

***Jayawant Shikshan Prasarak Mandal***

**Computer Laboratory-I**

**Laboratory Manual**

**Computer Engineering**

**Computer Laboratory-I**

**Laboratory Manual**

**Author : - Prof . R. A. Deshmukh**

**Creation Date: - June 2015**

**Last Updated : - June 2015**

**Version : - 1**

**© JSPM Group of Institutes, Pune. All Rights Reserved. All the information in this Course Manual is confidential. Participants shall refrain from copying, distributing, misusing or disclosing the content to any third parties any circumstances whatsoever.**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| Sr . No. | Topic | Page. No. |
| 1. | **Vision, Mission** |  |
| 2. | **How to Use This Manual** |  |
| 3. | **PEOs and POs** |  |
| 4. | **Course Objective** |  |
| 5. | **Laboratory Objective** |  |
| 6. | **Experiment Learning Outcome (ELO)** |  |
| 7. | **Lab Plan** |  |
| **Group A** | |  |
| **Sessions** | |  |
|  |  |  |
|  |  |  |
|  | Lexical analyzer for sample language using LEX. | 17 |
|  | Parser for sample language using YACC. | 24 |
|  | Int code generation for sample language using LEX and YACC. | 35 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Group B** | |  |
| **Sessions** | |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | Code optimization using DAG. | 49 |
|  | Code generation using DAG / labeled tree. | 67 |
|  | Generating abstract syntax tree using LEX and YACC. | 78 |
|  | Implementing recursive descent parser for sample language. | 86 |
|  | Write a program to implement SLR Parsing algorithm using Python for the ordered input Set in XML | 95 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Group C | | |
|  | Code generation using \iburg" tool. | 101 |
|  |  |  |
|  |  |  |
| 8. | **References** | 110 |

**Vision**

To produce globally competitive professionals and socially sensitive computer engineers enriched with knowledge and power of innovation.

**Mission**

To impart **quality education** to the students at all levels that meets the changing needs of the industry.

To provide state-of-art facilities and resources to **solve real-world complex problems** and for **discovery of new knowledge through innovative research** that encourages **entrepreneurship** and economic development to benefit our **global society**.

To promote active **learning, critical thinking, engineering judgment and, professional capabilities of students** coupled with business and also educate and follow **ethical, social and environmentally responsible engineering practice**.

**How to Use This Manual**

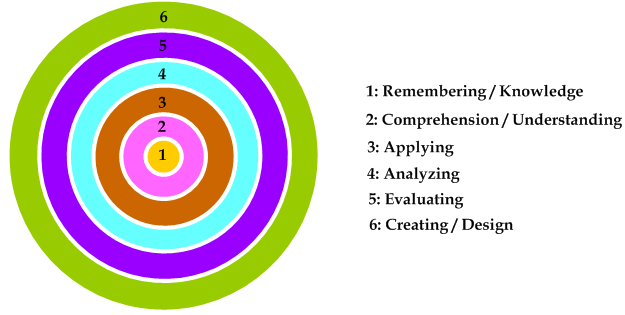
This Manual assumes that the facilitators are aware of Collaborative Learning Methodologies.

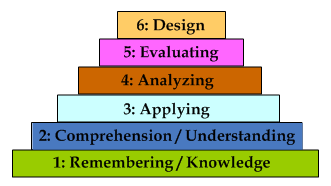
This Manual will only provide them tool they may need to facilitate the session on Computer Organization module in collaborative learning environment.

The Facilitator is expected to refer this Manual before the session.

|  |  |  |  |
| --- | --- | --- | --- |
| **K**  Applying Knowledge  (PO:a) | **A**  Problem Analysis  (PO:b) | **D**  Design & Development  (PO:c) | **I**  Investigation of problems  (PO:d) |
| **M**  Modern Tool Usage  (PO:e) | **E**  Engineer & Society  (PO:f) | **E**  Environment Sustainability  (PO:h) | **T**  Ethics  (PO:i) |
| **T**  Individual & Team work  (PO:g) | **O**  Communication  (PO:k) | **M**  Project Management & Finance (PO:j) | **I**  Life Long Learning (PO:l) |

**Disk Approach- Digital Blooms Taxonomy**





This Manual uses icons as visual cues to the interactivities during the session.

|  |  |
| --- | --- |
| **Icons** | **Graduate Attributes** |
|  | Applying Knowledge |
|  | Problem Analysis |
|  | Design and Development |
|  | Investigation of Problem |
|  | Modern Tool Usage |
|  | Engineer and Society |
|  | Environment Sustainability |
|  | Ethics |
|  | Individual and Teamwork |
|  | Communication |
|  | Project Management and Finance |
|  | Lifelong Learning |
|  | **Blooms Taxonomy** |
|  | Remembering |
|  | Understanding |
|  | Applying |
|  | Analyzing |
|  | Evaluating |
|  | Creating |

|  |  |
| --- | --- |
|  | This icon is used to indicate instructions for faculties. |
|  | This icon is used to indicate a statement to be made by faculty. |
|  | This icon is used to indicate a list of additional resources. |
|  | This icon indicates an activity to be conducted. |
|  | This icon indicates questions to be asked by faculty. |

**Program Educational Objectives: -**

|  |  |
| --- | --- |
| **PEO I** | To prepare graduate for productive engineering career in industry and also to pursue **higher studies** and **research**. |
| **PEO II** | The graduate of program will have **solid foundation** in Computer Engineering. |
| **PEO III** | The graduate of program will have exposure to cutting edge technology, adequate training and opportunities to work as **team** with effective **communication skills** and **leadership qualities**. |
| **PEO IV** | The graduate of program will have skills to **identify, analyze, design, implement** and **manage** the software projects **using modern tools** for benefit our global **society** and promote them to disseminate it. |
| **PEO V** | The graduate of program will able to keep pace with continuous **upgrading technology** as well as aware of **social, environmental issues** and professional **ethics** and codes of professional practices. |

**Program Outcomes: -**

|  |  |
| --- | --- |
| a | An ability to apply knowledge of computing, mathematics, science and engineering fundamentals appropriate to Computer Engineering |
| a1 | Knowledge of basic science |
| a2 | Fundamental knowledge of Computer Engineering |
| b | An ability to analyze a problem, identify and define the computing requirement appropriate to the solution in Computer Engineering domain |
| b1 | identify the problem and requirements |
| b2 | define and analyze the problem requirements |
| c | An ability to design, implement and evaluate a system, process, component and program to meet desired needs within realistic constraints |
| c1 | design the computer system |
| c2 | implement the computer system |
| c3 | evaluate the computer system |
| d | An ability to investigate, formulates, analyze and provide appropriate solution to the complex engineering problems |
| d1 | investigate and formulate the complex problem |
| d2 | analyze and solve |
| e | An ability to use modern engineering tools, technologies, technique and skills necessary for engineering practices as a Computer Engineering |
| e1 | modern engineering tools |
| e2 | modern engineering technologies, technique and skills |
| f | Apply reasoning informed by the contextual knowledge of Computer field to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice |
| f1 | Apply knowledge of computer to societal and cultural issues |
| f2 | professional engineering practice |
| g | An ability to function effectively as an individual or as a team member to accomplish the goal |
| g1 | function effectively as an individual |
| g2 | work in a team |
| h | An ability to understand the environmental issues and provide the sustainable system |
| h1 | environmental issues |
| h2 | contemporary solutions |
| i | An ability to understand professional, financial, ethical, legal, security, and social issues and responsibility |
| i1 | professional, financial, and security |
| i2 | ethical, legal and social issues |
| j | An ability to apply knowledge of project management and finance |
| j1 | project management |
| j2 | finance |
| k | An ability to communicate effectively with engineering community at different levels |
| k1 | verbal communication |
| k2 | nonverbal communication |
| l | An ability to keep abreast with contemporary technologies through lifelong learning |
| l1 | contemporary technologies |
| l2 | lifelong learning |

**PO to PEO Mapping with the help of Articulation Matrix: -**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PO/GA | | PEO: 1 | PEO: 2 | PEO: 3 | PEO: 4 | PEO: 5 |
|  | a |  |  |  |  |  |
|  | b |  |  |  |  |  |
|  | c |  |  |  |  |  |
|  | d |  |  |  |  |  |
|  | e |  |  |  |  |  |
|  | f |  |  |  |  |  |
|  | h |  |  |  |  |  |
|  | i |  |  |  |  |  |
|  | g |  |  |  |  |  |
|  | k |  |  |  |  |  |
|  | j |  |  |  |  |  |
|  | l |  |  |  |  |  |

**Course Outcomes:**

CO1: To understand working of different phases of compiler in detail & Implement those phases.  

CO2:To demonstrate the ability to use different tools like LEX, YACC and iburg. 

**CO to PO Mapping with the help of Articulation Matrix: -**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course Name | CO | PO | | | | | | | | | | | | | | | | | | | | | | | | |
|  | a | | b | | c | | | d | | e | | f | | g | | h | | i | | j | | k | | l | |
|  | a1 | a2 | b1 | b2 | c1 | c2 | c3 | d1 | d2 | e1 | e2 | f1 | f2 | g1 | g2 | h1 | h2 | i1 | i2 | j1 | j2 | k1 | k2 | l1 | l2 |
|  | CO1 |  | 2 |  |  |  | 1 |  |  |  | 1 | 1 |  |  | 2 | 1 |  |  |  |  |  |  |  |  | 2 | 2 |
| CO2 |  | 1 |  |  | 1 | 1 |  |  |  | 2 | 2 |  |  | 2 |  |  |  |  |  |  |  |  |  | 2 | 1 |
| CO3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CO4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CO5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CO6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CO7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Laboratory Objectives:**

1. Understand and develop programs in LEX & YACC for illustrating Lexical Analyzer, parser & intermediate code generator  
2. Understand & apply different code optimization techniques . 
3. To make use of different tools like LEX, YACC and iburg. 
4. Implement code generation algorithms to generate target code for given input. 
5. **LO to CO Mapping with the help of Articulation Matrix: -**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bloom Level | CO: 1 | CO: 2 | CO: 3 | CO: 4 | CO: 5 | CO: 6 | CO: 7 |
| LO: 1 |  |  |  |  |  |  |  |  |
| LO: 2 |  |  |  |  |  |  |  |  |
| LO: 3 |  |  |  |  |  |  |  |  |
| LO: 4 |  |  |  |  |  |  |  |  |
| LO: 5 |  |  |  |  |  |  |  |  |
| LO: 6 |  |  |  |  |  |  |  |  |
| LO: 7 |  |  |  |  |  |  |  |  |

**Experiment Learning Outcome: -**

1. Explain different concepts in lexical analyzer & write program for lexical analyzer
2. Write a program to implement Syntax analyzer phase of complier. 
3. Implement intermediate code generation phase of compiler using lex and yacc. 
4. Demonstrate iburg tool for code generation. 
5. Explain and apply different code optimization techniques. 
6. Explain and apply code generation algorithm on DAG.  
7. Explain the concept of Abstract Syntax Tree(AST) 
8. Write a program to implement SLR parsing algorithm. 
9. Explain and implement recursive descent parser. 

