Processamento de Linguagens (3º Ano MIEI)

Trabalho Prático 2 - YACC

Relatório de Desenvolvimento

Fábio Luís Baião da Silva (A75662)

João da Cunha Coelho (A74859)

Luís Miguel Moreira Fernandes (A74748)

12 de Junho de 2017

Resumo

Neste documento relata-se o processo de desenvolvimento da linguagem de programação imperativa simples (*LPIS*), motivada pelo segundo trabalho prático de *Processamento de Linguagens*, cujo objetivo é aumentar a experiência em engenharia de linguagens, em programação generativa (gramatical) e no desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe.

A construção da gramática tradutora que sustenta a codificação na linguagem, o desenvolvimento do compilador que gera o código *Assembly* para uma máquina de stack virtual e a apresentação de exemplos práticos que demonstram a utilização da linguagem compõem este relatório.

Conteúdo

Capítulo 1

Introdução

O problema em causa neste projeto é a definição de uma linguagem de programação imperativa simples que permita:

- a declaração e o manuseamento de variáveis atómicas do tipo inteiro, com as quais se possa realizar as habituais operações aritméticas, relacionais e lógicas;
- a declaração e manuseamento de variáveis estruturadas do tipo array (a 1 ou 2 dimensões) de inteiros, em relação às quais deve ser permitida a operação de indexação (índice inteiro);
- efetuar instruções algorítmicas básicas como a atribuição de expressões a variáveis;
- ler do standard input e escrever no standard output;
- efetuar instruções para controlo do fluxo de execução — condicionais e cíclicas — que possam ser aninhadas;
- definir e invocar subprogramas sem parâmetros, que possam retornar um resultado atómico (opcional).

Para a linguagem criada, em ambiente Linux, espera-se que se desenvolva um compilador com base na GIC criada e com recurso ao Gerador Yacc/ Flex. Este compilador deve gerar pseudo-código, *Assembly* da Máquina Virtual VM.

Capítulo 2

GIC da LPIS

```
GIC = < T = { DECLS, INSTRS, var, num, PRINT, READ, IF, ELSE, WHILE, cad },
        N = { Ling, Decls, Decl, Instrs, Instr, Atrib, Print, Read, CondS, IfCond,
              Ciclo, Oper, Cond, Valor, Array, ArrCabec, Atom, Var },
        S = Ling,
        P = {
              p1:   Ling -> DECLS Decls INSTRS Instrs
              p2:   Decls -> Decls ',' Decl
              p3:           | Decl
              p4:   Decl -> var
              p5:           | var '[' num ']'
              p6:           | var '[' num ']' '[' num ']'
              p7:   Instrs -> Instrs Instr
              p8:           | Instr
              p9:   Instr -> Atrib
              p10:          | Print
              p11:          | Read
              p12:          | CondS
              p13:          | Ciclo
              p14:  Atrib -> Var '=' Valor ';'
              p15:          | Var '=' Oper ';'
              p16:          | Array '=' Valor ';
              p17:          | Array '=' Oper ';'
              p18:          | Array '=' Cond ','
              p19:          | Var '=' '(' Cond '?' Valor ':' Valor ')' ';'
```

```
p20:   Print -> PRINT ':' Valor ';'
p21:           | PRINT ':' cad ';'
p22:   Read -> READ ':' Var ';'
p23:           | READ ':' Array ';'
p24:   CondS: IfCond
p25:           | IfCond ELSE '{' Instrs '}'
p26:   IfCond -> IF '(' Cond ')' '{'Instrs '}'
p27:   Ciclo -> WHILE '(' Cond ')' '{' Instrs '}'
p28:   Oper -> Valor '+' Valor
p29:           | Valor '-' Valor
p30:           | Valor '*' Valor
p31:           | Valor '/' Valor
p32:           | Valor '%' Valor
p33:   Cond -> Valor '=' '=' Valor
p34:           | Valor '!' '=' Valor
p35:           | Valor '<' '=' Valor
p36:           | Valor '>' '=' Valor
p37:           | Valor '<' Valor
p38:           | Valor '>' Valor
p39:           | Valor '&' '&' Valor
p40:   Valor -> Atom
p41:           | Array
p42:   Array -> ArrCabec
p43:           | ArrCabec '[' Atom ']'
p44:   ArrCabec -> Var '[' Atom ']'
p45:   Atom -> Var
p46:           | '-' Var
p47:           | num
p48:           | '-' num
p49:   Var -> var
} >
```

