Processamento de Linguagens (3º ano de Curso)

Trabalho Prático N 2

Relatório de Desenvolvimento

Filipe Costa Oliveira
a57816

30 de Maio de 2016

Conteúdo

Capítulo 1

Introdução

O presente trabalho prático foca-se no desenvolvimento de um compilador, que tem como fonte uma linguagem de alto nível (também esta desenvolvida especificamente para este trabalho prático) , gerando código para uma máquina de stack virtual.
Um compilador comum divide o processo de tradução em várias fases. Para o propósito específico desta unidade curricular iremos focar-nos nas seguintes:

- 1ª Fase de tradução – Análise Léxica, que agrupa sequências de caracteres em tokens. Recorreremos nesta fase à definição das expressões regulares que permitem definir os tokens.
- 2ª Fase de tradução – Reconhecimento(Parsing) da estrutura gramatical do programa, através do agrupamento dos tokens em produções. Recorreremos à definição de uma gramática independente de contexto por forma a definir as estruturas de programa válidas a reconhecer pelo parser. Denote que juntamente com o parsing é realizada a análise semântica, assim como a geração de código associando regras às produções anteriormente descritas.

Começaremos portanto por definir uma linguagem de programação imperativa simples, que chamaremos Algebra. A Algebra permitirá:

- 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. Aos arrays de duas dimensões, por se tratar de uma linguagem algébrica, chamaremos matrizes, dada a fácil associação a este tipo de variável à sua definição análoga da álgebra linear.
- 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.

Na nossa linguagem de programação por questões de estruturação e percepção, teremos como premissa que as varíaveis deverão ser declaradas no início do programa, não podendo haver re-declarações, nem utilizações sem declaração prévia. Não será permitida a declaração e associação de um valor inteiro na mesma instrução. Achamos essa solução pouco elegante. Assim, todas as variáveis terão o valor zero após a declaração.
Será desenvolvido portanto o compilador para a Algebra, com base na GIC criada acima e recurso ao Gerador Yacc/ Flex. O compilador de Algebra irá gerar pseudo-código, Assembly da Máquina Virtual VM cuja documentação completa está disponibilizada em anexo.

Por forma a facilitar e validar o trabalho, à medida que as funcionalidades forem descritas serão apensados exemplos ilustrativos.
Por fim, serão apresentados um conjunto de testes mais complexos(programas-fonte diversos e respectivo código produzido), que tentam testar de uma forma mais alargadas as funcionalidades da Algebra, sendo estes:

- lidos 3 números, escrever o maior deles.
- ler N (valor dado) números e calcular e imprimir o seu somatório.
- contar e imprimir os números pares de uma sequência de N números dados.
- ler e armazenar os elementos de um vetor de comprimento N, imprimido os valores por ordem crescente após fazer a ordenação do array por trocas diretas.
- ler e armazenar os elementos de uma matriz NxM, calculando e imprimindo de seguida a média e máximo dessa matriz.
- invocar e usar num programa uma função.

Capítulo 2

Concepção da Linguagem Algebra

2.1 Concepção/desenho da Resolução

Comecemos por descrever as funcionalidades da linguagem Algebra. Tal como descrito anteriormente a Algebra permitirá:

- 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. Aos arrays de duas dimensões, por se tratar de uma linguagem algébrica, chamaremos matrizes, dada a fácil associação a este tipo de variável à sua definição análoga da álgebra linear.
- 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.

2.1.1 Uma introdução às variáveis

Temos então que as variáveis poderão ser de 3 tipos: inteiros simples, arrays de inteiros, e matrizes de inteiros. Dessa premissa sabemos à partida que o código gerado para a nossa máquina virtual terá que suportar o tipo de variável inteiro. Sabemos ainda que aos tipos de dados mais complexos (arrays e matrizes) apenas é permita a realização de operações de indexação.

Na nossa linguagem de programação por questões de estruturação e percepção, teremos como premissa que as variáveis deverão ser declaradas no início do programa, não podendo haver re-declarações, nem utilizações sem declaração prévia. Não será permitida a declaração e associação de um valor inteiro na mesma instrução(à lá C). Achamos essa solução pouco elegante. Assim, todas as variáveis terão o valor zero após a declaração.

