**Innovative Assignment**

**Flight Data Analysis using Hive**

**Date:** 2 Nov, 2023

**Course Code and Name:** 2CS702 Big Data Analysis

|  |  |
| --- | --- |
| **Sr No** | **Title** |
| 1 | Problem Statement |
| 2 | Introduction |
| 3 | Hive |
| 4 | Implementation |
| 5 | Results |
| 6 | Applications |
| 7 | Conclusion |

**BY:**

**20BCE204**

**20BCE206**

**Problem statement:**

A. Find list of Airports operating in the Country India

B. Find the list of Airlines having zero stops

C. List of Airlines operating with code share

D. Which country (or) territory having highest Airports

E. Find the list of Active Airlines in United states

**Overview:**

**Introduction**

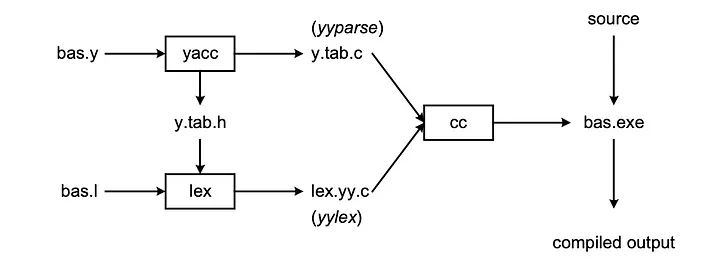
In the realm of algorithmic development and code translation, the task of seamlessly converting pseudocode into executable programming languages poses a significant challenge. This project addresses this challenge with a specialized focus on the Grammar-Based Approach, employing the powerful combination of Lex and Yacc tools.

The primary goal of this endeavor is to create a robust system capable of taking pseudocode as input and, through the application of Lex and Yacc, seamlessly generating equivalent C code. Unlike brute force methods, our approach revolves around systematically parsing the pseudocode, tokenizing it with Lex, and defining grammatical rules with Yacc to facilitate a structured conversion process.

The crux of our system lies in the Intermediate Code Generation (ICG) step, where pseudocode is transformed into a structured representation in the C programming language. This process ensures not only syntactic accuracy but also adherence to the logical flow inherent in the pseudocode.

As we delve into the intricate nuances of algorithmic translation, the utilization of Lex and Yacc emerges as a strategic choice, providing a rule-based and systematic means to navigate the complexities of pseudocode-to-C conversion. This project aims to deliver a specialized solution, streamlining the translation process and contributing to the efficiency and precision of algorithm implementation in C.

**Lex and yacc**



Lex and Yacc, often used in tandem, play crucial roles in the process of transforming high-level language descriptions into executable code. In this report, we delve into the implementation and significance of Lex and Yacc in the context of lexical and syntactic analysis.

**Lex: Generating Lexical Analyzers**

Lex is a powerful tool that generates lexical analyzers. Its primary function is to transform an input stream into a sequence of tokens. The lexical analyzer, created using Lex, reads the input stream and produces the source code in a specified language, commonly C. The process involves defining patterns in the input and associating them with corresponding actions to generate meaningful tokens.

**Lexical Analyzer Workflow:**

1. **Lex Program Generation:**

- A Lex program, typically named lex.1, is created to define patterns and corresponding actions.

2. **Compilation:**

- The Lex compiler processes lex.1, generating a C program named lex.yy.c.

3. **C Compiler Execution:**

- The C compiler then compiles lex.yy.c, producing an object program, usually named a.out.

4. **Tokenization:**

- The resulting a.out is a lexical analyzer that transforms an input stream into a sequence of tokens based on the defined patterns.

**Yacc: Compiler for LALR (1) Grammars**

Yacc, an acronym for "Yet Another Compiler Compiler," serves as a tool to produce parsers for a given grammar. It is specifically designed to compile LALR (1) grammars, providing a robust solution for generating syntactic analyzers.

**Yacc Workflow:**

**1.Grammar Specification:**

· Yacc takes a grammar or rule specification as input, defining the syntax of the language.

**2. Parser Generation:**

· The output of Yacc is a C program, often containing the syntactic analyzer for the language specified by the grammar.

