Relatório de Processamento de Linguagens
Trabalho Pŕatico nº2 - Compilador Yacc

Universidade do Minho, Escola Engenharia
MIEI - Grupo 61

Ana Fernandes A74321

José Cunha A74702

Miguel Miranda A74726

11 de Junho de 2017

Conteúdo

Capítulo 1

Introdução

Este último projeto, associado à unidade curricular de Processamento de Linguagens, tem como objetivo principal a criação de um compilador desenvolvido em Yaac, aplicado a uma linguagem de programação criada pelo grupo e cujo resultado do compilador deve ser um conjunto de instruções assembly válidas para uma Máquina Virtual disponibilizada para testes.

O projeto é iniciado com o desenvolvimento da estrutura de uma linguagem imperativa simples, inspirada na linguagem *C*. Posteriormente foi criada uma Gramática Independente do Contexto (GIC), em **Yacc**, que representa-se a sintaxe e regras da linguagem criada.

Em simultâneo com a GIC foi também desenvolvida a especificação em *Flex* que permiti-se reconhecer os símbolos terminais da linguagem desenvolvida. Após a criação da Gramática Independente de Contexto, foram definidas uma conjunto de ações semânticas, para cada uma das produções, concebendo assim uma Gramática Tradutora (GT) para o contexto da linguagem desenvolvida.

Estas ações semânticas realizam as operações necessárias, conforme a produção em que se inserem, para gerar as instruções assembly que irão correr na máquina virtual.

Foram ainda criados ficheiros de input para os programas solicitados no enunciado, sendo que a GT desenvolvida irá então fazer a transformação de um programa escrito em conformidade com a linguagem criada, para um conjunto de instruções em Assembly que irão ser usadas na Máquina Virtual.

O presente relatório encontra-se assim dividido em cinco capítulos, abordando com detalhe as decisões tomadas para cada uma das fases do projeto acima referidas. No final serão apresentado alguns exemplos de teste assim como uma conclusão fase à aprendizagem e dificuldades obtidas ao longo do desenrolar do projeto.

Capítulo 2

Linguagem desenvolvida

2.1 Estrutura geral

Para o processo de criação de uma linguagem que permita realizar operações semelhantes às realizadas em linguagens de programação imperativas, tomou-se como inspiração a linguagem nativa **C**, simplificando alguns dos seus conceitos e funções.

Para escrever um programa na linguagem desenvolvida, o código deve iniciar-se pela palavra PROGRAMA e ter contido dentro de chavetas as declarações e atribuições de variáveis, seguido das instruções a executar, que representam o corpo do programa. Para qualquer um destes campos, cada linha de código deve terminar com o caractere **;** tal como decorre na linguagem **C**.

De uma forma geral, esta *tag* PROGRAMA representaria o nome de uma função na linguagem **C** mas que na linguagem criada será sempre a mesma, independente do conteúdo ou funcionalidade do código escrito. Na implementação realizada não existe a possibilidade de identificar diferentes funções dentro do mesmo ficheiro, impossibilitando assim a chamada de outros métodos dentro do corpo de uma função.

```
PROGRAMA {
    // Declarações
    // Atribuições
    // Código
}
```

2.2 Declaração de variáveis

Para a declaração de variáveis foi reservada a zona inicial do programa.

Desta forma, todas as variáveis que se pretendam utilizar, devem ser previamente declaradas logo no inicio do programa. Devido à forma como foi implementada a gramática em torno da linguagem, uma variável será identificada por uma das 26 letras do alfabeto, não podendo ser utilizadas outras designações para se referir a variáveis.

O processo de declaração de variáveis segue a mesma estrutura que uma declaração em **C**:

- Em primeiro lugar deve ser definido o tipo da variável. Para isso foi reservada a palavra TIPO, que nesta solução inicial desenvolvida representará o tipo inteiro da linguagem **C**;
- Seguido do tipo deve surgir uma letra, que represente a variável a declarar e reservar;
- Depois de indicada a letra que representa a variável, deverá existir o caractere **;** (ponto e virgula), para representar o fim de uma declaração, ou então o caractere **,** (virgula), caso se pretenda assim continuar a declaração de mais variáveis dentro da mesma linha de código e associadas ao mesmo tipo.