**ELO to LO Mapping with the help of Articulation Matrix: -**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bloom Level | LO: 1 | LO: 2 | LO: 3 | LO: 4 | LO: 5 | LO: 6 | LO: 7 |
| ELO: 1 |  |  |  |  |  |  |  |  |
| ELO: 2 |  |  |  |  |  |  |  |  |
| ELO: 3 |  |  |  |  |  |  |  |  |
| ELO: 4 |  |  |  |  |  |  |  |  |
| ELO: 5 |  |  |  |  |  |  |  |  |
| ELO: 6 |  |  |  |  |  |  |  |  |
| ELO: 7 |  |  |  |  |  |  |  |  |
| ELO: 8 |  |  |  |  |  |  |  |  |
| ELO: 9 |  |  |  |  |  |  |  |  |
| ELO: 10 |  |  |  |  |  |  |  |  |
| ELO: 11 |  |  |  |  |  |  |  |  |
| ELO: 12 |  |  |  |  |  |  |  |  |
| ELO: 13 |  |  |  |  |  |  |  |  |
| ELO: 14 |  |  |  |  |  |  |  |  |
| ELO: 15 |  |  |  |  |  |  |  |  |
| ELO: 16 |  |  |  |  |  |  |  |  |

**GROUP A**

**EXPERIMENT NO.3**

**Lexical analyzer for sample language using LEX**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 15 | Concept of Lex, Lexical analyzer | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Lexical analyzer for sample language using LEX

**OBJECTIVES:**

Understand the importance and usage of LEX automated tool.

**PROBLEM STATEMENT:**

Implement a lexical analyzer for a sample language using LEX Implementation

**SOFTWARE REQUIRED**: Linux Operating Systems, GCC

**INPUT:** Input data as Sample language.

**OUTPUT**: It will generate tokens for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the class Weather Report such that

S={s, e, i, o, f, DD, NDD, success, failure}

s=Start of program

e = the end of program

i=Sample language statement.

o=Result of statement

Success-token is generated.

Failure-token is not generated or forced exit due to system error.

Computational Model

e1 e2

Where,

S={Start state}

A={genrate token()}

R={Final Result}

**THEORY:**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose langua ge which recognizes regular expressions. The regular expressions are specified by the user in the so urce specifications given to Lex.

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed

1) LEX Specifications :- Structure of the LEX Program is as follows

-----------------------------

Declaration Part

-----------------------------

%%

-----------------------------

Translation Rule

-----------------------------

%%

-----------------------------

Auxiliary Procedures

-----------------------------

2) Declaration part :- Contains the declaration for variables required for LEX program and C program.

Translation rules:- Contains the rules like

Reg. Expression { action1 }

Reg. Expression { action2 }

Reg. Expression { action3 }

-------------------------------------------

Reg. Expression { action-n }

3) Auxiliary Procedures :-

Contains all the procedures used in your C – Code.

* Built-in Functions i.e. yylex() , yyerror() , yywrap() etc.

1) yylex() :- This function is used for calling lexical analyzer for given translation rules.

2) yyerror() :- This function is used for displaying any error message.

3) yywrap() :- This function is used for taking i/p from more than one file.

* Built-in Variables i.e. yylval, yytext, yyin, yyout etc.

1) yylval :- This is a global variable used to store the value of any token.

2) yytext :-This is global variable which stores current token.

3) yyin :- This is input file pointer used to change value of input file pointer. Default file pointer is pointing to stdin i.e. keyboard.

4) yyout :- This is output file pointer used to change value of output file pointer. Default output file pointer is pointing to stdout i.e. Monitor.

* How to execute LEX program :- For executing LEX program follow the following steps
* Compile \*.l file with lex command

# lex \*.l It will generate lex.yy.c file for your lexical analyzer.

* Compile lex.yy.c file with cc command

# cc lex.yy.c It will generate object file with name a.out.

* Execute the \*.out file to see the output

# ./a.out

**CONCLUSION**

Thus we have studied Lexical Analyzer for sample language.

**OUTCOME**

**Upon completion Students will be able to:**

1.Explain Complier concept 

2. Explain lexical analysis 

**QUESTIONS**

1. What is compiler? 

Ans: Compiler is a program which takes one language as input and translates into an equivalent another Language

1. What is token? 

Ans: Token describes the class or category of input string. Eg identifier, constants are called tokens.

1. Define lexemes.

Ans: It is a sequence of character in source program that are matched with the pattern of token.

1. What is lex? 

Ans: Lex is a tool for generating lexical analyzer.

1. What is lex.yy.c file? 

Ans: The lex specification file contains the regular expression for tokens and lex.yy.c is a program that consist the tabular expression of the transistion diagram construced for regular expression of specification file.

1. What is the meaning of yytext? 

Ans: When lexer matches or recognizes the token form input then the lexeme is stored in a null terminated string called yytext.

1. What is yylex() fuction? 

Ans: As soon as call to yylex() is encounter scanner starts scanning the source program.

1. whether lexical analyzer detects any error?

Ans: Yes. Following errors can be following errors

1.Spelling mistakes.Hence get incorrect tokens.

2.Exceeding length of identifier or numeric constants.

3.Appearance of illegal character.

9. Explain the meaning of [],””,?, | symbols. 

Ans: 1.[] –A character class which matches any character within the bracket.

2.”” – String written in quotes matches literally.

3. ? –Matches zero or more occcurances of preceding regular expreission.

4.| - To represent the OR.

**EXPERIMENT NO.4**

**Parser for sample language using YACC.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 15 | Concept of YACC | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**Session Plan**

**TITLE:** Parser for sample language using YACC.

**OBJECTIVES:**

1. To understand Second phase of compiler: Syntax Analysis.
2. To learn and use compiler writing tools.
3. Understand the importance and usage of YACC automated tool.

**PROBLEM STATEMENT:**

Write an ambiguous CFG to recognize an infix expression and implement a parser that recognizes the infix expression using YACC.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as Sample language.

**OUTPUT**: It will generate parser for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the class Weather Report such that

S={s, e, i, o, f, DD, NDD, success, failure}

s=Start of program

e = the end of program

i=Arithmetic expression.

o=Result of arithmetic expression

Success-parser is generated.

Failure-parser is not generated or forced exit due to system error.

Computational Model

e1 e2 e3

R

B

A

S

Where,

S={Start state}

A={Genrate token()}

B={Parse\_token()}

R={Final Result}

**THEORY:**

1. YACC Specifications: - Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for “Yet Another Compiler Complier”. File in which parser generated is with .y extension. e.g. parser.y, which is containing YACC specification of the translator. After complete specification UNIX command. YACC transforms parser.y into a C program called y.tab.c using LR parser. The program y.tab.c is automatically generated. We can use command with –d option as yacc –d parser.y By using –d option two files will get generated namely y.tab.c and y.tab.h. The header file y.tab.h will store all the token information and so you need not have to create y.tab.h explicitly. The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. By compiling y.tab.c with the ly library that contains the LR parsing program using the command. cc ytabc – ly . we obtain the desired object program a.out that perform the translation specified by the original program. If procedure is needed, they can be compiled or loaded with y.tab.c, just as with any C program. LEX recognizes regular expressions, whereas YACC recognizes entire grammar. LEX divides the input stream into tokens, while YACC uses these tokens and groups them together logically.LEX and YACC work together to analyze the program syntactically. The YACC can report conflicts or ambiguities (if at all) in the form of error messages.



Structure of the YACC Program is as follows

-----------------------------

Declaration Section

-----------------------------

%%

-----------------------------

Translation Rule Section

-----------------------------

%%

-----------------------------

Auxiliary Procedures Section

-----------------------------

# Declaration Section :-

The definition section can include a literal block, C code copied verbatim to the beginning of the generated C file, usually containing declaration and #include lines. There may be %union, %start, %token, %type, %left, %right, and %nonassoc declarations. (See "%union Declaration," "Start Declaration," "Tokens," "%type Declarations," and "Precedence and Operator Declarations.") It can also contain comments in the usual C format, surrounded by "/\*" and "\*/". All of these are optional, so in a very simple parser the definition section may be completely empty.

Translation rule Section :-

Contains the rules / grammars

Production { action1 }

Production { action2 }

Production { action3 }

---------------------------------------

Production { action-n }

# Auxiliary Procedure Section :-

Contains all the procedures used in your C – Code.

1. Built-in Functions i.e. yyparse() , yyerror() , yywrap() etc.

1) yyparse() :-

This is a standard parse routine used for calling syntax analyzer for given translation rules.

2) yyerror() :-

This is a standard error routine used for displaying any error message.

3) yywrap() :-

This function is used for taking i/p from more than one file.

1. Built-in Types i.e. %token , %start , %prec , %nonassoc etc.

1) %token

Used to declare the tokens used in the grammar. The tokens that are declared in the declaration section will be identified by the parser.

