Universidade da Beira Interior

Departamento de Informática

Projeto de Processamento de Linguagens

Elaborado por:

Carolina Silva
Cristina Pinto
João Fraga

Orientador:

Professor Doutor Simão Melo de Sousa

12 de Janeiro de 2020

Conteúdo

List of code samples

1 Introdução

1.1 Enquadramento

Este projeto foi desenvolvido no âmbito da unidade curricular de Processamento de Linguagens, com o objetivo de construir um **interpretador** e um **compilador** para a linguagem **Natrix**. A Linguagem Natrix é uma linguagem simples para computação numérica elementar, que trabalha com inteiros (64 bits) e intervalos de inteiros positivos. Tem vetores definidos com base nestes intervalos, dispõe também de uma estrutura condicional e de uma estrutura cíclica. Finalmente esta linguagem Natrix tem funções (passagem por valor).

1.2 Motivação

A realização deste projeto é motivada pelo desejo de aplicar os conhecimentos adquiridos ao longo do semestre na unidade curricular de Processamento de Linguagens e consolidar um pouco mais esses conhecimentos.

1.3 Objetivos

Este projeto tem como objetivo desenvolver um interpretador e compilador para uma linguagem de programação elementar, **Natrix**, implementando de forma incremental:

- Uma gramática (LR)
- Um *lexer*
- Um *parser*
- Uma semântica operacional
- Um interpretador
- Um compilador

2 Implementação e Testes

2.1 Introdução

Neste capítulo serão apresentadas detalhadamente as opções de implementação deste projeto, de forma a facilitar a sua interpretação, bem como alguns screenshots de algumas das principais funcionalidades implementadas.

2.2 Gramática

Nesta secção serão apresentadas decisões tomadas para a linguagem.

2.2.1 Comentários

A linguagem Natrix permite o uso de comentários, tanto *single line*, como *multi line* através do uso dos símbolos "//", "(*"e "*)", respectivamente. Isto é representado pela expressão regular seguinte:

```
comment = "//" _* '\n' | "(*" _* "*)" ;
```

Code sample 1: Comentarios

Os comentários por bloco também ser aninhados o que não é possivel ser representado recorrendo apenas a esta expressão regular

2.2.2 Variaveis

A nomenclatura de uma variavel é representada nas seguintes regras gramaticais:

```
identifier = character { character | number } ;
```

Code sample 2: identifier

A declaração de variaveis é efectuada da seguinte forma:

```
declaration =
  "let" identifier ":" type "=" expression "in" {statement} ";"
| "let" identifier ":" array "filled by" expression "in" statement ";"
| "var" identifier ":" type "=" expression ";"
| "var" identifier ":" array "filled by" expression ";"
| "type" identifier "=" expression ";"
;
```

Code sample 3: declaração

O valor de uma variavel pode ser alterado fazendo uso da regra de atribuição:

```
attribution =
  variable ":=" expression ";"
| variable "[" expression "]" ":=" expression ";"
;
```

Code sample 4: atribuição

O acesso a uma variavel é feito recorrendo ao seu identificador ou, no caso dos arrays, um elemento pode ser acedido através do identificador e o indice correspondente

```
variable = identifier | identifier "[" number "]" ;
```

Code sample 5: variavel

2.2.3 Tipos

Na linguagem Natrix foram definidos varios tipos primitivos (null, int, bool, string) e foi permitida a definição de novos tipos através de combinações de tipos primitivos

```
type = primitives | identifier | range | array | "(" type "*" type ")"
    | "(" type "->" type ")" | "RANGE"  "(" expression ")";
```

Code sample 6: tipos

2.2.4 Expressões

Uma expressão é qualquer frase que represente um valor.

```
op = bitwise | math | comparison ;

operation =
| nop expression
| expression op expression
;

expression =
  "(" expression ")"
| constant
| variable
| operation
| expression "," expression
| identifier "(" {expression} ")" ";"
| "SIZE" "(" range ")" ";"
| "SIZE" "(" expression ")" ";"
;
```

Code sample 7: expressão

2.2.5 *Statement*

Um *statement* é uma frase que representa uma instrução a executar.

