TALLINNA TEHNIKAÜLIKOOL
INFOTEHNOLOOGIA TEADUSKOND
Arvutitehnika instituut

Algoritmikeelte analüüs

Kodutöö

Rooma arvud

Aivar Guitar
104607 IASM

Tallinn 2011

Sisukord

Sissejuhatus

Töö eesmärgiks on luua keel Rooma numbrtega antud täisarvudega arvutamiseks. Töö on realiseeritud C keeles mida kasutasime lähteprogrammi tesiendamiseks abstraktse süntaksi puu (Abstract Syntax Tree) kujule. Leksikaanalüüsiks kasutasime programmi FLEX ning süntaksianalüüsiks programmi Bison. Peale tavaarvutuse sai realiseeritud ka tsüklilause, tingimuslause ning väljatrükk.

Leksika

Vajaminev tähestik koosneb ladina tähtedest [a-zA-Z] ja sümbolitest [*/+-<>?=#]. Kasutusel on järgmised leksika elemendid:

- “while”
- “do”
- “if”
- “then”
- “end”
- “print”
- $[a-zA-Z_][a-zA-Z0-9_]* - muutuja
- [ivxlcdmIVXLCDM]* - rooma number
- [=] - omistamine

- [*/+-] - tehted – korrutamine, jagamine, liitmine ja lahutamine
- [<>?] - võrdlused - suurem, väiksem ja võrdne
- #.* - kommentaar kuni rea lõpuni

Lekseri sisendfail on ära toodus lisas - **lexer.l**

Rooma numbrite reeglid

Lekser küll eraldab sisendvoost rooma numbrid, kuid nende kontrollimiseks koostasime eraldi funktsiooni kasutades C keelt. Ladina tähestikus olevate sümbolite abil saab esitada positiivseid numbreid vahemikus 1 kuni 3999. Reeglite komplekt mille alusel numbri õigsust kontrollisime oli järgmine:

- lubatud sümbolid - I (1), V (5), X (10), L (50), C (100), D (500) and M (1000)
- D, L ja V võivad esineda ainult ühe korra
- M, C, X ja I ei tohi olla rohkem kui kolm korda järjest
- ainult I, X ja C võivad asetseda suurema numbri ees s.t. järgnevad kooslused on valed: VX, VL, VC, VD, VM, LC, LD, LM, DM. Eesseisev number ei tohi olla rohkem kui 10 korda väiksem järgnevast s.t. järgnevad kooslused on valed: IL, IC, ID, IM, XD, XM
- kui sümbolit on juba kasutatud järgneva sümboli väärtuse vähendamiseks, siis see sümbol ei tohi esineda uuest välja arvatud juhul kui ta ise on vähendatav
  s.t. I ei tohi esineda pärast IV ja IX
  ja C ei tohi esineda pärast CM ja CD, erandina kujul XC
  ja X ei tohi esineda pärast XL ja XC, erandina kujul IX
- kui numbrit on vähendatud siis ei tohi talle järgneda tema väärtus ega ka mitte pool tema väärtusest s.t.
  X ja V ei tohi järgneda IX
  C ja L ei tohi järgneda XC
  M ja D ei tohi järgneda CM
- sümbolit ei tohi kasutada vähendamiseks kui tema või temast viis korda suurem sümbol on juba esinenud s.t.

IV ja IX ei tohi järgneda I ega V

XL ja XC ei tohi järgneda X ega L

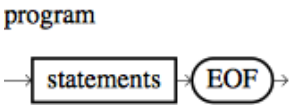
CD ja CM ei tohi järgneda C ega D

Antud reeglistikku on rakendatud funktsioonis validate, mis on ära toodud lisas – **rome_conv.c**

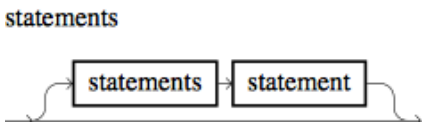
Süntaks

Järgnevalt toome ära graafiliselt süntaksireeglid mida kasutasime.

