MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS **L - 1**

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of Lecture: Introduction to Compiler

Introduction: (Maximum 5 sentences) : A **compiler** is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) and the **compiler** reports to its user the presence of errors in the source program.

Prerequisite knowledge for Complete understanding and learning of Topic:
(Max. Four important topics)

- Theory of computation basics
- Non-Deterministic Finite Automata
- Deterministic Finite Automata

Detailed content of the Lecture:

Translators - acts as a translator, convert <u>source language</u> into <u>target language</u>

Types of Translators

1.Compiler
2.Interpreter
3.Assembler

Compiler:

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low

level language).

- In the second part, object program translated into the target program through the assemble

Interpreter

An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Examples of interpreted languages are Perl, Python and Matlab.

Assembler

An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.

- **Cross Compiler** that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-source Compiler** or transcompiler or transpiler is a compiler that translates source code written in one programming language into source code of another programming language.
- **Loader** - A loader is a program that places machine code of the programs into memory and prepares them for execution
- **Link-editor** - The linker resolves external memory addresses, where the code in one file may refer to a location in another file, so the relocatable machine code may have to be linked together with other relocatable object files and library files. It is done by linker.

Compiler	Interpreter
It takes an entire program at a time	It takes a single line of code or instruction at a time
It generates intermediate object code	It does not produce any intermediate object code
The compilation is done before execution	Compilation and execution take place simultaneously
Comparatively faster	Slower
Memory requirement is more due to the creation of object code	It requires less memory as it does not create intermediate object code
Display all errors after compilation ,all at the same time	Displays error of each line one by one
Error detection is difficult	Easier comparatively
Eg: C,C++,C#	Eg: PHP,Perl,Python

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=KBulg_u-b3w

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 13-15

Course Teacher

Verified by HOD

LECTURE HANDOUTS | **L -2**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Basic Machines Finite Automata (FA) - Deterministic Finite Automata (DFA) – Nondeterministic Finite Automata (NFA), Finite Automata with Epsilon transitions

Introduction: (Maximum 5 sentences):

- A recognizer for a language is a program that takes as input a string x and answers yes if x is a sentence of the language and no otherwise.
- A regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Theory of computation basics
- Concept of complier

Detailed content of the Lecture:

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state ($q0$)
- Set of final states (qf)
- Transition function (δ)

Deterministic Finite Automata (DFA)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

Q: finite set of states
∑: finite set of the input symbol

q0: initial state
F: **final** state
δ: Transition function

Nondeterministic Finite Automata (NFA):

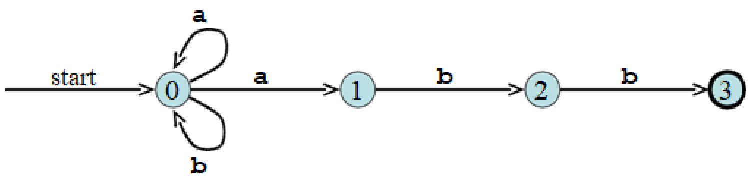
An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- **δ** is the transition function where δ: Q × ∑ → 2^Q

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).
- **F** is a set of final state/states of Q (F ⊆ Q).

Transition Graph:

An NFA can be diagrammatically represented by a labeled directed graph called a ***transition graph.***



$S = \{0,1,2,3\}$
$\Sigma = \{a,b\}$
$s_0 = 0$
$F = \{3\}$

Transition table

The **transition table** is basically a tabular representation of the **transition** function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

$\delta(0,a) = \{0,1\}$
$\delta(0,b) = \{0\}$
$\delta(1,b) = \{2\}$
$\delta(2,b) = \{3\}$

State	*Input* a	*Input* b
0	{0, 1}	{0}
1	-	{2}
2	-	{3}

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=Qkwj65l_96I

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 125-132

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS | **L - 3**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Converting Regular Expression to NFA

Introduction: (Maximum 5 sentences):

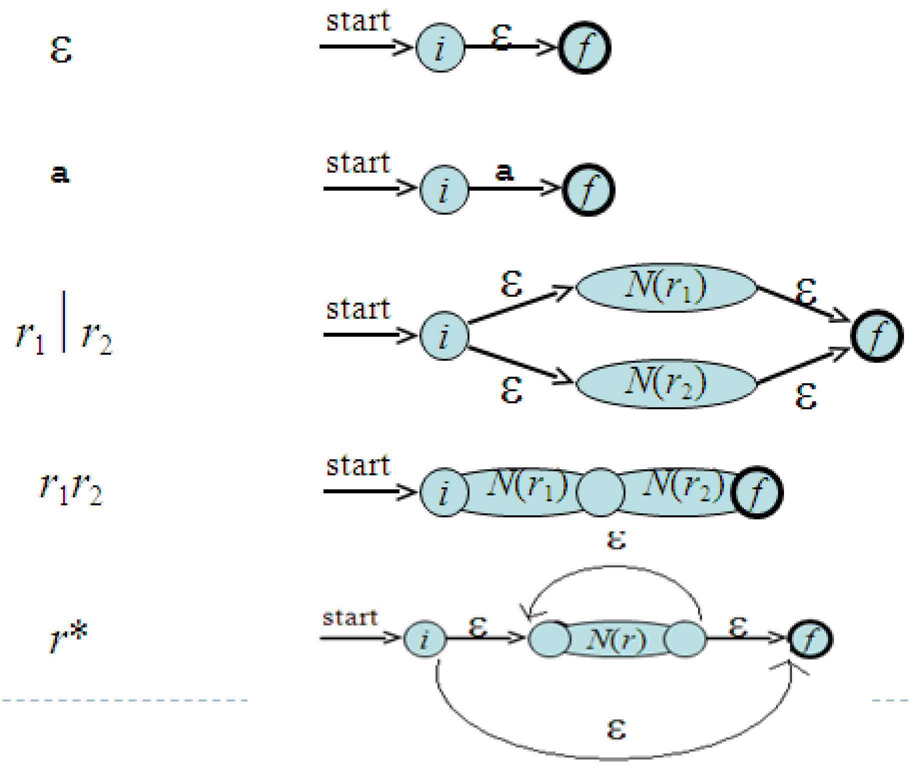
- Regular expression defines a language over the alphabet.
- Regular expression is an important notation for specifying patterns.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Non-Deterministic Finite Automata
- Deterministic Finite Automata

Detailed content of the Lecture:

Converting Regular Expression to NFA:



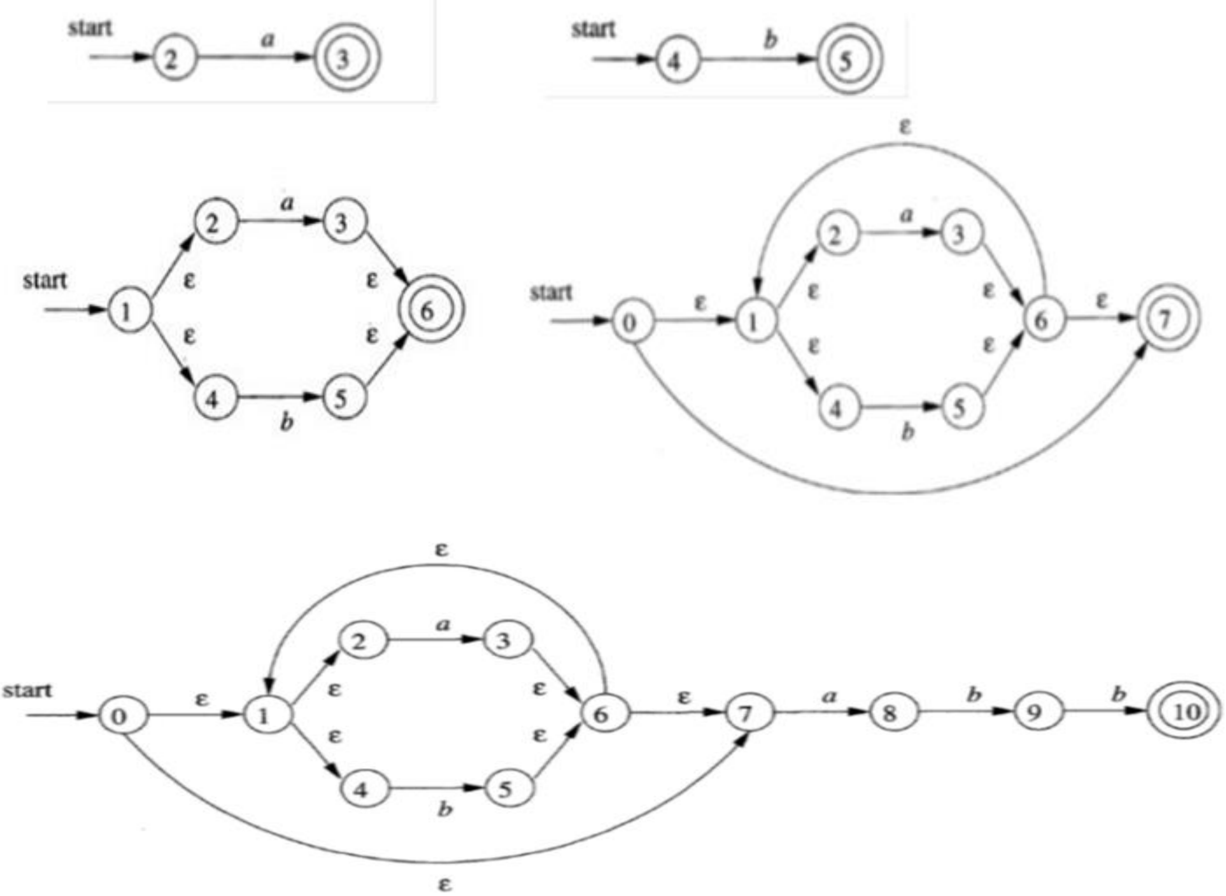
Eg: NFA for the RE =(a/b)*abb

Step 1: construct a, b

Step 2: constructing a | b

Step3: construct (a|b)*

Step4: concat it with a, then, b, then b



Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=7NA69TEd2iQ

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 133-140.

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

LECTURE HANDOUTS

L - 4

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Converting NFA to DFA, Minimization of DFA

Introduction: (Maximum 5 sentences):

- A NFA can have zero, one or more than one move from a given state on a given input symbol.
- An NFA can also have NULL moves (moves without input symbol).
- On the other hand, DFA has one and only one move from a given state on a given input symbol.

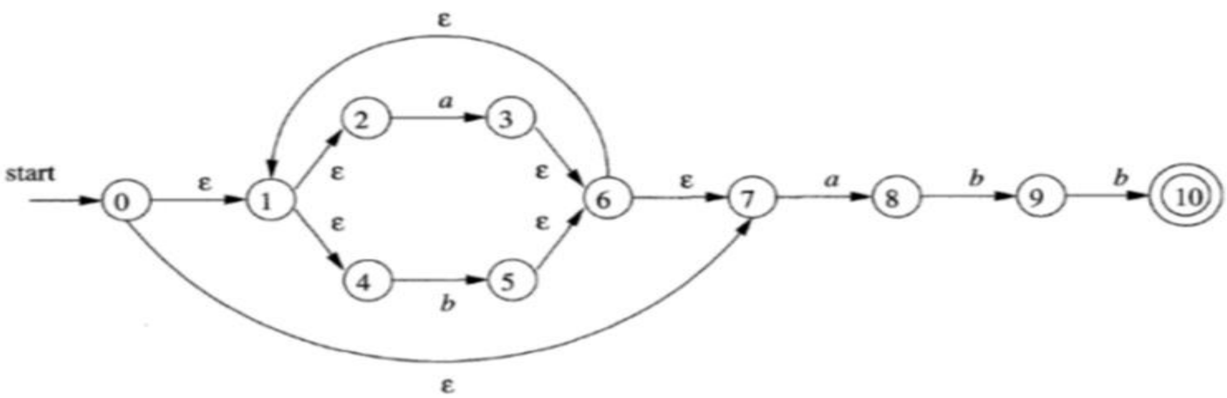
Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Non-Deterministic Finite Automata
- Deterministic Finite Automata
- Converting Regular Expression to NFA

Detailed content of the Lecture:

Converting NFA to DFA:

Step 1: NFA for RE



Step 2: Construct Subset to form DFA

Initially, e-*closure*(*s_0*) is the only state in *Dstates* and it is unmarked

while there is an unmarked state *T* in *Dstates* do

mark *T*

for each input symbol *a* Î S do *U* := e-*closure*(*move*(*T*,*a*))

if *U* is not in *Dstates* then

add *U* as an unmarked state to *Dstates*

end if

Dtran[*T*,*a*] := *U*

end do

end do

Step 1: ε – closure (0) = {0, 1, 2, 4, 7} ----- A

Step 2: ε -closure(move(A,a)) = move(A,a) = {3, 8}
ε- closure ({3,8} = {1, 2, 3, 4, 6, 7, 8} ------B
ε -closure(move(A,b)) = ε- closure ({5} = {1, 2, 4, 5, 6, 7 } ----------- C

Step 3 : ε -closure(move(B,a)) = ε- closure ({3,8} = {1, 2, 3, 4, 6, 7, 8 } ------ B
ε -closure(move(B,b)) = ε- closure ({5,9} = {1, 2, 4, 5, 6, 7,9} -------- D

Step 4 : ε -closure(move(C,a)) = ε- closure ({3,8} = {1, 2, 3, 4, 6, 7, 8 } -------B
ε -closure(move(C,b)) = ε- closure ({5} = {1, 2, 4, 5, 6, 7} ------------ C

Step 5 : ε -closure(move(D,a)) = ε- closure ({3,8} = {1, 2, 3, 4, 6, 7, 8 } -------B
ε -closure(move(D,b)) = ε- closure ({5,10} = {1,2, 4, 5, 6, 7, 10} -----E

Step 6: ε -closure(move(E,a)) = ε- closure ({3,8} = {1, 2, 3, 4, 6, 7, 8 }--------B
ε -closure(move(E,b)) = ε- closure ({5} = {1,2, 4, 5, 6, 7}-------------- C

Step 3: Transition table

State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

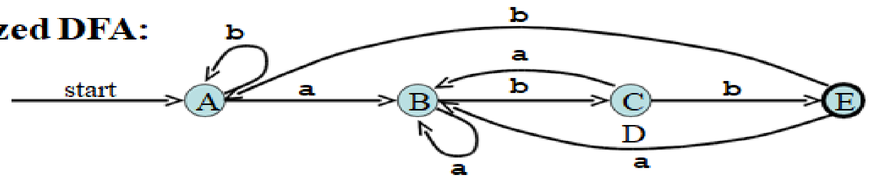
→

State	a	b
A	B	C
B	B	D
~~C~~	~~B~~	~~C~~
D	B	E
E	B	C

→

State	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Minimized DFA:



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=_9yzwYtfoK4

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 133-140 &146

Course Teacher

Verified by HOD

LECTURE HANDOUTS | **L - 5**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Converting Regular Expression to DFA

Introduction: (Maximum 5 sentences):

- Direct method is used to convert given regular expression directly into DFA.
- Uses augmented regular expression r#.
- Important states of NFA correspond to positions in regular expression that hold symbols of the alphabet.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Converting Regular Expression to NFA
- Converting NFA to DFA

Detailed content of the Lecture:

INPUT: A regular expression r

OUTPUT: A DFA D that recognizes L(r)

METHOD:

1.Constructing Syntaxtree for (r) #

2.Traverse the into tree to construct nullable(),firstpos(),lastpos(),followpos().

