Gerador de analisadores sintáticos – Menhir

Alexsandro Santos Soares

Universidade Federal de Uberlândia
Faculdade de Computação

Menhir

- O Menhir é um gerador de analisadores sintáticos LR(1) para o OCaml.
- Para muitas linguagens de programação profissionais existem geradores de analisadores sintáticos:
  - O Yacc (Yet Another Compiler Compiler) foi um dos primeiros, criado na década de 70, escrito em C e gerando código para C, ele trata gramáticas do tipo LALR(1).
  - O Yacc foi reescrito em várias linguagens, gerando códigos para elas: OCaml, ML, Ada, Pascal, Java, Python, Ruby, Go, Common Lisp, etc.
  - Um versão moderna é o Bison da GNU.

Calculadora simples

- Usaremos como exemplo uma calculadora simples contendo operações aritméticas sobre inteiros, parênteses e atribuição.
- Na primeira versão da calculadora estaremos preocupados apenas com a corretude dos aspectos léxicos e sintáticos.
- As versões seguintes serão elaboradas sobre essa inicial e vão acrescentando outras informações.

Analisador léxico

Arquivo `lexer.mll`

```
1  {
2  open Parser
3  }
4  let white = [' ' '\t']+
5  let digit = ['0'-'9']
6  let int = '-'? digit+
7  let letter = ['a'-'z' 'A'-'Z']
8  let id = letter+
9
10 rule read = parse
11   | white { read lexbuf }
12   | "+"   { PLUS }
13   | "("   { LPAREN }
14   | ")"   { RPAREN }
15   | "let" { LET }
16   | "="   { EQUALS }
17   | "in"  { IN }
18   | id    { ID (Lexing.lexeme lexbuf) }
19   | int   { INT (int_of_string (Lexing.lexeme lexbuf)) }
20   | eof   { EOF }
```

Analisador sintático – versão inicial, parte I

Arquivo `parser.mly`

```
1  %{
2  %}
3
4  %token <int> INT
5  %token <string> ID
6  %token PLUS
7  %token LPAREN
8  %token RPAREN
9  %token LET
10 %token EQUALS
11 %token IN
12 %token EOF
13
14 (* PLUS éassociativa àesquerda, IN não éassociativo e, além disso,
15   PLUS tem precedência mais alta que IN, pois PLUS aparece na linha
16   abaixo de IN. *)
17 %nonassoc IN
```

Analisador sintático – versão inicial, parte II

```
18 %left PLUS
19
20 %start <unit> prog
21
22 %%
23
24 prog:
25     | expr EOF { }
26     ;
27
28 expr:
29     | INT { }
30     | ID { }
31     | expr PLUS expr { }
32     | LET ID EQUALS expr IN expr { }
33     | LPAREN expr RPAREN { }
34     ;
```

Programa principal – versão inicial

Arquivo `main.ml`

```
1
2 let parse s =
3   let lexbuf = Lexing.from_string s in
4   let _ = Parser.prog Lexer.read lexbuf in
5   print_endline "Ok"
```

Passos para gerar e compilar

1. Gerando o analisador sintático

```
menhir parser.mly
```

Dois arquivos serão gerados:

`parser.mli` a interface contendo as definições dos tokens e a função principal do analisador sintático.

`parser.ml` o analisador sintático LR(1) propriamente dito.

2. Gerando o analisador léxico

```
ocamllex lexer.mll
```

Um novo arquivo é criado contendo o analisador léxico: `lexer.ml`.

3. Compilando o arquivo de interface do analisador sintático

```
ocamlc -c parser.mli
```

Gera o arquivo `parser.cmi`, a interface compilada do analisador.

Passos para gerar e compilar – continuação

4. Compilando o arquivo do analisador sintático

```
ocamlc -c parser.ml
```

Gera `parser.cmo`, o arquivo objeto do analisador sintático.

5. Compilando o arquivo do analisador léxico

```
ocamlc -c lexer.ml
```

Gera `lexer.cmo`, o arquivo objeto do analisador léxico.

Arquivo de incialização do Ocaml

O arquivo `.ocamlinit` é lido pelo intérprete do Ocaml e executado imediatamente antes de apresentar o prompt de comandos. Assim, ele é o lugar ideal para carregar arquivos e bibliotecas usados com frequência.

Arquivo `.ocamlinit`

```
1 #load "lexer.cmo";;
2 #load "parser.cmo";;
3 #use "main.ml";;
```

Testando os analisadores léxico e sintático

Entre no Ocaml:

```
rlwrap ocaml
```

Depois digite:

```
# parse "22";;
Ok

# parse "11+11";;
Ok

# parse "(10+1)+(5+6)";;
Ok

# parse "let x = 22 in x";;
Ok

# parse "let x = 0 in  let  x = 22 in x";;
Ok
```

Automatizando o processo de compilação

- À medida que o projeto fica maior, com a inclusão de mais arquivos, é mais difícil lembrar qual a sequência correta para a geração e/ou compilação desses arquivos.
- Podemos automatizar esse processo de várias formas, as mais usuais sendo:
  - Escrever um arquivo `Makefile` contendo as dependências entre arquivos com suas respectivas especifidades de compilação;
  - Usar o `OCamlbuild`, uma ferramenta específica para o OCaml que determina a sequência de chamadas ao compilador, juntamente com o conjunto correto de opções de linha de comando, para compilar um projeto.
- Vamos aprender como usar o `OCamlbuild` e como modificar o arquivo `.ocamlinit` para que tudo funcione.

