Name of the Paper: Compiler Design
Paper Code: CS604A
Contact (Periods/Week): 3L+1T / Week
Credit Point: 4
No. of Lectures: 35

Prerequisite:

- ✓ Mathematics
- ✓ Concept of programming languages
- ✓ Data structures
- ✓ Computer architecture
- ✓ Formal languages and automata theory
- ✓ Some advanced math might be required if you adventure in code optimization

Course Objectives:

To make the student understand the process involved in a compiler, create an overall view of various types of translators, linkers, loaders, and phases of a compiler, understand what is syntax analysis, various types of parsers especially the top down approach, awareness among students the various types of bottom up parsers, understand the syntax analysis and, intermediate code generation, type checking, the role of symbol table and its organization, Code generation, machine independent code optimization and instruction scheduling.

Course Outcomes:

CS604A .1. To illustrate the basic concept of compilers and discuss on the components as well as the strengths and weaknesses of various phases of designing a compiler.
CS604A .2. To formulate the theories of creating simple compilers using C programming languages.
CS604A .3. To design and analyze algorithms for syntactic and semantic analysis of the process of designing compilers.
CS604A .4. To explain the role of finite automata in compiler design.
CS604A .5. Identify the similarities and differences among various parsing techniques and grammar transformation techniques

Syllabus:

Module I [7L]
Compilers, Cousins of the Compiler, Analysis-synthesis model, The phases of the compiler.
The role of the lexical analyzer, Tokens, Patterns, Lexemes, Input buffering, Specifications of a token, Recognition of tokens, Finite automata, From a regular expression to an NFA, From a regular expression to DFA, Design of a lexical analyzer generator (Lex).

Module II [10L]
The role of a parser, Context free grammars, Writing a grammar, Top down Parsing, Non recursive Predictive parsing (LL), Bottom up parsing, Handles, Viable prefixes, Operator precedence parsing, LR parsers (SLR, LALR, Canonical LR), Parser generators (YACC), Error Recovery strategies for different parsing techniques.
Syntax directed translation: Syntax directed definitions, Construction of syntax trees, Bottom-up evaluation of S-attributed definitions, L-attributed definitions, Bottom-up evaluation of inherited attributes.

Module III [7L]

Type systems, Specification of a simple type checker, Equivalence of type expressions, Type conversions
Source language issues (Activation trees, Control stack, scope of declaration, Binding of names), Symbol tables, dynamic storage allocation techniques.

Module IV [3L]

Intermediate languages, Graphical representation, Three-address code, Implementation of three address statements (Quadruples, Triples, Indirect triples).

Module V [8L]

Consideration for Optimization, scope of optimization, local optimization, loop optimization, folding, DAG representation, Flow Graph, Data flow equation, global optimization, redundant sub expression elimination, induction variable elimination, copy propagation, basic blocks & flow graphs, transformation of basic blocks, DAG representation of basic blocks, peephole optimization
Object code forms, machine dependent code optimization, register allocation and assignment, generic code generation algorithms, DAG for register allocation.

Recommended Text Books:

[1] Alfred Aho, V. Ravi Sethi, D. Jeffery Ullman, “Compilers Principles, Techniques and Tools”, Addison Wesley, 2nd edition
[2] Holub Allen. Compiler Design in C, PHI, 1993.

Recommended reference Books:

[1] Chattopadhyay, Santanu. Compiler Design. PHI Learning Pvt. Ltd., 2005
[2] Tremblay and Sorenson Compiler Writing-McgrawHill International

CO-PO Mapping:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CS604A.1	3											
CS604A.2		3										
CS604A.3			3									
CS604A.4	3	2										
CS604A.5				3								

Module I

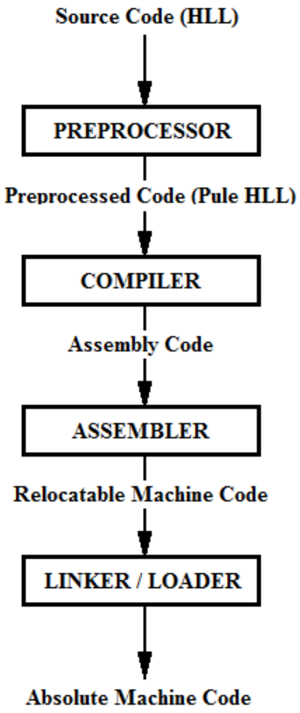
Lecture I

Introduction

A computer system is a well-adjusted combination of both the hardware and the software. The hardware is just a piece of electronic device which is driven by a compatible software. The system understands the instructions in the form of electronic charge, which is the equivalent to the binary language in software programming. To instruct the hardware, codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and burdensome task for the computer programmers to write codes in such formats, which is why we need compilers in practice.

Language Processing System

The set of hardware understands a language, which is quite difficult for humans to understand. The programmers, therefore, write programs in high-level languages, which is English-like and easier 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 understood by the machine. This is known as Language Processing System.



The high-level language (HLL) is converted into the Executable code (absolute machine code) through various phases. A compiler is basically a computer program that converts the high-level language into the assembly code. Likewise, an assembler is a computer program that converts the assembly code into the machine code.

Let us now understand how a program is executed on a host machine using a compiler (say, C compiler)

1. The programmer writes a program in C language (high level language).
2. The preprocessor performs its job and converts the source code into the pure high level language.

3. The C compiler then compiles the program written in the pure high level language and translates it into the corresponding assembly program (low level language).
4. An assembler then translates this assembly program into the machine code (object).
5. A linker (a computer program) is used to link all the parts of the program together for execution (executable machine code).
6. A loader (a computer program) loads all of them into the primary memory for getting them executed. Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor produce input to the compiler. They may perform the following functions.

- Macro processing: A preprocessor may allow a user to define macros that are short hands for longer constructs.
- File inclusion: A preprocessor may include header files into the program text.
- Rational preprocessor: these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
- Language Extensions: These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

Compiler

The compiler is a translator program that translates a program written in the pure high level language into an equivalent program in assembly level language. An important part of a compiler is that it points out errors in the computer program.

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.

Loader and Linker

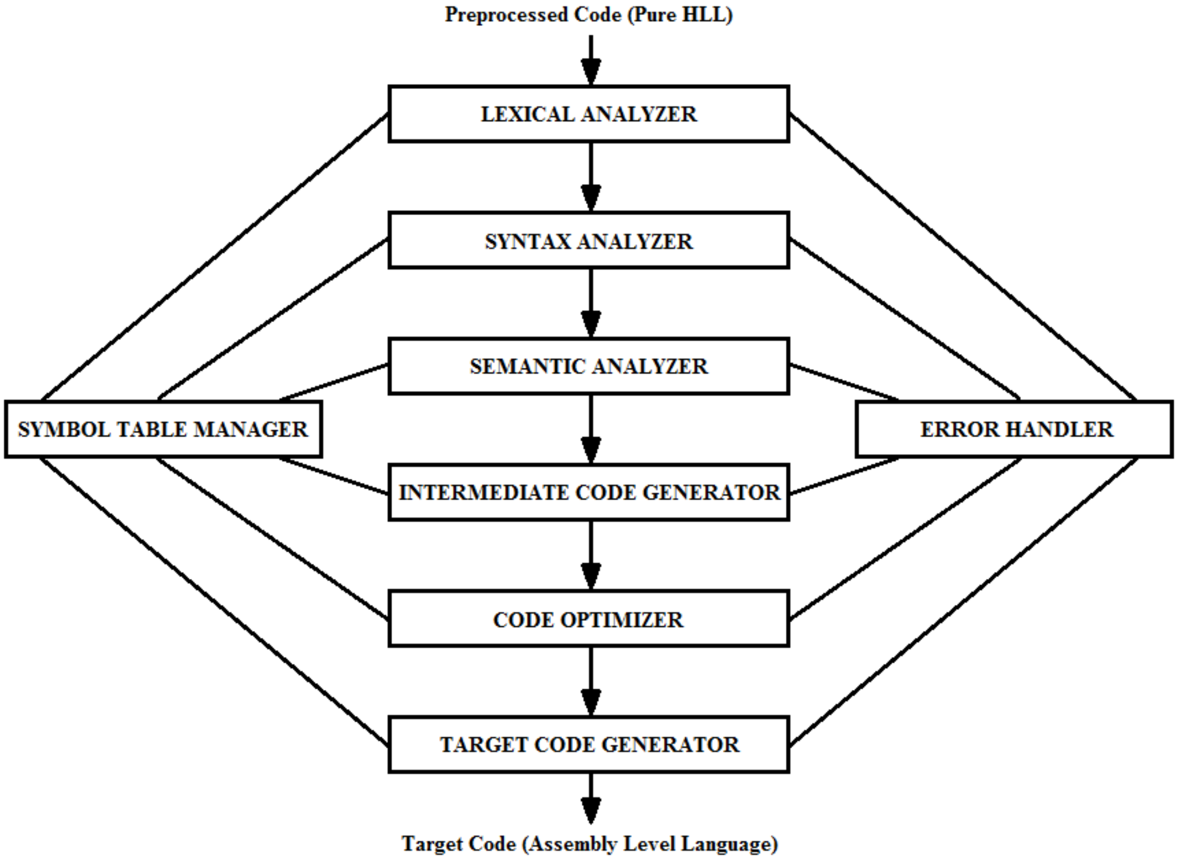
Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader.

A loader is a program that places programs into memory and prepares them for execution. It would be more efficient if subroutines could be translated into object form the loader could relocate directly behind the user's program.

Lecture II

Phases of a compiler

A compiler operates in different phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown below.



The above mentioned phases are classified into two categories as follows:

- Analysis (Machine Independent/Language Dependent)
    - ✓ Lexical Analyzer
    - ✓ Syntax Analyzer
    - ✓ Semantic Analyzer
    - ✓ Intermediate Code Generator
- Synthesis (Machine Dependent/Language independent)
    - ✓ Coder Optimizer
    - ✓ Target Code Generator

Lexical Analysis

The Lexical Analyzer (or Scanner) reads the source program one character at a time, figurine the source program into a sequence of automatic 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.

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.

Target 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 Manager

This is the portion to keep the names used by the program and records essential information about each. The data structure used to record this information called a Symbol Table.

Error Handlers

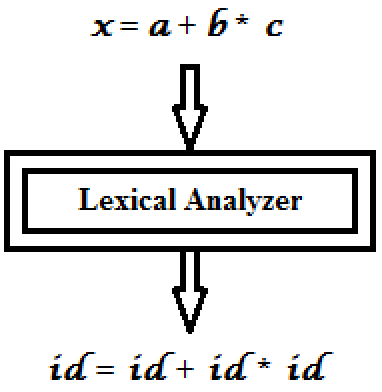
It is invoked when a flaw error in the source program is detected. The output of the Lexical Analyzer is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined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

Lecture III

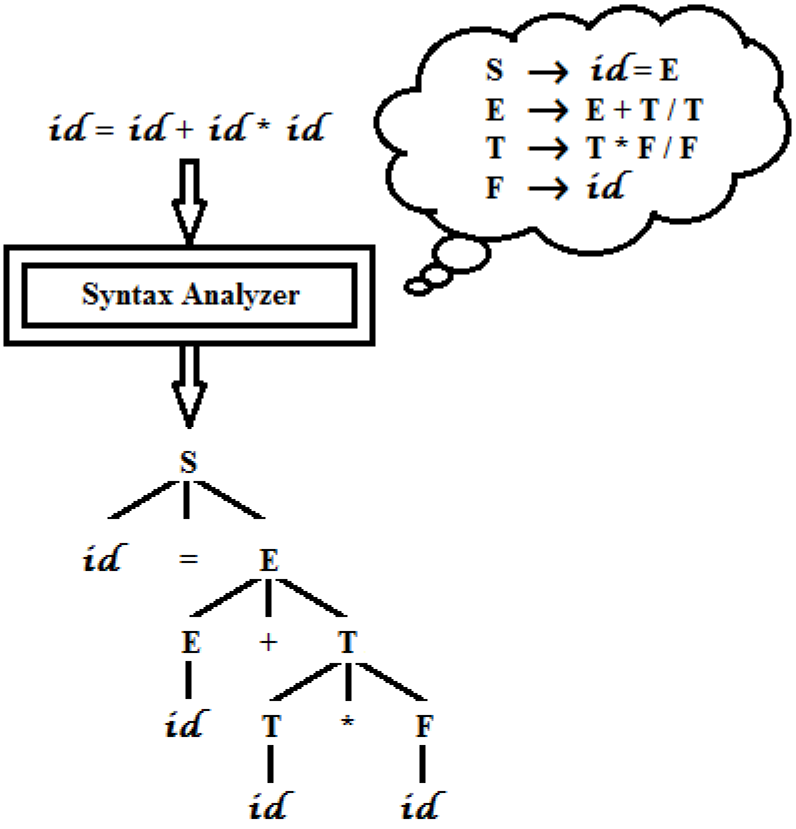
Phases of a compiler (Contd.)

We will now illustrate the function of the different phases of a compiler with a suitable example. Let us consider the source code (preprocessed) $x = a + b * c$.

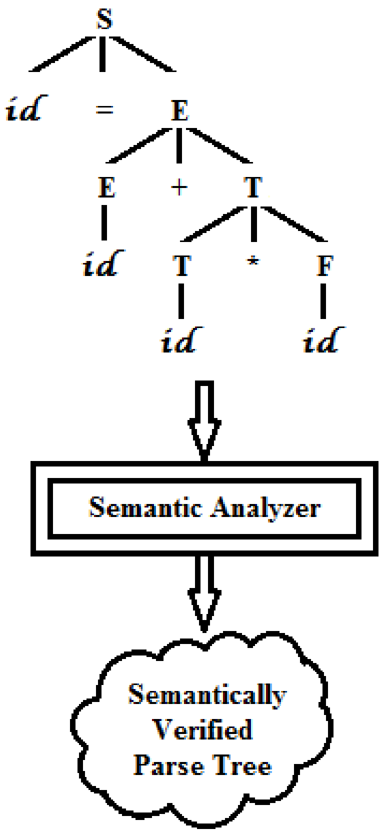
Lexical Analysis



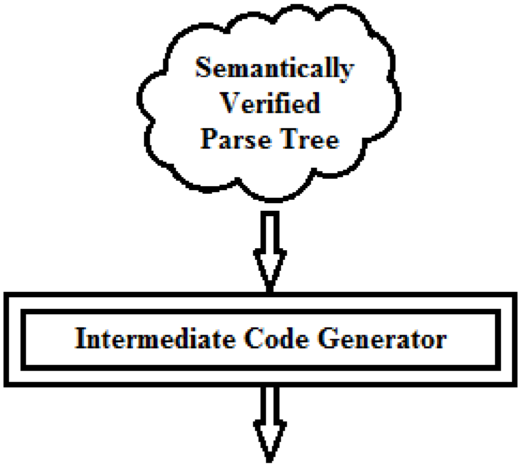
Syntax Analysis



Semantic Analyzer



Intermediate Code Generator

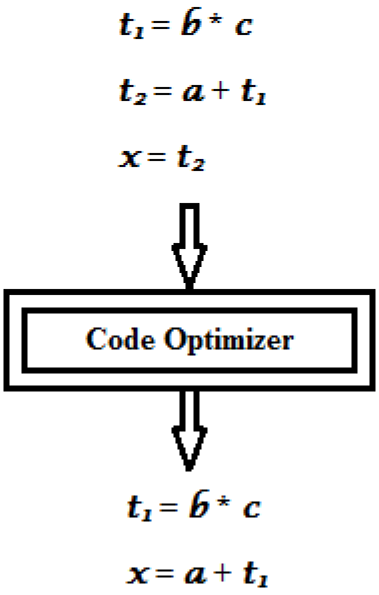


$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Code Optimizer



$t_1 = b * c$

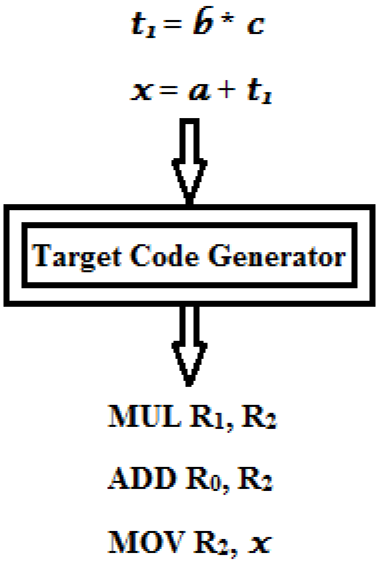
$t_2 = a + t_1$

$x = t_2$

$t_1 = b * c$

$x = a + t_1$

Target Code Generator



$t_1 = b * c$

$x = a + t_1$

MUL R_1, R_2

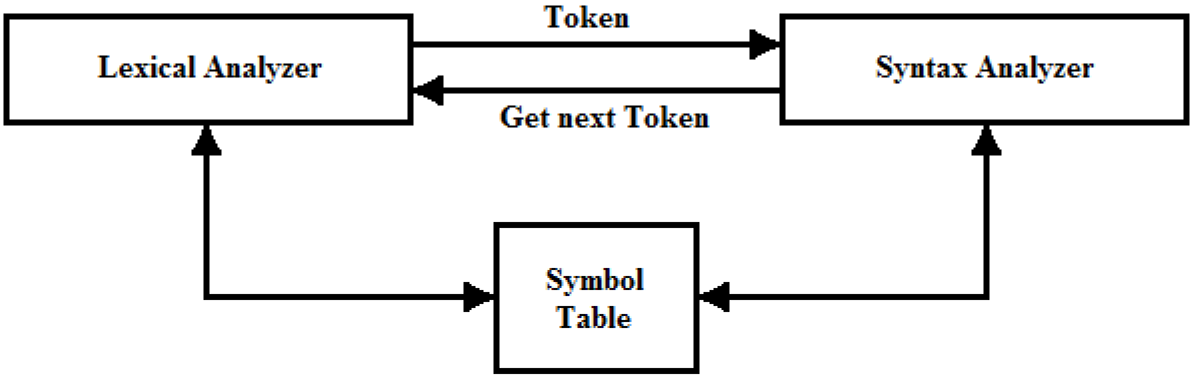
ADD R_0, R_2

MOV R_2, x

Lecture IV

Lexical Analyzer

The Lexical Analyzer is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



On receipt of the "get next token" command form the Parser, the Lexical Analyzer keeps on reading the input characters until it identifies the next token. The Lexical Analyzer returns to the parser representation for the token it has found.

