Universidade da Beira Interior
Departamento de Informática

Natrix

Elaborado por:

David Pires - 37272
Rafael Marques - 37157

Professor:

Professor Doutor Simão Melo de Sousa

12 de Janeiro de 2020

Conteúdo

Capítulo 1

Introduçaão

1.1 Enquadramento

O presente relatório complementa o trabalho prático desenvolvido no âmbito da Unidade Curricular de Processamento de Linguagens (cód. 11567). Tratará de apresentar a linguagem Natrix, nos seus moldes teóricos, bem como na sua aplicação prática. Adicionalmente, fornecerá documentação apropriada do código desenvolvido.

1.2 Descrição do Problema

O objectivo deste trabalho é implementar de forma incremental uma semântica operacional, uma gramática(LR) com os respectivos lexer e parser, um interpretador e por fim um compilador para a linguagem em questão.

Capítulo 2

Descriçao do Problema

2.1 Linguagem Natrix

A sintaxe da linguagem Natrix assenta em variáveis, constantes, expressões e instruções.
Podemos declarar/atribuir valores a variáveis:

```
\var nome : tipo = valor ;
// Isto é um comentário
// nome corresponde ao nome dado à variável
// tipo corresponde ao tipo da variável
// valor corresponde ao valor dado à variável (necessita ser
   sempre dado um valor inicializada).
```

Definição de tipos:

```
\type nome = expressão ;
// nome é o nome dado ao tipo
// expressão, pode ser: Um array ou um intervalo.
```

Funções pré definidas:

```
\print ( valor ) ;
size ( valor )
// valor corresponde a uma expressão ou tipo
```

Constantes pré definidas:

```
\maxint
minint
// Correspondem ao maior e menor inteiro de 64 bits
```

Arrays

```
\var arr : tipo filled by valor ;
// arr é o nome dado ao array
// tipo corresponde a um tipo array
// valor é o valor com que o array é inicializado
```

If e Else

```
\if (condição) then {  statements }
                else {  statements }
// condição corresponde a uma condição, com as operações
   l gicas: ==, !=, >, >=, <, <=
// statements são os statements a trabalhar
```

Ciclo foreach

```
\foreach nomeVariável in intervalo do { statements }
// nomeVariável corresponde ao nome dado à variável que irá
   percorrer o intervalo
// statements são os statements a trabalhar
```

2.2 Estrutura do Progrmama

Aqui será descrita a função de cada ficheiro no projecto:
ast.mli Ficheiro da árvore de sintaxe abstracta.
lexer.mll Ficheiro do Lexer
parser.mly Ficheiro do Parser
main.ml Ficheiro Main
interp.ml Ficheiro do Interpretador
compiler.ml Ficheiro do Compilador
Makefile Ficheiro Make, usado para compilar o projecto

2.2.1 Árvore de Sintaxe Abstracta (ast.mli)

A sintaxe abstracta desta linguagem caracteriza-se da seguinte forma:

```
\type ident = string

type binary_operation =
```

```
  | Badd | Bsub | Bmul | Bdiv
  | Bequal | Bnotequal | Bbigger | Bbiggerequal | Bsmaller |
    Bsmallerequal
  | Band | Bor

type value_type =
  | Tint

type constant =
  | Cnone
  | Cbool of bool
  | Cstring of string
  | Cint of int64

type expression =
  | Ecst of constant
  | Eident of ident
  | Ebinop of binary_operation * expression * expression
  | Einterval of expression * expression
  | Earray of ident
  | Esize of expression
  | Eget of ident * expression

and statement =
  | Sassign of ident * value_type * expression
  | Sassignarray of ident * ident * expression
  | Sreassign of ident * expression
  | Sreassignarray of ident * expression * expression
  | Sprint of expression
  | Sif of expression * statement
  | Sifelse of expression * statement * statement
  | Sforeach of ident * expression * statement
  | Sblock of statement list
  | Stype of ident * expression
  | Stypearray of ident * expression * value_type

and program = statement
```

2.3 Semântica Operacional

2.3.1 Operações

Adição

$$\frac{E, e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 + e2 \Rightarrow v1 + v2}$$