OCamlbuild

Vamos apagar todos os arquivos já gerados e compilados:

```
rm -f *.cmi *.cmo lexer.ml parser.ml parser.mli
```

Agora digite

```
ocamlbuild -use-menhir main.byte
```

- Precisamos informar ao `OCamlbuild` que é para usar o `Menhir` como gerador de analisadores sintáticos pois, do contrário, ele usará o `ocamlyacc`.
- Além disso, é necessário informar qual é o arquivo principal, aqui é o `main`, juntamente com a forma que desejamos compilar o projeto:

  `.byte` para gerar bytecodes para uso no intérprete;
  `.native` para gerar binários nativos.
- O `OCamlbuild` cria um diretório chamado `_build` onde são colocados todos os arquivos gerados e compilados.

Modificando o arquivo .ocamlinit

Para que o Ocaml saiba onde estão os arquivos compilados, precisamos alterar `.ocamlinit` para informar que eles estão no diretório `_build`.

Arquivo `.ocamlinit`

```
1 #directory "_build";;
2 #load "lexer.cmo";;
3 #load "parser.cmo";;
4 #load "main.cmo";;
5 open Main
```

Depois de modificado e salvo, basta entrar no Ocaml e a função `parse` da calculadora estará disponível.

Analisador sintático – versão 1

- Nessa versão acrescentaremos uma árvore sintática abstrata (ASA) à calculadora.
- A árvore será definida em um arquivo separado chamado `ast.ml` e incluído no arquivo do analisador sintático.
- A ASA será construída passo a passo a partir das regras de produção da gramática.
- No arquivo principal serão incluídos alguns testes.

Árvore sintática abstrata

Arquivo `ast.ml`

```
1 type expr =
2   | Var of string
3   | Int of int
4   | Add of expr * expr
5   | Let of string * expr * expr
```

Analisador sintático – versão 1, parte I

Arquivo `parser.mly`

```
1  %{
2  open Ast
3  %}
4
5
6  %token <int> INT
7  %token <string> ID
8  %token PLUS
9  %token LPAREN
10 %token RPAREN
11 %token LET
12 %token EQUALS
13 %token IN
14 %token EOF
15
16 %nonassoc IN
17 %left PLUS
```

Analisador sintático – versão 1, parte II

```
18
19 %start <Ast.expr> prog
20
21 %%
22
23 prog:
24   | e = expr; EOF { e }
25   ;
26
27 expr:
28   | i = INT { Int i }
29   | x = ID { Var x }
30   | e1 = expr; PLUS; e2 = expr { Add(e1,e2) }
31   | LET; x = ID; EQUALS; e1 = expr; IN; e2 = expr { Let(x,e1,e2) }
32   | LPAREN; e = expr; RPAREN {e}
33   ;
```

Programa principal – versão 1

Arquivo `main.ml`

```
1  open Ast
2
3  let parse s =
4    let lexbuf = Lexing.from_string s in
5    let ast = Parser.prog Lexer.read lexbuf in
6    ast
7
8  let testes () =
9    assert ( (Int 22) = parse "22");
10   assert ( (Add (Int 11, Int 11)) = parse "11+11");
11   assert ( (Add (Add (Int 10, Int 1), Add (Int 5, Int 6)))
12         = parse "(10+1)+(5+6)");
13   assert ( (Let ("x", Int 22, Var "x"))
14         = parse "let x = 22 in x");
15   assert ( (Let ("x", Int 0, Let ("x", Int 22, Var "x")))
16         = parse "let x = 0 in let x = 22 in x")
```

Arquivo `.ocamlinit` – versão 1

Arquivo `.ocamlinit`

```
1 #directory "_build";;
2 #load "lexer.cmo";;
3 #load "parser.cmo";;
4 #load "main.cmo";;
5 open Ast
6 open Main
```

Intérprete

- A segunda versão da calculadora acrescenta um intérprete que funciona da seguinte forma:
  1. Lê a expressão digitada.
  2. Analisa sintaticamente a expressão e, se tudo estiver correto, gera uma árvore sintática abstrata.
  3. Avalia a ASA e retorna uma expressão com o resultado da avaliação.
  4. Extrai o resultado da expressão e apresenta ao usuário.
- Avaliar uma expressão envolve reduzir a expressão a uma outra expressão contendo apenas um inteiro.

Função principal do intérprete

```
1 let interpreta e =
2   e |> parse |> multipasso |> extrai_valor
```

O operador `|>` do OCaml representa a aplicação reversa de uma função, ou seja, você pode colocar a função após seu argumento.

A função `parse` é a mesma da versão 1. Ela realiza a análise sintática e retorna a ASA.

A função `multipasso` avalia a ASA e retorna uma expressão com o resultado.

A função `extrai_valor` extrai um inteiro da expressão resultante, quando isso for possível ou gera um exceção, em caso contrário.

```
1 let extrai_valor = function
2   | Int i -> i
3   | _ -> failwith "Nao eh um valor"
```

Função de avaliação

```
1  let rec multipasso = function
2    | Int n -> Int n
3    | e     -> multipasso (passo e)
4
5  (* Um único passo de avaliação. *)
6  let rec passo = function
7    | Int n              -> failwith "Nao deveria ocorrer"
8    | Var _              -> failwith "Variavel nao ligada"
9    | Add(Int n1, Int n2) -> Int (n1+n2)
10   | Add(Int n1, e2)    -> Add(Int n1, passo e2)
11   | Add(e1, e2)        -> Add(passo e1, e2)
12   | Let(x, Int n, e2) -> subst x (Int n) e2
13   | Let(x, e1, e2)     -> Let(x, passo e1, e2)
```

Função de substituição

```
1
2 (* Substitui x por e1 em e2 *)
3 let rec subst x e1 e2 = match e2 with
4   | Var y       -> if x=y then e1 else e2
5   | Int c       -> Int c
6   | Add(esq,dir) -> Add(subst x e1 esq, subst x e1 dir)
7   | Let(y,exp_valor,exp_corpo) ->
8     if x=y
9     then Let(y, subst x e1 exp_valor, exp_corpo)
10    else Let(y, subst x e1 exp_valor, subst x e1 exp_corpo)
```

Nos slides seguintes apresento o código completo do intérprete.