Na abordagem implementada, a declaração e atribuição de variáveis não se pode conjugar dentro da mesma expressão, existindo assim zonas diferentes para realizar cada uma das tarefas.

O seguinte excerto de código procura demonstrar as possíveis formas de declarações possíveis no programa:

```
PROGRAMA {
    // Declarações
    TIPO x;
    TIPO a,b,c;
    TIPO k;
    // Atribuições
    // Código
}
```

2.3 Atribuição de valores a variáveis

Como já referido, o bloco de atribuição de valores a variáveis só será aceite se realizado depois do bloco de declaração de variáveis. Desta forma não é possível atribuir um valor a uma variável não declarada, tal como decorre na linguagem imperativa **C**. Ao longo das instruções do programa, caso necessário, podem ser atribuidos valores a variaveis.

A atribuição de um valor a uma variável é realizado através do caratere = (igual), tal como na linguagem **C**. É também aceite a atribuição implícita de valores a uma variável por exemplo através do comando *read X*, onde o valor lido através do stdin será colocado na variável X.

Na implementação desenvolvida, na fase de tratamento de um programa através do compilador em yaac, foi tido o cuidado de apenas aceitar a atribuição de valores a variáveis que fossem previamente identificadas e declaradas.

O seguinte excerto procura demonstrar a forma simples e minimalista de atribuir um valor a variáveis:

```
PROGRAMA {
    // Declarações
    TIPO x;
    TIPO a,b,c;
    TIPO k;
    // Atribuições
    x = 3;
    b = 1;
    // Código
}
```

2.4 Estruturas condicionais

2.4.1 Condição if(cond) then... else...

A linguagem desenvolvida permite ainda a declaração de regiões condicionais, nomeadamente do tipo **if then else**. A sua declaração segue a mesma forma que as declarações condicionais na linguagem **C**. A condição if pode ou não ser seguida por uma clausula do tipo *else* e uma condição representa qualquer expressão que se possa derivar num valor booleano. Tanto a condição if como a condição else, caso exista, deve ser seguida de um conjunto de instruções dentro de chavetas.

O seguinte excerto de código demonstra algumas das possíveis declarações aceites:

```
PROGRAMA {
    // Declarações
    TIPO x, d;
    // Atribuições
    x = 3;
    // Código
    if( x == 3){     d = 3;   }
    else{            d = 5:   }

    if(x > d){       d = 10; }
}
```

2.5 Métodos disponíveis

Depois de abordados os três principais blocos de código necessários para implementar um programa na linguagem desenvolvida, referimos ainda alguns dos métodos relevantes que existem na linguagem.

2.5.1 Imprimir no stdout

Para realizar a impressão de uma string, através do stdout, ao utilizador, foi criado o método *print*. Este método procura ser uma versão simplificada da função **printf** existente na linguagem **C**.

Apesar de ser possível inserir uma variável ou um valor inteiro depois da string a imprimir (não dará erro no compilador criado), na geração de código assembly da função print não se realiza essa funcionalidade.

Assim, na implementação desenvolvida apenas será imprimida a string integral no stdoud do utilizador. Numa fase de desenvolvimento posterior facilmente seria implementada essa funcionalidade de além da string agregar um valor inteiro ou uma variável declarada.

O seguinte exemplo permite demonstrar o uso da função:

```
PROGRAMA {
    // Declarações
    TIPO x;
    // Atribuições
    // Código
    read x;
    if(x == 3){
        print "O valor de x é 3";
    }
}
```

2.5.2 Leitura stdin

Caso se pretenda realizar a leitura de um valor inteiro, através do input introduzido pelo utilizador, está disponibilizada a função **read**.

