Construção de um compilador de Python para Dalvik usando Objective Caml

Miguel Henrique de Brito Pereia
miguelhbrito@gmail.com

Faculdade de Computação
Universidade Federal de Uberlândia

29 de junho de 2019

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este documento foi escrito para auxiliar na confecção do relatório da disciplina de Construção de Compiladores com a finalidade de detalhar todo o trabalho desenvolvido e os processos envolvidos da Construção de um Compilador, mais especificamente, um Compilador de Python para Dalvik, utilizando a linguagem OCaml para a construção do mesmo.

1.1 Sistema Operacional

Para esse trabalho foi utilizado o sistema operacional *Fedora 28*, sua instalação é fácil e rápida, basta acessar o site e seguir os passos descritos pela desenvolvedora do sistema.

1.2 Python

Python é uma ótima linguagem de programação orientada a objetos, interpretada e interativa é uma linguagem de programação orientada a objetos, interpretada, de script, interativa, funcional e de tipagem dinâmica. Criada por Guido van Rossum em 1991, hoje segue o modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos *Python Software Foundation*.

1.3 Dalvik

Desenvolvida por Dan Bornstein e com contribuições de outros engenheiros do Google, é uma máquina virtual baseada em registradores e foi projetada para ser utilizada no sistema operacional Android. É muito conhecida pelo seu bom desempenho, pelo baixo consumo de memória e foi projetada para permitir que múltiplas instâncias da máquina virtual rodem ao mesmo tempo. A *Dalvik* é frequentemante confundida com uma Java Virtual Machine, porém, o bytecode que ela opera é bastante diferente do bytecode da JVM.
A VM do Dalvik, executa um bytecode no formato .dex (Dalvik Executable), codigos em .dex

são ilegíveis aos humanos, portanto, neste trabalho usaremos Smali Code para apresentar os códigos.

1.4 Smali/Baksmali

O *smali/baksmali* é um *assembler/disassembler* para o formato *.dex* usado pela *Dalvik*, que gera um arquivo *SmaliCode*. A sintaxe é vagamente baseada na sintaxe do *Jasmin*, e suporta a funcionalidadesdo formato *.dex* (anotações, informações de depuração, informações de linha, etc.)

1.5 OCaml

Objective Caml, ou somente *OCaml*, é uma linguagem de programação funcional e fortemente tipada, da família ML com ênfase na expressividade e na segurança. É usada em aplicações sensíveis onde um único erro pode custar milhões.
Será utilizada na implementação de nosso compilador.

Capítulo 2

Instalações

2.1 Python

Já vem instalado por padrão em sistemas *GNU/Linux*, para conferir a versão, digite no terminal:

```
>  which python
```

2.2 Java

Para checar as versões disponíveis, digite no terminal:

```
>sudo dnf search openjdk
```

Instale a versão desejada digitando no terminal:

```
>sudo dnf install <openjdk-package-name>
```

Por exemplo:

```
>sudo dnf install java-1.8.0-openjdk.x86_64
```

Para verificar se foi instalado com sucesso digite:

```
>java -version
```

2.3 Dalvik

O Dalvik é uma VM executada em android, então neste trabalha usaremos o Android Studio. Para instalar basta ir no site baixar a versão que se aplica ao seu SO e configurar o PATH no terminal:

```
>export PATH=\$PATH:/diretoriolocal/android-studio/bin
```

Para executar o Android Studio, entre no diretório android-studio/bin e digite no terminal:

```
>sh studio.sh
```

Os componentes adicionais serão instalados com a ajuda do assistente de configuração na primeira execução do programa.

2.4 OCaml

Versão utilizada: 4.07.0

```
>sudo dnf install wget
>sudo dnf install git m4 mercurial darcs
>wget https://raw.github.com/ocaml/opam/master/shell/
>opam_installer.sh -O - | sh -s /dev/bin
>opam init
>eval `opam config env`
>opam repository add git git+https://github.com/ocaml/
>opam-repository
>opam update
```

Para saber qual a versão mais atual:

```
>opam switch
```

Instalando a versão 4.0.7:

```
>opam switch 4.07.0
>eval `opam config env`
```

Instalar o rlwraper para trabalhar melhor com o OCaml:

```
> sudo dnf update
> sudo dnf install rlwrap
```

Executar o OCaml com o rlwrap:

```
>rlwraper ocaml
```

2.5 Compilação

Como gerar .smeli a partir do .dex.

2.5.1 Compilando Java em .dex

Dentro do diretório execute no terminal:

```
>javac file.java
> diretorioSDK/build-tools/version/dx --dex --output=file.dex file.class
```

Onde esta "diretorioSDK"é o caminho no qual foi instalado o SDK com o assistente de configuração do Android Studio, assim, como "version"a versão que esta sendo utilizada.
Baixe o baksmali e smali, os dois na versão 2.2.6, no site para fazer o desassembly.
É importante ressaltar que tem que deixar o .dex/.smali e o baksmali/smali no mesmo diretório.

2.5.2 Complilando .dex em .smali

Dentro do diretório execute no terminal:

```
>java -jar baksmali-2.2.6.jar disassemble file.dex
```

2.5.3 Compilando .smali code em .dex

Dentro do diretório execute no terminal:

```
> java -jar smali-2.2.6.jar assemble file.smali -o file.dex
```

2.6 Executando o arquivo .dex no Android

Depois de compilado o .dex, para executa-lo usaremos um emulador que rode o sistema operacional android. Usaremos o modelo Nexus 5, android 5.1.

2.6.1 Utilizando Emulador AVD

Após instalado o Android Studio, execute-o e crie um novo projeto em branco. Na interface do programa, vá em: *Tools>ADV Manager*.

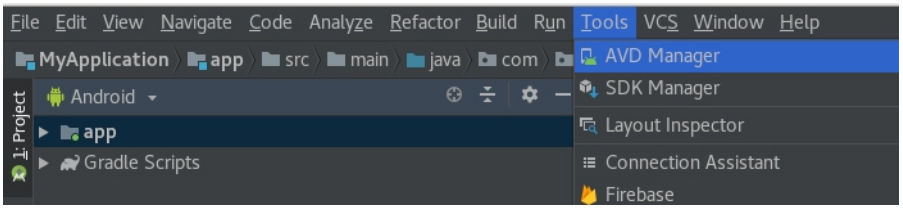

Figura 2.1: *ADV Manager*

A seguinte tela irá aparecer:

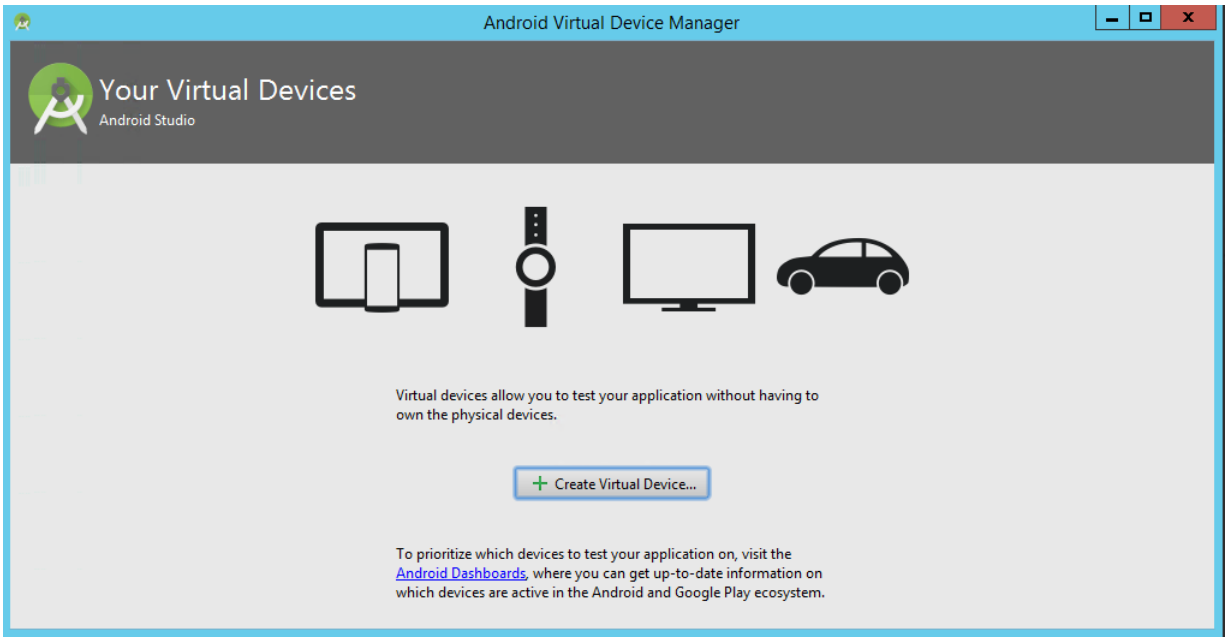

Figura 2.2: *Criar um novo device*

Nessa janela, selecione *Create Virtual Device*.
Logo em seguida irá aparecer essa janela:

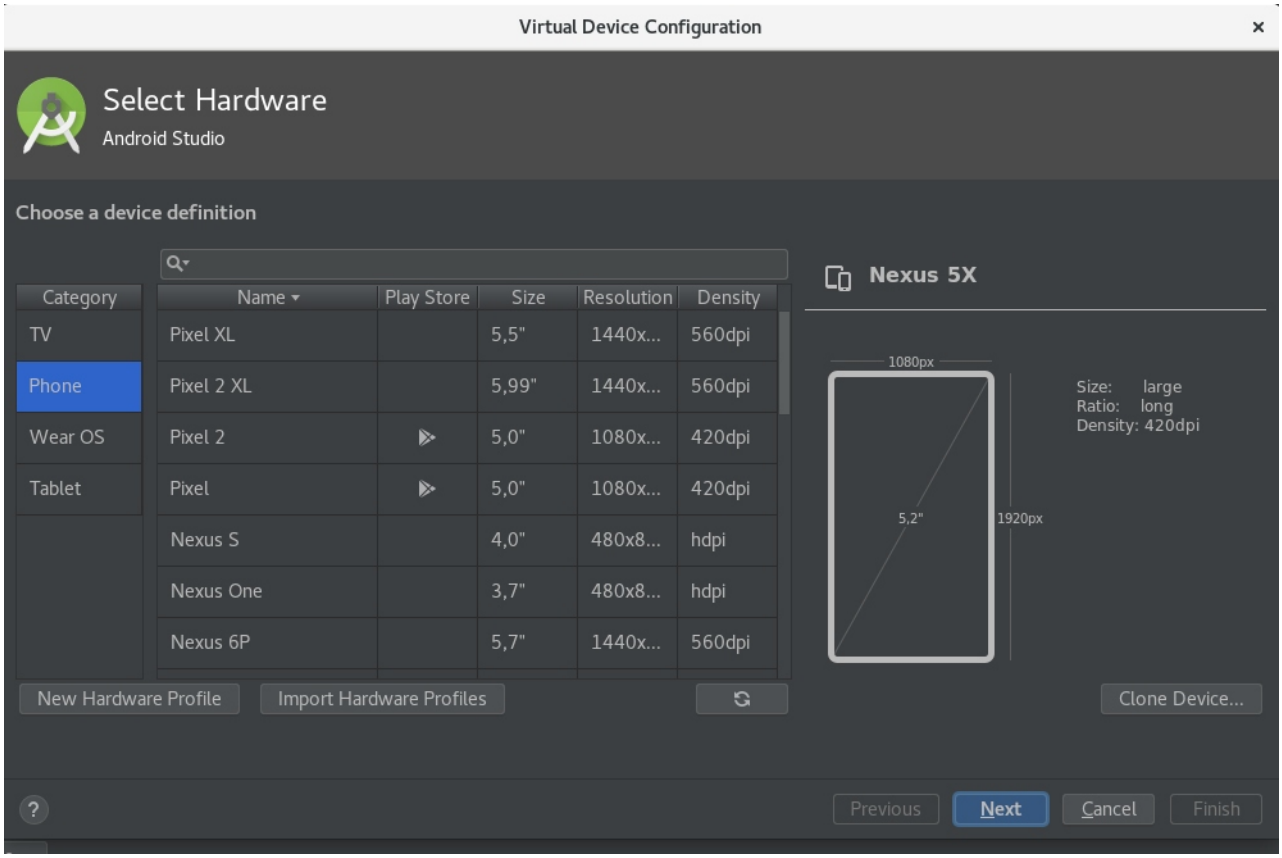

Figura 2.3: *Modelo Device*

Selecione o modelo que deseja emular, indico usar modelos Nexus, continue para as próximas

janelas clicando em next.
Abrindo o ADV Manager novamente, seu modelo emulado deverá aparecer como mostrado na figura abaixo, para iniciar, clique no icone *Play*.