Intérprete – versão 1, parte I

Arquivo `interp.ml`

```
1  open Ast
2
3  (* Substitui x por e1 em e2 *)
4  let rec subst x e1 e2 = match e2 with
5    | Var y          -> if x=y then e1 else e2
6    | Int c          -> Int c
7    | Add(esq,dir) -> Add(subst x e1 esq, subst x e1 dir)
8    | Let(y,exp_valor,exp_corpo) ->
9      if x=y
10     then Let(y, subst x e1 exp_valor, exp_corpo)
11     else Let(y, subst x e1 exp_valor, subst x e1 exp_corpo)
12
13 (* Um único passo de avaliação. *)
14 let rec passo = function
15   | Int n               -> failwith "Nao deveria ocorrer"
16   | Var _               -> failwith "Variavel nao ligada"
17   | Add(Int n1, Int n2) -> Int (n1+n2)
```

Intérprete – versão 1, parte II

```
18    | Add(Int n1, e2)   -> Add(Int n1, passo e2)
19    | Add(e1, e2)       -> Add(passo e1, e2)
20    | Let(x, Int n, e2) -> subst x (Int n) e2
21    | Let(x, e1, e2)    -> Let(x, passo e1, e2)
22
23  (* O fecho transitivo e reflexivo de passo. *)
24  let rec multipasso = function
25    | Int n -> Int n
26    | e     -> multipasso (passo e)
27
28  (* Analisa uma string em uma ASA *)
29  let parse s =
30    let lexbuf = Lexing.from_string s in
31    let ast = Parser.prog Lexer.read lexbuf in
32    ast
33
34
35
```

Intérprete – versão 1, parte III

```
36 (* Extrai o valor de um nó da ASA.
37    Gera uma exceção se o argumento não éuma expressão inteira. *)
38 let extrai_valor = function
39   | Int i -> i
40   | _ -> failwith "Nao eh um valor"
41
42 (* Interprete uma expressão *)
43 let interpreta e =
44   e |> parse |> multipasso |> extrai_valor
45
46 (* Execute alguns testes *)
47 let run_tests () =
48   assert (22 = interpreta "22");
49   assert (22 = interpreta "11+11");
50   assert (22 = interpreta "(10+1)+(5+6)");
51   assert (22 = interpreta "let x = 22 in x");
52   assert (22 = interpreta "let x = 0 in let x = 22 in x")
```

Minilinguagem Tipos

- Agora que já sabemos o básico da utilização do `Menhir`, apresentaremos uma linguagem mais elaborada: a Tipos.
- A gramática para essa linguagem será apresentada nos próximos slides.
- Em seguida, veremos exemplos de programas escritos nela.

Gramática EBNF da minilinguagem Tipos – parte 1

⟨*programa*⟩ ::= `programa` { ⟨*declaração*⟩ } `inicio` { ⟨*comando*⟩ } `fim` `;`

⟨*declaração*⟩ ::= `identificador` { `,` `identificador` } `:` ⟨*tipo*⟩ `;`

⟨*tipo*⟩ ::= ⟨*tipo_simples*⟩ | ⟨*tipo_arranjo*⟩ | ⟨*tipo_registro*⟩

⟨*tipo_simples*⟩ ::= `inteiro` | `cadeia` | `booleano`

⟨*tipo_arranjo*⟩ ::= `arranjo` `[` ⟨*limites*⟩ `]` `de` ⟨*tipo*⟩

⟨*tipo_registro*⟩ ::= `registro`
`identificador` `:` ⟨*tipo*⟩ `;`
{ `identificador` `:` ⟨*tipo*⟩ `;` }
`fim` `registro`

⟨*limites*⟩ ::= `INT` `..` `INT`

⟨*comando*⟩ ::= ⟨*comando_atribuição*⟩
| ⟨*comando_se*⟩
| ⟨*comando_entrada*⟩
| ⟨*comando_saída*⟩

Gramática EBNF da minilinguagem Tipos – parte 2

⟨*comando_atribuição*⟩ ::= ⟨*variavel*⟩ `:=` ⟨*expressão*⟩ `;`

⟨*comando_se*⟩ ::= `se (` ⟨*expressão*⟩ `)`
`então {` ⟨*comando*⟩ `}`
`[ senão {` ⟨*comando*⟩ `} ]`
`fim se ;`

⟨*comando_entrada*⟩ ::= `entrada` ⟨*variável*⟩ `{ ,` ⟨*variável*⟩ `} ;`

⟨*comando_saída*⟩ ::= `saída` ⟨*variável*⟩ `{ ,` ⟨*variável*⟩ `} ;`

⟨*expressão*⟩ ::= `[` ⟨*operando*⟩ ⟨*operador*⟩ `]` ⟨*operando*⟩

⟨*operando*⟩ ::= ⟨*variável*⟩ | `INT` | `STRING` | ⟨*booleano*⟩ | `(` ⟨*expressão*⟩ `)`

⟨*variável*⟩ ::= `identificador` | ⟨*variável*⟩ `.` `identificador`
| ⟨*variável*⟩ `[` ⟨*expressão*⟩ `]`

⟨*booleano*⟩ ::= `verdadeiro` | `falso`

⟨*operador*⟩ ::= `<` | `=` | `!=` | `>`
| `+` | `-` | `*` | `/`
| `^`
| `&&` | `||`

Exemplo 1 de programa em Tipos

Arquivo `ex1.tip`

```
1 programa
2    -- Operações aleatórias para teste com expressões
3    x, y: booleano;
4 inicio
5   x := 1+2 < 3+4 && 5+6*4 > 5;
6   y := 2 = 5 && 4 = 4;
7 fim;
```

