1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. по дисциплине «Формальные грамматики и теории компиляторов»
2. Выполнил
3. студент гр. 4851003/80802 Сошнев М.Д.

<*подпись*>

1. Преподаватель Мясников А.В.

<*подпись*>

1. Санкт-Петербург
2. 2021

Оглавление

[1 Цель работы 3](#__RefHeading___Toc4299_3652333103)

[2 Задача 3](#__RefHeading___Toc4301_3652333103)

[3 Ход работы 3](#__RefHeading___Toc4303_3652333103)

[Сложные арифметические выражения 4](#__RefHeading___Toc6825_3652333103)

[Хранение в регистрах 6](#__RefHeading___Toc6827_3652333103)

[Условия 8](#__RefHeading___Toc6829_3652333103)

[Циклы 9](#__RefHeading___Toc6831_3652333103)

[Инкремент и декримент 9](#__RefHeading___Toc6833_3652333103)

[4 Тестирование 11](#__RefHeading___Toc4305_3652333103)

[5 Вывод 14](#__RefHeading___Toc4307_3652333103)

[6 Приложение 15](#__RefHeading___Toc4309_3652333103)

[Makefile 15](#__RefHeading___Toc6837_3652333103)

[lexic.l 15](#__RefHeading___Toc6839_3652333103)

[grammar.y 18](#__RefHeading___Toc6841_3652333103)

[Vars.h 30](#__RefHeading___Toc6843_3652333103)

[SolverStack.h 31](#__RefHeading___Toc6845_3652333103)

[RegisterAllocator.h 40](#__RefHeading___Toc6847_3652333103)

# **Цель работы**

Понять принцип формирования бинарных файлов на практике.

# Задача

На основе предыдущей лабораторной работы исправить разбор выражений так, чтобы парсер при разборе на лексемы заменял их на ассемблерные команды и сохранял в файл.

# Ход работы

За основу был взят язык ассемблера с регистровой архитектурой, без стека. Для хранения данных имеется 32 регистра — R1 — R32 и куча (переменные).

Команды:

|  |  |
| --- | --- |
| Формат команды | Описание |
| MOV x, y | Загрузка значения из регистра/переменной y в регистр/переменную x. |
| LDI R, c | Загрузка числовой константы c в регистр R. |
| CALL func | Переход на функцию func, параметры лежат последовательно в R1, R2, … |
| JMP m | Безусловный переход на метку m |
| JE m | Переход, если равно (Jump Equal) |
| JNE m | Переход, если не равно (Jump Not Equal) |
| JG m | Переход, если больше (Jump Greater) |
| JGE m | Переход, если больше либо равно (Jump Greater Equal) |
| JL m | Переход, если меньше (Jump Less) |
| JLE m | Переход, если меньше либо равно (Jump Less Equal) |
| INC R | Инкремент регистра R |
| DEC R | Декримент регистра R |
| ADD R, y | Прибавить y к регистру R |
| SUB R, y | Отнять y от регистра R |
| MUL R, y | Умножить на y регистр R |
| DIV R, y | Разделить на y регистр R |
| CMP R, y | Сравнить регистр R с y и установить флаги |
| AND R, y | Побитовая дизъюнкция разрядов |
| OR R, y | Побитовая конъюнкция разрядов |
| XOR R, y | Побитовая строгая дизъюнкция разрядов |
| LS R, y | Сдвиг разрядов регистра R влево на y разрядов (Left Shift) |
| LR R, y | Сдвиг разрядов регистра R вправо на y разрядов (Right Shift) |
| MOD R, y | Взять остаток от деления R на y и положить в R |
| RET R | Возврат значения из R |

Задача — во время парсинга текстов формировать псевдоассемблерный файл, используя инструкции из даной таблицы.

## Сложные арифметические выражения

Первая проблема это сложное арифметическое выражение, записанное в одну строчку до ; где каждый оператор имеет свой приоритет и имеются скобки. Решение — формировать стек из операторов и операндов в постфиксной форме, где сначала идут два оператора, а затем операнд.

