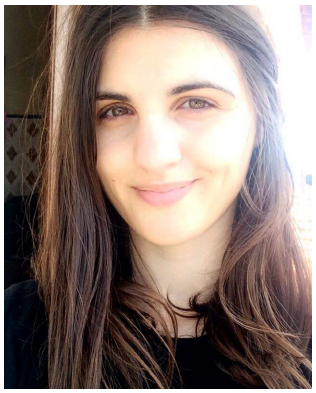
Universidade do Minho

Departamento de Engenharia Informática
Mestrado Integrado em Engenharia Informática
Processamento de Linguagens

Trabalho 2

Compilador de uma LPIS (Linguagem de Programação Imperativa Simples)

Grupo 38


Célia Figueiredo a67637

Gil Gonçalves a67738


Diogo Tavares a61044

Braga, 31 de Maio de 2016

Resumo

Este relatório descreve o processo de desenvolvimento e decisões tomadas para a realização do segundo trabalho prático da Unidade Curricular de Processamento de Linguagens.
O problema a resolver consiste na criação de uma linguagem imperativa simples (LPIS) e da respetiva gramatica independente de contexto **GIC** seguida do desenvolvimento de um compilador para a mesma que gera pseudo-código *Assembly*.
A linguagem desenvolvida foi baseada na linguagem de programação C, e suporta:

- Variáveis globais
- Ciclos: for, while, do while
- Estruturas de Condição: If ... Else
- Expressões Aritméticas e lógicas
- Funções com argumentos
- Declaração de variáveis locais dentro das funções

O compilador foi desenvolvido com recurso ao analisador léxico Flex e ao analisador sintático Yacc.

Conteúdo

Capítulo 1

Introdução

O presente trabalho enquadra-se na unidade curricular de Processamento de Linguagens da Licenciatura em Engenharia Informática da Universidade do Minho. O trabalho pretende aumentar a experiência em engenharia de linguagens, assim como incentivar o desenvolvimento de processadores de linguagens e compiladores em ambiente Linux. As ferramentas que suportam a sua implementação consistem no conjunto flex-yacc, sendo estes um gerador de analisadores léxicos e um gerador de analisadores sintáticos/semânticos, respetivamente.

Inicialmente é definida uma linguagem de programação imperativa simples, a qual deve permitir manusear variáveis do tipo inteiro assim como realizar operações básicas. As variáveis devem sem declaradas no início do programa e não pode haver re-declarações. Após a validação da linguagem criada com o docente, é desenvolvido um compilador para esta linguagem, com base na GIC criada e com recurso ao Gerador Yacc/Flex. O compilador da linguagem deve gerar pseudo-código Assembly da Máquina Virtual fornecida.

Estrutura do Relatório

A elaboração deste documento teve por base a estrutura do relatório fornecida pelo docente. O relatório encontra-se então estruturado da seguinte forma: no Capítulo 1 é feita uma contextualização ao assunto tratado neste trabalho, seguindo-se uma introdução onde são apresentadas as metas de aprendizagem pretendidas. Posteriormente, no Capítulo 2, é exposto o enunciado do trabalho e a descrição do problema.

Imediatamente após, no Capítulo 3, é apresentada a linguagem criada, a gramática correspondente e descrita a forma como foi gerado o código Assembly. São também descritos os módulos do programa e as estruturas utilizadas.

No Capítulo 4 são apresentados os testes realizados e os seus resultados.

Por último, no Capítulo 5, faz-se uma análise crítica relativa quer ao desenvolvimento do projeto quer ao seu estado final e ainda é feita uma abordagem ao trabalho futuro. No fim do documento estará incluido ficheiros anexos com o código desenvolvido.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema - Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples LPIS, a seu gosto.
Apenas deve ter em consideração que a LPIS terá de permitir:

- *declarar e manusear* variáveis atómicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *declarar e manusear* variáveis estruturadas do tipo *array (a 1 ou 2 dimensões) de inteiros*, em relação aos quais é apenas 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— *condicional* e *cíclica*—que possam ser aninhadas.
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado atómico (opcional).

Neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

2.2 Descrição do problema

Como descrito na secção anterior o desafio que nos foi proposto consiste na criação de uma linguagem imperativa simples apenas com as características mais básicas, o desenvolvimento de uma **GIC** para a mesma e ainda a criação de um compilador gerador de **pseudo-código** Assembly.
Posto isto tivemos então de, primeiramente, pensar e desenhar uma linguagem imperativa ao nosso gosto, seguindo os requisitos pedidos. De seguida foi necessário criarmos uma gramática que a descrevesse e, finalmente, com recurso a essa gramática, tornar possível o armazenamento dos dados vindos de um programa escrito na nossa linguagem necessários à geração do **pseudo-código** Assembly.