Exemplo 2 de programa em Tipos

Arquivo `ex2.tip`

```
1  programa
2     -- Lê e Soma dois números complexos
3     i, j, resultado: registro
4                         parte_real: inteiro;
5                         parte_imag: inteiro;
6                      fim registro;
7  inicio
8     entrada i.parte_real, i.parte_imag;
9     entrada j.parte_real, j.parte_imag;
10    resultado.parte_real := i.parte_real + j.parte_real;
11    resultado.parte_imag := i.parte_imag + j.parte_imag;
12    saida resultado.parte_real, resultado.parte_imag;
13 fim;
```

Analisador léxico para Tipos, parte I

Arquivo `lexico.mll`

```
{
  open Lexing
  open Printf
  open Sintatico

  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
     lexbuf.lex_curr_p <- { pos with
        pos_lnum = pos.pos_lnum + 1;
        pos_bol = pos.pos_cnum;
     }

  let msg_erro lexbuf c =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "%d-%d: caracter desconhecido %c" lin col c
```

Analisador léxico para Tipos, parte II

```
18 }
19
20 let digito = ['0' - '9']
21 let inteiro = '-'? digito+
22
23 let letra = ['a' - 'z' 'A' - 'Z']
24 let identificador = letra ( letra | digito | '_')*
25
26 let brancos = [' ' '\t']+
27 let novalinha = '\r' | '\n' | "\r\n"
28
29 let comentario = "//" [^ '\r' '\n' ]*
30
31 rule token =
32   parse
33   | brancos { token lexbuf }
34   | novalinha { incr_num_linha lexbuf; token lexbuf }
35   | comentario { token lexbuf }
```

Analisador léxico para Tipos, parte III

```
36  | "/*"      { comentario_bloco 0 lexbuf }
37  | '+'   { MAIS }
38  | '-'   { MENOS }
39  | '*'   { MULT }
40  | '/'   { DIV }
41  | '<'   { MENOR }
42  | '='   { IGUAL }
43  | "!="  { DIFER }
44  | '>'   { MAIOR }
45  | "&&"  { ELOG }
46  | "||"  { OULOG }
47  | '^'   { CONCAT }
48  | '('   { APAR }
49  | ')'   { FPAR }
50  | '['   { ACOL }
51  | ']'   { FCOL }
52  | ','   { VIRG }
53  | ".."  { PPTO }
```

Analisador léxico para Tipos, parte IV

```
54   | '.'   { PTO }
55   | ':'   { DPTOS }
56   | ';'   { PTV }
57   | ":="  { ATRIB }
58   | '"'   { let buffer = Buffer.create 1 in
59             let str = leia_string buffer lexbuf in
60                STRING str }
61   | "programa" { PROGRAMA }
62   | "inicio"  { INICIO }
63   | "fim"     { FIM }
64   | "inteiro" { INTEIRO }
65   | "cadeia"  { CADEIA }
66   | "booleano" { BOOLEANO }
67   | "arranjo" { ARRANJO }
68   | "de"      { DE }
69   | "registro" { REGISTRO }
70   | "fim"     { FIM }
71   | "se"      { SE }
```

Analisador léxico para Tipos, parte V

```
72    | "entao"   { ENTAO }
73    | "senao"   { SENAO }
74    | "entrada" { ENTRADA }
75    | "saida"   { SAIDA }
76    | "verdadeiro" { BOOL true }
77    | "falso"     { BOOL false}
78    | identificador as x  { ID x }
79    | inteiro as n { INT (int_of_string n) }
80    | _ as c { failwith (msg_erro lexbuf c) }
81    | eof  { EOF }
82
83  and comentario_bloco n = parse
84     "*/"   { if n=0 then token lexbuf
85              else comentario_bloco (n-1) lexbuf }
86  | "/*"    { comentario_bloco (n+1) lexbuf }
87  | _       { comentario_bloco n lexbuf }
88  | eof     { failwith "Comentário não fechado" }
89
```

Analisador léxico para Tipos, parte VI

```
90 and leia_string buffer = parse
91     '"'       { Buffer.contents buffer}
92   | "\\t"     { Buffer.add_char buffer '\t'; leia_string buffer lexbuf
         }
93   | "\\n"     { Buffer.add_char buffer '\n'; leia_string buffer lexbuf
         }
94   | '\\' '"' { Buffer.add_char buffer '"'; leia_string buffer lexbuf }
95   | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string buffer lexbuf
        }
96   | _ as c    { Buffer.add_char buffer c; leia_string buffer lexbuf }
97   | eof       { failwith "A string não foi fechada"}
```

Árvore sintática abstrata para Tipos, parte I

Arquivo `ast.mll`

```
1  (* The type of the abstract syntax tree (AST). *)
2  type ident = string
3
4  type programa = Programa of declaracoes * comandos
5  and declaracoes = declaracao list
6  and comandos = comando list
7  and declaracao = DecVar of ident * tipo
8
9  and tipo = TipoInt
10          | TipoString
11          | TipoBool
12          | TipoArranjo of tipo * int * int
13          | TipoRegistro of campos
14
15 and campos = campo list
16 and campo = ident * tipo
17
```

Árvore sintática abstrata para Tipos, parte II

```
18 and comando = CmdAtrib of variavel * expressao
19              | CmdSe of expressao * comandos * (comandos option)
20              | CmdEntrada of variaveis
21              | CmdSaida of variaveis
22
23 and variaveis = variavel list
24 and variavel = VarSimples of ident
25               | VarCampo of variavel * ident
26               | VarElemento of variavel * expressao
27
28 and expressao = ExpVar of variavel
29                | ExpInt of int
30                | ExpString of string
31                | ExpBool of bool
32                | ExpOp of oper * expressao * expressao
33
34 and oper = Mais
35          | Menos
```

