Processamento de Linguagens
(3º ano de Licenciatura em Engenharia Informática)

Trabalho Prático 1

Relatório de Desenvolvimento

André Geralades (67673) Patrícia Barros (67665)
Sandra Ferreira (67709)

3 de Junho de 2015

Resumo

Este relatório descreve todo 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 e da respetiva **GIC** seguida do desenvolvimento de um compilador para a mesma que gera pseudo-código *Assembly*.

Conteúdo

Capítulo 1

Introdução

A resolução deste trabalho prático passa pelo desenvolvimento de um Filtro de Texto em Flex para gerar ficheiros em HTML. Para isso utilizamos as técnicas leccionadas nas aulas da Unidade Curricular de Processamento de Linguagens. Pretendemos portanto com este trabalho aprimorar as nossas capacidades de escrever *Expressões Regulares (ER)* e também a nossa experiência na utilização da linguagem de programação C.
Neste relatório apresentamos todos os passos e decisões tomadas durante todo o processo, descrevemos as estruturas criadas para guardar o texto extraído pelo filtro e também uma apresentação do produto final (em HTML) obtido com a utilização do filtro criado por nós.

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: possuí um primeiro capítulo que faz uma contextualização ao assunto tratado neste trabalho, seguindo-se uma introdução onde são apresentadas as metas de aprendizagem pertendidas.
Posteriormente é exposto o tema escolhido para desenvolver o trabalho e as tarefas que nele estão envolvidas.
Imediatamente após, são exibidas as expressões regulares definidas para extrair do ficheiro XML as informações para a construção da página HTML, mostrando também os estados da aplicação e os módulos desta.
Não menos importante, seguem-se os capitulos de apresentação das estruturas de dados usadas no desenvolvimento do trabalho e dos testes realizados à aplicação com os devidos resultados. Por último, 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.

Capítulo 2

Análise e Especificação

2.1 Enunciado

Pretende-se que comece por definir uma linguagem de programação interativa simples (**LPIS**), a seu gosto.
Apenas deve ter em consideração que a **LPIS** terá de permitir manusear variáveis do tipo inteiro (escalar ou *array*) e realizar as operações básicas como atribuições de expressões a variáveis, ler do *standard input* e escrever no *standard output*. As instruções vulgares para controlo de fluxo de execução - *condicional* e *cíclica*- devem estar também previstas.
Sobre inteiros, estão disponíveis as habituais operações aritméticas, relacionais e lógicas, bem como operação de indexação sobre *arrays*.
Como é da praxe neste tipo de linguagens, as nossas variáveis deverão ser declaradas no inicio 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 indefinido.
Desenvolva, então, um compilador para a **LPIS**, com base na **GIC** criada acima e recurso ao Gerador **Yacc/Flex** ou **AnTLR**.
O compilador da **LPIS** deve gerar **pseudo-código**, Assembly da Máquina Virtual VM cuja documentação completa será disponibilizada no Bb.

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

Nos próximos capítulos iremos descrever todas as decisões tomadas no decorrer deste trabalho.

3.1 Desenho da Linguagem

O principal princípio que seguimos aquando de decidir as caraterísticas da nossa linguagem foi a simplicidade desta, desenhando-a de forma a que os passos seguintes do trabalho não fossem excessivamente complicados. Posto isto, decidimos que a nossa linguagem suportaria apenas os requisitos definidos no enunciado.
Concluímos que nos seria útil que qualquer programa da nossa linguagem estivesse dividido em duas partes: as declarações de variáveis e o corpo, constituído por instruções. Percebemos também que seria mais fácil se tanto a operação condicional como o ciclo tivessem sempre marcadores do seu início e fim, e escolhemos neste caso usar como marcadores palavras reservadas para os delimitarem.
Assim, um programa tem **obrigatoriamente** de iniciar com a palavra reservada **BEGIN** e fazer-se seguir de uma ou muitas declarações de variáveis (nunca poderá haver um programa sem nenhuma declaração pois consideramos que um programa sem variáveis não faz sentido), declarações estas que são constituídas por um tipo e uma ou mais variáveis (palavras) separadas por vírgulas. No caso de ser um *array* terá também de ser declarado o seu tamanho (um número entre parenteses colocado logo a seguir ao tipo). Cada declaração termina obrigatoriamente com um **;**. Os tipos das variáveis são palavras reservadas, podendo ser **INT** ou **ARRAY**.
Após terminadas as declarações damos então início ao corpo do programa, através da palavra reservada **BODY**.