**3. Parsing Tables:**

· The parsing tables, crucial for the syntactic analysis process, are stored in the output file named "file.output."

**4. Header Declarations:**

· Declarations necessary for the parser are stored in the file "file.tab.h."

**5. Parser Function:**

· The generated C program includes a parser function, typically named yyparse(), which expects to use a lexical analyzer function called yylex() to retrieve tokens from the input stream.

**Implementation**

**CC\_INNOV\_LEX.l file**

letter [a-zA-Z]

digit [0-9]

%{

#include<stdio.h>

#include<string.h>

#include "y.tab.h"

extern YYSTYPE yylval;

%}

%option yylineno

%%

"begin" return BEG;

"end" return END;

"int" {yylval.dataType=strdup(yytext);return INT;}

"Integer" {yylval.dataType=strdup(yytext);return INT;}

"float" {yylval.dataType=strdup(yytext);return FLOAT;}

"long" {yylval.dataType=strdup(yytext);return LONG;}

"char" {yylval.dataType=strdup(yytext);return CHAR;}

"character" {yylval.dataType=strdup(yytext);return CHAR;}

"double" {yylval.dataType=strdup(yytext);return DOUBLE;}

"assign" return INIT;

"initialize" return INIT;

"to" return TO;

"if" return IF;

"If" return IF;

"endwhile" return ENDWHILE;

"endif" return ENDIF;

"else" return ELSE;

"ELSE" return ELSE;

"then" return THEN;

"while" return WHILE;

"for" return FOR;

"do" return DO;

"print" return PRINT;

"Print" return PRINT;

"read" return READ;

"Read" return READ;

"," return COMMA;

"<=" {yylval.IdName=strdup(yytext); return LESS\_E; }

">=" {yylval.IdName=strdup(yytext); return GREAT\_E;}

"!=" {yylval.IdName=strdup(yytext); return NOT\_EQUAL;}

"==" {yylval.IdName=strdup(yytext); return EQUAL; }

"<" {yylval.IdName=strdup(yytext); return LESS;}

">" {yylval.IdName=strdup(yytext); return GREAT;}

"(" {return OPEN\_BRACKET;}

")" {return CLOSE\_BRACKET;}

"+" {yylval.IdName=strdup(yytext); return PLUS;}

"-" {yylval.IdName=strdup(yytext); return MINUS;}

"=" {yylval.IdName=strdup(yytext); return ASSIGN; }

"\*" {yylval.IdName=strdup(yytext); return MUL;}

"/" {yylval.IdName=strdup(yytext); return DIV;}

"||" {yylval.IdName=strdup(yytext); return OR;}

"&&" {yylval.IdName=strdup(yytext); return AND;}

\n {return NEWLINE;}

"start\_procedure" return START\_PROCEDURE;

"end\_procedure with return type" return END\_FUNCTION;

"return" return RETURN;

"break" return BREAK;

[0-9]+ {yylval.IdName=strdup(yytext);return NUM;}

[+-]?(([1-9]digit\*)|(0))([.,]digit+)? {yylval.IdName=strdup(yytext);return FLOATING\_NUMBER;}

{letter}({letter}|{digit})\*/[(] { yylval.IdName=strdup(yytext);return NAME\_PROCEDURE;}

{letter}({letter}|{digit})\* {yylval.IdName=strdup(yytext); return ID;}

\t ;

. ;

%%

**CC\_INNOV\_YACC.y file**

%{

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include <stdarg.h>

int yylex();

void yyerror(const char \*s);

extern char yytext[];

#define YYDEBUG\_LEXER\_TEXT yytext

char\* code\_concatenate(int arg\_count,...);

char\* gen\_addr(char\* string);

char\* get\_specifier(char \*type);

int t=0;

extern int yylineno;

%}

%token BEG END INIT TO IF FOR THEN WHILE DO PRINT READ ASSIGN FLOATING\_NUMBER BREAK RETURN END\_FUNCTION START\_PROCEDURE CLOSE\_BRACKET OPEN\_BRACKET EXP DIV MUL MINUS PLUS NOT ELSE ENDIF COMMA NEWLINE ENDWHILE