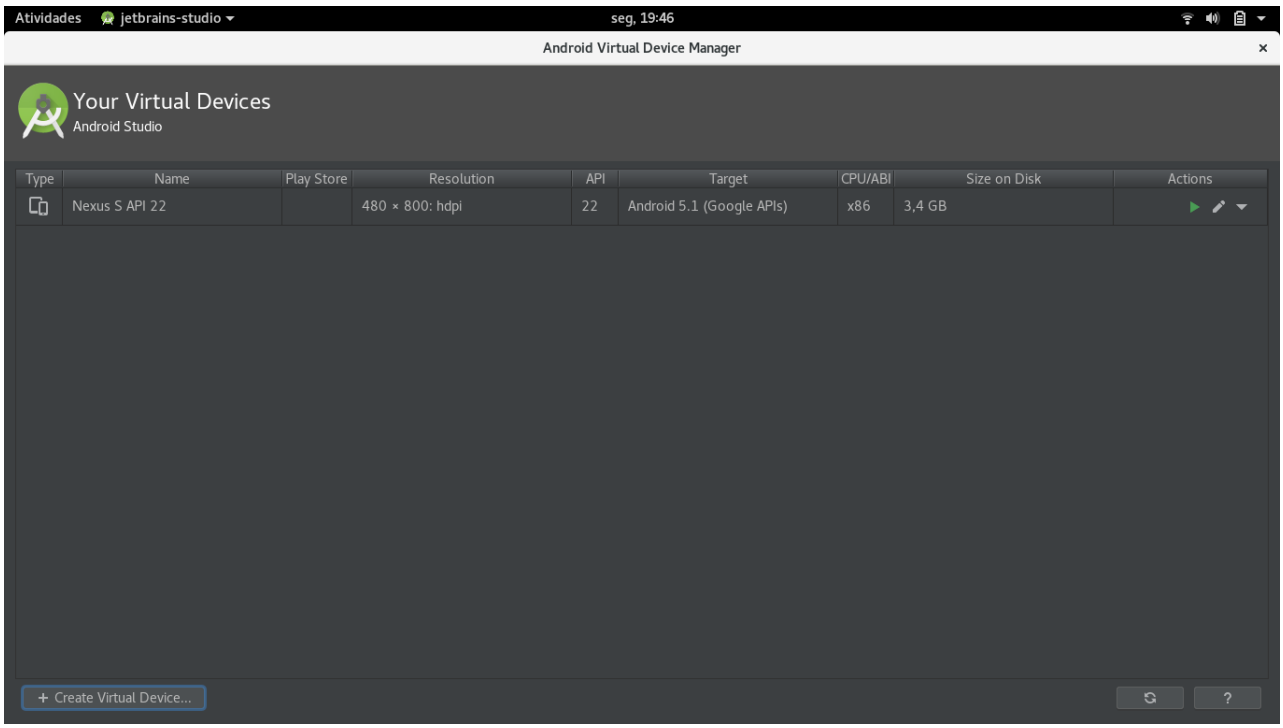

Figura 2.4: *Manager Devices*

Ao faazer esses passos ira aparecer um emulador da tela do celular, com isso ja podemos rodar arquivos .dex.
No terminal digite:

```
>./adb devices
>./adb push /home/fenrir/dev/Projetos/compilados/nano01.dex /data/local
> ./adb shell dalvikvm -cp /data/local/nano01.dex nano01
```

A execução da ferramenta adb deve ser feita dentro do diretório *androidpath/sdk/plataform-tools/*.

Capítulo 3

Nano Programas

Neste capítulo será apresentado alguns programas e suas respectivas versões em Python, Java e Smali Code.

3.1 Nano01

Listagem 3.1: Nano01.py

```
1 def main() -> None:
2     return
```

Listagem 3.2: Nano01.java

```
1 public class Nano01 {
2     public static void main(String[] args) {
3     }
4 }
```

Listagem 3.3: Nano01.smali

```
1 .class public LNano01;
2 .super Ljava/lang/Object;
3 .source "Nano01.java"
4
5
6 # direct methods
7 .method public constructor <init>()V
8     .registers 1
9
10    .prologue
11    .line 1
12    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14    return-void
15 .end method
16
17 .method public static main([Ljava/lang/String;)V
18    .registers 1
19
```

```
20      .prologue
21      .line 3
22      return-void
23  .end method
```

Saída : nenhuma.

3.2 Nano02

Listagem 3.4: Nano02.py

```
1 def main() -> None:
2     n: int = 0
```

Listagem 3.5: Nano02.java

```
1 public class Nano02 {
2     public static void main(String[] args) {
3         int n;
4     }
5  }
```

Listagem 3.6: Nano02.smali

```
1  .class public LNano02;
2  .super Ljava/lang/Object;
3  .source "Nano02.java"
4
5
6  # direct methods
7  .method public constructor <init>()V
8      .registers 1
9
10     .prologue
11     .line 1
12     invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14     return-void
15 .end method
16
17 .method public static main([Ljava/lang/String;)V
18     .registers 1
19
20     .prologue
21     .line 4
22     return-void
23 .end method
```

Saída : nenhuma.

3.3 Nano03

Listagem 3.7: Nano03.py

```
1 def main() -> None:
```

```
2     n: int = 1
```

Listagem 3.8: Nano03.java

```
1 public class Nano03 {
2     public static void main(String[] args) {
3         int n;
4         n = 1;
5     }
6 }
```

Listagem 3.9: Nano03.smali

```
1 .class public LNano03;
2 .super Ljava/lang/Object;
3 .source "Nano03.java"
4
5
6 # direct methods
7 .method public constructor <init>()V
8     .registers 1
9
10    .prologue
11    .line 1
12    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14    return-void
15 .end method
16
17 .method public static main([Ljava/lang/String;)V
18    .registers 1
19
20    .prologue
21    .line 4
22    .line 5
23    return-void
24 .end method
```

Saída : nenhuma.

3.4 Nano04

Listagem 3.10: Nano04.py

```
1 def main() -> None:
2     n: int = 1 + 2
```

Listagem 3.11: Nano04.java

```
1 public class Nano04 {
2     public static void main(String[] args) {
3        int n;
4        n = 1+2;
5     }
6 }
```

Listagem 3.12: Nano04.smali

```
.class public LNano04;
.super Ljava/lang/Object;
.source "Nano04.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 1

    .prologue
    .line 4
    .line 5
    return-void
.end method
```

Saída : nenhuma.

3.5 Nano05

Listagem 3.13: Nano05.py

```
def main() -> None:
    n: int = 2
    print(n)

main()
```

Listagem 3.14: Nano05.java

```
public class Nano05 {
    public static void main(String[] args) {
        int n;
        n = 2;
        System.out.println(n);
    }
}
```

Listagem 3.15: Nano05.smali

```
.class public LNano05;
.super Ljava/lang/Object;
.source "Nano05.java"


# direct methods
```

```
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 3

    .prologue
    .line 4
    const/4 v0, 0x2

    .line 5
    sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;

    invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(I)V

    .line 6
    return-void
.end method
```

Saída : 2.

3.6 Nano06

Listagem 3.16: Nano06.py

```
def main() -> None:
    n: int = 1 - 2
    print(n)

main()
```

Listagem 3.17: Nano06.java

```
public class Nano06 {
    public static void main(String[] args) {
        int n;
        n = 1-2;
        System.out.println(n);
    }
}
```

Listagem 3.18: Nano06.smali

```
.class public LNano06;
.super Ljava/lang/Object;
.source "Nano06.java"


# direct methods
.method public constructor <init>()V
```

```
8       .registers 1
9
10      .prologue
11      .line 1
12      invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14      return-void
15  .end method
16
17  .method public static main([Ljava/lang/String;)V
18      .registers 3
19
20      .prologue
21      .line 4
22      const/4 v0, -0x1
23
24      .line 5
25      sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;
26
27      invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(I)V
28
29      .line 6
30      return-void
31  .end method
```

Saída : -1.

3.7 Nano07

Listagem 3.19: Nano07.py

```
1 def main() -> None:
2     n = 1
3     if n == 1:
4         print(n)
5
6 main()
```

Listagem 3.20: Nano07.java

```
1 public class Nano07 {
2     public static void main(String[] args) {
3         int n;
4         n = 1;
5         if(n==1){
6             System.out.println(n);
7         }
8     }
9 }
```

Listagem 3.21: Nano07.smali

```
1 .class public LNano07;
2 .super Ljava/lang/Object;
3 .source "Nano07.java"
4
5
```

```
# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 3

    .prologue
    .line 4
    const/4 v0, 0x1

    .line 6
    sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;

    invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(I)V

    .line 8
    return-void
.end method
```

Saída : 1.

3.8 Nano08

Listagem 3.22: Nano08.py

```
def main() -> None:
    n: int = 1
    if n == 1:
        print(n)
    else:
        print(0)

main()
```

Listagem 3.23: Nano08.java

```
public class Nano08 {
    public static void main(String[] args) {
        int n;
        n = 1;
        if(n==1){
            System.out.println(n);
        }else{
            System.out.println(0);
        }
    }
}
```

Listagem 3.24: Nano08.smali

```
.class public LNano08;
.super Ljava/lang/Object;
.source "Nano08.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 3

    .prologue
    .line 4
    const/4 v0, 0x1

    .line 6
    sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;

    invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(I)V

    .line 10
    return-void
.end method
```

Saída : 1.