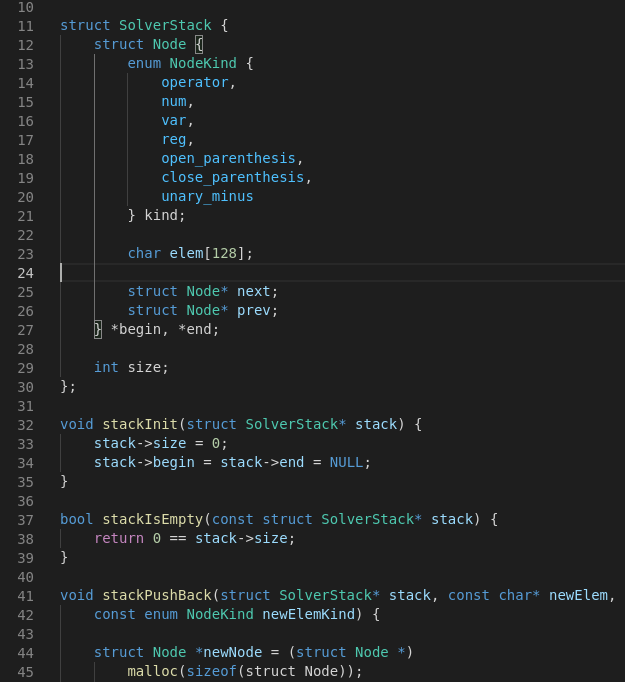
Например, выражение:

S1 = S2 = S3 = 6\*t + 7\*u — 8\*(y-i) / u;

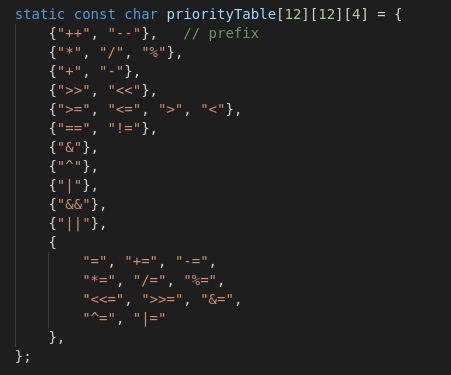
Будет записано, как:

S1 S2 S3 6 t \* 7 u \* + 8 y i - \* u / - = = =

Была написана структура стека на чистом Си с разными методами:

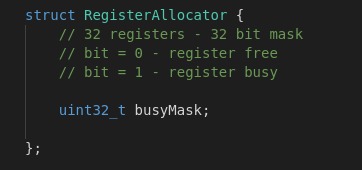
*Рисунок 1 — структура стек*

В таком случае каждая операция последовательно переписывается на язык ассемлера в порядке своего приоритета. Приоритет операторов определяется специальным методом, где сформирован массив операторов, упорядоченный по приоритету:

*Рисунок 2 — массив операторов, упорядоченный по приоритету*

## Хранение в регистрах

В момент подсчёта сложного выражения необходимы регистры для хранения промежуточного значения. Например, в выражении 6\*t — 8\*y в момент вычисления 8\*y необходимо в каком, то регистре хранить вычисленное 6\*t. Для этого была написана структура RegisterAllocator:

*Рисунок 3 - RegisterAllocator*

Были написаны методы regAllocatorAlloc() и regAllocatorFree(). Метод regAllocatorAlloc() ищет в маске аллокатора бит равный нулю, устанавливает его в 1 (помечает регистр как занятый) и возвращает номер этого бита (номер для регистра). Метод regAllocatorFree() принимает на вход номер регистра и помечает соответствующий бит маски аллокатора как 0 (освобождает регистр).

*Рисунок 4 — освобождение и выделение регистров*

Таким образом в любой момент времени занятые регистры помечены в аллокаторе разрядом 1 и выделение очередного регистра произойдет быстро и не нарушив необходимые данные, лежащие в занятых регистрах. Ситуация длинного выражения, когда 32 регистров не хватит для хранения временных значений не рассмотрена.