%union {char\* dataType; char\* IdName; struct attributes{

char\* code;

char\* addr;

char\* specifier;

}A;}

%token <dataType> INT FLOAT CHAR DOUBLE LONG

%token <IdName> ID NUM LESS GREAT LESS\_E GREAT\_E NOT\_EQUAL EQUAL OR AND NAME\_PROCEDURE

%type <dataType> datatype

%type <IdName> relOp logOp

%type<A> F T E assign\_stmt initialize\_stmt Stmt codes program Expr RelExpr LogExpr parameter\_list function\_call function read\_stmt print\_stmt

%type<A> datatypelist namelist

//%left ASSIGN

//%left GREAT\_E GREAT LESS\_E NOT LESS EQUAL NOT\_EQUAL

//%left PLUS MINUS

//%left MUL DIVISON

%%

program: BEG codes END NEWLINE {$$=$2;printf("\nC code: \n%s",$$.code); YYACCEPT;}

;

codes:Stmt {$$.code=$1.code;}

| codes Stmt {$$.code=code\_concatenate(2,$1.code,$2.code);}

;

Stmt:Expr {$$=$1;}

| RETURN Expr {$$.code=code\_concatenate(3,"return b",$2.code,";\n");}

| assign\_stmt {$$.code=$1.code;}

| initialize\_stmt {$$.code=$1.code;}

| read\_stmt {$$.code=$1.code;}

| print\_stmt {$$.code=$1.code;}

| BREAK {$$.code=code\_concatenate(1,"break;\n");}

| IF Expr THEN Stmt ENDIF {$$.code=code\_concatenate(5,"\nif(",$2.code,"){\n ",$4.code,"\n}");}

| IF Expr THEN Stmt ELSE Stmt ENDIF {$$.code=code\_concatenate(7,"\nif(",$2.code,"){\n ",$4.code,"\n}else{\n ",$6.code,"\n}");}

| WHILE Expr THEN Stmt ENDWHILE {$$.code=code\_concatenate(5,"\nwhile(",$2.code,"){\n ",$4.code," \n}\n");}

| DO Stmt WHILE Expr {$$.code=code\_concatenate(5,"\ndo\n{\n ",$2.code,"\n}while(",$4.code,");\n");}

| FOR assign\_stmt Expr assign\_stmt Stmt {$$.code=code\_concatenate(8,"\nfor(", $2.code, $3.code, "; ", $4.code, ")\n{\n ", $5.code, "\n}\n");}

| function {$$=$1;}

| function\_call {$$=$1;}

;

function\_call: NAME\_PROCEDURE OPEN\_BRACKET parameter\_list CLOSE\_BRACKET { $$.code=code\_concatenate(5,"\n",$1,"(",$3.code,");\n");}

;

Expr:RelExpr {$$=$1;}

| LogExpr {$$=$1;}

| E {$$=$1;}

;

print\_stmt: PRINT datatypelist COMMA Expr {$$.code=code\_concatenate(5,"printf(\" ",$2.code,"\",",$4.code,");\n");}

| PRINT datatypelist COMMA ID {$$.code=code\_concatenate(5,"printf(\" ",$2.code,"\" ,",$4,");\n");}

;

read\_stmt: READ datatypelist COMMA namelist {$$.code=code\_concatenate(5,"scanf(\"",$2.code,"\",",$4.code,");\n");}

;

RelExpr:E relOp E {$$.code = code\_concatenate(1,code\_concatenate(3,$1.addr,$2,$3.addr,";"));}

;

LogExpr:E logOp E {$$.code = code\_concatenate(1,code\_concatenate(3,$1.addr,$2,$3.addr,";"));}

;

assign\_stmt: INIT ID TO E { $$.addr=gen\_addr($2);

$$.code = code\_concatenate(3,$4.code,code\_concatenate(3,$$.addr," = ",$4.addr),";");}

| ID ASSIGN E { $$.addr = gen\_addr($1);

$$.code = code\_concatenate(3,$3.code,code\_concatenate(3,$$.addr," = ",$3.addr),";");}