The Lexical Analyzer may also perform certain secondary tasks as the user interface. One such task is eliminating the Command Line Arguments and unnecessary white spaces present in the form of blank and/or tab and/or new line characters. Another is correlating error message from the compiler with the source program.

Lexical Analysis vs. Syntax Analysis

- ✓ A Lexical Analyzer (Scanner) simply turns an input String (a file) into a list of tokens. These tokens represent elements like identifiers, keywords, operators etc.
- ✓ A Syntax Analyzer (Parser) converts this set of tokens into a tree-like object (Parse Tree) to represent how the tokens fit together to form the cohesive whole (referred to as a sentence).

Token, Lexeme, Pattern

Token: A token is a sequence of characters that can be treated as a single logical entity. Typical tokens are as follows:

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 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.

Lexical Errors

Lexical errors are the errors thrown by the lexer when unable to continue with. 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. Following are some potential Error-recovery actions:

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

Difference between Compiler and Interpreter

Compiler		Interpreter	
1	A compiler converts the high level instruction into machine language.	1	An interpreter converts the high level instruction into an intermediate form.
2	Before execution, entire program is translated by the compiler.	2	An interpreter translates the program line by line.
3	List of errors is created by the compiler after compilation.	3	An interpreter stops translating after the first error is encountered.
4	An independent executable file is created by the compiler.	4	The interpreter is required by an interpreted program every time.
5	In the process of compilation the program is analyzed only once and then the code is generated.	5	The source program is interpreted every time it is to be executed and every time the source program is analyzed.

Specification of tokens

There are three specifications of tokens:

1. Strings
2. Language
3. Regular expression

Strings and Languages

An alphabet or character class is a finite set of symbols. A string over an alphabet is a finite sequence of symbols drawn from that alphabet. A language is any countable set of strings over some fixed alphabet.

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

Operations on strings

The following string-related terms are commonly used:

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

Operations on languages

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

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

The following example shows the operations on strings:

Let, L = {0, 1} and S = {a, b, c}.

=>

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

Lecture V

Regular Expressions

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

1. ε is a regular expression, and L(ε) is {ε}, that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a' in its one position.
3. Suppose, r and s are regular expressions denoting the languages L(r) and L(s). Then,
   - ✓ (r)|(s) is a regular expression denoting the language L(r) U L(s).
   - ✓ (r)(s) is a regular expression denoting the language L(r)L(s).
   - ✓ (r)* is a regular expression denoting (L(r))*.
   - ✓ (r) is a regular expression denoting L(r).
4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative. has lowest precedence and is left associative.

Regular Definitions

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example:

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

Shorthand

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

One or more instances (+)

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

Zero or one instance (?)

- ✓ The unary postfix operator ? means "zero or one instance of".
- ✓ The notation r? is a shorthand for r | ε.

- ✓ If ‘r’ is a regular expression, then (r)? is a regular expression that denotes the language L(r) U {ε}.

<u>Character Classes</u>

- ✓ The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- ✓ Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- ✓ We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*.

Non-regular Set

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

Recognition of tokens

Consider the following grammar fragment:

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

expr → term relop term |term term → id |num

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

If → if

then → then

else → else

relop → <|<=|=|<>|>|>=

id → letter(letter|digit)*

num → digit+ (.digit+)?(E(+|-)?digit+)?

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

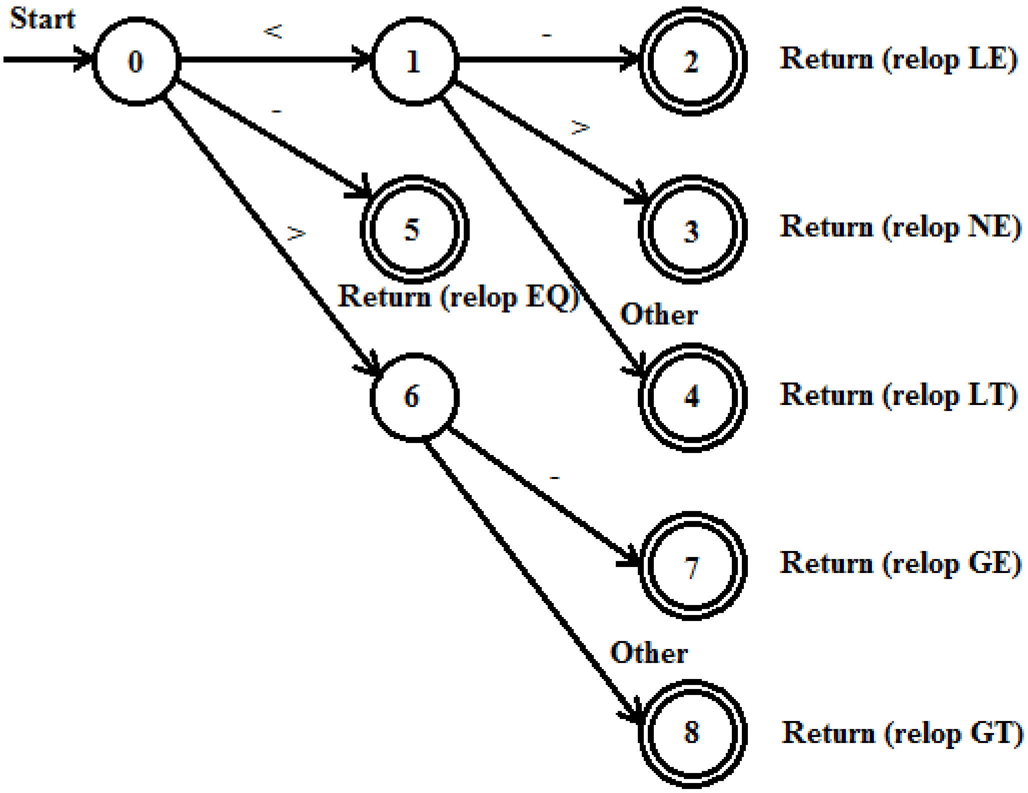
Lecture VI

Transition Diagram

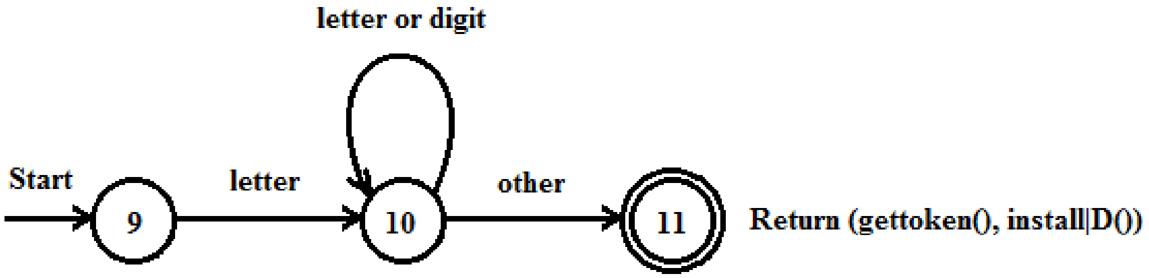
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are as follows:

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



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



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

Automata

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

- ✓ An automation in which the output depends only on the input is called automation without memory.
- ✓ An automation in which the output depends on the input and state also is called as automation with memory.
- ✓ An automation in which the output depends only on the state of the machine is called a Moore machine.
- ✓ An automation in which the output depends on the state and input at any instant of time is called a mealy machine.

Description of Automata

1. An automata has a mechanism to read input from input tape.
2. Any language is recognized by some automation, Hence these automation are basically language ‘acceptors’ or ‘language recognizers’.

Types of Finite Automata

1. Deterministic Automata
2. Non-Deterministic Automata

Deterministic Automata

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which

1. It has no transitions on input €
2. Each input symbol has at most one transition from any state.

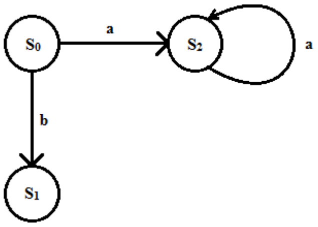
DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where Q is a finite ‘set of states’, which is non empty.

- ✓ ∑ is ‘input alphabets’, indicates input set.
- ✓ qo is an ‘initial state’ and qo is in Q ie, qo, ∑, Q F is a set of ‘Final states’,
- ✓ δ is a ‘transmission function’ or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S_0 for input 'a' there is only one path going to S_2. Similarly from S_0 there is only one path for input going to S_1.

Nondeterministic Automata

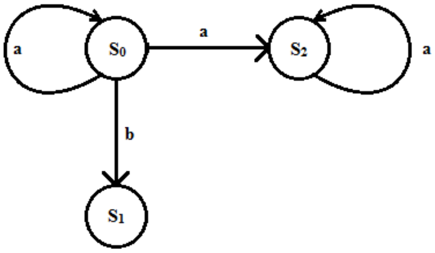
A NFA is a mathematical model consists of

- ✓ A set of states S.
- ✓ A set of input symbols ∑.
- ✓ A transition is a move from one state to another.
- ✓ A state so that is distinguished as the start (or initial) state
- ✓ A set of states F distinguished as accepting (or final) state.
- ✓ A number of transition to a single symbol.

A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as input symbols.

The transition graph for an NFA that recognizes the language (a|b)*abb is shown.



Lecture VII

Bootstrapping

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The bootstrap itself begins at address O in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all of the object code from device has been loaded, the bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

Loaders are of two types:

1. Linking loader.
2. Linkage editor.

Linkage loaders, perform all linking and relocation at load time.

Linkage editors, perform linking prior to load time and dynamic linking, in which the linking function is performed at execution time.

A linkage editor performs linking and some relocation; however, the linkaged program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.

Pass and Phases of Translation

Phases are collected into a front end and aback end.

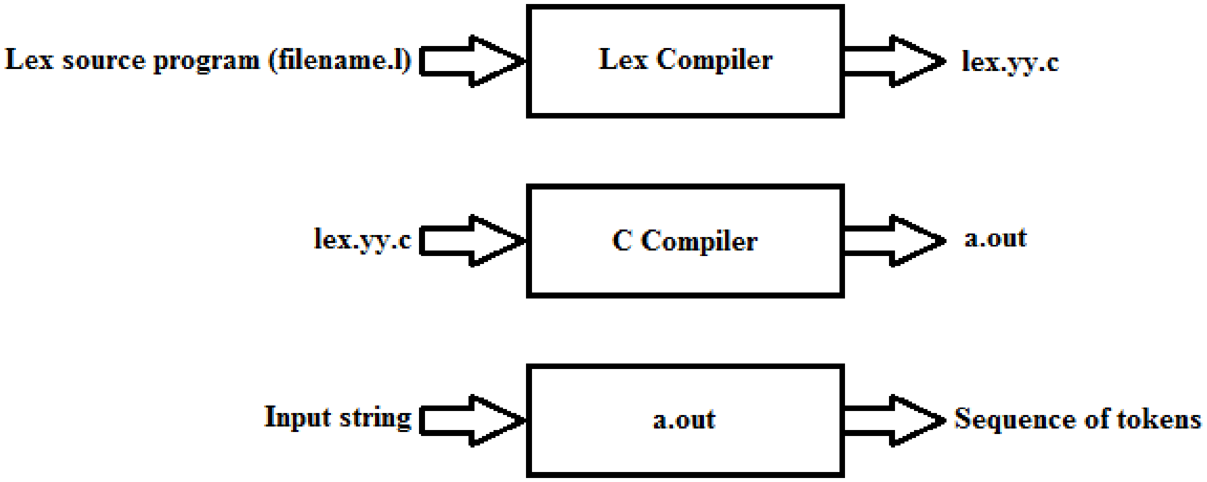
Frontend: The front end consists of those phases, or parts of phase, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by front end as well. The front end also includes the error handling and goes along with each of these phases.

Back end: The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language.

Lexical Analyzer Generator

First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

A Lex program consists of three parts:

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

- ✓ Definitions include declarations of variables, constants, and regular definitions.
- ✓ Rules are statements of the form p1 {action1}p2 {action2} … pn {action} where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.
- ✓ User subroutines are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Input Buffering

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

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. Figure 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.

Multiple Choice Type Questions

1. Which of the following is not a phase of compiler?

a. syntax b. lexiacal **c. testing** d. code generation

2. In compiler, Source program is read by

a. parser **b. lexical analyzer** c. developer d. analyst

3. Source program is read

a. character by character **b. line by line** c. page by page d. module wise

4. Output of lexical analysis phase is

a. token b. parse tree c. code d. object code

5. Compiler

a. reads source program b. converts it into target language **c. both a and b** d. none of these

6. a* means

a. {eppsilon,a,aa,...} b. {a,aa,..} c. {a,aa,aaa,aaa} d. {a,aa,aaa,aaa,...}

7. a+ means

a. {eppsilon,a,aa,...} **b. {a,aa,..}** c. {a,aa,aaa,aaa} d. {a}

8. token for If is

a. Id b. keyword **c. If** d. string

9. Token for "compiler" is

a. keyword b. string c. Id **d. literal**

10. Output of parser is

a. set of tokens **b. parse tree** c. object code d. intermediate code

Short Answer Type Questions

1. Briefly discuss on the cousins of the compiler

2. Illustrate the Analysis-synthesis model of the compiler

3. What is a pattern? Write a pattern over alphabet {x, y, z} containing atleast one 'y' and one 'z'.

4. What do you mean by lexemes and tokens? What are sentinels?

5. What do you mean by DFA and NFA? Illustrate with suitable examples.

Assignments:

1. Explain stages of compilation?10. Consider the following statement

    X=a+b*c

Explain what will be the output at each stages of compilation.

7. Write shorts:

    a) Compiler Vs Interpreter
    b) Linker vs Loader
    c) Lexeme, Tokens and Patterns

Web/ Video Links:

[1] https://www.youtube.com/watch?v=Qkwj65l_96I&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[2] https://www.youtube.com/watch?v=WccZQSERfCM&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=2

Module II

Lecture I

Context-free Grammars

Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where,

- ✓ V is a finite set of variables (or non-terminals). These describe sets of "related" strings.
- ✓ T is a finite set of terminals (i.e., tokens).
- ✓ P is a finite set of productions, each of the form $A \rightarrow \alpha$,
  Where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and non-terminals. $S \in V$ is the start symbol.

Example of CFG

E ==>EAE | (E) | -E | id A==> + | - | * | / |

Where E, A are the non-terminals while id, +, *, -, /,(,) are the terminals.

Syntax Analysis

In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure. This phase is often separated into two phases:

- ✓ **Lexical analysis**: which produces a stream of tokens.
- ✓ **Parsing:** which determines the phrase structure of the program based on the context-free grammar for the language.

Parsing

Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they construct the parse trees:

- ✓ **Top-down:** A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
- ✓ **Bottom-up:** A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

Parse Tree

A parse tree is the graphical representation of the structure of a sentence according to its grammar.

Example

Let the production P is:

E → T | E+T

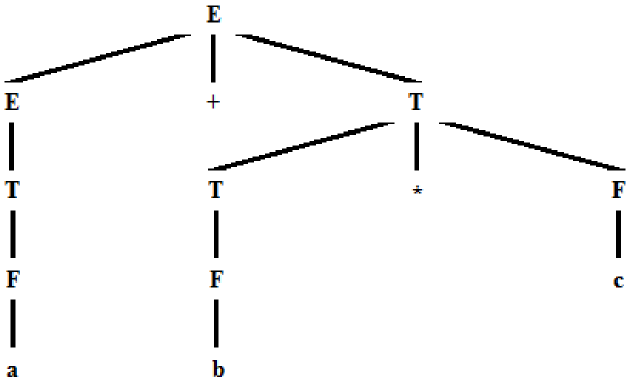
T → F | T*F

F → V | (E)

V → a | b | c |d

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

Parse tree for a + b * c will be as follows:



Syntax Trees

Parse tree can be presented in a simplified form with only the relevant structure information by:

- ✓ Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).
- ✓ Labeling the nodes with the operators in question rather than a non-terminal.

Syntax Error Handling

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors. The programs contain errors at many different levels.

For example, errors can be:

- ✓ Lexical – such as misspelling an identifier, keyword or operator
- ✓ Syntactic – such as an arithmetic expression with un-balanced parentheses.
- ✓ Semantic – such as an operator applied to an incompatible operand.
- ✓ Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered on the syntax analysis phase. The goals of error handler in a parser are:

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

Lecture II

Ambiguity

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parses for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

For example, any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

Left Recursion

If there is any non-terminal A, such that there is a derivation A the A for some string , then grammar is left recursive.

Algorithm for eliminating left Recursion

1. Group all the A productions together like this: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid - - - \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid - - - \mid \beta_n$

   Where,

   A is the left recursive non-terminal,
   α is any string of terminals and
   β is any string of terminals and non-terminals that does not begin with A.

2. Replace the above A productions by the following:

   $A \rightarrow \beta_1 A^I \mid \beta_2 A^I \mid - - - \mid \beta_n A^I$
   $A^I \rightarrow \alpha_1 A^I \mid \alpha_2 A^I \mid - - - \mid \alpha_1 A^I \mid$

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- ✓ This can lead to non-termination in a top-down parser.
- ✓ For a top-down parser, any recursion must be right recursion.
- ✓ We would like to convert the left recursion to right recursion.

Example

Remove the left recursion from the production: $A \rightarrow A\ \alpha \mid \beta$

Applying the transformation yields:
$A \rightarrow \beta\ A^I$
$A^I \rightarrow \alpha A^I \mid \epsilon$

Left Factoring

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

When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

Algorithm

For all A non-terminal, find the longest prefix that occurs in two or more right-hand sides of A. If then replace all of the A productions, A I | 2 | - - - | n | r

With

$A \rightarrow \alpha A^I \mid r$

$A^I \rightarrow \beta_1 | \ \beta_2 | \ - - - \ | \ \beta_n |$

Where, A^I is a new element of non-terminal. Repeat until no common prefixes remain.
It is easy to remove common prefixes by left factoring, creating new non-terminal.

Example

$V \rightarrow \alpha\beta | \ \alpha r$ Change to:

$V \rightarrow \alpha V^I$

$V^I \rightarrow \beta r$

Lecture III

Top down Parsing

Top down parsing is the construction of a Parse tree by starting at start symbol and "guessing" each derivation until we reach a string that matches input. That is, construct tree from root to leaves.

The advantage of top down parsing in that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.

Recursive Descent Parsing

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The special case of recursive –decent parsing, called predictive parsing, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.

Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use.

Recursive descent parser is a top down parser involving backtracking. It makes a repeated scans of the input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

Predictive Parsing

Predictive parsing is top- down parsing without backtracking or look ahead. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

Choose correct α_i by looking at first symbol it derive. If ϵ is an alternative, choose it last.