Eg. :- %token NAME NUMBER

2) %start :-

Used to declare the start symbol of the grammar.

Eg.:- %start STMT

3) %left

Used to assign the associatively to operators.

Eg.:- %left ‘+’ ‘-‘ - Assign left associatively to + & – with lowest precedence.

%left ‘\*’ ‘/‘ - Assign left associatively to \* & / with highest precedence.

4) %right :-

Used to assign the associatively to operators.

Eg.:- 1) %right ‘+’ ‘-‘

- Assign right associatively to + & – with lowest precedence

2) %right ‘\*’ ‘/‘

* + - Assign right left associatively to \* & / with highest precedence.

5) %nonassoc :-

Used to un associate.

Eg.:- %nonassoc UMINUS

6) %prec :-

Used to tell parser use the precedence of given code.

Eg. :- %prec UMINUS

7) %type :-

Used to define the type of a token or a non-terminal of the production written in the rules section of the .Y file.

Eg.:- %type <name of any variable> exp

Let us see both LEX and YACC specification for writing a program for calculator.

LEX Specification : (lex.l file)

1. Declaration Section :

%{

#include "y.tab.h"

#include<math.h>

extern int yylval;

%}

Here, we include the header file that is generated while executing the .y file. We also include math.h as we will be using a function atoi (that type casts string to integer).

Lastly, when a lexical analyzer passes a token to the parser, it can also pass a value for the token. In order to pass the value that our parser can use (for the passed token), the lexical analyser has to store it in the variable yylval.

Before storing the value in yylval we have to specify its data type. In our program we want to perform mathematical operations on the input, hence we declare the variable yylval as integer.

1. Rules Section :

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[ \t] ; /\* ignore white space \*/

\n return 0; /\* logical EOF \*/

. return yytext[0];

In rules section, we match the pattern for numbers and pass the token NUMBER to the parser. As we know the matched string is stored in yytext which is a string, hence we type cast the string value to integer. We ignore spaces, and for all other input characters we just pass them to the parser.

YACC Specification : (yacc.y file)

1. Declaration Section:

%token NUMBER

%left '+' '-'

%left '/' '\*'

%right '^'

%nonassoc UMINUS

In declaration section we declare all the variables that we will be using through out the program, also we include all the necessary files.

Apart from that, we also declare tokens that are recognized by the parser. As we are writing a parser specification for calculator, we have only one token that is NUMBER.

To deal with ambiguous grammar, we have to specify the associativity and precedence of the operators. As seen above +,-,\* and / are left associative whereas the Unary minus and power symbol are non-associative.

The precedence of the operators increase as we come down the declaration. Hence the lowest precedence is of + and – and the highest is of Unary minus.

1. Rules Section:

The rules section consists of all the production that perform the operations. One example is given as follows :

expression: expression '+' expression { $$ = $1 + $3; }

| NUMBER { $$ = $1; }

;

When a number token is returned by the lexical analyser, it converts it into expression and its value is assigned to expression (non-terminal).

When addition happens the value of both the expression are added and are assigned to the expression that has resulted from the reduction.

1. Auxillary Function Section:

In main we just call function yyparse.

We also have to define the function yyerror(). This function is called when there is a syntactic error.

**CONCLUSION:**

Thus we have studied Parser for sample language using YACC.

**OUTCOME**

**Upon completion Students will be able to:**

1. Explain different parsing technique
2. What is YACC 

**QUESTIONS**

1. What is a parser? 

**Ans:** Parse generator can be used to facilitate the construction of the front end of a compiler

1. Explain about yacc parser generator? 

**Ans:** Yacc is a parser generator that is a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language.

1. What is the difference between yylex() and scanf().

**Ans.** yylex() is used to accept input and call parser, but scanf() for only accepting data.

1. is Yacc a compiler!

**Ans:** No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.

1. Explain construction of yacc prg. 

**Ans:** a prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.

1. What are different sections of yacc 

**Ans:**Declarations, translation rule and support & routine sections.

1. Explain the grammar of yacc 

**Ans:** <left side> : <alternate1> {semantic action1}

| alternate2> {semantic action2}

1. what is the difference between Lex and Yacc 

**Ans:** lex prg to lex.yy.c to scanner to parser to exe

yac prg to y.tab.c to parser to exe

1. What is Yacc? What is the input to it? What is its output? 
2. What are regular expressions? 
3. What is a grammar? 
4. Explain this regular expression [^ \t\n]+ 
5. What are token definitions? 
6. What does y.tab.h do? 

**EXPERIMENT NO.5**

**Int. code generation for sample language using LEX and YACC.**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **C** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 15 | Concept of Intermediate Language | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Int. code generation for sample language using LEX and YACC.

**OBJECTIVES:**

1. To understand fourth phase of compiler: Intermediate code generation.
2. To learn and use compiler writing tools.
3. To learn how to write three address code for given statement.

**PROBLEM STATEMENT:**

Write an attributed translation grammar to recognize declarations of simple variables, "for", assignment, if, if-else statements as per syntax of C and generate equivalent three address code for the given input made up of constructs mentioned above using LEX and YACC .

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as Sample language.

**OUTPUT:** It will generate Intermediate language for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the class Weather Report such that

S={s, e, i, o, f, success, failure}

s=initial state of grammar

e = the end state of grammar.

i=Sample language Statement.

o=Intermediate code for language statement

Success-Intermediate code is is generated.

Failure-Intermediate code is not generated or forced exit due to system error.

Mathematical model for above system.



**THEORY:**

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates retargeting: enables attaching a back end for the new machine to an existing front end.





Close to source language close to machine language

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

Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

• flow–of-control checks

– Ex: Break statement within a loop construct

• Uniqueness checks

– Labels in case statements

• Name-related checks

Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

• A clear distinction between the machine-independent and machine-dependent parts of the compiler

• Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end

• We could apply machine independent code optimisation techniques

Intermediate representations span the gap between the source and target languages.

• High Level Representations

– closer to the source language

– easy to generate from an input program

– code optimizations may not be straightforward

• Low Level Representations

– closer to the target machine

– Suitable for register allocation and instruction selection

Intermediate Languages

Three ways of intermediate code representation:

Syntax tree

Postfix notation

Three address code

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

Graphical Representations

Syntax tree

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

a : = b \* - c + b \* - c

are as follows:

assign

a +

\* \*

b uminus b uminus

c c

Fig a. Syntax tree

assign

a +

\*

b uminus

c

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

Advantages of three-address code

The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.

The use of names for the intermediate values computed by a program allows three address code to be easily rearranged – unlike postfix notation. Three-address code is a liberalized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

Types of Three-Address Statements

The common three-address statements are:

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

Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number. Copy statements of the form x : = y where the value of y is assigned to x. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

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

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

param x1

param x2

. . . . . . .

param xn

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

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

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

Implementation of Three-Address Statements: A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are: Quadruples, Triples, Indirect triples.

A. Quadruples

A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result. The op field contains an internal code for the operator. The 3 address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result. The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created. Fig a) shows quadruples for the assignment a : b \* c + b \* c

B. Triples:

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | op | arg1 | arg2 | result |
| (0) | uminus | c |  | t1 |
| (1) | \* | b | t1 | t2 |
| (2) | uminus | c |  | t3 |
| (3) | \* | b | t2 | t4 |
| (4) | + | t2 | t4 | t5 |
| (5) | := | t5 |  | a |

Fig. a. Quadraples

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| (0) | uminus | c |  |
| (1) | \* | b | (0) |
| (2) | uminus | c |  |
| (3) | \* | b | (2) |
| (4) | + | (1) | (3) |
| (5) | := | a | (4) |

Fig.b. Triples

Quadruples & Triple representation of three-address statement

C. Indirect triples:

Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves. Let us use an array statement to list pointers to triples in the desired order. Fig c) shows the indirect triple representation.

|  |  |
| --- | --- |
| Statement | op |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

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

Fig c): Indirect triples representation of three address statements

Steps to execute the program

$ lex filename.l (eg: comp.l)

$ yacc -d filename.y (eg: comp.y)

$cc lex.yy.c y.tab.c –ll –ly –lm

$./a .out

ALGORITHM:

Write a LEX and YACC program to generate Intermediate Code for arithmetic expression

LEX program

1. Declaration of header files specially y.tab.h which contains declaration for Letter, Digit, expr.

2. End declaration section by %%

3. Match regular expression.

4. If match found then convert it into char and store it in yylval.p where p is pointer declared in YACC

5. Return token

6. If input contains new line character (\n) then return 0

8. If input contains „.‟ then return yytext[0]

9. End rule-action section by %%

10. Declare main function

a. open file given at command line

b.if any error occurs then print error and exit

c. assign file pointer fp to yyin

d.call function yylex until file ends

11. End

YACC program

1. Declaration of header files

2. Declare structure for threeaddresscode representation having fields of argument1, argument2, operator, result.

3. Declare pointer of char type in union.

4. Declare token expr of type pointer p.

5. Give precedence to „\*‟,‟/‟.

6. Give precedence to „+‟,‟-‟.

7. End of declaration section by %%.

8. If final expression evaluates then add it to the table of three address code.

9. If input type is expression of the form.

a. exp‟+‟exp then add to table the argument1, argument2, operator.

b.exp‟-‟exp then add to table the argument1, argument2, operator.

c. exp‟\*‟exp then add to table the argument1, argument2, operator.

d.exp‟/‟exp then add to table the argument1, argument2, operator.

e. „(„exp‟)‟ then assign $2 to $$.

f. Digit OR Letter then assigns $1 to $$.

10. End the section by %%.

11. Declare file \*yyin externally.

12. Declare main function and call yyparse function untill yyin ends

13. Declare yyerror for if any error occurs.

14. Declare char pointer s to print error.

15. Print error message.

16. End of the program.

In short:

Addtotable function

It will add the argument1, argument2, operator and temporary variable to the structure array of threeaddresscode.

