

Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра
Великого

—
Институт кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 2

по дисциплине «Формальные грамматики и теории
компиляторов»

Выполнил
студент гр. 4851003/80802

<подпись>

Сошнев М.Д.

Преподаватель

<подпись>

Мясников А.В.

Санкт-Петербург
2021

Оглавление

1 Цель работы.....	3
2 Задача.....	3
3 Ход работы.....	3
Сложные арифметические выражения.....	4
Хранение в регистрах.....	6
Условия.....	8
Циклы.....	9
Инкремент и декримент.....	9
4 Тестирование.....	11
5 Вывод.....	14
6 Приложение.....	15
Makefile.....	15
lexic.l.....	15
grammar.y.....	18
Vars.h.....	30
SolverStack.h.....	31
RegisterAllocator.h.....	40

1 ЦЕЛЬ РАБОТЫ

Понять принцип формирования бинарных файлов на практике.

2 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.

3 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 данной 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:

```
10
11 struct SolverStack {
12     struct Node {
13         enum NodeKind {
14             operator,
15             num,
16             var,
17             reg,
18             open_parenthesis,
19             close_parenthesis,
20             unary_minus
21         } kind;
22
23         char elem[128];
24
25         struct Node* next;
26         struct Node* prev;
27     } *begin, *end;
28
29     int size;
30 };
31
32 void stackInit(struct SolverStack* stack) {
33     stack->size = 0;
34     stack->begin = stack->end = NULL;
35 }
36
37 bool stackIsEmpty(const struct SolverStack* stack) {
38     return 0 == stack->size;
39 }
40
41 void stackPushBack(struct SolverStack* stack, const char* newElem,
42     const enum NodeKind newElemKind) {
43
44     struct Node *newNode = (struct Node *)
45     malloc(sizeof(struct Node));
```

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:

```
static const char priorityTable[12][12][4] = {
    {"++", "--", // prefix
    {"*", "/", "%"},
    {"+", "-"},
    {">>", "<<"},
    {">=", "<=", ">", "<"},
    {"==", "!="},
    {"&"},
    {"^"},
    {"|"},
    {"&&"},
    {"||"},
    {
        "=", "+=", "-=",
        "*=", "/=", "%=",
        "<<=", ">>=", "&=",
        "^=", "|="
    }
};
```

Рисунок 2 — массив операторов, упорядоченный по приоритету

Хранение в регистрах

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

```
struct RegisterAllocator {
    // 32 registers - 32 bit mask
    // bit = 0 - register free
    // bit = 1 - register busy

    uint32_t busyMask;
};
```

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).

```
24
25 int regAllocatorAlloc(struct RegisterAllocator* registerAllocator) {
26     for (uint i = 0; i < 32; ++i) {
27         if (GET_BIT(registerAllocator->busyMask, i)) {
28             continue;
29         }
30
31         SET_BIT(registerAllocator->busyMask, i);
32         return i + 1;
33     }
34
35     return -1;
36 }
37
38 int regAllocatorFree(struct RegisterAllocator* registerAllocator,
39                     const int registerNum) {
40     RESET_BIT(registerAllocator->busyMask, registerNum - 1);
41 }
42
43
44 #endif
45
```

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 образом, код

```
while (a > 0) {  
    code 1  
}
```

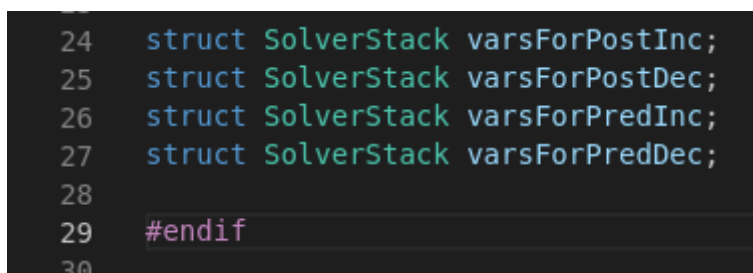
code 2

Будет переписан в

```
cycle_1_in:  
    asm code 1  
cycle_1_out:  
    asm code 2
```

Инкремент и декремент

Имеются 4 стека:



```
24 struct SolverStack varsForPostInc;  
25 struct SolverStack varsForPostDec;  
26 struct SolverStack varsForPredInc;  
27 struct SolverStack varsForPredDec;  
28  
29 #endif  
30
```

Рисунок 5 — стеки для переменных для инкремента/декремента

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

```

68
69 void endLineCallBack() {
70     stackForeachElem(&varsForPredInc, asmInc);
71     stackForeachElem(&varsForPredDec, asmDec);
72
73     solveExpr();
74
75     stackForeachElem(&varsForPostInc, asmInc);
76     stackForeachElem(&varsForPostDec, asmDec);
77
78     stackClear(&varsForPredInc);
79     stackClear(&varsForPredDec);
80     stackClear(&varsForPostInc);
81     stackClear(&varsForPostDec);
82 }

```

Рисунок 6 — callBack -функция

Данная колбек-функция вызывается, 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.