## Условия

Реализация условий требует внесения специальных меток в псевдоассемблерный код, по которым возможен переход. Метки для условий имеют следующий формат — каждый кейз помечается «case\_[x]\_[y]:» где x — номер кейза в текущем условии, y — номер условия в коде. Конец каждого условия помечается метой «out\_[y]:»

Например, следующий код

*if (…) {*

*code 1*

*} else if (…) {*

*code 2*

*} else {*

*code 3*

*}*

*if (…) {*

*code 4*

*} else {*

*code 5*

*}*

Будет переписан в:

case\_1\_1:

asm code 1

case\_2\_1:

asm code 2

case\_3\_1:

asm code 3

out\_1:

case\_1\_2:

asm code 4

case 2\_2:

asm code 5

out\_2:

Таким образом можно легким способом осуществлять прыжки по меткам.

## Циклы

Реализация циклов не сильно отличается от реализации условий. Тоже были введены метки, следующего формата: вход в цикл помечается меткой «cycle\_[x]\_in:» где x — порядковый номер цикла в коде, а выход - «cycle\_[x]\_out:».

Таким обазом, код

while (a > 0) {

code 1

}

code 2

Будет переписан в

cycle\_1\_in:

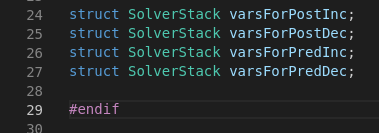
asm code 1

cycle\_1\_out:

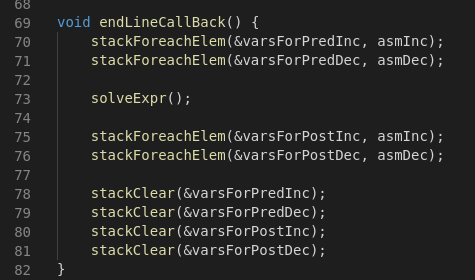
asm code 2

## Инкремент и декримент

Имеются 4 стека:

*Рисунок 5 — стеки для переменных для инкремента/декримента*

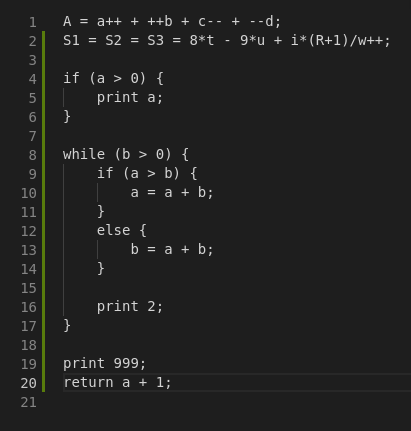
В которые записываются переменные которые необходимо инкрементировать или декриментировать префиксно или постфиксно. Префиксные стеки транслируют ассемблерный код до вычисления общего выражения, постфиксные — после.

*Рисунок 6 — callBack -функция*

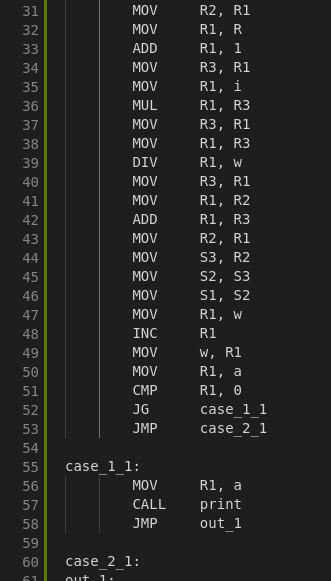
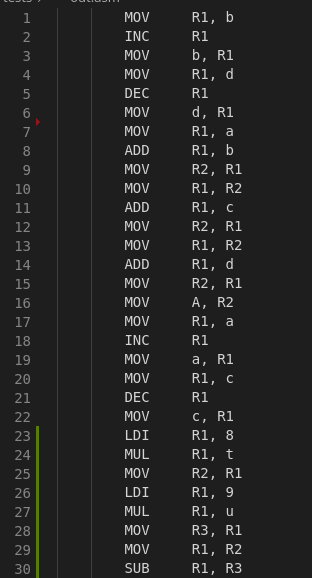
Данная колбек-функция вызывется, когда программа обнаруживает символ ;. Функция сначала выполняет префиксный инкремент и декримент, далее вычисляет выражение, которое уже записано в стеке в постфиксной форме, затем вычисляет постфиксные инкремент и декримент.