Árvore sintática abstrata para Tipos, parte III

```
36          | Mult
37          | Div
38          | Menor
39          | Igual
40          | Difer
41          | Maior
42          | E
43          | Ou
44          | Concat
```

Analisador sintático para Tipos, parte I

Arquivo `sintatico.mly`

```
1  %{
2  open Ast
3  %}
4
5  %token <int> INT
6  %token <string> ID
7  %token <string> STRING
8  %token <bool> BOOL
9  %token PROGRAMA
10 %token INICIO
11 %token FIM
12 %token VIRG DPTOS PTO PPTO PTV
13 %token ACOL FCOL
14 %token APAR FPAR
15 %token INTEIRO CADEIA BOOLEANO
16 %token ARRANJO DE
17 %token REGISTRO
```

Analisador sintático para Tipos, parte II

```
18 %token SE ENTAO SENAO
19 %token ENTRADA
20 %token SAIDA
21 %token ATRIB
22 %token MAIS
23 %token MENOS
24 %token MULT
25 %token DIV
26 %token MENOR
27 %token IGUAL
28 %token DIFER
29 %token MAIOR
30 %token ELOG
31 %token OULOG
32 %token CONCAT
33 %token EOF
34
35 %start <Ast.programa> programa
```

Analisador sintático para Tipos, parte III

```
36
37 %%
38
39 programa: PROGRAMA
40           ds = declaracao*
41         INICIO
42           cs = comando*
43         FIM
44         EOF { Programa (List.flatten ds, cs) }
45
46 declaracao: ids = separated_nonempty_list(VIRG, ID) DPTOS t = tipo {
47                  List.map (fun id -> DecVar (id,t)) ids
48         }
49
50 tipo: t=tipo_simples { t }
51     | t=tipo_arranjo { t }
52     | t=tipo_registro { t }
53
```

Analisador sintático para Tipos, parte IV

```
54
55 tipo_simples: INTEIRO { TipoInt  }
56              | CADEIA  { TipoString }
57              | BOOLEANO { TipoBool }
58
59
60 tipo_arranjo: ARRANJO ACOL lim=limites FCOL DE tp=tipo {
61                 let (inicio, fim) = lim in
62                 TipoArranjo (tp, inicio, fim)
63             }
64
65 tipo_registro:
66   REGISTRO
67      campos=nonempty_list(id=ID DPTOS tp=tipo PTV { (id,tp) } )
68   FIM REGISTRO { TipoRegistro campos }
69
70 limites: inicio=INT PPTO fim=INT { (inicio, fim) }
71
```

Analisador sintático para Tipos, parte V

```
72 comando: c=comando_atribuicao { c }
73         | c=comando_se         { c }
74         | c=comando_entrada    { c }
75         | c=comando_saida      { c }
76
77 comando_atribuicao: v=variavel ATRIB e=expressao PTV {
78       CmdAtrib (v,e)
79 }
80
81 comando_se: SE APAR teste=expressao FPAR ENTAO
82               entao=comando+
83               senao=option(SENAO cs=comando+ {cs})
84             FIM SE PTV {
85               CmdSe (teste, entao, senao)
86            }
87
88
89
```

Analisador sintático para Tipos, parte VI

```
90  comando_entrada:
91     ENTRADA xs=separated_nonempty_list(VIRG, variavel) PTV {
92              CmdEntrada xs
93     }
94
95  comando_saida:
96     SAIDA xs=separated_nonempty_list(VIRG, variavel) PTV {
97              CmdSaida xs
98     }
99
100 expressao:
101         | v=variavel { ExpVar v }
102         | i=INT     { ExpInt i   }
103         | s=STRING  { ExpString s }
104         | b=BOOL    { ExpBool b  }
105         | e1=expressao op=oper e2=expressao { ExpOp (op, e1, e2) }
106         | APAR e=expressao FPAR { e }
107
```

Analisador sintático para Tipos, parte VII

```
108 %inline oper:
109         | MAIS { Mais }
110         | MENOS { Menos }
111         | MULT { Mult }
112         | DIV  { Div  }
113         | MENOR { Menor }
114         | IGUAL { Igual }
115         | DIFER { Difer }
116         | MAIOR { Maior }
117         | ELOG { E    }
118         | OULOG { Ou  }
119         | CONCAT  { Concat  }
120
121 variavel:
122         | x=ID      { VarSimples x }
123         | v=variavel PTO x=ID { VarCampo (v,x) }
124         | v=variavel ACOL e=expressao FCOL { VarElemento (v,e) }
```

Teste do analisador sintático

Arquivo `sintaticoTest.ml`

```
1  open Ast
2
3  let parse s =
4    let lexbuf = Lexing.from_string s in
5    let ast = Sintatico.programa Lexico.token lexbuf in
6    ast
7
8  (* Para compilar:
9       ocamlbuild -use-menhir sintaticoTest.byte
10  *)
```

Arquivo de inicialização

Arquivo `.ocamlinit`

```
1 #directory "_build";;
2 #load "sintatico.cmo";;
3 #load "lexico.cmo";;
4 #load "ast.cmo";;
5 #load "sintaticoTest.cmo";;
6 open Ast
7 open SintaticoTest
```

Conflitos no analisador sintático

Ao executarmos o Menhir com a gramática `sintatico.mly`, obteremos a seguinte mensagem

```
menhir sintatico.mly

Warning: you are using the standard library and/or the %
    inline keyword. We
recommend switching on --infer in order to avoid obscure
     type error messages.
Warning: 11 states have shift/reduce conflicts.
Warning: 121 shift/reduce conflicts were arbitrarily
    resolved.
```

