Construção de um compilador de MiniPython para Parrot VM usando Objective Caml

Angelo Travizan Neto
travizanneto@gmail.com

Faculdade de Computação
Universidade Federal de Uberlândia

14 de dezembro de 2016

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este trabalho aborda o processo de instalação do OCaml e da plataforma Parrot VM, além da tradução dos códigos sugeridos pelo professor Alexsandro Santos Soares para a linguagem PASM (Parrot assembly), objetivando a familiarização com a mesma. As traduções são explicadas passo a passo.

1.1 Parrot Virtual Machine

Parrot é uma máquina virtual que tem o objetivo de executar programas escritos em linguagens dinâmicas de maneira eficiente. Ela possui dois níveis de assembly: PIR (Parrot Intermediate Representation) e PASM (Parrot Assembly Language). O PIR é uma linguagem de mais alto nível, sendo compilada para PASM. O próximo passo é gerar o bytecode do Parrot, o PBC, através do assembler da Parrot VM. Baseada em registradores, sua linguagem assembly é de mais baixo nível do que a linguagem PIR.

1.2 OCaml

Objective Caml, ou OCaml(Objective Categorical Abstract Machine Language) é uma linguagem de programação funcional. Ela possui adição de técnicas utilizadas para a orientação à objetos, além de ser uma linguagem forte e estática. Nesse trabalho, a linguagem OCaml será utilizada para a implementação do compilador da linguagem MiniPython para a Parrot VM.

1.3 Considerações iniciais

Foi utilizado o compilador Rakudo para compilar de Perl6 para PIR. Como o objetivo é a linguagem PASM, e o parrot não traduz de PIR para PASM, os códigos foram escritos sem a utilização de compiladores.

Capítulo 2

Instalações

2.1 Instalação da Parrot Virtual Machine

A instalação da Parrot VM se resume na instalação dos seguintes pacotes, digitando os seguintes comandos no terminal:

```
> sudo apt-get install parrot
> sudo apt-get install libparrot
> sudo apt-get install libparrot-dev
> sudo apt-get install parrot-doc
```

2.2 Instalação do OCaml

Para instalar o OCaml basta digitar o seguinte comando no terminal:

```
> sudo apt-get install OCaml
```

Capítulo 3

Códigos e tradução

3.1 Códigos de programas e explicação

3.1.1 Python

Segue abaixo os códigos dos programas traduzidos para Python que foram inicialmente sugeridos pelo professor Alexsandro Santos Soares em Pseudo-código.

Listagem 3.1: Módulo mínimo que caracteriza um programa

```
1 def nano01():
2   pass
```

Listagem 3.2: Declaração de uma variável

```
1 def nano02():
2   n = int(n)
```

Listagem 3.3: Atribuição de um inteiro à uma variável

```
1 def nano03():
2   n = int(n)
3   n=1
```

Listagem 3.4: Atribuição de uma soma de inteiros à uma variável

```
1 def nano04():
2   n = int(n)
3   n = 1 + 2
```

Listagem 3.5: Inclusão do comando de impressão

```
1 def nano05():
2   n = 2
3   print(n,end="")
4
5
6 nano05()
```

Listagem 3.6: Atribuição de uma subtração de inteiros à uma variável

```
def nano06():
  n = 1 - 2
  print(n,end="")


nano06()
```

Listagem 3.7: Inclusão do comando condicional

```
def nano07():
  n=1
  if n ==1:
    print(n,end="")


nano07()
```

Listagem 3.8: Inclusão do comando condicional com parte senão

```
def nano08():
  n=1
  if n ==1:
    print(n,end="")
  else:
    print(0,end="")


nano08()
```

Listagem 3.9: Atribuição de duas operações aritmáticas sobre inteiros à uma variável

```
def nano9():
  n=1
  if n ==1:
    print(n,end="")
  else:
    print(0,end="")


nano9()
```

Listagem 3.10: Atribuição de duas variáveis inteiras

```
def nano10():
  n=1
  m=2
  if n ==m:
    print(n,end="")
  else:
    print(0,end="")


nano10()
```

Listagem 3.11: Introdução do comando de repetição enquanto

```
def nano11():
  n=1
  m=2
  x=5
  while x >n:
    n = n + m
    print(n,end="")


nano11()
```

Listagem 3.12: Comando condicional aninhado em um comando de repetição

```
def nano12():
  n=1
  m=2
  x=5
  while x >n:
    if n ==m:
      print(n,end="")
    else:
      print(0,end="")
    x = x -1

nano12()
```

Listagem 3.13: Converte graus Celsius para Fahrenheit

```
def micro01():
  cel , far = 0.0 , 0.0
  print("   Tabela de conversao: Celsius -> Fahrenheit")
  print("Digite a temperatura em Celsius: ",end="")
  cel = input()
  far = (9*cel+160)/5
  print("A nova temperatura é:"+str(far)+"F")

micro01()
```

Listagem 3.14: Ler dois inteiros e decide qual é maior

```
def micro02():
  num1, num2 = 0 , 0
  print("Digite o primeiro numero: ")
  num1 = int(input())
  print("Digite o segundo numero: ")
  num2 = int(input())

  if num1 > num2:
    print("O primeiro numero "+str(num1)+" e maior que o segundo "+str(
        num2),end="")
  else:
    print("O segundo numero "+str(num2)+" e maior que o primeiro "+str(
        num1),end="")


micro02()
```

Listagem 3.15: Lê um número e verifica se ele está entre 100 e 200

```
def micro03():
  numero =0
  print("Digite um numero: ",end="")
  numero = int(input())
  if numero>= 100:
    if numero<= 200:
      print("O numero esta no intervalo entre 100 e 200")
    else:
      print("O numero nao esta no intervalo entre 100 e 200")
  else:
    print("O numero nao esta no intervalo entre 100 e 200")

micro03()
```

Listagem 3.16: Lê números e informa quais estão entre 10 e 150

```
def micro04():
  x,num,intervalo = 0,0,0

  for x in range(5):
    print("Digite o numero: ",end="")
    num = int(input())
    if num >=10:
      if num <=150:
        intervalo = intervalo +1

  print("Ao total, foram digitados "+str(intervalo)+" numeros no intervalo
      entre 10 e 150")

micro04()
```

Listagem 3.17: Lê strings e caracteres

```
def micro05():
  x,h,h = 0,0,0
  nome,sexo = "",""

  for x in range(5):
    print("Digite o nome: ",end="")
    nome = input()
    print("H - Homem ou M - Mulher",end="")
    sexo = input()
    if sexo == "H":
      h = h+1
    elif sexo == "M":
      m = m+1
    else:
      print("Sexo só pode ser H ou M!")

  print("Foram inseridos "+h+" Homens")
  print("Foram inseridas "+m+" Mulheres")


micro05()
```

Listagem 3.18: Escreve um número lido por extenso

```
def micro06():
  numero = 0

  print("Digite um numero de 1 a 5: ",end="")
  numero = int(input())
  if numero ==1:
    print("Um")
  elif numero == 2:
    print("Dois")
  elif numero ==3:
    print("Tres")
  elif numero ==4:
    print("Quatro")
  elif numero ==5:
    print("Cinco")
  else:
    print("Numero Invalido!!!")
micro06()
```

Listagem 3.19: Decide se os números são positivos, zeros ou negativos

```
def micro07():
  numero ,programa= 0,1
  opc = ""

  while programa ==1:
    print("Digite um número: ",end="")
    numero = int(input())

    if numero>0:
      print("Positivo")
    else:
      if numero==0:
        print("O numero e igual a 0")
      if numero <0:
        print("Negativo")

    print("Deseja Finalizar? (S/N) ",end="")
    opc = input()
    if opc == "S":
      programa = 0

micro07()
```

Listagem 3.20: Decide se um número é maior ou menor que 10

```
def micro08():
  numero =1
  while numero < 0 or numero >0:
    print("Digite o numero",end="")
    numero = int(input())
    if numero > 10:
      print("O numero "+str(numero)+" e maior que 10")
    else:
      print("O numero "+str(numero)+" e menor que 10")


micro08()
```

Listagem 3.21: Cálculo de preços

```
def micro09():
  preco, venda, novopreco = 0.0,0.0,0.0

  print("Digite o preco: ",end="")
  preco = int(input())
  print("Digite a venda: ",end="")
  venda = int(input())
  if venda < 500 or preco <30:
    novopreco = preco + 10/100 *preco
  elif (venda >= 500 and venda <1200) or (preco >= 30 and preco <80):
    novopreco = preco + 15/100 * preco
  elif venda >=1200 or preco >=80:
    novopreco = preco - 20/100 * preco


  print("O novo preco e: "+str(novopreco))

micro09()
```

Listagem 3.22: Calcula o fatorial de um número

```
def micro10():
  numero =0
  fat = 0
  print("Digite um numero: ",end="")
  numero = int(input())
  fat = fatorial(numero)

  print("O fatorial de "+str(numero)+" e "+str(fat),end="")



def fatorial(n):
  if n <=0:
    return 1
  else:
    return (n * fatorial(n-1))

micro10()
```

Listagem 3.23: Decide se um número é positivo, zero ou negativo com auxílio de uma função

```
def micro11():
  numero,x =0,0
  print("Digite um numero: ",end="")
  numero = int(input())
  x = verifica(numero)
  if x ==1:
    print("Numero Positivo")
  elif x ==0:
    print("Zero")
  else:
    print("Negativo")

def verifica(n):
  res = 0
```

```
15   if n>0:
16     res = 1
17   elif n<0:
18     res = -1
19   else:
20     res = 0
21
22   return res
23
24 micro11()
```

3.1.2 Código em PASM

Segue abaixo os códigos em PASM referentes aos anteriores, juntamente com uma breve explicação de como os comandos em Python foram traduzidos. Como os programas são complementares em nível de código, as explicações serão feitas somente às instruções que forem novas nos programas. Os códigos estão com identação, com a finalidade do melhor entendimento.

1. -

   Listagem 3.24: Módulo mínimo que caracteriza um programa

   ```
   1 end
   ```

   Inicialização básica de um programa em PASM.

2. -

   Listagem 3.25: Declaração de uma variável

   ```
   1 end
   ```

   A declaração de variáveis não é necessária para o PASM, já que ele é orientado à registradores, em que os de letras I, N, P, S são para inteiros, reais, PObjects (Objetos do Parrot), e strings, respectivamente.

3. -

   Listagem 3.26: Atribuição de um inteiro à uma variável

   ```
   1 set I1, 1
   2 end
   ```

   O comando set é utilizado para atribuição.

4. -

   Listagem 3.27: Atribuição de uma soma de inteiros à uma variável

   ```
   1 add I1, 1,2
   2 say I1
   3 end
   ```

O comando add é utilizado para soma, juntamente com a atribuição ao registrador I1.

5. -

Listagem 3.28: Inclusão do comando de impressão

```
1 set I1, 2
2 print I1
3 end
```

O comando 'print' é simples, em que exibe na tela o que está como argumento. Pode ser um registrador ou uma string. Existe também o comando 'say', que exibe, e já pula uma linha.

6. -

Listagem 3.29: Atribuição de uma subtração de inteiros à uma variável

```
1 sub I1, 1,2
2 print I1
3 end
```

O comando sub é semelhante ao add, mas efetuando uma subtração.

7. -

Listagem 3.30: Inclusão do comando condicional

```
1 set I1, 1
2
3 eq I1, 1, TRUE
4 branch FALSE
5
6 TRUE:
7 print I1
8 branch END
9
10 FALSE:
11
12 END:
13 end
```

O comando eq é utilizado para que ocorra um desvio, se os dois primeiros argumentos forem iguais. No caso, desviará para o que está no terceiro argumento. O opcode branch é utilizada para ocorrer um desvio imediatamente, para o que estiver como argumento, o qual será nomeado posteriormente, seguido de ':'.

8. -

Listagem 3.31: Inclusão do comando condicional com parte senão

```
1 set I1, 1
2
3 eq I1, 1, TRUE
4 branch FALSE
5
6 TRUE:
7 print I1
8 branch END
```

```
9
10 FALSE:
11 print "0"
12
13 END:
14 end
```

Foi utilizado o mesmo comando eq, mas com um artifício para que seja implementada a parte senão, nomeando uma outra parte do codigo como "False"para que ocorra o branch.

9. -

Listagem 3.32: Atribuição de duas operações aritmáticas sobre inteiros à uma variável

```
1 div I2, 1, 2
2 add I1, 1, I2
3
4 eq I1, 1, TRUE
5 branch FALSE
6
7 TRUE:
8 print I1
9 branch END
10
11 FALSE:
12 print "0"
13
14 END:
15 end
```

São utilizados os mesmos comandos citados anteriormente.

10. -

Listagem 3.33: Atribuição de duas variáveis inteiras

```
1 set I1, 1
2 set I2, 2
3 eq I1, I2, TRUE
4 branch FALSE
5
6 TRUE:
7 print I1
8 branch END
9
10 FALSE:
11 print 0
12 branch END
13
14 END:
15 end
```

Comandos utilizados citados anteriormente.

11. -

Listagem 3.34: Introdução do comando de repetição enquanto

```
set I1, 1
set I2, 2
set I3, 5


LOOP:
le I3, I1, END
add I1, I1, I2
say I1
branch LOOP


END:
end
```

O comando enquanto é escrito como um LOOP, em que a cada iteração, é feita uma comparação com o que estiver em seu argumento. Ou seja, a cada iteração é feito um 'if', para ocorrer o branch de sair do loop ou não. Neste caso, o comando 'le', que significa 'lesser or equal', é utilizado para ocorrer um salto para 'END', se I3 for menor ou igual à I1.

12. -

Listagem 3.35: Comando condicional aninhado em um comando de repetição

```
set I1, 1
set I2, 2
set I3, 5


LOOP:
le I3, I1, END
  ne I1, I2, FALSE
    say I1
    branch ENDIF
  FALSE:
    say 0
  ENDIF:  dec I3
branch LOOP


END:
end
```

Todos os opcodes utilizados neste programa já foram explicados anteriormente.

13. -

Listagem 3.36: Converte graus Celsius para Fahrenheit

```
.loadlib 'io_ops'

say " Tabela de conversao: Celsius -> Fahrenheit"
print "Digite a temperatura em Celsius: "
read S1, 4
set I1, S1
say I1
mul I1, I1, 9
say I1
```

```
10 add I1, I1, 160
11 say I1
12 div N1, I1, 5
13 say N1
14
15 print "A nova temperatura e:"
16 print N1
17 print " F"
18
19 END:
20 end
```

Foi utilizado o comando ".loadlib 'io ops'"para importar a biblioteca de IO extendida do Parrot. Dela, foi utilizado o opcode read, para ler a temperatura, e o set logo em seguida para transformar o que foi lido em um inteiro (já que o comando read lê uma string).

14. -

Listagem 3.37: Ler dois inteiros e decide qual é maior

```
1 .loadlib 'io_ops'
2
3
4 print "Digite o primeiro numero: "
5 read S1, 5
6 set I1, S1
7 print "Digite o segundo numero: "
8 read S2, 5
9 set I2, S2
10
11 le I1, I2, FALSE
12   print "O primeiro numero "
13   print I1
14   print " e maior que o segundo "
15   print I2
16   branch END
17 FALSE:
18   print "O segundo numero "
19   print I2
20   print " e maior que o primeiro "
21   print I1
22
23
24 END:
25 end
```

Todos os comandos já foram citados anteriormente.

15. -

Listagem 3.38: Lê um número e verifica se ele está entre 100 e 200

```
1 .loadlib 'io_ops'
2
3
4 print "Digite um numero: "
5 read S1, 5
6 set I1, S1
```

```
7
8 lt I1, 100, FALSE1
9   gt I1, 200, FALSE2
10    say "O numero esta no intervalo entre 100 e 200"
11    branch ENDIF2
12  FALSE2:
13    say "O numero nao esta no intervalo entre 100 e 200"
14  ENDIF2:
15  branch END
16 FALSE1:
17  say "O numero nao esta no intervalo entre 100 e 200"
18
19 END:
20 end
```

Neste programa, foi utilizado o opcode 'lt', que produz um salto para 'FALSE1' se I1 for menor que 100, como o próprio nome já diz: lesser than. E também, o comando 'gt', que produz um salto para 'FALSE2' se I1 for maior do que 200 (greater than).