Esta função permite então ler do *stdin* e colocar o valor lido na variável indicada. Tal como na atribuição de valores a variáveis, a atribuição de um valor lido a partir do *stdin* para uma variável exige que essa variável seja previamente declarada e conhecida.

```
PROGRAMA {
    // Declarações
    TIPO x;
    // Atribuições
    // Código
    read x;
}
```

Capítulo 3

Gramática tradutora

3.1 Gramática independente contexto

Depois de apresentada a estrutura genérica do código de num programa feito na linguagem criada, apresentamos em seguida a gramática desenvolvida para representar a sintaxe, forma e estrutura das frases válidas na linguagem.

No desenvolvimento da gramática foi tido o cuidado de, sempre que necessário, realizar a recursividade de uma produção à direita, usando uma nova produção para esse efeito, evitando assim os possíveis conflitos que decorrem dos algoritmos LL(1).

- T = { TIPO, WHILE, PRINT, READ, RETURN, IF, ELSE, WHILE, PROGRAMA, valor, variavel, string }
- N = { Prog, Decls, RDecls, Decl, DeclsVar, RDeclsVar, DeclVar, Atribs, RAtribs, Atrib, Codigos, RCodigos, Codigo, Cond, Opcao, exp, termom Conj, RConj }
- S = PROGRAMA
- Producões:

```
Prog        : PROGRAMA '{' Decls Codigos '}'
Decls       : Decl RDecls
            |
RDecls        : Decls
Decl        : TIPO DeclsVar ';'
DeclsVar    : DeclVar RDeclsVar
RDeclsVar   : ',' DeclsVar
            |
```

```
DeclVar     : variavel
            | variavel '[' valor ']'
Atribs      : Atrib RAtribs
RAtribs     : Atribs
            |
Atrib       : variavel '=' exp ';'
Codigos     : Codigo RCodigos
RCodigos    :
            | Codigos
Codigo      : PRINT string Opcao ';'
            | READ variavel ';'
            | IF '(' Cond ')' '{' Codigos '}'
            | IF '(' Cond ')' '{' Codigos '}' ELSE
            ↪  '{' Codigos '}'
            | RETURN exp ';'
            | WHILE '(' Cond ')' '{' Codigos '}'
            | Atribs
            |
Cond        : exp '='"="' exp
            | exp '>'"="' exp
            | exp '<'"="' exp
            | exp '!'"="' exp
            | exp '<' exp
            | exp '>' exp
            | exp
            | Conj
Opcao       : termo
            |
exp         : termo
            | exp '+' termo
            | exp '-' termo
            | exp '*' termo
            | exp '/' termo
termo       : variavel
            | valor
Conj        : '(' Cond ')' RCo
RConj       :
            | '&'"&' Conj
```

3.2 Especificação Flex

Estritamente associado com a gramática criada, apresentamos a especificação em flex de forma a reconhecer todos os símbolos terminais da GIC desenvolvida.

Sempre que for necessário utilizar o valor de um dos símbolos reconhecidos, é utilizado um dos campos da estrutura *yylval* que partilhe o mesmo tipo da variável a guardar.

```
%option noyywrap yylineno

%{    %}

%%
[-\,;\(\)\{\}=<>\&\*!+][\t \n]?    { return yytext[0];}

[-]?[0-9]+      { yylval.i = atoi(yytext);
                  return valor;}
"TIPO"          { return TIPO;  }
(?i:while)      { return WHILE;  }
(?i:print)      { return PRINT;  }
(?i:read)       { return READ;   }
(?i:return)     { return RETURN; }
(?i:if)         { return IF;     }
(?i:else)       { return ELSE;   }
(?i:programa)   { return PROGRAMA; }
[a-z]           { yylval.c = yytext[0];
                  return variavel; }
\"[^"]*\"       { yylval.s = strdup(yytext);
                  return string;}
.               { }
%%
```

3.3 Ações semânticas

Depois de criada a gramática da linguagem foi introduzido um conjunto de ações semânticas que permitissem, à medida que se reduzem as produções, gerar o pseudo código assembly correspondente ao programa processado pelo compilador yacc.

A introdução destas ações semânticas à gramática independente do contexto apresentada permite assim implementar uma gramática tradutora associada ao processamento da pseudo linguagem de programação criada.

