Processamento de Linguagens e Compiladores (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Luís Almeida
A84180

João Pedro Antunes
A86813

21 de janeiro de 2021

Resumo

O presente documento detalha o processo de construção de um compilador de uma linguagem de programação a que chamamos "**SIP** - Systematic Integer Programming". Esta linguagem de programação permite o uso de variáveis do tipo inteiro, bem como expressões aritméticas, relacionais e também o uso de *loops*. Apresenta-se e explica-se a gramática de *parsing* elaborada, assim como a deteção de erros semânticos e apresentam-se alguns exemplos de funcionamento do mesmo.

Conteúdo

Capítulo 1

Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens e Compiladores foi nos proposta a construção de um compilador de uma linguagem de programação criada por nós, de forma a consolidar os conhecimentos adquiridos nas aulas sobre *parsing*, *yacc* e gramáticas independentes de contexto. O objetivo deste relatório é fornecer uma explicação detalhada do processo de construção do compilador, bem como das decisões tomadas para resolver as várias dificuldades que encontramos ao longo da resolução do problema. O presente documento está dividido em:

Introdução - Breve introdução ao problema proposto

Análise e Especificação - Apresentação da linguagem "**SIP**"e requisitos de compilação, bem como o *assembly* da VM utilizada

Desenho da Resolução - Estratégia adotada para a construção do compilador pretendido

Codificação e Testes - Implementação da estratégia adotada e apresentação de alguns testes

Conclusão - Análise crítica do compilador criado e considerações sobre a expansão futura do mesmo

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

É nos pedida a construção de um compilador de uma linguagem que permita ao utilizador trabalhar com variáveis do tipo inteiro, fazendo diversas operações com as mesmas. Comecemos por definir esta linguagem.

2.2 Especificação dos Requisitos

2.2.1 Linguagem SIP

De acordo com o enunciado, a nossa linguagem de programação deve-nos permitir trabalhar com variáveis do tipo inteiro e com *arrays* destas variáveis, tendo obrigatoriamente de ser declaradas no início do programa. É nos também pedido que seja possível executar operações aritméticas e relacionais com as mesmas, assim como ciclos *for-do, while-do, repeat-until*. Sendo assim, podemos pensar na estrutura de um programa em como sendo:

{ DECLARAÇÃO DE VARIÁVEIS }
{ BLOCOS DE CÓDIGO (funções) }
{ MAIN (função principal que é executada no ínicio do programa) }

A declaração de variáveis terá de admitir a declaração singular de uma variável, bem como a declaração de várias variáveis por linha. Como tal, o bloco inicial de declaração de variáveis terá o seguinte aspeto:

```
DECLARE{
    x,y,z
    array[10]
}
```

As funções de um programa em poderão retornar um valor do tipo inteiro ou então não retornar nada. Terão de ser declaradas antes da função MAIN. Um exemplo de uma função seria:

```
auxiliarFunction{
```

```
        x = 3
        y = 2
        return x + y
}
```

A função principal irá controlar o fluxo de execução do programa e será a primeira a ser executada. Terá o seguinte aspeto:

```
MAIN{
    z = 5
    auxiliarFunction()
    write(x+y+z)
}
```

2.2.2 Assembly da VM

Tendo a linguagem definida, vamos agora olhar para o código *assembly* que terá de ser gerado pelo compilador para posterior execução na VM disponibilizada pelo professor.

1. Alocação de variáveis: Conforme referido, é necessário alocar variáveis do tipo inteiro e arrays das mesmas. Como tal, as instruções:

   ```
   PUSHI 0
   PUSHN 100
   ```

   São pertinentes para alocar uma variável inteira singular e um array de 100 variáveis inteiras, respetivamente. Para finalizar a alocação de memória a variáveis usa-se a instrução:

   ```
   START
   ```

   Para afetarmos o valor do *sp* ao *fp*.

2. Atribuição de valores a variáveis: Para atribuirmos um valor a uma variável é necessário armazena-lo no endereço de memória reservado à variável em questão. Esta operação corresponde à instrução:

   ```
   STOREG 3
   ```

   Para armazenar o valor do topo da *stack* no 3º endereço de memória das variáveis globais.

3. Operações Aritméticas e Relacionais: Para executarmos as várias operações de multiplicação, adição, igualdade, inferioridade, ... necessárias ao funcionamento da linguagem podemos utilizar as instruções:

   ```
   ADD
   SUB
   MUL
   DIV
   MOD
   ```

```
NOT
INF
INFEQ
SUP
SUPEQ
EQUAL
```

4. Controlo de fluxo: É também necessário que os nossos programas possam fazer comandos

   ```
   if <condição> then <código> else <código>
   ```

   Como tal, podemos criar etiquetas para cada trecho de código a executar consoante o teste da condição. Um exemplo seria:

   ```
   <condição>
   JZ E1
   <código a executar caso a condição se verifique>
   JUMP E2
   E1:
   <código a executar caso a condição não se verifique>
   E2:
   <resto do código do programa>
   ```

5. Ciclos: Para podermos utilizar ciclos nos nossos programas , teremos de criar uma etiqueta no código que sinalizará as instruções a executar caso a condição do ciclo seja verdadeira. Um exemplo de um ciclo seria:

   ```
   E0:
   PUSHI 2
   PUSHI 3
   ADD
   WRITEI
   <código que determina o valor da condição>
   NOT
   JZ E0
   ```

   Onde E0 é a etiqueta que sinaliza o início do ciclo. Caso a condição seja verdadeira é dado um salto para o início do ciclo, caso contrário o programa prossegue a sua execução.