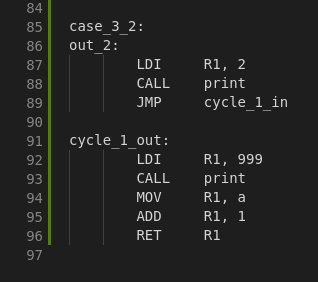
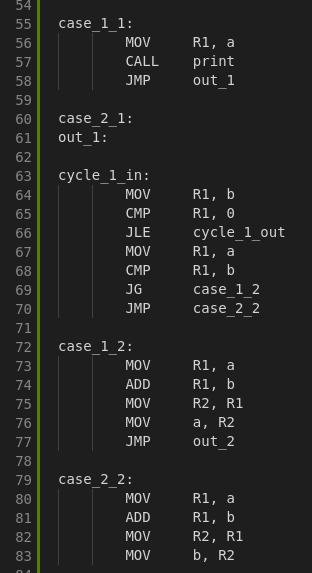
# Тестирование

Напишем тестовую программу:

*Рисунок 7 — тестовая программа*

Программа написана таким образом, что имеет инкременты и декрименты (line 1), сложное выражение (line 2), а также вложенность циклов и условий. В результате трансляции написанным компилятором был получен псевдоассемблерный код:

 *Рисунки 8 — 9 полученный ассемблерный код*

*Рисунки 10 - 11 — полученный ассемблерный код*

# Вывод

В ходе выполнения данной лабораторной работы был усовершенствован компилятор, теперь он умеет транслировать псевдокод имеющий циклы, условия и сложные арифметические выражения в псевдоассемблерный код.

# Приложение

## Makefile

all:

lex src/lexic.l

yacc -d src/grammar.y

mv lex.yy.c tmp/

mv y.tab.c tmp/

mv y.tab.h tmp/

cp src/SolverStack.h tmp/

cp src/RegisterAllocator.h tmp/

cp src/Vars.h tmp/

cc tmp/lex.yy.c tmp/y.tab.c -o compile -lfl

## lexic.l

%{

#include <stdlib.h>

#include "y.tab.h"

extern char lastVarName[128];

extern char lastNumber[128];

%}

%%

\{ return OBRACE;

\} return EBRACE;

\( return OPEN;

\) return CLOSE;

print return PRINT;

return return RETURN;

if return IF;

else\ if return ELSE\_IF;

else return ELSE;

while return WHILE;

= return ASSIGN;

; return SEMICOLON;

[ \t]+ /\* игнорируем пробелы и знаки табуляции \*/

(\r\n)|\n yylval++;

> return IS\_MORE;

\< return IS\_LESS;

>= return IS\_MEQUAL;

\<= return IS\_LEQUAL;

== return IS\_EQUAL;

!= return IS\_NOT\_EQUAL;

(\+=) return AASS;

(-=) return SASS;

(\\*=) return MASS;

(\/=) return DASS;

\+\+ return INC;

\-\- return DEC;

\+ return ADD;

\- return SUB;

\\* return MUL;

\/ return DIV;

&& return AND;

\|\| return OR;

& return BIT\_OR;

\| return BIT\_AND;

\^ return BIT\_XOR;

\<\< return BIT\_LEFT\_SHIFT;

>> return BIT\_RIGHT\_SHIFT;

\% return MOD;

[0-9]\* {

strcpy(lastNumber, yytext);

return NUMBER;

}

[a-zA-Z][a-zA-Z0-9]\* {

strcpy(lastVarName, yytext);

return VAR;

}

%%

## grammar.y

%{

#include <stdio.h>

#include <string.h>

#include "SolverStack.h"

#include "Vars.h"

void yyerror(char \*s) ;

int yylex(); // ????

void tmpStackPrint() {

fprintf(fout, "[stack: ");

for (struct Node\* i = stack.begin; i != NULL; i = i->next) {

fprintf(fout, "%s ", i->elem);

}

fprintf(fout, "]\n");

}

void solveExpr() {

struct Node n;

while (!stackIsEmpty(&tmp)) {

n = stackPopBack(&tmp);

stackPushBack(&stack, n.elem, n.kind);

}

// tmpStackPrint();

stackRun(&stack, fout);

stackClear(&stack);