Para além dos *statements* já mencionados em variaveis (2.2.2 declarações e atribuições), estão também incluidos nos *statements* a declaração de funções e instruções de controlo de fluxo (foreach, if then else)

```
conditional =
  "if" expression "then" "{" {statement} "}" "else" "{" {statement} "}"
    ";"
| "foreach" identifier "in" type "do" "{" statement "}" ";"
;

function =
  "function" identifier {identifier ":" type} "=" "{" {statement} "}" "
    ;"
| "return" expression ";"
| "print" "(" expression ")" ";"
;

statement =
  comment
| declaration
| attribution
| conditional
| function
;
```

Code sample 8: statement

2.2.6 Programa

Um programa na linguagem Natrix é representado por um conjunto de *statements*

```
program = {statement} ;
```

Code sample 9: programa

2.3 Código

2.3.1 Lexer.mll

O *lexer* é gerado através do uso da ferramenta ocamllex [2], neste são definidas quais as expressões regulares que serão reconhecidas pelo automato, nomeadamente:

```
1  let space = [' ' '\t']+
2
3  let comment = "//" [^ '\n']* '\n'
4
5  let character = ['a'-'z' 'A'-'Z' '_']
6  let number = ['0'-'9']+
7
8  let range = number space? ".." space? number
9
10 let identifier = character (character | number)*
```

Code sample 10: Regular expressions

É também definido de que forma tratar cada uma das expressões regulares referidas anteriormente atraves da regra *token*:

```
rule token = parse
  | '\n' { new_line lexbuf; token lexbuf }
  | (space | comment)+ { token lexbuf }
  | identifier as id { [id_or_kwd id] }
  | ":=" { [ATRIBUTION] }
  | "->" { [ARROW] }
  | '=' { [EQUAL] }
  | ':' { [COLON] }
  | '[' { [LSB] }
  | ']' { [RSB] }
  | '{' { [LCB] }
  | '}' { [RCB] }
  | '(' { [LP] }
  | ')' { [RP] }
  | '!' { [NOT] }
  | '&' { [AND] }
  | '|' { [OR] }
  | '^' { [XOR] }
  | '<' { [LT] }
  | '>' { [GT] }
  | "<=" { [LE] }
  | ">=" { [GE] }
  | "==" { [EQ] }
  | "!=" { [NE] }
  | '+' { [ADD] }
  | '-' { [SUB] }
  | '*' { [MUL] }
  | '/' { [DIV] }
  | '%' { [MOD] }
  | ',' { [COMMA] }
  | ';' { [SEMICOLON] }
  | eof     { [EOF] }
  | range as s {
    Buffer.reset num1_buffer; Buffer.reset num2_buffer; parse_range s
    true;
    [INTERVAL ((int_of_string (Buffer.contents num1_buffer),
    int_of_string (Buffer.contents num2_buffer)))]
  }
  | number as s {
    try [CST (Cint (int_of_string s))]
    with _ -> raise (Lexing_error ("constant too large: " ^ s))
  }
  | "(*"    { level := 1; comment lexbuf; token lexbuf }
  | '"'     { [CST (Cstring (string lexbuf))] }
  | _ as c  { raise (Lexing_error ("illegal character: " ^ String.make
    1 c)) }
```

Code sample 11: Regra token

De forma a obter um processamento mais eficiente no parser de *identifiers*, é feito uso da função *kwd_or_id* que implementa uma *hashtable* de forma a verificar (e devolver) os *tokens* correspondentes as palavras reservadas se estas forem reconhecidas, ou um *token* genérico caso contrário:

```
let id_or_kwd =
  let h = Hashtbl.create 32 in
  List.iter (fun (s, tok) -> Hashtbl.add h s tok) [
    "true", CST (Cbool true); "false", CST (Cbool false);
    "minint", CST (Climit "min"); "maxint", CST (Climit "max");
    "print", PRINT;
    "range", RANGE;
    "size", SIZE;
    "in", IN;
    "let", LET;
    "var", VAR;
    "type", TYPE; "int", INT; "null", NULL; "bool", BOOL; "string",
  STRING;
    "array", ARRAY; "of", OF; "filled", FILLED; "by", BY;
    "if", IF; "then", THEN; "else", ELSE;
    "foreach", FOREACH; "do", DO;
    "function", FUNCTION; "return", RETURN;
  ];
  fun s -> try Hashtbl.find h s with Not_found -> IDENTIFIER s
```