Capítulo 3

Gramática tradutora da LIPS

A gramática independente de contexto, explicitada na secção anterior, deu origem à gramática tradutora que aqui se apresenta.

3.1 Declaração de variáveis

As variáveis só podem tomar letras minúsculas e, para verificar re-declaração ou uso de variáveis não declaradas, é usada uma tabela de Hash que contém as variáveis declaradas. É no parsing da declaração de variáveis que esta tabela é preenchida - ao inserir uma variável, caso esta já exista na tabela a compilação termina. A verificação do uso de variáveis não declaradas é feita ao compilar as instruções, em que se confirma se a variável encontrada existe na tabela de Hash. Em caso negativo, o parsing termina.
Uma vez que é necessário guardar memória para cada variável, definiu-se uma estrutura a ser guardada como valor na tabela de Hash para cada variável. Esta estrutura contém dois valores: a posição em relação ao endereço gp e o tamanho de cada linha da matriz, ou seja, o número de colunas (caso a variável seja uma matriz). Para distinguir a posição de cada variável definiu-se também uma variável global que é incrementada à medida que são declaradas variáveis (1 caso seja uma variável "simples", n caso seja um array unidimensional, em que n é o tamanho do array, e n*m no caso da matriz em que n e m são os números de linhas e colunas, respetivamente).

3.2 Atribuições

É possível atribuir a variáveis: valores, operações ou condições. É ainda possível recorrer a um operador ternário. A atribuição pode ser feita quer a variáveis simples, quer a posições de arrays. Importa mencionar que um valor é um número ou o valor de uma variável, enquanto que uma operação é uma soma, subtração, multiplicação ou divisão de dois valores. Pode ainda ser calculado o resto da divisão inteira entre dois valores. Por sua vez, uma condição é um teste de igualdade, desigualdade, superioridade, etc., entre dois valores. Pode ser ainda calculado o valor lógico E entre dois valores (multiplicando os dois valores).

3.3 Escrita

Em cada PRINT apenas pode ser imprimido um valor ou uma cadeia de caracteres (cad).

3.4 Leitura

A leitura de valores pode ser feita para variáveis simples ou posições de arrays.

3.5 Controlo de fluxo

Para permitir efetuar instruções de controlo de fluxo é necessário criar labels diferentes entre cada uma das instruções. Para isso criou-se um contador que incrementa sempre que se adicionar uma instrução condicional ou cíclica.
No entanto, com esta solução é possível distinguir as labels se apenas existirem instruções encadeadas, já que, se existir alguma instrução de controlo de fluxo aninhada, quando se imprimir a label que se encontra no final do bloco esta vai ter um valor diferente da que está no início do bloco.
Para que seja possível incluir instruções aninhadas, construiu-se uma stack (LIFO) que guarda as labels que ainda não foram concluídas. Assim, sempre que se imprimir a primeira label, inclui-se essa label na stack e incrementa-se o seu valor. Quando chegar à altura de imprimir a segunda label, retira-se o seu valor da stack, pois a variável global pode ter sido alterada entretanto.