}

char\* getJmpTypeCommand(const char\* compareOperator) {

if (!strcmp("==", compareOperator))

return "JE";

if (!strcmp("!=", compareOperator))

return "JNE";

if (!strcmp(">", compareOperator))

return "JG";

if (!strcmp("<", compareOperator))

return "JL";

if (!strcmp(">=", compareOperator))

return "JGE";

if (!strcmp("<=", compareOperator))

return "JLE";

return "???";

}

char\* getOppositeJmpTypeCommand(const char\* jmpType) {

if (!strcmp("JE", jmpType))

return "JNE";

if (!strcmp("JNE", jmpType))

return "JE";

if (!strcmp("JG", jmpType))

return "JLE";

if (!strcmp("JL", jmpType))

return "JGE";

if (!strcmp("JLE", jmpType))

return "JG";

if (!strcmp("JGE", jmpType))

return "JL";

return "???";

}

void endLineCallBack() {

stackForeachElem(&varsForPredInc, asmInc);

stackForeachElem(&varsForPredDec, asmDec);

solveExpr();

stackForeachElem(&varsForPostInc, asmInc);

stackForeachElem(&varsForPostDec, asmDec);

stackClear(&varsForPredInc);

stackClear(&varsForPredDec);

stackClear(&varsForPostInc);

stackClear(&varsForPostDec);

}

%}

%start commands

%token OPEN CLOSE OBRACE EBRACE

%token NUMBER VAR

%token SEMICOLON

%token IF ELSE\_IF ELSE WHILE

%token PRINT RETURN

%token ADD SUB MUL DIV AND OR

%token DEC INC

%token BIT\_AND BIT\_OR BIT\_XOR

%token BIT\_LEFT\_SHIFT BIT\_RIGHT\_SHIFT MOD

%token ASSIGN

%token IS\_EQUAL IS\_NOT\_EQUAL

%token IS\_LESS IS\_MORE IS\_LEQUAL IS\_MEQUAL

%token AASS SASS MASS DASS

%%

commands:

/\* empty \*/ |

commands command;

semicolon:

SEMICOLON semicolon |

SEMICOLON {

endLineCallBack();

};

command:

PRINT expr semicolon {

fprintf(fout, "\t\tCALL\tprint\n");

} |

RETURN expr semicolon {

fprintf(fout, "\t\tRET\t\tR1\n");

} |

expr semicolon |

condition |

cycle\_while;

body:

OBRACE commands EBRACE;

condition:

IF OPEN expr CLOSE {

++condsNum;

solveExpr();

char\* jmpTypeCmd = getJmpTypeCommand(lastBinCompareOperator);

fprintf(fout, "\t\t%s\t\tcase\_1\_%d\n", jmpTypeCmd, condsNum);

fprintf(fout, "\t\tJMP\t\tcase\_2\_%d\n", condsNum);

curElseCasesNum = 0;

fprintf(fout, "\ncase\_%d\_%d:\n", 1, condsNum);

}

body {

fprintf(fout, "\t\tJMP\t\tout\_%d\n", condsNum);

}

else\_case {

fprintf(fout, "\ncase\_%d\_%d:\n", curElseCasesNum + 2, condsNum);

fprintf(fout, "out\_%d:\n", condsNum);

};

else\_case:

/\* empty \*/ |

ELSE\_IF OPEN {

++curElseCasesNum;

fprintf(fout, "\ncase\_%d\_%d:\n", curElseCasesNum + 1, condsNum);

}

expr CLOSE {

solveExpr();

char\* jmpTypeCmd = getJmpTypeCommand(lastBinCompareOperator);

jmpTypeCmd = getOppositeJmpTypeCommand(jmpTypeCmd);

fprintf(fout, "\t\t%s\t\tcase\_%d\_%d\n",

jmpTypeCmd, curElseCasesNum + 2, condsNum);

}

body {

fprintf(fout, "\t\tJMP\t\tout\_%d\n", condsNum);

}

else\_case |

ELSE {

++curElseCasesNum;

fprintf(fout, "\ncase\_%d\_%d:\n",

curElseCasesNum + 1, condsNum);

}

body;

cycle\_while:

WHILE OPEN expr CLOSE {

++cyclesNum;

fprintf(fout, "\ncycle\_%d\_in:\n", cyclesNum);

solveExpr();

char\* jmpTypeCmd = getJmpTypeCommand(lastBinCompareOperator);

jmpTypeCmd = getOppositeJmpTypeCommand(jmpTypeCmd);

fprintf(fout, "\t\t%s\t\tcycle\_%d\_out\n",

jmpTypeCmd, cyclesNum);