3.Converting RE directly into DFA

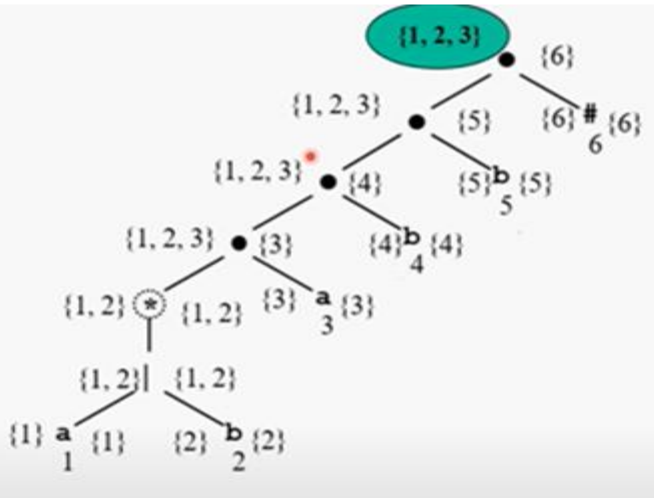
Augmented RE =(a/b)*abb #

Algorithm:

```
initialize Dstates to contain only the unmarked state firstpos(no),
where no is the root of syntax tree T for (r)#;
while ( there is an unmarked state S in Dstates )
{  mark S;
   for ( each input symbol a )
   {  let U be the union of followpos(p) for all p in S that correspond to a;
      if ( U is not in Dstates )
      add U as an unmarked state to Dstates; Dtran[S, a] = U    }    }
```

Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)	Followpos(n)
Leaf labeled by ε	true	∅	∅	*NA*
Leaf labeled by *i*	false	{*i*}	{*i*}	*NA*
| with children c_1, c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)	*NA*
• with children c_1, c_2	*nullable*(c_1) and *nullable*(c_2)	if *nullable*(c_1) then *firstpos*(c_1) ∪ *firstpos*(c_2) else *firstpos*(c_1)	if *nullable*(c_2) then *lastpos*(c_1) ∪ *lastpos*(c_2) else *lastpos*(c_2)	*If n is a* cat-node, *i= lastpos(c1), followpos(i)= firstpos(c2)*
* with child c_1	true	*firstpos*(c_1)	*lastpos*(c_1)	If n is a star-node, *i = lastpos(n), followpos(i)= firstpos(n)*

	Node	*followpos*
a	1	1,2,3
b	2	1,2,3
a	3	4
b	4	5
b	5	6
#	6	-



State	a	b
A	B	A
B	B	C
C	B	D
D	B	A

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=G8i_2CUHP_Y
https://www.youtube.com/watch?v=PsWFuqd2O8c

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 133-140 &146

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS

L-6

IT

III/V

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: The Phases of Compiler

Introduction: (Maximum 5 sentences): A **compiler** is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) and the **compiler** reports to its user the presence of errors in the source program.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Basic concepts of compiler
- Language processing system

Detailed content of the Lecture:

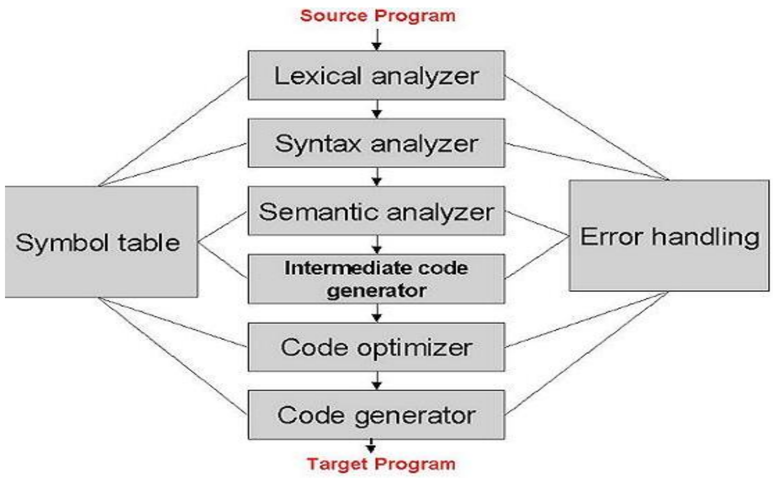
Compiler consist of 6 phases:

Analysis of the source program

– The analysis part carried out in three phases, they are lexical analysis, syntax analysis and Semantic Analysis
– The analysis part is often called the front end of the compiler.

Synthesis of a machine-language program

–The synthesis part carried out in three phases, they are Intermediate Code Generation, Code Optimization and Code Generation
– The synthesis part constructs the desired target program from the intermediate representation and the information are stored in the symbol table
– The synthesis part is called the back end of the compiler



1. Lexical Analysis

- The first phase of a compiler
- Other name- *scanning or linear analysis*
- The lexical analyzer reads source program and groups in to **tokens**

i.Token: Sequence of characters that can be treated as a single logical entity. Eg: Number, Identifiers ,keywords , etc..

ii. Pattern: Set of strings is described by a rule called a pattern associated with the token.

iii. Lexeme: Sequence of characters in the source program that is matched by the pattern for a token

2. Syntax Analysis

- Second phase of the compiler
- Other name - parsing or hierarchical analysis
- It generates hierarchical tree structure called parse tree or syntax tree

Properties of syntax tree

- 1. Each interior node represent - operator
- 2. Leaf node represent –token

3. Semantic Analysis

- Uses the syntax tree and the information in the symbol table to check semantic consistency
- It ensures the correctness of the program, matching of the parenthesis
- An important part of semantic analysis is type checking

4. Intermediate Code Generation

- After syntax and semantic analysis of the source program, many compilers generate intermediate representation

Intermediate representation has two properties:

1.It should be easy to produce

2. Easy to translate into the target machine

Eg: Three address code.

- Three address code have atmost 3 operand
- Atmost 1 operator additional to =
- Temporary variable to store result

5. Code Optimization

- The machine-independent code-optimization phase improve the intermediate code
- so that better target code will result
  - Faster
  - Shorter code
  - target code that consumes less power

6. Code Generator

- The code generator takes as input an intermediate representation of the source program and produce target language

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=Qkwj65l_96I

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 22-28.

Course Teacher

Verified by HOD

LECTURE HANDOUTS | **L-7**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: The Phases of Compiler

Introduction: (Maximum 5 sentences): A **compiler** is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) and the **compiler** reports to its user the presence of errors in the source program.

Prerequisite knowledge for Complete understanding and learning of Topic:
(Max. Four important topics)

- Basic concepts of complier
- Phases of Compiler

Detailed content of the Lecture:

Compiler consist of 6 phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generator

Additional to the phase compiler handle 2 activities

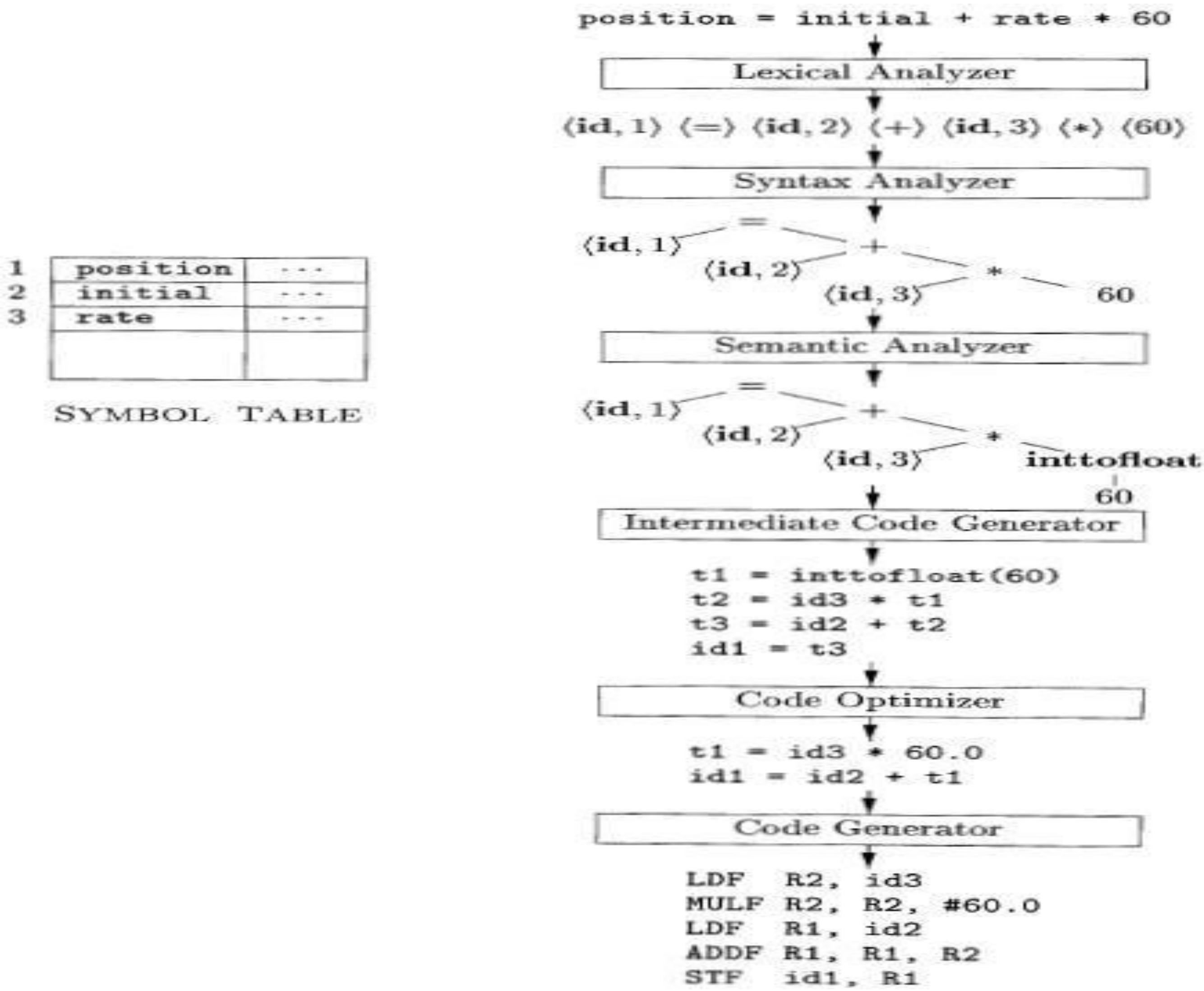
Symbol Table Management

- ➢ Symbol table is a data structure
- ➢ It contains a record for each variable name, with fields for its attributes of the name (address of the name, its type, its scope)

Error handler

- ➢ Important role of the compiler is to report errors in the program
- ➢ Each phases of compiler can encounter errors, after detecting errors, must be corrected to precede compilation process
- ➢ Error handler handles all types of errors like lexical errors, syntax errors, semantic errors and logical errors

Eg: for the input **position=initial+rate*60**



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=Qkwj65l_96I

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 22-28.

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS | **L-8**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Cousins of the Compiler, Compiler Construction Tools

Introduction: (Maximum 5 sentences):

- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

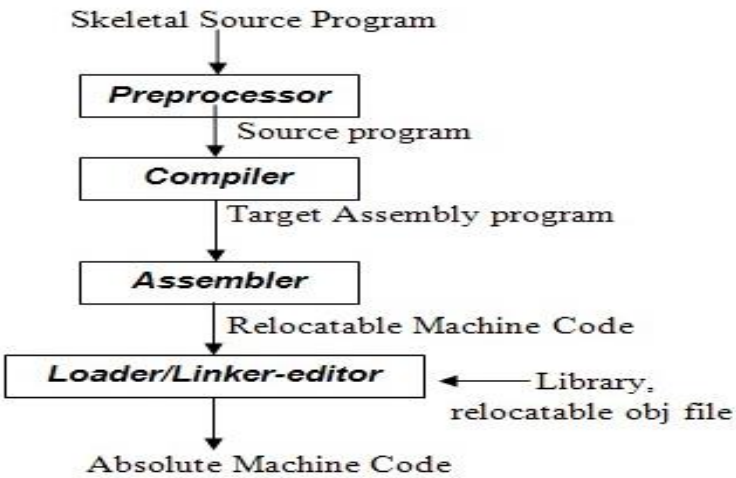
Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Basic concepts of complier
- Phases of Compiler

Detailed content of the Lecture:

Cousins of complier

To create an executable target program several programs may be required for process



1. Preprocessor

1. It produces input to compilers. They may perform the following functions

   1. Macro processing: A preprocessor may allow a user to define macros that are short hands for longer constructs.

   2. File inclusion: A preprocessor may include header files into the program text

   3. Rational preprocessor: these preprocessors augment older languages with more modern flow-of-control and data structuring facilities

4. Language Extensions: These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

2. Compiler - Compiler is a system software, that translate high level language (C, C++, Java) into machine level language. The translation done by a compiler is called compilation.

3. Interpreter - An interpreter is another translator that converts high level language in to machine level language statement by statement. The translation done by an interpreter is called Interpretation.

4. Assembler – convert assembly language into relocatable machine code as its output.

5. Loader - A loader is a program that places machine code of the programs into memory and prepares them for execution.

6. Link-editor - The linker resolves external memory addresses, where the code in one file may refer to a location in another file, so the relocatable machine code may have to be linked together with other relocatable object files and library files. It is done by linker.

Compiler construction tools

Compiler consists of 5 construction tools.

1.Scanner generator

The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (tokens). Scanner generators that produce lexical analyzers from a regular-expression. Unix has a tool for Scanner generator called LEX

2.Parser generator

Parser reads tokens. Parser automatically produce syntax analyzers (parse tree) from a CFG (Context-Free Grammar) Unix has a tool called YACC which is a parser generator

3.Syntax-directed translation engine

Perform two functions

Check the static semantics of each construct. Do the actual translation that produce intermediate code.