;

initialize\_stmt: INIT OPEN\_BRACKET ID datatype CLOSE\_BRACKET {$$.code=code\_concatenate(2,code\_concatenate(3,$4," ",$3),";\n");}

;

function: START\_PROCEDURE NAME\_PROCEDURE OPEN\_BRACKET parameter\_list CLOSE\_BRACKET Stmt END\_FUNCTION datatype { $$.code = code\_concatenate(8,$8," ",$2,"(",$4.code,")\n{\n",$6.code,"\n}\n");}

;

parameter\_list: ID datatype {$$.code=code\_concatenate(3,$2," ",$1);}

| parameter\_list COMMA ID datatype {$$.code=code\_concatenate(5,$1.code,",",$4," ",$3);}

;

E: E PLUS T {$$.addr = code\_concatenate(1,code\_concatenate(3,$1.addr," + ",$3.addr,";")); }

| E MINUS T {$$.addr = code\_concatenate(1,code\_concatenate(3,$1.addr," - ",$3.addr,";")); }

| T {$$.addr = $1.addr;}

;

T: T MUL F {$$.addr = code\_concatenate(1,code\_concatenate(3,$1.addr," \* ",$3.addr,";")); }

| T DIV F {$$.addr = code\_concatenate(1,code\_concatenate(3,$1.addr," / ",$3.addr,";")); }

| F {$$.addr = $1.addr;}

;

F: ID {$$.addr = gen\_addr($1); $$.code = code\_concatenate(1," ");}

| NUM {$$.addr = gen\_addr($1); $$.code = code\_concatenate(1," ");;}

| OPEN\_BRACKET E CLOSE\_BRACKET {$$=$2;}

;

relOp:LESS\_E {$$=$1;}

|GREAT\_E {$$=$1;}

|NOT\_EQUAL {$$=$1;}

|EQUAL {$$=$1;}

|LESS {$$=$1;}

|GREAT {$$=$1;}

;

logOp:AND {$$=$1;}

|OR {$$=$1;}

;

datatypelist: datatype { $$.specifier=get\_specifier($1);$$.code=code\_concatenate(1,$$.specifier);}

| datatypelist COMMA datatype { $$.specifier=get\_specifier($3);$$.code=code\_concatenate(3,$1.code,",",$$.specifier);}

;

namelist: ID {$$.code=code\_concatenate(1,$1);}

| namelist COMMA ID {$$.code=code\_concatenate(3,$1.code,",",$3);}

;

datatype: CHAR {$$ = $1;}

| INT {$$ = $1;}

| DOUBLE {$$ = $1;}

| FLOAT {$$ = $1;}

| LONG {$$ = $1;}

;

%%

int yywrap()

{

return 1;

}

void yyerror(const char\* arg)

{

printf("%s\n",arg);

}

char\* gen\_addr(char\* string)

{

char\* addr = (char\*)malloc(sizeof(string));

strcpy(addr, string);

return addr;

}

char\* code\_concatenate(int arg\_count,...)

{

int i;

va\_list ap;

va\_start(ap, arg\_count);

char\* temp = malloc(1000);

for (i = 1; i <= arg\_count; i++)

{

char\* a = va\_arg(ap,char\*);

temp = (char\*)realloc(temp,(strlen(temp)+strlen(a)+10));

strcat(temp,a);

}

return temp;

}

char\* get\_specifier(char \*type){

char\* data;

if(strcmp(type,"int")==0|| strcmp(type,"integer")==0)

data="%d";

else if(strcmp(type,"float")==0)

data="%f";

else if(strcmp(type,"char")==0 || strcmp(type,"character")==0)

data="%c";

else if (strcmp(type,"double")==0)

data="%f";

else if(strcmp(type,"long")==0)

data="%ld";

return data;

}

int main()

{

extern FILE \*yyin;

yyin=fopen("input.txt","r");

if(!yyparse())

{

printf("\n");

}

fclose(yyin);

return 0;