6. Armazenamento em Arrays: Para podermos armazenar um valor numa determinada posição num array temos de carregar o seu endereço na memória global para o topo da *stack*, empilhar o valor da posição que queremos aceder e empilhar o valor que queremos armazenar. Posto isto utilizamos a instrução:

   ```
   STOREN
   ```

7. Aceder ao valor de variáveis: Para acedermos ao valor das variáveis do programa é necessário utilizar a instrução:

```
PUSHG 2
```

   Assim empilhamos o valor da variável cujo endereço na memória global é 2 para o topo da *stack*. Caso a variável seja um array, teremos de utilizar:

```
PUSHGP
PUSHI 2
PADD
PUSHI 3
LOADN
```

   Para podermos aceder à 3º posição do array que se situa na posição 2 da memória global.

8. Funções: Torna-se necessário atribuir uma etiqueta a cada função do programa para podermos saltar para a mesma sempre que é chamada. Sendo assim, um exemplo de uma função seria:

```
E1:
PUSHG 0
PUSHG 1
ADD
STOREL -1
RETURN
```

   Usa-se a instrução STOREL -1 para conservar o topo da *stack* no fim da execução da função, garantindo assim que a função retorna um valor.

Capítulo 3

Concepção/desenho da Resolução

3.1 Gramática de *Parsing*

Precisamos agora de definir uma gramática que reconheça a linguagem utilizando o *yacc*. Terá também de ser capaz de acionar as ações semânticas em linguagem *C* que irão gerar as instruções *assembly* correspondentes ao comando reconhecido. No capítulo anterior foi apresentada a estrutura de um programa em linguagem . Apresentamos agora o axioma inicial da gramática construída:

```
    Programa: Header Blocos Main { asprintf(&res,"%s%sSTOP\n%s\n",$1,$3,$2); }
        ;
```

Utiliza-se a função *asprintf* para podermos construir a *string* que corresponde às instruções do programa incrementalmente, à medida que novos comandos são reconhecidos. Header corresponde ao bloco onde a declaração das variáveis é feita, Blocos corresponde às funções do programa e Main corresponde à função principal do programa.

```
    Header: DECLARE '{' ListIds '}'
      ;

    ListIds: ListIds ListId   //reconhecimento de várias linhas de declarações
           | ListId
           ;

ListId: Identificador
      | ListId ',' Identificador //reconhecimento de uma lista de declarações
      ;

Identificador: ID             //reconhecimento de uma variável singular
 | ID'['INT']'   //reconhecimento de um array
             | ID'['INT']''['INT']' //reconhecimento de um array 2D
 ;
```

Com estas regras de derivação preenchem-se os requisitos especificados no capítulo anterior, a gramática reconhece variáveis singulares, arrays de 1 dimensão e arrays de 2 dimensões. Torna-se necessário armazenar a informação relativa a cada variável numa tabela de *hash* para podermos monitorizar a "vida"da variável

ao longo da execução do programa, armazenando informação sobre a sua posição na memória e prevenindo redeclarações de variáveis.

```
Blocos:                    //um programa pode não ter funções
      | Bloco Blocos      //reconhecimento de 1 ou mais funções
      ;

Bloco: ID '{' Cmds '}'  //reconhecimento de uma função que não retorna um valor
    |  ID '{' Cmds RETURN ExprCmpInt '}'
     ;
```

Neste momento já conseguimos reconhecer funções que não retornam um valor e funções que retornam um valor. Mais uma vez, torna-se também necessário armazenar a informação relativa a cada função numa tabela de *hash*, armazenando o valor da sua etiqueta, o seu tipo (se retorna um valor ou não) e o seu nome, para impedir redeclarações de funções.

```
    Main: MAIN '{' Cmds '}'  //reconhecimento da MAIN
    ;

    Cmds : Cmd     // reconhecimento de 1 ou mais comandos
     | Cmds Cmd
     ;

Cmd : Atrib         //reconhecimento de uma atribuição
    | Read   //reconhecimento de uma leitura do stdin
    | Write  //reconhecimento de uma escrita no stdout
    | Condic  //reconhecimento de um controlo de fluxo
    | Repeat   //reconhecimento de um repeat-loop
    | While         //reconhecimento de um while-loop
    | ForL          //reconhecimento de um for-loop
    | SKIP          //reconhecimento de um skip (não gera instruções)
    | FuncCall      //reconhecimento de uma chamada de uma função
    ;
```