Threeaddresscode function

It will print the values from the structure in the form first temporary variable, argument1, operator, argument2

Quadruple Function

It will print the values from the structure in the form first operator, argument1, argument2, result field

Triple Function

It will print the values from the structure in the form first argument1, argument2, and operator. The temporary variables in this form are integer / index instead of variables.

**CONCLUSION:**

Hence, we have successfully studied concept of Intermediate code generation of sample language

**OUTCOME**

**Upon completion Students will be able to:**

1. Explain Intermediate code generation 
2. Explain DAG

**QUESTIONS:**

1. What is intermediate code generation? 
2. Explain the different forms of ICG? 
3. What are the difference between syntax tree and DAG? 
4. What are advantages of 3-address code? 
5. Which representation of 3-address code is better than other and why? Justify. 
6. What is role of Intermediate code in compiler? 
7. Explain quadruple, triples and indirect tuples representation for a : b \* c + b \* c 
8. What are the advantages of three-address code? 
9. Compare different types of Three-Address Statements? 

**GROUP B**

**EXPERIMENT NO. 4**

**Code Optimization Using DAG**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 20 | Concept of Code Optimization, DAG | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Code optimization using DAG

**OBJECTIVES:**

1. To express concept of code optimization and directed acyclic graph

2. To implement code optimization using directed acyclic graphs.

**PROBLEM STATEMENT:**

Implement code optimization using directed acyclic graph.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC.

**INPUT:** Input data as intermediate code.

**OUTPUT:** It will create a optimized code or optimized intermediate code.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of optimized code

S={s, e, i, o, f, DD, NDD, success, failure}

s=initial state that is intermediate or un-optimized code

e = the end state or optimized code.

i= input of the system.

o=output of the system.

DD-deterministic data it helps identifying the load store functions or assignment functions.

NDD- Non deterministic data of the system S to be solved.

**THEORY:**

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects.In the code optimization phase the intermediate code is improved to run the output faster and occupies the lesser space.

Output of this phase is another intermediate code to improve the efficiency. The basic requirement of optimization methods should comply with is that an optimized program must have the same output and side effects as its non-optimized version. This requirement, however, may be ignored in case the benefit from optimization is estimated to be more important than probable consequences of a change in the program behavior.

Optimization can be performed by automatic optimizers or programmers. An optimizer is either a specialized software tool or a built-in unit of a compiler (the so-called optimizing compiler).Optimizations are classified into high-level and low-level optimizations. High-level optimizations are usually performed by the programmer who handles abstract entities (functions, procedures, classes, etc.) and keeps in mind the general framework of the task to optimize the design of a system. Optimizations performed at the level of elementary structural blocks of source code - loops, branches, etc. - are usually referred to as high-level optimizations too. Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed. Assembler programmers, though, believe that no machine, however perfect, can do this better than a skilled programmer.

**Control-Flow Analysis:-**

In control-flow analysis, the compiler figures out even more information about ow the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

Control-flow analysis begins by constructing a control-flow graph , which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into basic blocks. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it.

Optimizations performed exclusively within a basic block are called "local optimizations".

Constant Folding :- Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant.

**Constant Propagation:-** If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.

**Code Motion :-** Code motion (also called code hoisting ) unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation.

**Peephole Optimizations :-** Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine-dependent code improvements.

### Redundant instruction elimination:-At source code level, the following can be done by the user

|  |  |  |  |
| --- | --- | --- | --- |
| int add\_ten(int x)  {  int y, z;  y = 10;  z = x + y;  return z;  } | int add\_ten(int x)  {  int y;  y = 10;  y = x + y;  return y;  } | int add\_ten(int x)  {  int y = 10;  return x + y;  } | int add\_ten(int x)  {  return x + 10;  } |

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

* MOV x, R0
* MOV R0, R1

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

MOV x, R1

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

Example:

void add\_ten(int x)

{

return x + 10;

printf(“value of x is %d”, x);

}

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

### Flow of control optimization

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

MOV R1, R2

GOTO L1

...

L1 : GOTO L2

L2 : INC R1

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

MOV R1, R2

GOTO L2

...

L2 : INC R1

### Algebraic expression simplification

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

### Strength reduction

There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.For example, x \* 2 can be replaced by x << 1, which involves only one left shift. Though the output of a \* a and a2 is same, a2 is much more efficient to implement.

### Accessing machine instructions

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

# Optimization Phases

The phase which represents the pertinent, possible flow of control is often called *control flow analysis*. If this representation is graphical, then a *flow graph* depicts *all* possible execution paths. Control flow analysis simplifies the data flow analysis. Data flow analysis is the proces of collecting information about the *modification*, *preservation* , and *use* of program "quantities"-- usually *variables* and *expressions*.Once control flow analysis and data flow analysis have been done, the next phase, the improvement phase, improves the program code so that it runs faster or uses less space. This phase is sometimes termed *optimization*. Thus, the term optimization is used for this final code improvement, as well as for the entire process which includes control flow analysis and data flow analysis.Optimization algorithms attempt to remove useless code, eliminate redundant expressions, move invariant computations out of loops, etc.

# Control Flow Analysis

Control flow analysis proceeds in two steps: (1) determine the basic blocks -- sequences of straight-line code and (2) build a flow graph by taking branches into account.

# Basic Blocks

A *basic block* is a sequence of intermediate representation constructs (quadruples, abstract syntax trees, whatever) which allow no flow of control *in* or *out* of the block except at the top or bottom. Figure 3 shows the structure of a basic block.

We will use the term *statement* for the intermediate representation and show it in quadruple form because quadruples are easier to read than IR in tree form.

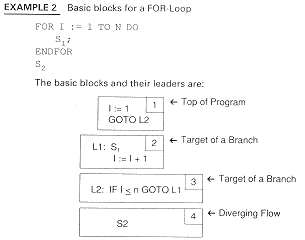
Leaders

A basic block consists of a *leader* and all code before the next leader. We define a *leader* to be (1) the first statement in the program (or procedure), (2) the target of a branch, identified most easily because it has a label, and (3) the statement after a "diverging flow " : the statement after a conditional or unconditional branch.

Basic blocks can be built during parsing if it is assumed that all labels are referenced or after parsing without that assumption. Following Example shows the outline of a FOR-Loop and its basic blocks.

Since a basic block consists of straight-line code, it computes a set of expressions. Many optimizations are really transformations applied to basic blocks and to sequences of basic blocks.

Example: Basic blocks for For Loop

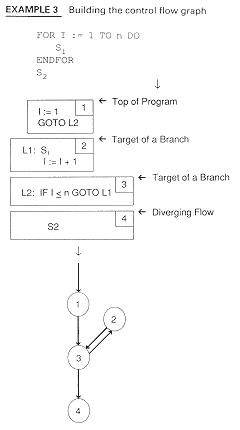


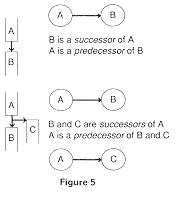
One we have computed basic blocks, we can create the control flow graph.

# Building a Flow Graph

A flow graph shows all possible execution paths. We will use this information to perform optimizations.

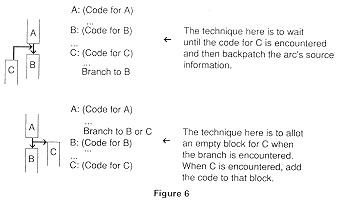
Formally, a *flow graph* is a directed graph G, with N nodes and E edges.

Example: Building the Control Flow Graph



# Techniques for Building Flow Graphs

Building a control flow graph involves finding predecessors and successors.



Consider the first situation in above figure where basic block A has been built and a leader is identified for the beginning of basic block B. We cannot build the tail of the arrow to basic block B until we find the diverging flow that leads here. *Backpatching:* returning and filling in the tail information, will define the source of this arc.

The second picture represents the opposite situation. In the second picture of above figure, we know we have an arc, but haven't found the leader to which the arc should point; the tail of the arc is known, but its head is not. Here, we can allocate a data structure for a basic block and then backpatch when it is found.

# Data Structure for building Control Flow Graphs

Basic blocks may be created and destroyed as the program is modified in the optimization process. Thus, space must be made available for growing collections of blocks. Storage reclamation is essential to handle optimization of large programs.

Since the structure of the flow graph varies with time, the data structure for blocks might keep pointers to lists of successors.

Since statements within the block change with time, it is reasonable to keep the intermediate representation as a linked list and reclaim storage of unused statements.

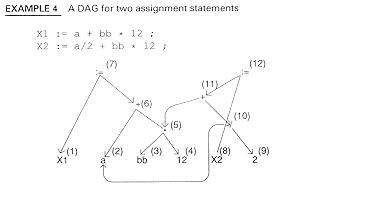
# Directed Acyclic Graphs (DAGs)

The previous sections looked at the control flow graph nodes of basic blocks. DAGs, on the other hands, create a useful structure for the intermediate representation *within* the basic blocks.

A directed graph with no cycles, called a *DAG* (Directed Acyclic Graph), is used to represent a basic block. Thus, the nodes of a flow graph are themselves graphs!

We can create a DAG *instead of* an abstract syntax tree by modifying the operations for constructing the nodes. If the node already exists, the operation returns a pointer to the existing node. Example 4 shows this for the two-line assignment statement example.

Example: A DAG for two-line assignment statement



In above example there are two references to *a* and two references to the quantity *bb \* 12* .

# Data Structure for a DAG

As above example shows, each node may have more than one pointer to it. We can represent a DAG internally by a process known as *value-numbering* . Each node is numbered and this number is entered whenever this node is reused. This is shown following Example.

EXAMPLE: Value-numbering

Node Left Child Right Child

1. X1

2. a

3. bb

4. 12

5. \* (3) (4)

6. + (2) (5)

8.y := (1) (6)

8. X2

9. 2

10. / (2) (9)

11. + (10) (5)

12. := (8) (11)

Advantages of using DAG

When a DAG is created, common subexpressions are detected. Also, creating a DAG makes it easy to see variables and expressions which are used or defined within a block.