This approach is also called as predictive parsing. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists and an error is returned.

The crucial property is that, the grammar must not be left-recursive.

Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

For example,

stmt →if exp then stmt else stmt | while expr do stmt | begin stmt-list end.

Then the keywords if, while and begin tell, which alternative is the only one that could possibly succeed if we are to find a statement.

A predictive parser has the following components:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists the string to be parsed, followed by \$, a symbol used as a right end marker to indicate the end of the input string.

The stack consists of a sequence of grammar symbols with \$ on the bottom, indicating the bottom of the stack. Initially the stack consists of the start symbol of the grammar on the top of \$.

Recursive descent and LL parsers are often called predictive parsers, because they operate by predicting the next step in a derivation.

The algorithm for the Predictive Parser Program is as follows: Input: A string w and a parsing table M for grammar G.

Output: if w is in L(g),a leftmost derivation of w; otherwise, an error indication.

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

```
Set ip to point to the first symbol of w$; repeat

let x be the top stack symbol and a the symbol pointed to by ip; if X is a terminal or $, then

    if X = a, then

        pop X from the stack and advance ip else error()

    else    /* X is a non-terminal */

        if M[X, a] = $X \rightarrow Y_1\ Y_2 \ldots\ldots Y_k$, then

            pop X from the stack;

            push Yk, Yk-1, . . . . . . . . . . .Y1 onto the stack, with Y1 on top; output the production $X \rightarrow Y_1\ Y_2 \ldots\ldots Y_k$

        else error();

Until X = $ /*stack is empty*/
```

Lecture IV

FIRST and FOLLOW

The construction of a predictive parser is aided by two functions with a grammar G. these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

Define FOLLOW (A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form S=>αAaβ for some α and β. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Computation of FIRST ()

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or € can be added to any FIRST set.

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

If X→€ is production, then add € to FIRST(X).

If X is nonterminal and X→Y1 Y2……Yk is a production, then place a in FIRST(X) if for some i,a is in FIRST(Yi),and € is in all of FIRST(Yi),and € is in

all of FIRST(Y1),….. FIRST(Yi-1);that is Y1………. Yi-1==>€.if € is in

FIRST(Yj), for all j=,2,3…….k, then add € to FIRST(X).for example, everything

in FIRST(Y1) is surely in FIRST(X).if Y1 does not derive €,then we add nothing

more to FIRST(X),but if Y1=>€,then we add FIRST(Y2) and so on.

FIRST (A) = FIRST (I) U FIRST (2) U - - - U FIRST (n) Where, A 1 | 2 | - - - | n, are all the productions for A. FIRST (A) = if FIRST (A) then FIRST (A)

else (FIRST (A) - { }) U FIRST ( )

pop X from the stack;

push Yk, Yk-1, ...Y1 onto the stack, with Y1 on top; output the production X □ Y1 Y2 . . . . . Yk

end

else error()

until X = $ /*stack is empty*/

Computation of FOLLOW ()

To compute FOLLOW (A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

Place $ in FOLLOW(s), where S is the start symbol and $ is input right end marker .

If there is a production A→αBβ, then everything in FIRST(β) except for € is placed in

FOLLOW(B).

If there is production A→αB, or a production A→αBβ where FIRST (β) contains € (i.e.,β→€),then everything in FOLLOW(A)is in FOLLOW(B).

Lecture V

LL (1) Grammar

The first L stands for “Left- to-right scan of input”. The second L stands for “Left-most derivation”. The ‘1’ stands for “1 token of look ahead”.

No LL (1) grammar can be ambiguous or left recursive.

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table, fill with synch for rest of the input symbols of FOLLOW set and then fill the rest of the columns with error term.

Terminal Variables	+	*	(	)	id	$
E	error	error	ETE	synch	ETE^I	synch
E_I	E^I $+TE^I$	error	error	E^I	error	E^I
T	synch	error	TFT	synch	TFT^I	synch
T_I	T^I	T^I*F	error	T^I	error	T^I
F	synch	synch	F(E)	synch	Fid	synch

If the parser looks up entry in the table as synch, then the non-terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

STACK	IN	REMARKS
$\$\ \mathbf{E}$	$\mathbf{)}$ id * +	Error, skip)
$\$\ \mathbf{E}$	**id** * +	
$\$\ E^{I}\ \mathbf{T}$	**id** * +	
$\$\ E^{I}\ T^{I}\ \mathbf{F}$	**id** * +	
$\$\ E^{I}\ T^{I}\ \mathbf{id}$	**id** * +	
$\$\ E^{I}\ \mathbf{T}^{I}$	$\mathbf{*}$ +	
$\$\ E^{I}\ T^{I}\ F\ *$	$\mathbf{*}$ +	
$\$\ E^{I}\ T^{I}\ \mathbf{F}$	$\mathbf{+}$	Error; F on + is synch; F has been popped.
$\$\ E^{I}\ \mathbf{T}^{I}$	+	
$\$\ \mathbf{E}^{I}$	+	
$\$\ E^{I}\ T\ \mathbf{+}$	$\mathbf{+}$	
$\$\ E^{I}\ \mathbf{T}$		
$\$\ E^{I}\ T^{I}\ \mathbf{F}$		
$\$\ E^{I}\ T^{I}\ \mathbf{id}$		
$\$\ E^{I}\ \mathbf{T}^{I}$		
$\$\ \mathbf{E}^{I}$		
$\mathbf{\$}$		Accept.

Lecture VI

Bottom up Parsing

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

$S\rightarrow r_0\rightarrow r_1\rightarrow r_2$ - - - $r_{n-1}\rightarrow r_n$ sentence

←

Top-down vs. Bottom-up parsing

Top-down	Bottom-up
1. Construct tree from root to leaves	1. Construct tree from leaves to root
2. “Guers” which RHS to substitute for Nonterminal	2. “Guers” which rule to “reduce” terminals
3. Produces left-most derivation	3. Produces reverse right-most derivation.
4. Recursive descent, LL parsers	4. Shift-reduce, LR, LALR, etc.
5. Recursive descent, LL parsers	5. “Harder” for humans.
6. Easy for humans	

Handles

Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

For example:

abbcde

aAbcde $A\rightarrow b$

aAAcde $A\rightarrow b$

struck

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

If the grammar is unambiguous, every right sentential form has exactly one handle.

More formally, a handle is a production A→β and a position in the current right-sentential form such that:

$S \rightarrow \alpha A$

$\omega \rightarrow \alpha / \beta \omega$

For example grammar, if current right-sentential form is a/Abcde

Then the handle is A   Ab at the marked position. ‘a’ never contains non-terminals.

Handle Pruning

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example

$E \rightarrow E+E \,/\, E*E \,/\, (E) \,/\, id$

Right-sentential form	Handle	Reducing production
a+b*c	a	E id
E+b*c	b	E id
E+E*C	C	E id
E+E*E	E*E	E E*E
E+E	E+E	E E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us.

Lecture VII

Shift- Reduce Parsing

Shift Reduce Parsing uses a stuck to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the state.

A shift-reduce parser has just four actions:

1. Shift-next word is shifted onto the stack (input symbols) until a handle is formed.
2. Reduce – right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off stack and push appropriate LHS.
3. Accept – stop parsing on successful completion of parse and report success.
4. Error – call an error reporting/recovery routine.

Possible Conflicts

Ambiguous grammars lead to parsing conflicts.

1. Shift-reduce: Both a shift action and a reduce action are possible in the same state (should we shift or reduce)
2. Reduce-reduce: Two or more distinct reduce actions are possible in the same state. (Which production should we reduce with 2).

Operator – Precedence Parsing

Precedence/ Operator grammar: The grammars having the property:

- ✓ No production right side is should contain ϵ.
- ✓ No production right side should contain two adjacent non-terminals.

Operator – precedence parsing has three disjoint precedence relations, <.,=and .> between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
a<.b	'a' yields precedence to 'b'.
a=b	'a' has the same precedence 'b'
a.>b	'a' takes precedence over 'b'.

Operator precedence parsing has a number of disadvantages:

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

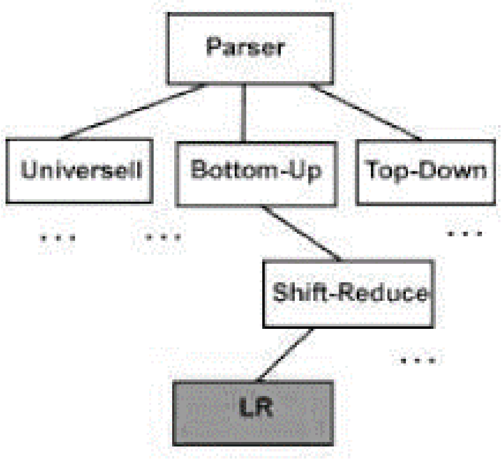
Disadvantages:

- ✓ L(G) ≠ L(parser)
- ✓ Error detection
- ✓ Usage is limited
- ✓ They are easy to analyze manually

Lecture VIII

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.

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.

Consider the following Grammar

S' -> S

S -> CC

C -> aC

C -> d

LR (1) Parser

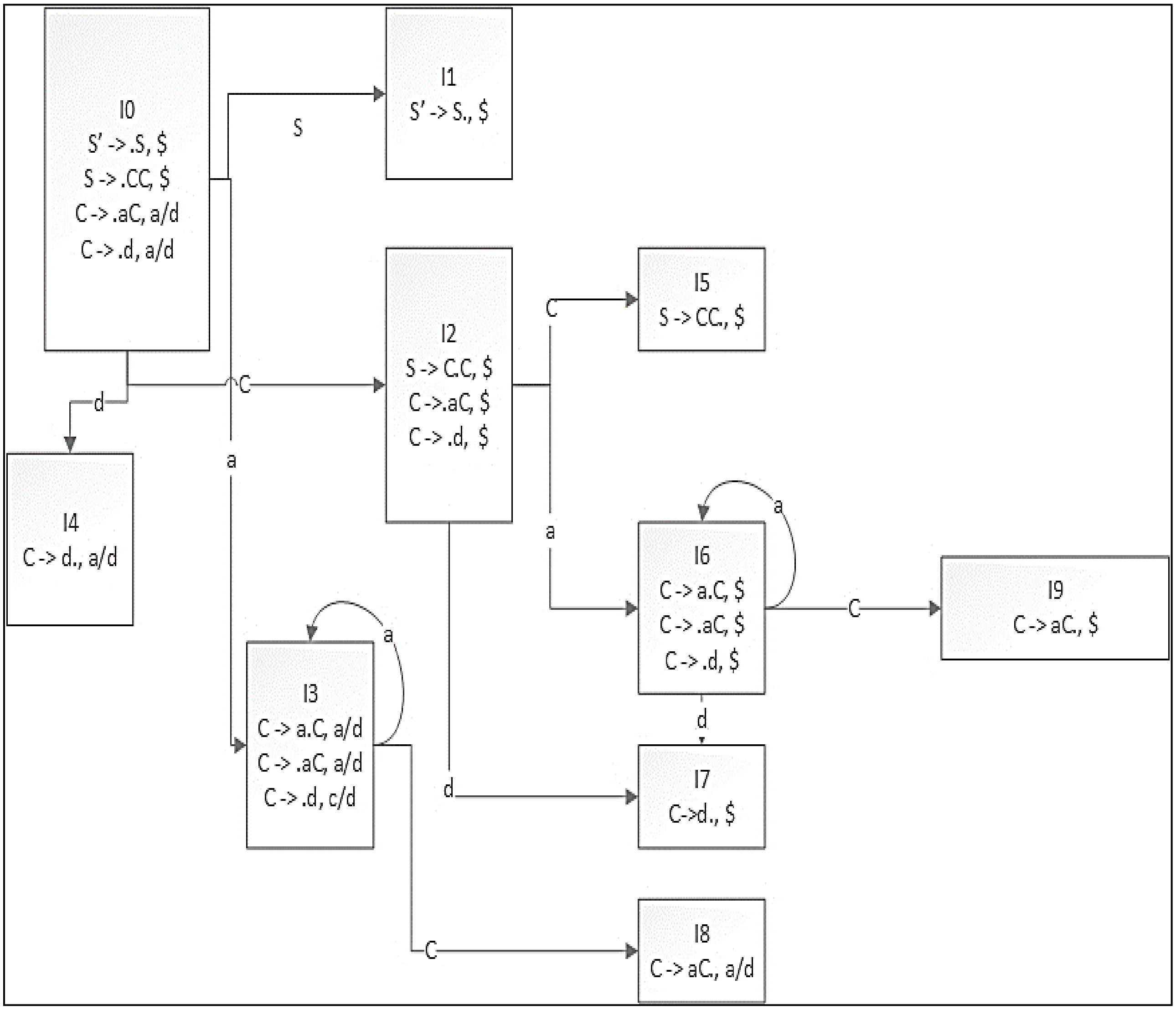
Our task is to construct the LR(1) sets of items for the above grammar. It can be done using the following 3 procedures which are interdependent on each other:

```
Set_Of_Items CLOSURE(I)

{

     repeat

           for ( each item [A -> α.Bβ, α] in I )

               for ( each production B ->  γ in G' )

                   for ( each terminal b in FIRST(βα) )

                        add [B -> .γ, b] to set I;

    until no more items are added to I;

    return I;

}


Set_Of_Items GOTO(I,X)

{

    initialize J to be the empty set;

    for ( each item [A -> α.Xβ, a] in I )

         add item [A -> αX.β, α] to set J;

    return CLOSURE(J);

}


void items(G’)

{

     initialize C to CLOSURE({[S'  -> .S, $]});

     repeat

          for ( each set of items I in C )
```

```
        for ( each grammar symbol X )
            if ( GOTO(I, X) is not empty and not in C )
                add GOTO(I,X) to C;
until no new sets of items are added to C;
}
```

The first step is to construct the augmented grammer G', then invoke items, which inturn will invoke GOTO and CLOSURE in order to construct the item set.

The following is the LR(1) set of items construction represented graphically:



The following is the Action and the Goto function table as seen from the above Graph:

States	a	d	$	S	C
0	s3	s4		1	2
1			accepted		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Clearly, the language of the Grammar is: a*da*d

Now we parse a string and show its implementation using 2 stacks.

Consider the following adad.

The following is the parsing table achieved via implementation of two stacks.

States	Symbol	Input	Action
0	$	adad$	Shift S3
03	$a	dad$	Shift S4
034	$ad	ad$	Reduce C -> d
038	$aC	ad$	Reduce C -> aC
02	$C	ad$	Shift 6
026	$Ca	d$	Shift 7
0267	$Cad	$	Reduce C -> d
0269	$CaC	$	Reduce C -> aC
025	$CC	$	Reduce S -> CC
01	$S	$	accepted

Lecture IX

LALR Parser

The LALR parsing table can be constructed using the LR table by implementing the following procedure:

1. Construct C = (I_0, I_1, ….. I_N,), the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let C' = {J_0, J_1,. . . , J_N,) be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as done for LR(1). If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

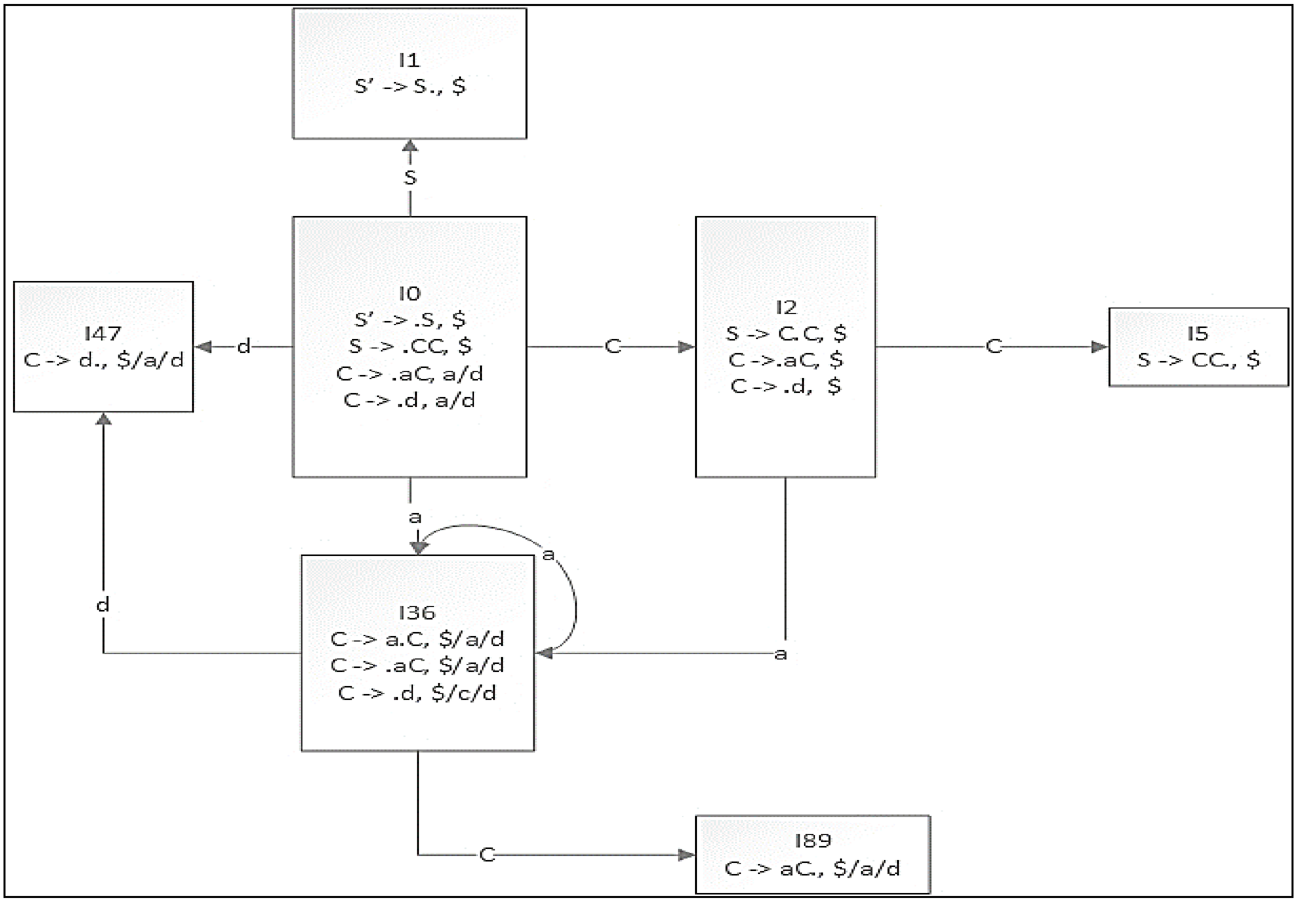
4. The GOTO table is constructed as follows. If J is the union of one or

more sets of LR(1) items, that is, J = $I_l \cap I_2 \cap$…. $\cap I_K$, then the cores of GOTO(I_1, X) , GOTO(I_2, X) , . . . , GOTO(I_k, X) are the same, since 1_1, 1_2, . . . , I_k all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X). Then GOTO(J, X) = K.