Subtracção

$$\frac{E, e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 - e2 \Rightarrow v1 - v2}$$

Multiplicação

$$\frac{E, e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 * e2 \Rightarrow v1 * v2}$$

Divisão

$$\frac{E, e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1/e2 \Rightarrow v1/v2}$$

Atribuição

$$\frac{E, e1 \Rightarrow v}{E, x \Leftarrow e \Rightarrow E\{x \Rightarrow v\}}$$

2.4 Tipagem

$\tau \vdash - : int * int \Rightarrow int$
$\tau \vdash * : int * int \Rightarrow int$
$\tau \vdash / : int * int \Rightarrow int$
$\tau \vdash + : int * int \Rightarrow int$

2.5 Implementação

2.5.1 Tipo 'value'

Usamos este tipo para representar as nossas expressões Natrix.

```
\type value =
  | Vnone
  | Vbool of bool
```

```
  | Vint of int64
  | Vstring of string
  | Varray of (value array * int64 * int64)
  | Vinterval of (int64 * int64)
```

Inteiros são de 64 bits, como pedido no enunciado, porem no caso dos Arrays, o seu tamanho terá no máximo um valor de 32 bits. Caso o tamanho do intervalo introduzido seja maior do que 32 bits, poderá originar problemas não tratados. Tipos intervalo e array, serão também tratados como este tipo no interpretador.

2.5.2 Expressões

As expressões retornam sempre um valor do tipo value baseado no pedido feito. Como auxilio a esta função, usamos uma Hash Table, com o nome de ctx onde iremos guardar as variáveis e tipos definidos durante a Interpretação. Esta Hash Table é então do tipo (ident, value). Aqui vamos explicar algumas das expressões mais importantes tratadas no nosso código.

Eident

```
\let rec expression ctx = function
 (*  ...  *)
  | Eident id -> begin
    match id with
      | "maxint" -> (Vint Int64.max_int)
      | "minint" -> (Vint Int64.min_int)
      | _ -> if not (Hashtbl.mem ctx id) then error ("unbound
          variable " ^ id);
      Hashtbl.find ctx id
    end
```

No caso de um ident, o programa irá consultar a HashTable para retornar o valor correspondente a essa variável. No caso particular de a ident corresponder a maxint ou minint, será retornado o valor da constante maxint ou minint, que é o Máximo ou mínimo de um Inteiro de 64 bits. Esta implementação, faz com que na tentativa de repor maxint ou minint, este pedido seja ignorado pelo interpretador. A soma de maxint ou subtraçao de minint, funciona como um ciclo, por exemplo: maxint + 1 = minint.

Einterval

```
\let rec expression ctx = function
 (*  ...  *)
  | Einterval (e1, e2) ->
      let linf, lsup = (expr_int ctx e1), (expr_int ctx e2) in
        if linf < Int64.zero or lsup < Int64.zero then error "
           Intervals need to be positive.";
        if linf > lsup then error "First limit need to be
           superior to the second limit.";
        Vinterval ((expr_int ctx e1), (expr_int ctx e2))
```

No caso de expressão do tipo intervalo, serão verificadas duas condições referidas no enunciado:
- Os intervalos apenas poderão ser positivos.
- O segundo valor precisar ser maior que o primeiro (a .. b; b > a).
Caso estes valores estejam conforme as condições, será retornado o valor do tipo Vinterval (linf, lsup), onde linf corresponde ao limite inferior do intervalo e lsup corresponde ao limite superior do intervalo.

Eget

```
\let rec expression ctx = function
 (*  ...  *)
  | Eget (id, index) -> begin
      if not (Hashtbl.mem ctx id) then error ("unbound variable
         " ^ id);
      match Hashtbl.find ctx id with
        | Varray (arr, linf, lsup) ->
          if (expr_int ctx index) < linf or (expr_int ctx index)
              > lsup
          then error "index out of bounds"
          else arr.( Int64.to_int (Int64.sub (expr_int ctx index
             ) linf) )    (*  !!!  64 BIT INTEGER CONVERSION !!!
               *)
        | _ -> error "array expected"
    end
```

Corresponde a expressões do tipo a[index], onde a corresponde a um Array e index ao index a procurar nesse array. Aqui poderemos obter um dos erros não tratados mencionados acima, ao realizar uma conversão de um inteiro de 64 bits para um inteiro normal.