DAGs also produce good code at code generation time. In Module 11, we will see a code generation algorithm which produces good (but not optimal!) code from a DAG.

Example 6 shows a bubble sort procedure and a set of quadruples to which it might be translated. *Quadruples* are an intermediate representation consisting of a single operation, up to two operands and a result. They are easier for humans to read than an abstract syntax tree. The quadruples shown here are in "assignment statement" form.

EXAMPLE 6 IR for a bubble sort procedure

FOR I := 1 TO n - 1 DO

FOR J := 1 TO I DO

IF A[J] > A[J+1] THEN

BEGIN

Temp := A[J]

A[J] := A[ J + 1 ]

A[ J + 1 ] := Temp

END

I := 1

GOTO ITest

ILoop: J := 1

GOTO JTest

JLoop: T1 := 4 \* J

T2 := A [T1] ; A[J]

T3 := J + 1

T4 := 4 \* T3

T5 := A[T4] ; A[ J + 1 ]

IF T2 <= T5 GOTO JPlus

T6 := 4 \* J

Temp := A[T6] ; Temp := A[J]

T7 := J + 1

T8 := 4 \* T7

T9 := A[T8] ; A { J + 1 ]

T10 := 4 \* J

A[T10] := T9 ; A[J] := A[ J + 1 ]

T11 := J + 1

T12 := 4 \* T11

A[T12] := Temp ; A [ J + 1 ] := Temp

JPlus: J := J + 1

JTest: IF J < = I GOTO JLoop

IPlus: I := I + 1

ITest: IF I <= n - 1 GOTO ILoop

The intermediate code shown in Example 6 is rather high-level. For machines which have no indexing modes, the references to A[...] will have to be computed by the compiler. Also, the IF statements really involve several quadruples Using "( )" to mean "contents of " and "addr" to mean "address of", the low-level quadruples for "Temp := List[i]" might be:

T1 := addr(List)

T2 := T1 + I

temp := (T2)

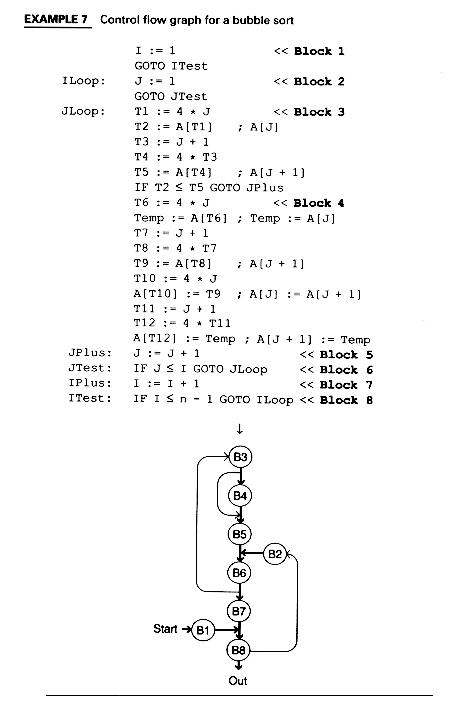
Similarly, if the array reference is on the left-hand side of an assignment statement as in "List[i] := temp", low-level quadruples would be:

T1 := addr (List)

T2 := T1 + I

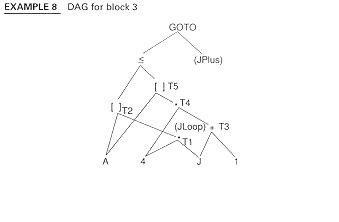
(T2) := Temp

Even if a machine has an indexing addressing mode, translating array references into their low-level format may allow optimizations (for example, if the subscript reference were a common subexpression).

Following Example shows the basic blocks and the control flow graph for the procedure in above case example.Example: Control Flow Graph for Bubble Sort

Following Example shows a DAG for block 3 of above example.

Example: DAG for above example Block 3



# Loops

Programs spend the bulk of their time in loops. Thus, the payoff in loop optimization is potentially high.

There are two main types of loop optimizations: (1) movement of loop-invariant computations outside of the loop and (2) elimination of excess induction variables, the variables which are a linear function of the loop index. Other optimizations often performed within loops are reductions in strength of certain multiplications to additions.

Following Example shows invariant code movement. The details of how this optimization is found and implemented are given in the next two modules. In this module we focus on the importance of being able to identify a loop.

EXAMPLE: Invariant code motion

FOR index := 1 TO 10000 DO t := y \* z

BEGIN FOR index := 1 TO 10000 DO

x := y \* z ; BEGIN

j := index \* 3 ; -- > x := t

END j := index \* 3

. END

. .

In above Example, y \* z is computed each time around the loop. If none of y, z, or x changes value in the loop (and this must be checked!), then it is more efficient to compute this product, t, one and to use t every place the computation appears in the loop (x itself may be able to be be eliminated).

**CONCLUSION:**

Hence, we have successfully implemented code optimization using directed acyclic graph.

**OUTCOME**

**Upon completion Students will be able to:**

1. Optimize the code.
2. Use DAG to optimize the code.

**QUESTIONS:**

1. What is code optimization? 
2. How DAG is used for code optimization? 
3. Define peephole optimization and list it’s characteristics. 
4. Mention the issues to be considered while applying the techniques for code optimization.
5. What do you mean by machine dependent and machine independent optimization? 
6. What are the criteria that need to be considered while applying the code

optimization techniques? 

**EXPERIMENT NO. 5**

**Code Generation using DAG/Labeled Tree**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 05 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 10 | Concept of DAG & Labeling Algorithm | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 25 | Explanation of code generation algorithm | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Code Generation using DAG/Labeled Tree

**OBJECTIVES:**

1. To express concept of DAG and Labeled Tree.

2. To apply the code generation algorithm to generate target code.

**PROBLEM STATEMENT:**

Accept Postfix expression. Create a DAG from that expression. Apply Labeling algorithm to DAG and then apply code generation algorithm to generate target code from DAG.

**SOFTWARE REQUIRED:** 64-bit Fedora or equivalent OS, LEX, YACC.

**INPUT:** Input data as DAG/Labeled Tree which is generated from postfix expression.

**OUTPUT:** It will create a target code.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the code generation such that

S={s, e, i, o, f, DD, NDD, success, failure}

s=initial state that is start of program

e = the end state or end of program

i= postfix expression

o=target code

f= {ComputeLabel ( ), PostTraverse( ), Gencode ( )}

ComputeLabel ( )={Applies Labeling Algorithm to tree generated from postfix expression}

PostTraverse( )={ displays tree in postorder}

Gencode() = { Handles different cases of tree & accordingly generate target code}

COMPUTATIONAL MODEL:

e2

e1 e3

**B**

**A**

**S**

Where S : Initial State

A : DAG /Labeled Tree

B: Target Code

Edge e1 : postfix expression

e2 : Compute Label

e3: root node of labeled tree

System Accepts postfix expression & enters into A state . In that state, it generates tree, at the same time, calculates label of tree. After that, it passes root node of Label tree to gencode function & enters into B state. In B state it applies code generation algorithm & generate target code.

Success- desired output is generated as target code in assembly language form

Failure- desired output is not generated as target code in assembly language form

**THEORY:**

The Labeling Algorithm

The labeling can be done by visiting nodes in a bottom-up order so that a node is not visited until all it’s children are labeled.

Fig.1.1 gives the algorithm for computing label at node n.



Fig.1.1 The Labeling Algorithm

In the important special case that n is binary node and it’s children have labels l1 and l2, the above formula reduces to



Example:



Fig. 1.2 Labeled Tree

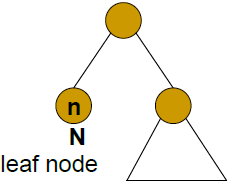
Node a is labeled 1 since it is left leaf. Node b is labeled 0 since it is right leaf. Node t1 is labeled 1 because the labels of it’s children are unequal and the maximum label of a child is 1. Fig.1.2 shows labeled tree that results.

Code generation from a Labeled Tree

Procedure GENCODE(n)

* RSTACK –stack of registers, R0,...,R(r-1)
* TSTACK –stack of temporaries, T0,T1,...
* A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top(RSTACK) ,and
  + the rest of RSTACK remains in the same state as the one before the call
* A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child

Procedure gencode(n);

{ /\* case 0 \*/

If

n is a leaf representing

operand N and is the

leftmost child of its parent

then

print(LOAD N, top(RSTACK))



/\* case 1 \*/

else if

n is an interior node with operator OP, left child n1, and right child n2

then

if label(n2) == 0 then {

let N be the operand for n2;

gencode(n1);

print(OP N, top(RSTACK));

}



/\* case 2 \*/

else if ((1 < label(n1) < label(n2))

and( label(n1) < r))

then {

swap(RSTACK); gencode(n2);

R := pop(RSTACK); gencode(n1);

/\* R holds the result of n2 \*/

print(OP R, top(RSTACK));

push (RSTACK,R);

swap(RSTACK);

}

The swap() function ensures that a node is evaluated into the same register as its left child



/\* case 3 \*/

else if ((1 < label(n2) < label(n1))

and( label(n2) < r))

then {

gencode(n1);

R := pop(RSTACK); gencode(n2);

/\* R holds the result of n1 \*/

print(OP top(RSTACK), R);

push (RSTACK,R);

}



/\* case 4, both labels are > r \*/

else {

gencode(n2); T:= pop(TSTACK);

print(LOAD top(RSTACK), T);

gencode(n1);

print(OP T, top(RSTACK));

push(TSTACK, T);

}

}

Example:

Consider Fig.1.2 Labeled Tree with rstack = R0, R1 initially. The sequence of calls to gencode and code printing steps is shown below.



Fig.1.3 Trace of gencode routine

**CONCLUSION:**

Hence, we have successfully studied DAG, Labeling algorithm and code generation from Labeled Tree.