Apresentam-se então a regra que nos permite reconhecer a função principal e as regras que nos permitem reconhecer cada comando especificado no capítulo anterior. Cada um destes comandos irá corresponder a um conjunto de instruções em *assembly* que será gerado pelas ações semânticas que o irão acompanhar.

```
Condic: IF ExprCmpInt THEN  '{'  Cmds  '}'  ELSE '{'  Cmds  '}'
      ;

Repeat: REPEAT '{' Cmds '}' UNTIL ExprCmpInt
      ;

While: WHILE ExprCmpInt '{' Cmds '}'
     ;
```

```
ForL: FOR '(' Cmd ',' ExprCmpInt ',' Cmd ')' '{' Cmds '}'
    ;

Atrib : ID    '='    ExprCmpInt
      | ID'['ExprInt']' '=' ExprCmpInt
      | ID'['ExprInt']''['ExprInt']' '=' ExprCmpInt
      ;


Write: WRITE '(' ExprCmpInt ')'
     ;


Read: READ '(' ID ')'
    | READ '(' ID '['ExprInt']' ')'
    | READ '(' ID '['ExprInt']' '['ExprInt']' ')'
    ;

FuncCall: ID'('')'
        ;
```

Estas serão as regras que permitirão reconhecer cada um dos comandos admitidos na linguagem . De seguida apresentam-se as regras que nos permitem reconhecer operações lógicas e aritméticas com variáveis do tipo inteiro.

```
    ExprCmpInt: ExprInt
  | ExprCmpInt EQ ExprCmpInt    // Permite reconhecer a operação p == q
          | ExprCmpInt NE ExprCmpInt    // Permite reconhecer a operação p != q
          | ExprCmpInt LT ExprCmpInt    // Permite reconhecer a operação p < q
          | ExprCmpInt LE ExprCmpInt    // Permite reconhecer a operação p <= q
          | ExprCmpInt GT ExprCmpInt    // Permite reconhecer a operação p > q
          | ExprCmpInt GE ExprCmpInt    // Permite reconhecer a operação p >= q
          | NOT ExprCmpInt       // Permite reconhecer a operação !p
          | ExprCmpInt OR ExprCmpInt    // Permite reconhecer a operação p || q
          | ExprCmpInt AND ExprCmpInt   // Permite reconhecer a operação p && q
          ;

ExprInt: Termo
       | ExprInt '+' Termo
       | ExprInt '-' Termo
       ;

Termo : Fator
    | Termo '*' Fator
    | Termo '/' Fator
    | Termo '%' Fator
    ;
```

```
Fator : INT       //Permite reconhecer a utilização de um valor inteiro
    | '-' INT     //Permite reconhecer a utilização de um valor inteiro negativo
    | TRUE   //Permite reconhecer a utilização do valor lógico true
    | FALSE       //Permite reconhecer a utilização do valor lógico false
    | ID'(')'    //Permite reconhecer um valor inteiro calculado por uma função
    | ID    //Permite reconhecer a utilização de um valor de uma variável
    | ID '['ExprInt']' //Permite reconhecer a utilização de um valor de um array
    | ID '['ExprInt']''['ExprInt']' //Permite reconhecer a utilização de um valor de um array 2D
    | '(' ExprCmpInt ')' //Permite reconhecer a prioridade de uma expressão
    ;
```

As ações semânticas de cada regra de derivação assim como a gramática completa poderão ser consultadas no anexo.

3.2 Erros Semânticos

Para termos em conta a possibilidade de alertar o utilizador da existência de erros semânticos no código a ser compilado, podemos lesmente ter uma variável global que nos indica se, no estado atual, existe algum erro. Assim podemos altera-la consoante o erro que tivermos e emitir uma mensagem personalizada para cada um.

3.3 Estruturas de Dados

Na secção anterior foi referida a necessidade de duas tabelas de *hash* para armazenar informação respetiva a variáveis e a funções. Podemos então definir 3 estruturas da seguinte forma:

```
    typedef struct var_list{
char* name;
int pos;         //posição da variável em memória
int type;        //tipo da variável (inteiro,array,array2D)
int line;        //linha da variável (relevante para os arrays 2D)
struct var_list* prox;
}*VAR_LIST;

typedef struct fun_list{
char* name;
int tag;        // etiqueta da função no código
int type;       // tipo da função (void ou retorna inteiro)
struct fun_list* prox;
}*FUN_LIST;

typedef struct hash_table{
VAR_LIST table[HASHSZ];
FUN_LIST fun_table[HASHSZ];
int used;  //monitoriza a posição na memória da próxima variável
}*HASH_TABLE;
```

Desta forma conseguimos armazenar toda a informação referida.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Nesta secção vamos apresentar alguns exemplos de ações semânticas e do tratamento de erros. Conforme referido no capítulo anterior, sempre que é declarada uma variável torna-se necessário armazená-la na tabela de *hash* e gerar as instruções *assembly* correspondentes a este processo. Como tal, escrevemos a função aloca() que armazena a variável declarada na tabela de *hash*

```
void aloca(char**instruction,HASH_TABLE hash_table, char* var_name,
int type, int lin, int col, int *flagError ){

int key = (int) hashFun(var_name);
VAR_LIST elem = lookup(hash_table,var_name);

if( elem != NULL){
*flagError = REALLOC;
return;
}


VAR_LIST allocate = new_list(var_name, hash_table->used);
allocate->type = type;
allocate->line = lin;
allocate->prox = hash_table->table[key];
hash_table->table[key] = allocate;
hash_table->used += lin*col;

asprintf(instruction,"PUSHN %d\n", lin*col);
}
```

Esta função será invocada sempre que é reconhecida a declaração de uma variável da seguinte forma:

```
    Identificador: ID
    {aloca(&$$,tabID,$1,INTEG,1,1,&flagError); if (flagError) return;}
```