Capítulo 3

Concepção/desenho da Resolução

O sistema desenvolvido é constituído por 2 modelos pricipais: gramatica.l e gramatica.y, que são respetivamente o analisador léxico e analisador sintático.

O analisador sintático utiliza o ficheiro compilador.h, sendo que este módulo responsável pelo tratamento das variáveis e funções existentes (adicionar/consultar variáveis).

O sistema utiliza também duas estruturas de dados: uma HashMap e uma Stack. A hashmap é utilizada para guardar as variáveis e as funções, enquanto que a stack permite o controlo das labels dos ciclos durante a compilação.

Na Figura 3.1, é possível observar as dependências entre os diversos ficheiros, a estrutura do nosso projeto que terá de passar pelo desenvolvimento de um analisador léxico em *Flex* que fará o reconhecimento dos *tokens* que são utilizados no analisador semântico que gera o código *Assembly*.

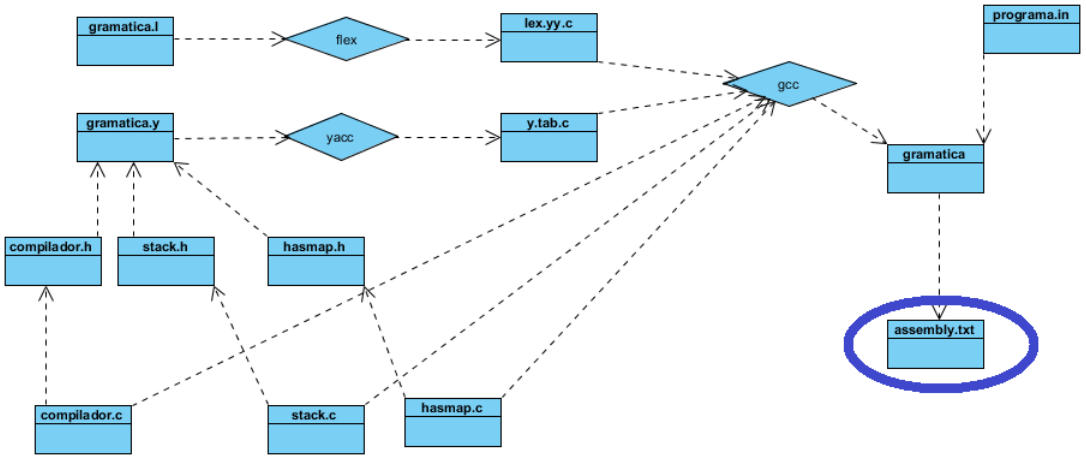


Figura 3.1: Esquema de dependências dos ficheiros

3.1 Desenho da Linguagem

O analisador léxico encontra-se desenvolvido com o suporte da ferramenta 'ylex' e deteta todos os símbolos terminais da linguagem (palavras reservadas, sinais e variáveis). Este analisador efetua também a deteção de comentários (linhas precedidas pelos símbolos '//'), ignorando o texto neles contidos. De forma a facilitar a deteção os erros de sintaxe, o parser conta as linhas que já interpretou. Este funcionalidade permite ao analisador sintático informar a linha onde ocorrer a anomalia em caso de erro de processamento.

3.2 Analisador sintático/semântico

O analisador sintático/semântico é o responsável por processar os tokens obtidos através do analisador léxico, utilizando as produções definidas para calcular a instrução a escrever no ficheiro de output final. O ficheiro onde se encontram todas as produções denomina-se gramatica.y,

Definição dos tokens

Previamente à definição das produções, estão definidos os diversos tokens assim como o tipo das diferentes variáveis presentes nas produções.

Produções iniciais

O processamento é iniciado na produção Prog, correspondente ao começo do programa. São realizadas as inserções iniciais no ficheiro de output, e as produções a serem efetuadas podem consistir numa lista de declarações, numa lista de funções, ou numa lista de instruções. Caso se encontre uma lista de declarações, é realizado o controlo de saltos de instruções, através da inserção da instrução "JUMP inicio"no ficheiro de output. Isto possibilita a declaração de variáveis antes da declaração das funções, de forma a que as funções possuam acesso às variáveis globais declaradas.

Produções de Funções

A declaração de funções é precedida pelo símbolo terminal '#', de forma a facilitar a sua leitura e distinção com outras instruções. O armazenamento das funções declaradas é realizado através da utilização de uma hashmap, e permite guardar o contexto de estas, ou seja, as declarações efetuadas dentro da função são tidas como variáveis locais, e apenas acessíveis dentro da função. No final da declaração da função, o contexto é encerrado. Por simplicidade, não é possível utilizar chamadas a funções como argumento de outra função.