4.Data-flow analysis engine

Facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization

5. Code-generator

produce a code generator from a collection of rules for translating each operation of the Intermediate language into the machine language for a target machine

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=Qkwj65l_96I

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 28-31 and 34-35.

Course Teacher

Verified by HOD

L-9

LECTURE HANDOUTS

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : I **Date of Lecture:**

Topic of lecture: Role of Lexical Analyzer, Input Buffering, Tokens Specification

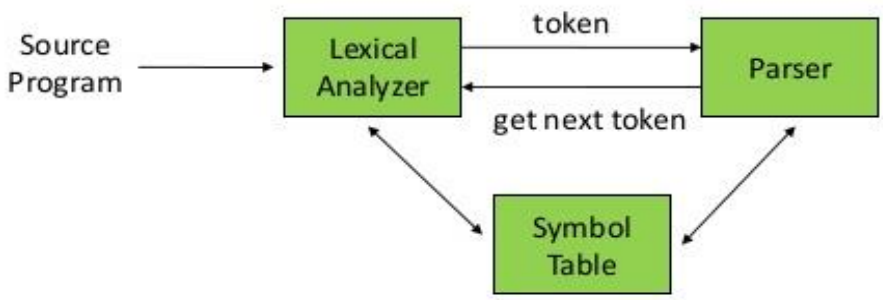
Introduction: (Maximum 5 sentences): Lexical analysis is the first phase of compilation process in which it translates source statement and generates sequences of token as output

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Basic concepts of complier
- Phases of Compiler

Detailed content of the Lecture:

- Lexical Analysis is the first phase of compiler
- It reads the input characters from left to right, one character at a time, from the Source program
- It generates the sequence of tokens for each lexeme
- Each token is a logical cohesive unit such as identifiers, keywords, operators and Punctuation marks.
- It enters that lexeme into the symbol table and also reads from the symbol table



Functions of lexical analyzer

- It produces stream of tokens
- It eliminates comments and whitespace
- It keeps track of line numbers
- It reports the error encountered while generating tokens

- It stores information about identifiers, keywords, constants into symbol table

Lexical analyzers are divided into two processes

- **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one
- **Lexical analysis** is the more complex portion, that produces the sequence of tokens as output

Issues in Lexical analysis

- Simplicity of design :The separation of lexical and syntactic analysis often allows us to simplify tasks. whitespace and comments removed by the lexical analyzer
- Compiler efficiency is improved : Specialized buffering techniques for reading input characters can speed up the compiler significantly
- Compiler portability is enhanced : Input-device-specific peculiarities can be restricted to the lexical analyze

Tokens, Patterns, and Lexemes

i.Token: Sequence of characters that can be treated as a single logical entity.

Eg: Number, Identifiers, keywords, etc..

ii. Pattern: Set of strings is described by a rule called a pattern

iii. Lexeme: Sequence of characters in the source program that is matched by the pattern for a token

iv. Attribute of the token Information about an identifier - e.g., its lexeme, its type, and the location at which it is first found - is kept in the symbol table

Lexical Errors

panic mode" recovery:

The simplest recovery strategy is "**panic mode**" recovery (delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left)

Other possible error-recovery actions are:

1. Delete one character from the remaining input
2. Insert a missing character into the remaining input
3. Replace a character by another character
4. Transpose two adjacent characters

Input Buffering, Tokens Specification

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Two pointers *lexemeBegin* and *forward* are maintained.

lexeme Begin points to the beginning of the current lexeme which is yet to be found.

forward scans ahead until a match for a pattern is found.

• Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.

• Current lexeme is the set of characters between two pointers.

Initially both the pointers point to the first character of the input string as shown below

Buffer pair:

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.

- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.
- *eof* is inserted at the end if the number of characters is less than N.

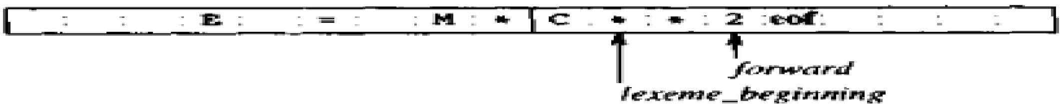


Fig. 3.3. An input buffer in two halves.

Sentinels:

- Sentinels is a special character that can not be part of the source program. (*eof* character is used as sentinel).
- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- • Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

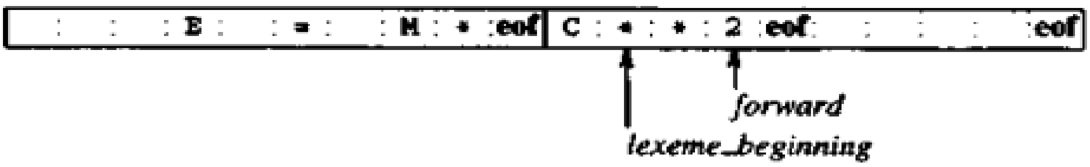


Fig. 3.5. Sentinels at end of each buffer half.

Specifications of Tokens

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

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

Strings: Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=Qkwj65l_96I,
https://www.youtube.com/watch?v=Rl0tIp3hbbs

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 96-97, 100-109

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

LECTURE HANDOUTS | **L - 10,11**

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Recognition Machine

Introduction : (Maximum 5 sentences)

- Lexical Analysis reads source program character by character and produces a stream of tokens.
- Recognize the tokens with the help of transition diagrams.

Prerequisite knowledge for Complete understanding and learning of Topic:

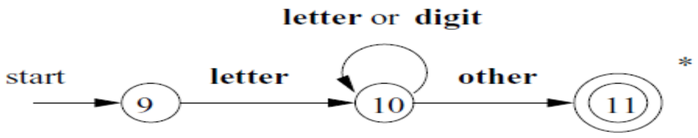
- Lexical Analysis

Detailed content of the Lecture:

- Lexical Analysis reads source program character by character and produces a stream of tokens.
- Token may be identifier, keyword, operator or constants. Tokens are specified with the help of Regular expression. Recognize the tokens with the help of transition diagrams.
    1. Recognition of Identifier
    2.Recognition of delimiters
    3.Recognition of Relational operators
    4. Recognition of keywords
    5. Recognition of numbers

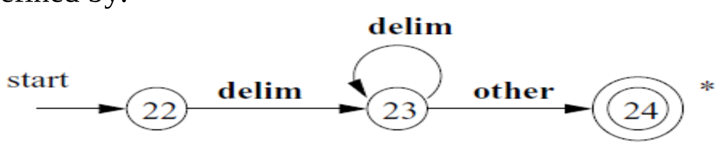
1. **Recognition of Identifier**

   Letter = a|b|c|..|z| A|B|…|Z|
   Digit = 0|1|2|…|9|
   Id=letter(letter|digit)*



2. Recognition of delimiters

- The lexical analyzer the job of stripping out white space, by recognizing the “token" ws defined by:



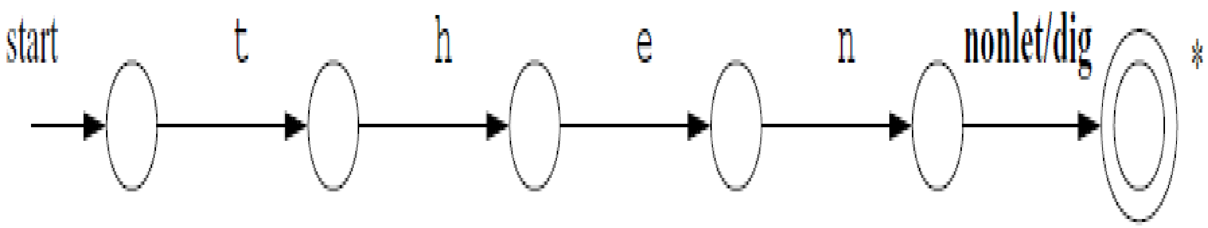
3. Recognition of Relational operators

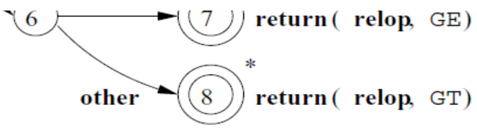
- a transition diagram that recognizes the lexemes matching the token relop. We begin in state 0, the start state.
- If we see < as the first input symbol, then among the lexemes that match the pattern for relop we can only be looking at <, <>, or <=.

- We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator.
- If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information.
- Note, however, that state 4 has a * to indicate that we must retract the input one position
- < | <= | = | <> | > | >=

.4. Recognition of keywords

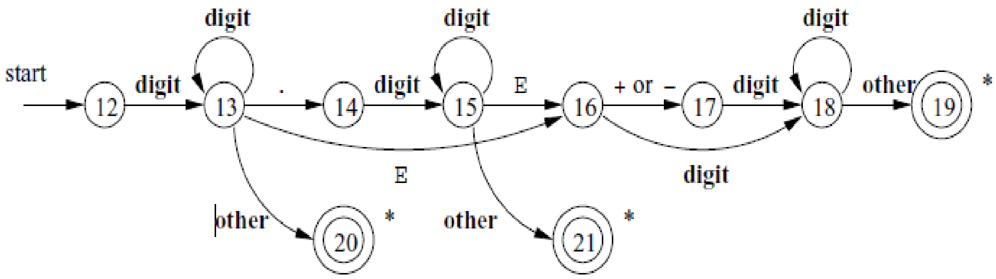
- if -> if
- then -> then
- else -> else





5. Recognition of numbers

- Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits.
- However, if we see anything but a digit, dot, or E, we have seen a number in the form of an integer; 123 is an example.
- That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered.



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 131-134

Course Faculty

Verified by HOD

LECTURE HANDOUTS

L - 12

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Typical Lexical Analysis Generator

Introduction : (Maximum 5 sentences)

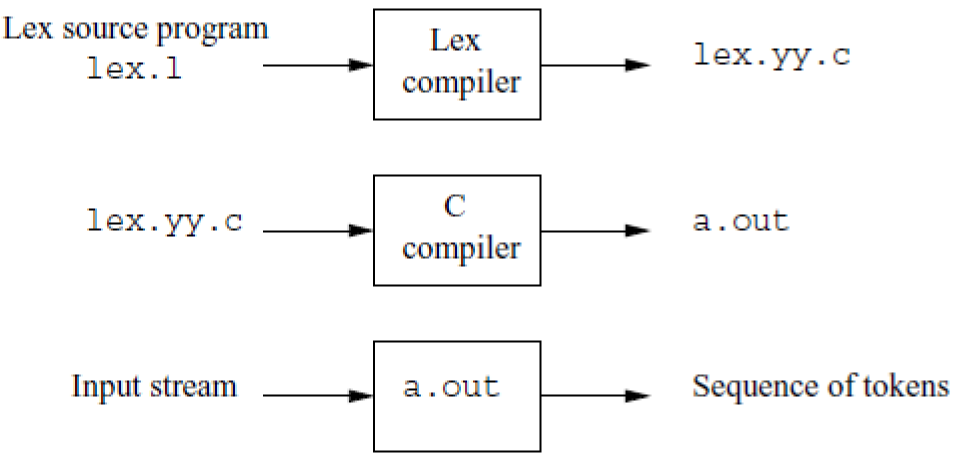
- Lexical Analysis reads source program character by character and produces a stream of tokens.
- Recognize the tokens with the help of transition diagrams.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Lexical Analysis

Detailed content of the Lecture:

- 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.



Structure of Lex programs

```
{ 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 action it 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.
- **Example**

