Relatório do TP2

Processamento de Linguagens

Mestrado Integrado em Engenharia Informática

Grupo 65

Carlos Pereira (A61887)
João Pires Barreira (A73831)
Rafael Costa (A61799)

Junho 2017

Universidade do Minho
Escola de Engenharia

Índice

1 Introdução

O presente trabalho consiste no desenvolvimento de uma linguagem de programação. Para isso, é necessário criar um analisador léxico para a linguagem (com o recurso ao *Flex*), bem como um compilador próprio (utilizando, para isso, o *YACC*) que converta o código para pseudo-*Assembly* da máquina de stack virtual (VM) fornecida.

Foi-nos, então, proposto que codificássemos uma linguagem em que fosse possível declarar e manusear variáveis inteiras e arrays, ler e escrever no *standard input* e no *standard output*, executar operações aritméticas e ciclos condicionais e cíclicos possivelmente aninhados.

Para efeitos de teste, foi-nos ainda proposto que criássemos exemplos de programas escritos na linguagem para resolver problemas relativos a todas as funcionalidades descritas acima.

Ao longo deste relatório explicaremos, com detalhe, a resolução de cada um destes problemas. Além disso, apresentaremos uma análise explicativa das decisões por trás do desenvolvimento da linguagem e da gramática.

2 Analisador Léxico

2.1 Expressões Regulares

2.1.1 num

As variáveis do tipo *num* estão associadas ao estado *NUMVAR*. Neste estado apenas, quando se encontram sequências de letras estas são devolvidas em *NAME*. Caso se encontrem carateres como '=', '[' e ']', estes são devolvidos diretamente, enquanto que '\n', '\r' e '\t' são ignorados. As sequência de números são devolvidas em NUM. Caso se encontre um ';', recomeça-se o estado *INITIAL*.

```
num                 { BEGIN NUMVAR; }
<NUMVAR>[a-zA-Z]+   { yylval.str = strdup(yytext);
                      return NAME; }
<NUMVAR>;           { BEGIN INITIAL;
                      return yytext[0]; }
<NUMVAR>[=\[\]]     { return yytext[0]; }
<NUMVAR>[0-9]+      { yylval.n = atoi(yytext);
                      return NUM; }
<NUMVAR>[ \n\r\t]   { }
```

2.1.2 read

O comando *read* está associado aos estados *READING* e *READINGTEXT*.

```
read { BEGIN READING;
       yylval.str = strdup(yytext);
       return READ; }

<READING>\(         { BEGIN READINGTEXT;
                      return yytext[0]; }
<READING>[ \n\r\t]  { }


<READINGTEXT>;     { BEGIN INITIAL;
                     return yytext[0]; }
<READINGTEXT>\)      { return yytext[0]; }
<READINGTEXT>[a-zA-Z>0-9 \<:"]* { yylval.str = strdup(yytext);
                                  return TEXT; }
<READINGTEXT>[\n\r\t]       { return yytext[0]; }
```

2.1.3 write

O comando *write* tem os estados *WRITING*, *WRITINGARGS* e *WRITINGTEXT* a ele associados.

```
write { BEGIN WRITING;
        yylval.str = strdup(yytext);
        return WRITE; }

<WRITING>\(      { BEGIN WRITINGARGS;
                   return yytext[0]; }
<WRITING>[ \n\r\t]  { }


<WRITINGARGS>;          { BEGIN INITIAL;
                          return yytext[0]; }
<WRITINGARGS>\"         { BEGIN WRITINGTEXT; }
<WRITINGARGS>\)         { return yytext[0]; }
<WRITINGARGS>[a-zA-Z]+  { yylval.str = strdup(yytext);
                          return NAME; }
<WRITINGARGS>[ \n\r\t]  { }
<WRITINGARGS>[\[\]]     { return yytext[0]; }
<WRITINGARGS>\+         { return yytext[0]; }


<WRITINGTEXT>[a-zA-Z0-9 \.\\><:]*  { yylval.str = strdup(yytext);
                                     return TEXT; }
<WRITINGTEXT>\"                { BEGIN WRITINGARGS; }
```