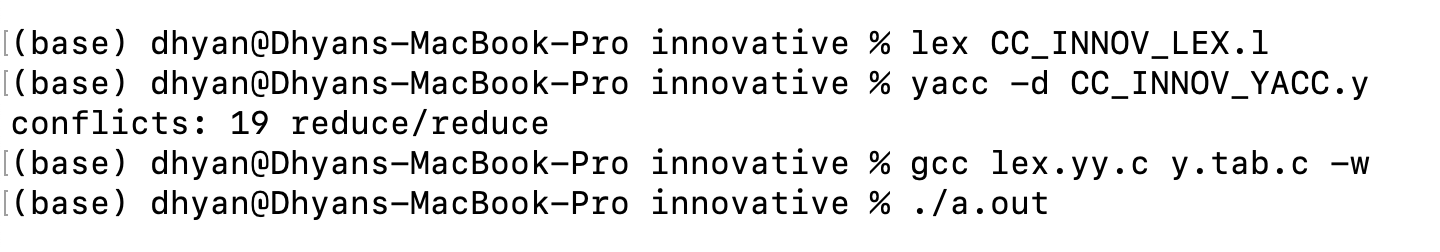
}

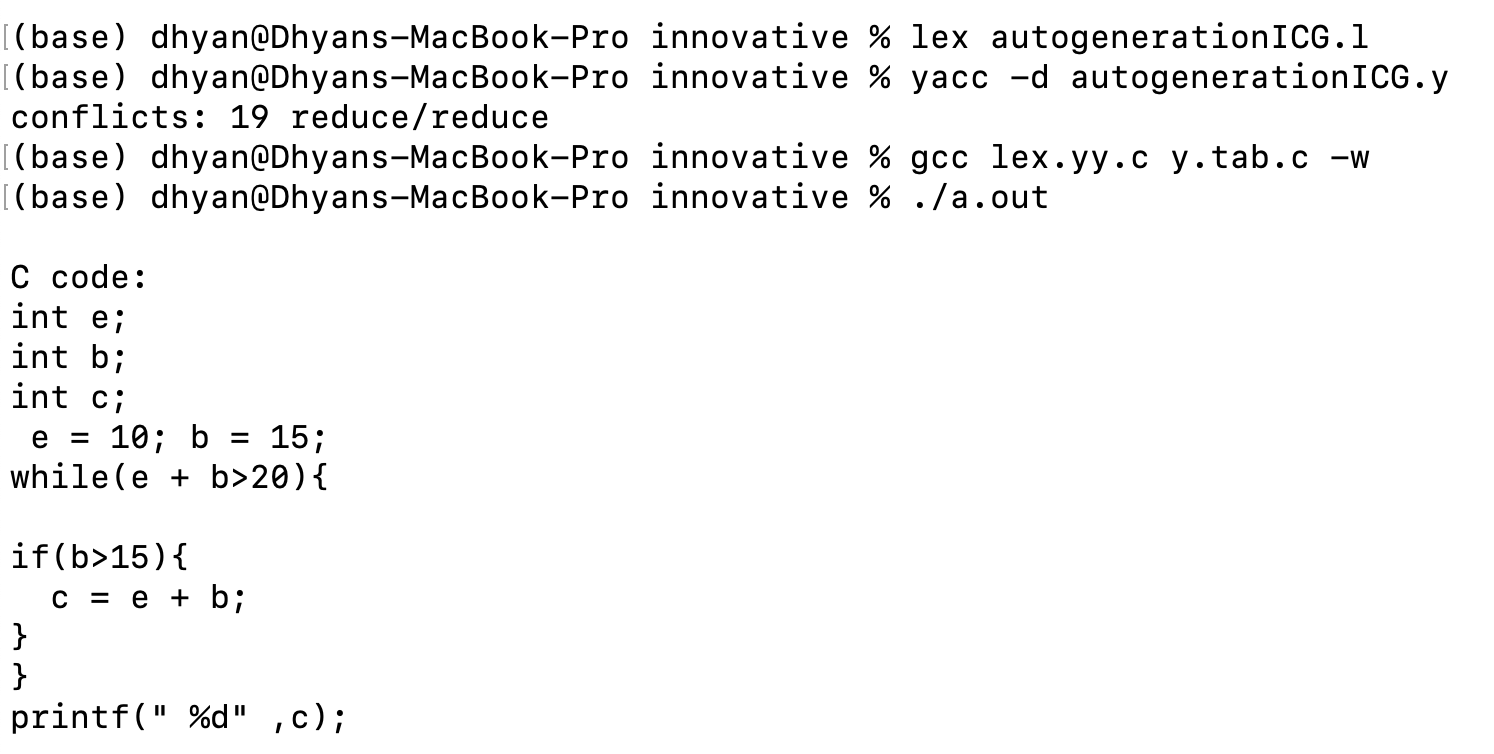
**Results:**

**Input1.txt**

begin assign ( e int ) assign ( b int ) assign ( c int ) assign e to 10 assign b to 15 while e+b > 20 then if b > 15 then c = e + b endif endwhile print int , c end