The following is the LALR ACTION and GOTO table construction for the same Grammar as used in LR(1) table construction.

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47		1	2
1			Accepted		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The following is the LALR item set construction as seen graphically:


I1
S' -> S., $
S
I47
C -> d., $/a/d
d
I0
S' -> .S, $
S -> .CC, $
C -> .aC, a/d
C -> .d, a/d
C
I2
S -> C.C, $
C ->.aC, $
C -> .d, $
C
I5
S -> CC., $
a
a
d
I36
C -> a.C, $/a/d
C -> .aC, $/a/d
C -> .d, $/c/d
a
C
I89
C -> aC., $/a/d

Lecture X

CLR Parser

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So , the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item.

LR(1) parsers are more powerful parser.

For LR(1) items we modify the Closure and GOTO function.

Closure Operation

```
Closure(I)

repeat

  for (each item [ A -> ?.B?, a ] in I )

    for (each production B -> ? in G')

      for (each terminal b in FIRST(?a))

        add [ B -> .? , b ] to set I;

until no more items are added to I;

return I;
```

Goto Operation

```
Goto (I, X)

Initialise J to be the empty set;

for ( each item A -> ?.X?, a ] in I )

  Add item A -> ?X.?, a ] to se J;   /* move the dot one step */

return Closure(J);    /* apply closure to the set */
```

LR (1) items

```
Void items(G')

Initialise C to { closure ({[S' -> .S, $]})};

Repeat

  For (each set of items I in C)
```

```
    For (each grammar symbol X)
        if( GOTO(I, X) is not empty and not in C)
            Add GOTO(I, X) to C;
Until no new set of items are added to C;
```

<u>Construction of GOTO graph</u>

State I_0 – closure of augmented LR(1) item.

Using I_0 find all collection of sets of LR(1) items with the help of DFA

Convert DFA to LR(1) parsing table

<u>Construction of CLR parsing table</u>

Input – augmented grammar G'

Construct C = { I_0, I_1, ……. I_n} , the collection of sets of LR(0) items for G'.

State i is constructed from I_i. The parsing actions for state i are determined as follows:

i) If [A -> ?.a?, b] is in I_i and GOTO(Ii , a) = I_j, then set ACTION[i, a] to "shift j". Here a must be terminal.

ii) If [A -> ?. , a] is in I_i , A ≠ S, then set ACTION[i, a] to "reduce A -> ?".

iii) Is [S -> S. , $] is in I_i, then set action[i, $] to "accept".

If any conflicting actions are generated by the above rules we say that the grammar is

not CLR.

The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(Ii, A) = I_j then GOTO [i, A] = j.

All entries not defined by rules 2 and 3 are made error.

Error Recovery

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

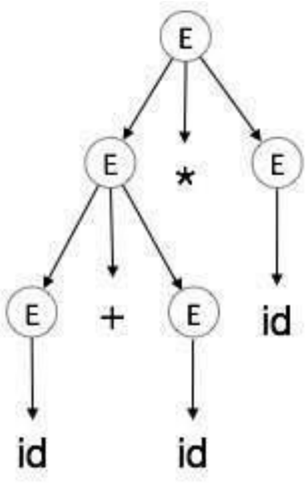
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

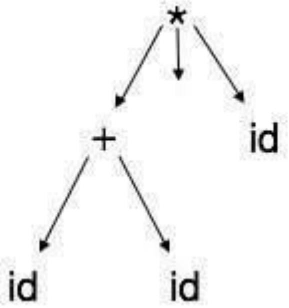
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

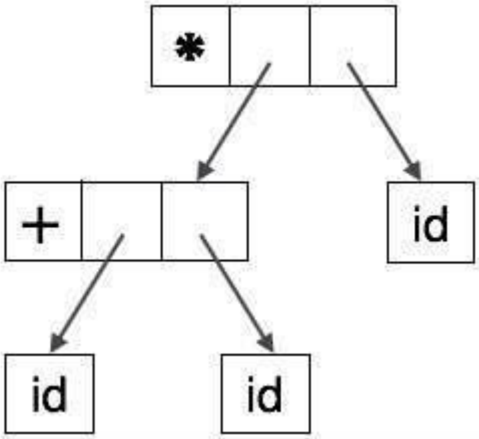
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

Semantic Analysis

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

```
E → E + T
```

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

```
CFG + semantic rules = Syntax Directed Definitions
```

For example:

```
int a = “value”;
```

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking
-

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

```
S → ABC
```

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

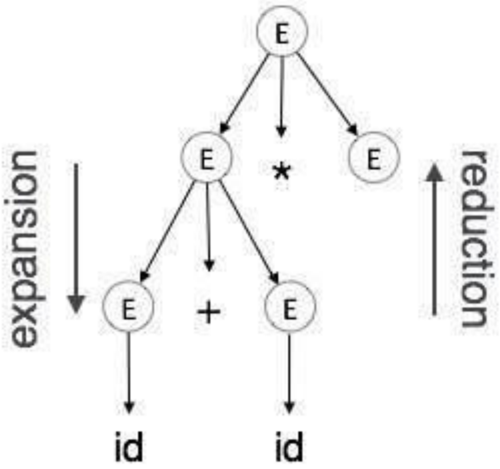
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → ABC
```

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

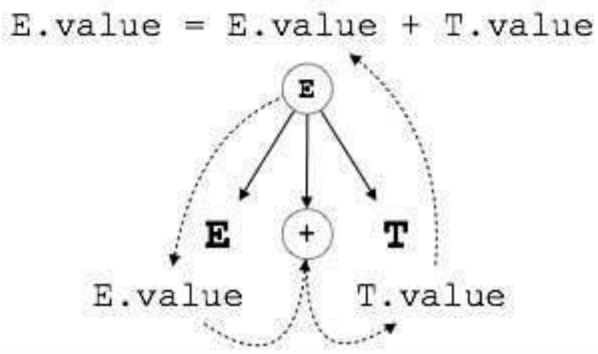
For example:

```
int value  = 5;
<type, “integer”>
<presentvalue, “5”>
```

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

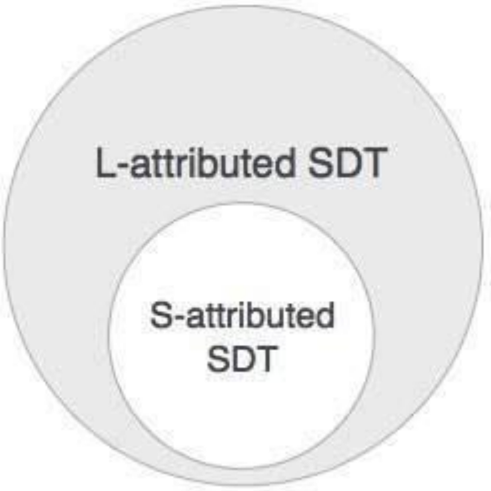
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Multiple Choice Type Questions

1. The concept of grammar is much used in this part of the compiler
   A. lexical analysis
   B. parser
   C. code generation
   D. code optimization
   ANSWER: B

2. Parsing is also known as
   A. syntax anlysis
   B. lexical analysis
   C. semantic analysis
   D. code generation
   ANSWER: A

3. Input buffer is
   A. symbol table
   B. divided into two halves
   C. divided into Three halves
   D. not divided
   ANSWER: B

4. Which symbols are used as synchronizing tokens in predictive parsing?
   a. FIRST symbols
   b. FOLLOW symbols
   c. Both of these
   d. None of these

5. The most powerful parser is
   a. SLR
   b. LALR
   c. CLR
   d. Operator precedence

6.shift reduce parser is
A. Bottom up parsing
B. Top down parsing
C. Mixture of both
D. none
ANSWER: D

7.which of the following is graph represation of derivation?
A. Parse tree
B. Oct tree
C. Binary tree
D. None
ANSWER: D

8.Which of the following is most powerful parsing ?
A. SLR
B. CLR
C. LALR
D. None
ANSWER: C

9. Grammer of the programming is checked at the phase of the compiler
A. Lexical
B. Syntax
C. Intermediate
D. Sematic
ANSWER: B

10.Top down generates
A. Leftmost derivation
B. Right most derivation
C. Right derivation in reverse
D. None
ANSWER: B

11.Bottom up parsing generates
A. LLeftmost derivation
B. Right most derivation
C. Right derivation in reverse
D. None
ANSWER: A

12.LL signifies
A. Left to right scanning and Left most derivation
B. Right to left Scanning and Left most derivation
C. Left to right scanning and Left most derivation
D. None
ANSWER: A

13.LR signifies
A. Left to right scanning and Left most derivation
B. Right to left Scanning and Left most derivation
C. Left to right scanning and Left most derivation
D. None
ANSWER: A

14.Top down parser generates ………………………
A. left-most derivation
B. right-most derivation
C. left-most derivation in reverse
D. right-most derivation in reverse
ANSWER: A

15.If x is a terminal then FIRST(x) will be ……………
A. ?
B. {x}
C. x
D. none of these
ANSWER: B

16.Consider the grammar: S-> AaAb|BbBa, A-> ? , B-> ? , Check the above grammar is LL(1) or not?
A. Yes LL(1)
B. No, It is not LL(1)
C. Enough information is not given.
D. None of these
ANSWER: A

17.Consider the grammar:Find out the FOLLOW(A. S-> AaAb|BbBa, A-> ? , B-> ?
A. {a}
B. {b}
C. {a,b}
D. {a,b,$}
ANSWER: C

18.Consider the grammar: S-> AaAb|BbBa, A-> ? , B-> ? , Find out the FOLLOW(B. .
A. {a}
B. {b}
C. {a,b}
D. {a,b,$}
ANSWER: C

19.Top down parser generates ………………………
A. left-most derivation
B. right-most derivation
C. left-most derivation in reverse
D. right-most derivation in reverse
ANSWER: A

20.If x is a terminal then FIRST(x) will be ……………

A. ?

B. {x}

C. x

D. none of these

ANSWER: B

21.The grammar E->E+E|E*E|id is

A. Ambiguous

B. unambiguous

C. not given sufficient information

D. none of these

ANSWER: A

Short Answer Type Questions

1. Briefly discuss on the classification of the parsers.
2. What are the condition a grammer to be LL(1)?
3. What is handle?
4. Consider the grammer E→E+E|E*E|id

   find the handles of the right sentential forms of reduction for the string id+id*id.

5. What do you mean by augmented grammar?
6. What do you mean by LR(0) items?
7. Consider the following grammar

   E→E+T|T

   T→T*F|F

   F→id

   Find out the closure and goto[I.T].

Assignments:

1. Consider the grammer:

   S->aABb

   A->c| Є

   B->d| Є

a) Find out the FOLLOW(A) and FOLLOW(B) (b) Draw the parse table.

(c) String acdb is accepted or not?

2. Draw the Parse Table for the following grammar:

   S → AaAb|BbBa

A → ϵ

B → ϵ

3. Construct the Predictive Parsing Table for the following grammar:

E → TE’

E’ → TE’ | ϵ

T → FT’

T’ → *FT | ϵ

F → (E) | id

4. Enlighten on the Common Prefix Problem and Backtracking

5. Give a suitable example of such a grammar which is not LL (1) compatible. Explain why.

8. Construct an SLR(1) parsing table for the following grammer:

S→xAy|xBy|xAz

A→qS|q

B→q

Web/ Video Link:

[1]. https://www.youtube.com/watch?v=9vmhcBpZDcE&index=3&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[2] https://www.youtube.com/watch?v=3_VCoBfrt9c&index=4&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[3] https://www.youtube.com/watch?v=N9UuAPU6DAg&index=5&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[4] https://www.youtube.com/watch?v=_uSlP91jmTM&index=6&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[5] https://www.youtube.com/watch?v=R1ZlWEZWMKk&index=7&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[6] https://www.youtube.com/watch?v=SH5F-rwWEog&index=8&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[7] https://www.youtube.com/watch?v=n5UWAaw_byw&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=9

[8] https://www.youtube.com/watch?v=APJ_Eh60Qwo&index=10&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[9] https://www.youtube.com/watch?v=0kiTNN2kHyY&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=11

[10] https://www.youtube.com/watch?v=MIg2ymmMn4k&index=12&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[11] https://www.youtube.com/watch?v=5s4CWn6GiwY&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=13

[12] https://www.youtube.com/watch?v=VSkfnRfNuwI&index=14&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS

[13] https://www.youtube.com/watch?v=queUceGJqh0&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=17

[14] https://www.youtube.com/watch?v=rdnAJBoFKOw&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS&index=19

Module III

LECTURE 1

Type systems

SEMANTIC CHECKING. A compiler must perform many semantic checks on a source program.

- Many of these are straightforward and are usually done in conjunction with syntax-analysis.
- Checks done during compilation are called *static checks* to distinguish from the *dynamic checks* done during the execution of the target program.
- Static checks include:

  Type checks.

  The compiler checks that names and values are used in accordance with type rules of the language.

  Type conversions.

  Detection of implicit type conversions.

  Dereferencing checks.

  The compiler checks that dereferencing is applied only to a pointer.

  Indexing checks.

  The compiler checks that indexing is applied only to an array.

  Function call checks.

  The compiler checks that a function (or procedure) is applied to the correct number and type of arguments.

  Uniqueness checks.

  In many cases an identifier must be declared exactly once.

  Flow-of-control checks.

  The compiler checks that if a statement causes the flow of control to leave a construct, then there is a place to transfer this flow. For instance when using **break** in C.

In this chapter we will focus on type checking and type conversions. Most of the other static checks are easy to implement.

TYPE INFORMATION may be needed for the code generation. For instance, for the translation of *overloaded symbols* like +. Indeed, overloading may be accompanied by coercion of types, where a compiler supplies an operator to convert an operand into the type expected by the context.

OVERLOADED SYMBOLS AND POLYMORPHIC FUNCTIONS. An *overloaded symbol* is a symbol that represent different operations in different contexts. This is different from a *polymorphic function* which is a function whose body can be executed with arguments of several types.

TYPES HAVE STRUCTURE. Generally types are either basic or structured.

- Basic types are the atomic types with no internal structure. For instance, in C: char, int, double, float, etc.
- Structured types are the types of constructs like arrays, sets, (typed) pointers, structs, classes, and functions.

TYPE EQUIVALENCE. Because of overloaded symbols and structured types, we will be concerned with the non trivial problem of type equivalence.

- That is, given two language constructs, decide whether they have the same type.
- We will use the notion of a *type expression*.

TYPE EXPRESSION. The idea is to associate each language construct with an expression describing its type and so called *type expression.* Type expressions are defined inductively from basic types and constants using *type constructors.* Here are the type constructors that we shall consider in the remaining of this chapter. They are close to type constructors of languages like C or PASCAL.

Basic types.

char, int, double, float are type expressions.

Type names.

It is convenient to consider that names of types are type expressions.

Arrays.

If T is a type expression and n is an int then

$$\mathrm{array}(n, T)$$

is a type expression denoting the type of an array with elements in T and indicies in the range $0 \cdots n - 1$.

Products.

If T_1 and T_2 are type expressions then

$$T_1 \times T_2$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. We assume that products of more than two types are defined in a similar way and that this product-operation is left-associative. Hence

$$(T_1 \times T_2) \times T_3 \text{ and } T_1 \times T_2 \times T_3$$

are equivalent.

Records.

If $name_1$ and $name_2$ are two type names, if T_1 and T_2 are type expressions then

$$\mathrm{record}(name_1 \colon T_1, name_2 \colon T_2)$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. The only difference between a record and a product is that the fields of a record have names.

Pointers.

If T is a type expression, then

$$\mathrm{pointer}(T)$$

is a type expression denoting the type *pointer to an object of type T*.

Function types.

If T_1 and T_2 are type expressions then

$$T_1 \longrightarrow T_2$$

is a type expression denoting the type of the functions associating an element from T_1 with an element from T_2.

GRAPHICAL REPRESENTATIONS FOR TYPE EXPRESSIONS. Let Σ_{val} be the alphabet consisting of

- the basic types,
- the possible values of an array index, say $\mathbb{N}$,
- the possible names for a type.

Let Σ_{op} be the alphabet consisting of the keywords array, ×, record, pointer and $\rightarrow$. Then it is not hard to see that the set of type expressions can be viewed as a language of expressions in the sense of the section about syntax trees in the chapter dedicated to syntax-directed definitions. Indeed we could write

(array n T), (record $name_1$ T_1 $name_2$ T_2) and (pointer T)

instead of

array(n, T), record($name_1$: T_1, $name_2$: T_2) and pointer(T).

That way we would satisfy the definition of a language of (arithmetic) expressions. It follows that we can use syntax trees and DAGs to represent type expressions graphically.

TYPE SYSTEMS. A *type system* is a set of rules assigning type expressions to different parts of the program.

- Type systems can (usually) be implemented in a syntax-directed way.
- The implementation of a type system is called a type checker.

A LANGUAGE IS STRONGLY TYPED if the compiler can guarantee that the accepted programs will **execute without type errors.** Lisp languages are usually weakly typed:

```
CannaLisp listener 3.5 beta 2
-> (defun f (l) (car l))
f
-> (f '(+ 1 2))
+
-> (f (+ 1 2))
Bad arg to car 3
```

LECTURE 2

SPECIFICATION OF A SIMPLE TYPE CHECKER

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

A Simple Language

Consider the following grammar:

P → D ; E

D → D ; D | id : T

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

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

Translation scheme:

P → D ; E

D → D ; D

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

T → char { T.type : = char }

T → integer { T.type : = integer }

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

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

In the above language,

→ There are two basic types : char and integer ; → type_error is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression

pointer (integer).

Type checking of expressions

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

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

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

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

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

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

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

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

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

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

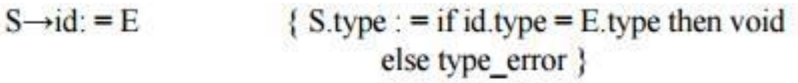
The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t of the object pointed to by the pointer E.

Type checking of statements

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

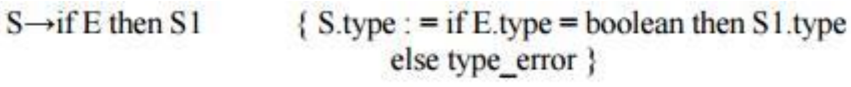
Translation scheme for checking the type of statements:

1. Assignment statement: S→id: = E



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

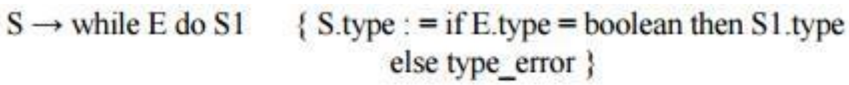
2. Conditional statement: S→if E then S1



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

3. While statement:

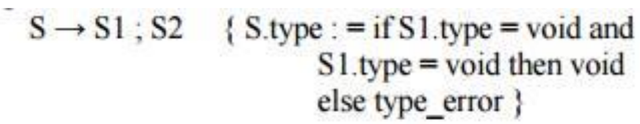
S → while E do S1



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

4. Sequence of statements:

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



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

5.Type checking of functions

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

E1.type = s → t then t else type_error }

LECTURE 3

Type conversions Source language issues:

1.Activation Trees

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

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

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

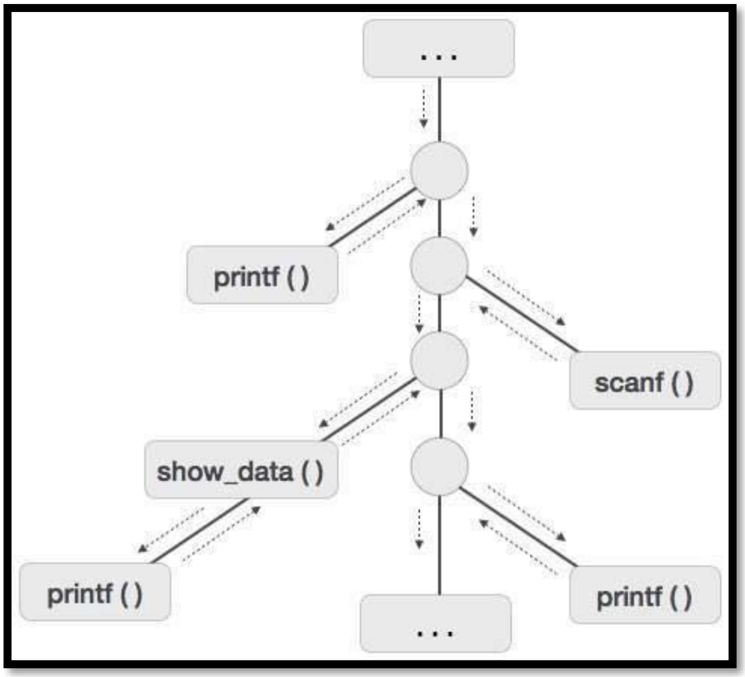
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is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

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

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

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

Below is the activation tree of the code given.



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

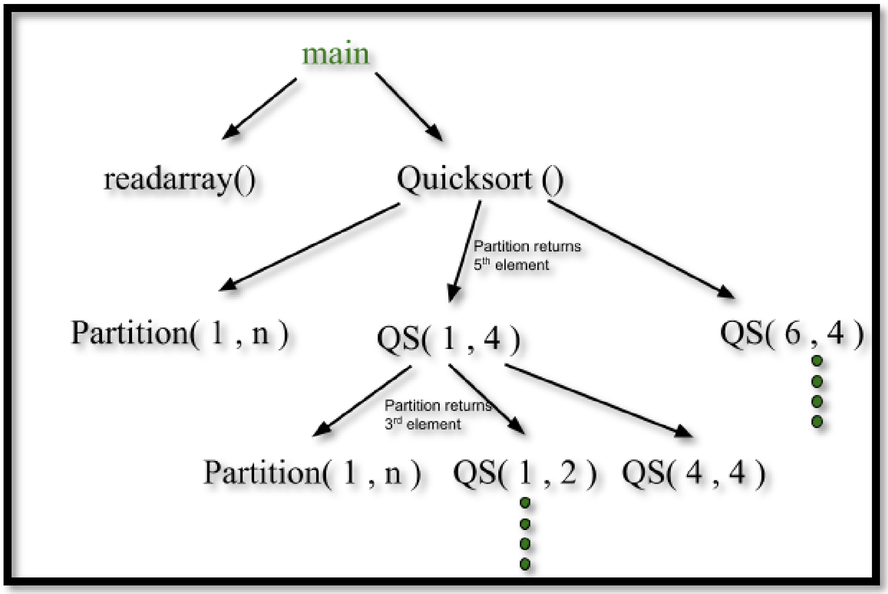
2.Control stack:

Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed. A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends). Information needed by a single execution of a procedure is managed using an activation record or frame. When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.

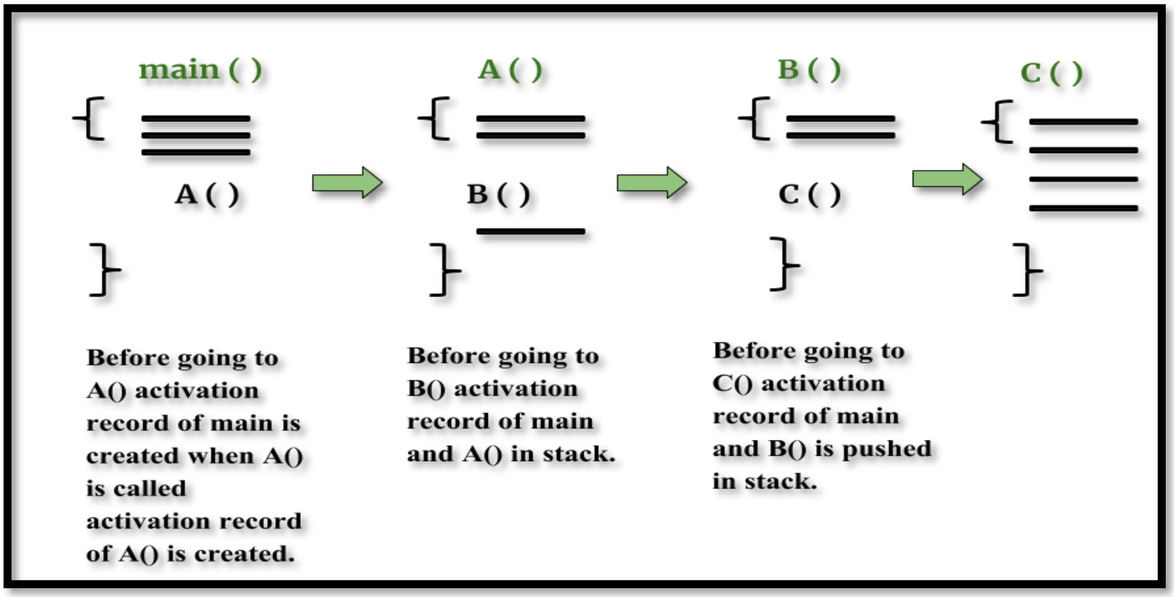
Example – Consider the following program of Quicksort

```
main() {

       Int n;
       readarray();
       quicksort(1,n);
}
quicksort(int m, int n) {

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

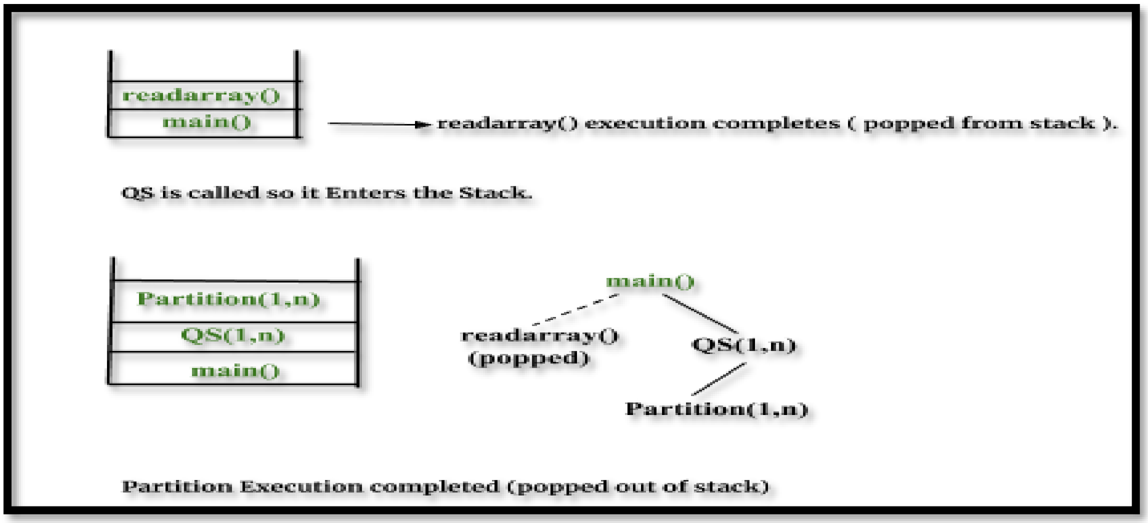
The activation tree for this program will be:



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


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

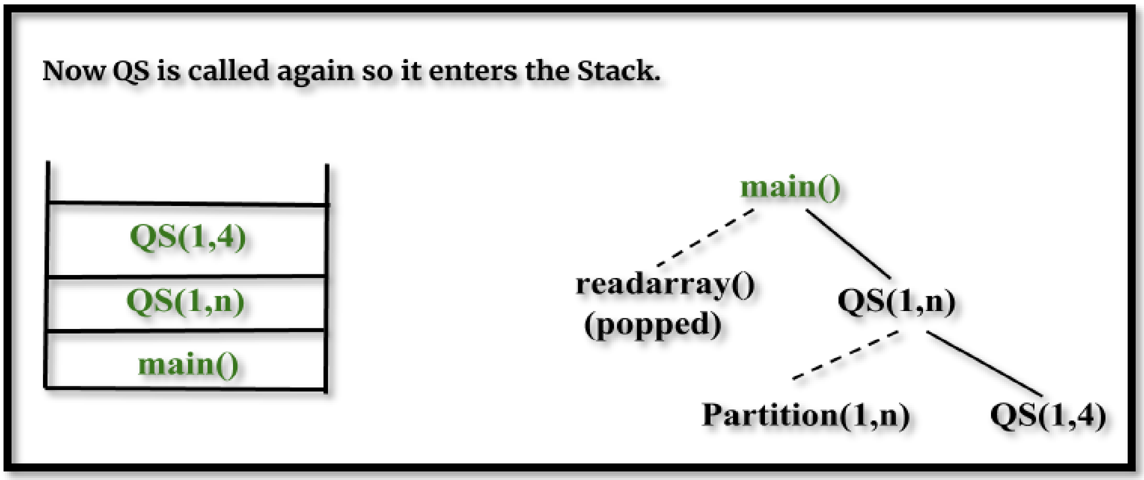
Control stack for the above quicksort example:



readarray() execution completes (popped from stack).

QS is called so it Enters the Stack.

Partition Execution completed (popped out of stack)



Now QS is called again so it enters the Stack.

LECTURE 4

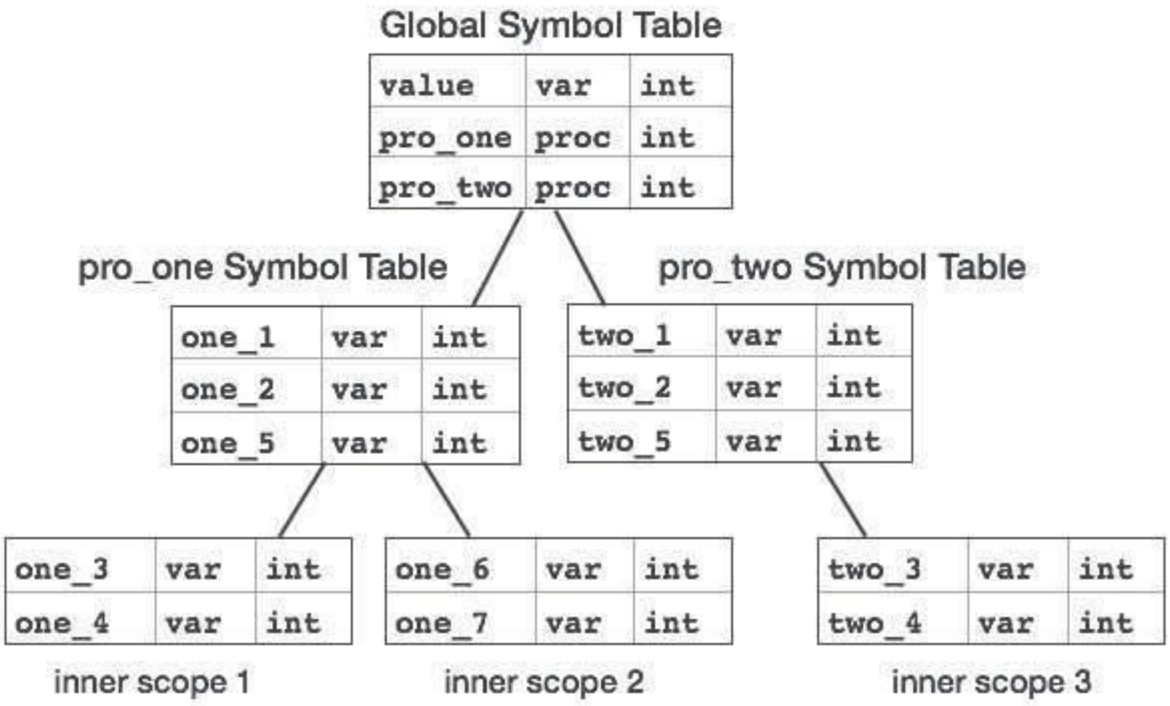
3.scope of declaration:

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

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

```
. . .
int value=10;

void pro_one()
   {
   int one_1;
   int one_2;

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

   int one_5;

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

void pro_two()
   {
   int two_1;
   int two_2;

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

   int two_5;
   }
. . .
```

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



Global Symbol Table

value	var	int
pro_one	proc	int
pro_two	proc	int

pro_one Symbol Table

one_1	var	int
one_2	var	int
one_5	var	int

pro_two Symbol Table

two_1	var	int
two_2	var	int
two_5	var	int

one_3	var	int
one_4	var	int

inner scope 1

one_6	var	int
one_7	var	int

inner scope 2

two_3	var	int
two_4	var	int

inner scope 3

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

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

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

LECTURE 5

4.Binding of names:

Parameter Passing

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

r-value

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

l-value

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

For example:

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

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

For example:

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

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

Formal Parameters

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

Actual Parameters

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

Example:

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

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

Pass by Value

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

Pass by Reference

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

Pass by Copy-restore

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

Example:

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

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

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

Pass by Name

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

LECTURE 6

Symbol tables:

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

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

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

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

```
<symbol name,  type,  attribute>
```

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

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

Implementation

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

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

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

Operations

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

insert()

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

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

For example:

```
int a;
```

should be processed by the compiler as:

```
insert(a, int);
```

lookup()

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

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

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

```
lookup(symbol)
```

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

LECTURE 7

Dynamic storage allocation techniques:

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

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

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

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

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

Static Allocation

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

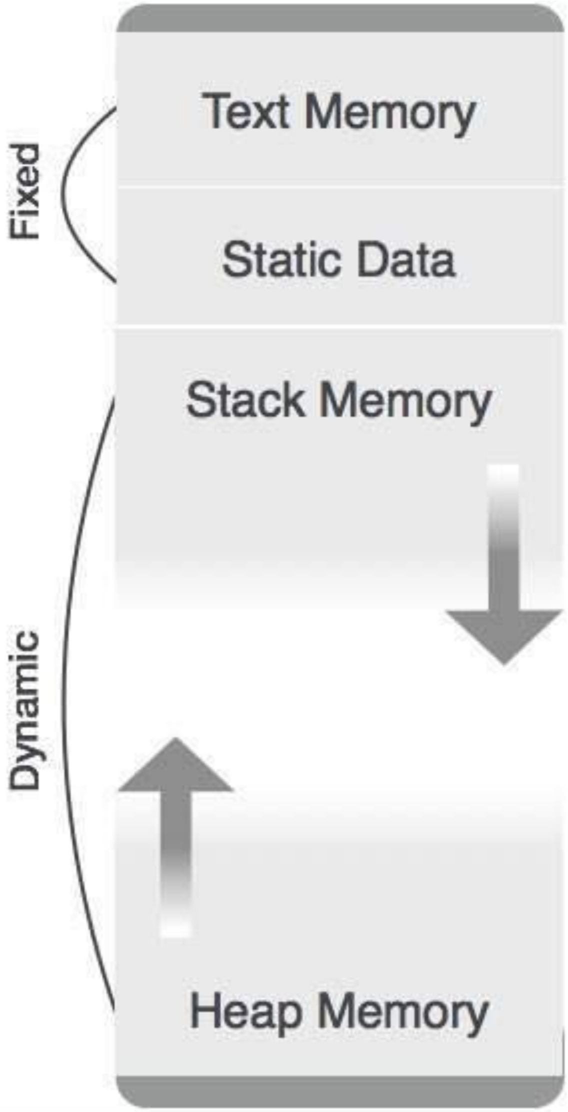
Stack Allocation

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

Heap Allocation

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

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



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

MCQ Questions:

1.Symbol table can be used in ……………………………….

A. checking compatibility

B. suppressing duplicate error messages

C. storage allocation

D. all of these

ANSWER: D

2.__________ keeps track of live procedure activations.

A. symbol table

B. activation record

C. DAG

D. postfix expressions

ANSWER: B

3.A compiler uses a __________ to deep track of scope and binding information about names and is searched every time a name is encountered in the source text.

A. Literal table

B. Symbol table

C. Activation Record

D. None of these

ANSWER: B

4.Symbol table can be used in ……………………………….

A. checking compatibility

B. suppressing duplicate error messages

C. storage allocation

D. all of these

ANSWER: D

5. Terminal table

A. contains all constants in the program.

B. is a permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.

C. consist of a full or partial list of the token is as they appear in the program created by lexical analysis and used for syntax analysis and interpretation.

D. is a permanent table which lists all keywords and special symbols of the language in symbolic form

ANSWER: D

6.The symbol table implementation is based on the property of locality of reference is

A. linear list

B. search tree

C. hash table

D. self-organization list

ANSWER: C

7.Which table is permanent databases that has an entry for each terminal symbol ?