2.1.4 loop

loop corresponde a um comando de ciclo, tendo a ele associado o estado *LOOPSTATE*.

```
loop { BEGIN LOOPSTATE;
       yylval.str = strdup(yytext);
       return LOOP; }

<LOOPSTATE>[ \n\t\r]              { }
<LOOPSTATE>[\(\)\&\|=<>!\+\-/\*;] { return yytext[0]; }
<LOOPSTATE>[a-zA-Z]+              { yylval.str = strdup(yytext);
                                    return NAME; }
<LOOPSTATE>[0-9]+                 { yylval.n = atoi(yytext);
                                    return NUM; }
```

```
<LOOPSTATE>\{                          { BEGIN INITIAL;
                                         return yytext[0]; }
```

2.1.5 case

case corresponde à primeira parte de uma instrução condicional. Tem a si associado o estado *CASESTATE*.

```
case { BEGIN CASESTATE;
       yylval.str = strdup(yytext);
       return CASE; }

<CASESTATE>[ \n\t\r]                { }
<CASESTATE>[\(\)\&\|=<>!\+\-/\*\[\]] { return yytext[0]; }
<CASESTATE>mod                      { yylval.str = strdup(yytext);
                                      return MOD; }
<CASESTATE>[a-zA-Z]+                { yylval.str = strdup(yytext);
                                      return NAME; }
<CASESTATE>[0-9]+                   { yylval.n = atoi(yytext);
                                      return NUM; }
<CASESTATE>\{                       { BEGIN INITIAL;
                                      return yytext[0]; }
```

2.1.6 otherwise

otherwise corresponde à segunda parte de uma instrução condicional. O estado *OTHERSTATE* está associado a este comando.

```
otherwise { BEGIN OTHERSTATE;
            yylval.str = strdup(yytext);
            return OTHERWISE; }

<OTHERSTATE>[ \n\t\r]  { }
<OTHERSTATE>\{         { BEGIN INITIAL;
                         return yytext[0]; }
```

3 Compilador

3.1 Gramática

Nesta secção apresentamos a gramática definida por nós para a linguagem *S++*. Foram definidos termos para a representação de números inteiros, operações aritméticas, operações de *input* e de *output*, ciclos e condições.

3.1.1 Lines

Aqui temos o termo que engloba todas as possibilidades de uma linha de código escrita em *S++*: *Lines*. Na linguagem de programação *S++*, uma linha pode ser um símbolo *Var* seguido por *';'*, uma operação de *Input* ou de *Output*, o símbolo *Cond* que representa uma condição ou o símbolo *Loop* que indica tratar-se de um ciclo.

```
Lines :  { }
      |  Var ';' Lines
      |  Input Lines
      |  Output Lines
      |  Cond Lines
      |  Loop Lines
      ;
```

3.1.2 Var

Em *S++*, uma variável é representada pelo símbolo *Var*, a que corresponde ao tipo número inteiro. Tal como em outras linguagens de programação, a uma variável pode-se atribuir um valor inicial (mas, por defeito, é atribuido o valor 0), realizar uma operação aritmética ou declarar um *array* unidimensional.

```
Var : NAME              { insertVarsTree($1,
                                        reg++,
                                        0);
                        }

    | NAME '[' NUM ']'  {
                            fprintf(fp,
                                    "\tpushn %d\n",
                                    $3);
                        }

    | NAME '=' NUM      { if (start == 0) {
                           insertVarsTree($1,
```

```
                                        reg++,
                                        $3);
                            }
                            else {
                             fprintf(fp,
                                     "\tpushi %d\n\tstoreg %d\n",
                                     $3,
                                     getVarReg($1));
                           }
                         }

    | NAME '[' NUM ']' '=' NUM  {  }
    | NAME '=' AritExp  { if (loopExpressions < loop) {
                        loopExpressions++;
                             lExpStack = pushString(lExpStack,
                                                    getAtrib($3,
getVarReg($1)));
                              $$ = strdup(getAtrib($3,
                                                   getVarReg($1)));
                       }
                       else {
                            fprintf(fp,
                                    "%s",
                                    getAtrib($3,
                                    getVarReg($1)));
                            }
                          }
    | NAME '[' NUM ']' '=' AritExp  {  }
    | NAME '[' NAME ']' '=' AritExp {
                                 fprintf(fp,
                                 "\tpushgp\n\t
                                    pushi2\n\t
                                    padd\n\t
                                    pushg %d\n
                                    %s\t
                                    storen\n",
                                 getVarReg($3),
                                 $6);
                               }
    ;
```