O corpo do programa é constituído por uma ou mais instruções (mais uma vez, também o corpo não pode ser vazio pois achamos que isso não faria sentido). Cada instrução pode ser de um de cinco tipos diferentes: atribuição, condição, ciclo, escrita ou leitura e termina sempre com um ponto e vírgula.
Uma atribuição consiste numa variável seguida de um sinal de igual que precede uma expressão. Neste caso uma variável pode ser uma palavra ou uma palavra seguida de uma expressão entre parênteses (para a operação de indexação de um *array*). Por sua vez uma expressão é um termo ou uma expressão seguida de um operador de adição que precede um termo, sendo que um termo é um fator ou um termo seguido de um operador multiplicativo que precede um fator. Um fator pode ser uma palavra, um número ou uma condição entre parênteses, sendo que uma condição neste caso é uma expressão ou duas expressões separadas por um operador relacional.
Uma condição é sempre iniciada pela palavra reservada **IF** que precede uma ou várias instruções e pode ou não ser seguida da palavra reservada **ELSE** seguida de mais instruções. Uma condição termina sempre com a palavra reservada **ENDIF**.
Um ciclo é sempre iniciado pela palavra reservada **WHILE** seguida de uma condição entre parênteses que precede uma ou mais instruções e termina sempre com a palavra reservada **ENDWHILE**. As instruções de leitura escrita são representadas por duas palavras reservadas: **READ** e **WRITE** respetivamente. No caso da leitura a palavra reservada segue-se de uma variável (inteira ou a posição de um *array*) entre parênteses, e no caso da escrita segue-se de uma expressão entre parênteses.
Os símbolos que utilizamos para os operadores aditivos, multiplicativos e relacionais são os seguintes:

```
opA  "+"|"-"|"||"
opM "&&"|"*"|"/"
opR ">>"|"<<"|">="|"<="|"=="|"|=|"
```

Para melhor compreensão da descrição acima segue-se um exemplo de um programa simples escrito na nossa linguagem.

```
1  BEGIN
2  INT x,y,y;
3  ARRAY(100) vect , vects;
4  BODY
5  x=1;
6  y=1+2;
7  IF (x>>5) y=y+1; ENDIF;
8  x=2;
9  WHILE (y<<2) y=y+1; IF (x>>5) w=y+1; ENDIF; ENDWHILE;
10 IF (x==1) y=y+1; y=2; ELSE y=3; ENDIF;
11 END
```

3.2 Desenho da Gramática

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: programa, desclaracoes, declaracao, tipo, variaveis, variavel instrucoes, instrucao, atribuicao, var, expressao, termo, fator, condicao, ciclo e cond.
Por sua vez os símbolo terminais são: num, pal, IF, ENDIF, WHILE, ENDWHILE, BEGINP, MIDDLE, ENDP, INT, ARRAY, OPM, OPR, OPA, ";", ",", "(","")"e "=".
Na concepção da nossa gramática utilizamos em todos os casos recursividade à esquerda. Segue-se então a gramática gerada:

```
1  programa          :       BEGINP
2
3  declaracoes       : declaracao
4  | declaracoes  declaracao
5  ;
6
7  declaracao        : tipo variaveis ';'
8  | tipo '(' num ')' variaveis ';'
9  ;
10
11 tipo              : INT
12 | ARRAY
13 ;
14
15 variaveis         :       variavel
16 |       variaveis ',' variavel
17 ;
18
19 variavel          :       pal
20 ;
21
22 instrucoes        :       instrucao ';'
23 |       instrucoes  instrucao ';'
24 ;
25
26 instrucao         :       atribuicao
27 |       condicao
28 |       ciclo
29 |       READ '(' var ')'
30 |       WRITE '('expressao')'
31 ;
32
33 atribuicao        :       var '=' expressao
34 ;
35
36 var               :       pal
```

```
37 |       pal '('expressao')'
38 ;
39
40 expressao       :       termo
41 |       expressao OPA termo
42 ;
43
44 termo           :       fator
45 |       termo OPM fator
46 ;
47
48 fator           :       var
49 |       num
50 |       '(' cond ')'
51 ;
52
53 condicao        :       IF '('cond')' instrucoes  ENDIF
54 |       IF '('cond')' instrucoes  ELSE instrucoes ENDIF
55 ;
56
57 ciclo           :       WHILE '('cond')' instrucoes ENDWHILE
58 ;
59
60 cond            :       expressao
61 |       expressao OPR expressao
62 ;
```