A geração de código assembly através das ações semânticas, segue a instruções disponibilizadas pela máquina virtual fornecida e foi construida conforme o tipo de código reconhecido no programa interpretado.

Para guardar todas as instruções de um programas, foi utilizada como estrutura de dados um array dinâmico, sendo cada nova instrução assembly vista como uma string que é colocada no final deste array. Sobre o array dinâmico foram criadas funções que permitam:

- Determinar a sua capacidade atual do array dinâmico;
- Determinar o índice da posição atual disponível para inserir um novo elemento;
- Obter um elemento na posição ***i*** da estrutura de dados.

Foram ainda criados os habituais métodos de inicialização da estrutura, alocando memória para os seus campos e libertação da memória reservada. Regra geral cada símbolo terminal leva à geração de uma string que contem nela a instrução assembly equivalente à produção reduzida.

Depois de atingido o end of file (EOF) do programa, se todas as produções da gramática transitarem entre si de forma válida, a principal ação semântica é desempenhada pela primeira produção. Esta ação consiste em iterar o array dinâmico construido e colocar cada um dos seus elementos numa nova linha do ficheiro de output. O ficheiro com o código assembly gerado, que terá como nome *saida.txt*, terá assim tantas linhas quantos elementos existentes no array dinâmico construido e cada linha representa uma instrução assembly atómica.

Relembrando a primeira produção da gramática criada, facilmente se delimita a zona de inicialização de variáveis, inicio do código do programa e término do programa. Desta forma, depois das instruções assembly que derivem do processamento do símbolo não terminal *Decls*, deverá surgir a instrução assembly *start* indicando assim o fim da declaração de variáveis e o inicio do programa.

De forma semelhante, depois de geradas as instruções assembly que derivem das iterações sobre o símbolo não terminal *Codigos*, deve surgir a instrução assembly *stop* para indicar o fim do programa a executar.

Fruto de uma inspiração inicial na linguagem *C*, existe ainda a possibilidade de retornar um valor no fim de um programa. Apesar de nesta fase não se ter implementado a chamada de funções externas dentro de um programa, antes de qualquer instrução assembly derivada do programa lido, deve ser feita a instrução assembly *push 0*, de forma a reservar um endereço na stack para eventualmente retornar um valor.

```
%{
(...)
ARRAY_DINAMICO instrucoes;
%}

%%
    Prog        : PROGRAMA '{' Decls Codigos '}'
                { insereElementoAD(instrucoes, "pushi 0\n");
                  insereElementoAD(instrucoes, $4);
                  insereElementoAD(instrucoes, "start\n");
                  insereElementoAD(instrucoes, $5);
                  insereElementoAD(instrucoes, "stop\n");
                  contaVariaveis++;
                  escreve_assembly(instrucoes);
                }
    (...)
%%
```

3.3.1 Declaração e atribuição de valores a variáveis

Tal como referido anteriormente, a declaração de variáveis só pode ser realizada sobre termos que representem uma das 26 letras do alfabeto.

Para o controlo das declarações de variáveis foram criados dois arrays de 26 posições, onde cada índice representa uma das possíveis variáveis a declarar e atribuir. Para determinar em que índice estão as informações relativas a uma variável é feito o calculo (ASCII-Variavel - 97). Por exemplo, informações sobre a variável 'a' estarão no índice 0 dos dois arrays, e a variável 'z' terá as suas informações na posição 25 dos arrays.

Um dos arrays, designado como ***declarados***, é utilizado para determinar

se a variável se encontra ou não declarada, contendo no índice que corresponde à variável o valor (-1) se não declarada ou (1) se declarada.

O outro array, designado por ***indices***, é utilizado para determinar qual o índice correspondente ao endereço da variável declarada na stack da máquina virtual. Este índice é determinado a partir de um contador que é sempre incrementado quando se adiciona uma nova variável à stack.

Por exemplo, se surgir uma declaração da variável **x**, é primeiro visto o valor existente na posição ***declarados[23]***, de forma a saber se a variável já se encontra ou não declarada. Se não se encontrar declarada, a posição ***declarados[23]*** irá passar a ter o valor (1) e a posição ***indices[23]*** terá o valor do próximo endereço da stack disponível para atribuir.