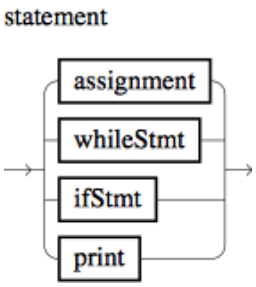
Programm koosneb lausetest ning lõppeb faili lõputunnusega



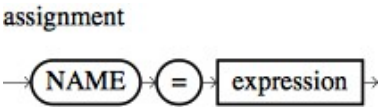
Lausehulk koosneb nullist või rohkemast lausest



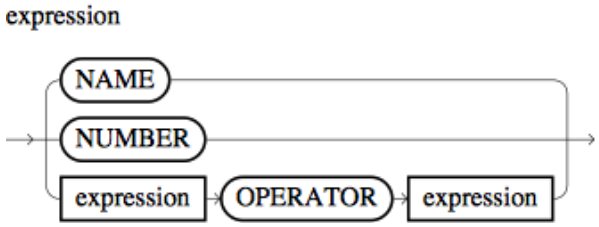
Lause koosneb omistamisest, tsüklilausest, tingimuslausest või väljatrükkimisest



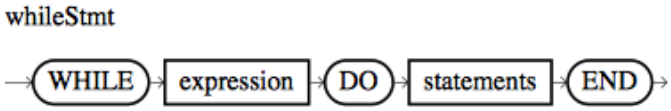
Omistamine koosneb muutujast ja omistatavast väljendist



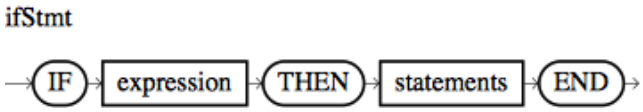
Väljend koosneb muutujast või väärtusest või tehetest väljenditega



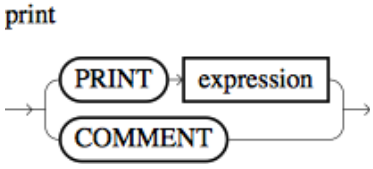
Tsüklilause koosneb tingimusest ja täidetavatest lausetest



Tingimuslause koosneb tingimusest ja täidetavadest lausetest



Välajtrükilause koosneb kas siis väljastatavast väärtusest või kommentaari reast mis väljastatakse



Parserina oli kasutusel Bison programm. Parseri sisendfail on ära toodud lisas **parser.y**

Parsimise ajal tekitatakse abstraktne süntaksipuu. Kui parsimine on õnnelikult lõpetatud siis funktsiooniga **runAST** käiakse süntaksipuu läbi. Teostatud on see C keeles ning nende lähtekoodid on lisades **createAST.c** ja **runAST.c**

Testprogrammid

Tulemuse kontrolliks tekitasime testprogrammid. Lihtsaim programm koosneb ühest reast ning väljastab rooma numbrid vahemikus 1 kuni 20.

```
$a = xx $b = $a while $a do $a = $a - i print $b - $a end
```

Antud programmi väljund näeb välja selline

```
$ ./rome < yks.rome
I
II
III
IV
V
VI
VII
VIII
IX
X
XI
XII
XIII
XIV
XV
XVI
XVII
XVIII
XIX
XX
```

Koostasime ka keerulisema programmi mis arvutab kahe arvu suurima ühise jagaja

```
# Arvutame suurima yhise jagaja, sisendid XLII ja LVI, eeldatav vastus XIV
$a = XLII
$b = LVI
while $b - $a do
  if $a > $b then
    $a = $a - $b
  end
  if $b > $a then
    $b = $b - $a
  end
end
print $a
```

Programmi väljun on järgnev

```
$ ./rome < gdc.rome
# Arvutame suurima yhise jagaja, sisendid XLII ja LVI, eeldatav vastus XIV
XIV
```

Lihtsustamaks rooma numbritega manipuleerimist tegime abiprogrammi, mis konverteeris araabia numbreid rooma numbriteks ning vastupidi. Programmi lähtekood on lisas **convert.c**