}

body {

fprintf(fout, "\t\tJMP\t\tcycle\_%d\_in\n", cyclesNum);

fprintf(fout, "\ncycle\_%d\_out:\n", cyclesNum);

};

var\_or\_number:

VAR {

lastExprKind = var;

stackPushBack(&stack, lastVarName, var);

} |

NUMBER {

lastExprKind = num;

stackPushBack(&stack, lastNumber, num);

};

expr:

var\_or\_number |

unary\_operation |

OPEN {

stackPushBack(&tmp, "(", open\_parenthesis);

}

expr

CLOSE {

struct Node n;

while (true) {

n = stackPopBack(&tmp);

if (n.kind == open\_parenthesis) {

break;

}

stackPushBack(&stack, n.elem, n.kind);

}

} |

expr

binary\_operator {

strcpy(lastBinOperator, yytext);

struct Node n;

while (!stackIsEmpty(&tmp)) {

bool compare;

if (!strcmp("=", lastBinOperator)) {

compare = (

getOperatorPriority(lastBinOperator) <

getOperatorPriority(tmp.end->elem)

);

} else {

compare = (

getOperatorPriority(lastBinOperator) <=

getOperatorPriority(tmp.end->elem)

);

}

if (!compare) {

break;

}

n = stackPopBack(&tmp);

stackPushBack(&stack, n.elem, n.kind);

}

stackPushBack(&tmp, lastBinOperator, operator);

}

expr;

unary\_operation:

INC VAR {

printf("++%s\n", lastVarName);

stackPushBack(&varsForPredInc, lastVarName, var);

lastExprKind = var;

stackPushBack(&stack, lastVarName, var);

} |

VAR INC {

printf("%s++\n", lastVarName);

stackPushBack(&varsForPostInc, lastVarName, var);

lastExprKind = var;

stackPushBack(&stack, lastVarName, var);

} |

DEC VAR {

printf("--%s\n", lastVarName);

stackPushBack(&varsForPredDec, lastVarName, var);

lastExprKind = var;

stackPushBack(&stack, lastVarName, var);

} |

VAR DEC {

printf("%s--\n", lastVarName);

stackPushBack(&varsForPostDec, lastVarName, var);

lastExprKind = var;

stackPushBack(&stack, lastVarName, var);

} |

ADD expr |

SUB expr;

binary\_operator:

ASSIGN {

if (lastExprKind == num) {

yyerror("syntax error");

exit(1);

}

strcpy(lastBinOperator, "=");

} |

ADD | SUB | MUL | DIV |

compare\_operator {

strcpy(lastBinCompareOperator, yytext);

} |

AASS | SASS | MASS | DASS |

AND | OR | MOD |

BIT\_AND | BIT\_OR | BIT\_XOR |

BIT\_RIGHT\_SHIFT | BIT\_LEFT\_SHIFT;

compare\_operator:

IS\_EQUAL | IS\_NOT\_EQUAL |

IS\_MORE | IS\_LESS |

IS\_MEQUAL | IS\_LEQUAL;

%%

void yyerror(char \*s)

{

++numError;

fprintf(stderr, "%s\n", s);

}

int yywrap()

{

if (0 == numError)

printf("OK\n");

return 1;

}

int main(int argc, void \*argv[])

{

yylval = 0;

if (4 != argc || strcmp("-o", argv[2]) != 0) {

printf("Incorrect arguments\n");

printf("Usage: ./compile [path src] -o [path asm]\n");

return -1;

}

char\* pathFile = argv[1];

yyin = fopen(pathFile,"r");

char\* pathAsmFile = argv[3];

fout = fopen(pathAsmFile, "w");

if (NULL == yyin) {

printf("No such file: %s\n", pathFile);

return -1;

}

stackInit(&stack);

stackInit(&tmp);

stackInit(&varsForPostInc);

stackInit(&varsForPostDec);

stackInit(&varsForPredInc);

stackInit(&varsForPredDec);

yyparse();

fclose(yyin);

fclose(fout);

return 0;