Sempre que quisermos usar o valor de uma variável para, por exemplo, executar operações aritméticas, é

necessário saber qual a posição na memória da variável em questão. Ora, como temos esta informação armazenada na tabela de *hash*, escrevemos a seguinte função:

```
void fetch_var(char** instruction, char* var_name, int type,
    char* inst_frstindex, char* inst_scndindex, HASH_TABLE tabID,
    int *flagError){

VAR_LIST elem = lookup(tabID,var_name);

if (elem == NULL){
*flagError = NOALLOC;
return;
}

if(elem->type != type){
*flagError = TYPDIFF;
}

switch (type) {

case ARRAY:
asprintf(instruction,"PUSHGP\nPUSHI %d\nPADD\n%sLOADN\n",elem->pos,inst_frstindex);
break;

case ARRTD:
asprintf(instruction, "PUSHGP\nPUSHI %d\nPADD\n%sPUSHI
%d\nMUL\n%sADD\nLOADN\n", elem->pos, inst_scndindex,
elem -> line, inst_frstindex);
break;

default:
asprintf(instruction, "PUSHG %d\n", elem->pos);
break;
}
}
```

Assim, sempre que estivermos a executar operações aritméticas com variáveis, invocamos esta função.

```
    Fator: ....
           ....
           | ID  { fetch_var(&$$,$1,INTEG,"","",tabID,&flagError);
           if (flagError) return;}
```

As funções seguem um procedimento análogo ao das variáveis, sempre que são declaradas são também alocadas numa tabela de *hash* e também é armazenada a *label* que identifica a zona de código respetiva a cada função. Apresenta-se de seguida a ação semântica que gera o código *assembly* referente a uma invocação de função em que o resultado da mesma é ignorado:

```
    FuncCall: ID'('')' {asprintf(&$$,"PUSHI 0\nPUSHA E%d\nCALL\n
```

```
    POP 1\n", fetch_fun($1,VOID,tabID,&flagError));}
        ;
```

Anteriormente foi referido de forma breve o mecanismo de deteção de erros semânticos do compilador. Apresentam-se de seguida os tipos de erros possíveis:

```
    void errorHand(){
  char*error = "Failed, Semantic Error: ";
  switch(flagError){
    case REALLOC:
    printf("%s VARIABLE REALLOCATED, line: %d\n", error, yylineno-1);
    break;

    case NOALLOC:
    printf("%s VARIABLE NOT ALLOCATED, line: %d\n", error, yylineno-1);
    break;

    case TYPDIFF:
    printf("%s INCOMPATIBLE TYPES, line: %d\n", error, yylineno-1);
    break;

    case NORETRN:
    printf("%s FUNCTION WITH NO RETURN CALLED IN EXPRESSION, line: %d\n", error, yylineno-1);
    break;

    case NODEFIN:
    printf("%s FUNCTION NOT DEFINED CALLED, line: %d\n", error, yylineno-1);
    break;

    case REDEFIN:
    printf("%s FUNCTION REDEFINED, line: %d\n", error, yylineno-1);
    break;
  }
}
```

4.2 Testes realizados e Resultados

Apresentam-se a seguir alguns testes feitos (valores introduzidos) e os respectivos resultados obtidos:
Função quadrado

```
    declare{
i,curr,l
}

main{

read(l)
```

```
curr = 1

for(i = 0, i < 3 and curr == 1 , i = i + 1){
read(curr)
}

if i == 3 then{
write(1)
}
else{
write(0)
}

}
```

Código *assembly* gerado:

```
    PUSHN 1
PUSHN 1
PUSHN 1
START
READ
ATOI
STOREG 2
PUSHG 2
STOREG 1
PUSHI 0
STOREG 0
E0:
PUSHG 0
PUSHI 3
PUSHG 1
PUSHG 2
EQUAL
MUL
INF
JZ E1
READ
ATOI
STOREG 1
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP E0
E1:
PUSHG 0
PUSHI 3
```

```
EQUAL
JZ E2
PUSHI 1
WRITEI
JUMP E3
E2:
PUSHI 0
WRITEI
E3:
STOP
```

Função menor

```
    declare{
N,men,curr,i
}

main{
read(N)
i = 0

repeat{

read(curr)

if i == 0 then {

men = curr

}

else{

if men > curr then {

men = curr

}

else {
men = men
}
}

i = i + 1

} until i == N
```

```
write(men)
}
```

Código *assembly* gerado:

```
    PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 1
START
READ
ATOI
STOREG 0
PUSHI 0
STOREG 3
E4:
READ
ATOI
STOREG 2
PUSHG 3
PUSHI 0
EQUAL
JZ E2
PUSHG 2
STOREG 1
JUMP E3
E2:
PUSHG 1
PUSHG 2
SUP
JZ E0
PUSHG 2
STOREG 1
JUMP E1
E0:
PUSHG 1
STOREG 1
E1:
E3:
PUSHG 3
PUSHI 1
ADD
STOREG 3
PUSHG 3
PUSHG 0
EQUAL
JZ E4
PUSHG 1
```

```
WRITEI
STOP
```

Função produtório

```
    declare{
accum
N
curr
}

main{

N = 5

accum = 1

repeat {

read(curr)

accum = accum * curr

N = N - 1

} until N == 0

write(accum)
}
```