**OUTCOME**

**Upon completion Students will be able to:**

* Explain the labeling algorithm. DB13.bmp
* Apply code generation algorithm to generate target code from Labeled Tree. DB12.bmp

**QUESTIONS:**

1. Describe issues in code generation phase DB13.bmp
2. Difference between a AST and DAG 
3. What are the applications of DAG 
4. Explain Labeling Algorithm DB13.bmp
5. Explain code generation algorithm for DAG DB13.bmp
6. How to Generate DAG from postfix expression. DB13.bmp
7. What are possible inputs to code generation phase. 
8. Compute DAG for following DB12.bmp
9. X1 = a+b\*12 (ii) X2 = a/2+b\*12
10. For following expression

(a+b) – ( c- ( d+e) )

* + - 1. Draw expression tree and Apply labeling algorithm to generate labeled tree DB12.bmp
      2. Show optimal code for labeled expression tree DB12.bmp

**PRACTICE ASSIGNMENT:**

1. Accept input as 3-aadress code & generate target code for same.
2. Implement a program to generate DAG and target code for following piece of code

Repeat {

p=p+x[i]/y[i]

}

Until i> 100

**EXPERIMENT NO.6**

**Generating abstract syntax tree using Lex and YACC.**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 15 | Concept of abstract syntax tree | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Generating abstract syntax tree using Lex and YACC.

**OBJECTIVES:** To express and apply the concept of abstract syntax tree.

**PROBLEM STATEMENT:** Generate an abstract syntax tree using Lex and YACC

**SOFTWARE REQUIRED:** Lex and YACC

**INPUT:** Input data as default values and user defined values.

**OUTPUT:** Abstract syntax tree.

**MATHEMATICAL MODEL**

Let S be the solution perspective of the abstract syntax tree such that

S={s, e, i, o, f, success, failure}

s=initial state

e = the end state

i= input of the system.

o=output of the system.

Success-desired outcome generated.

Failure-Desired outcome not generated or forced exit due to system error.

**s e1 e2**

Where,

s is the initial state

e1 is the input

B is the token generator and

R is the result.

**THEORY:**

What is abstract syntax tree?

In [computer science](http://en.wikipedia.org/wiki/Computer_science), an abstract syntax tree (AST), or just syntax tree, is a [tree](http://en.wikipedia.org/wiki/Directed_tree) representation of the [abstract syntactic](http://en.wikipedia.org/wiki/Abstract_syntax) structure of [source code](http://en.wikipedia.org/wiki/Source_code) written in a [programming language](http://en.wikipedia.org/wiki/Programming_language). Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping [parentheses](http://en.wikipedia.org/wiki/Bracket" \l "Parentheses) are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from [concrete syntax trees](http://en.wikipedia.org/wiki/Concrete_syntax_tree), traditionally designated [parse trees](http://en.wikipedia.org/wiki/Parse_tree), which are often built by a [parser](http://en.wikipedia.org/wiki/Parser) during the source code translation and [compiling](http://en.wikipedia.org/wiki/Compiler) process. Once built, additional information is added to the AST by means of subsequent processing, e.g., [contextual analysis](http://en.wikipedia.org/wiki/Semantic_analysis_(compilers)). Abstract syntax trees are also used in program analysis and [program transformation](http://en.wikipedia.org/wiki/Program_transformation) systems.

An abstract syntax tree (AST) is a tree model of an entire program or a certain “program structure” (e.g., a statement or an expression in a Java program). An AST is “abstract” in the sense that some of the actual characters used in the “concrete” program text do not appear in the AST .

The representation of [SourceCode](http://c2.com/cgi/wiki?SourceCode) as a tree of nodes representing constants or variables (leaves) and operators or statements (inner nodes). Also called a "parse tree". An [AbstractSyntaxTree](http://c2.com/cgi/wiki?AbstractSyntaxTree) is often the output of a parser (or the "parse stage" of a compiler), and forms the input to semantic analysis and code generation (this assumes a phased compiler; many compilers interleave the phases in order to conserve memory).

The [AbstractSyntaxTree](http://c2.com/cgi/wiki?AbstractSyntaxTree) reveals the lexical/syntactical structure of the program text - what blocks and statements are lexically contained within in what. This may - or may not - be related to the semantic structure of the program. Unlike concrete syntax, which consists of a linear sequence of characters and/or tokens, along with a set of rules for parsing them, abstract syntax doesn't (generally) have to worry about issues such as parser ambiguity, operator precedence, etc.

[AbstractSyntaxTree](http://c2.com/cgi/wiki?AbstractSyntaxTree)s are a common intermediate form during compilation of [SourceCode](http://c2.com/cgi/wiki?SourceCode).

They are widely used in [compilers](http://en.wikipedia.org/wiki/Compilers), due to their property of representing the structure of program code. An AST is usually the result of the [syntax analysis](http://en.wikipedia.org/wiki/Syntax_analysis) phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

Generating abstract syntax tree using Lex and YACC

Yacc actions appear to the right of each rule, much like lex actions. We can associate pretty much any C code that we like with each rule. Consider the yacc file. Tokens are allowed to have values, which we can refer to in our yacc actions. The $1 in printf("%s",$1) (the 5th rule for exp) refers to the value of the IDENTIFIER (notice that IDENTIFIER tokens are specified to have string values). The $1 in printf("%d",$1) (the 6th rule for exp) refers to the value of the INTEGER LITERAL. We use $1 in each of these cases because the tokens are the first items in the right-hand side of the rule.

Example

4 \* 5 + 6

First, we shift INTEGER LITERAL(4) onto the stack. We then reduce by the rule exp: INTEGER LITERAL, and execute the printf statement, printing out a 4. We then shift a TIMES, then shift an INTEGER LITERAL(5). Next, we reduce by the rule exp: INTEGER LITERAL and print out a 5. Next, we reduce by the rule exp: exp TIMES exp (again, remember those precedence directives!) and print out a \*. Next, we shift a PLUS and an INTEGER LITERAL(6). We reduce by exp: INTEGER LITERAL (printing out a 6), then we reduce by the rule exp: exp + exp (printing out a +), giving an output of:

4 5 \* 6 +

So what does this parser do? It converts infix expressions into postfix expressions.

Creating an Abstract Syntax Tree

C Definitions Between %{ and %} in the yacc file are the C definitions. The #include is necessary for the yacc actions to have access to the tree types and constructors, defined in treeDefinitions.h. The global variable root is where yacc will store the finished abstract syntax tree.

• %union: The %union{ } command defines every value that could occur on the stack – not only token values, but non-terminal values as well. This %union tells yacc that the types of values that we could push on the stack are integers, strings (char \*), and expressionTrees. Each element on the yacc stack contains two items – a state, and a union (which is defined by %union). by %union{ }.

• %token: For tokens with values, %token <field> tells yacc the type of the token. For instance, in the rule exp : INTEGER LITERAL { $$ = ConstantExpression($1)}, yacc looks at the union variable on the stack to get the value of $1. The command %token <integer value> INTEGER LITERAL tells yacc to use the integer value field of the union on the stack when looking for the value of an INTEGER LITERAL.

• %type: Just like for tokens, these commands tell yacc what values are legal for non-terminal commands

• %left Precedence operators: • Grammar rules The rest of the yacc file, after the %%, are the grammar rules, of the form

<non-terminal> : <rhs> { /\* C code \*/ }

where <non-terminal> is a non-terminal and <rhs> is a sequence of tokens and non-terminals.

Let’s look at an example. For clarity, we will represent the token INTEGER LITERAL(3) as just 3. Pointers will be represented graphically with arrows. The stack will grow down in our diagrams. Consider the input string 3+4\*5. First, the stack is empty, and the input is 3+4\*x.

Example: Creating an abstract syntax tree for simple expressions

%{

#include "treeDefinitions.h"

expressionTree root;

%}

%union{

int integer\_value;

char \*string\_value;

expressionTree expression\_tree;

}

%token <integer\_value> INTEGER\_LITERAL

%token <string\_value> IDENTIFIER

%token PLUS MINUS TIMES DIVIDE

%type <expresstion\_tree> exp

%left PLUS MINUS

%left TIMES DIVIDE

%%

prog : exp { root = $$; }

exp : exp PLUS exp { $$ = OperatorExpression(PlusOp,$1,$3); }

| exp MINUS exp { $$ = OperatorExpression(MinusOp,$1,$3); }

| exp TIMES exp { $$ = OperatorExpression(TimesOp,$1,$3); }

| exp DIVIDE exp { $$ = OperatorExpression(DivideOp,$1,$3); }

| IDENTIFIER { $$ = IdentifierExpression($1); }

| INTEGER\_LITERAL { $$ = ConstantExpression($1); }

/\* File treeDefinitions.c \*/

#include "treeDefinitions.h"

#include <stdio.h>

expressionTree operatorExpression(optype op, expressionTree left,

expressionTree right) {

expressionTree retval = (expressionTree) malloc(sizeof(struct expression));

retval->kind = operatorExp;

retval->u.oper.op = op;

retval->u.oper.left = left;

retval->u.oper.right = right;

return retval;

}

expressionTree IdentifierExpression(char \*variable) {

expressionTree retval = (expressionTree) malloc(sizeof(struct expression));

retval->kind = variableExp;

retval->u.variable = variable;

return retval;

}

expressionTree ConstantExpression(int constantval) {

expressionTree retval = (expressionTree) malloc(sizeof(struct expression));

retval->kind = constantExp;

retval->u.constantval = constantval;

return retval;

}

**CONCLUSION**

Hence, we have successfully studied concept of abstract syntax tree.

**OUTCOME**

**Upon completion Students will be able to:**

Explain and apply the abstract syntax tree**. DB12.bmp**

**QUESTIONS**

1. What is abstract syntax tree? 
2. Explain the advantage of abstract syntax tree. 
3. Abstract syntax tree is usually the result of the [syntax analysis](http://en.wikipedia.org/wiki/Syntax_analysis) phase of a compiler. Yes or no.
4. What is %union command? 
5. What is %token? 
6. What is %type? 
7. What is %left precedence operator? 