}

## Vars.h

#ifndef VARS\_H

#define VARS\_H

extern char\* yytext;

extern FILE\* yyin;

FILE\* fout;

char lastVarName[128];

char lastNumber[128];

char lastBinOperator[4];

char lastBinCompareOperator[4];

enum NodeKind lastExprKind;

int condsNum = 0;

int curElseCasesNum = 0;

int cyclesNum = 0;

int numError = 0;

struct SolverStack stack;

struct SolverStack tmp;

struct SolverStack varsForPostInc;

struct SolverStack varsForPostDec;

struct SolverStack varsForPredInc;

struct SolverStack varsForPredDec;

#endif

## SolverStack.h

#ifndef MY\_STACK\_H

#define MY\_STACK\_H

#include <string.h>

#include <stdbool.h>

#include <stdlib.h>

#include "RegisterAllocator.h"

#include "Vars.h"

struct SolverStack {

struct Node {

enum NodeKind {

operator,

num,

var,

reg,

open\_parenthesis,

close\_parenthesis,

unary\_minus

} kind;

char elem[128];

struct Node\* next;

struct Node\* prev;

} \*begin, \*end;

int size;

};

void stackInit(struct SolverStack\* stack) {

stack->size = 0;

stack->begin = stack->end = NULL;

}

bool stackIsEmpty(const struct SolverStack\* stack) {

return 0 == stack->size;

}

void stackPushBack(struct SolverStack\* stack, const char\* newElem,

const enum NodeKind newElemKind) {

struct Node \*newNode = (struct Node \*)

malloc(sizeof(struct Node));

newNode->kind = newElemKind;

strcpy(newNode->elem, newElem);

newNode->next = NULL;

if (!stackIsEmpty(stack)) {

stack->end->next = newNode;

newNode->prev = stack->end;

stack->end = newNode;

} else {

stack->end = stack->begin = newNode;

newNode->prev = NULL;

}

++stack->size;

}

struct Node stackPopBack(struct SolverStack\* stack) {

if (stackIsEmpty(stack)) {

struct Node n;

return n;

}

struct Node toReturn;

strcpy(toReturn.elem, stack->end->elem);

toReturn.kind = stack->end->kind;

struct Node\* toDelete = stack->end;

if (stack->size > 1) {

struct Node\* newEnd = stack->end->prev;

newEnd->next = NULL;

stack->end = newEnd;

} else {

stack->begin = stack->end = NULL;

}

free(toDelete);

--stack->size;

return toReturn;

}

void stackClear(struct SolverStack\* stack) {

while (!stackIsEmpty(stack)) {

stackPopBack(stack);

}

}

void asmInc(char\* var) {

fprintf(fout, "\t\tMOV\t\tR1, %s\n", (var));

fprintf(fout, "\t\tINC\t\tR1\n");

fprintf(fout, "\t\tMOV\t\t%s, R1\n", (var));

}

void asmDec(char\* var) {

fprintf(fout, "\t\tMOV\t\tR1, %s\n", (var));

fprintf(fout, "\t\tDEC\t\tR1\n");

fprintf(fout, "\t\tMOV\t\t%s, R1\n", (var));

}

void stackForeachElem(const struct SolverStack\* stack, void(\*func)(char\*)) {

for (struct Node\* n = stack->begin;

n != NULL; n = n->next) {

func(n->elem);

}

}

int getOperatorPriority(const char\* operator) {

static const char priorityTable[12][12][4] = {

{"++", "--"}, // prefix

{"\*", "/", "%"},

{"+", "-"},

{">>", "<<"},

{">=", "<=", ">", "<"},

{"==", "!="},

{"&"},

{"^"},

{"|"},

{"&&"},

{"||"},

{

"=", "+=", "-=",

"\*=", "/=", "%=",

"<<=", ">>=", "&=",

"^=", "|="

},

};

for (int i = 0; i < 12; ++i) {

for (int j = 0; j < 12; ++j) {

if (!strcmp(priorityTable[i][j], operator)) {

return 12 - i;

}

}

}

return 0;