Produções de declarações

As declarações de variáveis podem ocorrer dentro de uma função (i.e. variáveis locais) ou fora de qualquer contexto (i.e. variáveis globais). Desta forma, ao declarar uma variável, é efetuada o teste que verifica se está está declarada ou não dentro de uma função. Além disso, não é permitido re-declarar variáveis locais com o mesmo nome que variáveis globais já existentes, sendo apresentado um erro quando isso ocorre.

Produções de instruções e atribuições

São suportadas diversas instruções, no entanto vale a pena mencionar a instrução "return". Esta instrução calcula o endereço a retornar a partir do número de argumentos de entrada da função na qual se insere e do frame pointer (fp) salvaguardado. As atribuições permitem efetuar o incremento/decremento de variáveis através da sintaxe vars++/vars–, assim como atribuir o valor de expressões a variáveis escalares ou vetoriais.

Produções de input/output

As produções referentes à leitura e escrita no stdin. A produção de escrita suporta a impressão de expressões, sendo que a produção de leitura armazena as variáveis atribuídas conforme o contexto definido.

Produções de controlo de fluxo de execução condicional

A instrução de controlo de fluxo de execução condicional está definida como if...else, suportando um conjunto de instruções no seu contexto.

Produções de controlo de fluxo de execução cíclico

As produções correspondentes ao controlo de fluxo de execução cíclico suportam ciclos 'while', 'do while' e 'for'. Estas produções são suportadas por uma estrutura de dados (stack), sendo esta a que torna possível calcular os saltos a efetuar.

Produções de cálculo de expressões

O compilador possui a capacidade de processar expressões aritméticas e lógicas, tanto em forma de declaração como em forma de cálculo do índice de um array. Estas expressões possibilitam também a utilização de parêntesis aninhados.

Nota: Na compilação do código referente ao programa yacc, é apresentado um erro de compilação do tipo shift-reduce. No entanto, este erro não provoca qualquer comportamento indesejado no programa dado que por defeito, quando em dúvida, o yacc efetua um shift ao invés de um reduce, sendo este o comportamento pretendido para este caso.

3.3 Desenho da Gramatica

Após desenhada e descrita a linguagem imperativa que iremos usar, resta transformar todas as regras que definimos sobre ela numa **GIC**.
Os símbolos não-terminais da gramática que definimos são os seguintes: prog, ListaDecla, ListaFun, ListaInst, Funcao, Inst, TipoFun, IdFun, ListaArg, ListaArg2, Tipo, Decla, Var, ConjInst, Atrib, Print, Scan, If, Else, While, DoWhile, For, ForHeader ,ForAtrib, ExpLog, Exp, Termo, Fun, FunArgs, FunArgs2, TestExpLog.
Por sua vez os símbolos terminais são: INT, WHILE, FOR, IF, ELSE, RETURN, VOID, PRINT, SCAN, DO, num, id ";", , "(",")", "", , "=", "#", "[", "]", ≫", ≪".
Na concepção da nossa gramática utilizamos em todos os casos recursividade à esquerda. Segue-se então a gramática gerada:

```
Prog        : ListaDecla  ListaFun  ListInst
            ;

ListaDecla  :
            | ListaDecla Decla
            ;

ListaFun    :
            | ListaFun Funcao
            ;

ListInst    : Inst
            | ListInst Inst
            ;

// --------------------  FUNCAO -----------------------------------
/** A declaração de funções é precedida pelo símbolo terminal '\#'.
 */

Funcao      : '#' TipoFun IdFun  '(' ListaArg ')' '{' ListaDecla ListInst '}'
            ;

TipoFun     : VOID
            | INT
            ;
```

```
IdFun  : id
;

ListaArg    :
            | ListaArg2 ;

ListaArg2   : Tipo Var
            | ListaArg2  ’,’  Tipo Var
            ;

Tipo  : INT
;

// --------------------DECLARACAO -----------------------------------

Decla       : INT Var ’;’
            | INT Var ’[’ num ’]’ ’;’
            | INT Var ’[’ num ’]’ ’[’ num ’]’ ’;’
            ;

Var  : id
;

// -------------------INSTRUCAO -----------------------------------

ConjInst    :
            |’{’ ListInst ’}’
            ;

Inst        : If
            | While
            | DoWhile
            | For
            | Atrib ’;’
            | Print’;’
            | Scan ’;’
            | RETURN Exp ’;’
            | ELSE
            ;

// ---------------------------------- ATRIBUIÇAO ----------------------------------

Atrib       : Var ’=’ Exp
            | Var ’+’’+’
            | Var’[’ Exp ’]’ ’=’ Exp
            | Var’[’ Exp ’]’ ’[’ Exp ’]’ ’=’ Exp
            ;

// ---------------------------------- PRINT SCAN ----------------------------------

Print:     PRINT ’(’ Exp ’)’
            ;

Scan:      SCAN ’(’ Var’)’
```

```
            ;
// ----------------------------------- IF THEN ELSE ----------------------------------

If          :  IF  TestExpLog  ConjInst Else
            ;

Else        :
            | ELSE ConjInst
            ;

// ----------------------------------# WHILE -------------------------------------------

While       : WHILE TestExpLog ConjInst
            ;

// ----------------------------------# DO WHILE -----------------------------------------

DoWhile     : DO ConjInst WHILE TestExpLog ';'
            ;

// ----------------------------------# FOR ----------------------------------------------

For         : FOR ForHeader ConjInst
            ;

ForHeader   :  '(' ForAtrib ';' ExpLog ';' ForAtrib ')'
            ;

ForAtrib    : Atrib
            ;

// --------------------------------CALCULO DE EXPRESSOES ----------------------------------
ExpLog  : Exp
|Exp '=''=' Exp
|Exp '!''=' Exp
|Exp '>''=' Exp
|Exp '<''=' Exp
|Exp '<' Exp
|Exp '>' Exp
;

Exp  : Termo
|Exp '+' Termo
|Exp  '-' Termo
|Exp '|''|' ExpLog
;

Termo : Fun
| Termo '/' Fun
| Termo '*' Fun
| Termo '&''&' ExpLog
;

Fun      : num
```

```
            | Var
            | Var '['Exp ']'
            | Var  '['Exp ']' '['Exp ']'
            | IdFun '(' FunArgs')'
            | '(' Exp ')'
            ;

FunArgs     :
            | FunArgs2
            ;

FunArgs2    : Exp
            | FunArgs2 ',' Exp
            ;

TestExpLog  : '(' ExpLog ')'
            ;
```