3.9 Nano09

Listagem 3.25: Nano09.py

```
def main() -> None:
    n: int = 1 + (1 / 2)
    if n == 1:
        print(n)
    else:
        print(0)

main()
```

Listagem 3.26: Nano09.java

```
public class Nano09 {
    public static void main(String[] args) {
        int n;

        n = 1 + 1 / 2;
        if(n==1){
```

```
7               System.out.println(n);
8           } else {
9               System.out.println(0);
10          }
11      }
12  }
```

Listagem 3.27: Nano09.smali

```
1  .class public LNano09;
2  .super Ljava/lang/Object;
3  .source "Nano09.java"
4
5
6  # direct methods
7  .method public constructor <init>()V
8      .registers 1
9
10     .prologue
11     .line 1
12     invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14     return-void
15 .end method
16
17 .method public static main([Ljava/lang/String;)V
18     .registers 3
19
20     .prologue
21     .line 5
22     const/4 v0, 0x1
23
24     .line 7
25     sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;
26
27     invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(I)V
28
29     .line 11
30     return-void
31 .end method
```

Saída : 0.

3.10 Nano10

Listagem 3.28: Nano10.py

```
1 def main() -> None:
2       n: int = 1
3       m: int = 2
4       if n == m:
5           print(n)
6       else:
7           print(0)
8
9 main()
```

Listagem 3.29: Nano10.java

```
1 public class Nano10 {
2     public static void main(String[] args) {
3         int n, m;
4         n = 1;
5         m = 2;
6         if(n==m){
7             System.out.println(n);
8         }else{
9             System.out.println(0);
10        }
11    }
12
13 }
```

Listagem 3.30: Nano10.smali

```
1 .class public LNano10;
2 .super Ljava/lang/Object;
3 .source "Nano10.java"
4
5
6 # direct methods
7 .method public constructor <init>()V
8     .registers 1
9
10    .prologue
11    .line 1
12    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
13
14    return-void
15 .end method
16
17 .method public static main([Ljava/lang/String;)V
18    .registers 3
19
20    .prologue
21    .line 4
22    .line 9
23    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
24
25    const/4 v1, 0x0
26
27    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(I)V
28
29    .line 11
30    return-void
31 .end method
```

Saída : 0.

3.11 Nano11

Listagem 3.31: Nano11.py

```
1 def main() -> None:
```

```
    n: int = 1
    m: int = 2
    x: int = 5
    while x > n:
        n = n + m
        print(n)

main()
```

Listagem 3.32: Nano11.java

```
public class Nano11 {
    public static void main(String[] args) {
        int n, m, x;

        n = 1;
        m = 2;
        x = 5;
        while (x>n) {
            n = n + m;
            System.out.println(n);
        }
    }
}
```

Listagem 3.33: Nano11.smali

```
.class public LNano11;
.super Ljava/lang/Object;
.source "Nano11.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 5

    .prologue
    .line 5
    const/4 v0, 0x1

    .line 6
    const/4 v1, 0x2

    .line 7
    const/4 v2, 0x5

    .line 8
    :goto_3
    if-le v2, v0, :cond_c
```

```
33
34      .line 9
35      add-int/2addr v0, v1
36
37      .line 10
38      sget-object v3, Ljava/lang/System;->out:Ljava/io/PrintStream;
39
40      invoke-virtual {v3, v0}, Ljava/io/PrintStream;->println(I)V
41
42      goto :goto_3
43
44      .line 12
45      :cond_c
46      return-void
47  .end method
```

Saída : 3.
Saída : 5.

3.12 Nano12

Listagem 3.34: Nano12.py

```
1  def main() -> None:
2      n: int = 1
3      m: int = 2
4      x: int = 5
5      while x > n:
6          if n == m:
7              print(n)
8          else:
9              print(0)
10         x = x - 1
11
12 main()
```

Listagem 3.35: Nano12.java

```
1  public class Nano12 {
2      public static void main(String[] args) {
3          int n, m, x;
4          n = 1;
5          m = 2;
6          x = 5;
7          while(x>n){
8              if(n==m)
9                  System.out.println(n);
10             else
11                 System.out.println(0);
12             x = x - 1;
13         }
14     }
15 }
```

Listagem 3.36: Nano12.smali

```
1  .class public LNano12;
```

```
.super Ljava/lang/Object;
.source "Nano12.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 5

    .prologue
    .line 4
    const/4 v1, 0x1

    .line 6
    const/4 v0, 0x5

    .line 7
    :goto_2
    if-le v0, v1, :cond_d

    .line 11
    sget-object v2, Ljava/lang/System;->out:Ljava/io/PrintStream;

    const/4 v3, 0x0

    invoke-virtual {v2, v3}, Ljava/io/PrintStream;->println(I)V

    .line 12
    add-int/lit8 v0, v0, -0x1

    goto :goto_2

    .line 14
    :cond_d
    return-void
.end method
```

Saída : 0.
Saída : 0.
Saída : 0.
Saída : 0.

Capítulo 4

Analisador Léxico

Esse capítulo irá abordar como foi criado o analizador léxico para a linguagem Python.

A análise léxica é a primeira fase do compilador, e tem como tarefa analisar um alfabeto de uma determinada linguagem. Após receber uma sequência de caracteres, ele produz uma sequência de nomes, palavras-chaves e sinais de pontuação chamados de *tokens*. Ainda nessa fase, é de reponsabilidade do analisador o descarte de elementos "decorativos" do programa, tais como espaços em branco e comentários entre os tokens.

A ferramenta utilizada para construir o analisador léxico foi o ocamllex, que cria um analisador, muito semelhante ao funcionamento de um Automato Finito Deterministico, a partir de um conjunto de expressões regulares e ações semânticas para tais regras.

4.1 Lista de Tokens

Tipo	Representação	Tipo	Representação
ELOG	and	APAR	(
OULOG	or	FPAR	(
DEF	def	SETA	->
RETURN	return	MENORIGUAL	<=
WHILE	while	MAIORIGUAL	>=
FOR	for	EQUIVALENTE	==
IN	in	NAOEQUIVALENTE	!=
RANGE	range	VIRG	,
INPUT	input	DPONTOS	:
PRINT	print	SOMA	+
STRING	str	SUB	-
INTEIRO	int	NOT	!
BOOL	bool	MULT	*
CHAR	char	DIV	/
FLOAT	float	MOD	%
NONE	None	MAIOR	>
IF	if	MENOR	<
ELIF	elif	ATRIB	=
ELSE	else		
TRUE	True		
FALSE	False		

Tabela 4.1: *Tabela de Tokens*

4.2 Códigos

Foi incluido o código pre_processador.ml, como foi pedido pelo professor para quem estevesse fazendo o trabalho em Python.