Code sample 12: id_or_kwd

Os *ranges* são processados atrávez do uso da expressão regular *range* e a função *parse_range* que divide a expressão regular anterior em 2 inteiros:

```
let num1_buffer = Buffer.create 20
let num2_buffer = Buffer.create 20

let rec parse_range s flag =
  if String.length s > 0 then
    let c = String.get s 0 in
    let i = int_of_char c in
    if i >= 48 && i <= 57 then
      (if flag then
        Buffer.add_char num1_buffer c
      else
        Buffer.add_char num2_buffer c;
      parse_range (String.sub s 1 ((String.length s) - 1)) flag)
    else parse_range (String.sub s 1 ((String.length s) - 1)) false
```

Code sample 13: parse_range

O processamento de *strings* é feito através da regra *string* que constroi uma string e devolve o token correspondente do seguinte modo:

```
and string = parse
  | '"'
    {
      let s = Buffer.contents string_buffer in
      Buffer.reset string_buffer;
      s
    }
  | "\\t"
    {
      Buffer.add_char string_buffer '\t';
      string lexbuf
    }
  | "\\n"
    {
      Buffer.add_char string_buffer '\n';
      string lexbuf
    }
  | "\\\""
    {
      Buffer.add_char string_buffer '"';
      string lexbuf
    }
  | _ as c
    {
      Buffer.add_char string_buffer c;
      string lexbuf
    }
  | eof { raise (Lexing_error "unterminated string") }
```

Code sample 14: Regra string

O *lexer* também permite o reconhecimento de comentários aninhados através da regra *comment* que faz uso de uma referência (*level*) para manter o nível de profundidade do comentário:

```
and comment = parse
  | "*)"  { level := !level - 1; if !level > 0 then comment lexbuf }
  | "(*"  { level := !level + 1; comment lexbuf }
  | _     { comment lexbuf }
  | eof { raise (Lexing_error "unterminated comment") }
```

Code sample 15: Regra comment

2.3.2 Parser.mly

O *parser* é gerado através do uso da ferramenta menhir [1], neste estão definidos os *tokens* a ser utilizados, nomeadamente:

```
1  /* Token definition */
2
3  %token <Ast.identifier> IDENTIFIER
4  %token <Ast.interval> INTERVAL
5  %token <Ast.constant> CST
6
7  %token PRINT RANGE SIZE
8  %token NULL BOOL INT STRING
9  %token LET IN VAR TYPE ARRAY OF FILLED BY IF THEN ELSE FOREACH DO
       FUNCTION RETURN
10 %token ATRIBUTION ARROW EQUAL COLON LSB RSB LCB RCB LP RP COMMA
       SEMICOLON EOF
11 %token ADD SUB MUL DIV MOD
12 %token NOT AND OR XOR
13 %token GT LT GE LE EQ NE
```

Code sample 16: Tokens

A prioridades e associatividade foram definidas como:

```
1  /* Priority and Associativity definition of tokens */
2
3  %left OR
4  %left AND
5  %left XOR
6  %left ADD SUB
7  %left MUL DIV MOD
8
9  %nonassoc NOT
10 %nonassoc GT LT GE LE EQ NE
11 %nonassoc LSB
12 %nonassoc LCB
13 %nonassoc LP
14 %nonassoc COMMA
15 %nonassoc ARROW
```

Code sample 17: Prioridades e Associatividade

As seguintes definições de regras são baseadas na gramática definida previamente (2.2). Um programa é uma lista de *statements* (ponto de entrada).

```
/* Grammar start */
%start program

/* Return type */
%type <Ast.program> program

%%

program:
  stmt = nonempty_list( statement ) EOF { stmt }
;
```

Code sample 18: Regra program

Os *statements* foram separados nas regras *statement* e *simple_statement*. No *statement* são definidas as regras de construção de uma instrução *if ... then ... else ...*, um ciclo *foreach ... in ... do ...* e declaração de funções. Um *statement* é também um *simple_statement* seguido de um ";".

```
statement:
  s = simple_statement SEMICOLON { s }
| IF e = expression THEN b1 = block ELSE b2 = block { Sif(e, b1, b2) }
| FOREACH id = IDENTIFIER IN t = type_value DO b = block { Sfor(id, t,
    b) }
| FUNCTION id = IDENTIFIER COLON t = type_value p = parameters b =
    block { Sfunc(id, t, p, b) }
| LET id = IDENTIFIER COLON typ = type_value set e = expression IN b =
    block { Slet(id, typ, e, b)}
;
```