```
%{
        /* definitions of manifest constants
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim           [ \t\n]
ws                      {delim}+
Letter          [A-Za-z]
Digit           [0-9]
id                      {letter}({letter}|{digit})*
number                  {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}            {/* no action and no return */}
if                      {return(IF);}
Then            {return(THEN);}
else            {return(ELSE);}
{id}            {yylval = (int) installID(); return(ID); }
{number}        {yylval = (int) installNum();
                 return(NUMBER);}
```

Action :

1. Int installID()

   This function is called to place the lexeme found in the symbol table.

   Two variables are used

   1. yytext – lexemebegin pointer
   2. yyleng – length of the lexeme

   Token name ID is returned to the parser.
2. Int installNum()
   - Lexeme matching the pattern number.

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 140-145

Course Faculty

Verified by HOD

LECTURE HANDOUTS

L - 13

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Parsing

Introduction : (Maximum 5 sentences)

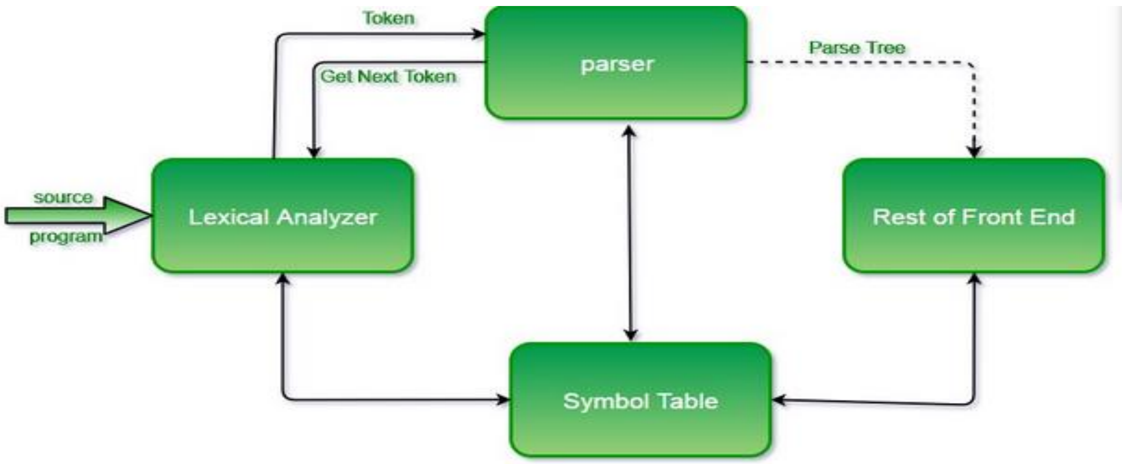
- **Parser** is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Phases of Compiler
- Lexical Analysis

Detailed content of the Lecture:

- 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 build the parse trees:
1. 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.
2. 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.

Context Free Grammers:

- The syntax of a programming language is described by a context free grammar (CFG). CFG consists of set of terminals, set of non terminals, a start symbol and set of productions.
  G=(V,T,P,S)
  T – finite set of Terminals(i.e Token)

V – finite set of Variables or Non terminals(related strings)
P – Finite set of Productions

Ambiguity :

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
  Eg- consider a grammar
  S -> aS | Sa | a ,Now for string aaa we will have 4 parse trees .

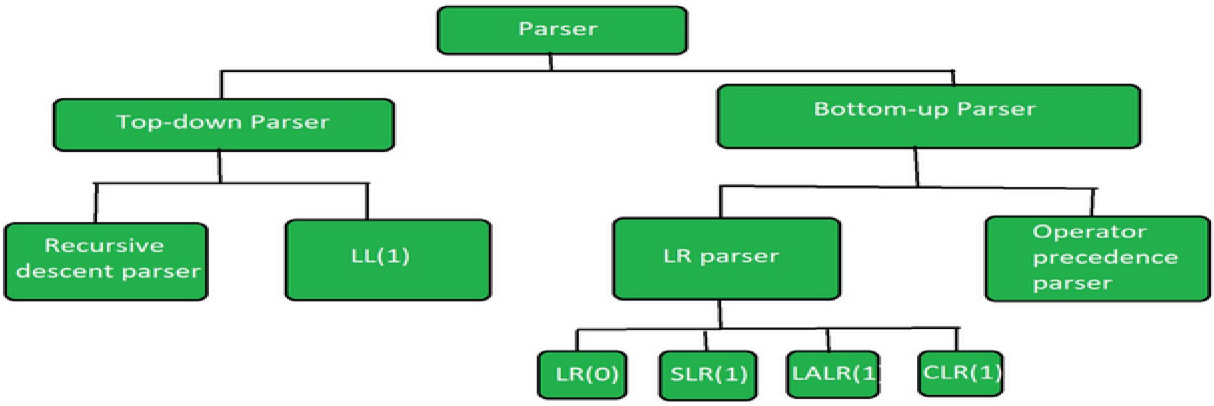
Left Recursion:

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

Left factoring:

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

Consider the grammar ,
S → iEtS | iEtSeS | a
E → b
Left factored, this grammar becomes
S → iEtSS' | a
S' → eS | ε
E → b



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 215-217

Course Faculty

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

LECTURE HANDOUTS

L - 14

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Top Down Parsing

Introduction : (Maximum 5 sentences)

- 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.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Phases of Compiler
- Parsing

Detailed content of the Lecture:

- 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.

Classification of Top-Down Parsing

1. **With Backtracking:** Recursive Descent Parsing
2. **Without Backtracking:** Predictive Parsing or Non-Recursive Parsing or LL(1) Parsing or Table Driver Parsing

For example, let us consider the grammar to see how top-down parser works:

S-> if E then S else S | while E do S | print
E -> true | False | id

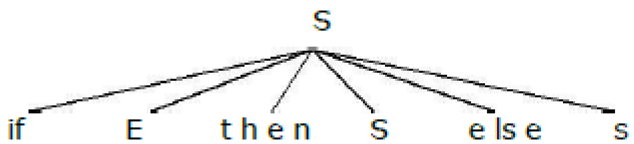
The input token string is: If id then while true do print else print.

1. Tree:

S

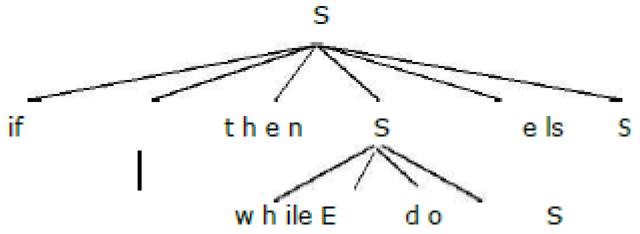
Input: if id then while true do print else print. And Action: Guess for S.

2. Tree



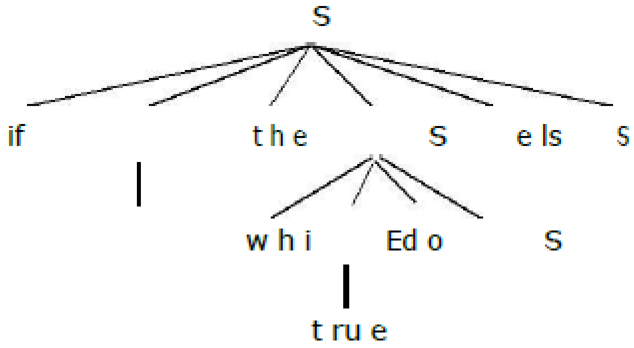
Input: if id then while true do print else print. Action: if matches; guess for E.

3. Tree :



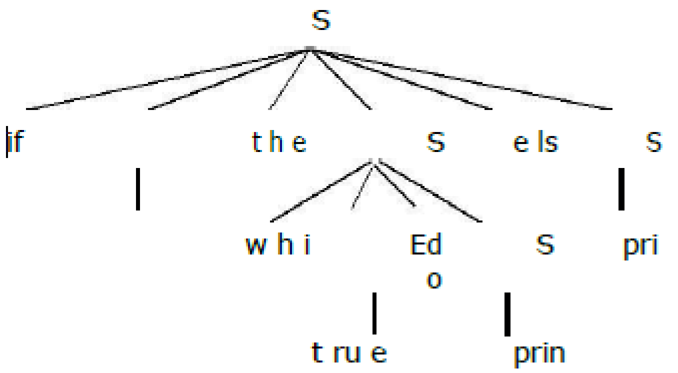
Input: while true do print else print. Action: while matches; guess for E.

4.Tree :



Input: true do print else print Action:true matches; do matches; guess S.

5. Tree:



Input: print. Action: print matches; input exhausted; done.

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 217-218

Course Faculty

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu
Estd. 2000

LECTURE HANDOUTS | **L - 15**

IT | **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Faculty : T.Manivel

Unit : II **Date of Lecture:**

Topic of Lecture: Recursive Descent Parser

Introduction : (Maximum 5 sentences)

- 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.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Parsing
- Top down Parsing

Detailed content of the Lecture:

- 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 a 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.

Example for backtracking :

Consider the grammar

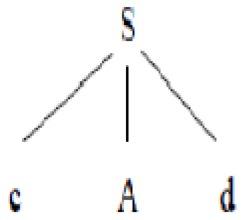
G : S → cAd

A → ab | a

input string w=cad.

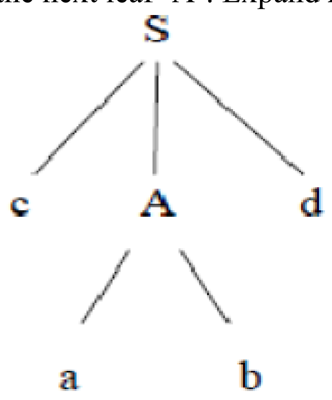
Step1:

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



Step2:

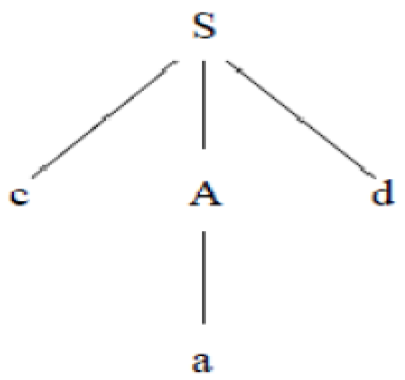
- The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’. Expand A using the first alternative.



Step3:

- The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’. But the third leaf of tree is b which does not match with the input symbol d.
- Hence discard the chosen production and reset the pointer to second position. **This is called backtracking.**

Step4: Now try the second alternative for A.



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

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 219-220

Course Faculty

Verified by HOD

LECTURE HANDOUTS | **L - 16**

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Recursive Descent Parser

Introduction : (Maximum 5 sentences)

- 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.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Parsing
- Top down Parsing

Detailed content of the Lecture:

Recursive-Descent Parsing Function:

```
void A( )
{
    Choose an A- production, A -> X1,X2,  ……Xk;
    for ( i = 1 to k )
       {
         if ( Xi is a nonterminal )
                call procedure Xi( );
         else if ( Xi equals the current input symbol a )
                advance the input to the next symbol;
         else /* an error has occurred */;
       }
}
```

Example for backtracking :

- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).
- To understand this, take the following example of CFG:

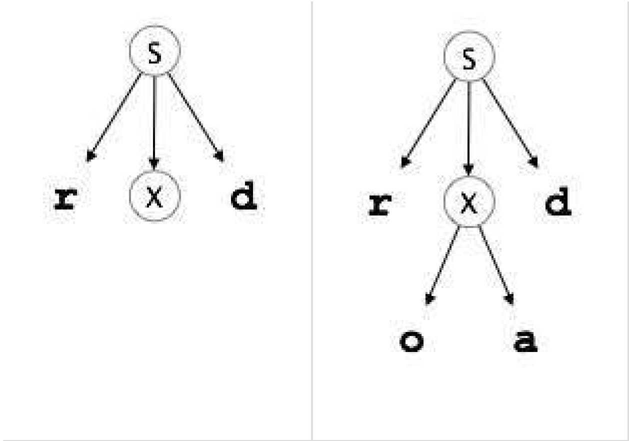
  S → rXd | rZd
  X → oa | ea
  Z → ai

For an input string: read, a top-down parser, will behave like this:

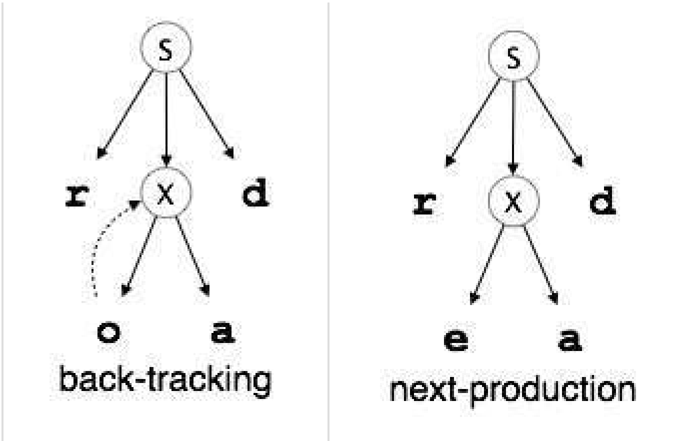
- It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it.
- So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa).
- It does not match with the next input symbol. So the top-down parser backtracks to obtain the

next production rule of X, (X → ea).

Now the parser matches all the input letters in an ordered manner. The string is accepted.



Apply Backtracking



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 219-220

Course Faculty

Verified by HOD

LECTURE HANDOUTS | **L - 17**

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Predictive Parsing

Introduction : (Maximum 5 sentences)

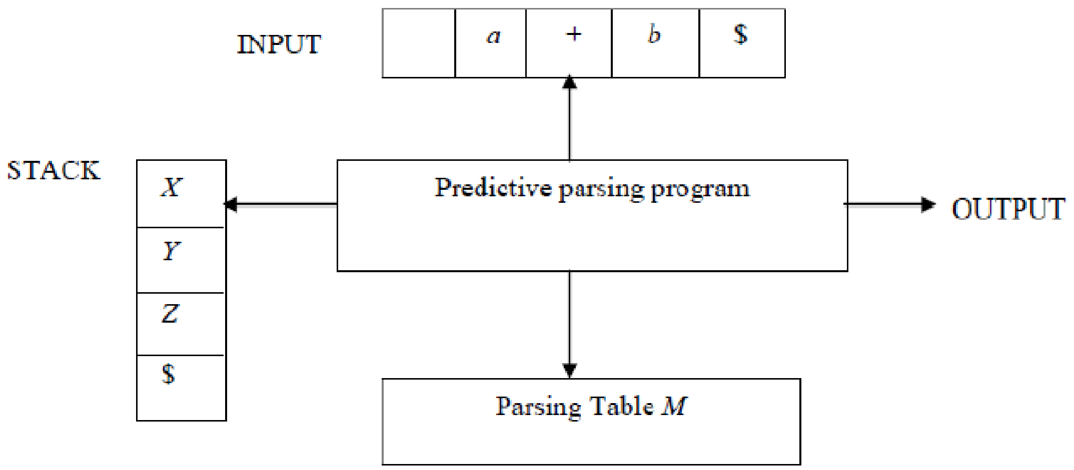
- A Predictive parser is one of the working models of a Top-down parser, which follows recursive decent parsing without backtracking.
- This parser can be implemented non-recursivley, by using stack data structure..

Prerequisite knowledge for Complete understanding and learning of Topic:

- Syntax Analysis
- Parsing

Detailed content of the Lecture:

- A Predictive parser is one of the working models of a Top-down parser, which follows recursive decent parsing without backtracking.
- This parser can be implemented non-recursivley, by using stack data structure.
- Hence, the following preprocessing steps are to be performed in the given grammar,
  1. Elimination of Left Recursion
  2. Elimination of Left Factoring



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

Input buffer:

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

Stack:

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

Parsing table:

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

Predictive parsing program:

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

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

Algorithm for nonrecursive predictive parsing:
Input : A string w and a parsing table M for grammar G.
Output : If w is in L(G), a leftmost derivation of w; otherwise, an error indication.
Method : Initially, the parser has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
let X be the top stack symbol and a the symbol pointed to by ip;
if X is a terminal or $ then
if X = a then
pop X from the stack and advance ip
else error()
else /* X is a non-terminal */
if M[X, a] = X →Y1Y2 … Yk then begin
pop X from the stack;
push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
output the production X → Y1 Y2 . . . Yk
end
else error()
until X = $
```

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 220-224

Course Faculty

Verified by HOD

LECTURE HANDOUTS

L - 18

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **II** **Date of Lecture:**

Topic of Lecture: Predictive Parsing

Introduction : (Maximum 5 sentences)

- A Predictive parser is one of the working models of a Top-down parser, which follows recursive decent parsing without backtracking.
- This parser can be implemented non-recursivley, by using stack data structure..

Prerequisite knowledge for Complete understanding and learning of Topic:

- Syntax Analysis
- Parsing

Detailed content of the Lecture:

Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-1 => ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

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

Example:

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

Step 1: After eliminating left-recursion the grammar is

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Step 2 : Computation of First( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

Computation of Follow( ):

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Step 3: Parse Table

Non Terminal	Id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'-> Ɛ	E'-> Ɛ
T	T->FT'		T->FT'			
T'		T'-> Ɛ	T'->*FT'		T'-> Ɛ	T'-> Ɛ
F	F->id			T->(E)		

Step 4 :Stack Implementation

STACK	INPUT	ACTION
E$	Id+id $	E->TE'
TE'$	Id+id$	T->FT'
FT'E'$	Id+id$	F->id
IdT'E'$	Id+id$	POP id
T'E'$	+id$	T'-> Ɛ
E'$	+id$	E'->+TE'
+TE'$	+id$	POP +
TE'$	id$	T->FT'
FT'E'$	id$	F->id
IdT'E'$	id$	POP id
T'E'$	$	T'-> Ɛ
E' $	$	E'-> Ɛ
$	$	Accept

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 224-228

Course Faculty

Verified by HOD

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : III **Date of Lecture:**

Topic of lecture: Role of the parser – Context-Free Grammars

Introduction: (Maximum 5 sentences):

- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.
- A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.
- Parsing is of two types: top down parsing and bottom up parsing.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Introduction to Compiler
- The Phases of Compiler
- Role of Lexical analyzer

Detailed content of the Lecture:

Role of Parser:

- The compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.
- The parser returns any syntax error for the source language.