3.6 Assembly da Máquina Virtual VM

Para gerar o compilador, foram tomadas algumas decisões a nível do *Assembly*, tais como:

- A atribuição de um valor a uma variável consiste em colocar o valor no topo da stack e executar a instrução "storeg P", onde P é a posição (em relação ao registo gp) onde se encontra a variável. Para atribuir um valor a uma posição de um array ou de uma matriz, é necessário colocar o apontador da matriz e o índice da posição no topo da stack. Para isso executam-se as seguintes instruções:

  - **ARRAY**
    - pushgp
    - pushi P
    - padd
    - pushi I
  - **MATRIZ**
    - pushgp
    - pushi P
    - padd
    - pushi I
    - push T
    - mul
    - push J
    - (onde P é a posição do inicio do array (ou da matriz), I é o indice (das linhas, no caso de uma matriz), T é o tamanho de cada linha (no caso de uma matriz) e J é o indice das colunas);
    - Nota: o cálculo da posição de uma matriz é: I*T+J.

  De seguida coloca-se o valor a atribuir no topo da stack e executa-se a instrução "storen".

- Para colocar um valor numérico no topo da stack executa-se a instrução "pushi V"sendo V o valor númerico. A colocação de um valor de uma variável no topo da stack consiste em executar a instrução "pushg P". Para colocar o valor de uma posição de um array ou de uma matriz no topo da stack executa-se as seguintes instruções:

  - **ARRAY**
    - pushgp
    - pushi P
    - padd
    - pushi I
    - loadn
  - **MATRIZ**
    - pushgp
    - pushi P
    - padd
    - pushi I
    - push T
    - mul
    - push J
    - loadn

- Imprimir um valor consiste em colocar o valor no topo da stack e exectuar a instrução "writei"de seguida, enquanto que para imprimir uma cadeia de caracteres é necessário fazer push da cadeia e executar o comando "writes".

- Uma vez que a linguagem apenas suporta inteiros (excetuando o caso de imprimir cadeias de caracteres), a leitura do standard input é feita com a instrução "read"seguida da instrução "atoi". Para guardar o valor numa variável ou numa posição de um array ou de uma matriz, o procedimento é o mesmo das atribuições.

- O cálculo de uma qualquer operação consiste em colocar dois valores no topo da stack e executar a instrução pretendida (add, sub, mul, div, mod). A atribuição do valor resultante a uma variável é feita tal como explicado anteriormente.

- Calcular uma condição segue a mesma lógica do cálculo das operações, variando apenas nas instruções usadas (equal, equal not, infeq, supeq, inf, sup). É ainda possível calcular o valor lógico E usando a instrução "mul".

- Para permitir efetuar instruções de controlo de fluxo é necessário criar labels diferentes entre cada uma das instruções. Para isso criou-se um contador que incrementa sempre que se adicionar uma instrução condicional ou cíclica. No entanto com esta solução é possível distinguir as labels se apenas existirem instruções encadeadas, já que se existir alguma instrução de controlo de fluxo aninhada quando se imprimir a label que se encontra no final do bloco esta vai ter um valor diferente da que está no inicio do bloco.
  Para que seja possível incluir instruções aninhadas construiu-se uma stack (LIFO) que guarda as labels que ainda não foram concluídas. Assim, sempre que se imprimir a primeira label, inclui-se essa label na stack e incrementa-se o seu valor. Quando chegar à altura de imprimir a segunda label retira-se o seu valor da stack, pois a variável global pode ter sido alterada entrentanto.
  Desta forma, no caso de uma instrução condicional coloca-se a instrução "jz labelI1"imediatamente depois de testar a condição, sendo I1 a variável global que contem o número da label, e coloca-se I1 na stack, incrementando-a. Se não existir bloco ELSE coloca-se a instrução "labelI1:"depois de executar o bloco, sendo I o valor que está na cabeça da stack, retirando-o.
  Caso haja um bloco ELSE inclui-se as instruções "jump labelI2"e "labelI1"a seguir ao bloco do IF e antes do bloco de instruções do ELSE, sendo I1 obtido da stack e I2 a variável global (como sempre esta variável é incrementada depois decolocada na stack). No final do bloco ELSE coloca-se a instrução "labelI2:"sendo I2 obtido da stack.