A nível de geração de código assembly, na linguagem desenvolvida não se permite a atribuição de valores logo no momento da declaração da variável. Assim, nas declarações de uma variável o código assembly será sempre do tipo *push 0*, reservando um endereço na stack dá maquina virtual.

```
Linguagem C                Linguagem Assembly
 Tipo x;      -------->        push 0
```

Quando posteriormente surgir uma atribuição sobre a variável **x**, será necessário carregar a mesma para o topo da stack e carregar para ela o valor indicado na atribuição. Esta sequência de comandos, na linguagem assembly, será do tipo:

```
Linguagem C                 Linguagem Assembly

 x = 7;       -------->       pushi 7
                              storeg N
```

Onde N representa o endereço gp[n] onde se localiza a variável **x** e é o mesmo índice que existe na posição indices[23].

3.3.2 Condições if then else

Para gerar o código assembly correspondente a um conjunto de instruções condicionais do tipo if, foram utilizadas as operações de controlo da stack, nomeadamente as instruções que permitem a alteração do registo pc, induzindo assim jumps para zonas especificas da stack.

Tal como decorria com as variáveis, existe também um contador de expressões if, permitindo identificar cada zona de código if na stack de forma inequívoca, com uma designação intuitiva. Este contador será incrementado

sempre que for reduzida a produção relativa a um if, com ou sem clausula else, na gramática da linguagem.

Para implementar uma condição do tipo **if(cond) then**, é necessário:

- marcar na stack a indicação de um salto condicional para o fim do if (jump if zero - jz), caso a condição **cond** falhe;
- identificar a zona do fim do if, que indica a continuação do restante código a executar do programa.

O seguinte exemplo procura demonstrar este processo, onde N representa o valor do contador de ifs quando o if é reconhecido:

```
Linguagem C                Linguagem Assembly

 if(cond){                     cond
  Codigo_If                    jz fim_if_N
 }             ------->        Codigo_If
 Codigo                       fim_if_N:
                              Cogido
```

Um processo semelhante é utilizado para gerar o código assembly de uma condição if then else:

- Caso a condição **cond** do if falhe, o apontador da stack deve saltar para a zona da stack referente à clausula do else;

  Caso a condição do if seja válida, deve ser executado o código dentro da clausula if e posteriormente feito o salto para a zona da stack que indica o final do if, evitando assim a execução da clausula else;

- Na zona da stack que indica o inicio do else, deve ser listado o código a executar dentro da clausula else, terminando posteriormente na zona da stack que indica o fim do if.

O seguinte exemplo procura demonstrar este processo:

```
Linguagem C                Linguagem Assembly

 if(cond1){                    cond
  Codigo_If                    jz else_if_N
 }else{         -------->     Codigo_If
  Codigo_Else                  jump fim_if_N
 }                             else_if_N:
 Codigo                        Codigo_Else
                               fim_if_N:
                               Codigo
```

3.3.3 Ciclos condicionais while

Para gerar código assembly que represente a estrutura de uma condição while, foram tidos os mesmos conceitos utilizados para as condições if.

Novamente, existe um contador de condições while, de forma a identificar de forma sugestiva cada zona while na sequência de instruções assembly geradas.

Em instruções assembly, uma condição while deve ter a seguinte estrutura:

- Uma zona que identifique o inicio das instruções da condição while;
- Caso a condição do ciclo não se verifique no inicio de cada iteração, deve ser feito o jump para o final do ciclo;
- Caso a condição do ciclo seja confirmada, são executadas as instruções do corpo do ciclo e feito o salto para a zona da stack que representa o inicio do ciclo.

O seguinte exemplo procura demonstrar este processo, onde N representa o valor do contador de condições while quando é feita a redução da produção correspondente à instrução while.

```
Linguagem C                 Linguagem Assembly

                             cond_while_N:
While(cond){                 cond
   Codigo_While  ------>     jz fim_while_N
}                            Codigo_While
Codigo                       jump cond_while_N
                             fim_while_N:
                             Codigo
```

3.3.4 Leitura do stdin e escrita no stdout

Tal como referido na descrição dos métodos existentes na linguagem criada, existe a possibilidade de solicitar um valor do tipo inteiro ao utilizador do programa, através do standard input, assim como a impressão de uma mensagem contida numa string, através do standard output.