Lisad

lexer.l

```
%option noyywrap

%{
#include <stdlib.h>
#include "parser.h"
#include "rome_conv.h"
%}

%%

"while" return WHILE;
"do"    return DO;
"if"    return IF;
"then"  return THEN;
"end"   return END;
"print"return PRINT;

$[a-zA-Z_][a-zA-Z0-9_]* {yylval.name = strdup(yytext); return NAME;}
[ivxlcdmIVXLCDM]* {if(validate(yytext)){yylval.val=r2a(yytext);return NUMBER;}else
exit(-1);}
[=]  {return *yytext;}
[*/+-<>?] {yylval.op = *yytext; return OPERATOR;}
[ \t\n] {}
#.* {int i = yytext-strchr(yytext,'\n');char *tmp = calloc(i,1);yylval.name =
strncpy(tmp,yytext,i);return COMMENT;}
```

parser.y

```
%error-verbose
%{
#include <stdlib.h>
#include "AST.h"

void yyerror(const char* const message);
#define YYDEBUG 1
#define YYPARSE_PARAM startAST
%}
```

```
%union {
    int val;
    char op;
    char* name;
    struct recordAST* ast; /* this is the new member to store AST elements */
}

%token WHILE DO IF THEN END PRINT
%token<name> COMMENT
%token<name> NAME
%token<val> NUMBER
%token<op> OPERATOR
%type<ast> program statements statement assignment expression whileStmt ifStmt print
%start program

%expect 1

%%

program: statements { (*(struct recordAST**)startAST) = $1; }
;

statements: {$$=0;}
    | statements statement {$$=createStatement($1, $2);}
;

statement:
      assignment {$$=$1;}
    | whileStmt {$$=$1;}
    | ifStmt {$$=$1;}
    | print {$$=$1;}
;

assignment: NAME '=' expression {$$=createAssignment($1, $3);}
;

expression: NAME {$$=createExpByName($1);}
    | NUMBER {$$=createExpByNum($1);}
    | expression OPERATOR expression {$$=createExp($1, $3, $2);}
;

ifStmt: IF expression THEN statements END {$$=createIf($2, $4);}
;

whileStmt: WHILE expression DO statements END {$$=createWhile($2, $4);}
;

print: PRINT expression {$$=createPrint(NULL,$2);}
       | COMMENT {$$=createPrint($1,NULL);}
;

%%

void yyerror(const char* const message)
{
    fprintf(stderr, "Parse error:%s\n", message);
    exit(1);
}

int main()
{
    yydebug = 0;
    struct recordAST *a = 0;
```

```
    yyparse(&a);
    struct symRec* e = calloc(1, sizeof(struct symRec));
    runAst(e, a);
}
```