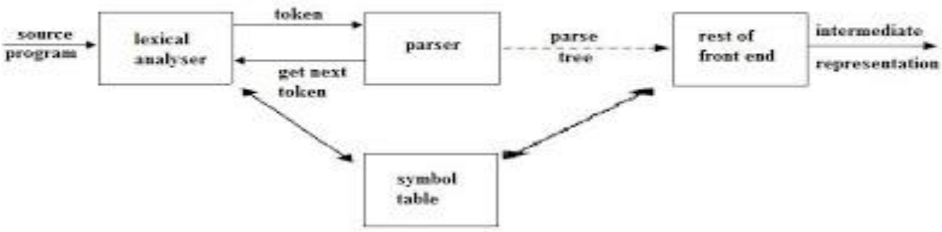


Fig 2.1 Position of parser in compiler model

The methods commonly used in compilers are classified as either top-down parsing or bottom-up parsing.
Top-down parsers build parse trees from the top (root) to the bottom (leaves).
Bottom-up parsers build parse trees from the leaves and work up to the root.
In both case input to the parser is scanned from left to right, one symbol at a time.
The output of the parser is some representation of the parse tree for the stream of tokens.
There are number of tasks that might be conducted during parsing. Such as;

- Collecting information about various tokens into the symbol table.
- Performing type checking and other kinds of semantic analysis.
- Generating intermediate code.
- Syntax Error Handling:
- Planning the error handling right from the start can both simplify the structure of a compiler and

improve its response to errors.

Much of the error detection and recovery in a compiler is centered on the syntax analysis phase.

- One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
- Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.

The error handler in a parser has simple goals:

- It should 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.

Error-Recovery Strategies:

- There are many different general strategies that a parser can employ to recover from a syntactic error.
- Panic mode
- Phrase level
- Error production
- Global correction

Panic mode:

- This is used by most parsing methods.
- On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens (delimiters; such as; semicolon or end) is found.
- Panic mode correction often skips a considerable amount of input without checking it for additional errors.
- It is simple.

Phrase-level recovery:

- On discovering an error; the parser may perform local correction on the remaining input; i.e., it may replace a prefix of the remaining input by some string that allows the parser to continue.
- e.g., local correction would be to replace a comma by a semicolon, deleting an extraneous semicolon, or insert a missing semicolon.

Error productions:

- If an error production is used by the parser, can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

Global correction:

- Given an incorrect input string x and grammar G, the algorithm will find a parse tree for a related string y, such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.

Context-Free Grammars :

G=(NT , T, P, S)

- NT is a finite set of non-terminals
- T is a finite set of terminals
- P is a finite subset of production rules
- S is the start symbol

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=QGz6tapRckQ

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 172-183

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-20

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: Bottom up parsing

Introduction: (Maximum 5 sentences):

- **B**ottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- we start from a sentence and then apply rightmost production rules in reverse manner in order to reach the start symbol.

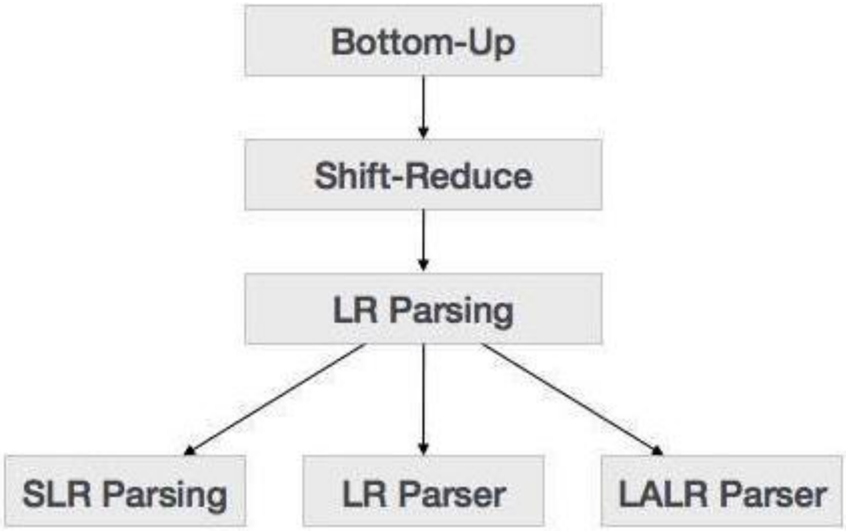
Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser
- Concepts of CFG

Detailed content of the Lecture:

Bottom up parsing:

The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

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

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is

known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

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

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

Handle : A substring that matches the right side of a production called handle

Handle pruning :Applying the production to the substring results in a *right-sentential form, i.e.,* a sentential form occurring in a right-most derivation called handle pruning

E ::= E+E   E ::= E*E   E ::= (E)   E ::= id

id + id * id

E + id * id

E + E * id

E + E * E

E+E

E

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=6TvYuJyHHqk

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 207-208

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L -21

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : III **Date of Lecture:**

Topic of lecture: Shift reduce parser

Introduction: (Maximum 5 sentences):

- Build the parse tree from leaves to root.
- Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser
- Bottomup parsing

Detailed content of the Lecture:

Shift reduce parser:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.

This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Actions of SR Parser

a. **Shift:** This involves moving of symbols from input buffer onto the stack.

b. **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

c. **Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

d. **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example

Grammar: E→E*E / E+E / id   Input string: id1+id2*id3

	Stack	Input	Action
(1)	$	$id_1 + id_2 * id_3\$$	shift
(2)	$\$id_1$	$+ id_2 * id_3\$$	reduce by $E \to id$
(3)	$\$E$	$+ id_2 * id_3\$$	shift
(4)	$\$E +$	$id_2 * id_3\$$	shift
(5)	$\$E + id_2$	$* id_3\$$	reduce by $E \to id$
(6)	$\$E + E$	$* id_3\$$	shift
(7)	$\$E + E *$	$id_3\$$	shift
(8)	$\$E + E * id_3$	$	reduce by $E \to id$
(9)	$\$E + E * E$	$	reduce by $E \to E * E$
(10)	$\$E + E$	$	reduce by $E \to E + E$
(11)	$\$E$	$	accept

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=HEl_0CMs3Rg
https://www.geeksforgeeks.org/shift-reduce-parser-compiler/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 208 - 214

Course Teacher

Verified by HOD

LECTURE HANDOUTS

L-22

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : III **Date of Lecture:**

Topic of lecture: Operator precedence parsing-precedence relations

Introduction: (Maximum 5 sentences):

- Build the parse tree from leaves to root.
- Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser
- Bottomup parsing

Detailed content of the Lecture:

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Eg. E → E op E | id
op → + | *
The above grammar is not an operator grammar but:
E → E + E | E* E | id

Operator precedence can only establish between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".
a ⋖ b means that terminal "a" has the lower precedence than terminal "b".
a ≐ b means that the terminal "a" and "b" both have same precedence.

Note:

- **id has higher precedence than any other symbol**
- **$ has lowest precedence.**
- **if two operators have equal precedence, then we check the Associativity of that particular operator.**

Consider the following grammar-

E → EAE | id
A → + | x

Construct the operator precedence parser and parse the string id + id x id.

Step-01:

We convert the given grammar into operator precedence grammar.
The equivalent operator precedence grammar is-

$$E \rightarrow E + E \mid E \times E \mid id$$

Step-02:

The terminal symbols in the grammar are { id, + , x , $ }
We construct the operator precedence table as-

	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

```
w ← input
a ← input symbol
b ← stack top
  Repeat
  {
    if(a is $ and b is $)
        return
    if(a .> b)
        push a into stack
        move input pointer
    else if(a <. b)
        c ← pop stack
        until(c .> b)
    else
        error()
  }
```

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=n5UWAaw_byw

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 215-226

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-23

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: Operator precedence parsing-precedence functions

Introduction: (Maximum 5 sentences):

- Operator precedence grammar is kinds of shift reduce parsing method.
- It is applied to a small class of operator grammars.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Types of parser
- Operator precedence parsing RF

Detailed content of the Lecture:

- There is also a disadvantage of operator precedence table – if we have n operators then size of table will be n*n and complexity will be $0(n^2)$. In order to decrease the size of table, we use **operator function table**.
- Operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way.
- Operator precedence parsers use **precedence functions** that map terminal symbols to integers, and the precedence relations between the symbols are implemented by numerical comparison.
- The parsing table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers. We select f and g such that:

1. f(a) < g(b) whenever a yields precedence to b
2. f(a) = g(b) whenever a and b have the same precedence
3. f(a) > g(b) whenever a takes precedence over b

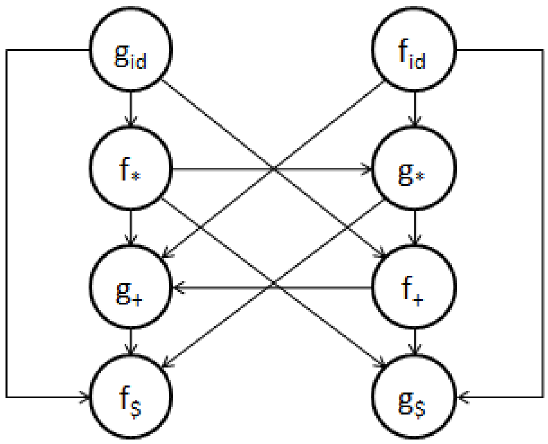
Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal a and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a \doteq b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a \lessdot b$, otherwise if $a \gtrdot b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively

- **Consider the following table:**

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

- Resulting graph:



- From the previous graph we extract the following precedence functions:

	id	+	*	$
f	4	2	4	0
id	5	1	3	0

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=oYAy_-76n-Q

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 215-226

Course Teacher

Verified by HOD

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: LR Parser-SLR Parser

Introduction: (Maximum 5 sentences):.

- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as LR(k) parsers.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser

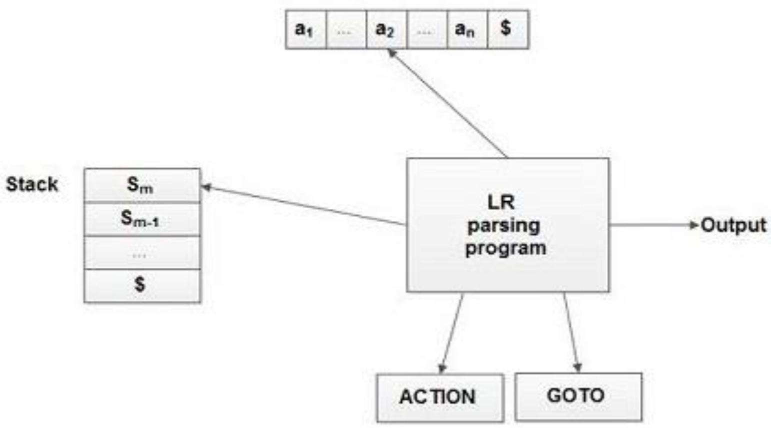
Detailed content of the Lecture:

LR Parser:

- LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.
- In the LR parsing, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a right most derivation in reverse.
- "K" is the number of input symbols of the look ahead used to make number of parsing decision.
- LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

LR algorithm:

- The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.



- Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.
- A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.
- Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

•

Input : Grammar G, Input string w, Stack

Method :

1.Compute Augmented grammar (S'→.S) S-Start symbol of the grammar

2. Compute Canonical LR(0) Collection of (S'→.S)

(i) Function Closure for given grammar (I_0)

(ii) Function Go-to(I_i, X) for each Item

3. Construct Parse table (action & goto)

4. Construct Parse Tree

Augment Grammar

Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Closure of item sets

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

• Initially, add every item I to CLOSURE(I).

• If A —> αB,β is in CLOSURE(I) and B —> γ is a production, then add the item B —> • γ to CLOSURE(i), if it is not already there. Apply this rule until no more items can be added to CLOSURE (i).

Construct canonical LR(O) collection

• Augmented grammar is defined with two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then augmented grammar G′ is G with a new start symbol S′ —> S.

• The role of augmented production is to stop parsing and notify the acceptance of the input i.e., acceptance occurs when and only when the parser performs reduction by S′ —> S.

Constructing SLR Parsing Table

1. Construct C={I0,I1,…,In}, the collection of sets of LR(0) items for G′ (augmented grammar).
2. Columns are the Terminal Symbols For ACTION and NonTerminal Symbols for GOTO.
3. If [A→a•ab] is in Ii where a is a terminal and goto(Ij,a)=Ij, the set action[i,a] to "shift j".
4. If [A→a•ab] is in Ii where a is a non -terminal and goto(Ij,A)=Ij, the set Goto[i,a] to " j".
5. If [S′→S•] is in Ii, then set action[i,$] to "accept".
6. If [A→a•] is in Ii, then set action[i,a] to "reduce A→a" for all a in FOLLOW(A).
7. All other entries are called Error

   Initial state of the parser is the state corresponding to the set of items including [S′ --> .S]

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://ecomputernotes.com/compiler-design/lr-parsers

https://www.youtube.com/watch?v=APJ_Eh60Qwo

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 227-228

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-25

LECTURE HANDOUTS

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: LR Parser-SLR Parser

Introduction: (Maximum 5 sentences):

- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as LR(k) parsers.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser
- LR Parser-SLR Parser

Detailed content of the Lecture:

LR Parsing Algorithm

Input : Input string w,LR-Parsing table with functions ACTION and GOTO for a grammar G

Output : If w is in L(G), the reduction steps of a bottom-up parse for w, otherwise, an error indication.