Listagem 4.1: lexico.mll

```
(* Analisador lexical para Mini Python
   Referência:
   https://docs.python.org/3/reference/lexical_analysis.html
*)

{
  open Lexing
  open Printf

  open Sintatico

  let pos_atual lexbuf = lexbuf.lex_start_p

  let linha_coluna_atual pos =
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol + 1 in
    (lin, col)

  let msg_erro pos msg =
    let lin, col = linha_coluna_atual pos in
    sprintf "%d-%d: %s" lin col msg

  type estado = {
    mutable indentacao : int;
    pilha : int Stack.t;
    mutable ignora_nl : int;
  }

  let cria_estado () =
    let pilha = Stack.create () in
    let _ = Stack.push 0 pilha in
      { indentacao = 0;
        pilha = pilha;
        ignora_nl = 0 }

  let ignore_nl e =
    e.ignora_nl <- succ e.ignora_nl

  let considere_nl e =
    e.ignora_nl <- pred e.ignora_nl

  exception Erro_lexico of string


  let id_ou_reservada str lexbuf =
    let pos = pos_atual lexbuf in
    match str with
     "and"      -> ELOG pos
    | "or"      -> OULOG pos
    | "def"     -> DEF pos
    | "return" -> RETURN pos
    | "while"  -> WHILE pos
```

```
53      | "for"    -> FOR pos
54      | "in"     -> IN pos
55      | "range"  -> RANGE pos
56      | "input"  -> INPUT pos
57      | "print"  -> PRINT pos
58      | "str"    -> STRING pos
59      | "int"    -> INTEIRO pos
60      | "bool"   -> BOOL pos
61      | "char"   -> CHAR pos
62      | "float"  -> FLOAT pos
63      | "None"   -> NONE pos
64      | "if"     -> IF pos
65      | "elif"   -> ELIF pos
66      | "else"   -> ELSE pos
67      | "True"   -> LITBOOL (true, pos)
68      | "False"  -> LITBOOL (false, pos)
69      | _      ->   ID (str, pos)
70
71
72  let conta_linhas s =
73    let n = ref 0 in
74    let _ = String.iter(fun c -> if c = '\n' then incr n) s
75    in !n
76
77  }
78
79
80  (* epsilon *)
81  let vazio = ""
82
83  let fim_de_linha  = '\r' | '\n' | "\r\n"
84  let espaco_branco = ' ' | '\t'
85  let comentario    = '#' [^ '\n' '\r' ]*
86  let linha_em_branco = espaco_branco* comentario?
87
88  let letra = ['a'-'z' 'A'-'Z' '_']
89  let digito = ['0'-'9']
90
91  let inteirodecimal = digito*
92  let parteinteira = digito+
93  let fracao = '.' digito+
94  let pontoflutuante = parteinteira? fracao | parteinteira '.'
95
96  let identificador = letra (letra | digito)*
97
98
99  rule token estado = parse
100 | vazio { let indentacao_atual  = estado.indentacao
101           and ultima_indentacao = Stack.top estado.pilha in
102             if indentacao_atual < ultima_indentacao
103             then (* fecha o bloco atual *)
104               let _ = Stack.pop estado.pilha
105               in DEDENTA
106             else if indentacao_atual > ultima_indentacao
107                  then (* inicia um novo bloco *)
108                    let _ = Stack.push indentacao_atual estado.pilha
109                    in INDENTA
110                  else (* mesma indentação, continua no mesmo bloco *)
111                    prox_token estado lexbuf
```

```
112   }
113
114 and prox_token estado = parse
115 | (linha_em_branco fim_de_linha)+
116     { let nlinhas = conta_linhas (lexeme lexbuf) in
117       let pos = lexbuf.lex_curr_p in
118         lexbuf.lex_curr_p <-
119           { pos with
120               pos_lnum = pos.pos_lnum + nlinhas;
121               pos_bol = pos.pos_cnum };
122         if estado.ignora_nl <= 0 then
123           let _ = estado.indentacao <- conta_identacao 0 lexbuf in
124           NOVALINHA
125         else prox_token estado lexbuf
126     }
127 | '\\' fim_de_linha espaco_branco*
128     { new_line lexbuf;
129       prox_token estado lexbuf
130     }
131
132 | espaco_branco+
133     { prox_token estado lexbuf }
134
135 (* palavras-chave e identificadores *)
136 | identificador as id
137     { id_ou_reservada id lexbuf}
138
139 (* símbolos *)
140
141 | '('  { ignore_nl estado; APAR (pos_atual lexbuf) }
142 | ')'  { considere_nl estado;  FPAR (pos_atual lexbuf) }
143 | "->"               { SETA           (pos_atual lexbuf) }
144 | "<="               { MENORIGUAL     (pos_atual lexbuf) }
145 | ">="               { MAIORIGUAL     (pos_atual lexbuf) }
146 | "=="               { EQUIVALENTE    (pos_atual lexbuf) }
147 | "!="               { NAOEQUIVALENTE (pos_atual lexbuf) }
148 | ','                { VIRG           (pos_atual lexbuf) }
149 | ':'                { DPONTOS        (pos_atual lexbuf) }
150 | '+'                { SOMA           (pos_atual lexbuf) }
151 | '-'                { SUB            (pos_atual lexbuf) }
152 | '!'                { NOT            (pos_atual lexbuf) }
153 | '*'                { MULT           (pos_atual lexbuf) }
154 | '/'                { DIV            (pos_atual lexbuf) }
155 | '%'                { MOD            (pos_atual lexbuf) }
156 | '<'                { MENOR          (pos_atual lexbuf) }
157 | '>'                { MAIOR          (pos_atual lexbuf) }
158 | '='                { ATRIB          (pos_atual lexbuf) }
159
160 (* literais *)
161 | inteirodecimal as s
162   { try LITINT ((int_of_string s), pos_atual lexbuf)
163     with _ -> raise (Erro_lexico ("constante muito grande: " ^ s)) }
164 | pontoflutuante as s
165   { try LITFLOAT ((float_of_string s), pos_atual lexbuf)
166     with _ -> raise (Erro_lexico ("constante muito grande: " ^ s)) }
167
168 | "'"_"'" as s       { let c = String.get s 1 in LITCHAR (c, (pos_atual
      lexbuf)) }
169 | "''"
```

```
170    { let buffer = Buffer.create 100 in
171      let str = string_multilinha buffer (pos_atual lexbuf) lexbuf
172      in LITSTRING (str, pos_atual lexbuf) }
173 | '\''
174    { let buffer = Buffer.create 100 in
175      let str = string_simples false buffer (pos_atual lexbuf) lexbuf
176      in LITSTRING (str, pos_atual lexbuf) }
177 | '"'
178    { let buffer = Buffer.create 100 in
179      let str = string_simples true buffer (pos_atual lexbuf) lexbuf
180      in LITSTRING (str, pos_atual lexbuf) }
181 | eof { EOF }
182
183 (* Erros *)
184 | _ as c
185    { raise
186        (Erro_lexico
187           (msg_erro (pos_atual lexbuf)
188                     (sprintf "caracter ilegal: %c" c))) }
189
190
191 and conta_identacao n = parse
192 | vazio { n }
193 | ' '   { conta_identacao (n+1) lexbuf }
194 | '\t'  { let n' = n + 8 - (n mod 8) in conta_identacao n' lexbuf }
195
196 and string_simples aspas buf pos = parse
197 | '\''
198 { if aspas
199   then  let _ = Buffer.add_char buf '\'' in
200         string_simples aspas buf pos lexbuf
201   else Buffer.contents buf }
202 | '"'
203 { if aspas
204   then Buffer.contents buf
205   else  let _ = Buffer.add_char buf '\'' in
206         string_simples aspas buf pos lexbuf
207 }
208 | "\\t"
209     { Buffer.add_char buf '\t'; string_simples aspas buf pos lexbuf }
210 | "\\n"
211     { Buffer.add_char buf '\n'; string_simples aspas buf pos lexbuf }
212 | "\\'"
213 { Buffer.add_char buf '\''; string_simples aspas buf pos lexbuf }
214 | "\\\""
215     { Buffer.add_char buf '"'; string_simples aspas buf pos lexbuf }
216 | "\\\\"
217     { Buffer.add_char buf '\\'; string_simples aspas buf pos lexbuf }
218 | [^ '\r' '\n' ] as c
219     { Buffer.add_char buf c; string_simples aspas buf pos lexbuf }
220 | fim_de_linha
221     { raise
222         (Erro_lexico
223            (msg_erro pos "uma string simples deve terminar na mesma linha"
               ))}
224 | eof
225    { raise (Erro_lexico (msg_erro pos "A string não foi fechada.")) }
226
227 and string_multilinha buf pos = parse
```

```
228 | "''''"
229     { Buffer.contents buf }
230 | "\\t"
231     { Buffer.add_char buf '\t'; string_multilinha buf pos lexbuf }
232 | "\\n"
233     { Buffer.add_char buf '\n'; string_multilinha buf pos lexbuf }
234 | "\\'"
235     { Buffer.add_char buf '\''; string_multilinha buf pos lexbuf }
236 | "\\\\"
237     { Buffer.add_char buf '\\'; string_multilinha buf pos lexbuf }
238 | fim_de_linha
239     { new_line lexbuf;
240       Buffer.add_char buf '\n'; string_multilinha buf pos lexbuf }
241 | _ as c
242     { Buffer.add_char buf c;    string_multilinha buf pos lexbuf }
243 | eof
244     { raise
245         (Erro_lexico
246            (msg_erro pos "A string multilinha não foi fechada.")) }
```