Isso nos informa que houve conflitos do tipo desloca/reduz, 121 deles, em 11 estados. Podemos ver em quais estados estes conflitos acontecerem e a razão deles existirem, executando o seguinte:

```
menhir --explain sintatico.mly
```

Arquivo de conflitos

Abaixo está um trecho do arquivo `sintatico.conflicts`

```
** Conflict (shift/reduce) in state 67.
** Tokens involved: OULOG MULT MENOS MENOR MAIS MAIOR IGUAL ELOG DIV DIFER CONC$
** The following explanations concentrate on token OULOG.
** This state is reached from programa after reading:

PROGRAMA list(declaracao) INICIO SE APAR expressao CONCAT expressao

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begi$

programa
PROGRAMA list(declaracao) INICIO list(comando) FIM EOF
                                comando list(comando)
                                comando_se
                                SE APAR expressao FPAR ENTAO nonempty_list(com$
                                        (?)

** In state 67, looking ahead at OULOG, shifting is permitted
** because of the following sub-derivation:

expressao CONCAT expressao
                 expressao . OULOG expressao

** In state 67, looking ahead at OULOG, reducing production
** expressao -> expressao CONCAT expressao
** is permitted because of the following sub-derivation:

expressao OULOG expressao // lookahead token appears
expressao CONCAT expressao .
```

Resolução de conflitos

- Analisando o arquivo de conflitos é possível perceber que o problema está na definição das expressões na gramática.
- A gramática como foi escrita é ambígua permitindo várias árvores sintáticas diferentes para uma mesma expressão.
- Existem duas formas de resolver isto:
  1. Pragmaticamente, sem modificar a gramática, usando técnicas que geradores de analisadores sintáticos, como o Menhir, disponibilizam. Essas técnicas envolvem definir a associatividade e a precedência dos operadores envolvidos nas expressões apenas fazendo declarações na gramática.
  2. Formalmente, podemos reescrever a gramática de forma a torná-la não ambígua para expressões, levando em conta a associatividade e a precedência dos operadores envolvidos nas expressões.

Declaração de precedência e associatividade

- Sem modificar as regras gramaticais, podemos inserir as seguintes declarações logo após às definições dos tokens:

```
36 %left OULOG
37 %left ELOG
38 %left IGUAL DIFER
39 %left MAIOR MENOR
40 %left CONCAT
41 %left MAIS MENOS
42 %left MULT DIV
```

- A ordem de declarações é importante: as declarações que aparecerem primeiro, de cima para baixo, no arquivo têm precedência menor que aquelas que aparecem mais abaixo.
- Por exemplo, o ou lógico, marcado pelo token OULOG, possui precedência menor que os operadores relacionais IGUAL e DIFER. Estes, por sua vez, possuem precedência menor que os operadores MAIOR e MENOR.
- Note que operadores com a mesma precedência são descritos na mesma linha, como é o caso de MAIS e MENOS, etc.
- A listagem completa desta nova versão do analisador sintático é apresentada nos próximos slides.

Analisador sintático para Tipos, versão 2, parte I

Arquivo `sintatico.mly`

```
1  %{
2  open Ast
3  %}
4
5  %token <int> INT
6  %token <string> ID
7  %token <string> STRING
8  %token <bool> BOOL
9  %token PROGRAMA
10 %token INICIO
11 %token FIM
12 %token VIRG DPTOS PTO PPTO PTV
13 %token ACOL FCOL
14 %token APAR FPAR
15 %token INTEIRO CADEIA BOOLEANO
16 %token ARRANJO DE
17 %token REGISTRO
```

Analisador sintático para Tipos, versão 2, parte II

```
18 %token SE ENTAO SENAO
19 %token ENTRADA
20 %token SAIDA
21 %token ATRIB
22 %token MAIS
23 %token MENOS
24 %token MULT
25 %token DIV
26 %token MENOR
27 %token IGUAL
28 %token DIFER
29 %token MAIOR
30 %token ELOG
31 %token OULOG
32 %token CONCAT
33 %token EOF
34
35
```

Analisador sintático para Tipos, versão 2, parte III

```
36 %left OULOG
37 %left ELOG
38 %left IGUAL DIFER
39 %left MAIOR MENOR
40 %left CONCAT
41 %left MAIS MENOS
42 %left MULT DIV
43
44 %start <Ast.programa> programa
45 %%
46
47 programa: PROGRAMA
48             ds = declaracao*
49           INICIO
50             cs = comando*
51           FIM PTV
52           EOF { Programa (List.flatten ds, cs) }
53
```

Analisador sintático para Tipos, versão 2, parte IV

```
54
55 declaracao: ids = separated_nonempty_list(VIRG, ID) DPTOS t = tipo
       PTV {
56                     List.map (fun id -> DecVar (id,t)) ids
57           }
58
59
60 tipo: t=tipo_simples { t }
61     | t=tipo_arranjo { t }
62     | t=tipo_registro { t }
63
64
65 tipo_simples: INTEIRO { TipoInt  }
66             | CADEIA  { TipoString }
67             | BOOLEANO { TipoBool }
68
69
70 tipo_arranjo: ARRANJO ACOL lim=limites FCOL DE tp=tipo {
```

Analisador sintático para Tipos, versão 2, parte V

```
71                let (inicio, fim) = lim in
72                TipoArranjo (tp, inicio, fim)
73            }
74
75 tipo_registro:
76   REGISTRO
77     campos=nonempty_list(id=ID DPTOS tp=tipo PTV { (id,tp) } )
78   FIM REGISTRO { TipoRegistro campos }
79
80 limites: inicio=INT PPTO fim=INT { (inicio, fim) }
81
82 comando: c=comando_atribuicao { c }
83        | c=comando_se        { c }
84        | c=comando_entrada   { c }
85        | c=comando_saida     { c }
86
87 comando_atribuicao: v=variavel ATRIB e=expressao PTV {
88       CmdAtrib (v,e)
```