Method

Initially, the parser has So on its stack, where So is the initial state, and w $ in the input buffer.

let a be the first symbol of w $

while(l) { //repeat forever

let s be the state on top of the stack;

if(ACTION[s, a] =shift t {

push t onto the stack;

let a be the next input symbol;

} else if (ACTION [s, a] = reduce A—> β) {

pop β symbols off the stack;

let state t now be on top of the stack;

push GOTO[t, A] onto the stack;

output the production A—> β;

} else if (ACTION [s, a] accept) break;

//parsing is done

else call error-recovery routine;

}

	STACK	INPUT	ACTION
1	0	id*id+id$	S5
2	0,id,5	*id+id$	R6(F ---> id
3	0,F	*id+id$	3
4	0,F,3	*id+id$	R4(T ---> F)
5	0,T	*id+id$	2
6	0,T,2	*id+id$	S7
7	0,T,2,*,7	id+id$	S5
8	0,2,*,7,id,5	+id$	R6(F ---> id
9	0,T,2,*,7,F,10	+id$	R6 (F ---> id
10	0,T,2	+id$	R2
11	0,E,1	+id$	S6
12	0,E,1,+,6	id$	S5
13	0,E,1,+,6,id,5	$	R6(F ---> id
14	0,E,1,+,6,F,3	$	R4(T ---> F)
15	0,E,1,+,6,T,9	$	R1(E --->
16	0,E,1	$	Accept

SLR Parse Table

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Video Content / Details of website for further learning (if any):
https://ecomputernotes.com/compiler-design/lr-parsers
https://www.youtube.com/watch?v=APJ_Eh60Qwo

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 227-228

Course Teacher

Verified by HOD

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: Canonical LR Parser

Introduction: (Maximum 5 sentences):

- In CLR parsing we will be using LR(1) items
- LR(1) parsers are more powerful parser.
- For LR(1) items we modify the Closure and GOTO function.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Role of parser
- Types of parser
- SLR parser

Detailed content of the Lecture:
Canonical LR Parser-example:

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

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>Constrcution of Set of LR(1) items.</u>

I_0:
S → •S, $
S → •CC, $
C → •cC, c/d
C → •d, c/d

I_1: GOTO(I_0, S)
S → S•, $

I_2: GOTO(I_0, C)
S → C•C, $
C → •cC, c/d
C → •d, c/d

I_3: GOTO(I_0, c)
C → c•C, c/d
C → •cC, c/d
C → •d, c/d

I_4: GOTO(I_0, d)
C → d•, c/d

I_5: GOTO(I_2, C)
S → CC•, $

I_6: goto(I_2, c)
C → c•C, $
C → •cC, $
C → •d, $

I_7: GOTO(I_2, d)
C → d•, $

I_8: GOTO(I_3, C)
C → cC•, c/d

I_9: GOTO(I_6, C)
C → cC•, $

Construction of CLR parsing table-

Input – augmented grammar G'

1. Construct C = { I0, I1, ……. In} , the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :
   i) If [A -> ?.a?, b] is in Ii and GOTO(Ii , a) = Ij, then set ACTION[i, a] to "shift j". Here a must be terminal.
   ii) If [A -> ?. , a] is in Ii , A ≠ S, then set ACTION[i, a] to "reduce A -> ?".
   iii) Is [S -> S. , $] is in Ii, then set action[i, $] to "accept".
   If any conflicting actions are generated by the above rules we say that the grammar is not CLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(Ii, A) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

CLR Parsing Algorithm

Input : Input string w,LR-Parsing table with functions ACTION and GOTO for a grammar G

Output : If w is in L(G), the reduction steps of a bottom-up parse for w, otherwise, an error indication.

Method

Initially, the parser has So on its stack, where So is the initial state, and w $ in the input buffer.

let a be the first symbol of w $

while(l) { //repeat forever

let s be the state on top of the stack;

if(ACTION[s, a] =shift t {

push t onto the stack;

let a be the next input symbol;

} else if (ACTION [s, a] = reduce A—> β) {

pop β symbols off the stack;

let state t now be on top of the stack;

push GOTO[t, A] onto the stack;

output the production A—> β;

} else if (ACTION [s, a] accept) break;

//parsing is done

else call error-recovery routine;

}

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.youtube.com/watch?v=fpPzWswvkJw

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 228-250

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-28

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **III** **Date of Lecture:**

Topic of lecture: LALR Parser

Introduction: (Maximum 5 sentences):

- LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- SLR parser
- CLR parser

Detailed content of the Lecture:

LALR Parser:

Consider the following augmented grammar. S --> CC , C --> c C / d Construct parsing table for LALR(1) parser and parse for the string ccdd.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOT0 for G'.

METHOD:

1. Construct C = ($I_0, I_1, \ldots, I_n$), the collection of sets of LR (1) items.
2. For each core present among the set of LR (1) items, **find all sets having same core in first part, and replace these sets by their union.**
3. Let C' = {$J_0, J_1, \ldots, J_m$} be the resulting sets of LR(1) items. The parsing actions for state *i* are constructed from J_i. 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 = $I_1 \cap I_2 \cap \ldots \cap I_k$, then the cores of GOTO(I_1, X) Let K be the union of all sets of items having the same core as GOTO(I_1, X) . Then GOTO(J, X) = K.

Constrcution of Set of LR(1) items.

I_0:
S → •S, $
S → •CC, $
C → •cC, c/d
C → •d, c/d

I_1: GOTO(I_0, S)
S → S•, $

I_2: GOTO(I_0, C)
S → C•C, $
C → •cC, c/d
C → •d, c/d

I_3: GOTO(I_0, c)
C → c•C, c/d
C → •cC, c/d
C → •d, c/d

I_4: GOTO(I_0, d)
C → d•, c/d

I_5: GOTO(I_2, C)
S → CC•, $

I_6: goto(I_2, c)
C → c•C, $
C → •cC, $
C → •d, $

I_7: GOTO(I_2, d)
C → d•, $

I_8: GOTO(I_3, C)
C → cC•, c/d

I_9: GOTO(I_6, C)
C → cC•, $

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

Parsing the input string "ccdd"

Stack	Input buffer	Action table	Goto table	Parsing action
$0	ccdd$	action[0, c]=s36		
$0c36	cdd$	action[36, c]=s36		Shift
$0c36c36	dd$	action[36, d]=s47		Shift
$0c36c36d47	d$	action[47, d]=r36	[36,C]=89	Reduce by C → d
$0c36c36C89	d$	action[89, d]=r2	[36,C]=89	Reduce by C → cC
$0c36C89	d$	action[89, d]=r2	[0, C]=2	Reduce by C → cC
$0C2	d$	action[2, d]=s47		Shift
$0C2d47	$	action[47, $]=r36	[2, C]=5	Reduce by C → d
$0C2C5	$	action[5, $]=r1	[0, S]=1	Reduce by S → CC
$0S1	$	accept		

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=-lGU9jCDu3w

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 250-269

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS

L - 28,29

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Intermediate Languages

Introduction : (Maximum 5 sentences)

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

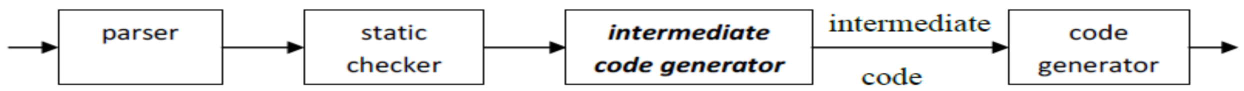
Prerequisite knowledge for Complete understanding and learning of Topic:

- Phases of Compiler

Detailed content of the Lecture:

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

Position of intermediate code generator



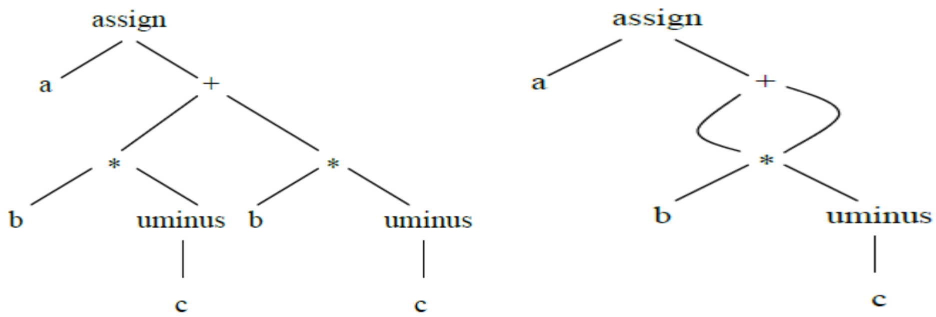
INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

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

Syntax tree:

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



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

Postfix notation:

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

  a b c uminus * b c uminus * + assign

Three-Address Code:

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

  x : = y op z

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

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

- where t1 and t2 are compiler-generated temporary names.

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

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

(a) Code for the syntax tree **(b) Code for the dag**

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 363-370

Course Teacher

Verified by HOD

LECTURE HANDOUTS **L - 30**

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Declarations

Introduction : (Maximum 5 sentences)

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

Prerequisite knowledge for Complete understanding and learning of Topic:

- Intermediate Languages

Detailed content of the Lecture:

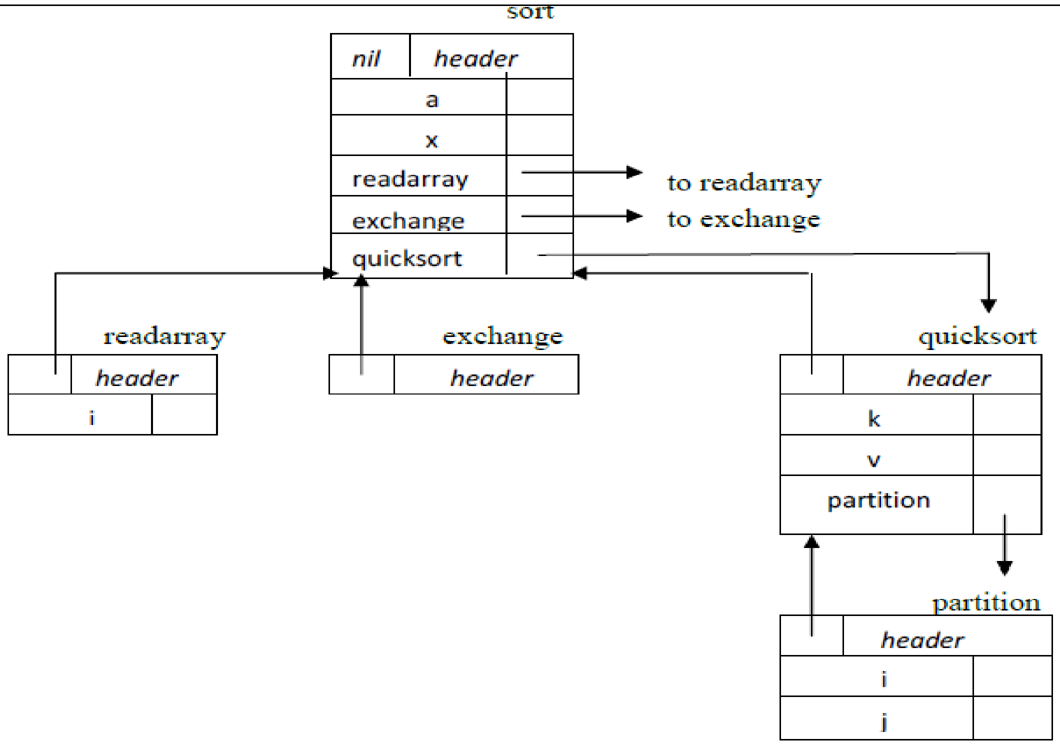
- The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next available relative address.

In the translation scheme shown below:

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

  Computing the types and relative addresses of declared names

D → id : T	*{ enter(id.name, T.type, offset);* *offset : = offset + T.width }*
T → integer	*{ T.type : = integer;* *T.width : = 4 }*



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

1. mktable(previous) creates a new symbol table and returns a pointer to the new table. The argument previous points to a previously created symbol table, presumably that for the enclosing procedure.
2. enter(table, name, type, offset) creates a new entry for name name in the symbol table pointed to by table. Again, enter places type type and relative address offset in fields within the entry.
3. addwidth(table, width) records the cumulative width of all the entries in table in the header associated with this symbol table.
4. enterproc(table, name, newtable) creates a new entry for procedure name in the symbol table pointed to by table. The argument newtable points to the symbol table for this procedure name.

Syntax directed translation scheme for nested procedures

P → M D	***{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }***
M → ε	***{ t : = mktable (nil); push (t,tblptr); push (0,offset) }***

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 373-378

Course Teacher

Verified by HOD

LECTURE HANDOUTS

L - 31

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Assignment Statements

Introduction : (Maximum 5 sentences)

- In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Parsing
- Declarations

Detailed content of the Lecture:

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

```
P -> M D
M -> ε
D -> D ; D | id : T | proc id ; N D ; S
N -> ε
```

- Non terminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.
- Translation scheme to produce three-address code for assignments

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

Addressing Array Elements:

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

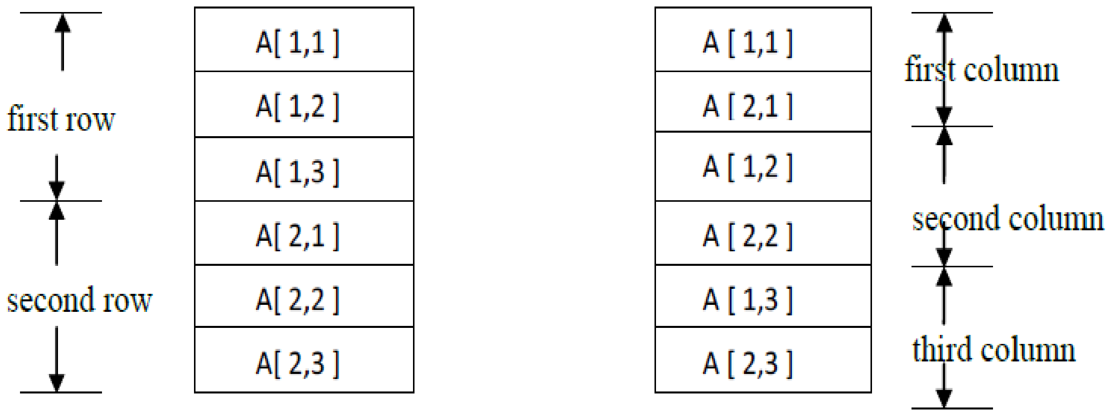
base + (i – low) x w

- where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is, base is the relative address of A[low].
- The expression can be partially evaluated at compile time if it is rewritten as

  i x w + (base – low x w)

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

1. Row-major (row-by-row)
2. Column-major (column-by-column)



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

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

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

Type conversion within Assignments :

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

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

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.javatpoint.com/translation-of-assignment-statements

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 378-383

Course Teacher

Verified by HOD

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Boolean Expressions

Introduction : (Maximum 5 sentences)

- Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E1 relop E2, where E1 and E2 are arithmetic expressions.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Declarations
- Assignment Statements

Detailed content of the Lecture:

- Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.
- Here we consider boolean expressions generated by the following grammar :
  E -> E or E | E and E | not E | (E) | id relop id | true | false

Methods of Translating Boolean Expressions:

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

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

Numerical Representation

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

For example :

- The translation for
  a or b and not c
  is the three-address sequence
  t1 : = not c
  t2 : = b and t1
  t3 : = a or t2
- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

Short-Circuit Code:

- We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression.