Listagem 4.2: carregador.ml

```
1  #load "lexico.cmo"
2  #load "pre_processador.cmo"
3
4  type nome_arq = string
5  type tokens = Lexico.token list
6
7  let rec tokens lexbuf =
8    let tok = Pre_processador.lexico lexbuf in
9    match tok with
10   | Lexico.EOF -> (|Lexico.EOF):tokens)
11   | _ -> tok :: tokens lexbuf
12 ;;
13
14 let lexico str =
15   let lexbuf = Lexing.from_string str in
16   tokens lexbuf
17 ;;
18
19 let lex (arq:nome_arq)=
20   let ic = open_in arq in
21   let lexbuf = Lexing.from_channel ic in
22   let toks = tokens lexbuf in
23   let _ = close_in ic in
24   toks
```

4.3 Compilação e execução

Para compilar navague pelo terminal ate o diretório dos arquivos e execute os comandos:

```
>ocamllex lexico.mll
>ocamlc -c lexico.ml
>ocamlc -c pre_processador.ml
```

Após a compilação execute:

```
>lrwrap ocaml
```

Dentro do *O caml* execute os comandos:

```
# #use "carregador.ml";;
# lex "codigo.py";;
```

Um exemplo com o arquivo teste.py com o código:

Listagem 4.3: teste.py

```
1 def paco(x):
2   x = x + 1
3   y = x + 2
4   y = x + 3 + ( 4 + 5)
5   return x
```

A saída do analisador léxico:

```
- : tokens =
  [Lexico.DEF; Lexico.ID "paco"; Lexico.APAR; Lexico.ID "x"; Lexico.FPAR;
   Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "x";
   Lexico.IGUAL; Lexico.ID "x"; Lexico.SOMA; Lexico.LITINT 1; Lexico.
      NOVALINHA;
   Lexico.ID "y"; Lexico.IGUAL; Lexico.ID "x"; Lexico.SOMA; Lexico.LITINT
      2;
   Lexico.NOVALINHA; Lexico.ID "y"; Lexico.IGUAL; Lexico.ID "x"; Lexico.
      SOMA;
   Lexico.LITINT 3; Lexico.SOMA; Lexico.APAR; Lexico.LITINT 4; Lexico.SOMA;
   Lexico.LITINT 5; Lexico.FPAR; Lexico.NOVALINHA; Lexico.RETURN;
   Lexico.ID "x"; Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.EOF]
```

4.4 Analise léxica Nanos

Analise léxica dos nanos programas em python vistos no capitulo anterior.

Listagem 4.4: nano01.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
      INDENTA;
3  Lexico.RETURN; Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.EOF]
```

Listagem 4.5: nano02.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
      INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 0; Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.EOF]
```

Listagem 4.6: nano03.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.EOF]
```

Listagem 4.7: nano04.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.SOMA; Lexico.LITINT 2; Lexico.NOVALINHA;
5  Lexico.DEDENTA; Lexico.EOF]
```

Listagem 4.8: nano05.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 2; Lexico.NOVALINHA; Lexico.ID "print"; Lexico.APAR;
5  Lexico.ID "n"; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA;
6  Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.9: nano06.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.SUB; Lexico.LITINT 2; Lexico.NOVALINHA;
5  Lexico.ID "print"; Lexico.APAR; Lexico.ID "n"; Lexico.FPAR;
6  Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR;
7  Lexico.FPAR; Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.10: nano07.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.IGUAL; Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.IF
       ;
4  Lexico.ID "n"; Lexico.EQUIVALENTE; Lexico.LITINT 1; Lexico.DPONTOS;
5  Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR;
6  Lexico.ID "n"; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA;
7  Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR;
8  Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.11: nano08.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.IF; Lexico.ID "n";
```

```
5  Lexico.EQUIVALENTE; Lexico.LITINT 1; Lexico.DPONTOS; Lexico.NOVALINHA;
6  Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR; Lexico.ID "n"; Lexico.
       FPAR;
7  Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.ELSE; Lexico.DPONTOS;
8  Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR;
9  Lexico.LITINT 0; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA;
10 Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR;
11 Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.12: nano09.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.SOMA; Lexico.APAR; Lexico.LITINT 1; Lexico.DIV;
5  Lexico.LITINT 2; Lexico.FPAR; Lexico.NOVALINHA; Lexico.IF; Lexico.ID "n";
6  Lexico.EQUIVALENTE; Lexico.LITINT 1; Lexico.DPONTOS; Lexico.NOVALINHA;
7  Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR; Lexico.ID "n"; Lexico.
       FPAR;
8  Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.ELSE; Lexico.DPONTOS;
9  Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR;
10 Lexico.LITINT 0; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA;
11 Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR;
12 Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.13: nano10.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.ID "m"; Lexico.DPONTOS;
5  Lexico.ID "int"; Lexico.IGUAL; Lexico.LITINT 2; Lexico.NOVALINHA; Lexico.
       IF;
6  Lexico.ID "n"; Lexico.EQUIVALENTE; Lexico.ID "m"; Lexico.DPONTOS;
7  Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print"; Lexico.APAR;
8  Lexico.ID "n"; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.ELSE
       ;
9  Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print";
10 Lexico.APAR; Lexico.LITINT 0; Lexico.FPAR; Lexico.NOVALINHA; Lexico.
       DEDENTA;
11 Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR;
12 Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.14: nano11.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.ID "m"; Lexico.DPONTOS;
5  Lexico.ID "int"; Lexico.IGUAL; Lexico.LITINT 2; Lexico.NOVALINHA;
6  Lexico.ID "x"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
7  Lexico.LITINT 5; Lexico.NOVALINHA; Lexico.WHILE; Lexico.ID "x";
8  Lexico.MAIOR; Lexico.ID "n"; Lexico.DPONTOS; Lexico.NOVALINHA;
9  Lexico.INDENTA; Lexico.ID "n"; Lexico.IGUAL; Lexico.ID "n"; Lexico.SOMA;
10 Lexico.ID "m"; Lexico.NOVALINHA; Lexico.ID "print"; Lexico.APAR;
```