3.1.3 Input

O *input* é feito através do comando "*read*"definido por nós.

```
Input : READ '(' TEXT ')' ';'        { readInput("", $3); }
      | NAME '=' READ '(' TEXT ')' ';' { readInput($1, $5); }
      | NAME '[' NAME ']' '=' READ '(' TEXT ')' ';' {
                fprintf(fp, "\tpushgp\n\t
                                pushi 2\n\t
                                padd\n\t
                                pushg %d\n",
                        getVarReg($3));
                fprintf(fp, "\tpushs %s\n\t
                                writes\n\t
                                read\n\t
                                atoi\n\t
                                storen\n",
                        $8);
      }
    ;
```

Pode-se fazer a leitura do que é escrito no teclado e atribuir o valor introduzido a uma variável ou simplesmente fazer uma leitura sem efetuar uma atribuição. Também é possível indicar uma mensagem, entre os parêntesis, a ser apresentada no ecrã no momento da escrita, por exemplo:

```
num c = read("Valor > ");

read("A ler valor >");
```

3.1.4 Output

O *Output* consiste na escrita de um conjunto de carateres no ecrã do utilizador. A palavra que representa esta escrita é *write*, onde deverá ser indicado entre parêntesis e delimitado por aspas os carateres a serem imprimidos.

```
Output : WRITE '(' OutArgs ')' ';' { }
       ;
```

Para ser feita a escrita do valor de uma variável, basta indicar o nome da variável fora dos parêntesis. Também é possível escrever no ecrã uma sequência de carateres e um conjunto de variáveis, na mesma instrução. Apenas é necessário separá-los pelo símbolo '+':

```
num c = 4;

write("Um quadrado tem " + c + " cantos");
```

3.1.5 OutArgs

Corresponde ao interior de uma expressão *write* (i.e. ao que está dentro de parêntesis). Assim sendo, pode corresponder apenas a um símbolo terminal *NAME* ou apenas a um TEXT (pedaço de texto). Pode ainda combinar ambos utilizando o símbolo de concatenação '+'.

OutArgs

```
OutArgs : OutArgs '+' OutArgs { }
        | NAME '[' NAME ']'   { fprintf(fp,
                                        "\t pushgp    \n\t
                                            pushi 2   \n\t
                                            padd      \n\t
                                            pushg %d  \n\t
                                            loadn     \n\t
                                            writei    \n",
                                        getVarReg($3)); }
       | NAME       { writeOutput($1, ""); }
       | TEXT       { writeOutput("", $1); }
       ;
```