3.4 Geração de Pseudo-Código Assembly

Para gerar o código Assembly do programa é necessário definir ações semânticas no *YACC*. Estas ações semânticas são blocos de código C que são executados aquando o reconhecimento da expressão que os antecede. Assim sendo, é realizada uma tradução da linguagem desenvolvida para a linguagem *assembly* da VM, à medida que cada instrução ou expressão é identificada.

3.4.1 Declaração das variáveis

Na parte das declarações de variáveis é necessário inseri-las na estrutura que críamos e gerar o código Assembly para alocar memória para elas. Para isso decidimos usar variáveis globais para guardar o tamanho, o endereço, o tipo e o nome de uma variável. Sempre que o analisador sintático reconhece uma declaração de um inteiro coloca a variável correspondente a 1 e se reconhecer um *array* coloca nessa mesma variável o seu tamanho. Da mesma forma é atualizada a variável correspondente ao tipo.

Quanto à variável correspondente ao endereço é uma variável global com valor inicial de zero que é incrementada sempre que uma nova variável é adicionada com sucesso à estrutura. Quando é encontrado o nome de uma variável já tem então guardado o seu tamanho e tipo e encontra-se já em condições de guardá-la na estrutura de dados. Nesta altura é feita a verificação da já existência de uma variável com o mesmo nome ou não. Caso já exista o programa termina e é gerada uma mensagem de erro. Caso contrário é gerado o código Assembly para empilhar a variável na *Stack*: PUSHN X, em que X é o tamanho da variável. De seguida é incrementada a variável do endereço tantas vezes quanto o tamanho da variável.

3.4.2 Inicio do programa e variáveis

Após todas as declarações terem sido reconhecidas é gerada a instrução Assembly START que indica o início das instruções do programa.

Em todas as instruções, sempre que é encontrada uma variável é feita a verificação da sua existência na estrutura. Se não existir o programa termina com um erro. Se existir, é guardado o seu registo para ser utilizado no código Assembly. No caso de se tratar de um acesso a uma posição de um *array* é também feita a verificação do seu tipo.

3.4.3 Atribuição e operações lógicas

Nas instruções de atribuição o código Assembly gerado, no caso de ser uma variável inteira escalar, é apenas um STOREG X, em que X é o endereço da variável em questão. No caso de ser um acesso a um vetor, é necessário aquando do reconhecimento do nome da variável efetuar um PUSHGP seguido de um PUSHI X, em que X é o endereço da variável em causa, e de um PADD. De seguida, após o reconhecimento das expressões constituintes da atribuição é gerado um STOREN.

Para as expressões, termos e condições que utilizam operadores aditivos, multiplicativos e relacionais, respetivamente, foi feito o reconhecimento de que operador estava a ser utilizado, e tendo em conta isso gerado o código Assembly respetivo a essa operação. De notar que não existe uma instrução Assembly para as operações lógicas e e ou, logo utilizamos a multiplicação e a soma, respetivamente, para representá-las.