Code sample 19: Regra statement

Na declaração de uma função, deve ser indicado o tipo de retorno bem como uma lista com todos os parametros e os tipos respectivos separados por ";". Ex:

```
  function add : int (n1 : int; n2 : int) {
    return n1 + n2;
  }
  print(add(40 ; 2));
```

```
parameters:
  LP  l = separated_list(SEMICOLON, single_parameter) RP { l }
```

Code sample 20: Regra parameters

```
single_parameter:
 id = IDENTIFIER COLON t = type_value { (id,t) }
```

Code sample 21: Regra single_parameter

Um *block* é utilizado na construção do *if* e *for* para o agrupamos de 1 ou mais *statement* dentro de chavetas.

```
block:
  LCB s = nonempty_list(statement) RCB { Sblock(s) }
| s = statement { s }
;
```

Code sample 22: Regra block

Uma *simple_statement* é uma instrução que não é definida à custa de outras instruções.

```
simple_statement:
  VAR id = IDENTIFIER COLON typ = type_value set e = expression { Svar(
    id, typ, e) }
| TYPE id = IDENTIFIER EQUAL t = type_value { Stype(id, t) }
| id = IDENTIFIER ATRIBUTION e = expression { Satr(id, e) }
| id = IDENTIFIER LSB i = expression RSB ATRIBUTION e = expression {
    Sset(id, i, e) }
| RETURN e = expression { Sret(e) }
| PRINT LP e = expression RP { Sprint(e) }
;
```

Code sample 23: Regra simple_statement

```
set:
  EQUAL {}
| FILLED BY {}
;
```

Code sample 24: Regra set

Uma expressão define uma instrução que representa um valor, tais como operações matemáticas, chamadas de funções e acesso a variáveis.

```
expression:
  LP e = expression RP { e }
| c = CST { Econst(c) }
| id = IDENTIFIER { Evar(id) }
| id = IDENTIFIER LSB e = expression RSB { Earr(id, e) }
| op = unop e = expression { OPun(op, e) }
| e1 = expression op = binop e2 = expression { OPbin(e1, op, e2) }
| e1 = expression COMMA e2 = expression { Epair(e1,e2) }
| f = IDENTIFIER LP x = separated_list(SEMICOLON, expression) RP {
    Ecall(f, x) }
| SIZE LP e = expression RP { EAsize(e) }
| SIZE LP i = INTERVAL RP { ERsize(Trange(i)) }
;
```

Code sample 25: Regra expression

Um *array* é um tipo de conjunto de dados presente na linguagem Natrix definido recorrendo à seguinte regra:

```
array:
  ARRAY c = CST OF t = type_value { TAcst(c, t) }
| ARRAY id = IDENTIFIER OF t = type_value { TAvar(id, t) }
| ARRAY i = INTERVAL OF t = type_value { Tarray(i, t) }
;
```

Code sample 26: Regra array

Por fim, um *type_value* representa todas as construções de tipos.

```
type_value:
  p = primitive { p }
| LP t1 = type_value MUL t2 = type_value RP { Tpair(t1, t2)}
| LP t1 = type_value ARROW t2 = type_value RP { Tfun(t1, t2)}
| id = IDENTIFIER { Tvar(id) }
| RANGE LP e = expression RP { Trange_fun(e) }
| i = INTERVAL { Trange(i) }
| a = array { a }
;
```

Code sample 27: Regra type_value

2.3.3 AST.mli

A *Abstract Syntax Tree* é definida atraves dos seguintes tipos:

```
type identifier = string

type interval = int * int

type constant =
  | Cint    of int
  | Cbool   of bool
  | Crange  of int * interval
  | Climit  of string
  | Cstring of string

type unop =
  | OPneg
  | OPnot

type binop =
  | OPand | OPor  | OPxor
  | OPadd | OPsub | OPmul | OPdiv | OPmod
  | OPeq  | OPne  | OPlt  | OPgt  | OPle  | OPge

type expression =
  | Econst  of constant
  | Evar    of identifier
  | Epair   of expression * expression
  | Earr    of identifier * expression
  | Ecall   of identifier * expression list
  | EAsize  of expression
  | ERsize  of typ
  | OPbin   of expression * binop * expression
  | OPun    of unop * expression

and typ =
  | Tnull
  | Tint
  | Tbool
  | Tstring
  | Trange      of interval
  | Tvar        of identifier
  | Tfun        of typ * typ
  | Tpair       of typ * typ
  | TAcst       of constant * typ
  | TAvar       of identifier * typ
  | Tarray      of interval * typ
  | Trange_fun  of expression
```