Podemos então aceitar como exemplo as declarações do tipo:

```
1 int a;
2 int auxiliar_1;
3 int array_1d[10];
4 int exemplo_2d[40,2];
```

Dado que toda a porção de código de alto nível julgamos essencial a possibilidade de existência de comentários. Atente no exemplo anterior agora com comentários que facilitam a percepção:

```
// variaveis do tipo inteiro
int a;
int auxiliar_1;
// arrays de inteiros
int array_1d[10];
// matrizes
int exemplo_2d[40,2];
```

Tal como poderá confirmar pela última declaração do exemplo anterior a declaração do tamanho das matrizes é feita da seguinte forma: **nome_variavel[nºlinhas,nºcolunas]**.

A forma de armazenamento e acesso às variáveis será posteriormente discutida nas secções seguintes deste relatório. Neste momento temos especial interesse na especificação da estrutura correcta de programas da nossa linguagem.

Expressões Regulares e acções resultantes

Podemos desde já enumerar as expressões regulares necessárias à produção dos tokens que permitam à GIC o agrupamento dos tokens em produções:

```
%{

%}

letter      [a-zA-Z]
digit       [0-9]
ignore      [\ \t\r\n]

%option yylineno

%%

[\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\_]      { return(yytext[0]); }
int                                             { return (TYPE_INT); }

{letter}({letter}|{digit}|\_)*                  { yylval.var = strdup(yytext); return(id); }
{digit}+                                        { yylval.qt = atoi(yytext); return(num); }
\/\/[^\n]*                                      { printf("%s\n",yytext);  }
{ignore}                                        { ; }

%%

int yywrap(){
  return(1);
}
```

Como é perceptível pela expressão regular correspondente, vulgo {letter}({letter}|{digit}|_)*, as variáveis do tipo inteiro terão sempre de ser iniciadas por uma letra (maiúscula ou minúscula), sendo que como segundo caracter poderão ter um número, letra, ou _.

Produções da GIC

Com os tokens produzidos pelo parser, podemos iniciar a definição da gramática independente de contexto, resultando nas seguintes produções:

```
%{

%}

%union {int qt; char* var;}

%token <var>id
%token <qt>num

%token TYPE_INT
```

```
11
12 %nonassoc PL_THEN
13 %nonassoc PL_ELSE
14
15 %start AlgebricScript
16
17 %%
18
19 AlgebricScript : Declarations
20                ;
21
22 Declarations  :
23               Declarations Declaration ';'
24               | /*empty*/
25               ;
26
27 Declaration  :
28              TYPE_INT id
29              | TYPE_INT id '[' num ',' num ']'
30              | TYPE_INT id '[' num ']'
31              ;
32 %%
33
34 #include "lex.yy.c"
35
36 int yyerror(char* s) {
37     if (strlen(yytext)>1){
38         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
39         fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
40     }
41     else {
42         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
43         fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
44     }
45     return 1;
46 }
47
48 int main () {
49     yyparse();
50     return 0;
51 }
```

2.1.2 Uma introdução às instruções

Da necessidade de realizar operações aritméticas, relacionais e lógicas sobre as variáveis do tipo inteiro atómicas, assim como da necessidade de realizar instruções algorítmicas básicas como a atribuição de expressões a variáveis, surgem as **instruções** na nossa linguagem ***Algebra***.

Consideramos que qualquer que seja a operação a ser realizada, o seu resultado terá que ser sempre atribuído a alguma variável.

Podemos desde já enumerar os tipos de operações permitidas na nossa linguagem, associando também o operador utilizador para representar as mesmas:

- Aritmética
  - Adição : '+'
  - Subtração : '-'
  - Multiplicação inteira : '*'
  - Divisão Inteira : '/'
  - Resto da Divisão Inteira : '
- Relaccional

  - Igualdade : '="="'
  - Diferença : '!"='
  - Superioridade : '>'
  - Superioridade ou Igualdade : '>"='
  - Inferioridade : '<'
  - Inferioridade ou Igualdade : '<"='
- Lógica
  - Negação Lógica : '!'
  - OR Lógico : '|"|'
  - AND Lógico : '&&'

Podemos então aceitar como exemplo de input válido o seguinte código:

```
1  // declaracoes iniciais
2  int a;
3  int b;
4  int c;
5
6  // operacoes de atribuicao
7  a = 7;
8  b = 3;
9
10 // operacoes aritmeticas
11 c = 1 + b*a / 2;
```

Como poderá constatar pelas linhas 7 e 8, e tal como é requerido já será possível realizar operações de atribuição.