Estas duas funcionalidade levaram à criação de duas produções cujas ações semânticas podem ser descritas da seguinte forma:

- Para escrever uma mensagem no stout, é necessário carregar para o topo da stack o conteúdo da mensagem a imprimir. Este processo é

realizado através da instrução *pushs* que arquiva uma string na zona de strings e empilha o seu endereço na stack. Para realizar a impressão da string no stdin, é utilizada a instrução writes, existente no conjunto de instruções da máquina virtual, que retira o endereço de uma string da pilha de instruções, imprimindo-a no standard output.

- Para realizar a leitura de um valor através do standard input, é utilizada a instrução em assembly *read*. Esta instrução lê uma string do teclado e empilha o seu endereço na stack. Contudo, como no contexto da linguagem desenvolvida as variáveis só são do tipo inteiro, é necessário converter esta string lida para um inteiro e atribui-la à variável indicada para guardar o valor lido.

  Para realizar a conversão de uma string para o tipo inteiro é utilizada a função *atoi*, que automaticamente retira da stack o endereço de uma string e coloca na mesma posição o endereço do inteiro correspondente. Para atribuir o valor lido e convertido a uma variável previamente declarada, é utilizada a instrução storeg.

  Para determinar qual é o endereço da variável sobre a qual se deseja atribuir o valor lido, é utilizado o valor da posição ***indices[i]***, onde ***i*** representa o índice da variável ***X*** onde se pretende guardar o valor lido.

O seguinte exemplo procura descrever a conversão para linguagem assembly destes dois métodos.

```
Linguagem C                    Linguagem Assembly

  TIPO x;                           push 0

  print "ola mundo!";               pushs "ola mundo!"
                               --->  writes
  print "insira um valor";          pushs "insira um valor"
  read x;                           writes
                                    read
                                    atoi
                                    storeg 1
```

Capítulo 4

Testes

4.1 Lados de um quadrado

O seguinte programa procura exemplificar um processo de leitura de 4 números inteiros e testar se podem ser os lados de um quadrado. Conforme o resultado do teste, imprime no stdout uma mensagem de sucesso ou insucesso, respetivamente.

4.1.1 Programa

```
PROGRAMA {
    TIPO x, y;
    TIPO z, k;
    print "Lados de um quadrado";
    print "Insira o primeiro numero:";
    read x;
    print "Insira o segundo numero:";
    read y;
    print "Insira o terceiro numero:";
    read z;
    print "Insira o quarto numero:";
    read k;
    if((x==y) && (x==z) && (x==k)) {
        print "Os lados sao de um quadrado.";
    }
    else {
        print "Os lados nao sao de um quadrado.";
    }
}
```

4.1.2 Código Assembly gerado

```
pushi 0
pushi 0
pushi 0
pushi 0
pushi 0
start
pushs "Lados de um quadrado"
writes
pushs "Insira o primeiro numero:"
writes
read
atoi
storeg 1
pushs "Insira o segundo numero:"
writes
read
atoi
storeg 2
pushs "Insira o terceiro numero:"
writes
read
atoi
storeg 3
pushs "Insira o quarto numero:"
writes
read
atoi
storeg 4
pushg 1
pushg 2
equal
pushg 1
pushg 3
equal
pushg 1
pushg 4
equal
jz else_if_0
pushs "Os lados sao de um quadrado."
```

```
writes
jump fim_if_0
else_if_0:
pushs "Os lados nao sao de um quadrado."
writes
fim_if_0:
stop
```

4.2 Maior número

O seguinte programa procura descrever o processo onde é solicitado o número de valores que o utilizador deseja introduzir. Posteriormente são lidos esses N valores, através do stdin e determinado qual o maior valor introduzido.