Code sample 28: AST

```
type statement =
  | Stype   of identifier * typ
  | Slet    of identifier * typ * expression * statement
  | Svar    of identifier * typ * expression
  | Satr    of identifier * expression
  | Sset    of identifier * expression * expression
  | Sfor    of identifier * typ * statement
  | Sif     of expression * statement * statement
  | Sfunc   of identifier * typ * (identifier * typ) list * statement
  | Sprint  of expression
  | Sret    of expression
  | Sblock  of statement list

type program = statement list
```

Code sample 29: AST

Esta arvore representa a estructura de todos os programas possíveis da linguagem Natrix. Uma determinada instanciação desta permite a interpretação e a compilação de frases da linguagem.

2.3.4 Main.ml

O ficheiro main.ml é o ponto de entrada do programa que trata de juntar todas as peças do interpretador e compilador

```
let () =
  let c = open_in file in
  let lexbuf = Lexing.from_channel c in
  try
    let program = Parser.program Lexer.next_token lexbuf in
    close_in c;
    if !parse_only then exit 0;
    if !intrepret then
      Interpret.file program
    else
      Compile.compile_program !ofile program
  with
  | Lexer.Lexing_error s ->
    report (lexeme_start_p lexbuf, lexeme_end_p lexbuf);
    Printf.eprintf "lexical error: %s\n" s;
    exit 1
  | Parser.Error ->
    report (lexeme_start_p lexbuf, lexeme_end_p lexbuf);
    Printf.eprintf "syntax error\n";
    exit 1
  | Interpret.Error s ->
    Printf.eprintf "error: %s\n" s;
    exit 1
  | Compile.Error s ->
    Printf.eprintf "error: %s\n" s;
    exit 1
  | e ->
    Printf.eprintf "Anomaly: %s\n" (Printexc.to_string e);
    exit 2
```

Code sample 30: Main.ml

O ficheiro de entrada é definido da seguinte forma:

```
let file =
  let file = ref None in
  let set_file s =
    if not (Filename.check_suffix s ".nx") then
      raise (Arg.Bad "no .nx extension");
    file := Some s
  in
  Arg.parse option set_file usage;
  match !file with
  | Some f -> f
  | None -> Arg.usage option usage; exit 1
```

Code sample 31: file

Os erros são apresentados fazendo uso da seguinte função:

```
let report (start, finish) =
  let line = start.pos_lnum in
  let first = start.pos_cnum - start.pos_bol + 1 in
  let last = finish.pos_cnum - finish.pos_bol + 1 in
  Printf.eprintf "Error in file: \"%s\", line %d, characters %d-%d:\n"
    file line first last
```

Code sample 32: report

2.3.5 Interpret.ml

O ficheiro interpret.ml é o responsável por interpretar a árvore de síntaxe abstracta e executar código equivalente na linguagem OCaml.

Núcleo implementado

Foi implementado um interpretador capaz de lidar com todas as funcionalidades definidas na gramática estabelecida (2.2)

Detalhes de implementação

O armazenamento de varíaveis foi feito através de uma *hashtable* do tipo (identifier, (value * typ)) que atribui a um identificador o seu valor e tipo.

O armazenamento de varíaveis de tipo foi feito através de um *hashtable* do tipo (identifier, typ) que atribui a um identificador o tipo respectivo.

A declaração de funções foi feita, uma vez mais, através do uso de uma *hashtable* do tipo (identifier, (typ * (identifier * typ) list * statement)) que atribui a um identificador o tipo de retorno da função, uma lista de parametros e os respectivos tipos e o código que esta contém.