4 ТЕСТИРОВАНИЕ

Напишем тестовую программу:

```
1  A = a++ + ++b + c-- + --d;
2  S1 = S2 = S3 = 8*t - 9*u + i*(R+1)/w++;
3
4  if (a > 0) {
5      print a;
6  }
7
8  while (b > 0) {
9      if (a > b) {
10         a = a + b;
11     }
12     else {
13         b = a + b;
14     }
15
16     print 2;
17 }
18
19 print 999;
20 return a + 1;
21
```

Рисунок 7 — тестовая программа

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

```

1      MOV     R1, b
2      INC     R1
3      MOV     b, R1
4      MOV     R1, d
5      DEC     R1
6      MOV     d, R1
7      MOV     R1, a
8      ADD     R1, b
9      MOV     R2, R1
10     MOV     R1, R2
11     ADD     R1, c
12     MOV     R2, R1
13     MOV     R1, R2
14     ADD     R1, d
15     MOV     R2, R1
16     MOV     A, R2
17     MOV     R1, a
18     INC     R1
19     MOV     a, R1
20     MOV     R1, c
21     DEC     R1
22     MOV     c, R1
23     LDI     R1, 8
24     MUL     R1, t
25     MOV     R2, R1
26     LDI     R1, 9
27     MUL     R1, u
28     MOV     R3, R1
29     MOV     R1, R2
30     SUB     R1, R3

```

```

31     MOV     R2, R1
32     MOV     R1, R
33     ADD     R1, 1
34     MOV     R3, R1
35     MOV     R1, i
36     MUL     R1, R3
37     MOV     R3, R1
38     MOV     R1, R3
39     DIV     R1, w
40     MOV     R3, R1
41     MOV     R1, R2
42     ADD     R1, R3
43     MOV     R2, R1
44     MOV     S3, R2
45     MOV     S2, S3
46     MOV     S1, S2
47     MOV     R1, w
48     INC     R1
49     MOV     w, R1
50     MOV     R1, a
51     CMP     R1, 0
52     JG      case_1_1
53     JMP     case_2_1
54
55 case_1_1:
56     MOV     R1, a
57     CALL    print
58     JMP     out_1
59
60 case_2_1:
61     out_1:

```

Рисунки 8 — 9 полученный ассемблерный код

```

54
55 case_1_1:
56     MOV     R1, a
57     CALL    print
58     JMP     out_1
59
60 case_2_1:
61 out_1:
62
63 cycle_1_in:
64     MOV     R1, b
65     CMP     R1, 0
66     JLE     cycle_1_out
67     MOV     R1, a
68     CMP     R1, b
69     JG      case_1_2
70     JMP     case_2_2
71
72 case_1_2:
73     MOV     R1, a
74     ADD     R1, b
75     MOV     R2, R1
76     MOV     a, R2
77     JMP     out_2
78
79 case_2_2:
80     MOV     R1, a
81     ADD     R1, b
82     MOV     R2, R1
83     MOV     b, R2
84

```

```

84
85 case_3_2:
86 out_2:
87     LDI     R1, 2
88     CALL    print
89     JMP     cycle_1_in
90
91 cycle_1_out:
92     LDI     R1, 999
93     CALL    print
94     MOV     R1, a
95     ADD     R1, 1
96     RET     R1
97

```

Рисунки 10 - 11 — полученный ассемблерный код

5 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.

6 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;
(\V=) 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\t\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 =

getJumpTypeCommand(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\ttout_%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    =
```

```
getJumpTypeCommand(lastBinCompareOperator);
```

```
                                jmpTypeCmd    =
```

```
getOppositeJumpTypeCommand(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 wasAssignment = false;

    // iterate 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);

        wasAssignment = 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 (!wasAssignment && !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

```