16. -

Listagem 3.39: Lê números e informa quais estão entre 10 e 150

```
1 .loadlib 'io_ops'
2
3 LOOP:
4 eq I1, 5, ENDLOOP
5   print "Digite um numero: "
6   read S2, 5
7   set I2, S2
8
9   lt I2, 10, INCREMENTO
10    gt I2, 150, INCREMENTO
11      inc I3
12
13  INCREMENTO:
14  inc I1
15  branch LOOP
16 ENDLOOP:
17
18 print "Ao total, foram digitados "
19 print I3
20 say " numeros no intervalo entre 10 e 150"
21
22 END:
23 end
```

O comando 'para' foi transcrito utilizando a mesma técnica do comando 'enquanto', pois como são dois loops, os assemblys são praticamente iguais. Foi utilizado o comando inc, para apenas incrementar o conteúdo do registrador I1 em uma unidade, que é o contador utilizado como parada deste loop.

17. -

Listagem 3.40: Lê strings e caracteres

```
1 .loadlib 'io_ops'
2
3 LOOP:
```

```
4 eq I1, 5, ENDLOOP
5   print "Digite o nome: "
6   read S2, 10
7   print "H - Homem ou M - Mulher: "
8   read S3, 2
9
10  ne S3, "H\n", FESC1
11    inc I2
12    branch INCREMENTO
13  FESC1:
14  ne S3, "M\n", FESC2
15    inc I3
16    branch INCREMENTO
17  FESC2:
18    say "Sexo so poder ser H ou M!"
19
20  INCREMENTO:
21  inc I1
22  branch LOOP
23 ENDLOOP:
24
25 print "Foram inseridos "
26 print I2
27 say " Homens"
28
29 print "Foram inseridos "
30 print I3
31 say " Mulheres"
32
33 END:
34 end
```

Foi utilizado o mesmo comando 'read' da biblioteca 'io ops' utilizado anteriormente. Seu primeiro parâmetro é onde será encaminhado o que for lido do teclado, e o segundo é o tamanho da string.

18. -

Listagem 3.41: Escreve um número lido por extenso

```
1 .loadlib 'io_ops'
2
3
4 print "Digite um numero de 1 a 5: "
5 read S1, 2
6 set I1, S1
7
8 ne I1, 1, FESC1
9   say "Um"
10  branch FESC
11 FESC1:
12 ne I1, 2, FESC2
13   say "Dois"
14   branch FESC
15 FESC2:
16 ne I1, 3, FESC3
17   say "Tres"
18   branch FESC
19 FESC3:
```

```
20 ne I1, 4, FESC4
21   say "Quatro"
22   branch FESC
23 FESC4:
24 ne I1, 5, FESC5
25   say "Cinco"
26   branch FESC
27 FESC5:
28   say "Numero Invalido!!!"
29 FESC:
30
31 END:
32 end
```

O comando 'switch' foi traduzido para o Python através de várias condições para saltos, assim como os comandos 'se'. Então, o assembly foi criado da mesma forma.

19. -

Listagem 3.42: Decide se os números são positivos, zeros ou negativos

```
1  .loadlib 'io_ops'
2
3
4  set I1, 1
5
6  LOOP:
7  ne I1, 1, FIMLOOP
8    print "Digite um numero: "
9    read S2, 5
10   set I2, S2
11   le I2, 0, FALSE1
12     say "Positivo"
13   FALSE1:
14     ne I2, 0, FIMSE2
15       say "O numero e igual a 0"
16       branch FIMSE2
17
18     FIMSE2:
19     ge I2, 0, FIMSE1
20       say "Negativo"
21   FIMSE1:
22
23   print "Deseja finalizar? (S/N) "
24   read S3, 2
25
26   ne S3, "S\n", LOOP
27     set I1, 0
28
29   branch LOOP
30 FIMLOOP:
31
32 END:
33 end
```

O comando 'ne' é utilizado para ocorrer um salto quando o primeiro argumento for diferente do segundo. Todos os outros comandos já foram explicados anteriormente.

20. -

Listagem 3.43: Decide se um número é maior ou menor que 10

```
.loadlib 'io_ops'


set I2, 1

LOOP:
eq I2, 0, FIMLOOP
  print "Digite um numero: "
  read S2, 5
  set I2, S2

  le I2, 10, FALSE1
    print "O numero "
    print I2
    say " e maior que 10"
    branch FIMSE1
  FALSE1:
    print "O numero "
    print I2
    say " e menor que 10"
  FIMSE1:
  branch LOOP
FIMLOOP:

END:
end
```

O comando 'le' é outro utilizado para salto, mas quando o primeiro argumento for menor ou igual ao segundo. Os outros comando já foram citados anteriormente.

21. -

Listagem 3.44: Cálculo de preços

```
.loadlib 'io_ops'


print "Digite o preco: "
read S1, 5
set N1, S1
print "Digite a venda: "
read S2, 5
set N2, S2

lt N2, 500, TRUE1
ge N1, 30, FALSE1
TRUE1:
  mul N3, 10, N1
  div N3, N3, 100
  add N3, N3, N1
  branch FIMSE1
FALSE1:
  lt N2, 500, TEST2
  lt N2, 1200, TRUE2
  TEST2:
  lt N1, 30, FALSE2
  ge N1, 80, FALSE2

```

```
  TRUE2:
    mul N3, 15, N1
    div N3, N3, 100
    add N3, N3, N1
    branch FIMSE1
  FALSE2:
    ge N2, 1200, TRUE3
    lt N1, 80, FIMSE1
    TRUE3:
      mul N3, 20, N1
      div N3, N3, 100
      sub N3, N1, N3
FIMSE1:

print "O novo preco e "
say N3

END:
end
```

Os 'se's que possuem 'E', são traduzidos com um salto para 'FALSE' diretamente, se o primeiro valor booleano for falso. Já os que possuem 'OU', saltam apenas se os dois valores booleanos forem falsos. Então, ocorreu uma manipulação dentre os opcodes do PASM assembly para que o programa tenha a função exigida no Python.

22. -

Listagem 3.45: Calcula o fatorial de um número

```
.loadlib 'io_ops'


print "Digite um numero: "
read S1, 5
set I1, S1

set I2, I1
set I4, I1
branch FATORIAL

CONTINUE:
print "O fatorial de "
print I1
print " e "
say I2
branch END


FATORIAL:
  gt I4, 1, FALSE1
    branch CONTINUE
  FALSE1:
    sub I3, I4, 1
    mul I2, I2, I3
    dec I4
    branch FATORIAL

END:
end
```

Como os opcodes da documentação do site www.parrotcode.org relativos às funções não foram reconhecidos, a tradução foi feita utilizando branch.

23. -

Listagem 3.46: Decide se um número é positivo, zero ou negativo com auxílio de uma função

```
.loadlib 'io_ops'


print "Digite um numero: "
read S1, 5
set I1, S1


branch VERIFICA


CONTINUE:

ne I2, 1, FALSE1
  say "Numero positivo"
  branch END
FALSE1:
  ne I2, 0, FALSE2
    say "Zero"
    branch END
  FALSE2:
    say "Numero negativo"

branch END


VERIFICA:
  le I1, 0, FALSEL1
    set I2, 1
    branch CONTINUE
  FALSEL1:
    ge I1, 0, FALSEL2
      set I2, -1
      branch FIMSEL1
    FALSEL2:
      set I2, 0
  FIMSEL1:
  branch CONTINUE

END:
end
```

O mesmo ocorrido ao código anterior.

3.2 Execução dos codigos

Para a execução de cada código .pasm utilizando a máquina Parrot VM, basta utilizar o seguinte comando no terminal:

```
> parrot <nome_do_arquivo>.pasm
```

Capítulo 4

Análise léxica

4.1 Analisador léxico simples

Neste capítulo encontra-se um analizador léxico simples, no qual o professor Alexsandro Santos Soares passou em sala como atividade.

4.1.1 Reconhecimento

As expressões reconhecidas por este analisador léxico seguem abaixo:

```
a := 1 + b
print (a*b)
if1 := a - 2
if if1 > 0
then print (if1)
else print (if2)
```

O analisador devolve uma lista de tokens (incluindo lexema e o token) que foram reconhecidos. Os diferentes tipos de tokens são:

- ID - Identificador. Ou seja, um nome de variável.
- := - Operador de atribuição.
- + - Operador de soma;
- - - Operador de subtração.
- > - Operador de maior.
- print - Palavra reservada.
- (e) - Abre e fecha parênteses.
- if - Palavra reservada.

- then - Palavra reservada.
- else - Palavra reservada.

4.1.2 Código

Este analisador foi implementado utilizando a linguagem funcional Ocaml. Segue no código abaixo, todas as funções utilizadas para a implementação do mesmo, juntamente com o autômato construído (em funções) para que possam ser reconhecidas as expressões encontradas em 4.1.1.

Listagem 4.1: Código do analisador léxico.

```
1 type estado = int
2 type entrada = string
3 type simbolo = char
4 type posicao = int
5
6 type dfa = {
7   transicao : estado -> simbolo -> estado;
8   estado: estado;
9   posicao: posicao
10 }
11
12 type token =
13  | If
14  | Then
15  | Else
16  | Print
17  | Maior
18  | AbrePar
19  | FechaPar
20  | Atribuic
21  | Addic
22  | Subtr
23  | Mult
24  | Id of string
25  | Int of string
26  | Branco
27  | EOF
28
29 type estado_lexico = {
30    pos_inicial: posicao; (* posição inicial na string *)
31    pos_final: posicao; (* posicao na string ao encontrar um estado final
          recente *)
32    ultimo_final: estado; (* último estado final encontrado *)
33    dfa : dfa;
34    rotulo : estado -> entrada -> token
35 }
36
37 let estado_morto:estado = -1
38
39 let estado_inicial:estado = 0
40
41 let eh_letra (c:simbolo) = ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z
       ')
42
```

```
43 let eh_digito (c:simbolo) = '0' <= c && c <= '9'
44
45 let eh_branco (c:simbolo) = c = ' ' || c = '\t' || c = '\n'
46
47 let eh_estado_final e el =
48   let rotulo = el.rotulo in
49   try
50       let _ = rotulo e "" in true
51   with _ -> false
52
53 let obtem_token_e_estado (str:entrada) el =
54   let inicio =  el.pos_inicial
55   and fim = el.pos_final
56   and estado_final = el.ultimo_final
57   and rotulo = el.rotulo in
58   let tamanho = fim - inicio + 1 in
59   let lexema = String.sub str inicio tamanho in
60   let token = rotulo estado_final lexema in
61   let proximo_el = { el with pos_inicial = fim + 1;
62                              pos_final = -1;
63                              ultimo_final = -1;
64                              dfa = { el.dfa with estado = estado_inicial;
65                                                  posicao = fim + 1 }}
66   in
67    (token, proximo_el)
68
69
70 let rec analisador (str:entrada) tam el =
71   let posicao_atual = el.dfa.posicao
72   and estado_atual = el.dfa.estado in
73   if posicao_atual >= tam
74   then
75     if el.ultimo_final >= 0
76     then let token, proximo_el = obtem_token_e_estado str el in
77       [token; EOF]
78     else [EOF]
79   else
80     let simbolo = str.[posicao_atual]
81     and transicao = el.dfa.transicao in
82     let proximo_estado = transicao estado_atual simbolo in
83     if proximo_estado = estado_morto
84     then let token, proximo_el = obtem_token_e_estado str el in
85       token :: analisador str tam proximo_el
86     else
87       let proximo_el =
88         if eh_estado_final proximo_estado el
89         then { el with pos_final = posicao_atual;
90                        ultimo_final = proximo_estado;
91                        dfa = { el.dfa with estado = proximo_estado;
92                                            posicao = posicao_atual + 1 }}
93         else { el with dfa = { el.dfa with estado = proximo_estado;
94                                            posicao = posicao_atual + 1 }}
95       in
96       analisador str tam proximo_el
97
98 let lexico (str:entrada) =
99   let trans (e:estado) (c:simbolo) =
100    match (e,c) with
101    | (0, '*') -> 1
```

```
102     | (0, '-') -> 2
103     | (0, '+') -> 3
104     | (0, '>') -> 4
105     | (0, '(') -> 5
106     | (0, ')') -> 6
107     | (0, ':') -> 7
108     | (0, _) when eh_digito c -> 9
109     | (0, 'i') -> 10
110     | (0, 'p') -> 12
111     | (0, 'e') -> 17
112     | (0, 't') -> 21
113     | (0, _) when eh_branco c -> 25
114     | (0, _) when eh_letra c -> 26
115     | (0, _) ->
116         failwith ("Erro lexico: caracter desconhecido " ^ Char.escaped c)
117     | (7, '=') -> 8
118     | (9, _) when eh_digito c -> 9
119     | (10, 'f') -> 11
120     | (10, _)  when eh_letra c || eh_digito c-> 26
121     | (11, _)  when eh_letra c || eh_digito c-> 26
122     | (12, 'r')  -> 13
123     | (12, _)  when eh_letra c || eh_digito c-> 26
124     | (13, 'i')  -> 14
125     | (13, _)  when eh_letra c || eh_digito c-> 26
126     | (14, 'n')  -> 15
127     | (14, _)  when eh_letra c || eh_digito c-> 26
128     | (15, 't')  -> 16
129     | (15, _)  when eh_letra c || eh_digito c-> 26
130     | (16, _)  when eh_letra c || eh_digito c-> 26
131     | (17, 'l')  -> 18
132     | (17, _)  when eh_letra c || eh_digito c-> 26
133     | (18, 's')  -> 19
134     | (18, _)  when eh_letra c || eh_digito c-> 26
135     | (19, 'e')  -> 20
136     | (19, _)  when eh_letra c || eh_digito c-> 26
137     | (20, _)  when eh_letra c || eh_digito c-> 26
138     | (21, 'h')  -> 22
139     | (21, _)  when eh_letra c || eh_digito c-> 26
140     | (22, 'e')  -> 23
141     | (22, _)  when eh_letra c || eh_digito c-> 26
142     | (23, 'n')  -> 24
143     | (23, _)  when eh_letra c || eh_digito c-> 26
144     | (24, _)  when eh_letra c || eh_digito c-> 26
145     | (25, _)  when eh_branco c -> 25
146     | (26, _)  when eh_letra c || eh_digito c -> 26
147     | _ -> estado_morto
148 and rotulo e str =
149  match e with
150   | 1 -> Mult
151   | 2 -> Subtr
152   | 3 -> Addic
153   | 4 -> Maior
154   | 5 -> AbrePar
155   | 6 -> FechaPar
156   | 8 -> Atribuic
157   | 9 -> Int str
158   | 11 -> If
159   | 16 -> Print
160   | 20 -> Else
```

```
161   | 24 -> Then
162   | 10
163   | 12
164   | 13
165   | 14
166   | 15
167   | 17
168   | 18
169   | 19
170   | 21
171   | 22
172   | 23
173   | 26 -> Id str
174   | 25 -> Branco
175   | _ -> failwith ("Erro lexico: sequencia desconhecida " ^ str)
176 in let dfa = { transicao = trans;
177                estado = estado_inicial;
178                posicao = 0 }
179 in let estado_lexico = {
180   pos_inicial = 0;
181   pos_final = -1;
182   ultimo_final = -1;
183   rotulo = rotulo;
184   dfa = dfa
185 } in
186   analisador str (String.length str) estado_lexico
```

4.1.3 Execução

Para que possa executar o analisador léxico, deve estar instalado o Ocaml. Seu tutorial de instalação encontra-se em 2.1.

1. Deve-se no terminal, abrir o Ocaml:

   ```
   > ocaml
   ```

2. Agora, para que o Ocaml possa compilar o código, deve-se digitar o comando abaixo:

   ```
   > #use "Nome_do_arquivo.ml";;
   ```

3. Agora já compilado, pode-se testar o analisador utilizando o comando "lexico". Segue abaixo um exemplo:

   Testando:

   ```
   # lexico "if if1 then then1 print(a + b)";;
   ```

   Retorno:

   ```
   - : token list = [If; Branco; Id "if1"; Branco; Then; Branco; Id "
     then1"; Branco; Print; AbrePar; Id "a"; Branco; Addic; Branco; Id
     "b"; FechaPar; EOF]
   ```

4.2 Analisador léxico do Mini-Python

Nesta seção encontra-se os tokens do Mini-Python, juntamente com o código em OCaml das definições para o analisador léxico que o OCaml já possui, e como executá-lo.