}

char\* getAsmCommand(const char\* operator) {

if (!strcmp("+", operator))

return "ADD";

if (!strcmp("-", operator))

return "SUB";

if (!strcmp("\*", operator))

return "MUL";

if (!strcmp("/", operator))

return "DIV";

if (!strcmp("==", operator) ||

!strcmp("!=", operator) ||

!strcmp(">", operator) ||

!strcmp(">=", operator) ||

!strcmp("<", operator) ||

!strcmp("<=", operator)) {

return "CMP";

}

// to continue

return "...";

}

void stackRun(const struct SolverStack\* stack, FILE\* fout) {

struct SolverStack tmp;

stackInit(&tmp);

struct RegisterAllocator regAllocator;

regAllocatorInit(&regAllocator);

int regNum = 1;

char r[16];

bool wasAssigment = false;

// itterate throw the src stack

for (const struct Node\* n = stack->begin;

n != NULL;

n = n->next) {

if (n->kind == num || n->kind == var) {

stackPushBack(&tmp, n->elem, n->kind);

}

else if (n->kind == operator) {

struct Node rightArg = stackPopBack(&tmp);

struct Node leftArg = stackPopBack(&tmp);

if (!strcmp("=", n->elem)) {

fprintf(fout, "\t\tMOV\t\t%s, %s\n",

leftArg.elem, rightArg.elem);

stackPushBack(&tmp, leftArg.elem, leftArg.kind);

wasAssigment = true;

}

else {

if (leftArg.kind == num) {

fprintf(fout, "\t\tLDI\t\tR1, %s\n", leftArg.elem);

}

else if (leftArg.kind == var || leftArg.kind == reg) {

fprintf(fout, "\t\tMOV\t\tR1, %s\n", leftArg.elem);

if (leftArg.kind == reg)

regAllocatorFree(

&regAllocator, (int)(leftArg.elem[1] - '0'));

}

if (rightArg.kind == reg)

regAllocatorFree(

&regAllocator, (int)(rightArg.elem[1] - '0'));

char\* asmCommand = getAsmCommand(n->elem);

fprintf(fout, "\t\t%s\t\tR1, %s\n",

asmCommand, rightArg.elem);

if (!stackIsEmpty(&tmp)) {

// save result to Rx

regNum = regAllocatorAlloc(&regAllocator);

if (-1 == regNum)

fprintf(fout, "!!! ERROR: ALL REGISTERS ARE BUSY !!!\n");

fprintf(fout, "\t\tMOV\t\tR%d, R1\n", regNum);

sprintf(r, "R%d", regNum);

stackPushBack(&tmp, r, reg);

}

}

}

}

if (!wasAssigment && !stackIsEmpty(&tmp)) {

if (regNum != 1)

fprintf(fout, "\t\tMOV\t\tR1, R%d\n", regNum);

else if (tmp.end->kind == num)

fprintf(fout, "\t\tLDI\t\tR1, %s\n", tmp.end->elem);

else if (tmp.end->kind == var)

fprintf(fout, "\t\tMOV\t\tR1, %s\n", tmp.end->elem);

}

}

#endif

## RegisterAllocator.h

#ifndef REGISTER\_ALLOCATOR\_H

#define REGISTER\_ALLOCATOR\_H

#include <stdint.h>

#define SET\_BIT(mask, bit) ((mask) |= (1 << (bit)))

#define RESET\_BIT(mask, bit) ((mask) &= ~(1 << (bit)))

#define GET\_BIT(mask, bit) ((mask) & (1 << (bit)))

struct RegisterAllocator {

// 32 registers - 32 bit mask

// bit = 0 - register free

// bit = 1 - register busy

uint32\_t busyMask;

};

void regAllocatorInit(struct RegisterAllocator\* registerAllocator) {

registerAllocator->busyMask = (uint32\_t)0;

SET\_BIT(registerAllocator->busyMask, 0);

}

int regAllocatorAlloc(struct RegisterAllocator\* registerAllocator) {

for (uint i = 0; i < 32; ++i) {

if (GET\_BIT(registerAllocator->busyMask, i)) {

continue;

}

SET\_BIT(registerAllocator->busyMask, i);

return i + 1;

}

return -1;

}

int regAllocatorFree(struct RegisterAllocator\* registerAllocator,

const int registerNum) {

RESET\_BIT(registerAllocator->busyMask, registerNum - 1);

}

#endif