3.4.4 Escrita

Para as instruções de escrita o único código Assembly necessário é um WRITEI.
Para as instruções de leitura, no caso de a leitura ser feita para uma variável escalar, é gerado o código READ para ler uma*string* do teclado, seguido do código ATOI para transformá-la num inteiro, e de seguida guardá-la no endereço da variável em questão com a instrução STOREG X, em que X é o endereço. Já no caso de se tratar de um vetor é necessário um PUSHGP seguido de um PUSHIX e de um PADD antes de proceder à leitura em si, que se dá da mesma forma que no caso do escalar, mas substituindo o STOREG por um STOREN.

3.4.5 Condicional

Quando às instruções condicionais, recorrendo também às *labels*, e aos *jumps* seguimos a seguinte lógica: após a condição do *if* temos um JZ para o início do bloco *else* caso este exista e para o fim do *if* caso contrário.

3.4.6 Ciclos

No caso dos ciclos utilizamos *labels* para marcar o início e o fim de um ciclo. Depois de reconhecida a expressão da condição do ciclo fazemos um JZ (salta se a expressão for falsa) para a *label* do fim de ciclo. No fim do ciclo temos um JUMP não condicional que salta sempre para o início do ciclo (antes da condição).

No ciclo 'for', é necessária a utilização de instruções 'JUMP', de forma a ser possível seguir o seu fluxo de execução normal. Após a identificação e execução das ações associadas à expressão lógica presente no ciclo 'for', é gerado o salto condicional respetivo, assim como um salto para as instruções associadas ao corpo do ciclo. Além disso, é gerada uma *label* que irá corresponder ao incremento do ciclo que irá ser identificado de seguida. Identificado o final do corpo do ciclo, é efetuado um salto para a *label* correspondente ao incremento do ciclo, que para além das respetivas instruções, conterá outro 'JUMP' para o teste da expressão lógica.

3.4.7 Funções

De forma a implementar adequadamente o processamento de funções, é necessário resolver certas implicações, sendo que esta funcionalidade obriga a tratar de diversos contextos dentro do programa (global e local).

As declarações das funções são efetuadas após as declarações das variáveis globais para que dentro das funções seja possível aceder às variáveis globais.

As declarações de variáveis são feitas no inicio da função, e no hashtable das variáveis é guardado não apenas o endereço mas também o contexto (local ou global). Desta forma, no acesso às variáveis, utiliza-se 'PUSHG' ou 'PUSHL' seja respetivamente variável global ou local.

A passagem de argumentos para a função é tratada como uma declaração especial na qual o endereço é negativo. Já o retorno da função é colocado também num endereço negativo que foi previamente alocado na chamada da função.

Em forma de exemplo, assumindo a chamada de uma função com 2 argumentos, os endereços são negativos ao fp, e o endereço onde a função colocará o retorno é $fp-3$. Após a execução da função é feito o 'pop' dos argumentos e assim o valor de retorno da função está no topo da stack.

3.5 Estruturas de Dados

3.5.1 Stack

De forma a evitar confusão na atribuição de *labels* relativas a *ifs* e *loops* é utilizado um contador de condições. À medida que é encontrada uma instrução que implique o uso de uma condição, este contador é incrementado e o seu valor é colocado numa stack. Deste modo, o valor que se encontra no topo da stack é relativo ao último *ciclo/if* encontrado. Sempre que é encontrado o final de uma condição, o valor no topo da stack é removido. Através do uso de um contador e de uma stack, é muito mais simples gerir as *labels* e as operações de controlo, como *JUMPs* e *JZs*. A stack utilizada implementa apenas as funções necessárias para a sua inicialização, inserção, remoção e consulta. Com as operações de push/pop são inseridos/removidos valores no topo da stack, e com a operação de get apenas é consultado o valor no topo da stack, sem que este seja removido. Esta última operação é útil para a geração de instruções 'JZ' na geração de código VM.

3.5.2 HashMap

Como foi referido anteriormente, é utilizada uma *hashmap* com o objetivo de guardar as variáveis e as funções. Quanto às variáveis, é necessário guardar e aceder a informação como o seu endereço e tipo. Sendo assim, foi criada uma estrutura de dados auxiliar para armazenar essa informação. O nome da variável é utilizado como chave e, a partir dela, conseguimos aceder à sua informação correspondente na hashmap. Relativamente às funções, é necessário guardar e aceder a informação como o seu nome, informação relativa aos seus argumentos de entrada e o tipo de dados de saída. Para esse feito, também foram necessárias estruturas de dados auxiliares como uma lista ligada capaz de armazenar informação relativa aos argumentos de entrada e uma outra que contem a informação relativa ao tipo de dados de saída, dados de entrada e nome da função. Neste caso, o nome da função funciona como chave na *hashmap* e o seu valor é a estrutura que contem a informação mais geral sobre a função. A *hashmap* utilizada implementa funções necessárias para a sua inicialização, inserção, remoção e consulta, sendo que também contém outras funções que não foram utilizadas no desenvolvimento deste projeto.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