4.2.1 Reconhecimento

O analisador do OCaml devolve uma lista de tokens (incluindo lexema e o token) que foram reconhecidos. Os diferentes tipos de tokens são (Token, e seu formato):

- FALSE - "False"
- NONE - "None"
- TRUE - "True"
- AND - "and"
- AS - "as"
- BREAK - "break"
- CONTINUE - "continue"
- DEF - "def"
- DEL - "del"
- ELIF - "elif"
- ELSE - "else"
- EXCEPT - "except"
- FOR - "for"
- FROM - "from"
- IF - "if"
- IMPORT - "import"
- IN - "in"
- IS - "is"
- NOT - "not"
- OR - "or"
- RETURN - "return"
- WHILE - "while"

- WITH - "with"
- ADICAO - "+"
- SUBTRACAO - -"
- MULTIPLICACAO - "*"
- DIVISAO - "/"
- DIVISAOINTEIRO - "//"
- MODULO - "%"
- EXPONENCIACAO - "**"
- EQUIVALENTE - "=="
- NAOEQUIVALENTE - "!="
- MENOR - «"
- MAIOR - »"
- MENORIGUAL - «="
- MAIORIGUAL - »="
- IGUAL - "="
- ABREPARENTESE - "("
- FECHAPARENTESE - ")"
- ABRECOLCHETE - "["
- FECHACOLCHETE - "]"
- ABRECHAVES - "{"
- FECHACHAVES - "}"
- PONTO - "."
- VIRGULA - ","
- DOISPONTOS - ":"
- PONTOEVIRGULA - ";"
- ARROBA - "@"
- ADICAOIGUAL - "+="
- SUBTRACAOIGUAL - -="
- MULTIPLICACAOIGUAL - "*="
- DIVISAOIGUAL - "/="

- DIVISAOINTEIROIGUAL - "//="
- MODULOIGUAL - "%="
- EXPONENCIACAOIGUAL - "**/"

4.2.2 Código

O analisador léxico utilizado foi o OCamlLex, que é o analisador disponibilizado pelo OCaml. Segue abaixo, o código de todas as definições necessárias para o OCamlLex definir um analisador para reconhecer os tokens em 4.2.

Listagem 4.2: Código para o analisador léxico.

```
{
  open Lexing
  open Printf

  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

type tokens = FALSE
  | NONE
  | TRUE
  | AND
  | AS
  | BREAK
  | CONTINUE
  | DEF
  | DEL
  | ELIF
  | ELSE
  | EXCEPT
  | FOR
  | FROM
  | IF
  | IMPORT
  | IN
  | IS
  | NOT
  | OR
  | RETURN
  | WHILE
  | WITH
  | ADICAO
  | SUBTRACAO
  | MULTIPLICACAO
```

```
44  | DIVISAO
45  | DIVISAOINTEIRO
46  | MODULO
47  | EXPONENCIACAO
48  | EQUIVALENTE
49  | NAOEQUIVALENTE
50  | MENOR
51  | MAIOR
52  | MENORIGUAL
53  | MAIORIGUAL
54  | IGUAL
55  | ABREPARENTESE
56  | FECHAPARENTESE
57  | ABRECOLCHETE
58  | FECHACOLCHETE
59  | ABRECHAVES
60  | FECHACHAVES
61  | PONTO
62  | VIRGULA
63  | DOISPONTOS
64  | PONTOEVIRGULA
65  | ARROBA
66  | ADICAOIGUAL
67  | SUBTRACAOIGUAL
68  | MULTIPLICACAOIGUAL
69  | DIVISAOIGUAL
70  | DIVISAOINTEIROIGUAL
71  | MODULOIGUAL
72  | EXPONENCIACAOIGUAL
73  | LITINT of int
74  | LITSTRING of string
75  | ID of string
76  | EOF
77
78
79 }
80
81 let digito = ['0' - '9']
82 let inteiro = digito+
83
84 let letra = ['a' - 'z' 'A' - 'Z']
85 let identificador = letra ( letra | digito | '_')*
86
87 let brancos = [' ' '\t']+
88 let novalinha = '\r' | '\n' | "\r\n"
89
90 let comentario = "#" [^ '\r' '\n' ]*
91
92
93 rule token = parse
94   brancos { token lexbuf}
95 | comentario  { token lexbuf}
96 |  "False"  { FALSE }
97 | "None"  { NONE }
98 | "True"  { TRUE }
99 | "and"   { AND }
100 | "as"    { AS }
101 | "break" { BREAK }
102 | "continue"  { CONTINUE }
```

```
103 | "def"    { DEF }
104 | "ELIF"   { ELIF }
105 | "else"   { ELSE }
106 | "except"   { EXCEPT }
107 | "for"    { FOR }
108 | "from"   { FROM }
109 | "if"     { IF }
110 | "import"   { IMPORT }
111 | "in"     { IN }
112 | "is"     { IS }
113 | "not"    { NOT }
114 | "or"     { OR }
115 | "return"   { RETURN }
116 | "while"  { WHILE }
117 | "with"   { WITH }
118 | "+"    { ADICAO }
119 | "-"    { SUBTRACAO }
120 | "*"    { MULTIPLICACAO }
121 | "/"    { DIVISAO }
122 | "//"     { DIVISAOINTEIRO }
123 | "%"    { MODULO }
124 | "**"     { EXPONENCIACAO }
125 | "=="     { EQUIVALENTE }
126 | "!="     { NAOEQUIVALENTE }
127 | "<"    { MENOR }
128 | ">"    { MAIOR }
129 | "<="     { MENORIGUAL }
130 | ">="     { MAIORIGUAL }
131 | "="    { IGUAL }
132 | "("    { ABREPARENTESE }
133 | ")"    { FECHAPARENTESE }
134 | "["    { ABRECOLCHETE }
135 | "]"    { FECHACOLCHETE }
136 | "{"    { ABRECHAVES }
137 | "}"    { FECHACHAVES }
138 | "."    { PONTO }
139 | ","    { VIRGULA }
140 | ":"    { DOISPONTOS }
141 | ";"    { PONTOEVIRGULA }
142 | "@"    { ARROBA }
143 | "+="     { ADICAOIGUAL }
144 | "-="     { SUBTRACAOIGUAL }
145 | "*="     { MULTIPLICACAOIGUAL }
146 | "/="     { DIVISAOIGUAL }
147 | "//="    { DIVISAOINTEIROIGUAL }
148 | "%="     { MODULOIGUAL }
149 | "**="    { EXPONENCIACAOIGUAL }
150 | brancos     { token lexbuf }
151 | novalinha  { incr_num_linha lexbuf; token lexbuf }
152 | comentario { token lexbuf }
153 | "''''"       { comentario_bloco 0 lexbuf }
154 | inteiro as num { let numero = int_of_string num in
155                    LITINT numero  }
156 | identificador as id { ID id }
157 | '"'        { let buffer = Buffer.create 1 in
158                let str = leia_string buffer lexbuf in
159                 LITSTRING str }
160 | _ as c  { failwith (msg_erro lexbuf c) }
161 | eof        { EOF }
```

```
and comentario_bloco n = parse
    """"    { if n=0 then token lexbuf
              else comentario_bloco (n-1) lexbuf }
| """"      { comentario_bloco (n+1) lexbuf }
| _         { comentario_bloco n lexbuf }
| eof       { failwith "Comentário não fechado" }
and leia_string buffer = parse
    '"'     { Buffer.contents buffer}
| "\\t"     { Buffer.add_char buffer '\t'; leia_string buffer lexbuf }
| "\\n"     { Buffer.add_char buffer '\n'; leia_string buffer lexbuf }
| '\\' '"'  { Buffer.add_char buffer '"'; leia_string buffer lexbuf }
| '\\' '\\'  { Buffer.add_char buffer '\\'; leia_string buffer lexbuf }
| _ as c     { Buffer.add_char buffer c; leia_string buffer lexbuf }
| eof       { failwith "A string não foi fechada"}
```

4.2.3 Execução

Para que possa executar o analisador léxico, deve estar instalado o Ocaml. Seu tutorial de instalação encontra-se em 2.1

1. Deve-se no terminal, compilar o arquivo lexico.mll para gerar o arquivo lexico.ml:

   ```
   > ocamllex lexico.mll
   ```

2. Agora, o compilador do OCaml deve compilar o arquivo lexico.ml, gerando o arquivo "carregador.ml", utilizando o comando abaixo:

   ```
   > ocamlc -c lexico.ml
   ```

3. Já compilado, deve-se abrir o OCaml utilizando o seguinte comando:

   ```
   >ocaml
   ```

4. Para que o OCaml possa carregar o analisador léxico, deve-se digitar o seguinte comando:

   ```
   # #use "carregador.ml";;
   ```

   Analisador carregado.

5. Neste momento já é possível testá-lo. Pode-se utilizar da seguinte forma, para analisar o codigo escrito em um arquivo chamado "codigo":

   ```
   # lex "codigo";;
   ```

   Supondo o seguinte arquivo de código abaixo:

   Listagem 4.3: Exemplo de código de Mini-Python.

   ```
   def funcao():

     a = "aquiehstring"
     x = 2*3 + 1

   ```

```
6    ''' desconsidera
7
8  desconsidera
9  desconsidera '''
10   return a
11
12 # desconsidera tambem
13
14
15 a += 2
16 b -= 3
```

A saída do analisador léxico é a seguinte:

```
- : Lexico.tokens list =
                      [Lexico.DEF; Lexico.ID "funcao"; Lexico.
                         ABREPARENTESE; Lexico.FECHAPARENTESE; Lexico.
                         DOISPONTOS; Lexico.ID "a"; Lexico.IGUAL;
                         Lexico.LITSTRING "aquiehstring"; Lexico.ID "x"
                         ; Lexico.IGUAL; Lexico.LITINT 2; Lexico.
                         MULTIPLICACAO; Lexico.LITINT 3; Lexico.ADICAO;
                          Lexico.LITINT 1; Lexico.RETURN; Lexico.ID "a"
                         ; Lexico.ID "a"; Lexico.ADICAOIGUAL; Lexico.
                         LITINT 2; Lexico.ID "b"; Lexico.SUBTRACAOIGUAL
                         ;Lexico.LITINT 3; Lexico.EOF]
```

Capítulo 5

Análise sintática

Este capítulo destina-se à construção de um analisador sintático.

5.1 Analisador sintatico passado em aula

Essa seção contém a implementação de um analisador sintático preditivo que foi passado em sala de aula pelo professor Alexsandro Santos Soares.

5.1.1 Gramática

Segue abaixo a gramática em que o analisador sintático reconhece.

S -> XYZ

X -> aXb

X ->

Y -> cYZcX

Y -> d

Z -> eZYe

Z -> f

Sendo letras maiúsculas variáveis, e letras minúsculas símbolos terminais.

5.1.2 Códigos

Segue os códigos do analisador sintático, juntamente com um arquivo com uma lista de tokens para esse analisador.

Listagem 5.1: Lista de tokens

```
type tokens = A
            | B
            | C
            | D
            | E
            | F
            | EOF
```

Listagem 5.2: Codigo do analisador sintatico

```
(* Parser preditivo *)
#load "lexico.cmo";;
open Sintatico;;

type variavel = S of variavel * variavel * variavel
            | X of tokens * variavel * tokens
            | Y of tokens * variavel * variavel * tokens * variavel
            | Z of tokens * variavel * variavel * tokens
            | X_vazio
            | Y_d of tokens
            | Z_f of tokens

let tk = ref EOF (* variável global para o token atual *)
let lexbuf = ref (Lexing.from_string "")

(* lê o próximo token *)
let prox () = tk := Lexico.token !lexbuf

let to_str tk =
  match tk with
    A -> "a"
  | B -> "b"
  | C -> "c"
  | D -> "d"
  | E -> "e"
  | F -> "f"
  | EOF -> "eof"

let erro esp =
  let msg = Printf.sprintf "Erro: esperava %s mas encontrei %s"
                           esp (to_str !tk)
  in
  failwith msg

let consome t = if (!tk == t) then prox() else erro (to_str t)

let rec ntS () =
  match !tk with
    A
   |C
   |D     ->
              let cmd1 = ntX() in
              let cmd2 = ntY() in
              let cmd3 = ntZ() in
              S (cmd1, cmd2, cmd3)
  | _ -> erro "a, c ou d"
and ntX () =
  match !tk with
```

```
49      B
50     |C
51     |D
52     |E
53     |F     -> X_vazio
54     |A     -> let _ = consome A in
55               let cmd = ntX() in
56               let _ = consome B in
57               X (A, cmd, B)
58     | _ -> erro "a"
59 and ntY () =
60   match !tk with
61     C     -> let _ = consome C in
62              let cmd = ntY() in
63              let cmd2 = ntZ() in
64              let _ = consome C in
65              let cmd3 = ntX() in
66              Y (C,cmd,cmd2, C, cmd3)
67    |D      -> let _ = consome D in
68              Y_d (D)
69    |_      -> erro "c ou d"
70 and ntZ () =
71   match !tk with
72     E     -> let _ = consome E in
73              let cmd = ntZ() in
74              let cmd2 = ntY() in
75              let _ = consome E in
76              Z (E, cmd, cmd2, E)
77    |F     -> let _ = consome F in
78              Z_f (F)
79    |_     -> erro "e ou f"
80
81 let parser str =
82   lexbuf := Lexing.from_string str;
83   prox (); (* inicializa o token *)
84   let arv = ntS () in
85   match !tk with
86     EOF -> let _ = Printf.printf "Ok!\n" in arv
87   | _ -> erro "fim da entrada"
88
89 let testar_entrada entrada =
90   parser entrada
```

5.1.3 Execução

Para que possa executar o analisador sintático, deve estar instalado o Ocaml. Seu tutorial de instalação encontra-se em 2.1

1. Deve-se no terminal, compilar o arquivo lexico.mll para gerar o arquivo lexico.ml:

```
> ocamllex lexico.mll
```

2. Agora, o compilador do OCaml deve compilar o arquivo lexico.ml, gerando o arquivo "carregador.ml", utilizando o comando abaixo:

```
> ocamlc -c lexico.ml
```

3. Já compilado, deve-se abrir o OCaml utilizando o seguinte comando:

```
>ocaml
```

4. Para que o OCaml possa carregar o analisador léxico, para que o sintático possa funcionar, deve-se digitar o seguinte comando:

```
# #load "lexico.cmo";;
```

   Analisador carregado.

5. Para que o OCaml possa utilizar o analisador sintático, deve-se digitar o seguinte comando:

```
# #use "sintaticoArv.ml";;
```

   Analisador carregado, e utilizando o sintático.

6. Neste momento já é possível testá-lo. Pode-se utilizar da seguinte forma:

```
# testar_entrada("ENTRADA");;
```

   Sendo a entrada passada por parâmetro, substituindo a palavra ENTRADA, pela entrada desejada.

   Supondo a seguinte entrada que é aceita:

```
"cdfcf"
```

   A saída do analisador sintático é a seguinte:

```
- : variavel = S (X_vazio, Y (C, Y_d D, Z_f F, C, X_vazio), Z_f F)
```

   Supondo a seguinte entrada que não é aceita:

```
"cdfcfaaa"
```

   A saída do analisador sintático é a seguinte:

```
Exception: Failure "Erro: esperava fim da entrada mas encontrei a".
```

5.2 Analisador sintático do Mini-Python

Essa seção contém a implementação do analisador sintático preditivo para o reconhecimento da gramática do Mini-Python. Ele foi construído com o auxílio do Menhir, que é um gerador de parser, que tem suporte para Ocaml.

5.2.1 Gramática e Códigos

Segue no arquivo abaixo o código em Ocaml, lido pelo Menhir, da gramática do Mini-Python.