- Nas instruções ciclicas, coloca-se a instrução "labelI1:"antes de testar a condição, sendo o I1 o valor obtido da variável global (sendo colocada na stack e incrementada de seguida). Imediatamente depois de calcular a condição de paragem é colocada a instrução "jz labelI2"(mais uma vez é colocada na stack e incrementada). No final das instruções pertencentes ao bloco WHILE são adicionadas as instruções "jump labelI1"e "labelI2:", onde I1 e I2 são obtidos da stack.

NOTAS:
Nas produções relativas aos não terminais CondS e Array numa primeira abordagem colocou-se as seguintes produções:

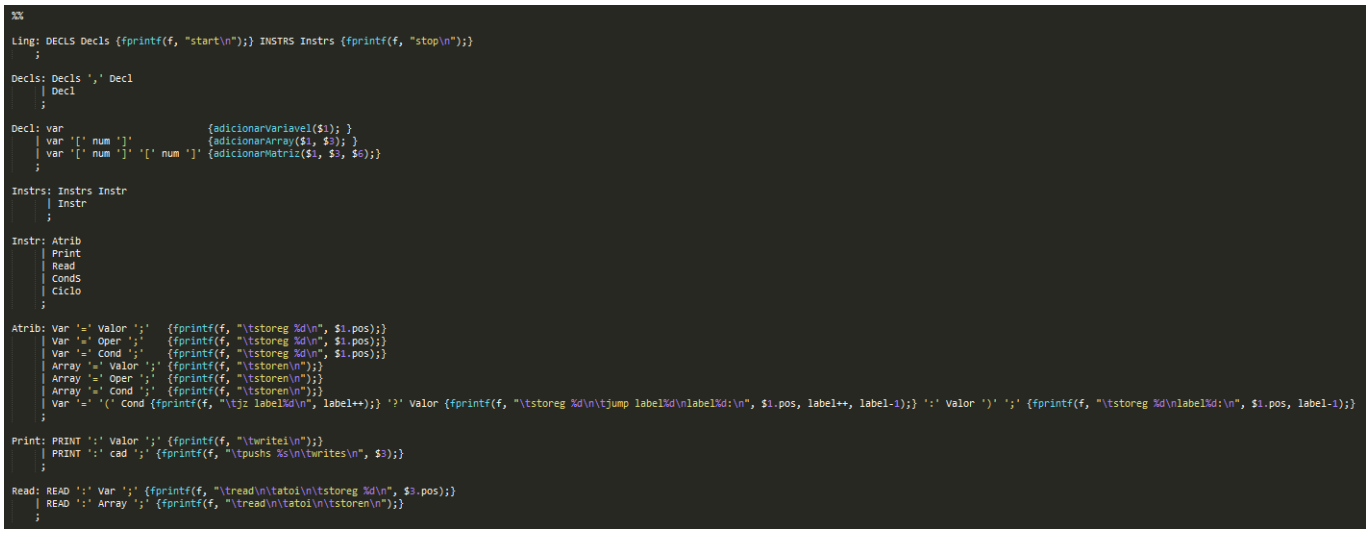
```
CondS: IF '(' Cond ')' { fprintf(f, "\tjz label%d\n", label); push(); }
   '{' Instrs '}' { fprintf(f, "label%d:\n", pop()); }

     | IF '(' Cond ')' { fprintf(f, "\tjz label%d\n", label); push(); }
       '{' Instrs '}' ELSE { fprintf(f, "\tjump label%d\n", label);
              fprintf(f, "label%d:\n", pop());
         push(); } '{' Instrs '}' { fprintf(f, "label%d:\n", pop()); }

Array: Var {fprintf(f, "\tpushgp\n\tpushi %d\n\tpadd\n", $1.pos);} '[' Atom ']'

     | Var {fprintf(f, "\tpushgp\n\tpushi %d\n\tpadd\n", $1.pos);} '[' Atom ']'
           { fprintf(f, "\tpushi %d\n\tmul\n", $1.tamL);} '[' Atom ']' {fprintf(f, "\tadd\n"); }
```

Uma vez que esta abordagem gerava conflitos passou-se as partes iniciais que eram comuns às duas alternativas de cada não terminal para uma outra produção. Desta forma eliminou-se os conflitos.