```
11  Lexico.ID "n"; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA;
12  Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR;
13  Lexico.NOVALINHA; Lexico.EOF]
```

Listagem 4.15: nano12.py

```
1 [Lexico.DEF; Lexico.ID "main"; Lexico.APAR; Lexico.FPAR; Lexico.SUB;
2  Lexico.MAIOR; Lexico.NONE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.
       INDENTA;
3  Lexico.ID "n"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
4  Lexico.LITINT 1; Lexico.NOVALINHA; Lexico.ID "m"; Lexico.DPONTOS;
5  Lexico.ID "int"; Lexico.IGUAL; Lexico.LITINT 2; Lexico.NOVALINHA;
6  Lexico.ID "x"; Lexico.DPONTOS; Lexico.ID "int"; Lexico.IGUAL;
7  Lexico.LITINT 5; Lexico.NOVALINHA; Lexico.WHILE; Lexico.ID "x";
8  Lexico.MAIOR; Lexico.ID "n"; Lexico.DPONTOS; Lexico.NOVALINHA;
9  Lexico.INDENTA; Lexico.IF; Lexico.ID "n"; Lexico.EQUIVALENTE; Lexico.ID "
       m";
10  Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.INDENTA; Lexico.ID "print";
11  Lexico.APAR; Lexico.ID "n"; Lexico.FPAR; Lexico.NOVALINHA; Lexico.DEDENTA
       ;
12  Lexico.ELSE; Lexico.DPONTOS; Lexico.NOVALINHA; Lexico.INDENTA;
13  Lexico.ID "print"; Lexico.APAR; Lexico.LITINT 0; Lexico.FPAR;
14  Lexico.NOVALINHA; Lexico.DEDENTA; Lexico.ID "x"; Lexico.IGUAL;
15  Lexico.ID "x"; Lexico.SUB; Lexico.LITINT 1; Lexico.NOVALINHA;
16  Lexico.DEDENTA; Lexico.DEDENTA; Lexico.ID "main"; Lexico.APAR; Lexico.
       FPAR;
17  Lexico.NOVALINHA; Lexico.EOF]
```

Capítulo 5

Análise sintática

O processo de ler uma sequência de tokens de entrada e determinar uma estrutura gramatical, definida através de uma gramatica, e ainda verificar se esta correta, é uma tarefa para o analisador sintático.

No processo seguinte, o analisador sintatico transforma a sequencia de tokens em uma árvore sintática.

O analisador sintático preditivo para o reconhecimento da gramática do Python foi construído com o auxílio do Menhir, que é um gerador de parser, que tem suporte para Ocaml.

Para instalar o Menhir digite no terminal:

```
opam menhir
```

5.1 Código

Código do sintatico.mly:

Listagem 5.1: sintatico.mly

```
%{
open Ast
%}
(* literais *)
%token <int * int * token list>   Linha
%token <int> LITINT
%token <char> LITCHAR
%token <bool> LITBOOL
%token <float> LITFLOAT
%token <string> LITSTRING
%token <string> ID
(* palavras reservadas *)
%token INTEIRO
%token FLOAT
%token CHAR
%token STRING
```

```
17 %token BOOL
18 %token NONE
19 %token DEF
20 %token RETURN
21 %token RANGE
22 %token INPUT
23 %token PRINT
24 %token ATRIB
25 %token EOF
26 %token ORLOG
27 %token ANDLOG
28 %token NAO
29 (* loops *)
30 %token IF
31 %token ELIF
32 %token ELSE
33 %token WHILE
34 %token FOR
35 %token IN
36 (* operadores *)
37 %token SOMA
38 %token MULT
39 %token DIV
40 %token MOD
41 %token SUB
42 %token MENOR
43 %token MAIOR
44 %token EQUIVALENTE
45 %token MAIORIGUAL
46 %token MENORIGUAL
47 %token NAOEQUIVALENTE
48 (* simbolos *)
49 %token APAR
50 %token FPAR
51 %token VIRG
52 %token DPONTOS
53 %token SETA
54 (* leiaute *)
55 %token INDENTA
56 %token DEDENTA
57 %token NOVALINHA
58 (* tokens left *)
59 %left ORLOG
60 %left ANDLOG
61 %left EQUIVALENTE
62 %left NAOEQUIVALENTE
63 %left MAIOR
64 %left MENOR
65 %left MAIORIGUAL
66 %left MENORIGUAL
67 %left SOMA
68 %left MENOS
69 %left MULT
70 %left DIV
71 %left MOD
72
73 %start <Ast.programa> programa
74
75 %%
```

```
76
77 programa: ins=instrucao*
78           EOF
79           { Programa ins }
80
81 instrucao:
82         func = funcao  { func }
83         | cmd = comando  { Cmd cmd }
84
85 comandos: cmd = comando+ { cmd }
86
87 funcao:
88     DEF nome=ID APAR args=separated_list(VIRG, parametro) FPAR SETA
          retorno=tipo DPONTOS
89     NOVALINHA INDENTA cmd=comandos DEDENTA
90     {
91         Funcao {
92             fn_nome = nome;
93             fn_tiporet = retorno;
94             fn_formais = args;
95             fn_corpo = cmd
96         }
97     }
98
99 parametro:
100     nome = ID DPONTOS tp=tipo { (nome, tp) }
101
102 expressoes:
103     exp = separated_list(VIRG, expressao) { exp }
104
105 comando:
106          c=comando_declaracao { c }
107        | c=comando_atribuicao { c }
108        | c=comando_input { c }
109        | c=comando_input_declaracao { c }
110        | c=comando_if { c }
111        | c=comando_if2 { c }
112        | c=comando_while { c }
113        | c=comando_for { c }
114        | c=comando_for_dec { c }
115        | c=comando_print { c }
116        | c=comando_chamada { c }
117        | c=comando_retorno { c }
118
119 comando_declaracao:
120     nome = parametro option(ATRIB) exp=option(expressao) NOVALINHA {
          CmdDeclaracao (nome, exp) }
121
122 comando_atribuicao:
123     nome = ID ATRIB exp=expressao NOVALINHA { CmdAtrib (nome, exp) }
124
125 comando_input:
126     nome = ID ATRIB tp = tipo APAR INPUT APAR exp = option(expressao) FPAR
           FPAR
127     NOVALINHA { CmdInput (nome, (exp, tp)) }
128
129 comando_input_declaracao:
130     nome=parametro ATRIB tp=tipo APAR INPUT APAR exp=option(expressao)
          FPAR FPAR
```