rome_conv.c

```
#include <ctype.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

char validate(char *);
int strcnt(char *, char);
char strfollow(char *, char *, char *);
char strfollowX(char *, char *, char *, char *);
int r2a(char *);
int r_value(char);
void rome_number(char *,char *,char *,int,char *);
char a2r(int, char *);

int strcnt(char *buf, char c) {
        int count = 0;
        int i;
        for (i = 0; i < strlen(buf); i++)
                if (buf[i] == c) count++;
        return count;
}

char strfollow(char *a, char *b, char *buf) {
        char *first = strstr(buf,b);
        if (first == NULL) return FALSE;
        first = first + strlen(b);
        char *second = strstr(first,a);
        if (second == NULL) return FALSE;
        return TRUE;
}

char strfollowX(char *a, char *b, char *buf, char *c) {
        char *first = strstr(buf,b);
        if (first == NULL) return FALSE;
        first = first + strlen(b);
        char *second = strstr(first,a);
        if (second == NULL) return FALSE;
        char *third = strstr(first,c);
        if (third == first) return FALSE;
        return TRUE;
}

char validate(char *buf) {
        if (strlen(buf) == 0) return FALSE;
        int i,j = 0;
        for (i = 0; i < strlen(buf); i++) {
                buf[i] = toupper(buf[i]);
                switch(buf[i]) {
                // lubatud ainult sümbolid -
                // I (1), V (5), X (10), L (50), C (100), D (500) and M (1000)
                case 'I':
                case 'V':
```

```
        case 'X':
        case 'L':
        case 'C':
        case 'D':
        case 'M':
                j++;
        }
}
if (i != j) return FALSE;
// D, L ja V võivad esineda ainult ühe korra
if (strcnt(buf,'D') > 1) return FALSE;
if (strcnt(buf,'L') > 1) return FALSE;
if (strcnt(buf,'V') > 1) return FALSE;
// M, C, X ja I ei tohi olla rohkem kui kolm korda järjest
if (strstr(buf,"MMMM") != NULL) return FALSE;
if (strstr(buf,"CCCC") != NULL) return FALSE;
if (strstr(buf,"XXXX") != NULL) return FALSE;
if (strstr(buf,"IIII") != NULL) return FALSE;
// ainult I, X ja C võivad asetseda suurema numbri ees
// s.t. järgnevad kooslused on valed: VX, VL, VC, VD, VM, LC, LD, LM, DM
// eesseisev number ei tohi olla rohkem kui 10 korda väiksem järgnevast
// s.t. järgnevad kooslused on valed: IL, IC, ID, IM, XD, XM
if (strstr(buf,"VX") != NULL) return FALSE;
if (strstr(buf,"VL") != NULL) return FALSE;
if (strstr(buf,"VC") != NULL) return FALSE;
if (strstr(buf,"VD") != NULL) return FALSE;
if (strstr(buf,"VM") != NULL) return FALSE;
if (strstr(buf,"LC") != NULL) return FALSE;
if (strstr(buf,"LD") != NULL) return FALSE;
if (strstr(buf,"LM") != NULL) return FALSE;
if (strstr(buf,"DM") != NULL) return FALSE;
if (strstr(buf,"IL") != NULL) return FALSE;
if (strstr(buf,"IC") != NULL) return FALSE;
if (strstr(buf,"ID") != NULL) return FALSE;
if (strstr(buf,"IM") != NULL) return FALSE;
if (strstr(buf,"XD") != NULL) return FALSE;
if (strstr(buf,"XM") != NULL) return FALSE;
// kui sümbolit on juba kasutatud järgneva sümboli väärtuse vähendamiseks,
// siis see sümbol ei tohi esineda uuest välja arvatud juhul kui ta ise on
// vähendatav s.t. I ei tohi esineda pärast IV ja IX
// ja C ei tohi esineda pärast CM ja CD, erandina kujul XC
// ja X ei tohi esineda pärast XL ja XC, erandina kujul IX
if (strfollow("I", "IV", buf)) return FALSE;
if (strfollow("I", "IX", buf)) return FALSE;
if (strfollowX("C", "CD", buf, "XC")) return FALSE;
if (strfollowX("C", "CM", buf, "XC")) return FALSE;
if (strfollowX("X", "XL", buf, "IX")) return FALSE;
if (strfollowX("X", "XC", buf, "IX")) return FALSE;
// kui numbrit on vähendatud siis ei tohi talle järgneda tema väärtus ega ka
// mitte pool tema väärtusest s.t.
// X ja V ei tohi järgneda IX
// C ja L ei tohi järgneda XC
// M ja D ei tohi järgneda CM
if (strfollow("X", "IX", buf)) return FALSE;
if (strfollow("V", "IX", buf)) return FALSE;
if (strfollow("C", "XC", buf)) return FALSE;
if (strfollow("L", "XC", buf)) return FALSE;
if (strfollow("M", "CM", buf)) return FALSE;
if (strfollow("D", "CM", buf)) return FALSE;
// sümbolit ei tohi kasutada vähendamiseks kui tema või temast viis korda suurem
// sümbol on juba esinenud s.t.
// IV ja IX ei tohi järgneda I ega V
// XL ja XC ei tohi järgneda X ega L
```

```
        // CD ja CM ei tohi järgneda C ega D
        if (strfollow("IV", "I", buf)) return FALSE;
        if (strfollow("IX", "I", buf)) return FALSE;
        if (strfollow("IX", "V", buf)) return FALSE;
        if (strfollow("XL", "X", buf)) return FALSE;
        if (strfollow("XC", "X", buf)) return FALSE;
        if (strfollow("XC", "L", buf)) return FALSE;
        if (strfollow("CD", "C", buf)) return FALSE;
        if (strfollow("CM", "C", buf)) return FALSE;
        if (strfollow("CM", "D", buf)) return FALSE;

        return TRUE;
}

int r_value(char c) {
        switch(c) {
        case 'I':
                return 1;
        case 'V':
                return 5;
        case 'X':
                return 10;
        case 'L':
                return 50;
        case 'C':
                return 100;
        case 'D':
                return 500;
        case 'M':
                return 1000;
        default:
                return 0;
        }
}

int r2a(char *buf) {
        int pos,arab,value,prev,count;
        arab = value = prev = count = 0;
        for (pos=strlen(buf)-1;pos>-1;pos--) {
                value = r_value(buf[pos]);
                if (value == 0) return 0;
                if (value > prev) {
                        arab += value;
                        prev = value;
                        count = 0;
                } else {
                        if (value == prev) {
                                if (buf[pos]=='V' || buf[pos]=='L' || buf[pos]=='D')
                                        return 0;
                                count++;
                                if (count > 2) return 0;
                                arab += value;
                        } else {
                                if (prev/value > 10) return 0;
                                arab -= value;
                                count = 3;
                        }
                        prev = value;
                }
   }
   return arab;
}
```

```
void rome_number(char *one,char *five,char *ten,int number,char *buf)
{
        int i;
        switch(number) {
        case 1:
        case 2:
        case 3:
                for (i=0;i<number;i++)
                        strcat(buf,one);
                break;
        case 4:
                strcat(buf,one);
                strcat(buf,five);
                break;
        case 5:
        case 6:
        case 7:
        case 8:
                strcat(buf,five);
                for (i=0;i<number-5;i++)
                        strcat(buf,one);
                break;
        case 9:
                strcat(buf,one);
                strcat(buf,ten);
                break;
        }
}

char a2r(int arab, char *buf)
{
        char *R_1    = "I";
        char *R_5    = "V";
        char *R_10   = "X";
        char *R_50   = "L";
        char *R_100  = "C";
        char *R_500  = "D";
        char *R_1000 = "M";

        buf[0] = 0;
        if (arab < 1) return FALSE;
        if (arab > 3999) return FALSE;
        rome_number(R_1000,"","",arab/1000,buf);
        arab %= 1000;
        rome_number(R_100,R_500,R_1000,arab/100,buf);
        arab %= 100;
        rome_number(R_10,R_50,R_100,arab/10,buf);
        arab %= 10;
        rome_number(R_1,R_5,R_10,arab,buf);
        return TRUE;
}
```