Listagem 5.3: Gramática do Mini-Python (parser.mly)

```
%{
  open Ast

%}
%token <float> LITFLOAT
%token <string> ID
%token <string> LITSTRING
%token <int> LITINT
%token IMPORT
%token DEF
%token ASPASSIMPLES
%token DOISPONTOS
%token PONTO
%token VIRG
%token MAIOR
%token MAIORIGUAL
%token MENOR
%token MENORIGUAL
%token DIFERENTE
%token IGUALDADE
%token ADICAO
%token SUBTRACAO
%token DIVISAO
%token MULTIPLICACAO
%token MODULO
%token ABREPARENTESE
%token FECHAPARENTESE
%token ABRECOLCHETES
%token FECHACOLCHETES
%token ATRIB
%token IN
%token PRINT
%token INPUT
%token WHILE
%token FOR
%token IF
%token ELSE
%token RANGE
%token LEN
%token OU
%token E
%token NOT
%token RETURN
%token TRUE
%token FALSE
%token ASPASDUPLAS
%token INCREMENTA
%token DECREMENTA
%token SETA
%token EOF
%token ADICAOIGUAL
%token ATRIBSUBTRACAO
```

```
%token ATRIBMULT
%token ATRIBDIV
%token ABRECHAVES
%token FECHACHAVES
%token PV
%token LIST
%token PASS
%token ELIF
%token VOID
%token STR
%token BREAK
%token IS
%token FROM
%token INT
%token FLOAT
%token REAL
%token DOUBLE
%token CHAR
%token INCR
%token DECR
%token BOOL
%token  <int * int * token list> Linha
%token INDENTA
%token DEDENTA
%token NOVALINHA

%left OU
%left E
%left IGUALDADE DIFERENTE
%left MAIOR MAIORIGUAL MENOR MENORIGUAL
%left ADICAO SUBTRACAO
%left MULTIPLICACAO DIVISAO MODULO

%start <Ast.prog> prog

%%

prog:
    | s=seq+ EOF          { Prog(s) }
    | s=seq+ NOVALINHA EOF        { Prog(s) }
    | NOVALINHA s=seq+ EOF        { Prog(s) }
    | NOVALINHA s=seq+ NOVALINHA EOF      { Prog(s) }
    ;


seq:
  | e=expr    { Expressao(e) }
  | c=comando     { Comando(c) }
  | f=funcao    { Funcao(f) }
  ;


expr:
   | o=operacao NOVALINHA   { ExprOperacao(o) }
   | IMPORT c=comando NOVALINHA   { ExprImport(c) }
   | FROM c1=comando IMPORT c2=comando NOVALINHA    { ExprFromImport(c1,c2
      ) }
   | INPUT ABREPARENTESE FECHAPARENTESE NOVALINHA   { ExprInput }
   | c=comando ATRIB o=operacao NOVALINHA   { ExprAtribCmdOp(c,o) }
```

```
111     | c1=comando ATRIB c2=comando NOVALINHA    { ExprAtribCmdCmd(c1,c2) }
112     | c=comando ATRIB INT ABREPARENTESE INPUT ABREPARENTESE FECHAPARENTESE
            FECHAPARENTESE NOVALINHA    { ExprAtribCmdInput(c) }
113     | c=comando ATRIB INPUT ABREPARENTESE FECHAPARENTESE NOVALINHA     {
            ExprAtribCmdInput(c) }
114     | c1=comando ATRIB c2=comando ABREPARENTESE c3=comando FECHAPARENTESE
            NOVALINHA    { ExprAtribCmdCmdCmd(c1,c2,c3) }
115     | c1=comando ATRIB c2=comando ABREPARENTESE FECHAPARENTESE NOVALINHA
              { ExprAtribCmdCmd(c1,c2) }
116     | PRINT ABREPARENTESE c=comando FECHAPARENTESE NOVALINHA     {
            ExprPrint(c) }
117     | PRINT ABREPARENTESE c1=comando VIRG c2=comando FECHAPARENTESE
            NOVALINHA    { ExprPrintCmd(c1,c2) }
118     | PRINT ABREPARENTESE c1=comando ADICAO c2=comando FECHAPARENTESE
            NOVALINHA    { ExprPrintCmd(c1,c2) }
119     | PRINT ABREPARENTESE c1=comando ADICAO STR ABREPARENTESE c2=comando
            FECHAPARENTESE FECHAPARENTESE NOVALINHA     { ExprPrintCmd(c1,c2) }
120     | WHILE v=verificacao DOISPONTOS NOVALINHA
121      INDENTA s=seq+ DEDENTA    { ExprWhileVerificacao(v,s) }
122     | WHILE b=booleano DOISPONTOS NOVALINHA
123      INDENTA s=seq+ DEDENTA    { ExprWhileBooleano(b,s) }
124     | WHILE c=comando DOISPONTOS NOVALINHA
125      INDENTA s=seq+ DEDENTA    { ExprWhileCmd(c,s) }
126     | FOR c1=comando IN RANGE ABREPARENTESE c2=comando FECHAPARENTESE
            DOISPONTOS NOVALINHA
127      INDENTA s=seq+ DEDENTA    { ExprForRange(c1,c2,s) }
128     | FOR c1=comando IN c2=comando DOISPONTOS NOVALINHA
129      INDENTA s=seq+ DEDENTA    { ExprForId(c1,c2,s) }
130     | IF v=verificacao DOISPONTOS NOVALINHA
131      INDENTA s=seq+ DEDENTA    { ExprIf(v,s) }
132     | ELIF v=verificacao DOISPONTOS NOVALINHA
133      INDENTA s=seq+ DEDENTA    { ExprElif(v,s) }
134     | ELSE DOISPONTOS NOVALINHA
135      INDENTA s=seq+ DEDENTA    { ExprElse(s) }
136     | RETURN c=comando NOVALINHA     { ExprReturn(c) }
137     | STR ABREPARENTESE c=comando FECHAPARENTESE NOVALINHA     {
            ExprStrCast(c) }
138     | INT ABREPARENTESE c=comando FECHAPARENTESE NOVALINHA   { ExprIntCast(
            c) }
139     ;
140
141 comando:
142     | l=LITINT    { LITINT(l) }
143     | l=LITFLOAT    { LITFLOAT(l) }
144     | l=LITSTRING     { LITSTRING(l) }
145     | b=booleano    { BOOLEANO(b) }
146     | i=ID  { ID(i) }
147     ;
148
149 parametro:
150     | c=comando tp=tiposPrimitivos    { Param(c,tp) }
151     ;
152
153 funcao:
154     | DEF c=comando ABREPARENTESE p=parametro* FECHAPARENTESE DOISPONTOS
          SETA tp=tipos NOVALINHA
155      INDENTA s=seq+ DEDENTA     { DefFuncao(c,p,tp,s) }
156       ;
157
```

```
158 op:
159   | ADICAO      { Adicao }
160   | SUBTRACAO       { Subtracao }
161   | DIVISAO       { Divisao }
162   | MULTIPLICACAO     { Multiplicacao }
163   | MODULO      { Modulo }
164   ;
165
166 operacao_op:
167    | c1=comando o=op c2=comando     { Operacao_op(c1,o,c2) }
168    ;
169
170 operacao:
171   | op = operacao_op      { Operacao(op) }
172   | op1=operacao_op o=op op2=operacao_op    { OperacaoOperacao(op1,o,op2)
        }
173   | op=operacao_op o=op c=comando     { OperacaoComando(op,o,c) }
174   ;
175
176 comparador:
177   | MENORIGUAL    { MenorIgual }
178   | IGUALDADE     { Igualdade }
179   | MAIORIGUAL    { MaiorIgual }
180   | MAIOR     { Maior }
181   | MENOR     { Menor }
182   ;
183
184 comparacao:
185    | c1=comando o=comparador c2=comando     { Comparacao(c1,o,c2) }
186    ;
187
188 logica:
189   | E     { ELogico }
190   | OU    { OULogico }
191   ;
192
193 verificacao:
194   | c=comparacao          { Verificacao(c) }
195   | c1=comparacao l=logica c2=comparacao    { VerificacaoDupla(c1,l,c2) }
196   | ABREPARENTESE c1=comparacao l1=logica c2=comparacao FECHAPARENTESE l2=
        logica ABREPARENTESE c3=comparacao l3=logica c4=comparacao
        FECHAPARENTESE     { VerificacaoMultipla(c1,l1,c2,l2,c3,l3,c4) }
197   ;
198
199 tipos:
200   | BOOL  { BOOL }
201   | INT   { INT }
202   | FLOAT { FLOAT }
203   | STR   { STR }
204   ;
205
206 tiposPrimitivos:
207   | DOISPONTOS BOOL     { BOOL }
208   | DOISPONTOS INT    { INT }
209   | DOISPONTOS FLOAT    { FLOAT }
210   | DOISPONTOS STR    { STR }
211   ;
212
213 booleano:
```

```
214   | TRUE         { Verdadeiro }
215   | FALSE        { Falso }
216   ;
```

5.2.2 Árvore Sintática

Segue abaixo o código em Ocaml, da configuração da árvore semântica da linguagem mencionada.

Listagem 5.4: Árvore Sintática do Mini-Python (ast.ml)

```
1  type identificador = string
2
3  type prog = Prog of seq list
4
5  and seq = Expressao of expr
6      | Comando of cmd
7      | Funcao of func
8
9  and cmd = LITINT of int
10       | LITFLOAT of float
11       | LITSTRING of string
12       | BOOLEANO of valor_logico
13       | ID of string
14
15 and valor_logico = Verdadeiro
16          | Falso
17
18 and parametro = Param of cmd * tiposPrimitivos
19
20 and tiposPrimitivos = BOOL
21                | INT
22                | FLOAT
23                | STR
24
25 and func = DefFuncao of cmd * parametro list * tiposPrimitivos * seq list
26
27 and op = Adicao
28      | Subtracao
29      | Divisao
30      | Multiplicacao
31      | Modulo
32
33 and operacao_op = Operacao_op of cmd * op * cmd
34
35 and operacao = Operacao of operacao_op
36        | OperacaoOperacao of operacao_op * op * operacao_op
37        | OperacaoComando of operacao_op * op * cmd
38
39 and cmp = MenorIgual
40     | Igualdade
41     | MaiorIgual
42     | Maior
43     | Menor
44
45 and comparacao = Comparacao of cmd * cmp * cmd
```

```
46
47 and logica = ELogico
48         | OULogico
49
50 and verificacao = Verificacao of comparacao
51              | VerificacaoDupla of comparacao * logica * comparacao
52              | VerificacaoMultipla of comparacao * logica * comparacao *
                   logica  * comparacao * logica * comparacao
53
54 and expr = ExprImport of cmd
55       | ExprFromImport of cmd * cmd
56       | ExprOperacao of operacao
57       | ExprInput
58       | ExprAtribCmdOp of cmd * operacao
59       | ExprAtribCmdCmd of cmd * cmd
60       | ExprAtribCmdCmdCmd of cmd * cmd * cmd
61       | ExprAtribCmdInput of cmd
62       | ExprPrint of cmd
63       | ExprPrintCmd of cmd * cmd
64       | ExprWhileVerificacao of verificacao * seq list
65       | ExprWhileBooleano of valor_logico * seq list
66       | ExprWhileCmd of cmd * seq list
67       | ExprForRange of cmd *cmd * seq list
68       | ExprForId of cmd * cmd * seq list
69       | ExprIf of verificacao * seq list
70       | ExprElif of verificacao * seq list
71       | ExprElse of seq list
72       | ExprReturn of cmd
73       | ExprStrCast of cmd
74       | ExprIntCast of cmd
```

5.2.3 Execução

Para que possa executar o analisador sintático, deve estar instalado o Ocaml. Seu tutorial de instalação encontra-se em 2.1

É necessário a instalação do Menhir, para que possam ser compilados os códigos da seção anterior, da seguinte maneira:

```
> sudo apt-get install menhir
```

1. Deve-se no terminal, no diretório principal do compilador, compilar o arquivo main.ml da seguinte maneira:

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

2. Agora, basta abrir o Ocaml:

   ```
   > rlwrap ocaml
   ```

3. Aberto, é necessário utilizar o arquivo "main.ml":

   ```
   # #use "main.ml";;
   ```

4. Para que possa testar um arquivo escrito em Mini-Python com nome de "algoritmo.py", basta digitar o seguinte comando:

```
# parse_arq "algoritmo.py";;
```

Supondo o seguinte algoritmo abaixo, a resposta da árvore sintática segue:

Listagem 5.5: Algoritmo.py

```
1  if 3 > 2:
2    print(b)
3  else:
4    a = 4
5
6
7  while True:
8    print ("exibe")
9
10 for a in range(b):
11   print ("loop")
12
13
14 def nano10(num1:int num2:int num3:float): -> int
15   num1=1
16   num2=2
17   if num1 == num2:
18     print(num1)
19   else:
20     if a > b:
21         print("aqui")
22     print(0)
23   print(a)
24
25 #desconsidera
```

Árvore sintática, juntamente com os níveis de identação:

```
Linha(identacao=0,nivel_par=0)
Nivel: 0
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=0,nivel_par=0)
Nivel: 0
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=0,nivel_par=0)
Nivel: 0
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=0,nivel_par=0)
Nivel: 0
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=0,nivel_par=0)
Nivel: 0
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=8,nivel_par=0)
```

```
Nivel: 8
Linha(identacao=16,nivel_par=0)
Nivel: 16
Linha(identacao=8,nivel_par=0)
Nivel: 8
Linha(identacao=16,nivel_par=0)
Nivel: 16
Linha(identacao=32,nivel_par=0)
Nivel: 32
Linha(identacao=16,nivel_par=0)
Nivel: 16
Linha(identacao=8,nivel_par=0)
Nivel: 8
EOF
- : Ast.prog =
Ast.Prog
 [Ast.Expressao
   (Ast.ExprIf
     (Ast.Verificacao
       (Ast.Comparacao (Ast.LITINT 3, Ast.Maior, Ast.LITINT 2)),
     [Ast.Expressao (Ast.ExprPrint (Ast.ID "b"))]));
  Ast.Expressao
   (Ast.ExprElse
     [Ast.Expressao (Ast.ExprAtribCmdCmd (Ast.ID "a", Ast.LITINT 4))
        ]);
  Ast.Expressao
   (Ast.ExprWhileCmd (Ast.ID "True",
     [Ast.Expressao (Ast.ExprPrint (Ast.LITSTRING "exibe"))]));
  Ast.Expressao
   (Ast.ExprForRange (Ast.ID "a", Ast.ID "b",
     [Ast.Expressao (Ast.ExprPrint (Ast.LITSTRING "loop"))]));
  Ast.Funcao
   (Ast.DefFuncao (Ast.ID "nano10",
     [Ast.Param (Ast.ID "num1", Ast.INT); Ast.Param (Ast.ID "num2",
        Ast.INT);
      Ast.Param (Ast.ID "num3", Ast.FLOAT)],
     Ast.INT,
     [Ast.Expressao (Ast.ExprAtribCmdCmd (Ast.ID "num1", Ast.LITINT
        1));
      Ast.Expressao (Ast.ExprAtribCmdCmd (Ast.ID "num2", Ast.LITINT
         2));
      Ast.Expressao
       (Ast.ExprIf
         (Ast.Verificacao
           (Ast.Comparacao (Ast.ID "num1", Ast.Igualdade, Ast.ID "
              num2")),
         [Ast.Expressao (Ast.ExprPrint (Ast.ID "num1"))]));
      Ast.Expressao
       (Ast.ExprElse
         [Ast.Expressao
           (Ast.ExprIf
             (Ast.Verificacao
               (Ast.Comparacao (Ast.ID "a", Ast.Maior, Ast.ID "b")),
             [Ast.Expressao (Ast.ExprPrint (Ast.LITSTRING "aqui"))]))
              ;
          Ast.Expressao (Ast.ExprPrint (Ast.LITINT 0))]);
      Ast.Expressao (Ast.ExprPrint (Ast.ID "a"))]))]
```

5.2.4 Mensagens de Erro no Analisador Sintático

Nesta seção mensagens de erro foram adicionadas para informar ao programador de um possível erro sintático. Para gerar as mensagens de erro no ocaml basta usar o seguinte comando:

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

O próximo passo é configurar as mensagens no arquivo gerado: parser.msg. As mensagens devem substituir os fragmentos de código dado por: «YOUR SYNTAX ERROR MESSAGE HERE>". A seguir, um trecho do arquivo parser.msg, no qual é configurada uma mensagem para um erro na de falta de operador binário ('+', '-', etc ou o token 'do'). A mensagem pode ser observada na linha 12.

```
01. program: WHILE TRUE WHILE
02. ##
03. ## Ends in an error in state: 202.
04. ##
05. ## exp -> exp . binop exp [ SUBTRACAO SOMA PONTOPONTO OR
    MULTIPLICACAO
MODULO MENORIGUAL MENOR MAIORIGUAL MAIOR IGUALDADE EXPONENCIACAO DO
DIVISAO DIFERENTE AND ]
06. ## stat -> WHILE exp . DO block END [ WHILE UNTIL RETURN REPEAT
    PRINT
PONTOEVIRGULA LOCAL IF ID GOTO FUNCTION FOR EOF END ELSEIF ELSE
DOISDOISPONTOS DO BREAK ABREPARENTESE ]
07. ##
08. ## The known suffix of the stack is as follows:
09. ## WHILE exp
10. ##
11.
12. Esperava: operador binario ou 'do'
```

O próximo passo é configurar o arquivo oculto .ocamlinit

```
#use "topfind";;
#require "menhirLib";;
#directory "_build";;
#load "erroSint.cmo";;
#load "parser.cmo";;
#load "lexer.cmo";;
#load "ast.cmo";;
#load "main.cmo";;
open Ast
open Main
```

Feita as devidas configurações, basta compilar:

```
> menhir -v --list-errors parser.mly --compile-errors parser.msg >
erroSint.ml
> ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -
    package
menhirLib main.byte
```

O próximo passo é abrir e executar o analisador sintático para o arquivo de nome "arquivo"no ocaml:

```
> rlwrap ocaml
> # parse_arq "arquivo";;
```

Caso o arquivo "arquivo"não contenha erro sintático, então será gerada a árvore de derviação. Do contrário, uma mensagem de erro será exibida. Uma alteração possível para gerar um erro sintático no arquivo micro10.lua é a omissão do caracter '*' na linha 5:

```
1. def fatorial(n):
2.      if(n <= 0):
3.          return 1
4.      else:
5.          return (n fatorial(n-1))
6.
7. end
```

A compilação do arquivo micro10.lua com a linha 4 alterada como visto no console anteriormente, gera a seguinte mensagem de erro sintático:

```
Erro sintático na linha 4, coluna 25 69 - Esperava: operador binario
   ou ')
'
.
- : Ast.program option option = None
```

Capítulo 6

Análise Semântica

Esta seção é destinada à criação de um análisador semântico da linguagem MiniPython.