**PRACTICE ASSIGNMENTS**

1. Generate a abstract syntax tree using Lex and YACC.

**EXPERIMENT NO.7**

**Implementing Recursive Descent Parser for Sample Language**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 05 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Concept of Recursive Descent Parser | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 20 | Explanation of how to implement RDP | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Implementing RDP for sample language

**OBJECTIVES:**

1. To express concept of RDP.

2. To implement RDP.

**PROBLEM STATEMENT:**

Accept sample language expression and generate recursive decent parser for the same.

**SOFTWARE REQUIRED:** 64-bit Fedora or equivalent OS, TurboC.

**INPUT:** Sample language expression.

**OUTPUT:** It will create parser for language expression

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the recursive descent parser such that

S={s, e, i, o, f, success, failure}

s=initial state that is start of program

e = the end state or end of program

i= input expression

o=parsed output

f= {IsTerminal(),Match ( ), Error( )}

IsTerminal()={Check whether input Character is terminal or non-terminal}

Match ( )={Matches the current input token against a predicted token }

Error( )={ Generate Error message}

**COMPUTATIONAL MODEL:**

e1 e2

**T**

**S**

e3

Where S : Initial State

T: Function checking terminals

P: Procedure For terminal

B: Match non-terminal

Edge e1 : Scan input character

e2 : Call procedure for Terminal

e3: matches non-terminal and print it.

System Accepts input expression & enters into T state . In that state, it checks current input for terminal .If it is terminal then procedure for current terminal is created .If current symbol is non-terminal then it will be matched with grammar defined.

**Success**- Given input is successfully parsed.

**Failure-** parser giving unable to process the input language

**THEORY:**

**Tokenizing**

Tokenization is the process of converting input program text into a sequence of tokens

**LL(1) Grammars**

A context-free grammar whose Predict sets are always disjoint (for the same non-terminal) is said to be LL(1). LL(1) grammars are ideally suited for top-down parsing because it is always possible to correctly predict the expansion of any non-terminal. No backup is ever needed. LL(1) grammars are easy to parse in a top-down manner since predictions are always correct.

**Recursive Descent Parsing:**Recursive descent parsing is a method of writing a compiler as a collection of recursive functions. This is usually done by converting a BNF grammar specification directly into recursive functions.

**Recursive Descent Parsers**

A *recursive descent parser* is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools. It is a simple and effective technique, but is not as powerful as some of the shift-reduce parsers -- not the one presented in class, but fancier similar ones called *LR parsers*.

This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

Perhaps the hardest part of a recursive descent parser is the scanning: repeatedly fetching the next token from the scanner. It is tricky to decide when to scan, and the parser doesn't work at all if there is an extra scan or a missing scan.

Although parsers can be generated by parser generators, it is still sometimes convenient to write a parser by hand. However, LALR(1) grammars are not easy to use to manually construct parsers. Instead, we want an LL(1) grammar if we are going to manually construct a parser. An LL(1) grammar can be used to construct a top-down or recursive descent parser where an LALR(1) grammar is typically used to construct a bottom-up parser. A top-down parser constructs (or at least traverses) the parse tree starting at the root of the tree and proceeding downward. A bottom-up parser constructs or traverses the parse tree in a bottom-up fashion.

In a recursive descent parser, each non-terminal in the grammar becomes a function in a program. The right hand side of the productions becomes the bodies of the functions. An LALR(1) grammar is not appropriate for constructing a recursive descent parser. To create a recursive-descent parser (the topic of this page) we must convert the LALR(1) grammar above to an LL(1) grammar. Typically, there are two steps involved.

* Eliminate left recursion
* Perform left factorization where appropriate

**Eliminate Left Recursion**

Eliminating left recursion means eliminating rules like *Expr Expr + Term*. Rules like this are left recursive because the *Expr* function would first call the *Expr* function in a recursive descent parser. Without a base case first, we are stuck in infinite recursion (a bad thing). To eliminate left recursion we look to see what *Expr* can be rewritten as. In this case, *Expr* can be only be replaced by a *Term* so we replace *Expr* with *Term* in the productions. The usual way to eliminate left recursion is to introduce a new non-terminal to handle all but the first part of the production. So we get

*Expr -> Term RestExpr*

*RestExpr -> + Term RestExpr | - Term RestExpr | <null>*

We must also eliminate left recursion in the *Term Term \* Factor | Term / Factor* productions in the same way. We end up with an **LL(1) grammar** that looks like this:

*Prog -> Expr EOF*

*Expr -> Term RestExpr*

*RestExpr -> + Term RestExpr | - Term RestExpr | <null>*

*Term -> Storable RestTerm*

*RestTerm -> \* Storable RestTerm | / Storable RestTerm | <null>*

*Storable -> Factor S | Factor*

*Factor -> number | R | ( Expr )*

**Perform Left Factorization**

Left factorization isn't needed on this grammar so this step is skipped. Left factorization is needed when the first part of two or more productions is the same and the rest of the similar productions are different. Left factorization is important in languages like Prolog because without it the parser is inefficient. However, it isn't needed and does not improve efficiency when writing a parser in an imperative language like Java, for instance.

**Building A Recursive Descent Parser**

We start with a procedure Match, that matches the current input token against a predicted token:

void Match(Terminal a)

{

if (a == currentToken)

currentToken = Scanner();

else

SyntaxErrror();}

To build a parsing procedure for a non-terminal A, we look at all productions with A on the lefthand side:

A→X1...Xn |

A→Y1...Ym | ...

We use predict sets to decide which production to match (LL(1) grammars always have disjoint

predict sets) We match a production’s right hand side by calling Match to match terminals, and calling parsing procedures to match non-terminals.

The general form of a parsing procedure for

A→X1...Xn | A→Y1...Ym | ... is

void A()

{

if (currentToken in Predict(A→X1...Xn))

for(i=1;i<=n;i++)

if (X[i] is a terminal)

Match(X[i]);

Else

X[i]();

Else

if (currentToken in Predict(A→Y1...Ym))

for(i=1;i<=m;i++)

if (Y[i] is a terminal)

Match(Y[i]);

else

Y[i]();

else

// Handle other A→...productions

else // No production predicted

SyntaxError();

}

**CONCLUSION:**

Hence, we have successfully studied to eliminate Left Recursion and generate a Recursive Descent Parser.

**OUTCOME**

**Upon completion Students will be able to:**

* Explain the concept of Recursive Descent Parser. DB13.bmp
* Apply Recursive decent parsing to parse sample language consruct . DB12.bmp

**QUESTIONS:**

1. Describe Top down parsing. DB13.bmp
2. Difference between Top down parsing and bottom up parsing 
3. What left recursion. 
4. Explain how to eliminate left recursion from grammar . DB13.bmp
5. Explain procedure for recursive decent parser. DB13.bmp
6. Remove left recursion form grammar if any. DB12.bmp

E→ E + T | T

T→ T \* F | F

F→ (F) |id

**PRACTICE ASSIGNMENT:**

1. Implement a recursive descent parser for arithmetic expression.

**EXPERIMENT NO.8**

**SLR Parsing algorithm using Python for the ordered input Set in XML**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 10 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 15 | Explanation of Problem statement | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens,  Participates, | Knowledge, intrapersonal, Application |
| 15 | Concept of SLR | Demonstration, Presentation | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 10 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** SLR Parsing algorithm using Python for the ordered input Set in XML

**OBJECTIVES:**.

To express and apply the concept of SLR

**PROBLEM STATEMENT:** Write a program to implement SLR Parsing algorithm using Python for the ordered input Set in XML

{ P-> E, E->E+T, E->T, T->T\*F, T->F, F->(E), F->i, END. }

**SOFTWARE REQUIRED:** Python

**INPUT:** Grammar G with some productions

**OUTPUT:**

**MATHEMATICAL MODEL:**

**THEORY:**

**Concept of SLR**

Simple LR or SLR parser is a type of [LR parser](http://en.wikipedia.org/wiki/LR_parser) with small [parse tables](http://en.wikipedia.org/wiki/LR_parser" \l "Constructing_LR.280.29_parsing_tables) and a relatively simple parser generator algorithm. As with other types of LR(1) parser, an SLR parser is quite efficient at finding the single correct [bottom-up parse](http://en.wikipedia.org/wiki/Bottom-up_parsing) in a single left-to-right scan over the input stream, without guesswork or backtracking. The parser is mechanically generated from a formal grammar for the language. SLR generators calculate that lookahead by an easy approximation method based directly on the grammar, ignoring the details of individual parser states and transitions. This ignores the particular context of the current parser state. If some nonterminal symbol S is used in several places in the grammar, SLR treats those places in the same single way rather than handling them individually. The SLR generator works out Follow(S), the set of all terminal symbols which can immediately follow some occurrence of S. In the parse table, each reduction to S uses Follow(S) as its LR(1) lookahead set. Such follow sets are also used by generators for LL top-down parsers. A grammar that has no shift/reduce or reduce/reduce conflicts when using Follow sets is called an [SLR grammar](http://en.wikipedia.org/wiki/SLR_grammar).

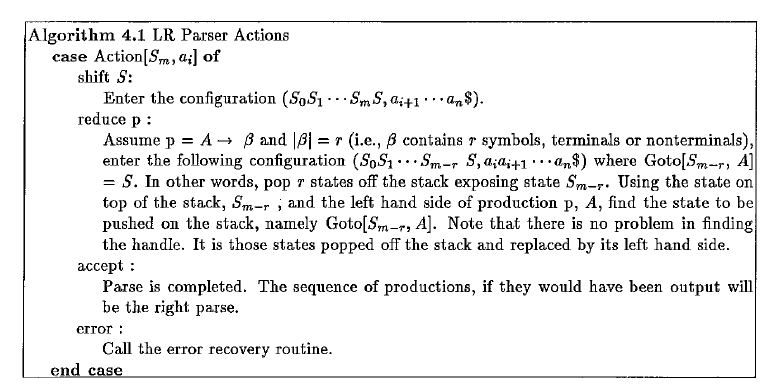
* For example, the production A à XYZ yields four items,
  + A à .XYZ
  + A à X.YZ
  + A à XY.Z
  + A à XYZ.
* A production rule of the form A à e yields only one item A à . . Intuitively, an item shows how much of a production we have seen till the current point in the parsing procedure.
* Augmented Grammar G’: This equals G È {S’ à S} where S is the start state of G. The start state of G’ = S’. This is done to signal to the parser when the parsing should stop to announce acceptance of input.
* Closure Operation: If I[] is a set of items for a grammar G,

closure(I) = I È {B à .g | (A à a.Bb Î closure(I)) Ù ((B à g) Î grammar G}

* Goto Operation: For a set of items I, and grammar symbol X,

goto(I, X) = { Closure (all items containing A à aX.b) such that A à a.Xb is in I}

= set of items that are valid for the viable prefix gX, where I is valid for some viable prefix g.