Esize

```
\let rec expression ctx = function
 (*  ...  *)
  | Esize e -> begin
      match expression ctx e with
        | Vnone -> Vint Int64.zero
        | Vint n -> Vint n
        | Vbool b -> if b then Vint Int64.one else Vint Int64.
          zero
        | Varray (arr, a, b) -> Vint (Int64.add (Int64.sub b a)
          Int64.one)
        | Vinterval (a, b) -> Vint (Int64.add (Int64.sub b a)
          Int64.one)
        | _ -> Vint Int64.zero
    end
```

Corresponde a expressões do tipo size(a), que corresponde à função 'size' presente no enunciado. Decidimos tratar esta função como uma expressão desta forma, devido ao facto de ser a única função presente nesta linguagem.

2.5.3 Statements

Sassign

```
\and statement ctx = function
    | Sassign (id, t, e) ->
        Hashtbl.replace ctx id (expression ctx e)
```

Este statement corresponde à declaração de uma variável, e esta é guardada na HashTable. Se já exixtir uma variável com esse nome na HashTable, essa será substituida pela nova.

Sassignarray

```
\and statement ctx = function
 (*  ...  *)
  | Sassignarray (id, t, e) ->
  begin
    if not (Hashtbl.mem ctx t) then error ("unbound type " ^ t);
    match (Hashtbl.find ctx t) with
      | Varray (arr, linf, lsup) ->
        let narr = (Array.make (Int64.to_int (Int64.sub lsup
          linf) + 1) (expression ctx e)) in    (* !!!  Int64
          CONVERSION  !!! *)
```

```
          Hashtbl.replace ctx id (Varray (narr, linf, lsup))
    | _ -> error "type array expected"
  end
```

Corresponde à declaração de um Array. Arrays são previamente declarados com o type t e aqui apenas é feita uma cópia dessa estrutura e inicializado Array com valores e.

Stypearray

```
\and statement ctx = function
 (*   ...   *)
    | Stypearray (id, indices, vt) ->
      begin
        match expression ctx indices with
          | Vint n ->
            Hashtbl.replace ctx id (Varray ([||], Int64.zero, n)
                )
          | Vinterval (a, b) ->
            Hashtbl.replace ctx id (Varray ([||], a, b))
          | _ -> error "integer or interval expected"
      end
```

Corresponde ao tipo array, onde é indicado o tamanho do Array. Este tamanho, pode ser um intervalo ou um numero inteiro (No caso de um inteiro n, o intervalo será de 0 a n).

Sblock

```
\and statement ctx = function
 (*   ...   *)
    | Sblock sl ->
        block ctx sl

and block ctx = function
    | [] -> ()
    | s :: sl -> statement ctx s; block ctx sl
```

Este programa funciona como blocos, que são na verdade Listas de statements. O próprio programa é um bloco, e também os statements dentro de um if, else ou for.

2.6 Funcionalidades

2.6.1 Funcionalidades do compilador

Impressão no stdout ("print()")
Operações como: Adições, subtrações, multiplicações e divisões;

2.6.2 Funcionalidades do interpretador

Impressão no stdout ("print()")
Operações como: Adições, subtrações, multiplicações e divisões;
Declaraçao de Interios de 64 bits
Arrays
Statements If e Else
Comparações booleanas
Foreach
Intervalos
size()
Tipos intervalo e array

Capítulo 3

Avaliação do Programa

Relativamente à avaliação do programa, devem ser efectuados vários testes, sendo que a maioria desses testes devem ser feitos no ficheiro de teste . Todos eles passam pela elaboração de um programa baseado na linguagem anteriormente descrita. A implementação desses programas pode ser feita da maneira que o programador bem entender, desde que a sequência de acções seja lógica e esteja de acordo com o que o compilador espera.