6.1 Verificação de tipos

A listagem a seguir mostra um verificador de tipos para a linguagem MiniPython:

Legenda:

- Γ : ambiente
- $\Gamma + e : t$: no ambiente Γ, a expressão e possui tipo t.
- $\frac{\Gamma + a : t_1}{\Gamma + b : t_2}$: Dado que o tipo de a no ambiente Γ é t_1, infere-se que b no ambiente Γ terá o tipo t_2.

A seguir as regras:

6.1.1 Tipos:

- $\frac{\Gamma + a : t \quad \Gamma + b : t}{\Gamma + (a \quad op \quad b) : t}$, dado $t = \{\text{INT ou FLOAT}\}$ e $op = \{+, -, *, /, \%\}$
- $\frac{\Gamma + a : t \quad \Gamma + b : t}{\Gamma + (a \quad + \quad b) : t}$, dado $t = \{\text{INT, FLOAT ou STRING}\}$
- $\frac{\Gamma + a : INT \quad \Gamma + b : FLOAT}{\Gamma + (a \quad op \quad b) : FLOAT}$, dado $op = \{+, -, *, /, \%\}$
- $\frac{\Gamma + a : t \quad \Gamma + b : STRING}{\Gamma + (a \quad + \quad b) : STRING}$, dado $t = \{\text{INT ou FLOAT}\}$

6.1.2 Expressões:

- $\frac{\Gamma + a : BOOL \quad \Gamma + b : BOOL}{\Gamma + (a \quad LOGICO \quad b) : BOOL}$, dado $LOGICO = \{$AND, OR ou XOR$\}$

- $\frac{\Gamma + a : t \quad \Gamma + b : t}{\Gamma + (a \quad COMP \quad b) : BOOL}$, dado $t = \{== \text{ ou } !=\}$

6.1.3 Funções:

- $\frac{\Gamma + a1 : t1 \quad ...\Gamma + an : tn}{\Gamma + f(a1 \quad ... \quad an) : t}$, dado $f(t1, ..., tn) -> te\Gamma$

6.1.4 Comandos:

- $\frac{\Gamma + e : LOGICO \quad \Gamma + s : Valido}{\Gamma While(e) \quad s \quad Valido}$

- $\frac{\Gamma + e : LOGICO \quad \Gamma + s : Valido}{\Gamma IF(e) \quad s \quad Valido}$

- $\frac{\Gamma + s : Valido}{\Gamma Else(e) \quad s \quad Valido}$

- $\frac{\Gamma + e : EXPRESSO \quad \Gamma + s : Valido}{\Gamma Print(e) \quad s \quad Valido}$

6.2 Intérprete do código

Nesta parte seguem exemplos do intérprete criado pelo Menhir para interpretar o MiniPython, além de como utilizá-lo.

6.2.1 Descrição

Este intérprete foi criado na linguagem Ocaml, com o auxílio do Menhir. Ele é utilizado para a interpretação e análise semântica, utilizando o verificador de tipos mencionado na Seção 6.1.

6.2.2 Execução

Para que seja feita a execução do Intérprete do código, primeiramente deve-se compilar os arquivos utilizando o seguinte comando no terminal:

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

Depois de compilados, deve abrir o Ocaml:

```
> rlwrap ocaml
```

No Ocaml já é possível utilizar o interpretador, passando como parâmetro o nome do arquivo que se deseja ser interpretado:

```
# interprete "<NOME_AQUI.py>";;
```

Exemplos

Esta parte é destinada à exemplos de execução do interpretador dos códigos em MiniPython.

1. Supondo o arquivo em Python a seguir:

```
# interprete "algoritmos/a1.py"
```

Listagem 6.1:

```
1  def main() -> int:
2    numero = 0
3    x =0
4    print("Digite um numero: ")
5    inputi(numero)
6    numero = verifica(numero)
7
8    if numero == 1:
9      print("Positivo")
10   elif numero == 0:
11     print("zero")
12   else:
13     print("Negativo")
14
15
16   return 0
17
18
19 def verifica(n:int) -> int:
20   res = 0
21   if n > 0:
22     return 1
23   if n < 0:
24     return 3
25   else:
```

```
26      return 0
27
28 main()
```

O resultado do interpretador é dado abaixo, em 2 execuções distintas:

```
Digite um numero: 430
Positivo- : unit = ()
```

```
Digite um numero: -140
Negativo- : unit = ()
```

2. Supondo o arquivo em Python a seguir:

```
# interprete "algoritmos/a2.py"
```

Listagem 6.2:

```
1 def main() -> int:
2   numero = 0
3   x =0
4   print("Digite um numero: ")
5   inputi(numero)
6   numero = verifica(numero)
7
8   if numero == 1:
9     print("Positivo")
10  elif numero == 0:
11    print("zero")
12  else:
13    print("Negativo")
14
15
16  return 0
17
18
19 def verifica(n:int) -> int:
20  res = 0
21  if n > 0:
22    return 1
23  if n < 0:
24    return 3
25  else:
26    return 0
27
28 main()
```

O resultado do interpretador é dado abaixo, em 3 execuções distintas:

```
Digite um numero:  2
numero menor que 100- : unit = ()
```

```
Digite um numero:  100
numero entre 100 e 200- : unit = ()
```

```
Digite um numero:  1000
numero maior que 200- : unit = ()
```

3. Supondo o arquivo em Python a seguir:

```
# interprete "algoritmos/a7.py"
```

Listagem 6.3:

```
1 def main() -> None:
2   numero =0
3   fat = 0
4   print("Digite um numero: ")
5   inputi(numero)
6   fat = fatorial(numero)
7
8   print("O fatorial eh ")
9   print(fat)
10
11 def fatorial(n: int) -> int:
12   if n <= 0:
13     return 1
14   else:
15     return n * fatorial(n - 1)
16
17 main()
```

O resultado do interpretador é dado abaixo, em 2 execuções distintas:

```
Digite um numero: 15
O fatorial eh 1307674368000- : unit = ()
```

```
Digite um numero: 5
O fatorial eh 120- : unit = ()
```

Capítulo 7

Códigos

Este apêndice é dedicado à exposição de todos os códigos necessários para o funcionamento do compilador de MiniPython para a máquina virtual *Parrot Virtual Machine*.

- ambiente.ml

Listagem 7.1: ambiente.ml

```
1  module Tab = Tabsimb
2  module A = Ast
3
4  type entrada_fn = {
5    tipo_fn:  A.tipo;
6    formais: (string * A.tipo) list;
7  }
8
9  type entrada = EntFun of entrada_fn
10              |  EntVar of A.tipo
11
12 type t = {
13   ambv : entrada Tab.tabela
14 }
15
16 let novo_amb xs = { ambv = Tab.cria xs }
17
18 let novo_escopo amb = { ambv = Tab.novo_escopo amb.ambv }
19
20 let busca amb ch = Tab.busca amb.ambv ch
21
22 let insere_local amb ch t =
23   Tab.insere amb.ambv ch (EntVar t)
24
25 let insere_param amb ch t =
26   Tab.insere amb.ambv ch (EntVar t)
27
28 let insere_fun amb nome params resultado =
29   let ef = EntFun {
30     tipo_fn = resultado;
31     formais = params
32   } in Tab.insere amb.ambv nome ef
```

- ambiente.mli

Listagem 7.2: ambiente.mli

```
type entrada_fn = { tipo_fn:  Ast.tipo;
                    formais: (string * Ast.tipo) list;
                    (*      locais: (string * Asabs.tipo) list *)
}

type entrada =  EntFun of entrada_fn
             |  EntVar of Ast.tipo

type t

val novo_amb :  (string * entrada) list -> t
val novo_escopo : t -> t
val busca: t -> string -> entrada
val insere_local : t -> string -> Ast.tipo -> unit
val insere_param : t -> string -> Ast.tipo -> unit
val insere_fun : t -> string -> (string * Ast.tipo) list -> Ast.tipo
    -> unit
```

- ambInterp.ml

Listagem 7.3: ambInterp.ml

```
module Tab = Tabsimb
module A = Ast
module T = Tast

type entrada_fn = {
  tipo_fn:  A.tipo;
  formais: (A.identificador * A.tipo) list;
  corpo: T.expressao A.comandos
}

type entrada = EntFun of entrada_fn
             |  EntVar of A.tipo * (T.expressao option)


type t = {
  ambv : entrada Tab.tabela
}

let novo_amb xs = { ambv = Tab.cria xs }

let novo_escopo amb = { ambv = Tab.novo_escopo amb.ambv }

let busca amb ch = Tab.busca amb.ambv ch

let atualiza_var amb ch t v =
  Tab.atualiza amb.ambv ch (EntVar (t,v))

let insere_local amb nome t v =
  Tab.insere amb.ambv nome (EntVar (t,v))

let insere_param amb nome t v =
  Tab.insere amb.ambv nome (EntVar (t,v))

let insere_fun amb nome params resultado corpo =
```

```
35   let ef = EntFun { tipo_fn = resultado;
36                     formais = params;
37                     corpo = corpo }
38   in Tab.insere amb.ambv nome ef
```

- ast.ml

Listagem 7.4: ast.ml

```
1  (* The type of the abstract syntax tree (AST). *)
2  type identificador = string
3  (* posicao no arquivo *)
4  type 'a pos = 'a * Lexing.position
5
6  type 'expr programa  = Programa of 'expr instrucoes
7  and 'expr comandos   = 'expr comando list
8  and 'expr instrucoes = 'expr instrucao list
9  and 'expr expressoes = 'expr list
10 and 'expr instrucao  =
11     Funcao of 'expr decfn
12   | Cmd    of 'expr comando
13 and 'expr decfn = {
14   fn_nome:    identificador pos;
15   fn_tiporet: tipo;
16   fn_formais: (identificador pos * tipo) list;
17   fn_corpo:   'expr comandos
18 }
19 and tipo =
20     TipoInt
21   | TipoStr
22   | TipoBool
23   | TipoChar
24   | TipoFloat
25   | TipoNone
26 and 'expr comando =
27     CmdAtrib  of 'expr * 'expr
28   | CmdIf     of 'expr * 'expr comandos * ('expr comando option)
29   | CmdElse   of 'expr comandos
30   | CmdWhile  of 'expr * 'expr comandos
31   | CmdReturn of 'expr option
32   | CmdInputi of 'expr
33   | CmdInputf of 'expr
34   | CmdInputc of 'expr
35   | CmdInputs of 'expr
36   | CmdPrint  of 'expr
37   | CmdChmd   of 'expr
38
39 and operador =
40     Mais
41   | Menos
42   | Mul
43   | Div
44   | Mod
45   | Maior
46   | Menor
47   | MaiorIgual
48   | MenorIgual
49   | Igual
50   | Difer
```

```
51   | Elog
52   | Oulog
53   | Not
```