**CONCLUSION:**

Hence, we have successfully studied concept of SLR Parsing algorithm using Python for the ordered input Set in XML

**OUTCOME**

**Upon completion Students will be able to:**

1. What is parsing? 
2. Explain SLR parsing. 

**QUESTIONS**

* 1. What are different types of parsing. 
  2. Explain SLR parsing advantages. 

**GROUP C**

**EXPERIMENT NO. 1**

**Code Generation using iburg tool**

**Session Plan**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time**  **( min)** | **Content** | **Learning Aid / Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill / Competency Developed.** |
| 05 | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains | Listens, Participates, Discusses | Knowledge, intrapersonal |
| 40 | Explanation of code generation algorithm | Demonstration, Presentation,  Chalk & Talk | Explains, Facilitates, Monitors | Listens,  Participates,  Discusses | Knowledge, intrapersonal,  interpersonal  Application |
| 60 | Implementation of problem statement | N/A | Guides, Facilitates  Monitors | Participates, Discusses | Comprehension,  Hands on experiment |
| 10 | Assessment | N/A | Monitors | Participates, Discusses | Knowledge, Application |
| 05 | Conclusions | Keywords | Lists, Facilitates | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension |

**TITLE:** Code Generation using iburg tool

**OBJECTIVES:**

1. How to use iburg tool ?

2. To apply the iburg tool to generate target code.

**PROBLEM STATEMENT:**

Study of iburg tool & generate target code using that tool

**SOFTWARE REQUIRED:** 64-bit Fedora or equivalent OS, iburg.

**INPUT:** Tree grammer

**OUTPUT :** C code forParse Tree

**MATHEMATICAL MODEL:**

S={s, e, i, o, f, success, failure}

s=initial state that is start of program

e = the end state or end of program

i= Grammer rules in in Backus-Naur form

o= C source code for grammer-specific tree parsers

**Success**- desired output is generated

**Success**- desired output is not generated

**Function Mapping**

C source code for grammer specific tree parser

Tree grammer rules in Backus-Naur form

**Iburg program**

**THEORY:**

Iburg is a program that generates a fast tree parser. It is compatible with Burg. Both programs accept a cost-augmented tree grammar and emit a C program that discovers an optimal parse of trees in the language described by the grammar. They have been used to construct fast optimal instruction selectors for use in code generation. Burg uses BURS; Iburg's matchers do dynamic programming at compile time

Iburg that reads a burg specification and writes a matcher that does DP at compile time. The matcher is hard coded.

Iburg was built to test early versions of what evolved into burg's specification language and interface, but it is useful in its own right because it is simpler and thus easier for novices to understand, because it allows dynamic cost computation, and because it admits a larger class of tree grammars

Iburg tree parser generator has been mainly developed at Princeton University. It accepts Backus-Naur form specifications of tree grammers and generates C source code for grammer-specific tree parsers.



Figure 1 shows an extended BNF grammar for burg and iburg specifications. Grammar symbols are displayed in slanted type and terminal symbols are displayed in typewriter type. { X } denotes zero or more instances of X, and [X] denotes an optional X. Specifications consist of declarations, a %% separator, and rules. The declarations declare terminals | the operators in subject trees | and associate a unique, positive external symbol number with each one. Non-terminals are declared by their presence on the left side of rules. The %start declaration optionally declares a non-terminal as the start symbol. In Figure 1, term and nonterm denote identifiers that are terminals and non-terminals, respectively

Rules define tree patterns in a fully parenthesized prefix form. Every non-terminal denotes a tree. Each operator has a fixed arity, which is inferred from the rules in which it is used. A chain rule is a rule whose pattern is another non-terminal. If no start symbol is declared, the non-terminal defined by the first rule is used. Each rule has a unique, positive external rule number , which comes after the pattern and is preceded by a \=". As described below, external rule numbers are used to report the matching rule to a user-supplied semantic action routine. Rules end with an optional non-negative, integer cost; omitted costs default to zero.

Figure 2 shows a fragment of a burg specification for the VAX. This example uses upper-case for terminals and lower-case for non-terminals. Lines 1{2 declare the operators and their external symbol numbers, and lines 4{15 give the rules.





The external rule numbers correspond to the line numbers to simplify interpreting subsequent figures. In practice, these numbers are usually generated by a preprocessor that accepts a richer form of specification (e.g., including YACC-style semantic actions), and emits a burg specification . Only the rules on lines 4, 6, 7, and 9 have non-zero costs. The rules on lines 5, 9, 12, and 13 are chain rules.

The operators in Figure 2 are some of the operators in lcc's intermediate language. The operator names are formed by concatenating a generic operator name with a one-character type suffix like C, I, or P, which denote character, integer, and pointer operations, respectively. The operators used in Figure 2 denote integer addition (ADDI), forming the address of a local variable (ADDRLP), integer assignment (ASGNI), an integer constant (CNSTI), \widening" a character to an integer (CVCI), the integer 0 (I0I), and fetching a character (INDIRC). The rules show that ADDI and ASGNI are binary, CVCI and INDIRC are unary, and ADDRLP, CNSTI, and I0I are leaves.

**MATCHING**

Both versions of burg generate functions that the client calls to label and reduce subject trees. The labeling function, label(p), makes a bottom-up, left-to-right pass over the subject tree p computing the rules that cover the tree with the minimum cost, if there is such a cover. Each node is labeled with (M;C) to indicate that \the pattern associated with external rule M matches the node with cost C."



Figure 3 shows the intermediate language tree for the assignment expression in the C fragment

{ int i; char c; i = c + 4; }

The left child of the ASGNI node computes the address of i. The right child computes the address of c, fetches the character, widens it to an integer, and adds 4 to the widened value, which the ASGNI assigns to i.

The other annotations in Figure 3 show the results of labeling. (M;C) denote labels from matches and [M;C] denote labels from chain rules. The rule from Figure 2 denoted by each M is also shown. Each C sums the costs of the nonterminals on right-hand side and the cost of the relevant pattern or chain rule. For example, the pattern in line 11 of Figure 2 matches the node ADDRLP i with cost 0, so the node is labeled with (11; 0). Since this pattern denotes a disp, the chain rule in line 9 applies with a cost of 0 for matching a disp plus 1 for the chain rule itself. Likewise, the chain rules in lines 5 and 13 apply because the chain rule in line 9 denotes a reg.

Nodes are annotated with (M;C) only if C is less than all previous matches for the non-terminal on the left-hand side of rule M. For example, the ADDI node matches the disp pattern in line 10 of Figure 2, which means it also matches all rules with disp alone on the right-hand side, namely line 9. By transitivity, it also matches the chain rules in lines 5 and 13. But all three of these chain rules yield cost 2, which isn't better than previous matches for those non-terminals.

Once labeled, a subject tree is reduced by traversing it from the top down and performing appropriate semantic actions, such as generating and emitting code. Reducers are supplied by clients, but burg generates functions that assist in these traversals, e.g., one function that returns M and another that identifies subtrees for recursive visits. iburg does the dynamic programming at compile time and annotates nodes with data equivalent to (M;C). Its \state numbers" are really pointers to records that hold these data.

**IMPLEMENTATION**

iburg generates a state function that uses a straightforward implementation of tree pattern matching. It generates hard code instead of tables. Its \state numbers" are pointers to state records, which hold vectors of the (M;C) values for successful matches. The state record for the specification in Figure 2 is

struct state {

int op;

struct state \*left, \*right;

short cost[6];

short rule[6];

};

iburg also generates integer codes for the non-terminals, which index the cost and rule vectors:

#define stmt\_NT 1

#define disp\_NT 2

#define rc\_NT 3

#define reg\_NT 4

#define con\_NT 5

By convention, the start non-terminal has value 1.

State records are cleared when allocated, and external rule numbers are positive. Thus, a non zero value for p->rule[X] indicates that p's node matched a rule that defines non-terminal X.

**CONCLUSION:**

Hence, we have successfully studied iburg tool & how to generate target code using that tool.

**OUTCOME**

**Upon completion Students will be able to:**

* Explain the working of iburg tool. DB13.bmp
* Demonstrate iburg tool to generate target code. DB12.bmp

**QUESTIONS:**

1. Describe advantages of iburg DB13.bmp
2. Describe drawbacks of iburg if any DB13.bmp
3. What are the applications of iburg 
4. What are different versions of iburg 
5. What are possible inputs to code generation phase. 
6. What are issues in code generation phase. 

**PRACTICE ASSIGNMENT:**

* 1. Write iburg program for any application in DSP or embedded system etc.

**REFERENCES**

1. A V Aho, R Sethi, J D Ullman, \Compilers: Principles, Techniques, and Tools",

Pearson Edition, ISBN 81-7758-590-8.

1. J R Levin, T Mason, D Brown, \Lex and Yacc", O'Reilly, 2000 ISBN 81-7366-061-X
2. Compiler Construction Using Java, JavaCC and Yacc, Anthony J. Dos Reis, Wiley

ISBN 978-0-470-94959-7

1. [http://campus.indiraicem.ac.in/document/Assignment%20no%205.pdf](http://campus.indiraicem.ac.in/document/Assignment no 5.pdf)