**Output1:**

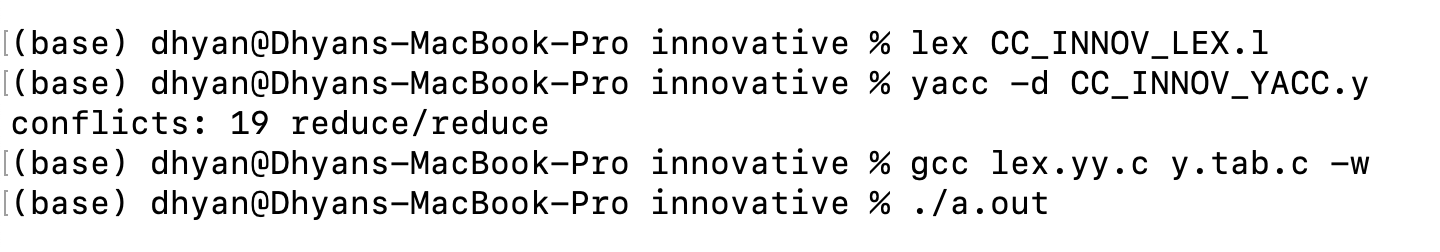
****

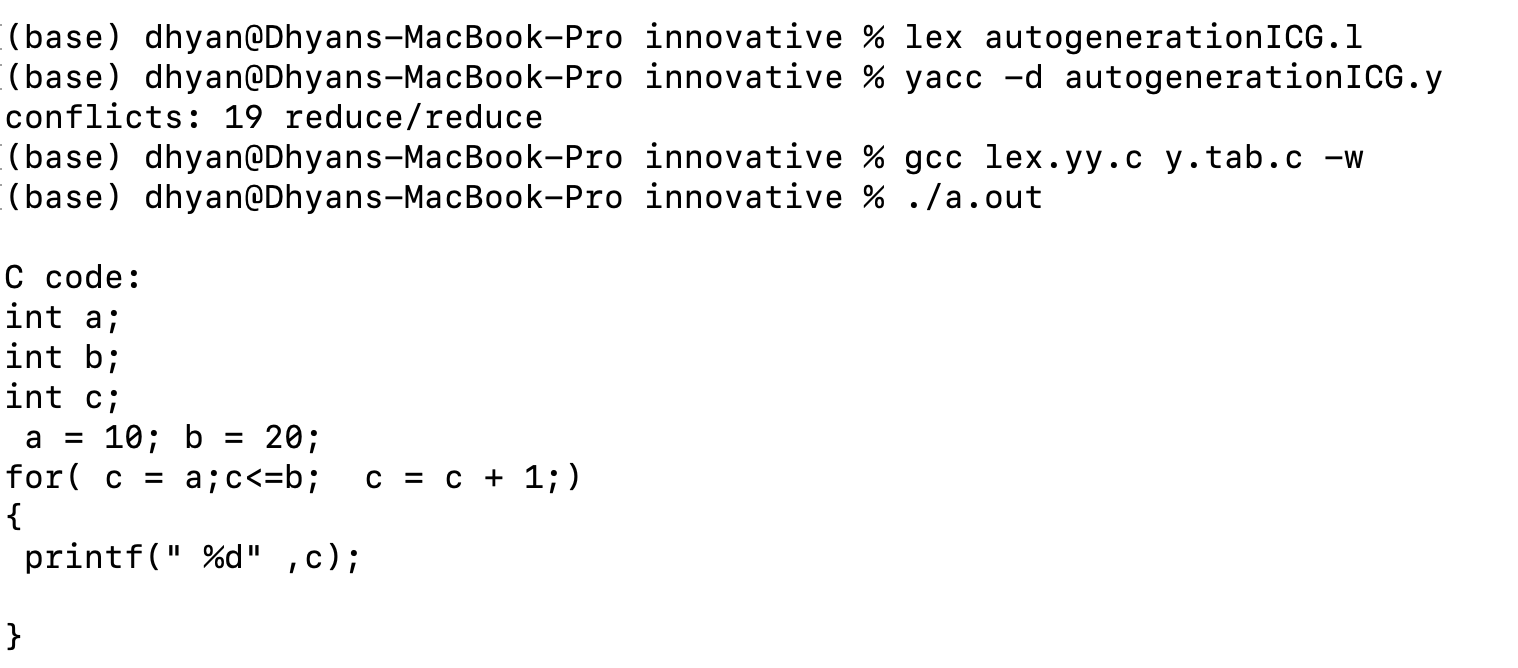
****

**Input2.txt**

begin assign ( a int ) assign ( b int ) assign ( c int ) assign a to 10 assign b to 20 for assign c to a c <= b c = c+1 print int , c end