Código *assembly* gerado:

```
    PUSHN 1
PUSHN 1
PUSHN 1
START
PUSHI 5
STOREG 1
PUSHI 1
STOREG 0
E0:
READ
ATOI
STOREG 2
PUSHG 0
PUSHG 2
MUL
STOREG 0
```

```
PUSHG 1
PUSHI 1
SUB
STOREG 1
PUSHG 1
PUSHI 0
EQUAL
JZ E0
PUSHG 0
WRITEI
STOP
```

Função ímpar:

```
    declare{
cont, curr, N
}

main{

N = 5

cont = 0

repeat {

read(curr)

if curr % 2 == 1 then{

write(curr)

cont = cont + 1

}

else {

cont = cont

}

N = N - 1

} until N == 0


write(cont)
```

```
}
```

Código *assembly* gerado:

```
    PUSHN 1
PUSHN 1
PUSHN 1
START
PUSHI 5
STOREG 2
PUSHI 0
STOREG 0
E2:
READ
ATOI
STOREG 1
PUSHG 1
PUSHI 2
MOD
PUSHI 1
EQUAL
JZ E0
PUSHG 1
WRITEI
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP E1
E0:
PUSHG 0
STOREG 0
E1:
PUSHG 2
PUSHI 1
SUB
STOREG 2
PUSHG 2
PUSHI 0
EQUAL
JZ E2
PUSHG 0
WRITEI
STOP
```

Função que inverte um array:

```
    declare{
```

```
array[5],N
}

main{

N = 5

repeat{
read(array[N-1])
N = N - 1
}until N == 0

repeat{
write(array[N])
N = N + 1
}until N == 5

}
```

Código *assembly* gerado:

```
    PUSHN 5
PUSHN 1
START
PUSHI 5
STOREG 5
E0:
PUSHGP
PUSHI 0
PADD
PUSHG 5
PUSHI 1
SUB
READ
ATOI
STOREN
PUSHG 5
PUSHI 1
SUB
STOREG 5
PUSHG 5
PUSHI 0
EQUAL
JZ E0
E1:
PUSHGP
PUSHI 0
PADD
```

```
PUSHG 5
LOADN
WRITEI
PUSHG 5
PUSHI 1
ADD
STOREG 5
PUSHG 5
PUSHI 5
EQUAL
JZ E1
STOP
```

Função potência

```
    declare{
base,exp,accum,x
}

potencia{
read(base)
read(exp)
accum = 1

while exp != 0 {
accum = accum * base
exp = exp - 1
}

return accum
}

main{
x = potencia()

write(x)
}
```

Código *assembly* gerado:

```
    PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 1
START
PUSHI 0
PUSHA E2
CALL
```

```
STOREG 3
PUSHG 3
WRITEI
STOP
E2:
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHI 1
STOREG 2
E0:
PUSHG 1
PUSHI 0
EQUAL
NOT
JZ E1
PUSHG 2
PUSHG 0
MUL
STOREG 2
PUSHG 1
PUSHI 1
SUB
STOREG 1
JUMP E0
E1:
PUSHG 2
STOREL -1
RETURN
```

Capítulo 5

Conclusão

Tendo em conta os resultados obtidos e comparando-os com os requisitos pedidos, podemos concluir que a nossa solução funciona corretamente, contudo apresenta as limitações das estruturas de dados adotadas, no sentido de serem estáticas e terem de ter alterado o seu tamanho consoante a necessidade do utilizador. No futuro, caso seja relevante adicionar novos tipos de variáveis por exemplo, este trabalho é bastante les. Passa apenas por adicionar regras de derivação à gramática de *parsing* e definir o tipo na tabela de *hash*, bem como escrever as ações semânticas de acordo com as instruções *assembly* correspondente.