A. Terminal table

B. Literal table

C. Identiier table

D. None of these

ANSWER: A

8.The table created by lexical analysis to describe all literals used in the source program is

A. Terminal table

B. Literal table

C. Identiier table

D. Reductions

ANSWER: A

Short Answer Type Questions:

1. What is Symbol table?
2. What is terminal table?
3. What is literal table?
4. What do you mean by Activation of records?
5. What do you mean by Nesting depth approach?
6. What is stack?
7. What is Heap?

Assignment:

1. What is type checking? Differentiate between static and dynamic type checking?

2. Write short notes:

   a) Symbol Table and its organization

   b) Activation record

   c) Nesting Depth Approach

Web/Video links:

[1] https://nptel.ac.in/courses/106108052/2

[2] https://nptel.ac.in/courses/106108052/3

[3] https://nptel.ac.in/courses/106108052/4

[4] https://nptel.ac.in/courses/106108052/5

[5]https://www.youtube.com/watch?v=1FUQyZT9gQQ&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=36

[6] https://www.youtube.com/watch?v=gxorCOk0YPw&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=37

[7] https://www.youtube.com/watch?v=FGKcX9mEYN8&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=38

[8] https://www.youtube.com/watch?v=4a1scz1jTxA&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=39

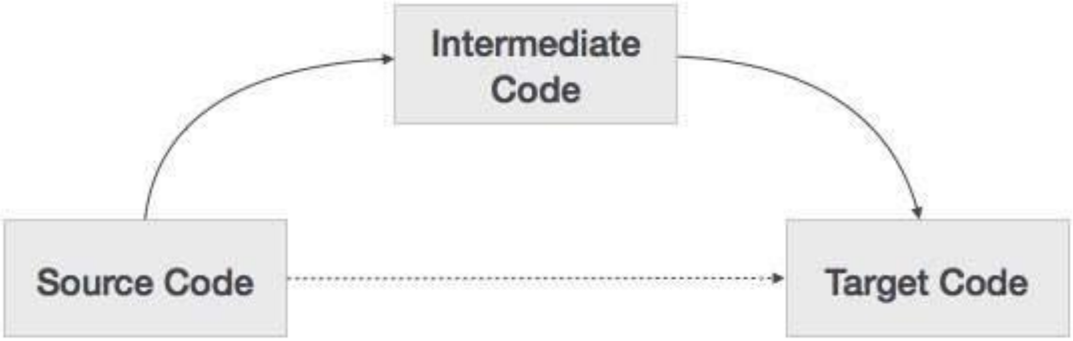
[9] https://www.youtube.com/watch?v=7WGszSgY17M&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=40

[10] https://www.youtube.com/watch?v=fzq0_pfeNxA&index=41&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1

[11] https://www.youtube.com/watch?v=IKSdN13DMwI&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=42

Module IV
LECTURE 1
Intermediate languages and Graphical representation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR -** High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR -** This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

The following are commonly used intermediate code representation:

1. Postfix Notation –

The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b

The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example – The postfix representation of the expression (a – b) * (c + d) + (a – b) is :   ab – cd + *ab -+.

Read more: Infix to Postfix

2. Three-Address Code –

A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

T 3 = T 2 + d

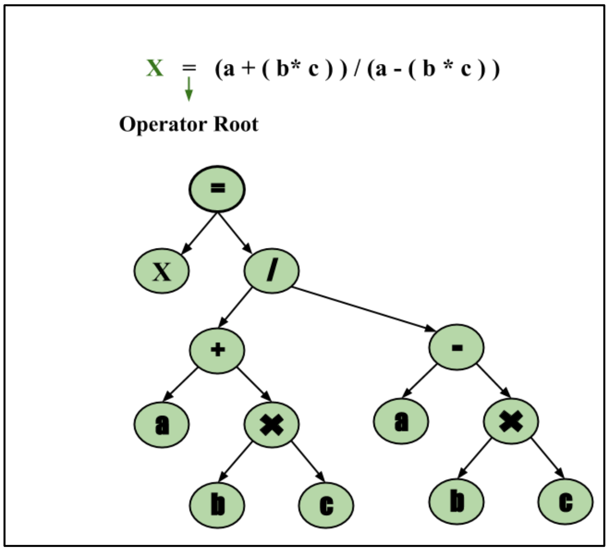
T 1 , T 2 , T 3 are temporary variables.

3. Syntax Tree –

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example –

x = (a + b * c) / (a – b * c)



LECTURE 2

Three-address code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage

(register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Implementation of three address statements

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg$_1$	arg$_2$	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op arg$_1$ arg$_2$

*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;

Allocation process:
{offset = 0}

   int a;
   id.type = int
   id.width = 2

offset = offset + id.width
{offset = 2}

   float b;
   id.type = float
   id.width = 4
```

```
offset = offset + id.width
{offset = 6}
```

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

MCQ Questions:

1.Three address code involves
A. Exactly three address
B. Atmost 2 address
C. 1 address
D. All of the above
ANSWER: A
2.What are the various kinds of intermediate representations for intermediate code generation?
A. Syntax trees
B. Postfix notation
C. Three address code
D. All of the above
ANSWER: D
3.An intermediate code form is
A. postfix notation
B. syntax trees
C. three address codes
D. all of these
ANSWER: D

Short Answer Type Questions:

1. What do you mean by three address code?Give an example?

Assignment:

1. Consider the following expression and represent it into quadruple, triple, indirect three address mode representation: x=(a+b)*(c+d)+(a+b+c)

2. Generate the three address code for the following code:

```
        while(A<C and B>D)do
            if A=1 then C=C+1
              else
            while A<=D do
              A=A+3
```

Web/Video links:

[1] https://www.youtube.com/watch?v=EpAzj7zXrbk

[2] https://www.youtube.com/watch?v=jNbSX8dID3g

[3] https://www.youtube.com/watch?v=uzjk7UopdSk

Module V
LECTURE 1

Consideration for Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
   item = 10;
   value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
   value = value + item;
} while(value<100);
```

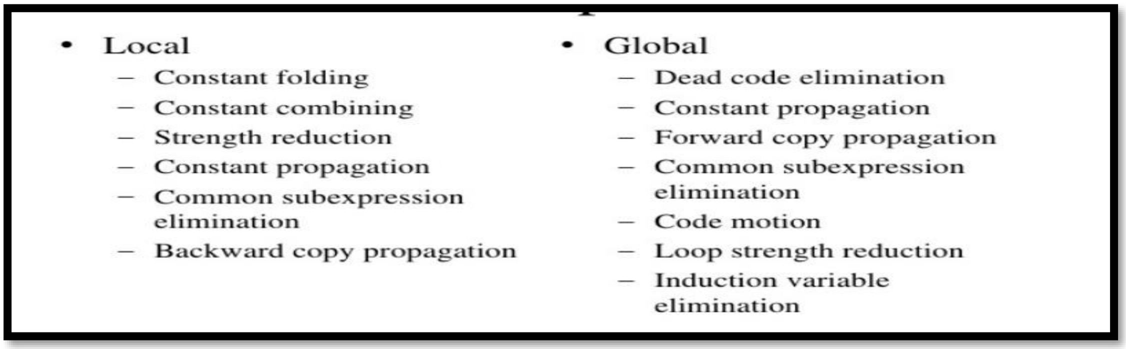
should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Scope of Optimization

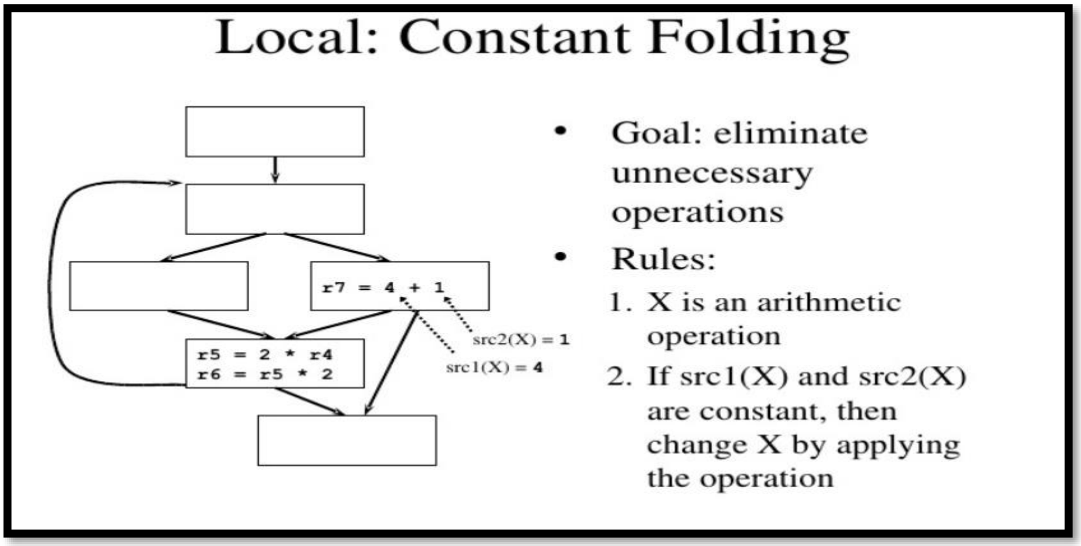
Local vs Global Optimization:



- Local
  - Constant folding
  - Constant combining
  - Strength reduction
  - Constant propagation
  - Common subexpression elimination
  - Backward copy propagation
- Global
  - Dead code elimination
  - Constant propagation
  - Forward copy propagation
  - Common subexpression elimination
  - Code motion
  - Loop strength reduction
  - Induction variable elimination

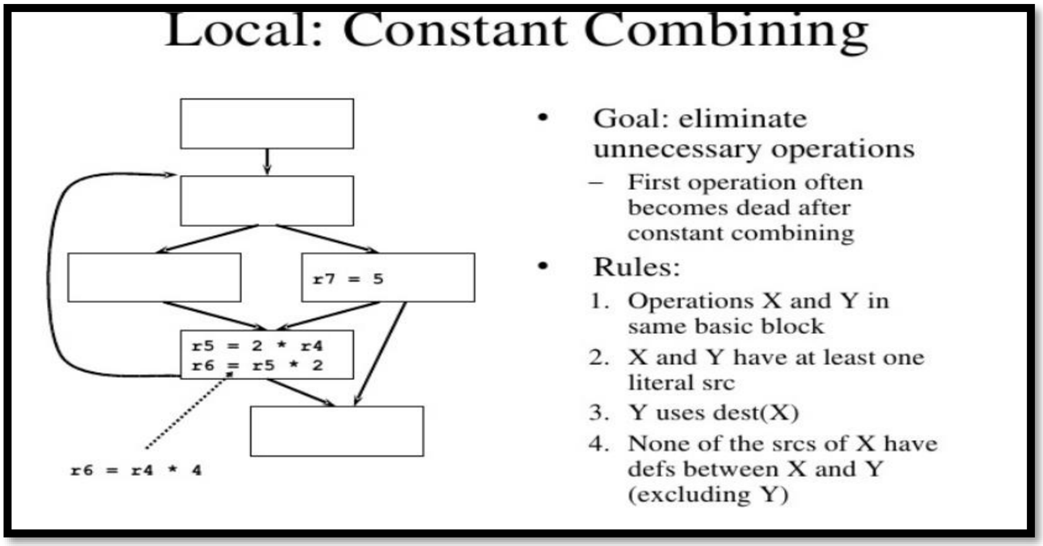
Loop Optimization:

Folding:



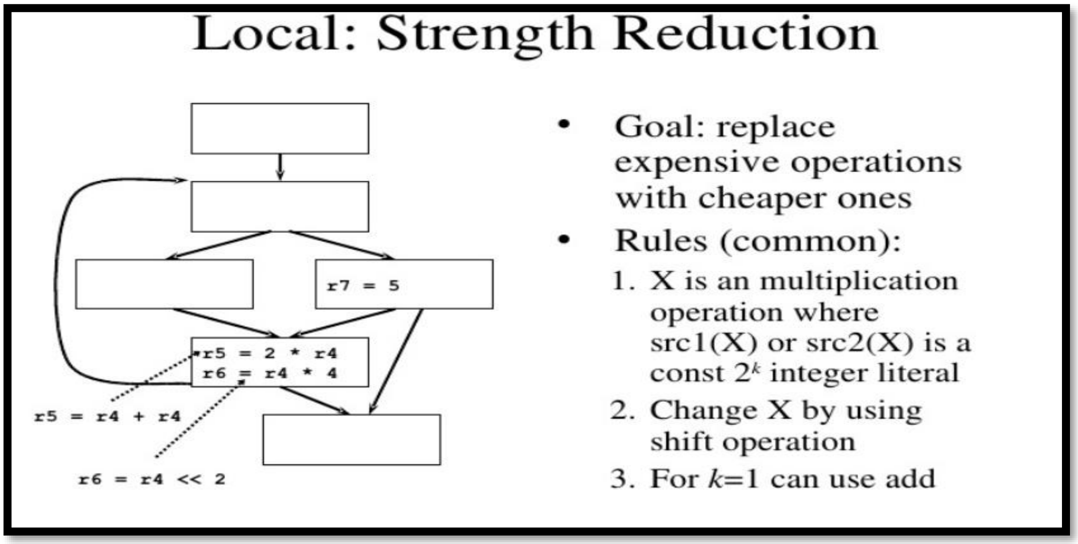
Local: Constant Folding

- Goal: eliminate unnecessary operations
- Rules:
  1. X is an arithmetic operation
  2. If src1(X) and src2(X) are constant, then change X by applying the operation



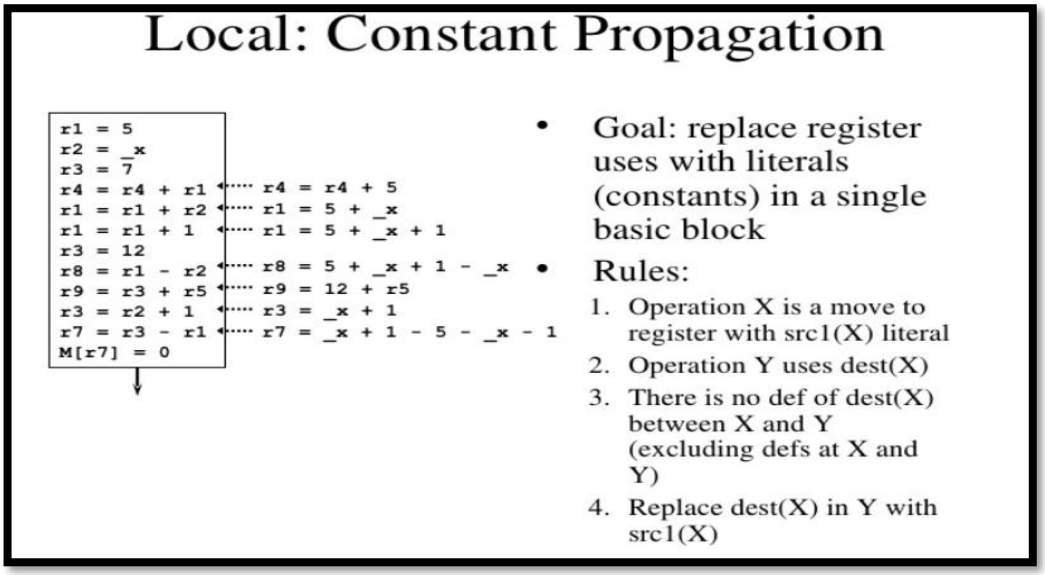
Local: Constant Combining

- Goal: eliminate unnecessary operations
  - First operation often becomes dead after constant combining
- Rules:
  1. Operations X and Y in same basic block
  2. X and Y have at least one literal src
  3. Y uses dest(X)
  4. None of the srcs of X have defs between X and Y (excluding Y)



Local: Strength Reduction

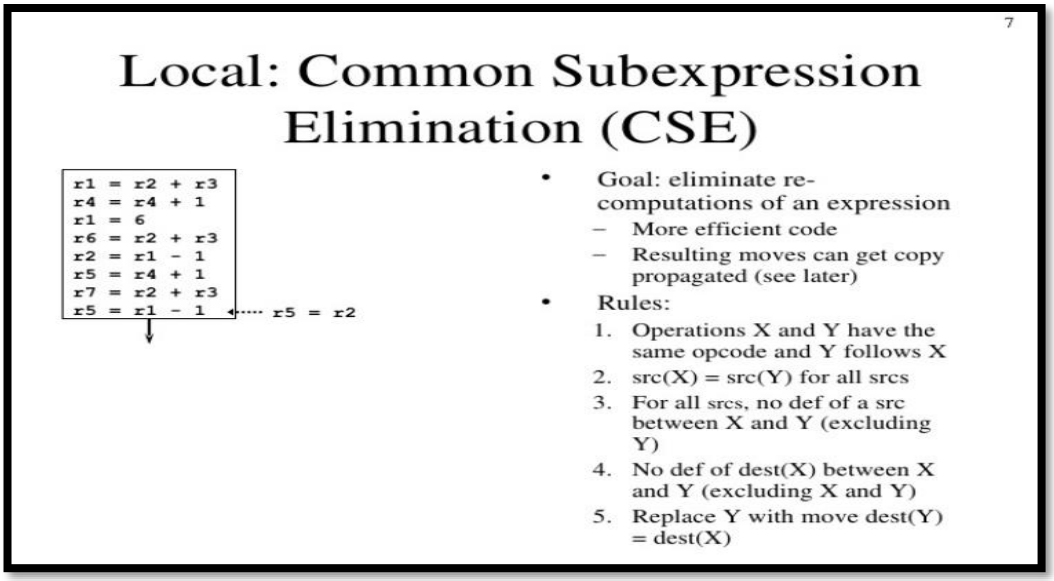
- Goal: replace expensive operations with cheaper ones
- Rules (common):
  1. X is an multiplication operation where src1(X) or src2(X) is a const 2^k integer literal
  2. Change X by using shift operation
  3. For k=1 can use add



Local: Constant Propagation

```
r1 = 5
r2 = _x
r3 = 7
r4 = r4 + r1 <····· r4 = r4 + 5
r1 = r1 + r2 <····· r1 = 5 + _x
r1 = r1 + 1  <····· r1 = 5 + _x + 1
r3 = 12
r8 = r1 - r2 <····· r8 = 5 + _x + 1 - _x
r9 = r3 + r5 <····· r9 = 12 + r5
r3 = r2 + 1  <····· r3 = _x + 1
r7 = r3 - r1 <····· r7 = _x + 1 - 5 - _x - 1
M[r7] = 0
```