3.7 Exposição da GT

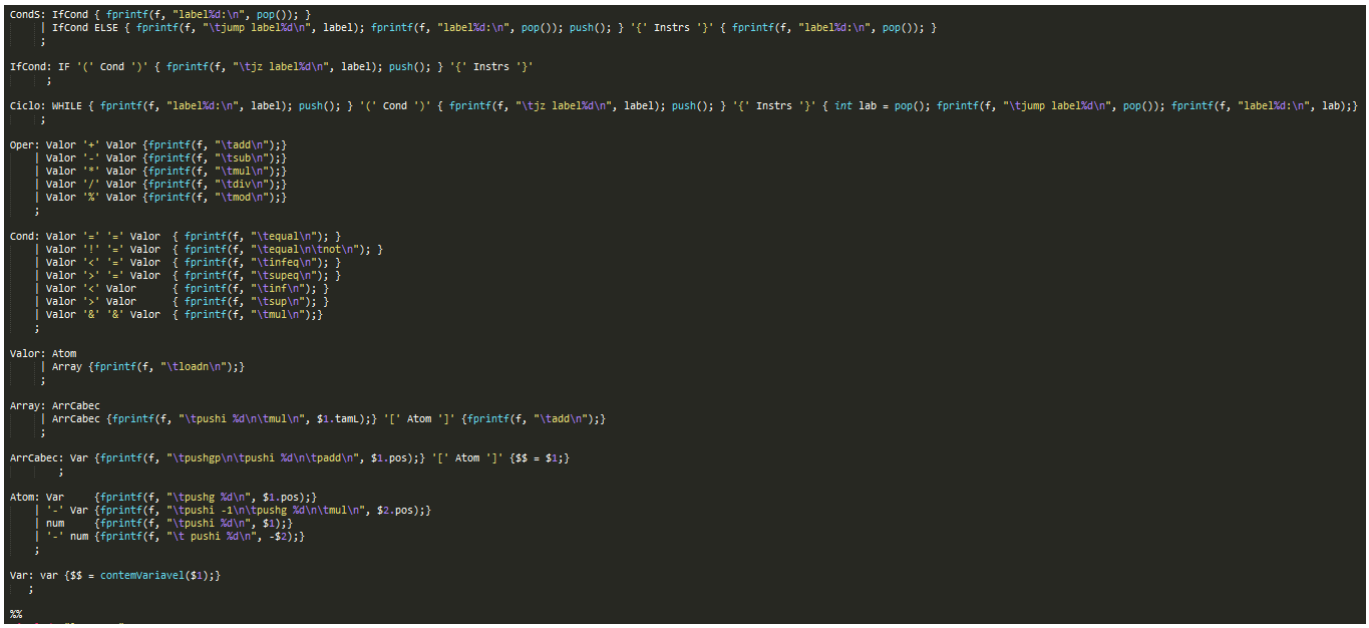
```
%{
    #include <glib.h>
    #include <stdio.h>

    void yyerror(char*);
    int yylex();

    void adicionarMatriz(char*, int, int);
    void adicionarArray(char*, int);
    void adicionarVariavel(char*);

    GHashTable *vars;
    int pos, label;
    FILE *f;

    typedef struct{int pos, tamL;} DadosVar;
    DadosVar contemVariavel(char*);

    typedef struct stack{
        int v;
        struct stack *prox;
    } *Stack;
    Stack s;
    void push();
    int pop();
%}

%token DECLS INSTRS var num PRINT READ IF ELSE WHILE cad

%union{int n; char *v; DadosVar d;}

%type<v> var cad
%type<n> num
%type<d> Var ArrCabec
```