- interprete.ml

Listagem 7.5: interprete.ml

```
1  module Amb = AmbInterp
2  module A = Ast
3  module S = Sast
4  module T = Tast
5
6  exception Valor_de_retorno of T.expressao
7
8  let obtem_nome_tipo_var exp = let open T in
9    match exp with
10     | ExpVar (nome,tipo) -> (nome,tipo)
11     | _                  -> failwith "obtem_nome_tipo_var1: nao eh
         variavel"
12
13 let pega_int exp = match exp with
14   | T.ExpInt (i,_)  -> i
15   | _               -> failwith "pega_int: nao eh inteiro"
16
17 let pega_float exp = match exp with
18   | T.ExpFloat (f,_)-> f
19   | _               -> failwith "pega_float: nao eh inteiro"
20
21 let pega_char exp = match exp with
22   | T.ExpChar (c,_) -> c
23   | _               -> failwith "pega_char: nao eh inteiro"
24
25 let pega_str exp = match exp with
26   | T.ExpStr (s,_)  -> s
27   | _               -> failwith "pega_string: nao eh string"
28
29 let pega_bool exp = match exp with
30   | T.ExpBool (b,_) -> b
31   | _               -> failwith "pega_bool: nao eh booleano"
32
33 type classe_op = Aritmetico | Relacional | Logico
34
35 let classifica op = let open A in
36   match op with
37   | Mais
38   | Menos
39   | Mul
40   | Div
41   | Mod       -> Aritmetico
42   | Maior
43   | Menor
44   | MaiorIgual
45   | MenorIgual
46   | Igual
47   | Difer     ->Relacional
48   | Elog
49   | Oulog
50   | Not       -> Logico
```

```
51
52 let rec interpreta_exp amb exp =
53   let open A in
54   let open T in
55     match exp with
56     | ExpInt   _
57     | ExpStr   _
58     | ExpChar  _
59     | ExpBool  _
60     | ExpFloat _ -> exp
61     | ExpVar (nome, tipo) ->
62       (match (Amb.busca amb nome) with
63         | Amb.EntVar (_, v) ->
64           (match v with
65             | Some valor -> valor
66             | None       -> failwith "variável nao inicializada: "
67           )
68         |  _ -> failwith "interpreta_exp: expvar"
69       )
70     | ExpOperB ((op, top), (esq, tesq), (dir, tdir)) ->
71       let vesq = interpreta_exp amb esq
72       and vdir = interpreta_exp amb dir in
73       let interpreta_aritmetico () =
74         (match top with
75           | TipoInt ->
76             (match op with
77               | Mais  -> ExpInt (pega_int vesq +   pega_int vdir, top
                  )
78               | Menos -> ExpInt (pega_int vesq -   pega_int vdir, top
                  )
79               | Mul   -> ExpInt (pega_int vesq *   pega_int vdir, top
                  )
80               | Div   -> ExpInt (pega_int vesq /   pega_int vdir, top
                  )
81               | Mod   -> ExpInt (pega_int vesq mod pega_int vdir, top
                  )
82               | _     -> failwith "interpreta_aritmetico"
83             )
84           | TipoFloat ->
85             (match op with
86               | Mais  -> ExpFloat (pega_float vesq +. pega_float vdir
                  , top)
87               | Menos -> ExpFloat (pega_float vesq -. pega_float vdir
                  , top)
88               | Mul   -> ExpFloat (pega_float vesq *. pega_float vdir
                  , top)
89               | Div   -> ExpFloat (pega_float vesq /. pega_float vdir
                  , top)
90               | _     -> failwith "interpreta_aritmetico"
91             )
92           | _ -> failwith "interpreta_aritmetico"
93         )
94       and interpreta_relacional () =
95         (match tesq with
96           | TipoInt ->
97             (match op with
98               | Menor      -> ExpBool (pega_int vesq <  pega_int vdir
                  , top)
99               | Maior      -> ExpBool (pega_int vesq >  pega_int vdir
```

```
                , top)
100           | MaiorIgual -> ExpBool (pega_int vesq >= pega_int vdir
                , top)
101           | MenorIgual -> ExpBool (pega_int vesq <= pega_int vdir
                , top)
102           | Igual      -> ExpBool (pega_int vesq == pega_int vdir
                , top)
103           | Difer      -> ExpBool (pega_int vesq != pega_int vdir
                , top)
104           | _          -> failwith "interpreta_relacional"
105         )
106       | TipoStr ->
107         (match op with
108           | Menor      -> ExpBool (pega_str vesq <  pega_str vdir
                , top)
109           | Maior      -> ExpBool (pega_str vesq >  pega_str vdir
                , top)
110           | MaiorIgual -> ExpBool (pega_str vesq >= pega_str vdir
                , top)
111           | MenorIgual -> ExpBool (pega_str vesq <= pega_str vdir
                , top)
112           | Igual      -> ExpBool (pega_str vesq == pega_str vdir
                , top)
113           | Difer      -> ExpBool (pega_str vesq != pega_str vdir
                , top)
114           | _          -> failwith "interpreta_relacional"
115         )
116       | TipoChar ->
117         (match op with
118           | Menor      -> ExpBool (pega_char vesq <  pega_char
                vdir, top)
119           | Maior      -> ExpBool (pega_char vesq >  pega_char
                vdir, top)
120           | MaiorIgual -> ExpBool (pega_char vesq >= pega_char
                vdir, top)
121           | MenorIgual -> ExpBool (pega_char vesq <= pega_char
                vdir, top)
122           | Igual      -> ExpBool (pega_char vesq == pega_char
                vdir, top)
123           | Difer      -> ExpBool (pega_char vesq != pega_char
                vdir, top)
124           | _          -> failwith "interpreta_relacional"
125         )
126       | TipoBool ->
127         (match op with
128           | Menor      -> ExpBool (pega_bool vesq <  pega_bool
                vdir, top)
129           | Maior      -> ExpBool (pega_bool vesq >  pega_bool
                vdir, top)
130           | MaiorIgual -> ExpBool (pega_bool vesq >= pega_bool
                vdir, top)
131           | MenorIgual -> ExpBool (pega_bool vesq <= pega_bool
                vdir, top)
132           | Igual      -> ExpBool (pega_bool vesq == pega_bool
                vdir, top)
133           | Difer      -> ExpBool (pega_bool vesq != pega_bool
                vdir, top)
134           | _          -> failwith "interpreta_relacional"
135         )
```

```
136           | TipoFloat ->
137             (match op with
138               | Menor      -> ExpBool (pega_float vesq <  pega_float
                     vdir, top)
139               | Maior      -> ExpBool (pega_float vesq >  pega_float
                     vdir, top)
140               | MaiorIgual -> ExpBool (pega_float vesq == pega_float
                     vdir, top)
141               | MenorIgual -> ExpBool (pega_float vesq == pega_float
                     vdir, top)
142               | Igual      -> ExpBool (pega_float vesq == pega_float
                     vdir, top)
143               | Difer      -> ExpBool (pega_float vesq != pega_float
                     vdir, top)
144               | _          -> failwith "interpreta_relacional"
145             )
146           | _ ->  failwith "interpreta_relacional"
147         )
148       and interpreta_logico () =
149         (match tesq with
150           | TipoBool  ->
151             (match op with
152               | Oulog -> ExpBool (pega_bool vesq || pega_bool vdir,
                     top)
153               | Elog  -> ExpBool (pega_bool vesq && pega_bool vdir,
                     top)
154               | _     ->  failwith "interpreta_logico"
155             )
156           | _         ->  failwith "interpreta_logico"
157         )
158       in
159       let valor =
160         (match (classifica op) with
161           | Aritmetico -> interpreta_aritmetico ()
162           | Relacional -> interpreta_relacional ()
163           | Logico     -> interpreta_logico ()
164         )
165       in valor
166     | ExpOperU ((op, top), (exp, texp)) ->
167       let vexp = interpreta_exp amb exp in
168       let interpreta_not () =
169        (match texp with
170         | A.TipoBool -> ExpBool (not (pega_bool vexp), top)
171         | _          -> failwith "Operador unario indefinido")
172       and interpreta_negativo () =
173        (match texp with
174         | A.TipoInt   -> ExpInt   (-1   *  pega_int   vexp, top)
175         | A.TipoFloat -> ExpFloat (-1.0 *. pega_float vexp, top)
176         | _           -> failwith "Operador unario indefinido")
177       in
178       let valor =
179        (match op with
180           | Not   -> interpreta_not ()
181           | Menos -> interpreta_negativo ()
182           | _     -> failwith "Operador unario indefinido")
183       in  valor
184     | ExpChmd (id, args, tipo) ->
185       let open Amb in
186         (match (Amb.busca amb id) with
```

```
187           | Amb.EntFun {tipo_fn; formais; corpo} ->
188             let vargs    = List.map  (interpreta_exp amb) args in
189             let vformais = List.map2 (fun (n,t) v -> (n, t, Some v))
                    formais vargs
190             in  interpreta_fun amb vformais corpo
191           | _ -> failwith "interpreta_exp: expchamada"
192         )
193     | ExpNone -> T.ExpNone
194 and interpreta_cmd amb cmd =
195   let open A in
196   let open T in
197   match cmd with
198     CmdReturn exp ->
199     (* Levantar uma exceção foi necessária pois, pela semântica do
           comando de   *)
200     (* retorno, sempre que ele for encontrado em uma função, a
           computação       *)
201     (* deve parar retornando o valor indicado, sem realizar os demais
            comandos. *)
202     (match exp with
203       (* Se a função não retornar nada, então retorne ExpVoid *)
204       | None -> raise (Valor_de_retorno ExpNone)
205       | Some e ->
206         (* Avalia a expressão e retorne o resultado *)
207         let e1 = interpreta_exp amb e in
208         raise (Valor_de_retorno e1))
209   | CmdIf (teste, entao, senao) ->
210       let teste1 = interpreta_exp amb teste in
211       (match teste1 with
212         | ExpBool (true,_) ->
213         (* Interpreta cada comando do bloco 'então' *)
214         List.iter (interpreta_cmd amb) entao
215         | _ ->
216           (* Interpreta cada comando do bloco 'senão', se houver *)
217           (match senao with
218             | None -> ()
219             | Some bloco -> interpreta_cmd amb bloco))
220   | CmdElse comandos ->
221         List.iter (interpreta_cmd amb ) comandos
222   | CmdAtrib (elem, exp) ->
223       let resp = interpreta_exp amb exp in
224         (match elem with
225           | T.ExpVar (id,tipo) ->
226            (try
227               begin
228                 match (Amb.busca amb id) with
229                   | Amb.EntVar (t, _) -> Amb.atualiza_var amb id tipo
                          (Some resp)
230                   | Amb.EntFun _      -> failwith "falha na
                        atribuicao"
231               end
232             with Not_found ->
233               let _ = Amb.insere_local amb id tipo None in
234               Amb.atualiza_var amb id tipo (Some resp))
235           | _ -> failwith "Falha CmdAtrib"
236         )
237   | CmdChmd   exp -> ignore( interpreta_exp amb exp )
238   | CmdInputi exp
239   | CmdInputf exp
```

```
240   | CmdInputc exp
241   | CmdInputs exp ->
242     (* Obtem os nomes e os tipos de cada um dos argumentos *)
243     let nt = obtem_nome_tipo_var exp in
244     let leia_var (nome,tipo) =
245      let _ =
246        (try
247           begin
248             match (Amb.busca amb nome) with
249               | Amb.EntVar (_,_) -> ()
250               | Amb.EntFun _     -> failwith "falha no input"
251           end
252         with Not_found ->
253           let _ = Amb.insere_local amb nome tipo None in ()
254         )
255       in
256       let valor =
257        (match tipo with
258           | TipoInt   -> T.ExpInt   (read_int   ()   , tipo)
259           | TipoStr   -> T.ExpStr   (read_line  ()   , tipo)
260           | TipoChar  -> T.ExpChar  (input_char stdin, tipo)
261           | TipoFloat -> T.ExpFloat (read_float ()   , tipo)
262           | _         -> failwith "Fail input")
263       in  Amb.atualiza_var amb nome tipo (Some valor)
264     in leia_var nt
265   | CmdPrint exp ->
266     let resp = interpreta_exp amb exp in
267       (match resp with
268         | T.ExpInt   (n,_) -> print_int    n
269         | T.ExpFloat (n,_) -> print_float  n
270         | T.ExpStr   (n,_) -> print_string n
271         | T.ExpChar  (n,_) -> print_char   n
272         | _ -> failwith "Fail print"
273       )
274   | CmdWhile (cond, cmds) ->
275         let rec laco cond cmds =
276           let condResp = interpreta_exp amb cond in
277                 (match condResp with
278                   | ExpBool (true,_) ->
279                       (* Interpreta cada comando do bloco 'então' *)
280                       let _ = List.iter (interpreta_cmd amb) cmds in
281                         laco cond cmds
282                   | _ -> ())
283         in laco cond cmds
284
285 and interpreta_fun amb fn_formais fn_corpo =
286   let open A in
287  (* Estende o ambiente global, adicionando um ambiente local *)
288   let ambfn = Amb.novo_escopo amb in
289   (* Associa os argumento
290   s aos parâmetros e insere no novo ambiente *)
291   let insere_parametro (n,t,v) = Amb.insere_param ambfn n t v in
292   let _ = List.iter insere_parametro fn_formais in
293       (* Interpreta cada comando presente no corpo da função usando o
              novo *)
294       (* ambiente
                                                                          *)
295       try
296         let _ = List.iter (interpreta_cmd ambfn) fn_corpo in T.
```

```
            ExpNone
297       with
298         Valor_de_retorno expret -> expret
299
300 let insere_declaracao_fun amb dec =
301   let open A in
302     match dec with
303       | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
304         let nome = fst fn_nome in
305         let formais = List.map (fun (n,t) -> ((fst n), t)) fn_formais
                in
306         Amb.insere_fun amb nome formais fn_tiporet fn_corpo
307       | _ -> failwith "Erro de declaacao de funcao"
308
309 (* Lista de cabeçalhos das funções pré definidas *)
310 let fn_predefs = let open A in [
311     ("inputi", [("x", TipoInt  )], TipoNone, []);
312     ("inputf", [("x", TipoFloat)], TipoNone, []);
313     ("inputc", [("x", TipoChar )], TipoNone, []);
314     ("inputs", [("x", TipoStr  )], TipoNone, []);
315     ("printi", [("x", TipoInt  )], TipoNone, []);
316     ("printf", [("x", TipoFloat)], TipoNone, []);
317     ("printc", [("x", TipoChar )], TipoNone, []);
318     ("prints", [("x", TipoStr  )], TipoNone, []);
319 ]
320
321 (* insere as funções pré definidas no ambiente global *)
322 let declara_predefinidas amb =
323   List.iter (fun (n,ps,tr,c) -> Amb.insere_fun amb n ps tr c)
       fn_predefs
324
325 let interprete ast =
326   let open Amb in
327   let amb_global = Amb.novo_amb [] in
328   let _ = declara_predefinidas amb_global in
329   let A.Programa instr = ast in
330     let decs_funs = List.filter (fun x ->
331     (match x with
332     | A.Funcao _ -> true
333     | _          -> false)) instr in
334     let _ = List.iter (insere_declaracao_fun amb_global) decs_funs in
335       (try begin
336         (match (Amb.busca amb_global "main") with
337             | Amb.EntFun { tipo_fn ; formais ; corpo } ->
338               let vformais = List.map (fun (n,t) -> (n, t, None))
                      formais in
339               let _        = interpreta_fun amb_global vformais corpo
                    in ()
340             | _ -> failwith "variavel declarada como 'main'")
341       end with Not_found -> failwith "Funcao main nao declarada ")
```

- interprete.mli

Listagem 7.6: interprete.mli

```
1 val interprete :  Tast.expressao Ast.programa -> unit
```

- lexico.mll

Listagem 7.7: lexico.mll

```
{
    open Sintatico
    open Lexing
    open Printf

    exception Erro of string

    let nivel_par = ref 0

    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
*)
    let pos_atual lexbuf = lexbuf.lex_start_p

}

let digito  = ['0' - '9']
let inteiro = digito+
let frac    = '.'digito*
let exp     = ['e' 'E']['-' '+']?digito+
let float   = digito* frac exp?
let restante   = [^ ' ' '\t' '\n' ] [^ '\n']+
let brancos    = [' ' '\t']+
let novalinha  = '\r' | '\n' | "\r\n"
let comentario = "#"[ ^ '\n' ]*
let linha_em_branco = [' ' '\t' ]* comentario
let letra = [ 'a'-'z' 'A' - 'Z']
let identificador = letra(letra|digito|'_')*

rule preprocessador indentacao = parse
  linha_em_branco         { preprocessador 0 lexbuf }
| [' ' '\t' ]+ '\n'       { incr_num_linha lexbuf; preprocessador 0
    lexbuf }
| ' '                     { preprocessador (indentacao + 1) lexbuf }
| '\t'                    { let nova_ind = indentacao + 8 - (
    indentacao mod 8) in preprocessador nova_ind lexbuf }
| novalinha               { incr_num_linha lexbuf; preprocessador 0
    lexbuf }
| restante as linha {
                          let rec tokenize lexbuf =
                              let tok = token lexbuf in
                               match tok with
                                EOF -> []
                               | _  -> tok :: tokenize lexbuf in
                                      let toks = tokenize (Lexing.
                                          from_string linha) in
                                           Linha(indentacao,!nivel_par,
                                               toks)
```

```
54                     }
55 | eof { nivel_par := 0; EOF }
56
57 and token = parse
58   brancos             { token lexbuf }
59 | comentario          { token lexbuf }
60 | "\"\"\""            { comentario_bloco 0 lexbuf }
61 | '('                 { incr(nivel_par); APAR (pos_atual lexbuf)}
62 | ')'                 { decr(nivel_par); FPAR (pos_atual lexbuf)}
63 | "->"                { SETA       (pos_atual lexbuf)}
64 | "+="                { SOMAATRIB  (pos_atual lexbuf)}
65 | "-="                { SUBATRIB   (pos_atual lexbuf)}
66 | "*="                { MULATRIB   (pos_atual lexbuf)}
67 | "/="                { DIVATRIB   (pos_atual lexbuf)}
68 | "%="                { MODATRIB   (pos_atual lexbuf)}
69 | "<="                { MENORIGUAL (pos_atual lexbuf)}
70 | ">="                { MAIORIGUAL (pos_atual lexbuf)}
71 | "=="                { IGUAL      (pos_atual lexbuf)}
72 | "!="                { DIFERENTE  (pos_atual lexbuf)}
73 | ','                 { VIRG   (pos_atual lexbuf)}
74 | ':'                 { DPTOS  (pos_atual lexbuf)}
75 | '+'                 { MAIS   (pos_atual lexbuf)}
76 | '-'                 { MENOS  (pos_atual lexbuf)}
77 | '*'                 { MUL    (pos_atual lexbuf)}
78 | '/'                 { DIV    (pos_atual lexbuf)}
79 | '%'                 { MOD    (pos_atual lexbuf)}
80 | '<'                 { MENOR  (pos_atual lexbuf)}
81 | '>'                 { MAIOR  (pos_atual lexbuf)}
82 | '='                 { ATRIB  (pos_atual lexbuf)}
83 | "not"               { NAO    (pos_atual lexbuf)}
84 | "and"               { ELOG   (pos_atual lexbuf)}
85 | "or"                { OULOG  (pos_atual lexbuf)}
86 | "def"               { DEF    (pos_atual lexbuf)}
87 | "return"            { RETURN (pos_atual lexbuf)}
88 | "while"             { WHILE  (pos_atual lexbuf)}
89 | "for"               { FOR    (pos_atual lexbuf)}
90 | "in"                { IN     (pos_atual lexbuf)}
91 | "range"             { RANGE  (pos_atual lexbuf)}
92 | "inputi"            { INPUTI (pos_atual lexbuf)}
93 | "inputf"            { INPUTF (pos_atual lexbuf)}
94 | "inputc"            { INPUTC (pos_atual lexbuf)}
95 | "inputs"            { INPUTS (pos_atual lexbuf)}
96 | "print"             { PRINT  (pos_atual lexbuf)}
97 | "str"               { STRING (pos_atual lexbuf)}
98 | "int"               { I32    (pos_atual lexbuf)}
99 | "bool"              { BOOL   (pos_atual lexbuf)}
100 | "char"             { CHAR   (pos_atual lexbuf)}
101 | "float"            { F32    (pos_atual lexbuf)}
102 | "None"             { NONE   (pos_atual lexbuf)}
103 | "if"               { IF     (pos_atual lexbuf)}
104 | "elif"             { ELIF   (pos_atual lexbuf)}
105 | "else"             { ELSE   (pos_atual lexbuf)}
106 | "True"             { LITBOOL(true,pos_atual lexbuf)}
107 | "False"            { LITBOOL(false,pos_atual lexbuf)}
108 | "'"_"'" as s       { let c = String.get s 1 in LITCHAR (c,pos_atual
      lexbuf) }
109 | inteiro as num     { let numero = int_of_string num   in LITINT   (
     numero,pos_atual lexbuf) }
110 | float as num       { let numero = float_of_string num in LITFLOAT (
```

```
      numero,pos_atual lexbuf) }
111 | '"'                 { let buffer = Buffer.create 1 in
112                         let str = leia_string buffer lexbuf in
113                             LITSTRING (str, pos_atual lexbuf) }
114 | identificador as id{ ID (id, pos_atual lexbuf) }
115 | _                   { raise ( Erro ("Caracter desconhecido: " ^
      Lexing.lexeme lexbuf)) }
116 | eof                 { EOF     }
117
118 and comentario_bloco n = parse
119     "\"\"\""{ token lexbuf                      }
120     | _     { comentario_bloco n lexbuf         }
121     | eof   { raise (Erro "Comentário não terminado") }
122 and leia_string buffer = parse
123     | '"'       { Buffer.contents buffer }
124     | "\\t"     { Buffer.add_char buffer '\t'; leia_string buffer
          lexbuf }
125     | "\\n"     { Buffer.add_char buffer '\n'; leia_string buffer
          lexbuf }
126     | '\\' '"'  { Buffer.add_char buffer '"'; leia_string buffer
          lexbuf  }
127     | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string buffer
          lexbuf }
128     | _ as c    { Buffer.add_char buffer c; leia_string buffer lexbuf
              }
129     | eof       { raise (Erro "A string não foi fechada.") }
```