3.1.6 Loop

Loop representa um ciclo. Aqui temos uma expressão lógica (*LoopExp*) que indica se o ciclo deverá continuar a ser executado mais uma vez, sendo esta obrigatória. Depois, opcionalmente, poderá efetuar-se a declaração da variável de controlo de ciclo e da expressão aritmética que altera o seu valor (1º caso) ou simplesmente apenas indicar essa expressão (2º caso).

```
Loop : LoopBegin '(' Var ';' LoopExp ';' Var ')' '{' Lines '}' {
        fprintf(fp,
                "%s%s%s\t  jz endLoop%d  \n\t
                          jump loop%d   \n
                          endLoop%d:    \n",
                $3, $7, $5,
                topInt(loopStack),
                topInt(loopStack),
                topInt(loopStack));

        loopStack = popInt(loopStack);
        lExpStack = popString(lExpStack);
    }
     | LoopBegin '(' LoopExp ';' Var ')' '{' Lines '}' {
        fprintf(fp,
```

```
                "%s%s\t jz endLoop%d  \n\t
                        jump loop%d   \n
                        endLoop%d:    \n",
                $5,
                $3,
                topInt(loopStack),
                topInt(loopStack),
                topInt(loopStack));
        loopStack = popInt(loopStack);
        lExpStack = popString(lExpStack);
    }
     | LoopBegin '(' LoopExp ')' '{' Lines '}' {
        fprintf(fp,
                "%s\t jz endLoop%d \n\t
                      jump loop%d  \n
                      endLoop%d:   \n",
                $3,
                topInt(loopStack),
                topInt(loopStack),
                topInt(loopStack));
        loopStack = popInt(loopStack);
        lExpStack = popString(lExpStack);
     }
     ;
```

```
num i;

loop(i < 10; i = i + 1) {
    write("Iteracao: " + i + "\n");
}
```

3.1.7 LoopBegin

Este termo faz o *push* à *stack* de ciclo a indicar que está presente mais um ciclo, relativamente ao anterior.

```
LoopBegin : LOOP {
                writeStart();
                fprintf(fp, "loop%d:\n", loop);
                loopStack = pushInt(loopStack, loop++);
            }
          ;
```

3.1.8 Cond

Uma condição é representada pelp termo *case*, seguida da expressão lógica, entre parêntesis, que se tem de verificar para que o código dessa condição seja executado. *CondBegin* representa a verificação da condição, enquanto que *OtherCond* indica o código que deverá ser executado caso a expressão não se verifique.

```
Cond : CondBegin '(' CondExp ')' '{' Lines '}' {
            fprintf(fp,
                    "\t jump endCond%d \n",
                    topInt(condStack));
       }
     | Cond OtherCond       { }
     ;
```

Em *S++*, uma condição é algo do género:

```
case (numero == 1) {
    write("Sim\n");
}
otherwise {
    write("Nao\n");
}
```

3.1.9 OtherCond

Aqui temos o termo que indica quando a expressão de uma condição não se verifica e deverá ser executado um código alternativo.

```
OtherCond : OtherCondBegin '{' Lines '}' {
                fprintf(fp,
                        "endCond%d: \n",
                        topInt(condStack));
                condStack = popInt(condStack);
            }
          | OtherCondBegin Cond          { }
          ;
```

3.1.10 OtherCondBegin

Inicialização de quando o valor da expressão da condição é falso, representado pelo símbolo *otherwise*.

```
OtherCondBegin : OTHERWISE { writeOtherCond(); }
               ;
```

3.1.11 CondExp

Este termo refere-se à expressão de um condicional. É representado como sendo a operação condicional entre dois *AritExp*, descritos abaixo. Estas operações podem ser '<', '>', '<=', '>=', '==', '!=', '&' e '|'.

```
CondExp : AritExp '<' AritExp     {
            fprintf(fp,
                    "%s%s\t inf \n\t
                            jz otherCond%d \n",
                    $1,
                    $3,
                    topInt(condStack));
          }
        | AritExp '>' AritExp     {
            fprintf(fp,
                    "%s%s\t sup \n\t
                            jz otherCond%d\n",
                    $1,
                    $3,
                    topInt(condStack));
          }
        | AritExp '<' '=' AritExp {
            fprintf(fp,
                    "%s%s\t infeq \n\t
                            jz otherCond%d \n",
                    $1,
                    $4,
                    topInt(condStack));
          }
        | AritExp '>' '=' AritExp {
            fprintf(fp,
                    "%s%s\t supeq \n\t
                            jz otherCond%d \n",
                    $1,
                    $4,
                    topInt(condStack));
          }
        | AritExp '=' '=' AritExp {
            fprintf(fp,
                    "%s%s\t equal \n\t
                            jz otherCond%d \n",
                    $1,
                    $4,
                    topInt(condStack));
          }
```

```
| AritExp '!' '=' AritExp {
    fprintf(fp,
            "%s%s\t equal \n\t
                    jnz otherCond%d \n",
            $1,
            $4,
            topInt(condStack));
  }
| ModExp    {
       fprintf(fp,
               "%s\t jz otherCond%d \n",
               $1,
               topInt(condStack));
  }
| CondExp '&' CondExp     { }
| CondExp '|' CondExp     { }
;
```