Figura 3.1: Declarações e estruturas de dados.



```
%%

Ling: DECLS Decls {fprintf(f, "start\n");} INSTRS Instrs {fprintf(f, "stop\n");}
    ;

Decls: Decls ',' Decl
     | Decl
     ;

Decl: var                       {adicionarVariavel($1); }
    | var '[' num ']'           {adicionarArray($1, $3); }
    | var '[' num ']' '[' num ']' {adicionarMatriz($1, $3, $6);}
    ;

Instrs: Instrs Instr
      | Instr
      ;

Instr: Atrib
     | Print
     | Read
     | Conds
     | Ciclo
     ;

Atrib: Var '=' Valor ';'   {fprintf(f, "\tstoreg %d\n", $1.pos);}
     | Var '=' Oper ';'    {fprintf(f, "\tstoreg %d\n", $1.pos);}
     | Var '=' Cond ';'    {fprintf(f, "\tstoreg %d\n", $1.pos);}
     | Array '=' Valor ';' {fprintf(f, "\tstoren\n");}
     | Array '=' Oper ';'  {fprintf(f, "\tstoren\n");}
     | Array '=' Cond ';'  {fprintf(f, "\tstoren\n");}
     | Var '=' '(' Cond {fprintf(f, "\tjz label%d\n", label++);} '?' Valor {fprintf(f, "\tstoreg %d\n\tjump label%d\nlabel%d:\n", $1.pos, label++, label-1);} ':' Valor ')' ';' {fprintf(f, "\tstoreg %d\nlabel%d:\n", $1.pos, label-1);}
     ;

Print: PRINT ':' Valor ';' {fprintf(f, "\twritei\n");}
     | PRINT ':' cad ';' {fprintf(f, "\tpushs %s\n\twrites\n", $3);}
     ;

Read: READ ':' Var ';' {fprintf(f, "\tread\n\tatoi\n\tstoreg %d\n", $3.pos);}
    | READ ':' Array ';' {fprintf(f, "\tread\n\tatoi\n\tstoren\n");}
    ;
```

Figura 3.2: Gramática tradutora (1).



```
Conds: IfCond { fprintf(f, "label%d:\n", pop()); }
     | IfCond ELSE { fprintf(f, "\tjump label%d\n", label); fprintf(f, "label%d:\n", pop()); push(); } '{' Instrs '}' { fprintf(f, "label%d:\n", pop()); }
     ;

IfCond: IF '(' Cond ')' { fprintf(f, "\tjz label%d\n", label); push(); } '{' Instrs '}'
      ;

Ciclo: WHILE { fprintf(f, "label%d:\n", label); push(); } '(' Cond ')' { fprintf(f, "\tjz label%d\n", label); push(); } '{' Instrs '}' { int lab = pop(); fprintf(f, "\tjump label%d\n", pop()); fprintf(f, "label%d:\n", lab);}
     ;

Oper: Valor '+' Valor {fprintf(f, "\tadd\n");}
    | Valor '-' Valor {fprintf(f, "\tsub\n");}
    | Valor '*' Valor {fprintf(f, "\tmul\n");}
    | Valor '/' Valor {fprintf(f, "\tdiv\n");}
    | Valor '%' Valor {fprintf(f, "\tmod\n");}
    ;

Cond: Valor '=' '=' Valor  { fprintf(f, "\tequal\n"); }
    | Valor '!' '=' Valor  { fprintf(f, "\tequal\n\tnot\n"); }
    | Valor '<' '=' Valor  { fprintf(f, "\tinfeq\n"); }
    | Valor '>' '=' Valor  { fprintf(f, "\tsupeq\n"); }
    | Valor '<' Valor      { fprintf(f, "\tinf\n"); }
    | Valor '>' Valor      { fprintf(f, "\tsup\n"); }
    | Valor '&' '&' Valor  { fprintf(f, "\tmul\n");}
    ;

Valor: Atom
     | Array {fprintf(f, "\tloadn\n");}
     ;

Array: ArrCabec
     | ArrCabec {fprintf(f, "\tpushi %d\n\tmul\n", $1.tamL);} '[' Atom ']' {fprintf(f, "\tadd\n");}
     ;

ArrCabec: Var {fprintf(f, "\tpushgp\n\tpushi %d\n\tpadd\n", $1.pos);} '[' Atom ']' {$$ = $1;}
        ;

Atom: Var     {fprintf(f, "\tpushg %d\n", $1.pos);}
    | '-' Var {fprintf(f, "\tpushi -1\n\tpushg %d\n\tmul\n", $2.pos);}
    | num     {fprintf(f, "\tpushi %d\n", $1);}
    | '-' num {fprintf(f, "\t pushi %d\n", -$2);}
    ;

Var: var {$$ = contemVariavel($1);}
   ;

%%
```