- pre_processador.ml

Listagem 7.8: pre_processador.ml

```
1  open Sintatico
2  open Lexico
3  open Printf
4
5  (* Pre processa o arquivo gerando os tokens de indenta e dedenta *)
6  let preprocessa lexbuf =
7    let pilha = Stack.create ()
8    and npar = ref 0 in
9      let _ = Stack.push 0 pilha in
10     let off_side toks nivel =
11     if !npar != 0 (* nova linha entre parenteses *)
12     then toks     (* nao faz nada *)
13     else if nivel > Stack.top pilha
14          then begin
15            Stack.push nivel pilha;
16            INDENTA :: toks
17          end
18     else if nivel = Stack.top pilha
19          then toks
20     else begin
21     let prefixo = ref toks in
22     while nivel < Stack.top pilha do
23       ignore (Stack.pop pilha);
24       if nivel > Stack.top pilha
25         then failwith "Erro de indentacao"
26       else prefixo := DEDENTA :: !prefixo
27    done;
28    !prefixo
```

```
29    end
30  in
31
32  let rec dedenta sufixo =
33    if Stack.top pilha != 0
34    then let _ = Stack.pop pilha in
35         dedenta (DEDENTA :: sufixo)
36    else sufixo
37  in
38  let rec get_tokens () =
39    let tok = Lexico.preprocessador 0 lexbuf in
40    match tok with
41      Linha(nivel,npars,toks) ->
42      let new_toks = off_side toks nivel in
43      npar := npars;
44      new_toks @ (if npars = 0
45                  then NOVALINHA :: get_tokens ()
46                  else get_tokens ())
47    | _ -> dedenta []
48  in get_tokens ()
49
50
51 (* Chama o analisador lexico *)
52 let lexico =
53   let tokbuf = ref None in
54   let carrega lexbuf =
55     let toks = preprocessa lexbuf in
56     (match toks with
57        tok::toks ->
58        tokbuf := Some toks;
59        tok
60      | [] -> EOF)
61   in
62   fun lexbuf ->
63   match !tokbuf with
64     Some tokens ->
65     (match tokens with
66        tok::toks ->
67        tokbuf := Some toks;
68        tok
69      | [] -> carrega lexbuf)
70   | None -> carrega lexbuf
```

- sast.ml

Listagem 7.9: sast.ml

```
1  open Ast
2
3  type expressao =
4      ExpVar   of identificador pos
5    | ExpInt   of int pos
6    | ExpStr   of string pos
7    | ExpChar  of char pos
8    | ExpBool  of bool pos
9    | ExpFloat of float pos
10   | ExpOperB of operador pos * expressao * expressao
11   | ExpOperU of operador pos * expressao
12   | ExpChmd  of identificador pos * (expressao expressoes)
```

- semantico.ml

Listagem 7.10: semantico.ml

```
1  module Amb = Ambiente
2  module A = Ast
3  module S = Sast
4  module T = Tast
5
6  let rec posicao exp =
7    let open S in
8    match exp with
9    | ExpVar       (_,pos)       -> pos
10   | ExpInt       (_,pos)       -> pos
11   | ExpStr       (_,pos)       -> pos
12   | ExpBool      (_,pos)       -> pos
13   | ExpChar      (_,pos)       -> pos
14   | ExpFloat     (_,pos)       -> pos
15   | ExpOperB    ((_,pos),_,_) -> pos
16   | ExpOperU    ((_,pos),_)   -> pos
17   | ExpChmd     ((_,pos),_)   -> pos
18
19 type classe_op = Aritmetico | Relacional | Logico
20
21 let classifica op =
22   let open A in
23     match op with
24       Mais
25     | Menos
26     | Mul
27     | Div
28     | Mod       -> Aritmetico
29     | Maior
30     | Menor
31     | MaiorIgual
32     | MenorIgual
33     | Igual
34     | Difer     -> Relacional
35     | Elog
36     | Oulog
37     | Not       -> Logico
38
39 let msg_erro_pos pos msg =
40   let open Lexing in
41   let lin = pos.pos_lnum
42   and col = pos.pos_cnum - pos.pos_bol - 1 in
43   Printf.sprintf "Semantico -> linha %d, coluna %d: %s" lin col msg
44
45 (*  argumento nome é do tipo S.tipo  *)
46 let msg_erro nome msg =
47   let pos = snd nome in
48   msg_erro_pos pos msg
49
50 let nome_tipo t =
51   let open A in
52     match t with
53       TipoInt     -> "inteiro"
54     | TipoStr     -> "string"
55     | TipoBool    -> "booleano"
56     | TipoChar    -> "caracter"
```

```
57      | TipoFloat   -> "real"
58      | TipoNone    -> "vazio"
59
60  let mesmo_tipo pos msg tinf tdec =
61    if tinf <> tdec then
62      let msg = Printf.sprintf msg (nome_tipo tinf) (nome_tipo tdec) in
63      failwith (msg_erro_pos pos msg)
64
65  let rec infere_exp amb exp =
66    match exp with
67    | S.ExpInt   i -> (T.ExpInt   (fst i, A.TipoInt  ), A.TipoInt  )
68    | S.ExpStr   s -> (T.ExpStr   (fst s, A.TipoStr  ), A.TipoStr  )
69    | S.ExpBool  b -> (T.ExpBool  (fst b, A.TipoBool ), A.TipoBool )
70    | S.ExpChar  c -> (T.ExpChar  (fst c, A.TipoChar ), A.TipoChar )
71    | S.ExpFloat f -> (T.ExpFloat (fst f, A.TipoFloat), A.TipoFloat)
72    | S.ExpVar variavel ->
73        let nome = fst variavel in
74          (try begin
75              (match (Amb.busca amb nome) with
76              | Amb.EntVar tipo -> (T.ExpVar (nome, tipo), tipo)
77              | Amb.EntFun _    ->
78                  let msg = "Nome de funcao usado como nome de variavel
                        : "^nome in
79                    failwith (msg_erro variavel msg))
80          end with Not_found ->
81            let msg = "Variavel "^nome^" nao declarada" in
82              failwith (msg_erro variavel msg))
83    | S.ExpOperB (op, exp_esq, exp_dir) ->
84        let (esq, tesq) = infere_exp amb exp_esq
85        and (dir, tdir) = infere_exp amb exp_dir in
86        let verifica_aritmetico () =
87          (match tesq with
88          | A.TipoInt
89          | A.TipoFloat ->
90              let _ = mesmo_tipo (snd op)
91                      "Operando esquerdo do tipo %s, mas o tipo do
                           direito eh %s"
92                      tesq tdir
93              in tesq (* Tipo inferido para a operação *)
94          | demais      ->
95              let msg = "O tipo "^
96                        (nome_tipo demais)^
97                        " nao eh valido em um operador aritmético" in
98                failwith (msg_erro op msg))
99        and verifica_relacional () =
100         (match tesq with
101         | A.TipoInt
102         | A.TipoStr
103         | A.TipoBool
104         | A.TipoChar
105         | A.TipoFloat ->
106             (let _ = mesmo_tipo (snd op)
107                     "Operando esquerdo do tipo %s, mas o tipo do
                          direito eh %s"
108                     tesq tdir
109             in A.TipoBool) (* Tipo inferido para a operação *)
110         | demais      ->
111             (let msg = "O tipo "^
112                       (nome_tipo demais)^
```

```
113                     " nao eh valido em um operador relacional" in
114             failwith (msg_erro op msg)))
115     and verifica_logico () =
116       (match tesq with
117       | A.TipoBool ->
118           let _ = mesmo_tipo (snd op)
119                   "Operando esquerdo do tipo %s, mas o tipo do
                          direito eh %s"
120                   tesq tdir
121           in A.TipoBool (* Tipo inferido para a operação *)
122       | demais ->
123           let msg = "O tipo "^
124                     (nome_tipo demais)^
125                     " nao eh valido em um operador logico" in
126             failwith (msg_erro op msg))
127     in
128     let oper = fst op in
129     let tinf =
130        (match (classifica oper) with
131       | Aritmetico -> verifica_aritmetico ()
132       | Relacional -> verifica_relacional ()
133       | Logico     -> verifica_logico () )
134     in (T.ExpOperB ((oper, tinf), (esq, tesq), (dir, tdir)), tinf)
135 | S.ExpOperU (op, exp) ->
136   let (exp, texp) = infere_exp amb exp in
137   let verifica_not () =
138     match texp with
139     | A.TipoBool ->
140         let _ = mesmo_tipo (snd op)
141                 "O operando eh do tipo %s, mas espera-se um %s"
142                 texp A.TipoBool
143         in A.TipoBool
144     | demais     ->
145         let msg = "O tipo "^
146                   (nome_tipo demais)^
147                   " nao eh valido para o operador not" in
148           failwith (msg_erro op msg)
149   and verifica_negativo () =
150     match texp with
151     | A.TipoFloat ->
152         let _ = mesmo_tipo (snd op)
153                 "O operando eh do tipo %s, mas espera-se um %s"
154                 texp A.TipoFloat
155         in A.TipoFloat
156     | A.TipoInt ->
157         let _ = mesmo_tipo (snd op)
158                 "O operando eh do tipo %s, mas espera-se um %s"
159                 texp A.TipoInt
160         in A.TipoInt
161     | demais     ->
162         let msg = "O tipo "^
163                   (nome_tipo demais)^
164                   " nao eh valido para o operador menos" in
165           failwith (msg_erro op msg)
166   in
167   let oper = fst op in
168   let tinf =
169     let open A in
170       match oper with
```

```
171         | Not   -> verifica_not ()
172         | Menos -> verifica_negativo ()
173         | demais->
174             let msg = "Operador unario indefinido"
175             in failwith (msg_erro op msg)
176     in  (T.ExpOperU ((oper, tinf), (exp, texp)), tinf)
177   | S.ExpChmd (nome, args) ->
178     let rec verifica_parametros ags ps fs =
179       match (ags, ps, fs) with
180       | (a::ags), (p::ps), (f::fs) ->
181           let _ = mesmo_tipo (posicao a)
182                   "O parametro eh do tipo %s mas deveria ser do tipo
                        %s"
183                   p f
184           in verifica_parametros ags ps fs
185       | [], [], [] -> ()
186       | _ -> failwith (msg_erro nome "Numero incorreto de parametros"
              )
187     in
188     let id = fst nome in
189       try
190         begin
191           let open Amb in
192             match (Amb.busca amb id) with
193             | Amb.EntFun {tipo_fn; formais} ->
194               let targs    = List.map (infere_exp amb) args
195               and tformais = List.map snd formais in
196               let _ = verifica_parametros args (List.map snd targs)
                      tformais in
197                 (T.ExpChmd (id, (List.map fst targs), tipo_fn),
                        tipo_fn)
198             | Amb.EntVar _ -> (* Se estiver associada a uma variável,
                      falhe *)
199               let msg = id ^ " eh uma variavel e nao uma funcao" in
200                 failwith (msg_erro nome msg)
201         end
202       with Not_found ->
203         let msg = "Nao existe a funcao de nome " ^ id in
204         failwith (msg_erro nome msg)
205
206 let rec verifica_cmd amb tiporet cmd =
207   let open A in
208     match cmd with
209     | CmdReturn exp ->
210       (match exp with
211       | None ->
212           let _ = mesmo_tipo (Lexing.dummy_pos)
213             "O tipo retornado eh %s mas foi declarado como %s"
214             TipoNone tiporet
215           in CmdReturn None
216       | Some exp ->
217           let (e1,tinf) = infere_exp amb exp in
218           let _ = mesmo_tipo (posicao exp)
219             "O tipo retornado eh %s mas foi declarado como %s"
220             tinf tiporet
221           in CmdReturn (Some e1))
222     | CmdChmd  exp -> let (exp,tinf) = infere_exp amb exp in CmdChmd
          exp
223     | CmdInputi exp ->
```

```
224         (match exp with
225           S.ExpVar (id,pos) ->
226            (try
227               begin
228                 (match (Amb.busca amb id) with
229                     Amb.EntVar tipo ->
230                       let expt = infere_exp amb exp in
231                       let _ = mesmo_tipo pos
232                         "inputi com tipos diferentes: %s = %s"
233                         tipo (snd expt) in
234                         CmdInputi (fst expt)
235                   | Amb.EntFun _ ->
236                       let msg = "nome de funcao usado como nome de
                              variavel: " ^ id in
237                       failwith (msg_erro_pos pos msg) )
238               end
239             with Not_found ->
240               let _ = Amb.insere_local amb id A.TipoInt in
241               let expt = infere_exp amb exp in
242               CmdInputi (fst expt) )
243           | _ -> failwith "Falha Inputi"
244         )
245     | CmdInputf exp ->
246         (match exp with
247           S.ExpVar (id,pos) ->
248            (try
249               begin
250                 (match (Amb.busca amb id) with
251                     Amb.EntVar tipo ->
252                       let expt = infere_exp amb exp in
253                       let _ = mesmo_tipo pos
254                         "Inputf com tipos diferentes: %s = %s"
255                         tipo (snd expt) in
256                         CmdInputf (fst expt)
257                   | Amb.EntFun _ ->
258                       let msg = "nome de funcao usado como nome de
                              variavel: " ^ id in
259                       failwith (msg_erro_pos pos msg) )
260               end
261             with Not_found ->
262               let _ = Amb.insere_local amb id A.TipoFloat in
263               let expt = infere_exp amb exp in
264               CmdInputf (fst expt) )
265           | _ -> failwith "Falha Inputf"
266         )
267     | CmdInputs exp ->
268         (match exp with
269           S.ExpVar (id,pos) ->
270            (try
271               begin
272                 (match (Amb.busca amb id) with
273                     Amb.EntVar tipo ->
274                       let expt = infere_exp amb exp in
275                       let _ = mesmo_tipo pos
276                         "Inputs com tipos diferentes: %s = %s"
277                         tipo (snd expt) in
278                         CmdInputs (fst expt)
279                   | Amb.EntFun _ ->
280                       let msg = "nome de funcao usado como nome de
```

```
                        variavel: " ^ id in
281                   failwith (msg_erro_pos pos msg) )
282             end
283           with Not_found ->
284             let _ = Amb.insere_local amb id A.TipoStr in
285             let expt = infere_exp amb exp in
286             CmdInputs (fst expt) )
287       | _ -> failwith "Falha Inputs"
288     )
289   | CmdInputc exp ->
290     (match exp with
291       S.ExpVar (id,pos) ->
292        (try
293           begin
294             (match (Amb.busca amb id) with
295                 Amb.EntVar tipo ->
296                   let expt = infere_exp amb exp in
297                   let _ = mesmo_tipo pos
298                     "Input com tipos diferentes: %s = %s"
299                     tipo (snd expt) in
300                     CmdInputc (fst expt)
301               | Amb.EntFun _ ->
302                   let msg = "nome de funcao usado como nome de
                          variavel: " ^ id in
303                   failwith (msg_erro_pos pos msg) )
304             end
305           with Not_found ->
306             let _ = Amb.insere_local amb id A.TipoChar in
307             let expt = infere_exp amb exp in
308             CmdInputc (fst expt) )
309       | _ -> failwith "Falha InputChar"
310     )
311   | CmdPrint exp -> let expt = infere_exp amb exp in CmdPrint (fst
        expt)
312   | CmdWhile (cond, cmds) ->
313     let (expCond, expT ) = infere_exp amb cond in
314     let comandos_tipados =
315       (match expT with
316         | A.TipoBool -> List.map (verifica_cmd amb tiporet) cmds
317         | _ -> let msg = "Condicao deve ser tipo Bool" in
318                   failwith (msg_erro_pos (posicao cond) msg))
319     in CmdWhile (expCond,comandos_tipados)
320   | CmdIf (teste, entao, senao) ->
321     let (teste1,tinf) = infere_exp amb teste in
322     let _ = mesmo_tipo (posicao teste)
323             "O teste do if deveria ser do tipo %s e nao %s"
324             TipoBool tinf in
325     let entao1 = List.map (verifica_cmd amb tiporet) entao in
326     let senao1 =
327       match senao with
328       | None       -> None
329       | Some bloco -> let c = verifica_cmd amb tiporet bloco in
            Some c
330     in CmdIf (teste1, entao1, senao1)
331   | CmdElse comandos ->
332     let comandos = List.map (verifica_cmd amb tiporet) comandos
        in
333       CmdElse comandos
334   | CmdAtrib (elem, exp) ->
```

```
335         let (var1, tdir) = infere_exp amb exp in
336         ( match elem with
337           S.ExpVar (id,pos) ->
338            (try
339               begin
340                 (match (Amb.busca amb id) with
341                     Amb.EntVar tipo ->
342                       let _ = mesmo_tipo pos
343                         "Atribuicao com tipos diferentes: %s = %s"
344                         tipo tdir in
345                         CmdAtrib (T.ExpVar (id, tipo), var1)
346                   | Amb.EntFun _ ->
347                       let msg = "nome de funcao usado como nome de
                              variavel: " ^ id in
348                       failwith (msg_erro_pos pos msg) )
349               end
350             with Not_found ->
351               let _ = Amb.insere_local amb id tdir in
352               CmdAtrib (T.ExpVar (id, tdir), var1))
353           | _ -> failwith "Falha CmdAtrib"
354         )
355 and verifica_fun amb ast =
356   let open A in
357   match ast with
358   | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
359     (* Estende o ambiente global, adicionando um ambiente local *)
360     let ambfn = Amb.novo_escopo amb in
361     (* Insere os parâmetros no novo ambiente *)
362     let insere_parametro (v,t) = Amb.insere_param ambfn (fst v) t in
363       let _ = List.iter insere_parametro fn_formais in
364     (* Verifica cada comando presente no corpo da função usando o
           novo ambiente *)
365     let corpo_tipado = List.map (verifica_cmd ambfn fn_tiporet)
          fn_corpo in
366       Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo =
            corpo_tipado}
367   | Cmd _ -> failwith "Instrucao invalida"
368
369 let rec verifica_dup xs =
370   match xs with
371   | [] -> []
372   | (nome,t)::xs ->
373     let id = fst nome in
374     if (List.for_all (fun (n,t) -> (fst n) <> id) xs)
375     then (id, t) :: verifica_dup xs
376     else let msg = "Parametro duplicado " ^ id in
377       failwith (msg_erro nome msg)
378
379 let insere_declaracao_fun amb dec =
380   let open A in
381     match dec with
382     | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
383       let formais = verifica_dup fn_formais in
384       let nome = fst fn_nome in
385       Amb.insere_fun amb nome formais fn_tiporet
386     | Cmd _ -> failwith "Instrucao invalida"
387
388 let fn_predefs =
389   let open A in [
```

```
390     ("printi", [("x", TipoInt  )], TipoNone);
391     ("prints", [("x", TipoStr  )], TipoNone);
392     ("printc", [("x", TipoChar )], TipoNone);
393     ("printf", [("x", TipoFloat)], TipoNone);
394     ("inputi", [("x", TipoInt  )], TipoNone);
395     ("inputs", [("x", TipoStr  )], TipoNone);
396     ("inputc", [("x", TipoChar )], TipoNone);
397     ("inputf", [("x", TipoFloat)], TipoNone)]
398
399 let declara_predefinidas amb =
400   List.iter (fun (n,ps,tr) -> Amb.insere_fun amb n ps tr) fn_predefs
401
402 let semantico ast =
403   let amb_global = Amb.novo_amb [] in
404   let _ = declara_predefinidas amb_global in
405   let A.Programa instr = ast in
406   let decs_funs = List.filter(fun x ->
407     (match x with
408     | A.Funcao _ -> true
409     | _          -> false)) instr in
410     let _ = List.iter (insere_declaracao_fun amb_global) decs_funs in
411       let decs_funs = List.map (verifica_fun amb_global) decs_funs in
412       (A.Programa decs_funs, amb_global)
```