Apêndice A

Código do Programa

A.1 Gramática de *parsing*

```
    %{
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "string.h"
#include "lex.yy.c"
#include "auxFile.c"


int flagError = 0;
HASH_TABLE tabID;
int tag_num = 0;
char* res;

extern int yylineno;

%}

%union{ int intg; char*string;}
%token AND OR NOT
%token EQ NE LT LE GT GE
%token TRUE FALSE IF THEN ELSE REPEAT UNTIL FOR WHILE RETURN
%token <string> DECLARE ID MAIN WRITE READ SKIP
%token <intg> INT

%type <string> Header ListIds ListId Main Cmds Cmd Atrib Write Read
ExprCmpInt ExprInt Termo Fator Condic Repeat ForL While Identificador
%type <string> Bloco Blocos FuncCall
```

```
%%
Programa: Header Blocos Main              { asprintf(&res,"%s%sSTOP\n%s\n",$1,$3,$2); }
        ;

Blocos:                                     {$$ = strdup("");}
      | Bloco Blocos                        { asprintf(&$$,"%s%s", $1,$2);}
      ;

Bloco: ID '{' Cmds '}' {asprintf(&$$,"E%d:\n%sRETURN\n",tag_num,$3);
aloca_fun($1,VOID,tabID,tag_num,&flagError); tag_num++;}
     | ID '{' Cmds RETURN ExprCmpInt '}'      {
     asprintf(&$$,"E%d:\n%s%sSTOREL -1\nRETURN\n",tag_num,$3,$5);
     aloca_fun($1,INTE,tabID,tag_num,&flagError); tag_num++;}
     ;

Header: DECLARE '{' ListIds '}'     { $$ = strdup($3); }
      ;

ListIds: ListIds ListId       { asprintf(&$$, "%s%s", $1,$2); }
   | ListId              { $$ = strdup($1); }
   ;


ListId: Identificador
  | ListId ',' Identificador { asprintf(&$$,"%s%s",$1,$3);}
  ;

Identificador: ID                    { aloca(&$$,tabID,$1,INTEG,1,1,&flagError);
if (flagError) return;}
 | ID'['INT']' {
 aloca(&$$,tabID,$1,ARRAY,$3,1,&flagError);
 if (flagError) return;}
       | ID'['INT']''['INT']'        {
       aloca(&$$,tabID,$1,ARRTD,$3,$6,&flagError); if (flagError) return;}
 ;

Main: MAIN '{' Cmds '}'       { asprintf(&$$,"START\n%s",$3); }
    ;


Cmds : Cmd   { $$ = strdup($1); }
     | Cmds Cmd { asprintf(&$$, "%s%s", $1,$2); }
     ;


Cmd : Atrib             { $$ = strdup($1); }
    | Read              { $$ = strdup($1); }
```

```
    | Write             { $$ = strdup($1); }
| Condic            { $$ = strdup($1); }
| Repeat          { $$ = strdup($1); }
    | While             { $$ = strdup($1); }
    | ForL              { $$ = strdup($1); }
| SKIP              { $$ = strdup(""); }
    | FuncCall          { $$ = strdup($1); }
    ;


Condic: IF ExprCmpInt THEN  '{'  Cmds  '}'  ELSE '{'  Cmds  '}'
    {asprintf(&$$,"%sJZ E%d\n%sJUMP E%d\nE%d:\n%sE%d:\n",$2,
    tag_num, $5,tag_num+1,tag_num,$9,tag_num+1);tag_num += 2;}
      ;

Repeat: REPEAT '{' Cmds '}' UNTIL ExprCmpInt
    {asprintf(&$$,"E%d:\n%s%sJZ E%d\n", tag_num, $3, $6, tag_num);
    tag_num++;}
      ;

While: WHILE ExprCmpInt '{' Cmds '}'
    {asprintf(&$$,"E%d:\n%sJZ E%d\n%sJUMP
    E%d\nE%d:\n",tag_num,$2,tag_num+1,$4,tag_num,tag_num+1); tag_num
    +=2;}
     ;

ForL: FOR '(' Cmd ',' ExprCmpInt ',' Cmd ')' '{' Cmds '}'
    {asprintf(&$$,"%sE%d:\n%sJZ E%d\n%s%sJUMP
    E%d\nE%d:\n",$3,tag_num,$5,tag_num+1,$10,$7,tag_num,tag_num+1);
    tag_num +=2;}
    ;

Atrib : ID    '='    ExprCmpInt
{atribui(&$$,$1,$3,"","",tabID,INTEG,&flagError);
if (flagError) return;}
      | ID'['ExprInt']' '=' ExprCmpInt                          {
      atribui(&$$,$1,$6,$3,"",tabID,ARRAY,&flagError);
      if (flagError) return;}
      | ID'['ExprInt']''['ExprInt']' '=' ExprCmpInt             {
      atribui(&$$,$1,$9,$3,$6,tabID,ARRTD,&flagError);
      if (flagError) return;}
      ;


Write: WRITE '(' ExprCmpInt ')' {asprintf(&$$,"%sWRITEI\n", $3); }
     ;
```

```
Read: READ '(' ID ')' { le(&$$,$3,INTEG,"","",tabID,&flagError);
if (flagError) return;}
    | READ '(' ID '['ExprInt']' ')'
    {le(&$$,$3,ARRAY,$5,"",tabID,&flagError); if (flagError) return;}
    | READ '(' ID '['ExprInt']' '['ExprInt']' ')'    {
    le(&$$,$3,ARRTD,$5,$8,tabID,&flagError); if (flagError) return; }
    ;

FuncCall: ID'('')' {asprintf(&$$,"PUSHI 0\nPUSHA E%d\nCALL\nPOP
1\n", fetch_fun($1,VOID,tabID,&flagError));}
        ;

ExprCmpInt: ExprInt                  { asprintf(&$$, "%s",$1); }
      | ExprCmpInt EQ ExprCmpInt               { asprintf(&$$, "%s%sEQUAL\n",$1,$3); }
          | ExprCmpInt NE ExprCmpInt               { asprintf(&$$, "%s%sEQUAL\nNOT\n",$1,$3); }
          | ExprCmpInt LT ExprCmpInt               { asprintf(&$$, "%s%sINF\n",$1,$3); }
          | ExprCmpInt LE ExprCmpInt               { asprintf(&$$, "%s%sINFEQ\n",$1,$3); }
          | ExprCmpInt GT ExprCmpInt               { asprintf(&$$, "%s%sSUP\n",$1,$3); }
          | ExprCmpInt GE ExprCmpInt               { asprintf(&$$, "%s%sSUPEQ\n",$1,$3); }
          | NOT ExprCmpInt               { asprintf(&$$, "%sNOT\n",$2); }
          | ExprCmpInt OR ExprCmpInt               { asprintf(&$$, "%s%sADD\n",$1,$3);}
          | ExprCmpInt AND ExprCmpInt              { asprintf(&$$, "%s%sMUL\n",$1,$3); }
          ;


ExprInt: Termo                               { asprintf(&$$, "%s",$1); }
       | ExprInt '+' Termo                   { asprintf(&$$, "%s%sADD\n",$1,$3); }
       | ExprInt '-' Termo                   { asprintf(&$$, "%s%sSUB\n",$1,$3); }
       ;

Termo : Fator                                { asprintf(&$$, "%s",$1); }
    | Termo '*' Fator                        { asprintf(&$$, "%s%sMUL\n",$1,$3); }
    | Termo '/' Fator                        { asprintf(&$$, "%s%sDIV\n",$1,$3); }
    | Termo '%' Fator                        { asprintf(&$$, "%s%sMOD\n",$1,$3); }
    ;


Fator : INT                                  { asprintf(&$$,"PUSHI %d\n",$1); }
    | '-' INT                                { asprintf(&$$,"PUSHI -%d\n",$2); }
    | TRUE                   { asprintf(&$$,"PUSHI 1\n"); }
    | FALSE                   { asprintf(&$$,"PUSHI 0\n"); }
    | ID'('')' { asprintf(&$$,"PUSHI 0\nPUSHA
    E%d\nCALL\n",fetch_fun($1,INTE,tabID,&flagError));
    if (flagError) return;}
    | ID { fetch_var(&$$,$1,INTEG,"","",tabID,&flagError);
    if (flagError) return;}
    | ID '['ExprInt']'
    {fetch_var(&$$,$1,ARRAY,$3,"",tabID,&flagError);
```

```
    if (flagError) return;}
    | ID '['ExprInt']''['ExprInt']'
    { fetch_var(&$$, $1,ARRTD,$3,$6,tabID,&flagError);
    if (flagError) return;}
    | '(' ExprCmpInt ')' {asprintf(&$$,"%s",$2);}
    ;

%%

void yyerror(const char* msg){
  if (!flagError){
    printf("Frase invalida: %s, line: %d\n",msg, yylineno);
  }
}


int main(int argc, char const *argv[]){
  int syntErr;
  int fdIn;
  FILE * fdOut;

  tabID = new_hash_table();

  if (argc < 2){
    printf("Error: not enough arguments to Parse\n");
    return 1;
  }

  if (!(fdIn = open(argv[2],O_RDONLY))){
    printf("Error: file to Parse \"%s\" does not exist\n",argv[2]);
    return 2;
  }

  if (!(fdOut = fopen(argv[1],"w"))){
    printf("Error: An error ocurried creating the compiled file\n");
    return 3;
  }

  dup2(fdIn,0);

  syntErr = yyparse();

  if (!(syntErr && flagError)){
    fprintf(fdOut,"%s\n",res);
    return 0;
  }

  errorHand();
```

```
  return flagError + 3;
}



void errorHand(){
  char*error = "Failed, Semantic Error: ";
  switch(flagError){
    case REALLOC:
    printf("%s VARIABLE REALLOCATED, line: %d\n", error, yylineno-1);
    break;

    case NOALLOC:
    printf("%s VARIABLE NOT ALLOCATED, line: %d\n", error, yylineno-1);
    break;

    case TYPDIFF:
    printf("%s INCOMPATIBLE TYPES, line: %d\n", error, yylineno-1);
    break;

    case NORETRN:
    printf("%s FUNCTION WITH NO RETURN CALLED IN EXPRESSION, line: %d\n", error, yylineno-1);
    break;

    case NODEFIN:
    printf("%s FUNCTION NOT DEFINED CALLED, line: %d\n", error, yylineno-1);
    break;

    case REDEFIN:
    printf("%s FUNCTION REDEFINED, line: %d\n", error, yylineno-1);
    break;
  }
}
```