rome_conv.h

```
char validate(char *);
int r2a(char *);
char a2r(int, char *);
```

createAST.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "AST.h"

struct symRec *sym_table;

void* checkAlloc(size_t sz) {
    void* result = calloc(sz, 1);
    if (!result) {
        fprintf(stderr, "alloc failed\n");
        exit(1);
    }
}

struct symRec *getSym(char const *s_name) {
        struct symRec *ptr = sym_table;
        while(ptr != NULL) {
                if (strcmp(ptr->name, s_name) == 0) return ptr;
                ptr = ptr->next;
        }
        return NULL;
}

struct symRec *putSym(char const *s_name) {
        struct symRec *ptr = (struct symRec *)calloc(sizeof(struct symRec),1);
        ptr->name = (char *)checkAlloc(strlen(s_name)+1);
        strcpy(ptr->name, s_name);
        ptr->next = sym_table;
        sym_table = ptr;
        return ptr;
}

struct recordAST* createAssignment( char*name, struct recordAST* val) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comAss;
    result->data.assignment.name = name;
    struct symRec *ptr = getSym(name);
    if (ptr == NULL) {
        ptr = putSym(name);
        ptr->value = val->data.val;
    }
    result->data.assignment.right = val;
    return result;
}

struct recordAST* createExpByNum(int val) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comNum;
    result->data.val = val;
    return result;
}

struct recordAST* createExpByName(char*name) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comId;
    result->data.name = name;
    struct symRec *ptr = getSym(name);
    if (ptr == NULL) {
        ptr = putSym(name);
```