- This style of evaluation is sometimes called “short-circuit” or “jumping” code. It is possible to evaluate boolean expressions without generating code for the boolean operators and or, and not if we represent the value of an expression by a position in the code sequence.

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

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

Flow-of-Control Statements

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

```
 S -> if E then S1
| if E then S1 else S2
| while E do S1
```

- In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.
- In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.
- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.javatpoint.com/boolean-expressions

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 399-408

Course Teacher

Verified by HOD

LECTURE HANDOUTS | **L - 33**

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Case Statements

Introduction : (Maximum 5 sentences)

- Case or switch statements evaluate the controlling expression, Branch to the selected case, Execute the code for that case, Branch to the statements after the case.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Control Structures

Detailed content of the Lecture:

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

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

- There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, including a default "value" which always matches the expression if no other value does. The intended translation of a switch is code to:
  1. Evaluate the expression.
  2. Find which value in the list of cases is the same as the value of the expression.
  3. Execute the statement associated with the value found.
- Step (2) can be implemented in one of several ways :
- By a sequence of conditional goto statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say imin to imax, and the number of different values is a reasonable fraction of imax - imin, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j -imin and the label for the default in entries not filled otherwise. To perform switch, evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset j-imin .

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```
switch E
  begin
      case V1 : S1
      case V2 : S2
       . . .
      case Vn-1 : Sn-1
   default : Sn
end
```

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

```
        code to evaluate E into t
        goto test
L1 : code for S1
        goto next
L2 : code for S2
        goto next
         . . .
Ln-1 : code for Sn-1
          goto next
Ln : code for Sn
        goto next
test : if t = V1 goto L1
        if t = V2 goto L2
         . . .
        if t = Vn-1 goto Ln-1
        goto Ln
next :
```

To translate into above form :

- When keyword switch is seen, two new labels test and next, and a new temporary t are generated.
- As expression E is parsed, the code to evaluate E into t is generated. After processing E , the jump goto test is generated.
- As each case keyword occurs, a new label Li is created and entered into the symbol table. A pointer to this symbol-table entry and the value Vi of case constant are placed on a stack (used only to store cases).

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 418-420

Course Teacher

Verified by HOD

LECTURE HANDOUTS **L - 34**

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Back Paching

Introduction : (Maximum 5 sentences)

- In which lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified.
- Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Intermediate Languages
- Jumping statements

Detailed content of the Lecture:

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

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

Boolean Expressions:

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

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

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

- Consider production E E1 and M E2. If E1 is false, then E is also false, so the statements on E1.falselist become part of E.falselist. If E1 is true, then we must next test E2, so the target for the statements E1.truelist must be the beginning of the code generated for E2. This target is obtained using marker nonterminal M.
- Attribute M.quad records the number of the first statement of E2.code. With the production M -> ε we associate the semantic action
  { M.quad : = nextquad }
- The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E1.truelist when we have seen the remainder of the production E -> E1 and M E2. The translation scheme is as follows:

```
(1) E -> E1 or M E2     { backpatch ( E1.falselist, M.quad);
                         E.truelist : = merge( E1.truelist, E2.truelist);
                         E.falselist : = E2.falselist }
(2) E -> E1 and M E2  { backpatch ( E1.truelist, M.quad);
                         E.truelist : = E2.truelist;
                         E.falselist : = merge(E1.falselist, E2.falselist) }
(3) E -> not E1         { E.truelist : = E1.falselist;
                         E.falselist : = E1.truelist; }
(4) E -> ( E1 )          { E.truelist : = E1.truelist;
                         E.falselist : = E1.falselist; }
(5) E -> id1 relop id2  { E.truelist : = makelist (nextquad);
                         E.falselist : = makelist(nextquad + 1);
                        emit('if' id1.place relop.op id2.place 'goto_')
                        emit('goto_') }
(6) E -> true           { E.truelist : = makelist(nextquad);
                        emit('goto_') }
(7) E -> false          { E.falselist : = makelist(nextquad);
                        emit('goto_') }
(8) M -> ε              { M.quad : = nextquad }
```

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.ques10.com/p/9481/explain-back-patching-with-an-example-1/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 410-416

Course Teacher

Verified by HOD

LECTURE HANDOUTS

L - 35

IT

III/V

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Back patching

Introduction : (Maximum 5 sentences)

- In which lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified.
- Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

Prerequisite knowledge for Complete understanding and learning of Topic:

- Intermediate Languages
- Jumping Statements

Detailed content of the Lecture:

Flow-of-Control Statements:

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

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

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

Scheme to implement the Translation:

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

The semantic rules for the revised grammar are as follows:

(1) S -> if E then M1 S1N else M2 S2
{ backpatch (E.truelist, M1.quad);
backpatch (E.falselist, M2.quad);
S.nextlist : = merge (S1.nextlist, merge (N.nextlist, S2.nextlist)) }

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

(2) N -> ε { N.nextlist : = makelist(nextquad);
emit('goto _') }

(3) M -> ε { M.quad : = nextquad }

(4) S -> if E then M S1 { backpatch(E.truelist, M.quad);
S.nextlist : = merge(E.falselist, S1.nextlist) }

(5) S -> while M1 E do M2 S1 { backpatch(S1.nextlist, M1.quad);
backpatch(E.truelist, M2.quad);
S.nextlist : = E.falselist
emit('goto' M1.quad) }

(6) S -> begin L end { S.nextlist : = L.nextlist }

(7) S -> A { S.nextlist : = nil }

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

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

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

(9) L -> S { L.nextlist : = S.nextlist }

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.ques10.com/p/9481/explain-back-patching-with-an-example-1/

Important Books/Journals for further learning including the page nos.:

Alfred Aho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles Techniques and Tools", Pearson Education, 2014, Page no: 410-416

Course Teacher

Verified by HOD

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Faculty : **T.Manivel**

Unit : **IV** **Date of Lecture:**

Topic of Lecture: Procedure Calls

Introduction : (Maximum 5 sentences)

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

Prerequisite knowledge for Complete understanding and learning of Topic:

- Procedures
-

Detailed content of the Lecture:

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

Let us consider a grammar for a simple procedure call statement
(1) S -> call id (Elist)
(2) Elist -> Elist , E
(3) Elist -> E

Calling Sequences:

- The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence :
- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be generated.

For example, consider the following syntax-directed translation
(1) S -> call id (Elist)

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

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

- Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement.
- queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.
- Function types - The type of a function must encode the return type and the types of the formal parameters. Let void be a special type that represents no parameter or no return type.
- Symbol tables - Let s be the top symbol table when the function definition is reached. The function name is entered into s for use in the rest of the program.

  Function calls - When generating three-address instructions for a function call id(E;E; : : : ; E), it is sufficient to generate the three-address instructions for evaluating or reducing the parameters E to addresses, followed by a param instruction for each parameter.

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
http://www.brainkart.com/article/Procedure-Calls_8098/

Important Books/Journals for further learning including the page nos.:
Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles Techniques and Tools”, Pearson Education, 2014, Page no: 422-424

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-37

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **V** **Date of Lecture:**

Topic of lecture: Introduction to code optimization

Introduction: (Maximum 5 sentences):

- 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.

Prerequisite knowledge for Complete understanding and learning of Topic:
(Max. Four important topics)

- Phases of Compiler
- Intermediate languages

Detailed content of the Lecture:

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.

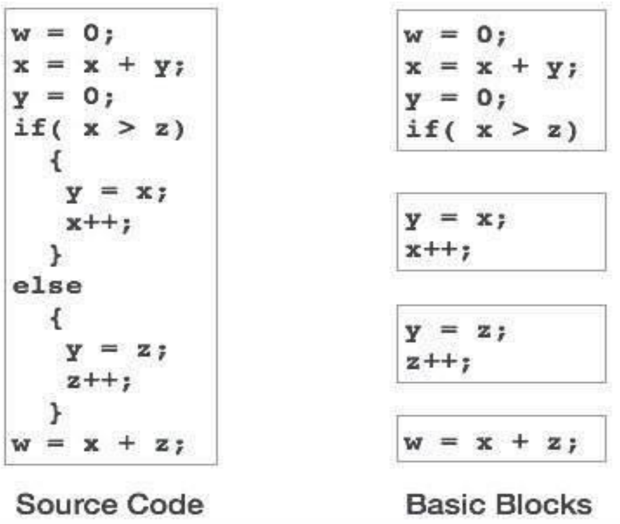
Optimization can be categorized broadly into two types : machine independent and machine dependent.

Basic Blocks

- A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.

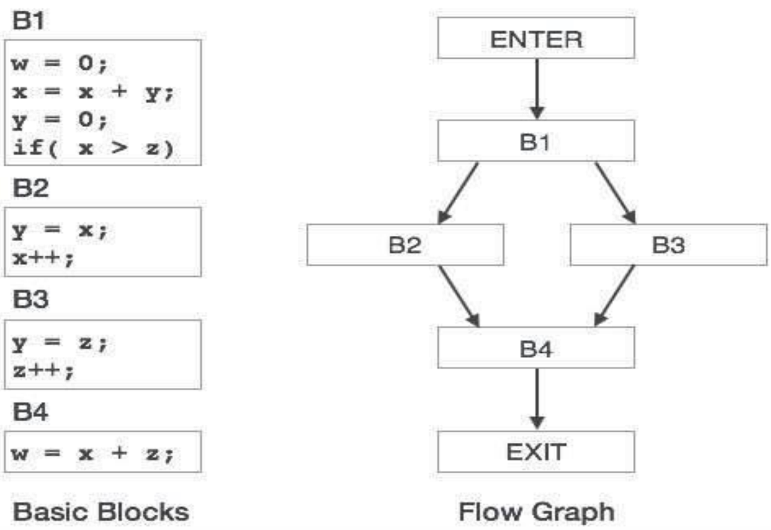
Basic block identification

- 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.



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.



Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
https://www.youtube.com/watch?v=aJKuM4U-eYg

Important Books/Journals for further learning including the page nos.:
J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 597-603

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-38

LECTURE HANDOUTS

IT | **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **V** **Date of Lecture:**

Topic of lecture: Principal Sources of Optimization

Introduction: (Maximum 5 sentences):

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Prerequisite knowledge for Complete understanding and learning of Topic:
(Max. Four important topics)

- Introduction to code optimization
- Phases of Compiler

Detailed content of the Lecture:

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

1.Function-preserving (or semantics preserving) transformations:
a. Common subexpression elimination, b. copy propagation, c. dead-code elimination, and d. constant folding

2. Loop optimization.: a. code motion, b. induction variable and c. reduction in strength

Function-preserving transformations

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

a. Local common-sub expression elimination
An occurrence of a common sub-expression within a block called Local common sub-expression

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```
B_5

(a) Before

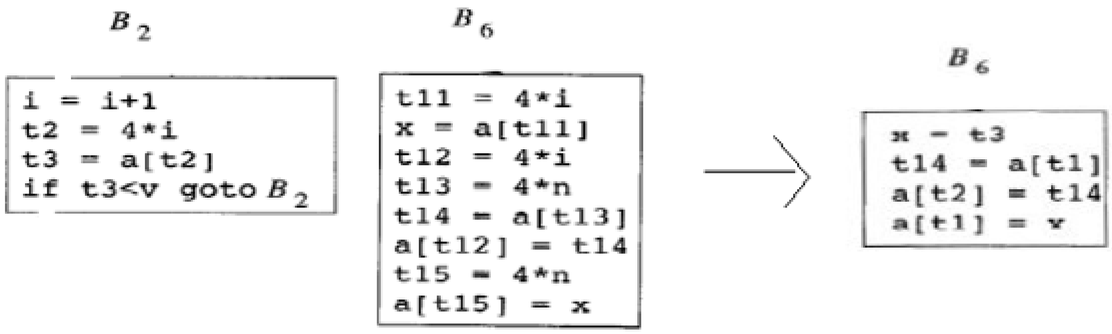
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```
B_5

(b)After

We avoid re-computing E if we can use its previously computed value
The assignments to t7 and t10 in B5 compute the common sub-expressions 4 * i and 4 * j, respectively. These steps have been eliminated and which uses t6 instead oft7 and t8 instead of t10

b. Global Common Sub-expressions

An occurrence of a common sub-expression between the blocks called global common sub-expression



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

The assignments t2 in B2 compute same expressions 4 * i in B6 in t11 & t12 . These steps have been eliminated in B6.

Copy Propagation:

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

• For example:

x=Pi;

A=x*r*r;

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

Here the variable x is eliminated

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

http://www.brainkart.com/article/Optimization-of-Basic-Blocks_8112/

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, "Introduction to Automata Theory, Languages and Computations", Pearson Education, 2003, Page no: 597-603

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

L-39

LECTURE HANDOUTS

IT

III/V

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : V

Date of Lecture:

Topic of lecture: Principal Sources of Optimization

Introduction: (Maximum 5 sentences):

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Prerequisite knowledge for Complete understanding and learning of Topic:
(Max. Four important topics)
Concepts of Code optimization phases
Concept of code generation phases
Cloud Based Complier

Detailed content of the Lecture:

2.Copy Propagation

Assignments of the form u = v called copy statements, or copies or short
For example, the assignment x = t3 in block B6 of is a copy. Instead of x replace t3

B_6

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

→

B_6

```
x = t3
a[t2] = t5
a[t4] = t3
```

Basic block B6 after copy propagation

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

3.Dead-Code Elimination

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. Dead (or useless) code statements that compute values that never get used.

B_6