Figura 3.3: Gramática tradutora (2).

```c
#include "lex.yy.c"

void push (){
    Stack h = malloc(sizeof(struct stack));
    h->v = label++;
    h->prox = s;
    s = h;
}

int pop (){
    int n = s->v;
    Stack aux = s->prox;
    free(s);
    s = aux;
    return n;
}

DadosVar contemVariavel(char *variavel){
    DadosVar *posicao = g_hash_table_lookup(vars, variavel);
    if(!posicao){
        yyerror(variavel);
        return *posicao;
    }
    return *posicao;
}

void adicionarMatriz(char *variavel, int n, int m){
    DadosVar *posicao = malloc(sizeof(DadosVar));
    posicao->pos = pos;
    posicao->tamL = m;
    if(!g_hash_table_insert(vars, variavel, posicao)){
        yyerror(variavel);
    }
    pos+=n*m;
    fprintf(f, "\tpushn %d\n", n*m);
}
```

Figura 3.4: Funções auxiliares criadas.

```c
void adicionarArray(char *variavel, int n){
    DadosVar *posicao = malloc(sizeof(DadosVar));
    posicao->pos = pos;
    if(!g_hash_table_insert(vars, variavel, posicao)){
        yyerror(variavel);
    }
    pos+=n;
    fprintf(f, "\tpushn %d\n", n);
}

void adicionarVariavel(char *variavel){
    DadosVar *posicao = malloc(sizeof(DadosVar));
    posicao->pos = pos;
    if(!g_hash_table_insert(vars, variavel, posicao)){
        yyerror(variavel);
    }
    pos++;
    fprintf(f, "\tpushi 0\n");
}

void yyerror(char *s){
    fprintf(stderr, "%d: %s\n\t%s\n", yylineno, yytext, s);
}

int main(int argc, char* argv[]){
    f = stdout;
    if(argc == 2){
        yyin = fopen(argv[1], "r");
    }
    else if (argc == 3){
        yyin = fopen(argv[1], "r");
        f = fopen(argv[2], "w");
    }
    vars = g_hash_table_new(g_str_hash, g_str_equal);
    pos = 0;
    label = 0;
    s = NULL;
    yyparse();
    return 0;
}
```

Figura 3.5: Funções auxiliares e main.