```
    }
    return result;
}

struct recordAST* createExp(struct recordAST* left, struct recordAST* right, char op) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comBinExp;
    result->data.expression.left = left;
    result->data.expression.right = right;
    result->data.expression.op = op;
    return result;
}

struct recordAST* createStatement(struct recordAST* result, struct recordAST* toAppend)
{
    if (!result) {
        result = checkAlloc(sizeof(*result));
        result->kind = comStat;
        result->data.statements.count = 0;
        result->data.statements.statements = 0;
    }
    result->data.statements.count++;
    result->data.statements.statements = realloc(result->data.statements.statements,
result->data.statements.count*sizeof(*result->data.statements.statements));
    result->data.statements.statements[result->data.statements.count-1] = toAppend;
    return result;
}

struct recordAST* createIf(struct recordAST* cond, struct recordAST* exec) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comIf;
    result->data.whileStmt.cond = cond;
    result->data.whileStmt.statements = exec;
    return result;
}

struct recordAST* createWhile(struct recordAST* cond, struct recordAST* exec) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comWhile;
    result->data.whileStmt.cond = cond;
    result->data.whileStmt.statements = exec;
    return result;
}

struct recordAST* createPrint(char *text,struct recordAST* param) {
    struct recordAST* result = checkAlloc(sizeof(*result));
    result->kind = comPrint;
    result->data.print.text = text;
    result->data.print.param = param;
    return result;
}
```

runAST.h

```
#include <stdlib.h>
#include <stdio.h>

#include "AST.h"
#include "rome_conv.h"
```

```
int runTermExp(struct symRec* e, struct recordAST* a);
int runBinExp(struct symRec* e, struct recordAST* a);
void runAssign(struct symRec* e, struct recordAST* a);
void runIf(struct symRec* e, struct recordAST* a);
void runWhile(struct symRec* e, struct recordAST* a);
void runPrint(struct symRec* e, struct recordAST* a);
void runStmt(struct symRec* e, struct recordAST* a);

int (*valExecs[])(struct symRec* e, struct recordAST* a) = {
    runTermExp,
    runTermExp,
    runBinExp,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

void (*runExecs[])(struct symRec* e, struct recordAST* a) = {
    NULL,
    NULL,
    NULL,
    runAssign,
    runIf,
    runWhile,
    runPrint,
    runStmt,
};

int dispatchExpression(struct symRec* e, struct recordAST* a) {
    return valExecs[a->kind](e, a);
}

void dispatchStatement(struct symRec* e, struct recordAST* a) {
    runExecs[a->kind](e, a);
}

int runTermExp(struct symRec* e, struct recordAST* a) {
    if (comNum == a->kind) {
        return a->data.val;
    } else {
        if (comId == a->kind) {
              e = getSym(a->data.name);
            return e->value;
        }
    }
    fprintf(stderr, "runAST: Unknown expression(%d)\n", a->kind);
    exit(1);
}

int runBinExp(struct symRec* e, struct recordAST* a) {
    const int left = dispatchExpression(e, a->data.expression.left);
    const int right = dispatchExpression(e, a->data.expression.right);
    switch(a->data.expression.op)
    {
        case '+': return left + right;
        case '-': return left - right;
        case '*': return left * right;
        case '/': return left / right;
        case '<': return left < right;
        case '>': return left > right;
        case '?': return left == right;
```

```
        default:
            fprintf(stderr,  "runAST: Unknown operator:%c\n", a->data.expression.op);
            exit(1);
    }
}

void runAssign(struct symRec* e, struct recordAST* a) {
    struct recordAST* r = a->data.assignment.right;
    e = getSym(a->data.assignment.name);
    e->value = dispatchExpression(e, r);
}

void runIf(struct symRec* e, struct recordAST* a) {
    struct recordAST* const c = a->data.ifStmt.cond;
    struct recordAST* const s = a->data.ifStmt.statements;
    if (dispatchExpression(e, c))  dispatchStatement(e, s);
}