- Goal: replace register uses with literals (constants) in a single basic block
- Rules:
  1. Operation X is a move to register with src1(X) literal
  2. Operation Y uses dest(X)
  3. There is no def of dest(X) between X and Y (excluding defs at X and Y)
  4. Replace dest(X) in Y with src1(X)

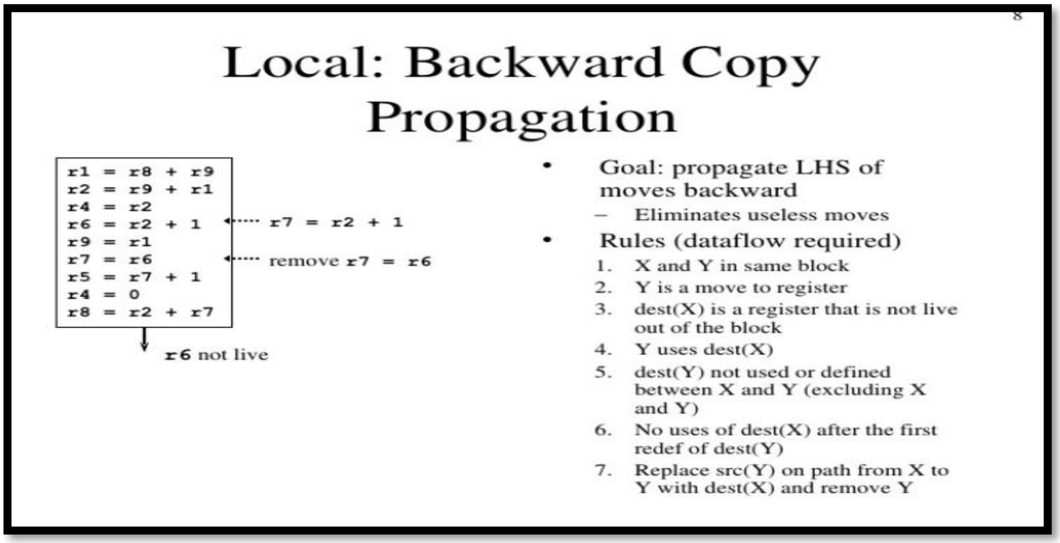


7

Local: Common Subexpression Elimination (CSE)

```
r1 = r2 + r3
r4 = r4 + 1
r1 = 6
r6 = r2 + r3
r2 = r1 - 1
r5 = r4 + 1
r7 = r2 + r3
r5 = r1 - 1   <····· r5 = r2
```

- Goal: eliminate re-computations of an expression
  - More efficient code
  - Resulting moves can get copy propagated (see later)
- Rules:
  1. Operations X and Y have the same opcode and Y follows X
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src between X and Y (excluding Y)
  4. No def of dest(X) between X and Y (excluding X and Y)
  5. Replace Y with move dest(Y) = dest(X)



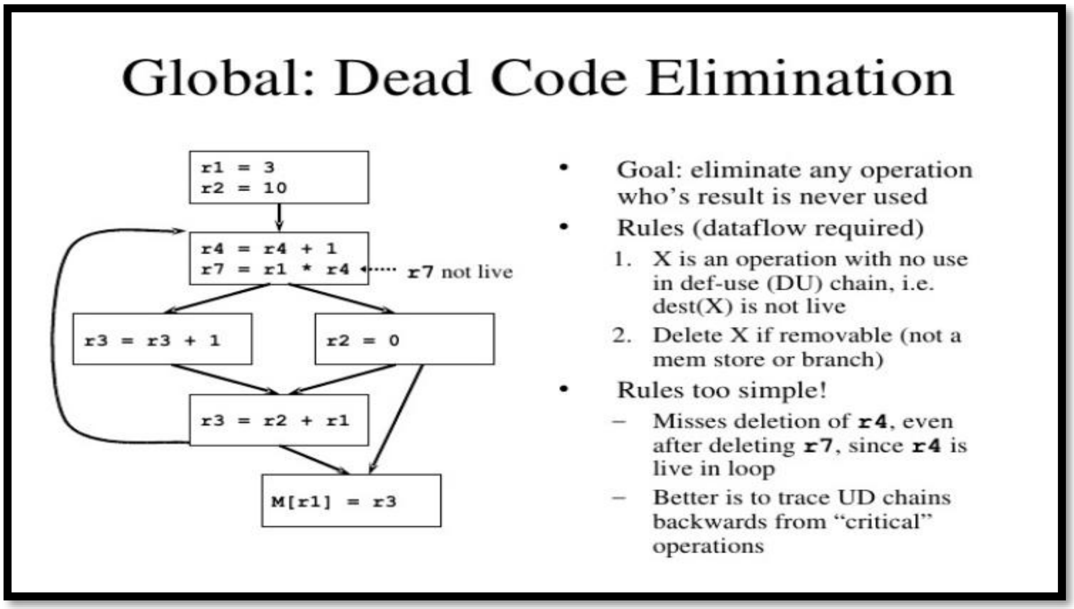
8

Local: Backward Copy Propagation

```
r1 = r8 + r9
r2 = r9 + r1
r4 = r2
r6 = r2 + 1   <····· r7 = r2 + 1
r9 = r1
r7 = r6       <····· remove r7 = r6
r5 = r7 + 1
r4 = 0
r8 = r2 + r7
```

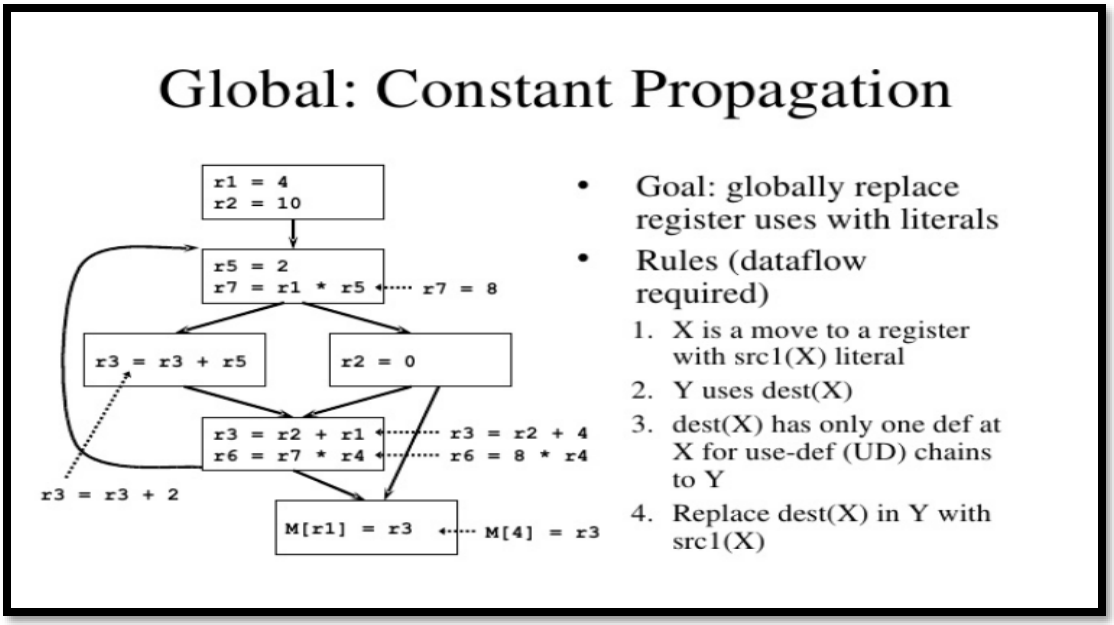
r6 not live

- Goal: propagate LHS of moves backward
  - Eliminates useless moves
- Rules (dataflow required)
  1. X and Y in same block
  2. Y is a move to register
  3. dest(X) is a register that is not live out of the block
  4. Y uses dest(X)
  5. dest(Y) not used or defined between X and Y (excluding X and Y)
  6. No uses of dest(X) after the first redef of dest(Y)
  7. Replace src(Y) on path from X to Y with dest(X) and remove Y



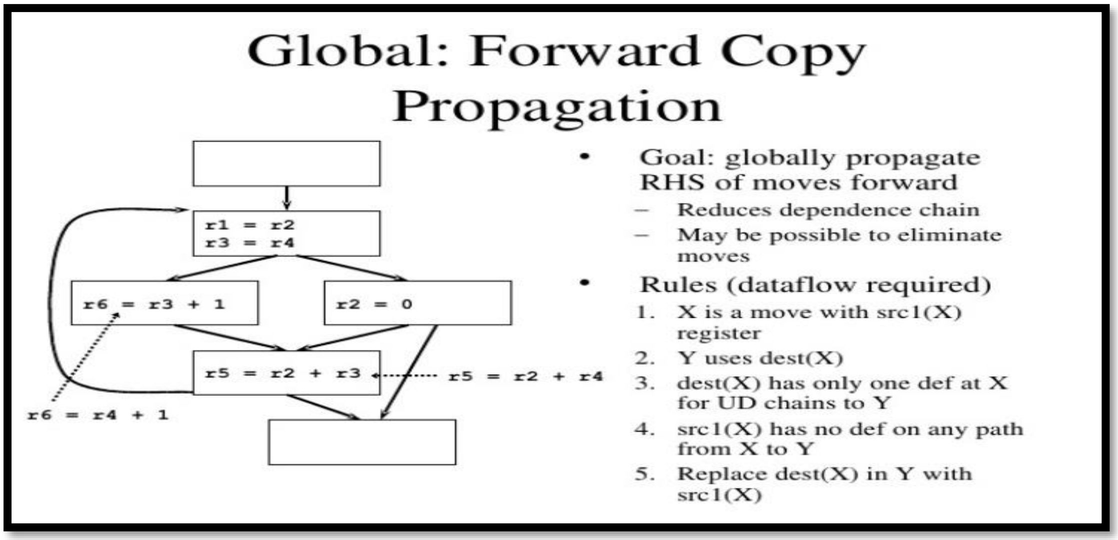
Global: Dead Code Elimination

- Goal: eliminate any operation who's result is never used
- Rules (dataflow required)
  1. X is an operation with no use in def-use (DU) chain, i.e. dest(X) is not live
  2. Delete X if removable (not a mem store or branch)
- Rules too simple!
  - Misses deletion of **r4**, even after deleting **r7**, since **r4** is live in loop
  - Better is to trace UD chains backwards from "critical" operations



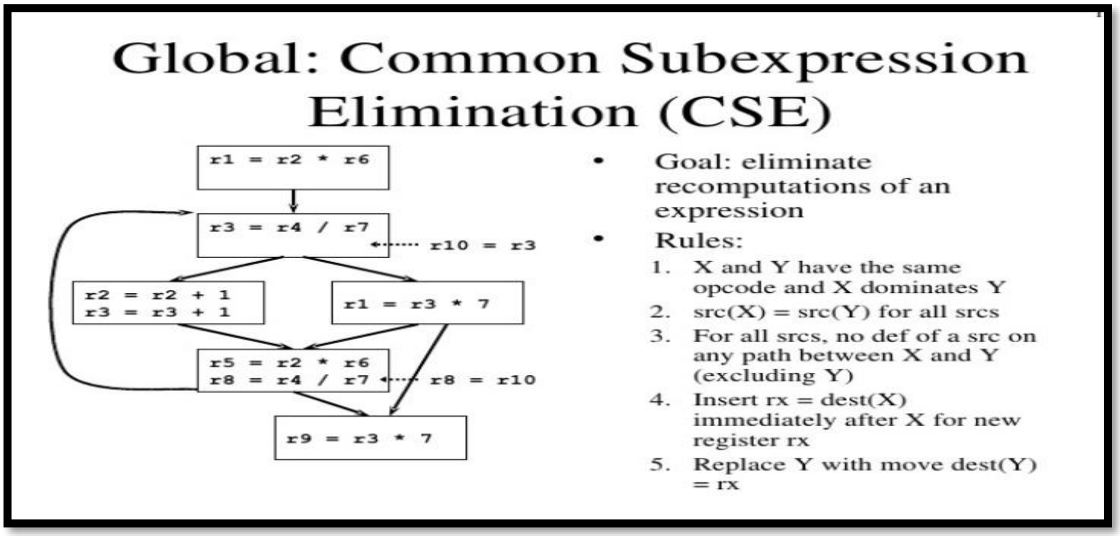
Global: Constant Propagation

- Goal: globally replace register uses with literals
- Rules (dataflow required)
  1. X is a move to a register with src1(X) literal
  2. Y uses dest(X)
  3. dest(X) has only one def at X for use-def (UD) chains to Y
  4. Replace dest(X) in Y with src1(X)



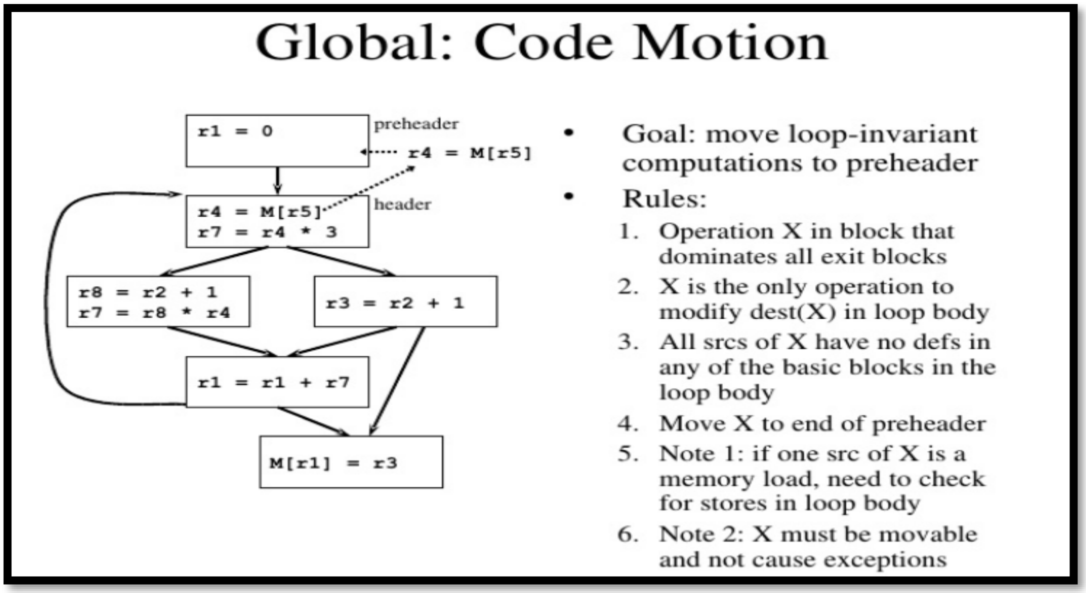
Global: Forward Copy Propagation

- Goal: globally propagate RHS of moves forward
  - Reduces dependence chain
  - May be possible to eliminate moves
- Rules (dataflow required)
  1. X is a move with src1(X) register
  2. Y uses dest(X)
  3. dest(X) has only one def at X for UD chains to Y
  4. src1(X) has no def on any path from X to Y
  5. Replace dest(X) in Y with src1(X)



Global: Common Subexpression Elimination (CSE)

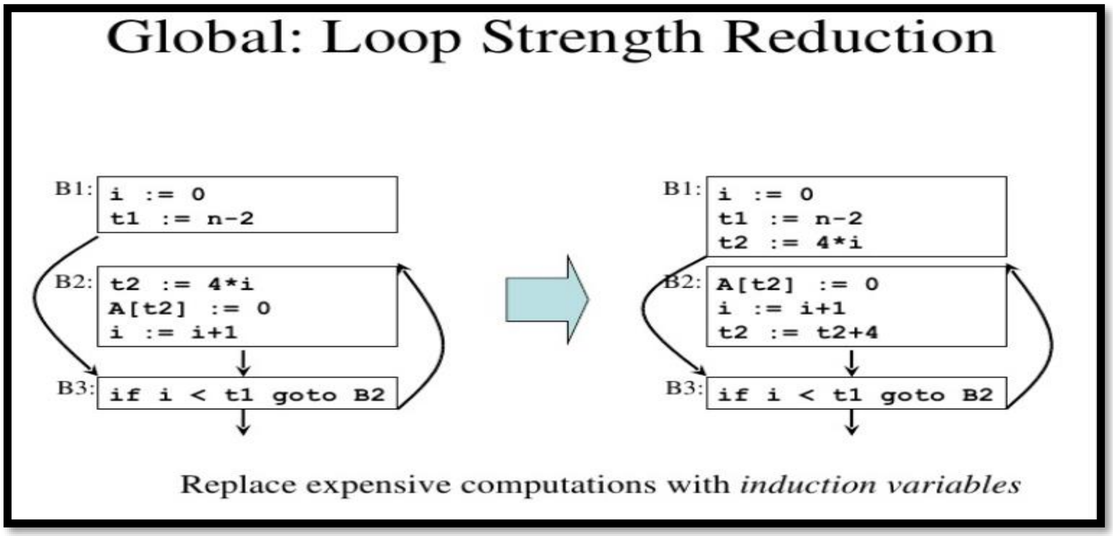
- Goal: eliminate recomputations of an expression
- Rules:
  1. X and Y have the same opcode and X dominates Y
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src on any path between X and Y (excluding Y)
  4. Insert rx = dest(X) immediately after X for new register rx
  5. Replace Y with move dest(Y) = rx



Global: Code Motion

```
r1 = 0
        r4 = M[r5]

r4 = M[r5]
r7 = r4 * 3

r8 = r2 + 1
r7 = r8 * r4

r3 = r2 + 1

r1 = r1 + r7

M[r1] = r3
```

- Goal: move loop-invariant computations to preheader
- Rules:
  1. Operation X in block that dominates all exit blocks
  2. X is the only operation to modify dest(X) in loop body
  3. All srcs of X have no defs in any of the basic blocks in the loop body
  4. Move X to end of preheader
  5. Note 1: if one src of X is a memory load, need to check for stores in loop body
  6. Note 2: X must be movable and not cause exceptions