Expressões Regulares e acções resultantes

Relativamente às expressões regulares necessárias para proceder correctamente ao parsing não é necessário alterar os ficheiro Flex presente na seção 2.1.1, uma vez que a expressão regular `[\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\_]` já engloba todos os símbolos necessárias até à fase actual.

Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.1, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
1  %{
2
3  %}
4
5  %union {int qt; char* var;}
6
7  %token <var>id
8  %token <qt>num
9
10 %token TYPE_INT
11
12 %start AlgebricScript
13
14 %%
15
16 AlgebricScript : Declarations Instructions
17                ;
18
19 Declarations  :
20                Declarations Declaration ';'
```

```
21                | /*empty*/
22                ;
23
24 Declaration   :
25                TYPE_INT id
26                | TYPE_INT id '[' num ',' num ']'
27                | TYPE_INT id '[' num ']'
28                ;
29
30 Instructions : Instructions Instruction
31                | /*empty*/
32                ;
33
34 Instruction : Assignment ';'
35              ;
36
37 Assignment : id '=' Assignement_Value
38             | Vectors '=' Assignement_Value
39             ;
40
41 Assignement_Value : Arithmetic_Expression
42                    ;
43
44 Vectors : id '[' Arithmetic_Expression Second_Dimension Dimension_End
45         ;
46
47 Second_Dimension : ',' Arithmetic_Expression
48                   | /*empty*/
49                   ;
50
51 Dimension_End : ']'
52                ;
53
54 Arithmetic_Expression : Term
55                        | Arithmetic_Expression '+' Term
56                        | Arithmetic_Expression '-' Term
57                        ;
58
59 Term     : Factor
60          | Term '*' Factor
61          | Term '/' Factor
62          | Term '%' Factor
63          ;
64
65 Factor   : num
66          | id
67          | Vectors
68          | '(' Arithmetic_Expression ')'
69          ;
70
71 Logical_Expressions : Logical_Expressions Logical_Expression
72                      |
73                      ;
74
75 Logical_Expression : '!' Relational_Expression
76                    | Relational_Expression
77                    | Logical_Expression '|''|' Relational_Expression
78                    | Logical_Expression '&''&' Relational_Expression
79                    ;
80
81 Relational_Expression : Arithmetic_Expression
82                        | Arithmetic_Expression '=''=' Arithmetic_Expression
83                        | Arithmetic_Expression '!''=' Arithmetic_Expression
84                        | Arithmetic_Expression '>' Arithmetic_Expression
85                        | Arithmetic_Expression '>''=' Arithmetic_Expression
86                        | Arithmetic_Expression '<' Arithmetic_Expression
87                        | Arithmetic_Expression '<''=' Arithmetic_Expression
88                        | '(' Logical_Expressions ')'
```

```
                        ;
%%

#include "lex.yy.c"

int yyerror(char* s) {
    if (strlen(yytext)>1){
        printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
        fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
    }
    else {
        printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
        fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
    }
    return 1;
}

int main () {
    yyparse();
    return 0;
}
```

As produções relativas às expressões lógicas e relacionais terão especial importância na adição da capacidade de inclusão de instruções para controlo do fluxo de execução – condicional e cíclica – que possam ser aninhadas, na nossa linguagem ***Algebra***, que passaremos de seguida e especificar.

2.1.3 Uma introdução às instruções condicionais e cíclicas

Instruções condicionais

Por forma à ***Algebra*** ter utilidade real, é necessária a inclusão de instruções que permitam mudar o fluxo de execução. Necessitamos portanto de incluir a possibilidade de declarar instruções condicionais na nossa linguagem.

Para criarmos uma estrutura condicional, deveremos recorrer a expressões do tipo:

if (Expressão Lógica)
then [{Instruções}|Instrução]
else [{Instruções}|Instrução|/*empty*/]

Pela análise do esquema anterior sabemos que o bloco de código **else** [{Instruções}|Instrução|/*empty*/] é opcional, sendo que, em caso de os fluxos de execução representarem apenas uma instrução na nossa linguagem ***Algebra*** não existe a necessidade de inclusão de parêntesis entre os diferentes fluxos.

Tal como requerido, deverá ser também possível o aninhamento de instruções condicionais.

Podemos então aceitar como exemplo de input válido o seguinte código:

```
// declaracoes inciais
int a;
int b;
int c;
int maior;

// operacoes de atribuicao
a = 15;
b = 7 * 4;
c = 120 % 1;

// instrucoes condicionais
if ( a >= b && a >= c ) then {
  maior = a;
}
else {
  if ( b > a && b >= c  ) then {
    maior = b;
  }
```

```
20    else {
21      // esta condicao era desnecessaria
22      // mas desta forma provamos o correcto aninhamento de condicionais
23      if (c > a && c > b ) then {
24        maior = c;
25      }
26      // este condicional nao tem o fluxo else
27    }
28  }
```

Instruções cíclicas

Uma instrução cíclica irá permitir ao programador executar um determinado bloco de código um determinado número de vezes, de acordo com uma condição lógica.

Para criarmos uma estrutura cíclica, deveremos recorrer a expressões do tipo:

do [{Instruções}|Instrução]
while (Expressão Lógica)

Pela análise do esquema anterior sabemos que em caso de o fluxo de execução representar apenas uma instrução na nossa linguagem ***Algebra*** não existe a necessidade de inclusão de parêntesis entre as palavras reservadas **do** e **while**.

Tal como requerido, deverá ser também possível o aninhamento de instruções condicionais.

Podemos então aceitar como exemplo de input válido o seguinte código:

```
1  // declaracoes inciais
2  int a;
3  int b;
4  int c;
5  int maior;
6
7  // operacoes de atribuicao
8  a = 1;
9  a = b; // estamos a atribuir directamente a b o valor de a
10 c = 20 % 1;
11
12 // instrucoes ciclicas
13 do {
14   do {
15     a = a + 1;
16   }
17   while ( a < c )
18   b = b + 1;
19 }
20 while ( b < c )
```

Expressões Regulares e acções resultantes

Tomando por base o ficheiro Flex presente na seção 2.1.1, podemos proceder à adição de expressões regulares por forma a produzir os tokens necessários para o correcto reconhecimento pela GIC.

```
1  %{
2
3  %}
4
5  letter      [a-zA-Z]
6  digit       [0-9]
7  ignore      [\ \t\r\n]
8
9  %option yylineno
10
11 %%
```