3.3 Módulos da Aplicação

makefile Ficheiro com a configuração de compilação.
tp2.l Ficheiro com código de um analisador léxico que serve de suporte à nossa gramática.
tp2.y Ficheiro que contém o código **YACC** correspondente à nossa gramática e inclui nesta o código necessário para a geração do código Assembly resultante.
arvore.c Ficheiro onde se encontra o código da estrutura definida para guardar as informações das variáveis encontradas no programa.
arvore.h Ficheiro com a estrutura definida para guardar informações das variáveis e com as assinaturas das funções definidas no arvore.c.

3.4 Estruturas de Dados

Para guardar as informações relativas às variáveis que se encontram nos programas escritos na nossa linguagem foi necessário criar uma estrutura.
A estrutura mais apropriada para este efeito é uma Tabela de Hash devido a ser mais eficiente a procura por *strings*, no entanto, devido à nossa inexperiência em

trabalhar com esse tipo de estrutura, optamos por utilizar uma Árvore Binária de Procura, com a qual estamos mais familiarizados, para este efeito.
Foi necessário guardar, para cada variável declarada, o seu nome, tipo, registo e tamanho.
Decidimos que o tamanho seria 1 para todas as variáveis de tipo inteiro, ficando os *arrays* com o tamanho definido aquando da declaração.
O valor do registo é sequencial, ou seja, existe uma variável global inteira que é incrementada de cada vez que uma variável é declarada. No caso dessa variável ser um *array* o valor do registo é incrementado tantas vezes quanto o tamanho do *array*.
Segue-se o código referente à declaração da estrutura que utilizamos para guardar estas informações:

```
1 typedef struct node {
2 char* nome;
3 char* tipo;
4 int tamanho;
5 int registo;
6 struct node *esq;
7 struct node *dir;
8 } Node, *Tree;
```

Capítulo 4

Testes realizados e Resultados

Capítulo 5

Conclusão

Apêndice A

Código Flex

```
%{
        #include "y.tab.h"
        void yyerror(char *);
%}

num             [0-9]+
pal             [a-zA-Z]+
espacos         [ \t]
opA             "+"|"-"|"||"
opM                     "&&"|"*"|"/"
opR                     ">>"|"<<"|">="|"<="|"=="|"|="
inicio          "BEGIN"
corpo           "BODY"
fim                     "END"
condicao        "IF"
ifelse          "ELSE"
fimCond         "ENDIF"
ciclo           "WHILE"
fimciclo        "ENDWHILE"
inteiro         "INT"
vetor           "ARRAY"
read            "READ"
write           "WRITE"

%%
[,;()=]             { return yytext[0]; }
{inicio}                    { return BEGINP; }
{corpo}                     { return MIDDLE; }
{fim}                       { return ENDP; }
{condicao}                  { return IF; }
{ifelse}                    { return ELSE; }
{fimCond}                   { return ENDIF; }
{ciclo}                     { return WHILE; }
```

```
34 {fimciclo}                    { return ENDWHILE; }
35 {opA}                         { yylval.vals = strdup(yytext)
       ;return (OPA); }
36 {opR}                         { yylval.vals = strdup(yytext)
       ;return (OPR); }
37 {opM}                         { yylval.vals = strdup(yytext)
       ;return (OPM);}
38 {vetor}                       { yylval.vals = strdup(yytext)
       ;return (ARRAY);}
39 {inteiro}                     { yylval.vals = strdup(yytext)
       ;return (INT);}
40 {read}                        { return (READ);}
41 {write}                       { return (WRITE);}
42 {num}                         { yylval.vali = atoi(yytext);
       return num; }
43 {pal}                         { yylval.vals = strdup(yytext)
       ; return pal; }
44
45 .|\n                          { ; }
46
47
48
49 %%
50
51 int yywrap(){
52         return(1);
53 }
```