Analisador sintático para Tipos, versão 2, parte VI

```
89 }
90
91 comando_se: SE APAR teste=expressao FPAR ENTAO
92               entao=comando+
93               senao=option(SENAO cs=comando+ {cs})
94             FIM SE PTV {
95               CmdSe (teste, entao, senao)
96             }
97
98 comando_entrada:
99   ENTRADA xs=separated_nonempty_list(VIRG, variavel) PTV {
100     CmdEntrada xs
101   }
102
103 comando_saida:
104   SAIDA xs=separated_nonempty_list(VIRG, variavel) PTV {
105     CmdSaida xs
106   }
```

Analisador sintático para Tipos, versão 2, parte VII

```
107
108 expressao:
109         | v=variavel { ExpVar v }
110         | i=INT     { ExpInt i   }
111         | s=STRING  { ExpString s }
112         | b=BOOL    { ExpBool b  }
113         | e1=expressao op=oper e2=expressao { ExpOp (op, e1, e2) }
114         | APAR e=expressao FPAR { e }
115
116 %inline oper:
117         | MAIS { Mais }
118         | MENOS { Menos }
119         | MULT { Mult }
120         | DIV  { Div  }
121         | MENOR { Menor }
122         | IGUAL { Igual }
123         | DIFER { Difer }
124         | MAIOR { Maior }
```

Analisador sintático para Tipos, versão 2, parte VIII

```
125         | ELOG { E    }
126         | OULOG { Ou   }
127         | CONCAT  { Concat  }
128
129 variavel:
130         | x=ID      { VarSimples x }
131         | v=variavel PTO x=ID { VarCampo (v,x) }
132         | v=variavel ACOL e=expressao FCOL { VarElemento (v,e) }
```

Modificando a gramática

- Um jeito mais formal de eliminar a ambiguidade é reescrever as regras de produção da gramática que levaram à ambiguidade.
- Abaixo e no próximo slide há um esboço dessa solução para a gramática da Tipos.

```
1  expressão: expressao OULOG exp1 {}
2           | exp1                 {}
3
4  exp1: exp1 ELOG exp2           {}
5      | exp2                     {}
6
7  exp2: exp2 IGUAL exp3          {}
8      | exp2 DIFER exp3          {}
9      | exp3                     {}
10
11 exp3: exp3 MAIOR exp4          {}
12     | exp3 MENOR exp4          {}
13     | exp4                     {}
```

Modificando a gramática

```
14 exp4: exp4 CONCAT exp5        {}
15     | exp5                    {}
16
17 exp5: exp5 MAIS exp6          {}
18     | exp5 MENOS exp6         {}
19     | exp6                    {}
20
21 exp6: exp6 MULT exp7          {}
22     | exp6 DIV exp7           {}
23     | exp7                    {}
24
25 exp7:
26     | variavel                {}
27     | INT                     {}
28     | STRING                  {}
29     | BOOL                    {}
30     | APAR expressao FPAR     {}
```

Recuperação de erros

- Para que o analisador sintático sinalize corretamente os erros, precisamos fazer algumas alterações no analisador léxico.
- A principal delas é que ao invés de falhar quando um erro léxico acontecer, geraremos uma exceção que será tratada por outro módulo.
- Os arquivos com a gramática e com a estrutura da árvore sintática abstrata não serão alterados.
- O módulo principal, `sintaticoTest.ml` será bastante modificado para que possa sinalizar os erros sintáticos, apontando a linha e a coluna onde aconteceram.

Analisador léxico para Tipos, versão 3, parte I

Arquivo `lexico.mll`

```
1  {
2    open Lexing
3    open Printf
4    open Sintatico
5
6    exception Erro of string
7
8    let incr_num_linha lexbuf =
9      let pos = lexbuf.lex_curr_p in
10     lexbuf.lex_curr_p <-
11       { pos with pos_lnum = pos.pos_lnum + 1;
12                  pos_bol = pos.pos_cnum
13       }
14
15 }
16
17 let digito = ['0' - '9']
```

Analisador léxico para Tipos, versão 3, parte II

```
18 let inteiro = '-'? digito+
19
20 let letra = ['a' - 'z' 'A' - 'Z']
21 let identificador = letra ( letra | digito | '_')*
22
23 let brancos = [' ' '\t']+
24 let novalinha = '\r' | '\n' | "\r\n"
25
26 let comentario = "--" [^ '\r' '\n' ]*
27
28 rule token =
29   parse
30   | brancos { token lexbuf }
31   | novalinha { incr_num_linha lexbuf; token lexbuf }
32   | comentario { token lexbuf }
33   | "/*"      { comentario_bloco 0 lexbuf }
34   | '+'   { MAIS }
35   | '-'   { MENOS }
```

Analisador léxico para Tipos, versão 3, parte III

```
36    | '*'   { MULT }
37    | '/'   { DIV }
38    | '<'   { MENOR }
39    | '='   { IGUAL }
40    | "!="  { DIFER }
41    | '>'   { MAIOR }
42    | "&&"  { ELOG }
43    | "||"  { OULOG }
44    | '^'   { CONCAT }
45    | '('   { APAR }
46    | ')'   { FPAR }
47    | '['   { ACOL }
48    | ']'   { FCOL }
49    | ','   { VIRG }
50    | ".."  { PPTO }
51    | '.'   { PTO }
52    | ':'   { DPTOS }
53    | ';'   { PTV }
```