4.1.1 Makefile

O principal objetivo da Makefile é facilitar a compilação e execução do programa. Para isso criamos o seguinte ficheiro:

```
gramatica: y.tab.c lex.yy.c stack.o compilador.o hashmap.o
gcc -o gramatica y.tab.c stack.c compilador.c hashmap.c

lex.yy.c: gramatica.l
flex gramatica.l

y.tab.c: gramatica.y
yacc -d gramatica.y

stack.o: stack.c stack.h
gcc -c stack.c
compilador.o : compilador.c compilador.h
gcc -c compilador.c
hashmap.o: hashmap.c hashmap.h
gcc -c hashmap.c

clean:
rm  *.o lex.yy.c y.tab.c y.tab.h gramatica
```

4.2 Testes realizados e Resultados

4.2.1 Teste 1

4.2.2 Teste 2

4.2.3 Teste 3

4.2.4 Teste 4

4.2.5 Teste 5

4.2.6 Teste 6

Capítulo 5

Conclusão

Finalizado o desenvolvimento do trabalho, é possível analisar o resultado final e o impacto que as diversas decisões tiveram sobre este. Um dos principais pontos positivos consiste na implementação do processamento e compilação de funções, sendo esta a funcionalidade mais trabalhosa e sobre a qual recaiu maior parte do tempo despendido. Relativamente a estas, de notar uma mudança na forma como estas foram implementadas. Anteriormente, a estrutura relativa ao armazenamento de dados de uma função possuía a capacidade de dar acesso às variáveis declaradas dentro do seu contexto, no entanto decidiu-se que esta funcionalidade era desnecessária para o funcionamento do compilador, sendo esta informação descartada. A implementação das expressões de controlo de execução possuíram também uma dificuldade acrescida, obrigando à utilização de estruturas de dados mais complexas, tais como hashmaps e stacks. Creemos ter alcançado os objetivos definidos aquando da proposta do trabalho, tendo desenvolvido um compilador capaz de processar uma LPIS, com a possibilidade de dar feedback sobre o código de input definido e criar o ficheiro com instruções em Assembly correspondentes.

Apêndice A

Código Flex

```
%{
#include "compilador.h"
#include "y.tab.h"
int ccLine = 1;


%}


%%

int             {return(INT);}
while           {return(WHILE);}
for             {return(FOR);}
if              {return(IF);}
else            {return(ELSE);}
return          {return(RETURN);}
void            {return(VOID);}
print               {return(PRINT);}
scan                {return(SCAN);}
do              {return(DO);}
\=                          {return('=');}
\.                  {return('.');}
\;                  {return(';');}
\(                  {return('(');}
\)                  {return(')');}
\{              {return('{');}
\}                  {return('}');}
\[                  {return('[');}
\]                  {return(']');}
\,                  {return(',');}
\<                  {return('<');}
\>                  {return('>');}
\+                  {return('+');}
\-                  {return('-');}
\*                  {return('*');}
\/                  {return('/');}
\%                  {return('%');}
\#              {return('#');}
\|                          {return('|');}
\&                          {return('&');}
[a-zA-Z]+           {yylval.var_nome = strdup(yytext); return(id);}
```

```
43 [0-9]+            {yylval.valor = atoi(yytext); return(num);}
44 [\n]          {ccLine++;}
45 \/\/.*        { ; }
46 .             { ; }
47
48
49 %%
50
51 int yywrap(){
52         return(1);
53         }
```