Foram implementadas três funções principais para intrepretação do programa:

1. **file** Itera sobre os *statements* do programa
2. **stmt** Recebe um *statement* e realiza a sua execução
3. **expr** Recebe uma expressão e retorna o seu valor

Para toda esta implementação foi necessário a definição do tipo value que define os valores aceites pela linguagem Natrix.

```
type value =
  | Vnull
  | Vint of int
  | Vstring of string
  | Vbool of bool
  | Vpair of value * value
  | Vrange of interval
  | Varray of interval * value array
```

Code sample 33: Value

2.3.6 Compile.ml

O ficheiro compile é o responsável por converter a árvore de síntaxe abstracta do ficheiro lido em código *assembly*

Núcleo implementado

Foi implementado um compilador para uma versão simplificada da linguagem Natrix que implementa a função de Print, a atribuição de valores a variaveis e o acesso às mesmas, implementa também operações aritméticas sobre inteiros e *bitwise operations*.

2.4 Testes

Durante a implementação do intrepretador e compilador, foram criados vários testes (tanto positivos como negativos) de forma a ser possivel verificar que estes se estavam a comportar da forma esperada.

Os testes encontram se todos na pasta *tests* numa de duas subdirectories:

1. **good** (testes positivos)
2. **bad** (testes negativos)

De forma a tornar mais simples a verificação dos testes foi utilizado o script: *run-tests*

2.5 Manual de Utilização

O Interpretador e Compilador foi desenvolvido de forma a correr em qualquer plataforma onde seja possível correr *OCaml*. O compilador gera codigo assembly para processadores *X86_64* e é executável no Sistema Operativo *Linux*.

2.5.1 Dependências

De forma a ser possivel compilar o projecto é necessário ter o programa *Menhir* instalado, para tal, deverá ser executado o commando: *$ opam install menhir*

2.5.2 Compilação

Para proceder a compilação do natrix deverá ser executado na linha de comandos: *$ make*

2.5.3 Modos de utilização

O programa Natrix possui dois modos de utilização:

1. **interpretador** (*$ natrix –interpret filename.nx*)
2. **Compilador** (*$ natrix [-o file] filename.nx*)

O utilizador poderá também passar o comando: *$ ./natrix* para obter o menu de ajuda e visualizar todos os comandos descritos

3 Reflexão Crítica

3.1 Introdução

Neste capítulo será feita uma reflexão critica sobre o desenvolvimento do projeto, elencando os objetivos propostos e objetivos alcançados, principais problemas encontrados e como foram superados.

3.2 Problemas Encontrados

Durante a realização do projecto foram sentidas dificuldades na tomada de decisões e na interpretação da linguagem do enunciado.

A implementação de uma gramática sem conflitos foi um processo trabalhoso e levou ao aumento da complexidade da mesma.

A tipagem foi também um ponto desafiante principalmente devido a introdução do tipo intervalo e de tipos definidos pelos utilizadores.

3.3 Análise Crítica

Após realização deste projeto, considerando a proposta feita inicialmente, consideramos que o núcleo implementado do compilador ficou um pouco reduzido, no entanto consideramos que o interpretador atingiu todas as funcionalidades esperadas e até outras adicionais, tal como funções e strings. Os testes implementados poderiam também ter sido mais exaustivos e implementar programas mais genéricos.

3.4 Conclusão

De um modo geral, os objetivos inicialmente propostos para este projeto foram alcançados. A boa interação e organização entre os vários elementos do grupopermitiu o bom desenvolvimento das atividades definidas para este projeto.

4 Conclusões e Trabalho Futuro

4.1 Conclusão

O desenvolvimento deste projeto ajudou no aprofundamento de competências adquiridas na unidade curricular de Processamento de Linguagens, de acordo com os princípios introduzidos nas aulas teóricas e práticas da unidade curricular em questão.

4.2 Trabalho Futuro

Algumas melhorias que poderão ser consideradas futuramente seriam aumentar o núcleo implementado do compilador da linguagem Natrix, melhorar as mensagens de erro aprensentadas tanto pelo compilador como pelo interpretador, adicionar novas funcionalidades ao interpretador de forma a o tornar mais rico e flexivel.

Referências

[1] Menhir documentation, 2020. "`http://gallium.inria.fr/~fpottier/menhir/manual.html`". Acedido em 10 de Janeiro de 2020.

[2] Ocamllex documentation, 2020. "`https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`". Acedido em 10 de Janeiro de 2020.