4.2.1 Programa

```
PROGRAMA {
    TIPO x, n, m;
    TIPO i;
    i = 1;
    print "Insira o numero de inteiros a ler.";
    read n;
    print "Insira um numero:";
    read m;
    while(i<n){
        print "Insira um novo numero:";
        read x;
        if(x > m){
            m = x;
        }
    }
    print "O maior valor lido foi " m;
}
```

4.2.2 Código Assembly gerado

```
pushi 0
pushi 0
pushi 0
```

```
pushi 0
pushi 0
start
pushi 1
storeg 4
pushs "Insira o numero de inteiros a ler."
writes
read
atoi
storeg 2
pushs "Insira um numero:"
writes
read
atoi
storeg 3
cond_while_0:
pushg 4
pushg 2
inf
jz fim_while_0
pushs "Insira um novo numero:"
writes
read
atoi
storeg 1
pushg 1
pushg 3
sup
jz fim_if_0
pushg 1
storeg 3
fim_if_0:
jump cond_while_0
fim_while_0:
pushs "O maior valor lido foi "
writes
stop
```

4.3 Conclusao e trabalho futuro

O segundo trabalho prático realizado nesta unidade curricular foi, na apreciação do grupo, positivo. No entanto, face aos anteriores projetos onde funcionalidades adicionais foram acrescentadas aos enunciados pedidos, nesta fase decorreram dificuldades notórias a nível da geração do código assembly, que levaram à não finalização e implementação de todos os requisitos solicitados.

Esta falha na implementação de todos os requisitos deve-se a uma abordagem inicial demasiado complexa perante o problema do projeto.

Ainda assim, foi desenvolvida uma gramática tradutora em *Yacc* capaz de suportar várias operações comuns em linguagens de programação imperativas. A linguagem criada aceita assim expressões aritméticas e operações relacionais sobre valores inteiros ou aplicadas a variáveis previamente declaradas no programa.

Além disso, são ainda reconhecidas construções condicionais tais como o ciclo while, bem conhecido nas linguagens de programação imperativas. Contudo, na implementação atual da gramática do projeto não é possível o reconhecimento de chamadas de funções com argumentos e declaração de variáveis arrays.

Como pontos a melhorar no trabalho realizado, destaca-se a extensão da gramática produzida para suportar chamadas de funções e de arrays tal como foi referido, e ainda outros blocos de códigos condicionais como os ciclos for ou operadores *switch*.

A geração de código em assembly foi parcialmente conseguida, obtendo com sucesso a geração de código assembly depois de interpretado uma frase da linguagem criada. Contudo, apesar do código gerado correr corretamente na máquina virtual disponibilizada, os programas não desempenham corretamente aquilo que era esperado. Além deste aspeto, existem alguns problemas compilando outros exemplos criados. Em algumas situações, o programa escrito é corretamente reconhecido pela GT, seguindo assim a forma e sintaxe da linguagem, mas a execução das ações semânticas leva a falhas de compilação, devido a problemas relacionados com alocação de memória. Apesar de todas as fases de alocação e libertação de memória a variáveis e estruturas terem sido revistas, alguns dos exemplos criados continuaram a dar erros de compilação, nomeadamente erros do tipo *memory corruption*.

Tendo sido analisados estes casos a fundo, ainda que sem sucesso na sua resolução, restou-nos entregar o projeto final conscientes de que alguns dos exemplos criados não estão totalmente funcionais.

Contudo, e eventualmente mais relevantes que as limitações do projeto apresentado, é reconhecido entre o grupo que todos os conceitos associados

com expressões regulares, GICs, ações semânticas e modo de funcionamento de um compilador simples, foram corretamente percebidos e o conhecimento sobre processamento de linguagens enriquecido.

Capítulo 5

Referências

Stephen C. Johnson. (2017). Yacc: Yet Another Compiler-Compiler.
[online] Available at:
http://epaperpress.com/lexandyacc/download/yacc.pdf
[Accessed 7 Jun. 2017].

Tom Niemann. (2017). LEX YACC TUTORIAL
[online] Available at:
http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf
[Accessed 9 Jun. 2017].