Apêndice B

Código Yacc

```
%{
#include "compilador.h"
#include <stdio.h>
#include <string.h>
#include "stack.h"
#include <stdlib.h>
#include "y.tab.h"

extern ccLine;
static int total;
FILE *f;
static Stack s;

%}

%union{
char* var_nome;
int valor;
Tipo tipo;
struct sVarAtr
{
char* var_nome;
int valor;
int size;
} varAtr;
}


%token INT WHILE FOR IF ELSE RETURN VOID PRINT SCAN DO
%token <valor>num
%token <var_nome>id

%type <tipo> TipoFun
%type <tipo> Tipo
%type <var_nome> IdFun
%type <varAtr> Var
```

```
// --------------------PROGRAMA -----------------------------------
/**
Um programa é uma lista de declarações, lista de Funcões , e uma lista de Instruções
*/

%%

Prog       :
ListaDecla          {fprintf(f,"start\n");fprintf(f,"jump inicio\n");}
ListaFun  {fprintf(f,"inicio:nop\n");}
ListInst     {fprintf(f,"stop\n");}

;

ListaDecla  :
| ListaDecla Decla
;

ListaFun    :
| ListaFun Funcao
;

ListInst    : Inst
| ListInst Inst
;

// --------------------  FUNCAO -----------------------------------
/** A declaração de funções é precedida pelo símbolo terminal '\#'.
*/

Funcao      : '#' TipoFun IdFun                 {inserFuncao($2,$3);}
'(' ListaArg ')'
'{' ListaDecla ListInst '}'     {fim();}
;

TipoFun     : VOID               {$$ =_VOID;}
| INT                   {$$ =_INTS;}
;

IdFun  : id
;

ListaArg    :
| ListaArg2 ;

ListaArg2   : Tipo Var
| ListaArg2  ','  Tipo Var
;

Tipo  : INT                 {$$ =_INTS;}
;
```

```
// --------------------DECLARACAO -----------------------------------

Decla       : INT id ';'                                {decVar($2,1,'S');fprintf(f,"pushi 0\n"); vars++; }
| INT id '[' num ']' ';'                    {if(testeColuna($4)==1)
{decVar($2,$4,'A');fprintf(f,"pushn%d\n",$4);}
else {yyerror("Tamanho menor que 1");}
}
| INT id '[' num ']' '[' num ']' ';'       {if(testeMatriz($4,$7)==1)
{decVar($2,$4*$7,'M');fprintf(f,"pushn %d\n",$4*$7);}
else {yyerror("Tamanho menor que 1");}
}

;


Var  : id                                       {Endereco a=getEndereco($1); $$.var_nome=strdup($1); $$.valor=
;

// --------------------INSTRUCAO -----------------------------------

ConjInst    :
|'{' ListInst '}'
;

Inst        : If
| While
| DoWhile
| For
| Atrib ';'
| Print';'
| Scan ';'
| RETURN Exp ';'     {fprintf(f,"storel %d\n",decFunRetAddr());fprintf(f,"return\n");}
| ELSE               { yyerror("'Else' sem um 'If' anteriormente");return 0;}
;

// ------------------------------------ ATRIBUIÇAO -----------------------------------

Atrib       : Var '=' Exp                        {Endereco a = getEndereco($1.var_nome);
if(a.tipo == _INTS){ fprintf(f,"store%c %d\n",a.tipoVar, a.addr);}
else {yyerror("Tipos incompativeis");return 0;  }
}

| Var '+''+'                          {Endereco a = getEndereco($1.var_nome);
if(a.tipo ==_INTS){fprintf(f,"pushi 1\n push%c %d\n add\n store%c %d\n",a.tipoVar,a.addr, a.tipoVar, a.addr);
else{yyerror("Tipos incompativeis"); return 0; }
}

| Var'[' Exp ']' '=' Exp            {Endereco a = getEndereco($1.var_nome); fprintf(f, "push%cp \n push%c %d
a.tipoVar, a.addr); fprintf(f, "storen\n");}


| Var'[' Exp ']' '[' Exp ']' '=' Exp  {Endereco a = getEndereco($1.var_nome); fprintf(f, "push%cp \n push%c %
a.tipoVar, a.addr);}
;
```

```
// ----------------------------------- PRINT SCAN ----------------------------------

Print:     PRINT '(' Exp ')'   {fprintf(f,"writei\n");}
;

Scan:      SCAN '(' Var')'    {Endereco a= getEndereco($3.var_nome); fprintf(f,"read\n atoi\n store%c %d\n",a
;
// ----------------------------------- IF THEN ELSE --------------------------------

If          :  IF                   {total++; push(s,total);}
TestExpLog            {fprintf(f,"jz endCond%d\n", get(s));}
ConjInst           {fprintf(f," endCond%d\n", pop(s));}
Else
;

Else        :
| ELSE ConjInst
;

// ------------------------------------# WHILE ---------------------------------------

While       : WHILE             {total++; push(s,total); fprintf(f, "ciclo%d: NOP\n", get(s));}