A.2 Funções Auxiliares em C

```
    #include "auxFile.h"

unsigned long hashFun(char *str){
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash % HASHSZ;
}
```

```
VAR_LIST new_list(char* name, int pos){
VAR_LIST new = (VAR_LIST) malloc(sizeof(struct var_list));

new -> name = strdup(name);
new -> pos = pos;
new -> prox = NULL;

return new;
}

HASH_TABLE new_hash_table(){

   HASH_TABLE new = (HASH_TABLE) malloc(sizeof(struct hash_table));

   int i;

   for(i = 0; i < HASHSZ; i++){
    new->table[i] = NULL;
    new->fun_table[i] = NULL;
   }

   new -> used = 0;

   return new;
}


VAR_LIST lookup(HASH_TABLE hash_table, char* var_name){
int key = (int) hashFun(var_name);

VAR_LIST curr = hash_table->table[key];

while(curr != NULL && strcmp(curr->name,var_name) != 0 ){
curr = curr -> prox;
}

return curr;
}



void aloca(char**instruction,HASH_TABLE hash_table, char* var_name,
int type, int lin, int col, int *flagError ){
int key = (int) hashFun(var_name);
VAR_LIST elem = lookup(hash_table,var_name);

if( elem != NULL){
```

```
*flagError = REALLOC;
return;
}


VAR_LIST allocate = new_list(var_name, hash_table->used);
allocate->type = type;
allocate->line = lin;
allocate->prox = hash_table->table[key];
hash_table->table[key] = allocate;
hash_table->used += lin*col;

asprintf(instruction,"PUSHN %d\n", lin*col);
}



void fetch_var(char** instruction, char* var_name, int type, char*
inst_frstindex, char* inst_scndindex, HASH_TABLE tabID,
int*flagError){
VAR_LIST elem = lookup(tabID,var_name);

if (elem == NULL){
*flagError = NOALLOC;
return;
}

if(elem->type != type){
*flagError = TYPDIFF;
}

switch (type) {

case ARRAY:
asprintf(instruction,"PUSHGP\nPUSHI %d\nPADD\n%sLOADN\n",elem->pos,inst_frstindex);
break;

case ARRTD:
asprintf(instruction, "PUSHGP\nPUSHI %d\nPADD\n%sPUSHI
%d\nMUL\n%sADD\nLOADN\n", elem->pos, inst_scndindex,
elem -> line, inst_frstindex);
break;

default:
asprintf(instruction, "PUSHG %d\n", elem->pos);
break;
}
}
```

```
void atribui(char**instruction, char* var_name, char*inst_var_val,
char* inst_frstindex, char* inst_scndindex, HASH_TABLE tabID, int
type, int *flagError){
VAR_LIST elem = lookup(tabID,var_name);
if (elem == NULL){
*flagError = NOALLOC;
return;
}

if(elem -> type != type){
*flagError = TYPDIFF;
return;
}

switch (type) {

case ARRAY:
asprintf(instruction,"PUSHGP\nPUSHI
%d\nPADD\n%s%sSTOREN\n",elem->pos,inst_frstindex,inst_var_val);
break;


case ARRTD:
asprintf(instruction, "PUSHGP\nPUSHI %d\nPADD\n%sPUSHI
%d\nMUL\n%sADD\n%sSTOREN\n", elem->pos, inst_scndindex,
elem -> line, inst_frstindex,inst_var_val);
break;

default:
asprintf(instruction,"%sSTOREG %d\n",inst_var_val,elem->pos);
break;
}
}



void le(char**instruction,char* var_name, int type, char*
inst_frstindex, char* inst_scndindex, HASH_TABLE tabID, int* flagError){
atribui(instruction,var_name,"READ\nATOI\n",inst_frstindex,
inst_scndindex,tabID,type,flagError);
}


int fetch_fun(char* fun_name, int type, HASH_TABLE hash_table,
int* flagError){
int key = (int) hashFun(fun_name);
FUN_LIST curr = hash_table->fun_table[key];
```

```
while(curr != NULL && strcmp(curr->name,fun_name) != 0 ){
curr = curr -> prox;
}

if (curr == NULL){
*flagError = NODEFIN;
return -1;
}

if (type == VOID && curr->type != VOID){
printf("WARNING: RETURN VALUE OF FUNCTION: %s() IS
IGNORED\n", curr->name);
}

if (type != VOID && curr->type == VOID){
*flagError = NORETRN;
return -1;
}

return curr->tag;
}


void aloca_fun(char* fun_name, int type, HASH_TABLE hash_table, int tag_num, int* flagError){
int x;
int key = (int) hashFun(fun_name);
fetch_fun(fun_name,VOID,hash_table,&x);

if (x != NODEFIN){
*flagError = REDEFIN;
return;
}

FUN_LIST allocate = new_fun_list(fun_name);
allocate->tag = tag_num;
allocate->type = type;
allocate->prox = hash_table->fun_table[key];
hash_table->fun_table[key] = allocate;
}


FUN_LIST new_fun_list(char*name){
FUN_LIST new = (FUN_LIST) malloc(sizeof(struct fun_list));

new -> name = strdup(name);
new -> prox = NULL;
return new;
}
```