```
x = t3
a[t2] = t5
a[t4] = t3
```

→

B_6

```
a[t2] = t5
a[t4] = t3
```

Basic block B6 after dead code elimination

4. Constant folding

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

For example,

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

Loop optimization

- Loops are a very important place for optimizations
- Modification that decreases the amount of code in a loop is code motion.

1. Code motion

This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluates the expression before the loop.

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

After Code motion ,result in the equivalent code

t = limit-2

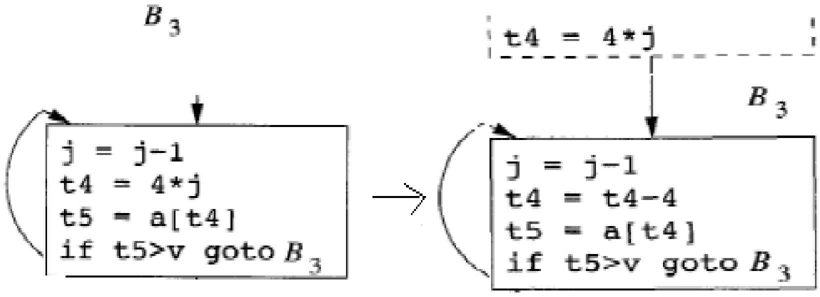
while (i <= t) /* statement does not change limit or t */

2. Induction Variables and Reduction in Strength

- The transformation of replacing an expensive operation, such as multiplication by a cheaper one, is known as **strength reduction.**

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

- The values of j and t4 remain in lock step; every time the value of j decreases by 1,the value of t4 decreases by 4, because 4 * j is assigned to t4. These variables, j and t4, thus form a good example of a pair of **induction variables**



Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

http://www.brainkart.com/article/Optimization-of-Basic-Blocks_8112/

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 604-610

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

LECTURE HANDOUTS

L-40

IT **III/V**

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **V** **Date of Lecture:**

Topic of lecture: Optimization of basic Blocks

Introduction: (Maximum 5 sentences): Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Introduction to optimization
- Principal Sources of Optimization

Detailed content of the Lecture:

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

- Ø **Structure-Preserving Transformations**
- Ø **Algebraic Transformations**

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

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

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

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

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

Dead code elimination:

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

Renaming of temporary variables:

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

Interchange of two independent adjacent statements:

- Two statements

  t1:=b+c

  t2:=x+y

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

Algebraic Transformations:

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

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

  a :=b+c

  e :=c+d+b

the following intermediate code may be generated: a :=b+c

t :=c+d e :=t+b

Example:

  x:=x+0 can be removed

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

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

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.slideshare.net/ishwarya516/optimization-of-basic-blocks

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 610-614

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-41

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : V **Date of Lecture:**

Topic of lecture: DAG representation of Basic Blocks

Introduction: (Maximum 5 sentences):

- 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.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Introduction to optimization
- Optimization of basic Blocks

Detailed content of the Lecture:

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

1. The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph is labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.

- DAGs are a type of data structure. It is used to implement transformations on basic blocks.
- DAG provides a good way to determine the common sub-expression.
- It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Algorithm for construction of DAG

- **Input:**It contains a basic block
- **Output:** It contains the following information: Each node contains a label. For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed values.

Case (i) x:= y OP z
Case (ii) x:= OP y
Case (iii) x:= y

Method:

Step 1:

If y operand is undefined then create node(y).

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

Step 2:

For case(i), create node(OP) whose right child is node(z) and left child is node(y).

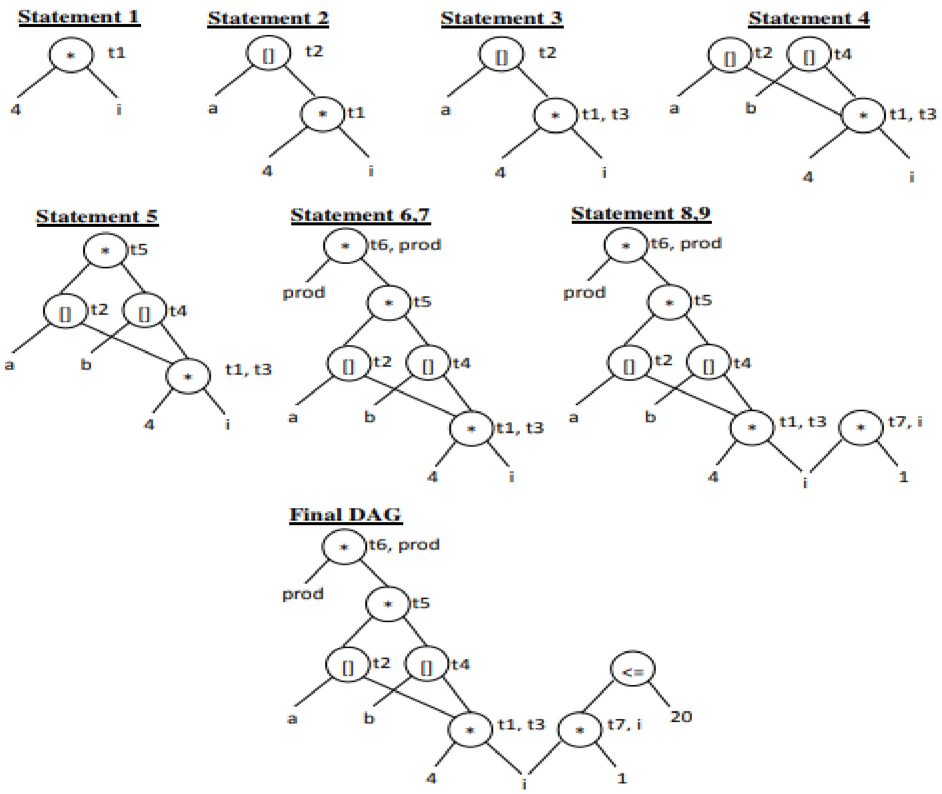
For case(ii), check whether there is node(OP) with one child node(y).
For case(iii), node n will be node(y).

Output:

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

Example: Construct DAG from the basic block.

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



Advantage of DAG

a) We can eliminate local common subexpressions,

b) We can eliminate dead code, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another;

d) We can apply algebraic laws to reorder operands of three-address instructions

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

https://www.javatpoint.com/dag-representation-for-basic-blocks

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 552-566

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-42

LECTURE HANDOUTS

IT

III/V

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : V

Date of Lecture:

Topic of lecture: Peephole Optimization

Introduction: (Maximum 5 sentences):

- A simple but effective technique for improving the target code is peephole optimization.
- A method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Introduction to optimization
- Prinicipal sources of optimization

Detailed content of the Lecture:

- Peephole optimization is a type of Code Optimization performed on a small part of the code. It is performed on the very small set of instructions in a segment of code.
- It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output.
- Peephole is the machine dependent optimization.

Characteristics of peephole optimizations:

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

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a

(2) MOV a,R0

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

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….

If ( debug ) {
Print debugging information

}
```

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

```
If debug =1 goto L1 goto L2

L1: print debugging information L2: ………………………… (a)
```

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

```
        If debug ≠1 goto L2
        Print debugging information
L2: …………………………… (b)

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

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

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

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

```
goto L1
        ….

L1: gotoL2 (d)
```

by the sequence

```
        goto L2
        ….

L1: goto L2
```

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

```
if a < b goto L1
    ….
 L1: goto L2 (e)
```

can be replaced by

If a < b goto L2

….

L1: goto L2

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

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2

goto L3

…….

L3:

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

Algebraic Simplification:

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

x := x+0 or

x := x * 1

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

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

X2 → X*X

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes.

i:=i+1 → i++

i:=i-1 → i- -

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

http://www.brainkart.com/article/Optimization-of-Basic-Blocks_8112/

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, "Introduction to Automata Theory, Languages and Computations", Pearson Education, 2003, Page no: 566-569

Course Teacher

Verified by HOD

Course Name with Code : **Principles of Compiler Design – 16ITD08**

Course Teacher : **T.Manivel**

Unit : **V** **Date of Lecture:**

Topic of lecture: Code generation, Issues in the design of code generator

Introduction: (Maximum 5 sentences):

- The final phase in our compiler model is the code generator.
- It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Introduction to Compiler
- Phases of compiler

Detailed content of the Lecture:

Output code must be correct and of high quality, meaning it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

Position of code generator

7 Issues in code generator

1. **Input to the code generator**
2. **Target programs**
3. **Memory management**
4. **Instruction selection**
5. **Register allocation**
6. **Choice of evaluation order**
7. **Approaches to code generation**

1. Input to the code generator:

- The input of the code generator is intermediate representation of the source program produced by the front end.
- Together with information in the symbol table that is used to determine the run time addresses of the data objects.
- The input may take variety of forms: Linear representations such as postfix notation, Three address representations such as quadruples. Virtual machine representations such as syntax trees and DAG.

2. Target programs

- The output of code generator is the target program.
- The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.
- Absolute machine language: program can be placed in a location in memory and immediately

executed.

- Relocatable machine language: program allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
- If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

3. **Memory management**

- Mapping of names in the source program to addresses of data object in run time memory is done by front end and the code generator.
- Labels in three address statements have to be converted to addresses of instructions.
- Three address statements must be converted in to machine code. After that,
- Fill the code in to proper machine location for all instructions by loader.

4. **Instruction selection**

- The uniformity and completeness of the instruction set are important factors.
- Ex: every three address statement of the form x: = y + z, where x, y, and z can be translated into the code sequence

```
MOV y, R0     /* load y into register R0 */
ADD z, R0     /* add z to R0 */
MOV R0, x     /* store R0 into x */
```

- The quality of the code is determined by its speed and size.
- Ex: Three address statement a := a+1

```
INC  a  // Implemented Efficiently by the single instruction
```

- Rather than

```
MOV a, R0   //use more registers ,space
ADD #1,R0     and execution time
MOV R0, a
```

- Instruction speed and machine idioms are other important factors for instruction selection

5. **Register allocation**

- Instructions involving register operands are shorter and faster than operands in memory. Therefore, efficient utilization of register is particularly important in generating good code.
- The use of registers is divided into two subproblems
  - ➢ During register allocation, select the set of variables that will reside in registers at a point in the program.
  - ➢ During a subsequent register assignment phase, pick the specific register that a variable will reside in

6. **Choice of evaluation order**

- Order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order is another difficult, NP-complete problem.

7. **Approaches to code generation**

- Important criterion for a code generator is produce correct code.
- On correctness, designing a code generator. so, it can be easily implemented, tested, and maintained is an important design goal.

Video Content / Details of website for further learning (if any):

https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/

http://www.brainkart.com/article/Code-Generation_8178/

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 525-531

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-44

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : V **Date of Lecture:**

Topic of lecture: The Target machine- A Simple code generator Algorithm

Introduction: (Maximum 5 sentences): Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)

- Code generation
- Issues in the design of code generator

Detailed content of the Lecture:

The target computer is a byte-addressable machine with 4 bytes to a word.
It has ngeneral-purpose registers, R0, R1, . . . , Rn-1.
It has two-address instructions of the form:
op source, destination where, opis an op-code, and source and destination are datafields.
It has the following op-codes :

- MOV (move sourcetodstination)
- ADD (add Source To Destination)
- SUB (subtract source from destination)

Address modes together with its costs are follows:

Mode	form	ADDRESS	COST
Absolute	M	M	1
REGISTER	R	R	0
INDEXED	C(R)	C+CONTENT(R)	1
INDIRECT REGISTER	*R	CONTENT(R)	0
INDIRECT INDEXED	*C(R)	CONTENT(C+CONTENT(R))	1
Constant(literal)	#C		0

For example : MOV R0, M stores contents of Register R0 into memory location M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.

• Address modes involving memory location or literal have cost one.

• Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.

For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since

occupies only one word of memory.
• The three-address statement a : = b + c can be implemented by many different instruction sequences :

i) MOV b, R0
ADD c, R0 cost = 6 MOV R0, a
ii) MOV b, a
ADD c, a cost = 6
iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c : MOV *R1, *R0
ADD *R2, *R0 cost = 2

• In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.
A **Simple code generator** generates target code for a sequence of three-address instructions.

A big issue is proper use of the registers, which are often in short supply, and which are used/required for several purposes. Some operands ***must be in registers.***

- 1. Holding temporaries thereby avoiding expensive memory ops.
- 2. Holding **inter-basic-block values (loop index).**
- 3. Storage management (e.g., stack pointer).

-produce code for three address statement a:=b+c if we generate a single instruction ADD Rj,Ri with cost one, leaving the result a in register Ri. Only if Ri contains b, Rj contains c and b is not live after the statement.
If Ri contains b but c is in a memory ,we can generate ADD c,Rj cost=2 or MOV c,Rj; ADD Rj,Ri Cost=3.
Register and Address Descriptors
A register descriptor keeps track of the variable names whose current value is in that register
An address descriptor keeps track of the location or locations where the current value of that variable can be found.

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
http://www.brainkart.com/article/A-Simple-Code-Generator_8104/
Important Books/Journals for further learning including the page nos.:
J.E. Hopcroft, R. Motwani and J.D Ullman, “Introduction to Automata Theory, Languages and Computations”, Pearson Education, 2003, Page no: 531-534

Course Teacher

Verified by HOD

MUTHAYAMMAL ENGINEERING COLLEGE
(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

L-45

LECTURE HANDOUTS

IT **III/V**

Course Name with Code : Principles of Compiler Design – 16ITD08

Course Teacher : T.Manivel

Unit : V **Date of Lecture:**

Topic of lecture: The Target machine- A Simple code generator Algorithm

Introduction: (Maximum 5 sentences): A code generator generates target code for a sequence of three-address statements and effectively uses registers to store operands of the statements.

Prerequisite knowledge for Complete understanding and learning of Topic: (Max. Four important topics)
Concept of code generation

Detailed content of the Lecture:

<u>A Code generation Algorithm</u>

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

1. Invoke a function **getreg** to determine the location L where the result of the computation y op z should be stored, L will usually be a register.
2. Consult the address descriptor for y to determine y′
3. Generate the instruction op z′, L where z′ is a current location of Z.
4. Update the address descriptor of x, to indicate that x is in location L.If L is a register , update its descriptor to indicate that it contains the value of x and remove x from all other register descriptors.
5. If the current values of y and / or z have no next uses and are in registers, after the register descriptor to indicate that, those registers no longer will contain y and / or z respectively.

<u>Function GetReg</u>

a. An essential part of the algorithm is a function getReg(I), which selects registers for each memory location associated with the three-address instruction
b. Function getReg has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block.
c. In a three-address instruction such as x = y + z, A possible improvement to the algorithm is to generate code for both x = y + z and x = z + y whenever + is a commutative operator, and pick the better code sequence.

Statements	Code generated	Register descriptor	Address descriptor	Cost
		Registers empty		
t:= a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0	2 2
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1contains u	t in R0 u in R1	2 2
v:= t + u	ADD R1,R0	R0 contains v R1contains u	u in R1 v in R0	1 1
d : = v + u	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 din R0 and memory	2
			Total cost	12

Code sequence for indexed assignments

Statement	i IN REGISTER Ri		i IN MEMORY Mi		i IN STACK	
	Code	Cost	Code	Cost	Code	Cost
a := b[i]	MOV b(Ri) , R	2	MOV Mi, R MOV b(R), R	4	MOV Si(A) , R MOV b (R),R	4
a[i] := b	MOV b,a(Ri)	3	MOV Mi,R MOV b, a(R)	5	MOV Si(A) , R MOV b ,a(R)	5

Video Content / Details of website for further learning (if any):
https://learnengineering.in/pdf-principles-of-compiler-design-by-alfred-v-aho-j-d-ullman-free-download/
ewww.brainkart.com/article/A-Simple-Code-Gnerator_8104/

Important Books/Journals for further learning including the page nos.:

J.E. Hopcroft, R. Motwani and J.D Ullman, "Introduction to Automata Theory, Languages and Computations", Pearson Education, 2003, Page no: 547-553

Course Teacher

Verified by HOD