**Output2:**

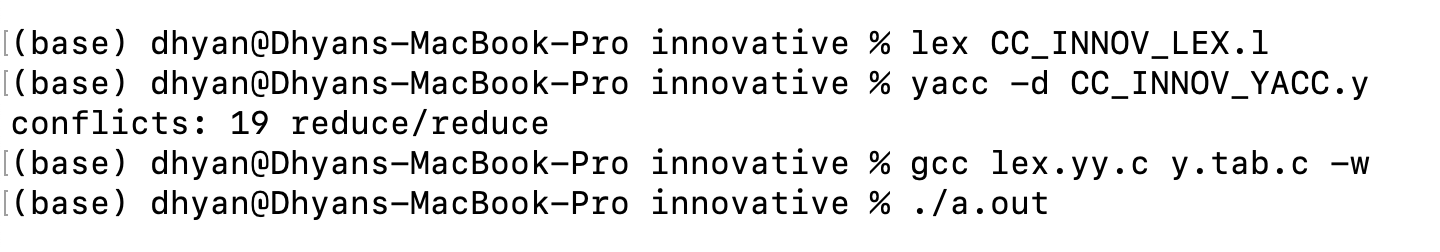
****

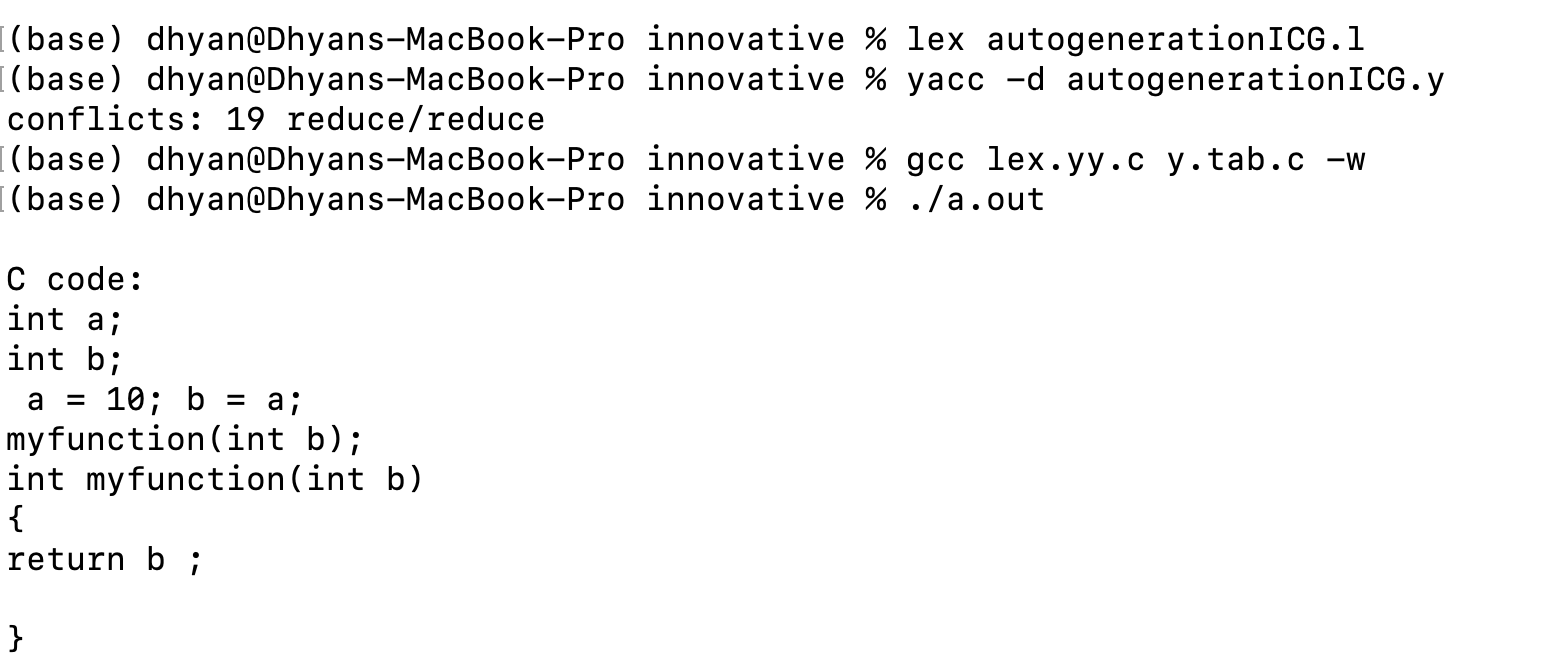
****

**Input3.txt**

begin assign ( a int ) assign ( b int ) assign a to 10 assign b to a myfunction(b int) start\_procedure myfunction(b int) return b end\_procedure with return type int end

**Output3:**

****

****

**Applications:**

Certainly! Here are some potential uses or applications of a pseudocode to code generator implemented using Yacc and Lex:

**1. Teaching and Learning:**

· The tool can be employed as an educational resource to assist students in understanding the translation process from pseudocode to actual code. It provides a hands-on experience in parsing and generating code based on algorithmic descriptions.

**2. Rapid Prototyping:**

· Developers can use the tool for rapid prototyping of algorithms. By expressing the logic in pseudocode, they can quickly generate C code, allowing for faster testing and validation of algorithmic ideas before implementing them in a more formal coding environment.

**3. Algorithm Documentation:**

· The generator can be utilized to convert algorithmic pseudocode into executable code snippets, aiding in the documentation of algorithms. This can be particularly useful in research papers, technical documentation, and instructional materials.

**4. Code Snippet Generation:**

· Developers can use the tool to convert algorithmic ideas expressed in pseudocode into functional code snippets. This can be handy for incorporating specific algorithms into larger codebases or frameworks.

**5. Cross-Language Translation:**

· With modifications, the generator can be extended to support the translation of pseudocode into languages other than C. This versatility can be valuable for developers working with multiple programming languages.

**6. Code Review and Collaboration:**

· The tool can facilitate collaboration by allowing team members to express algorithmic logic in a standardized pseudocode format. The generator ensures consistency in the translation process, making it easier for team members to review and discuss algorithmic implementations.

**Conclusion:**

In summary, a pseudocode to code generator implemented with Yacc and Lex provides a versatile tool with applications ranging from education to rapid prototyping and documentation. Its flexibility allows developers to streamline various aspects of the software development lifecycle.The pseudocode to C code generator crafted using Lex and Yacc exemplifies the power of compiler tools. It tokenizes high-level constructs through Lex and structures them into C code via Yacc's grammatical rules. This automation bridges the gap between algorithm design and executable code, providing a seamless translation from pseudocode, enhancing coding efficiency, and serving as an educational insight into compiler design. It underscores the sophistication of code generation technology and its pivotal role in software development, streamlining the transition from concept to functional software.