3.1.12 LoopExp

Este termo refere-se à expressão condicional de um loop. É representado como sendo a uma operação condicional entre dois *AritExp*, descritos abaixo. Estas operação podem ser '<', '>', '<=', '>=', '==', '!=', '&' e '|'.

```
LoopExp : AritExp '<' AritExp     {
    strcpy($$,
           getLoopExp("\t inf \n",
                      $1, $3));
    }
| AritExp '>' AritExp     {
    strcpy($$,
           getLoopExp("\t sup \n",
                      $1, $3));
    }
| AritExp '<' '=' AritExp {
    strcpy($$,
           getLoopExp("\t infeq \n",
                      $1, $4));
    }
| AritExp '>' '=' AritExp {
    strcpy($$,
           getLoopExp("\t supeq \n",
                      $1, $4));
    }
| AritExp '=' '=' AritExp {
```

```
        strcpy($$,
               getLoopExp("\t equal \n",
               $1, $4));
        }
    | AritExp '!' '=' AritExp {
        strcpy($$,
               getLoopExp("\t diff \n",
               $1, $4));
        }
    | LoopExp '&' LoopExp     { }
    | LoopExp '|' LoopExp     { }
    ;
```

3.1.13 AritExp

Este termo corresponde a uma expressão aritmética e é codificado como sendo a soma, subtração, multiplicação ou divisão de duas *AritExp*. Pode corresponder também à negação de uma *AritExp*, a uma *AritExp* entre parêntesis ou aos símbolos terminais *NAME* e *NUM*.

```
AritExp : AritExp '+' AritExp  {
            sprintf($$,
                    "%s%s\t add \n",
                    $1, $3);
          }
        | AritExp '-' AritExp  {
            sprintf($$, "%s%s\t sub \n",
                    $1, $3);
          }
        | AritExp '*' AritExp  {
            sprintf($$,
                    "%s%s\t mul \n",
                    $1, $3);
          }
        | AritExp '/' AritExp  {
            sprintf($$, "%s%s\t div \n",
                    $1, $3);
          }
        | '-' AritExp          { }
        | '(' AritExp ')'      { }
        | NAME                 {
              $$ = strdup(getAritVar($1));
          }
        | NUM                  {
              $$ = strdup(getAritNum($1));
```

```
  }
| NAME '[' NAME ']'    {
        $$ = strdup(getArrayVar($3));
  }
;
```

3.2 ModExp

Este termo codifica a expressão de um módulo

```
ModExp: | AritExp MOD AritExp  {
        sprintf($$, "%s%s\tmod\n", $1, $3);
  }
;
```

4 Problemas propostos

4.1 Ler 4 números e dizer se podem ser os lados de um quadrado

Para a resolução deste problema basta declarar quatro variáveis do tipo *num*, lendo para as mesmas os números recebidos com a instrução *read*. Depois, foi utilizado um ciclo condicional (*case*/*otherwise*) que compara os inteiros recebidos, vendo se são iguais. Imprime-se depois uma mensagem (utilizando a instrução *write*) a informar se os lados são ou não os de um quadrado, consoante a veracidade da condição anterior.

```
num a = 0;
num b;
num c = 3;
num d = 0;

a = read("Inteiro > ");
b = read("Inteiro > ");
c = read("Inteiro > ");
d = read("Inteiro > ");

case (a == c & a == d) {
 write("Quadrado\n");
}
otherwise {
 write("Nao e quadrado\n");
}
```