Analisador léxico para Tipos, versão 3, parte IV

```
54   | ":=" { ATRIB }
55   | '"'  { let buffer = Buffer.create 1 in
56            let str = leia_string buffer lexbuf in
57               STRING str }
58   | "programa" { PROGRAMA }
59   | "inicio"  { INICIO }
60   | "fim"     { FIM }
61   | "inteiro" { INTEIRO }
62   | "cadeia"  { CADEIA }
63   | "booleano" { BOOLEANO }
64   | "arranjo" { ARRANJO }
65   | "de"      { DE }
66   | "registro" { REGISTRO }
67   | "fim"     { FIM }
68   | "se"      { SE }
69   | "entao"   { ENTAO }
70   | "senao"   { SENAO }
71   | "entrada" { ENTRADA }
```

Analisador léxico para Tipos, versão 3, parte V

```
72    | "saida"   { SAIDA }
73    | "verdadeiro" { BOOL true }
74    | "falso"     { BOOL false}
75    | identificador as x  { ID x }
76    | inteiro as n { INT (int_of_string n) }
77    | _  { raise (Erro ("Caracter desconhecido: " ^ Lexing.lexeme
          lexbuf)) }
78    | eof  { EOF }
79
80  and comentario_bloco n = parse
81     "*/"  { if n=0 then token lexbuf
82             else comentario_bloco (n-1) lexbuf }
83  | "/*"   { comentario_bloco (n+1) lexbuf }
84  | _      { comentario_bloco n lexbuf }
85  | eof    { raise (Erro "Comentário não terminado") }
86
87  and leia_string buffer = parse
88     '"'      { Buffer.contents buffer}
```

Analisador léxico para Tipos, versão 3, parte VI

```
89 | "\\t"    { Buffer.add_char buffer '\t'; leia_string buffer lexbuf
       }
90 | "\\n"    { Buffer.add_char buffer '\n'; leia_string buffer lexbuf
       }
91 | '\\' '"' { Buffer.add_char buffer '"'; leia_string buffer lexbuf }
92 | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string buffer lexbuf
        }
93 | _ as c   { Buffer.add_char buffer c; leia_string buffer lexbuf }
94 | eof       { raise (Erro "A string não foi terminada") }
```

Arquivo principal, versão 3, parte I

Arquivo `sintaticoTest.ml`

```
1  open Printf
2  open Lexing
3
4  open Ast
5  open ErroSint (* nome do módulo contendo as mensagens de erro *)
6
7  exception Erro_Sintatico of string
8
9  module S = MenhirLib.General (* Streams *)
10 module I = Sintatico.MenhirInterpreter
11
12 let posicao lexbuf =
13     let pos = lexbuf.lex_curr_p in
14     let lin = pos.pos_lnum
15     and col = pos.pos_cnum - pos.pos_bol - 1 in
16     sprintf "linha %d, coluna %d" lin col
17
```

Arquivo principal, versão 3, parte II

```
18 (* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
       checkpoint *)
19
20 let pilha checkpoint =
21   match checkpoint with
22   | I.HandlingError amb -> I.stack amb
23   | _ -> assert false (* Isso não pode acontecer *)
24
25 let estado checkpoint : int =
26   match Lazy.force (pilha checkpoint) with
27   | S.Nil -> (* O parser está no estado inicial *)
28      0
29   | S.Cons (I.Element (s, _, _, _), _) ->
30      I.number s
31
32 let sucesso v = Some v
33
34 let falha lexbuf (checkpoint : Ast.programa I.checkpoint) =
```

Arquivo principal, versão 3, parte III

```
35   let estado_atual = estado checkpoint in
36   let msg = message estado_atual in
37   raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
38                                         (Lexing.lexeme_start lexbuf) msg))
39
40 let loop lexbuf resultado =
41   let fornecedor = I.lexer_lexbuf_to_supplier Lexico.token lexbuf in
42   I.loop_handle sucesso (falha lexbuf) fornecedor resultado
43
44
45 let parse_com_erro lexbuf =
46   try
47     Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.
           lex_curr_p))
48   with
49   | Lexico.Erro msg ->
50      printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
51      None
```

Arquivo principal, versão 3, parte IV

```
52    | Erro_Sintatico msg ->
53       printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
54       None
55
56  let parse s =
57    let lexbuf = Lexing.from_string s in
58    let ast = parse_com_erro lexbuf in
59    ast
60
61  let parse_arq nome =
62    let ic = open_in nome in
63    let lexbuf = Lexing.from_channel ic in
64    let result = parse_com_erro lexbuf in
65    let _ = close_in ic in
66    match result with
67    | Some ast -> ast
68    | None -> failwith "A analise sintatica falhou"
69
```

Arquivo principal, versão 3, parte V

```
70
71 (* Para compilar:
72      menhir -v --list-errors sintatico.mly > sintatico.msg
73      menhir -v sintatico.mly --compile-errors sintatico.msg >
            erroSint.ml
74      ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table"
            -package menhirLib sintaticoTest.byte
75  *)
```

Arquivo de incialização, versão 3

Arquivo `.ocamlinit`

```
1  #use "topfind";;
2  #require "menhirLib";;
3  #directory "_build";;
4  #load "erroSint.cmo";;
5  #load "sintatico.cmo";;
6  #load "lexico.cmo";;
7  #load "ast.cmo";;
8  #load "sintaticoTest.cmo";;
9  open Ast
10 open SintaticoTest
```

Compilação do projeto

Para compilar os arquivos do projeto, primeiro gere:

```
menhir -v --list-errors sintatico.mly > sintatico.msg
```

Depois modifique o arquivo sintatico.msg com as suas mensagens de erro.

Agora basta gerar o arquivo `erroSint.ml` que contém as mensagens de erro:

```
menhir sintatico.mly --compile-errors sintatico.msg >
    erroSint.ml
```

Para usar o `ocamlbuild` para compilar todo o projeto, digite

```
ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --
    table" -package menhirLib sintaticoTest.byte
```