TestExpLog          {fprintf(f, "jz endciclo%d\n", get(s)); }
ConjInst            {fprintf(f, "jump ciclo%d\n endCiclo%d\n", get(s), get(s));  pop(s); }
;

// ------------------------------------# DO WHILE ------------------------------------

DoWhile     : DO                            {total++; push(s,total); fprintf(f, "ciclo%d: NOP\n", get(s));}
ConjInst WHILE TestExpLog ';'   {fprintf(f, "jz endciclo%d\n jump ciclo%d\n endciclo%d: NOP\n", get(s),get(s)
;

// ------------------------------------# FOR -----------------------------------------

For         : FOR ForHeader ConjInst     {fprintf(f,"jump ciclo%dA\nendciclo%d\n", get(s), get(s)); pop(s);}
;

ForHeader   :  '(' ForAtrib ';'         {total++; push(s,total); fprintf(f,"ciclo%d: nop\n", get(s));}
ExpLog ';'                 {fprintf(f,"jz endciclo%d\njump ciclo%dB\nciclo%dA: nop\n", get(s), get(s), get(s)
ForAtrib ')'               {fprintf(f,"jump ciclo%d\nciclo%dB: nop\n", get(s), get(s));}
;


ForAtrib    : Atrib
;

// ------------------------------------------------------------------CALCULO DE EXPRESSOES ------------------
ExpLog  : Exp
|Exp '=''=' Exp  {fprintf(f,"equal\n");}
|Exp '!''=' Exp         {fprintf(f,"equal\npushi 0\nequal\n");}
|Exp '>''=' Exp  {fprintf(f,"supeq\n");}
|Exp '<''=' Exp  {fprintf(f,"infeq\n");}
|Exp '<' Exp  {fprintf(f,"inf\n");}
```

```
|Exp '>' Exp  {fprintf(f,"sup\n");}
;


Exp  : Termo
|Exp '+' Termo   {fprintf(f,"add\n");}
|Exp  '-' Termo  {fprintf(f,"sub\n");}
|Exp '|''|' ExpLog          {fprintf(f, "add\n jz endCond%d:nop\n",get(s));}
;


Termo : Fun
| Termo '/' Fun  {fprintf(f,"div\n");}
| Termo '*' Fun  {fprintf(f,"mul\n");}
| Termo '&''&' ExpLog       {fprintf(f, "pushi 1\nequal\njz endCond%d: nop\n",get(s));}
;

Fun      : num                        {fprintf(f, "pushi %d\n",$1 );}
| Var                       {Endereco a = getEndereco($1.var_nome); fprintf(f, "push%c %d\n",a.tipoVar, a.add

| Var '['Exp ']'            {Endereco a = getEndereco($1.var_nome);
fprintf(f, "push%cp\npush%c %d\npadd\n",(a.tipoVar=='l')?'f':'g', a.tipoVar, a.addr);
fprintf(f, "loadn\n");}

| Var  '['Exp ']' '['Exp ']'
| IdFun                     {funcaoExiste($1); fprintf(f, "pushi 0\n");}
'(' FunArgs')'              {fprintf(f, "call %s\n",$1); fprintf(f, "pop%d\n",numeroArgumentos());}

| '(' Exp ')'
;

FunArgs     :
| FunArgs2
;

FunArgs2    : Exp                    {proximoArgumento(_INTS);}
| FunArgs2 ',' Exp      {proximoArgumento(_INTS);}
;

TestExpLog  : '(' ExpLog ')'
;


%%

int testeMatriz(int linha, int coluna) {

if(linha<=0) {

return -1;
}

else {
if(coluna<=0) {
```

```
return -1;

}

else return 1;


}
}

int testeColuna(int linha) {

if(linha<=0) {

return -1;
}

else return 1;
}



#include "lex.yy.c"



/*
int yyerror(char* s) {
printf("\n\x1b[10;01m%s (line %d) \x1b[0m\n", s, yylineno);
return 0;
}
*/


int yyerror(char *s) {

fprintf(stderr,"ERRO: Syntax LINHA: %d MSG: %s\n",ccLine,s);
return 0;
}


/*

int  yyerror(char *s){
fprintf(stderr,"ERRO: Syntax LINHA: %d MSG: %s\n",ccLine,s);
exit(0);
return 0;
}

*/

/*
```

```
void inicio()
{
s = initStack();
total = 0;
f = fopen("assembly.out", "w");
}
*/


int main(int argc, char* argv[]){
total=0;
lvars=0;
vars=0;
initVGlobalMap();
s=initStack();

f=fopen("assembly.txt","w+");
yyparse();
fclose(f);
free(s);
return 0;
}
```

Apêndice C

Código do Programa

```
#ifndef __COMPILADOR_H__
#define __COMPILADOR_H__

typedef struct variavel *Variavel;

typedef struct funcao *Funcao;

typedef enum eTipo{_VOID,_INTS,_INTA,_INTM}Tipo;

typedef struct sEndereco{
int addr;
char tipoVar;
Tipo tipo;
} Endereco;

int initVGlobalMap();
Funcao existeFuncao(char * func);
int inserFuncao(Tipo tipo,char * name);
int decVar(char* varName, int linha,char tipo);
int decArgumentos(char * nome);
int funcaoExiste(char * nome);
int proximoArgumento(Tipo type);
int numeroArgumentos();
void fim();
void decFunArgRefresh();
int decFunRetAddr();

Endereco getEndereco(char * varName);
```