- semantico.mli

Listagem 7.11: semantico.mli

```
1 val semantico : (Sast.expressao Ast.programa) -> Tast.expressao Ast.
    programa * Ambiente.t
```

- sintatico.mly

Listagem 7.12: sintatico.mly

```
1 %{
2 open Ast
3 open Sast
4 %}
5
6 %token <int * int * token list>   Linha
7 %token <int * Lexing.position>    LITINT
8 %token <char * Lexing.position>   LITCHAR
9 %token <bool * Lexing.position>   LITBOOL
10 %token <float * Lexing.position>  LITFLOAT
11 %token <string * Lexing.position> LITSTRING
12 %token <string * Lexing.position> ID
13 %token <Lexing.position> DEF
14 %token <Lexing.position> RETURN
15 %token <Lexing.position> RANGE INPUTI INPUTF INPUTC INPUTS PRINT
16 %token <Lexing.position> I32 F32 CHAR STRING BOOL NONE
17 %token <Lexing.position> APAR FPAR
18 %token <Lexing.position> VIRG DPTOS SETA
19 %token <Lexing.position> OULOG ELOG NAO
20 %token <Lexing.position> MENOR MAIOR IGUAL MAIORIGUAL MENORIGUAL
     DIFERENTE
21 %token <Lexing.position> SOMAATRIB SUBATRIB MULATRIB DIVATRIB
     MODATRIB
22 %token <Lexing.position> MAIS MENOS MUL DIV MOD
```

```
%token <Lexing.position> ATRIB
%token <Lexing.position> IF ELIF ELSE
%token <Lexing.position> WHILE FOR IN
%token INDENTA DEDENTA
%token NOVALINHA
%token EOF

%left  OULOG
%left  ELOG
%left  IGUAL DIFERENTE
%left  MAIOR MENOR MAIORIGUAL MENORIGUAL
%left  MAIS MENOS
%left  MUL DIV MOD
%nonassoc unary_minus

%start <Sast.expressao Ast.programa> programa

%%

programa: ins= instrucao*
          EOF
          { Programa ins }

funcao: DEF  nome=ID
        APAR args=separated_list(VIRG, parametro) FPAR
        SETA retorno=tipo DPTOS NOVALINHA
        INDENTA
        cmd=comandos
        DEDENTA
        {
          Funcao {
            fn_nome    = nome;
            fn_tiporet = retorno;
            fn_formais = args;
            fn_corpo   = cmd
          }
        }

parametro: nome = ID DPTOS t = tipo { (nome, t) }

instrucao: func=funcao   { func    }
         | cmd= comando  { Cmd cmd }

comandos: cmd=comando+ { cmd }

tipo: I32     { TipoInt   }
    | STRING  { TipoStr   }
    | BOOL    { TipoBool  }
    | CHAR    { TipoChar  }
    | F32     { TipoFloat }
    | NONE    { TipoNone  }

comando: c=comando_atribuicao       { c }
       | c=comando_atribuicao_div   { c }
       | c=comando_atribuicao_mais  { c }
       | c=comando_atribuicao_mod   { c }
       | c=comando_atribuicao_mul   { c }
       | c=comando_atribuicao_sub   { c }
       | c=comando_se               { c }
```

```
82        | c=comando_while            { c }
83        | c=comando_for              { c }
84        | c=comando_inputi           { c }
85        | c=comando_inputf           { c }
86        | c=comando_inputc           { c }
87        | c=comando_inputs           { c }
88        | c=comando_print            { c }
89        | c=comando_chamada          { c }
90        | c=comando_retorno          { c }
91
92 comando_atribuicao: v=ID ATRIB e=expressao NOVALINHA {
93   CmdAtrib (ExpVar v , e) }
94
95 comando_atribuicao_div: v=ID SUBATRIB e=expressao NOVALINHA {
96   CmdAtrib (ExpVar v , ExpOperB ((Div, snd v), ExpVar v, e)) }
97
98 comando_atribuicao_sub: v=ID DIVATRIB e=expressao NOVALINHA {
99   CmdAtrib (ExpVar v , ExpOperB ((Menos, snd v), ExpVar v, e)) }
100
101 comando_atribuicao_mais: v=ID SOMAATRIB e=expressao NOVALINHA {
102   CmdAtrib (ExpVar v , ExpOperB ((Mais, snd v), ExpVar v, e)) }
103
104 comando_atribuicao_mod: v=ID MODATRIB e=expressao NOVALINHA {
105   CmdAtrib (ExpVar v , ExpOperB ((Mod, snd v), ExpVar v, e)) }
106
107 comando_atribuicao_mul: v=ID MULATRIB e=expressao NOVALINHA {
108   CmdAtrib (ExpVar v , ExpOperB ((Mul, snd v), ExpVar v, e)) }
109
110 comando_inputi: INPUTI exp=expressao NOVALINHA { CmdInputi exp }
111 comando_inputf: INPUTF exp=expressao NOVALINHA { CmdInputf exp }
112 comando_inputs: INPUTS exp=expressao NOVALINHA { CmdInputs exp }
113 comando_inputc: INPUTC exp=expressao NOVALINHA { CmdInputc exp }
114
115 comando_print: PRINT exp=expressao NOVALINHA { CmdPrint exp }
116
117 comando_se :  IF cond=expressao DPTOS NOVALINHA INDENTA entao=
      comandos DEDENTA
118                 cmd1=option(comando_se2){ CmdIf (cond, entao, cmd1)
                       }
119
120 comando_se2:  ELIF cond1=expressao DPTOS NOVALINHA INDENTA entao1=
      comandos DEDENTA
121                   cmd1 =option(comando_se2){ CmdIf (cond1, entao1,
                         cmd1) }
122             | ELSE DPTOS NOVALINHA INDENTA cmd2=comandos DEDENTA {
                    CmdElse cmd2 }
123
124 comando_while: WHILE cond=expressao DPTOS NOVALINHA INDENTA cmd=
      comandos DEDENTA
125             { CmdWhile(cond,cmd) }
126
127 comando_for: FOR v=ID IN RANGE APAR n1=expressao VIRG n2=expressao
      FPAR DPTOS NOVALINHA
128              INDENTA
129              cmd=comandos
130              DEDENTA
131              {
132                 CmdIf ( ExpBool (true, snd v),
133                         [
```

```
134                             CmdAtrib (ExpVar v , n1) ;
135                             CmdWhile (
136                               ExpOperB  ((Menor, snd v),
137                                          ExpVar v,
138                                          n2
139                                         ),
140                               List.append cmd [CmdAtrib (ExpVar v ,
141                                                   ExpOperB (
142                                                    (Mais, snd v),
143                                                     ExpVar v,
144                                                     ExpInt (1, snd v))
145                                                 )
146                                                ]
147                             )
148                           ],
149                           None
150                         )
151             }
152
153 comando_chamada: exp=chamada NOVALINHA { CmdChmd exp }
154
155 chamada : nome=ID APAR args=separated_list(VIRG, expressao) FPAR {
      ExpChmd (nome, args) }
156
157 comando_retorno: RETURN e=expressao? NOVALINHA { CmdReturn e }
158
159 expressao:
160       f=chamada   {          f }
161     | v=ID        { ExpVar   v }
162     | i=LITINT    { ExpInt   i }
163     | c=LITCHAR   { ExpChar  c }
164     | f=LITFLOAT  { ExpFloat f }
165     | s=LITSTRING { ExpStr   s }
166     | b=LITBOOL   { ExpBool  b }
167     | op=operU e=expressao %prec unary_minus { ExpOperU (op,e  ) }
168     | l=expressao op=operB r=expressao       { ExpOperB (op,l,r) }
169     | APAR e=expressao FPAR { e }
170
171 %inline operB:
172       pos = ELOG       { (Elog, pos)      }
173     | pos = OULOG      { (Oulog,pos)      }
174     | pos = MAIS       { (Mais, pos)      }
175     | pos = MENOS      { (Menos,pos)      }
176     | pos = MUL        { (Mul,  pos)      }
177     | pos = DIV        { (Div,  pos)      }
178     | pos = MOD        { (Mod,  pos)      }
179     | pos = IGUAL      { (Igual,pos)      }
180     | pos = DIFERENTE  { (Difer,pos)      }
181     | pos = MAIOR      { (Maior,pos)      }
182     | pos = MENOR      { (Menor,pos)      }
183     | pos = MAIORIGUAL { (MaiorIgual,pos) }
184     | pos = MENORIGUAL { (MenorIgual,pos) }
185 %inline operU:
186     | pos = MENOS      { (Menos,pos)      }
187     | pos = NAO        { (Not  ,pos)      }
```

- tabsimb.ml

Listagem 7.13: tabsimb.ml

```
type 'a tabela = {
    tbl: (string, 'a) Hashtbl.t;
    pai: 'a tabela option;
}

exception Entrada_existente of string;;

let insere amb ch v =
  if Hashtbl.mem amb.tbl ch
  then raise (Entrada_existente ch)
  else Hashtbl.add amb.tbl ch v

let substitui amb ch v = Hashtbl.replace amb.tbl ch v

let rec atualiza amb ch v =
    if Hashtbl.mem amb.tbl ch
    then Hashtbl.replace amb.tbl ch v
    else match amb.pai with
       None -> failwith "tabsim atualiza: chave nao encontrada"
     | Some a -> atualiza a ch v

let rec busca amb ch =
  try Hashtbl.find amb.tbl ch
  with Not_found ->
    (match amb.pai with
      None   -> raise Not_found
    | Some a -> busca a ch)

let rec cria cvs =
  let amb = {
    tbl = Hashtbl.create 5;
    pai = None
  } in
  let _ = List.iter (fun (c,v) -> insere amb c v) cvs
  in amb

let novo_escopo amb_pai = {
  tbl = Hashtbl.create 5;
  pai = Some amb_pai
}
```

- tabsimb.mli

Listagem 7.14: tabsimb.mli

```
type 'a tabela

exception Entrada_existente of string

val insere    : 'a tabela -> string -> 'a -> unit
val substitui : 'a tabela -> string -> 'a -> unit
val atualiza  : 'a tabela -> string -> 'a -> unit
val busca     : 'a tabela -> string -> 'a
val cria      : (string * 'a) list  -> 'a tabela

val novo_escopo : 'a tabela -> 'a tabela
```

- tast.ml

Listagem 7.15: tast.ml

```
1  open Ast
2
3  type expressao =
4    | ExpVar  of identificador * tipo
5    | ExpNone
6      | ExpInt  of int    * tipo
7      | ExpStr    of string * tipo
8      | ExpChar   of char   * tipo
9      | ExpBool   of bool   * tipo
10     | ExpFloat  of float  * tipo
11     | ExpOperB  of (operador * tipo) * (expressao * tipo) * (
           expressao * tipo)
12     | ExpOperU  of (operador * tipo) * (expressao * tipo)
13     | ExpChmd   of identificador * (expressao expressoes) * tipo
```