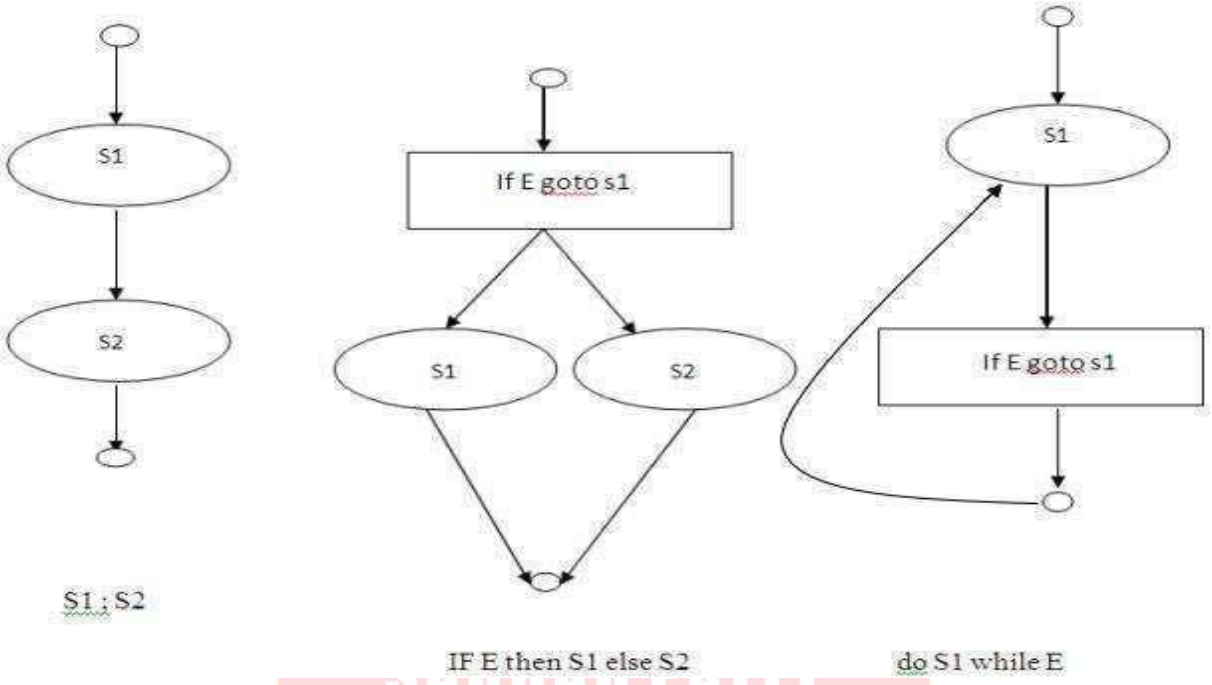
6.2 Reaching definitions:

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.



6.3 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.

7. CODE IMPROVING 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.

7.1 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.

Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.

Algorithm will miss the fact that a*z and c*z must have the same value in

a :=x+y c :=x+y

vs

b :=a*z d :=c*z

Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

7.2 Copy propagation:

- Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x.
- Statement s must be the only definition of x reaching u.
- On every path from s to including paths that go through u several times, there are no assignments to y.
- Condition (1) can be checked using ud-changing information. We shall set up a new dataflow 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.

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 z is defined prior to statement s in that block, do the following.

[redacted]scover the evaluations of y+z that reach s[redacted]ck, we follow flow graph edges, searching backward from s[redacted]ck. Howe[redacted], we do not g[redacted]gh any b[redacted]hat evaluates y+z. The last evaluation of y+z in each block e[redacted]ntered is an evaluation of y[redacted]reache[redacted]

Create new var[redacted]

Replace each st[redacted]t w: =y+z found i[redacted]y

u : = y + z

w : = u

- Replace statement s by x:=u.
- Some remarks about this algorithm are in order.

INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, and with c_in[B] representing t
solution to equations that is the set of copies x:=y that reach block B along every path, with no assignment t
or y following the last occurrence of x:=y on the path. We also need ud-chains giving the uses of each definitio

OUTPUT: A revised flow graph.

METHOD: For each copy s : x:=y do the following:

Determine those uses of x that are reached by this definition of namely, s: x: =y.

Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular us
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
the only definition of x that reaches B.

7.3 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.

Performing code motion:

Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.

The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.

There is no other statement in the loop that assigns to x. Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.

No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

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.

8. SYMBOLIC DEBUGGING SCHEME FOR OPTIMIZED CODE:

Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level. In response to a user query, the debugger retrieves the value of a source variable in a manner consistent with respect to the source statement where execution has halted.

Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level [Hen82]. Symbolic debugging must ensure that in response to a user inquiry, the debugger is able to retrieve and display the value of a source variable in a manner consistent with respect to a breakpoint in the source code. Code optimization techniques usually makes symbolic debugging harder. While code optimization techniques such as transformations must have the property that the optimized code is functionally equivalent to the un- optimized code, such optimization techniques may produce a different execution sequence from the source statements and alter the intermediate results. Debugging un-optimized rather than optimized code is not acceptable for several reasons , including:

- while an error in the un-optimized code is undetectable, it is detectable in the optimized code,
- optimizations may be necessary to execute a program due to hardware limitations, and
- a symbolic debugger for optimized code is often the only tool for finding errors in an optimization tool.

In a design-for-debugging (DfD) approach that enables retrieval of source values for a globally optimized behavioral specification. The goal of the DfD technique is to modify the original code in a pre-synthesis step such that every variable of the source code is controllable and observable in the optimized program.