void runWhile(struct symRec* e, struct recordAST* a) {
    struct recordAST* const c = a->data.whileStmt.cond;
    struct recordAST* const s = a->data.whileStmt.statements;
    while (dispatchExpression(e, c)) dispatchStatement(e, s);
}

void runPrint(struct symRec* e, struct recordAST* a) {
       if (a->data.print.text == NULL) {
              char *response = (char *)checkAlloc(32);
              int i = dispatchExpression(e, a->data.print.param);
              if (a2r(i,response)) printf("%s \n",response);
              else {
                     fprintf(stderr,  "runAST: Number not in range (1-3999):%i\n",i);
                     exit(1);
              }
       } else
              printf("%s\n", a->data.print.text);
}

void runStmt(struct symRec* e, struct recordAST* a) {
    int i;
    for (i=0; i<a->data.statements.count; i++) {
        dispatchStatement(e, a->data.statements.statements[i]);
    }
}

void runAst(struct symRec* e, struct recordAST* a) {
    dispatchStatement(e, a);
}
```

AST.h

```
struct symRec {
       int value;
       char *name;
       struct symRec *next;
};

struct recordAST
{
    enum {comId, comNum, comBinExp, comAss, comIf, comWhile, comPrint, comStat } kind;
    union
```

```
    {
        int val;
        char* name;
        struct
        {
            struct recordAST *left, *right;
            char op;
        }expression;
        struct
        {
            char*name;
            struct recordAST* right;
        }assignment;
        struct
        {
            int count;
            struct recordAST** statements;
        }statements;
        struct
        {
            struct recordAST* cond;
            struct recordAST* statements;
        } ifStmt;
        struct
        {
            struct recordAST* cond;
            struct recordAST* statements;
        } whileStmt;
        struct
        {
               char *text;
            struct recordAST* param;
        }print;
    } data;
};

struct symRec *getSym(char const *);
void* checkAlloc(size_t sz);
struct recordAST* createAssignment(char*name, struct recordAST* val);
struct recordAST* createExpByNum(int val);
struct recordAST* createExpByName(char*name);
struct recordAST* createExp(struct recordAST* left, struct recordAST* right, char op);
struct recordAST* createStatement(struct recordAST* dest, struct recordAST* toAppend);
struct recordAST* createIf(struct recordAST* cond, struct recordAST* exec);
struct recordAST* createWhile(struct recordAST* cond, struct recordAST* exec);
struct recordAST* createPrint(char *text, struct recordAST* param);

void runAst(struct symRec* e, struct recordAST* a);
```

convert.c

```
#include <stdio.h>
#include <stdlib.h>
#include "rome_conv.h"

int main(void) {
        char buf[32];
        char response[32];
        int i;
```

```
        printf("Number please (Roman or Arabic) : ");
        scanf("%s",buf);
        printf("Got - '%s'\n",buf);
        if (isalpha(buf[0])) {
                if (validate(buf)) {
                        i = r2a(buf);
                        if (a2r(i,response)) {
                                if (!strcmp(buf,response)) {
                                        printf("Roman - %s => %i\n",response,i);
                                        return EXIT_SUCCESS;
                                }
                        } else {
                                printf("Use number between 1 and 3999, %i\n",i);
                                return EXIT_FAILURE;
                        }
                }
        } else {
                i = atoi(buf);
                if (a2r(i,response)) {
                        int j = r2a(response);
                        if (i==j) {
                                printf("Arabic - %i => %s\n",j,response);
                                return EXIT_SUCCESS;
                        }
                }
                else {
                        printf("Use number between 1 and 3999, %i\n",i);
                        return EXIT_FAILURE;
                }
        }
        printf("Failure - %s\n",buf);
        return EXIT_FAILURE;
}
```

Makefile

```
all: rome convert

rome: parser.y lexer.l runAST.c createAST.c AST.h rome_conv.c rome_conv.h
        bison parser.y -d -o parser.c
        flex -o lexer.c lexer.l
        gcc -o rome lexer.c parser.c createAST.c runAST.c rome_conv.c

convert: convert.c rome_conv.c rome_conv.h
        gcc -o convert convert.c rome_conv.c
```