3.1 Resultados Experimentais

Aqui vamos testar os programas feitos no primeiro exercício e apresentar o resultado.

3.1.1 fatorial.nx

```
\// fatorial de 10


var res : int = 1 ;

foreach i in 2 .. 10 do {

  res := (res * i) ;

}

print(res) ;
```

Resultado:

```
\3628800
```

3.1.2 fib.nx

```
\// print do fibonacci até 10

var n1 : int = 1 ;
var n2 : int = 1 ;
var temp : int = 0 ;

print(n1) ;
print(n2) ;

foreach i in 2..10 do {

  temp := n1 + n2 ;
  n1 := n2 ;
  n2 := temp ;

  print(temp) ;

}
```

Resultado:

```
\1
1
2
3
5
8
13
21
34
55
89
```

3.1.3 multiplicacao.nx

```
\// multiplicação de dois numeros
```

```
var x : int = 5 ;
var y : int = 10 ;

var multi : int = (x * y) ;

print(multi) ;
```

Resultado:

```
\50
```

3.1.4 sum_array.nx

```
\

// soma dois arrays (inicializados de maneira diferente) e faz
    print


var t_size : int = 25 ;


var t_arr = array t_size of int filled by 2 ;


type s = 1 .. t_size ;
type custom_array : array s of int ;


var c_arr : custom_array filled by 0 ;
foreach i in 0 .. t_size do { c_arr[i] := i ; }


foreach i in 0 .. t_size do {

  c_arr[i] := t_arr[i] + c_arr[i] ;
  print(c_arr[i]) ;

}
```

Resultado:

```
\File "sum_array.nx", line 9, characters 11-12:
syntax error
```

Obtivemos syntax error deviado a uma má interpretação do enunciado inicialmente. Necessitamos referir o tipo para declarar um array, logo obtivemos 'syntax error' na linha 9.

3.1.5 e_fatorial.nx

```
\// fatorial de 10

var res : int = 1 ;
foreach i in 2 .. 10 do {


  res = (res * i) ;     // erro de sintaxe (=)

}

print(res) ;
```

Resultado:

```
\File "e_fatorial.nx", line 10, characters 6-7:
syntax error
```

3.1.6 e_fib.nx

```
\// print do fibonacci até 10

var n1 : int = 1 ;
var n2 : int = 1 ;
var temp : int = 0 ;

print(n1) ;
print(n2) ;

foreach i in 2..10 do {

  temp := n1 + n2 ;
  n1 := n23 ;        // variavel não definida n23
  n2 := temp ;

  print(temp) ;
```

```
}
```

Resultado:

```
\1
1
error: unbound variable n23
```

3.1.7 e_print_array.nx

```
\
// print de um array

var i : int = 0 ;
var i_max = 5 ;

type arr = array i of i_max ;

a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5

print(a)

// vários erros
```

Resultado:

```
\File "e_print_array.nx", line 1, characters 1-2:
lexical error: illegal character:
```

3.1.8 e_sum_array.nx

```
\// soma dois arrays (inicializados de maneira diferente) e faz
    print

var t_size : int = -25 ;     // erro, numero negativo
```

```
var t_arr = array t_size of int filled by 2 ;

type s = 1 .. t_size ;
type custom_array : array s of int ;

var c_arr : custom_array filled by 0 ;
foreach i in 0 .. t_size do { c_arr[i] := i ; }

foreach i in 0 .. t_size do {

  c_arr[i] := t_arr[i] + c_arr[i] ;
  print(c_arr[i]) ;

}
```

Resultado:

```
\File "e_sum_array.nx", line 6, characters 20-21:
syntax error
```

3.2 Melhoramentos

Este projeto pode ser melhorado através da implementação de um maior conjunto de funcionalidades tanto no compilador como no interpretador, o tratamento de alguns erros/expressões bem como a declaração de variáveis locais.

3.3 Conclusão

Este projeto partiu da adaptação de um compilador para Natrix usando Assembly x86_64. O resultado final providenciou uma boa perspectiva do processo de compilação e de interpretação de uma linguagem.