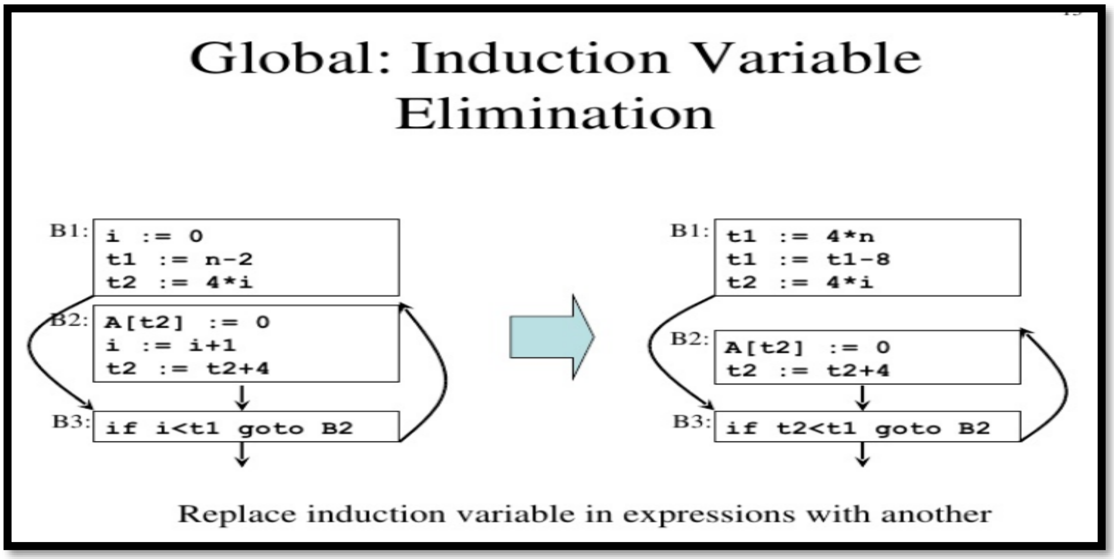


Global: Loop Strength Reduction

```
B1: i  := 0
    t1  := n-2

B2: t2  := 4*i
    A[t2]  := 0
    i  := i+1

B3: if i < t1 goto B2
```

```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i < t1 goto B2
```

Replace expensive computations with *induction variables*



Global: Induction Variable Elimination

```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i<t1 goto B2
```

```
B1: t1  := 4*n
    t1  := t1-8
    t2  := 4*i

B2: A[t2]  := 0
    t2  := t2+4

B3: if t2<t1 goto B2
```

Replace induction variable in expressions with another

LECTURE 2

DAG Representation

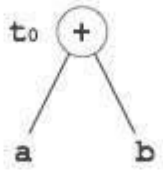
Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

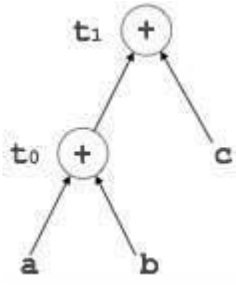
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

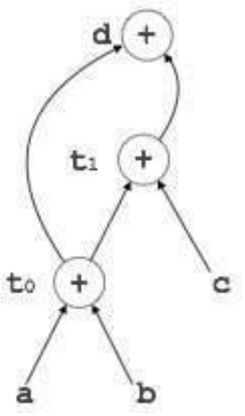
$t_0 = a + b$
$t_1 = t_0 + c$
$d = t_0 + t_1$



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

LECTURE 3

Basic Blocks & Flow Graphs

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

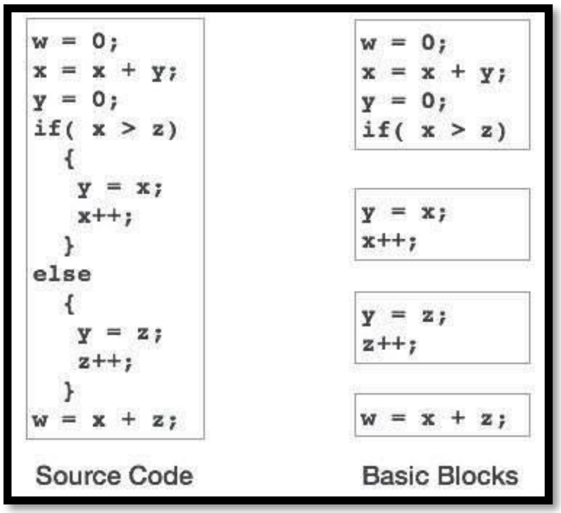
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.
    - Statements that are target of any branch (conditional/unconditional).
    - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
    y = x;
    x++;
  }
else
  {
    y = z;
    z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

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

```
y = z;
z++;
```

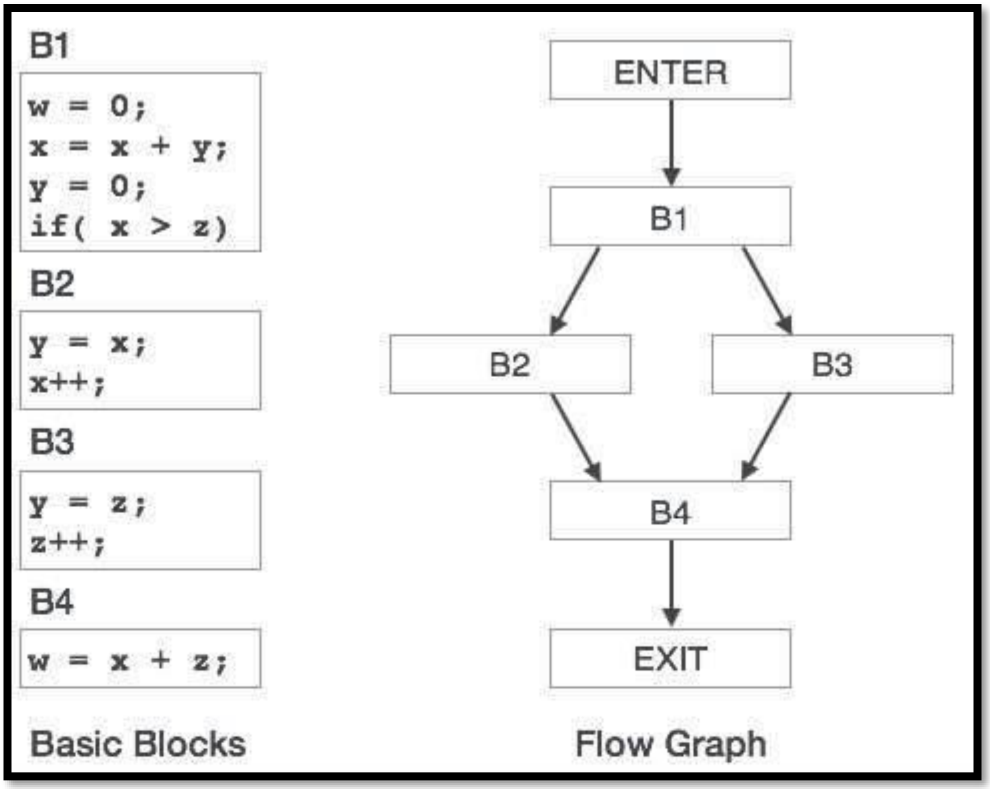
```
w = x + z;
```

Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



B1

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

B2

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

B3

```
y = z;
z++;
```

B4

```
w = x + z;
```

Basic Blocks

Flow Graph

LECTURE 4

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x) int add_ten(int x) int add_ten(int x) int add_ten(int x)
   {                  {                  {                  {
   int y, z;          int y;             int y = 10;        return x + 10;
   y = 10;            y = 10;            return x + y;      }
   z = x + y;         y = x + y;         }
   return z;          return y;
   }                  }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
   return x + 10;
   printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

LECTURE 5

.Code Generation

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Issues in the design of a code generator

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

The following issue arises during the code generation phase:

Input to code generator –

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

Target program –

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

Absolute machine language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.

Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.

Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

Memory Management –

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

Instruction selection –

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

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

```
P:=Q+R

S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S
```

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

Register allocation issues –

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

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

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

As the number of variables increase, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

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

Evaluation order –

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

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

Correct

Easily maintainable

Testable

Maintainable

LECTURE 6

Register Allocation and Assignment:

In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. Register allocation can happen over a basic block (local register allocation), over a whole function/procedure (global register allocation), or across function boundaries traversed via call-graph (interprocedural register allocation). When done per function/procedure the calling convention may require insertion of save/restore around each call-site.

Local register allocation

Register allocation is only within a basic block. It follows top-down approach.

Assign registers to the most heavily used variables

Traverse the block

Count uses

Use count as a priority function

Assign registers to higher priority variables first

Advantage

Heavily used values reside in registers

Disadvantage

Does not consider non-uniform distribution of uses

Need of global register allocation

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

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

Register allocation depends on:

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

Register allocation by graph coloring

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

Basic idea:

1. Identify the live range of each variable

2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)

3. Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

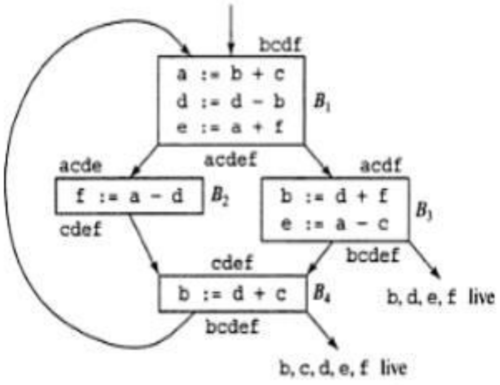
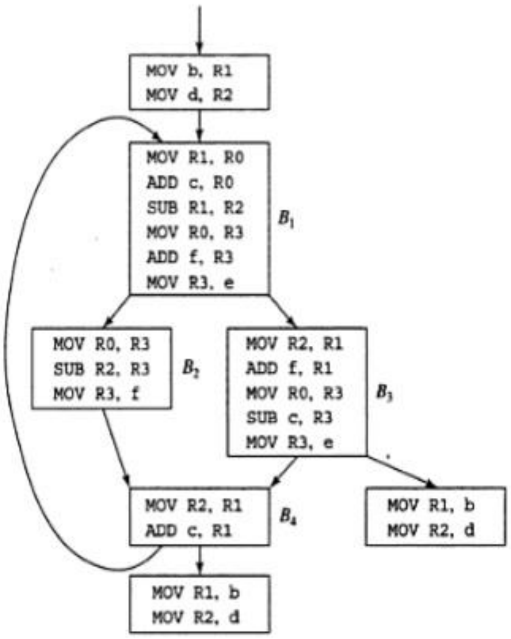


Fig 4.3 Flow graph of an inner loop

Fig 4.3 Flow graph of an inner loop



```
MOV b, R1
MOV d, R2
```

```
MOV R1, R0
ADD c, R0
SUB R1, R2
MOV R0, R3
ADD f, R3
MOV R3, e
```

```
MOV R0, R3
SUB R2, R3
MOV R3, f
```

```
MOV R2, R1
ADD f, R1
MOV R0, R3
SUB c, R3
MOV R3, e
```

```
MOV R2, R1
ADD c, R1
```

```
MOV R1, b
MOV R2, d
```

```
MOV R1, b
MOV R2, d
```

Fig 4.4 Code sequence using global register assignment

Fig 4.4 Code sequence using global register assignment

LECTURE 7

DAG for Register Allocation:

Code generation from DAG is much simpler than the linear sequence of three address code
With the help of DAG one can rearrange sequence of instructions and generate and efficient code .There exist various algorithms which are used for generating code from DAG. They are:Code Generation from DAG:-

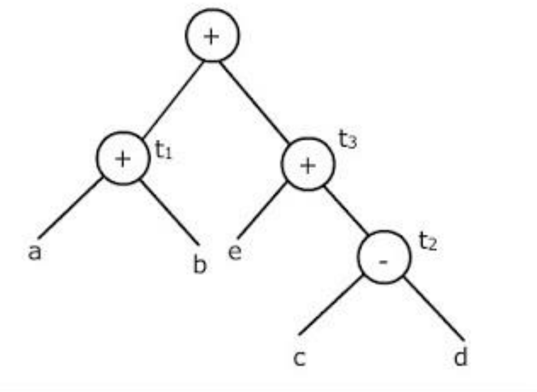
Rearranging Order
Heuristic Ordering
Labeling Algorithm

Rearranging Order

These address code's order affects the cost of the object code which is being generated
Object code with minimum cost can be achieved by changing the order of computations
Example:

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

For the expression (a+b) + (e+(c-d)), a DAG can be constructed for the above sequence as shown below



DAG for *(a + b) + (e + (c - d))*

The code is thus generated by translating the three address code line by line

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t          t1:= a+b
MOV e, R0          R1 has c-d
ADD R0, R1           /* R1 contains e + (c – d)*/
MOV t1, R0         /R0 contains a + b*/
ADD R1, R0
MOV R0, t4
```

Now, if the ordering sequence of the three address code is changed

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

Then, an improved code is obtained as:

```
MOV c, R0
SUB D, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, t4
```

Heuristic Ordering

The algorithm displayed below is for heuristic ordering. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.

```
{
While unlisted interior nodes remain
do
{
select an unlisted node n, all of whose parents have been listed;
List n;
While the leftmost child 'm' of 'n' has no unlisted parents and is not a leaf
do
{
/*since na was just listed, surely m is not yet listed*/
}
{
list m
n =m;
}
}
order = reverse of the order of listing of nodes
}
```

Labeling Algorithm

Labeling algorithm deals with the tree representation of a sequence of three address statements

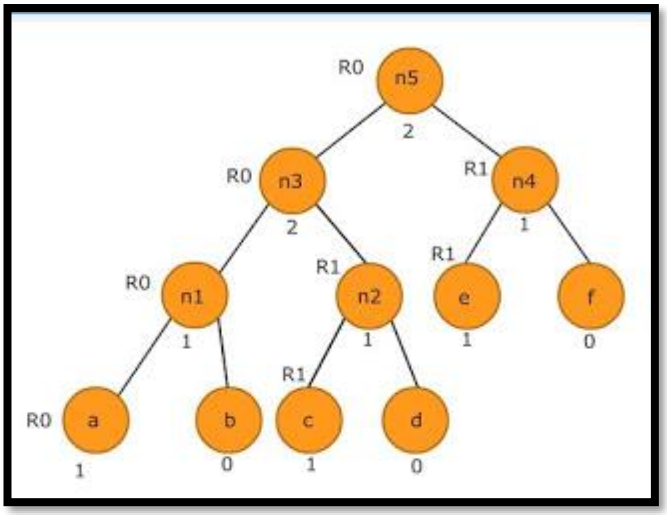
It could similarly be made to work if the intermediate code structure was a parse tree. This algorithm has two parts:

- The first part labels each node of the tree from the bottom up, with an integer that denotes the minimum number of registers required to evaluate the tree, and with no storing of intermediate results
- The second part of the algorithm is a tree traversal that travels the tree in an order governed by the computed labels in the first part, and which generates the code during the tree traversal

```
if n is a leaf then
    if n is leftmost child of its parents then
        Label(n) = 1
    else label(n) = 0
else
```

```
{
        /*n is an interior node*/
        let n1, n2, …..,nk be the children of ordered by label,
        so label(n1) >= label(n2) >= … >= label(nk);
        label(n) = max (label (ni) + i – 1)
}
```

Example:



MCQ Questions:

1.Three address code involves

A. Exactly three address

B. Atmost 2 address

C. 1 address

D. All of the above

ANSWER: A

2.DAG representation of basic block

A. Detection of Common sub expression

B. Detection of Loop

C. Flow graph

D. none

ANSWER: D

3.___________ keeps track of live procedure activations.

A. symbol table

B. activation record

C. DAG

D. postfix expressions

ANSWER: B

4.The phases of a compiler are Lexical analysis,Syntax analysis,Semantic Analysis,Intermediate code generation, __________________________,Code generation

A. Code Translation

B. Dynamic Optimization

C. Code Optimization

D. None of the above

ANSWER: C

5.What are the various kinds of intermediate representations for intermediate code generation?

A. Syntax trees

B. Postfix notation

C. Three address code

D. All of the above

ANSWER: D

6.A useful data structure for automatically analyzing basic blocks is

A. Control flow graph (CFG)

B. directed acyclic graph (DAG)

C. Hash Table

D. Linear List

ANSWER: A

7.The object file contains the

A. assembly code

B. machine code

C. modified source code

D. none of the mentioned

Answer: B

8.The graph that shows basic blocks and their successor relationship is called

A. DAG

B. Data Flow graph

C. control flow graph

D. Hamiltonion graph

ANSWER: C

9.An intermediate code form is

A. postfix notation

B. syntax trees

C. three address codes

D. all of these

ANSWER: D

10.A compiler program written in a high level language is called

A. source program

B. object program

C. machine language program

D. none of these

ANSWER: A

Short Answer Type Questions:

1. What is DAG?
2. What is local and global optimization?
3. What is peephole optimization?
4. What is basic Block?
5. What is CFG?

Assignment:

1. What is DAG? Explain briefly?

2. Construct the DAG for the following basic block:

d=b*c
e=a+b
b=b*c
a=e-d

3. Consider some interblock code optimization without any data flow analysis by treating each extended basic block as if it is basic block .Give algorithms to the following optimizations within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have.

i)Common sub-expression elimination

ii)Constant folding

iii)Copy propagation

4. Design a dependency graph and DAG for the following expression a+a*(b-c)+(b-c)*d

5. Write shorts:

a) Loop optimization

b) Loop unrolling

c) Loop Jamming

d) Copy Propagation

e) Peephole Optimization

f) Register allocation

6. What are the issues are there in register allocation algorithm? Explain.

Web/Video links:

[1] https://nptel.ac.in/courses/106108052/6

[2] https://nptel.ac.in/courses/106108052/7

[3] https://nptel.ac.in/courses/106108052/8

[4] https://nptel.ac.in/courses/106108052/9

[5] https://nptel.ac.in/courses/106108052/10

[6] https://nptel.ac.in/courses/106108052/11

[7] https://nptel.ac.in/courses/106108052/12

[8] https://nptel.ac.in/courses/106108052/13

[9] https://nptel.ac.in/courses/106108052/14

[10]https://www.youtube.com/watch?v=JSR65MTjTOY&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=44

[11] https://www.youtube.com/watch?v=EVc4sD6FMMA&index=49&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1

[12] https://www.youtube.com/watch?v=w3AX5KOmbEE&index=50&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1

[13] https://www.youtube.com/watch?v=Zbf_XzgtEQ4&index=51&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1

[14] https://www.youtube.com/watch?v=MgfsFiq3Kl4&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1&index=52

[15] https://www.youtube.com/watch?v=gHdqF2iM7kk&index=53&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1

[16] https://www.youtube.com/watch?v=wV9NSi9zdOY&index=54&list=PLG9aCp4uE-s3XepZyd94jGic7qMFa7CW1