4.2 Ler um inteiro N, depois ler N números e escrever o menor deles

Para a resolução deste problema, temos de declarar quatro variáveis para guardar o inteiro N, os números lidos, o inteiro correspondente ao ciclo e ainda o menor dos números lidos. É também necessário um ciclo *loop* para ler os números até N (com a instrução *read*). Depois, guarda-se o número lido em aux e, num ciclo condicional (*case*/*otherwise*), verifica-se se o número lido é menor do que o menor atual. Caso seja, o número lido passa a ser o menor. Por fim imprime-se o menor número usando a instrução *write*.

```
num n;
num m;
num aux;
num i = 1;
```

```
n = read("N: ");

m = read("Inteiro: ");

loop (i < n; i = i + 1){
 aux = read("Inteiro: ");

 case (aux < m) {
    m = aux;
 }
 otherwise {

 }
}

write("Menor: " + m);
```

4.3 Ler N (constante do programa) números e calcular e imprimir o seu produtório

Para a resolução deste problema, são necessárias quatro variáveis do tipo inteiro para guardar o N, os números que são lidos, o acumulador dos produtos e a variável corresponde ao ciclo. Depois, basta ler o N (com a instrução *read*) e dentro de um ciclo *loop* ler os números recebidos e multiplicar o acumulador por cada um deles. No fim, é apenas necessário imprimir o acumulador correspondente ao produtório dos números recebido ((com a instrução *write*).

```
num n;
num i;
num acc = 1;
num x;

n = read("N: ");

loop (i < n; i = i + 1) {
 x = read("Inteiro: ");

 acc = acc * x;
}

write("Produtorio: " + acc);
```

4.4 Contar e imprimir os números ímpares de uma sequência de números naturais

Para a resolução deste problema, são necessárias quatro variáveis do tipo inteiro para guardar o N, os números que são lidos, o contador de ímpares e a variável corresponde ao ciclo. Depois, basta ler o N (com a instrução *read*) e dentro de um ciclo *loop* ler os números recebidos e para cada um deles verificar se é impar com o recurso a um condicional e às operações de igualdade e módulo. Imprime-se uma mensagem (com a instrução *read*) indicando se o número lido é ou não ímpar e, caso seja, incrementa-se o contador. No fim, é necessário imprimir o contador que corresponderá ao total dos números ímpares introduzidos.

```
num count = 0;
num i = 0;
num n;
aux = 0;

n = read("N: ");

loop (i < n; i = i + 1) {
 aux = read("Inteiro: ");

 case (aux mod 2) {
    write("Impar\n");
    count = count + 1;
 }
 otherwise {
 }
}


write("Impares: " + count);
```

4.5 Ler e armazenar os elementos de um vetor de comprimento N e imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas

```
num i;
num j;
num aux;
num array[3];

loop (i < 3; i = i + 1) {
   array[i] = read("Inteiro: ");
```

```
    }

    i = 0;

    loop (i < 2; i = i + 1) {

        j = i + 1;
        loop (j < 3; j = j + 1) {
            case (array[i] < array[j]) {
                aux = array[i];
                array[i] = array[j];
            array[j] = aux;
            }
        otherwise {

            }
        }
     }

     i = 0;

     loop (i < 3; i = i + 1) {
        write(array[i] + " ");
     }
```

4.6 Ler e armazenar N números num array e imprimir os valores por ordem inversa

```
num n = 5;
num array[5];
num i;

loop (i < n; i = i + 1) {
  array[i] = read("Inteiro: ");
}

loop (n >= 0; n = n - 1) {
  write(array[n] + "\n");
}
```

5 Conclusão

Com a conclusão deste trabalho, pudemos pôr em prática os conceitos adquiridos durante as aulas de Processamento de Linguagens sobre Flex e Yacc.

Podemos dizer que este trabalho se revelou bastante interessante devido ao facto de termos tido a oportunidade de criar a nossa própria linguagem de programação e percebido um pouco dos procedimentos por detrás das linguagens que utilizamos todos os dias.