3.8 Analisador léxico da gramática

```
%option noyywrap yylineno

%%

DECLS                       {return DECLS;}
INSTRS                      {return INSTRS;}
PRINT                       {return PRINT;}
READ                        {return READ;}
IF                          {return IF;}
ELSE                        {return ELSE;}
WHILE                       {return WHILE;}
[,=;:(){}+*/%!<>?&]|\[|\]|- {return yytext[0];}
[a-z]+                      {yylval.v = strdup(yytext); return var;}
-?[0-9]+                    {yylval.n = atoi(yytext); return num;}

\"([^"]|\\\")*\"            {yylval.v = strdup(yytext); return cad;}

[ \n\t]                     {}

%%
```

Capítulo 4

Apresentação de exemplos de utilização

A extensão escolhida para a LPIS foi *.ex*.
O enunciado do projeto propõe a apresentação de seis exemplos, que se mencionam a seguir juntamente com a sua resolução em LPIS:

- ler 4 números e dizer se podem ser os lados de um quadrado:

```
DECLS
x, y, i
INSTRS
READ: x;
i = 0;
WHILE (i < 3){
    READ: y;
    IF (y == x){
        i = i + 1;
    }
    ELSE{
        PRINT: "Nao é quadrado!";
        i = 4;
    }
}
IF (i == 3){
    PRINT: "É quadrado!";
}
```

- ler um inteiro N, depois ler N números e escrever o menor deles:

```
DECLS
n, m, v
INSTRS
READ: n;
READ: m;
n = n - 1;
WHILE(n > 0){
    READ: v;
    IF(v < m){
        m = v;
    }
    n = n - 1;
}
PRINT: "Menor: ";
PRINT: m;
```

- ler N (constante do programa) números e calcular e imprimir o seu produtório:

```
DECLS
n, p, v
INSTRS
READ: n;
p = 1;
WHILE(n > 0){
    READ: v;
    p = p * v;
    n = n - 1;
}
PRINT: "Produtorio: ";
PRINT: p;
```

- contar e imprimir os números impares de uma sequência de números naturais:

```
DECLS
a[10], i, c, r
INSTRS
i = 0;
WHILE(i < 10){
    a[i] = i;
    i = i + 1;
}
i = 0;
c = 0;
WHILE(i < 10){
    r = a[i] % 2;
    IF(r == 1){
        PRINT: "Impar: ";
        PRINT: a[i];
        PRINT: " \n";
        c = c + 1;
    }
    i = i + 1;
}
PRINT: "Numero de impares: ";
PRINT: c;
```

- ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas:

```
DECLS
a[5], i, v, aux, j, m
INSTRS
i = 0;
WHILE(i < 5){
    READ: v;
    a[i] = v;
    i = i + 1;
}
i = 0;
WHILE (i < 5){
    j = i + 1;
    m = i;
    WHILE(j < 5){
        IF(a[j] > a[m]){
            m = j;
        }
        j = j + 1;
    }
    aux = a[i];
    a[i] = a[m];
    a[m] = aux;
    i = i + 1;
}
i = 0;
PRINT: "Ordem descrescente\n";
WHILE(i < 5){
    PRINT: a[i];
    PRINT: " \n";
    i = i + 1;
}
```

- ler e armazenar N números num array; imprimir os valores por ordem inversa:

```
DECLS
a[5], i
INSTRS
i = 0;
WHILE(i < 5){
    READ: a[i];
    i = i + 1;
}
i = 5 - 1;
PRINT: "Ordem inversa\n";
WHILE(i >= 0){
    PRINT: a[i];
    PRINT: " \n";
    i = i - 1;
}
```

Capítulo 5

Conclusão

Concluído o projeto, destaca-se o sucesso na criação de uma linguagem de programação imperativa simples que faculta funcionalidades para a obtenção dos resultados pretendidos nos programas sugeridos no enunciado.
Este trabalho foi importante para cimentar os conhecimentos sobre YACC e FLEX, bem como relembrar conhecimentos sobre análise de código em linguagem máquina *Assembly* e posterior implementação nesta mesma linguagem.
Quanto a trabalho futuro, seria lógico tentar estender a linguagem de forma a esta ficar mais completa, sendo possível outros tipos de dados e não apenas inteiros, ser possível a criação de estruturas de dados, entre outras que são conhecidas das linguagens de programação imperativas mais conhecidas.