```
12
13 [\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\-]      { return(yytext[0]); }
14 do                                              { return (PL_DO); }
15 while                                           { return (PL_WHILE); }
16 if                                              { return (PL_IF); }
17 then                                            { return (PL_THEN); }
18 else                                            { return (PL_ELSE); }
19 int                                             { return (TYPE_INT); }
20
21 {letter}({letter}|{digit}|\_)*                  { yylval.var = strdup(yytext); return(id); }
22 {digit}+                                        { yylval.qt = atoi(yytext); return(num); }
23 \/\/[^\n]*                                      { printf("%s\n",yytext);  }
24 {ignore}                                        { ; }
25
26 %%
27
28 int yywrap(){
29   return(1);
30 }
```

Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.1, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
1 %{
2
3 %}
4
5 %union {int qt; char* var;}
6
7 %token <var>id
8 %token <qt>num
9 %token <var>string
10
11 %token TYPE_INT
12
13 %token PL_IF PL_THEN PL_ELSE
14 %token PL_DO PL_WHILE
15
16 %start AlgebricScript
17
18 %%
19
20 AlgebricScript : Declarations Instructions
21                ;
22
23 Declarations  :
24               Declarations Declaration ';'
25               | /*empty*/
26               ;
27
28 Declaration  :
29              TYPE_INT id
30              | TYPE_INT id '[' num  ',' num  ']'
31              | TYPE_INT id '[' num  ']'
32              ;
33
34 Instructions : Instructions Instruction
35              | /*empty*/
36              ;
37
38 Instruction : Assignment ';'
39             | Conditional
40             | Cycle
41             ;
```

```
42
43 Assignment : id '=' Assignement_Value
44            | Vectors '=' Assignement_Value
45            ;
46
47 Assignement_Value : Arithmetic_Expression
48                   ;
49 Vectors : id
50         '['
51         Arithmetic_Expression
52         Second_Dimension Dimension_End
53         ;
54
55 Second_Dimension : ','  Arithmetic_Expression
56                  | /*empty*/
57                  ;
58
59 Dimension_End : ']'
60               ;
61
62 Arithmetic_Expression : Term
63                       | Arithmetic_Expression '+' Term
64                       | Arithmetic_Expression '-' Term
65                       ;
66
67 Term     : Factor
68         | Term '*' Factor
69         | Term '/' Factor
70         | Term '%' Factor
71         ;
72
73 Factor   : num
74         | id
75         | Vectors
76         | '(' Arithmetic_Expression ')'
77         ;
78
79 Logical_Expressions : Logical_Expressions Logical_Expression
80                     |
81                     ;
82
83 Logical_Expression : '!' Relational_Expression
84                    | Relational_Expression
85                    | Logical_Expression '|''|' Relational_Expression
86                    | Logical_Expression '&''&' Relational_Expression
87                    ;
88
89 Relational_Expression : Arithmetic_Expression
90                       | Arithmetic_Expression '=''=' Arithmetic_Expression
91                       | Arithmetic_Expression '!''=' Arithmetic_Expression
92                       | Arithmetic_Expression '>' Arithmetic_Expression
93                       | Arithmetic_Expression '>''=' Arithmetic_Expression
94                       | Arithmetic_Expression '<' Arithmetic_Expression
95                       | Arithmetic_Expression '<''=' Arithmetic_Expression
96                       | '(' Logical_Expressions ')'
97                       ;
98
99 Conditional : If_Starter PL_THEN '{' Instructions '}' Else_Clause
100            | If_Starter PL_THEN  Instruction  Else_Clause
101            ;
102
103 If_Starter : PL_IF '(' Logical_Expressions ')'
104            ;
105
106 Else_Clause : PL_ELSE '{' Instructions '}'
107             | PL_ELSE Instruction
108             | /*empty*/
109             ;
```

```
110
111 Cycle : PL_DO  '{' Instructions '}' PL_WHILE '(' Logical_Expressions ')'
112        | PL_DO  Instruction PL_WHILE '(' Logical_Expressions ')'
113        ;
114
115 %%
116
117 #include "lex.yy.c"
118
119 int yyerror(char* s) {
120     if (strlen(yytext)>1){
121         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
122         fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
123     }
124     else {
125         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
126         fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
127     }
128     return 1;
129 }
130
131 int main () {
132     yyparse();
133     return 0;
134 }
```

Capítulo 3

Conclusão

Um olhar mais atento às expressão regulares definidas até ao momento permitem-nos identificar um outro tipo de token – **string**, até ao momento não apresentado. Denote que na nossa linguagem não é permitido realizar operações sobre strings (como concatenação ou comparação), apenas será permitir escrever as mesmas no standard output.

Nas secções futuras iremos abordar o tipo string de uma forma mais aprofundada.

Relativamente ao estado final do projecto acredito que foram cumpridos todos os requisitos, sendo que o segundo exercício foi sem dúvida o mais desafiante dada a enorme quantidade de dados e o tipo de dados em si a serem analisados. Reconhecer por si só quais as sequências de caracteres válidas foi um desafio.

Naturalmente que a partir da alínea 2.2.b a alínea 2.2.c foi de extrema facilidade, uma vez que todo o trabalho de análise já estava realizado.

Foi ainda tido em conta a possiblidade de recuperar de erros de leitura na alínea 2.2.b o que facilitou o input correct de dados e posterior tratamento. O recurso à biblioteca Glib, recomendada pelo professor José João num aula laboratorial permitiu-me ambientar ainda mais com código desenvolvido por terceiros e sua correcta análise e integração nos meus projectos.

Faço um balanço positivo do trabalho prático, pois, apesar de ser extremamente "time consuming"retirei muito conhecimento no que da análise de dados e processamento de linguagens diz respeito.

Apêndice A

Código do Programa da alínea 1a

Apêndice B

Código do Programa da alínea 2a

Apêndice C

Código do Programa da alínea 2b

Apêndice D

Código do Programa da alínea 3a