A.3 Analisador Léxico

```
    %{
#include "y.tab.h"
#include "string.h"
#include <stdio.h>
#include <ctype.h>
%}

%option noyywrap
%option yylineno


%%

[+\-*/()=\{\},%\[\]] {return yytext[0];}
(?i:false) {return FALSE;}
(?i:true) {return TRUE;}
(?i:or) {return OR;}
(?i:and) {return AND;}
(?i:main)            {return MAIN;}
(?i:declare)         {return DECLARE;}
(?i:if)                                 {return IF;}
(?i:WHILE)                              {return WHILE;}
(?i:FOR)                                {return FOR;}
(?i:then)                               {return THEN;}
(?i:else)                               {return ELSE;}
(?i:repeat)                             {return REPEAT;}
(?i:until)                              {return UNTIL;}
(?i:skip)                               {return SKIP;}
(?i:return)                             {return RETURN;}
"==" {return EQ;}
"!=" {return NE;}
"<=" {return LE;}
"<" {return LT;}
">=" {return GE;}
">" {return GT;}
"not" {return NOT;}
"read" {return READ;}
"write" {return WRITE;}
[a-zA-Z]+ {yylval.string = strdup(yytext);return ID;}
[0-9]+ {yylval.intg = atoi(yytext);return INT;}
~[ ](.)* {;} // comentario
[~][/][^/]*[/]+([^/~][^/]*[/]+)*[~]     {;} // comentario multi linha
.             {;}
[\n] {;}
%%
```