```
131     NOVALINHA { CmdInputDeclaracao (nome, (exp, tp)) }
132
133 comando_if:
134     IF exp=expressao DPONTOS
135     NOVALINHA INDENTA cmd1=comandos DEDENTA cmd2=option(comando_if2) {
            CmdIf (exp, cmd1, cmd2) }
136
137 comando_if2:
138     ELSE DPONTOS NOVALINHA INDENTA cmd2=comandos DEDENTA { CmdElse cmd2
139 }
140             | ELIF cond1=expressao DPONTOS NOVALINHA INDENTA entao1=
                    comandos DEDENTA cmd1=option(comando_if2) {
141   CmdIf (cond1, entao1, cmd1)
142 }
143
144 comando_while:
145     WHILE exp=expressao DPONTOS NOVALINHA INDENTA cmd=comandos DEDENTA {
            CmdWhile (exp, cmd) }
146
147 comando_for_dec:
148     FOR p=parametro IN RANGE APAR exp1=expressao VIRG exp2=expressao FPAR
            DPONTOS
149     NOVALINHA INDENTA cmd=comandos DEDENTA { CmdFor_Dec (p, (exp1, exp2),
            cmd) }
150
151 comando_print:
152     PRINT APAR args=separated_list(VIRG, expressao) FPAR NOVALINHA {
            CmdPrint args }
153
154 comando_for:
155     FOR v=ID IN RANGE APAR exp1 = expressao VIRG exp2 = expressao FPAR
            DPONTOS
156     NOVALINHA INDENTA cmd=comandos DEDENTA { CmdFor (v, (exp1, exp2), cmd)
             }
157
158 comando_chamada:
159     exp=chamada NOVALINHA { CmdChmd exp }
160
161 chamada:
162     nome = ID APAR args=separated_list(VIRG, expressao) FPAR { ExpChmd (
            nome, args) }
163
164 comando_retorno: RETURN exp=option(expressao) NOVALINHA { CmdReturn exp }
165
166 tipo:
167       INTEIRO { TipoInt }
168     | STRING { TipoStr }
169     | BOOL { TipoBool }
170     | CHAR { TipoChar }
171     | FLOAT { TipoFloat }
172     | NONE { TipoNone }
173
174 expressao:
175       f=chamada { f }
176     | v=variavel { ExpVar v }
177     | i=LITINT { ExpInt i }
178     | c=LITCHAR { ExpChar c }
179     | f=LITFLOAT { ExpFloat f }
180     | s=LITSTRING { ExpStr s }
```

```
181     | b=LITBOOL { ExpBool b }
182     | e1=expressao op=operB e2=expressao { ExpOperB(op,e1,e2) }
183     | APAR exp=expressao FPAR { exp }
184     | e1=expressao op=operC e2=expressao { ExpComp(op,e1,e2) }
185     | n=operN exp=expressao { ExpNot(n,exp) }
186
187 %inline operB:
188       ANDLOG { ANDlog }
189     | ORLOG { Orlog }
190     | SOMA { Soma }
191     | SUB { Sub }
192     | MULT { Mult }
193     | DIV { Div }
194     | MOD { Mod }
195
196 %inline operC:
197       NAOEQUIVALENTE { NaoEquivalente }
198     | EQUIVALENTE { Equivalente }
199     | MAIOR { Maior }
200     | MENOR { Menor }
201     | MAIORIGUAL { MaiorIgual }
202     | MENORIGUAL { MenorIgual }
203
204 operN:
205       NAO { Not }
206
207 variavel:
208     | x = ID {VarSimples x}
```

Código da árvore sintática, ast.ml:

Listagem 5.2: ast.ml

```
1  (* The type of the abstract syntax tree (AST). *)
2  type identificador = string
3
4  type programa = Programa of instrucoes
5  and comandos   = comando list
6  and instrucoes = instrucao list
7  and expressoes = expressao list
8  and instrucao  =
9      Funcao of decfn
10   | Cmd    of comando
11
12 and decfn = {
13   fn_nome: identificador;
14   fn_tiporet: tipo;
15   fn_formais: (identificador * tipo) list;
16   fn_corpo: comandos
17 }
18
19 and tipo =
20     TipoInt
21   | TipoStr
22   | TipoBool
23   | TipoChar
24   | TipoFloat
25   | TipoNone
26
```

```
and comando =
    CmdDeclaracao of (identificador * tipo) * expressao option
  | CmdAtrib  of identificador * expressao
  | CmdInputDeclaracao of (identificador * tipo) * (expressao option *
      tipo)
  | CmdInput  of identificador * (expressao option * tipo)
  | CmdPrint  of expressao list
  | CmdIf     of expressao * comandos * (comando option)
  | CmdElse   of comandos
  | CmdReturn of expressao option
  | CmdWhile  of expressao * comandos
  | CmdFor of identificador * (expressao * expressao) * comandos
  | CmdFor_Dec of (identificador * tipo) * (expressao * expressao) *
      comandos
  | CmdChmd of expressao
and variaveis = variavel list

and variavel = VarSimples of identificador

and expressao =
    ExpVar   of variavel
  | ExpInt   of int
  | ExpStr   of string
  | ExpChar  of char
  | ExpBool  of bool
  | ExpFloat of float
  | ExpOperB of operador * expressao * expressao
  | ExpOperU of operador * expressao
  | ExpChmd  of identificador * expressoes
  | ExpComp  of (operador_comparacao * expressao * expressao)
  | ExpNot   of (operador_not * expressao)

and operador =
    Soma
  | Sub
  | Mult
  | Div
  | Mod
  | ANDlog
  | Orlog

and operador_comparacao =
    Maior
  | Menor
  | MaiorIgual
  | MenorIgual
  | Equivalente
  | NaoEquivalente

and operador_not =
    Not
```

Código do sintático para execução dos arquivos:

Listagem 5.3: sintaticoTest.ml

```
open Printf
open Lexing

```

```
4  open Ast
5  open ErroSint (* nome do módulo contendo as mensagens de erro *)
6
7  exception Erro_Sintatico of string
8
9  module S = MenhirLib.General (* Streams *)
10 module I = Sintatico.MenhirInterpreter
11
12 let posicao lexbuf =
13     let pos = lexbuf.lex_curr_p in
14     let lin = pos.pos_lnum
15     and col = pos.pos_cnum - pos.pos_bol - 1 in
16     sprintf "linha %d, coluna %d" lin col
17
18 (* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
       checkpoint *)
19
20 let pilha checkpoint =
21   match checkpoint with
22   | I.HandlingError amb -> I.stack amb
23   | _ -> assert false (* Isso não pode acontecer *)
24
25 let estado checkpoint : int =
26   match Lazy.force (pilha checkpoint) with
27   | S.Nil -> (* O parser está no estado inicial *)
28      0
29   | S.Cons (I.Element (s, _, _, _), _) ->
30      I.number s
31
32 let sucesso v = Some v
33
34 let falha lexbuf (checkpoint : Ast.programa I.checkpoint) =
35   let estado_atual = estado checkpoint in
36   let msg = message estado_atual in
37   raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
38                                         (Lexing.lexeme_start lexbuf) msg))
39
40 let loop lexbuf resultado =
41   let fornecedor = I.lexer_lexbuf_to_supplier Pre_processador.lexico
         lexbuf in
42   I.loop_handle sucesso (falha lexbuf) fornecedor resultado
43
44
45 let parse_com_erro lexbuf =
46   try
47     Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.lex_curr_p))
48   with
49   | Lexico.Erro msg ->
50      printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
51      None
52   | Erro_Sintatico msg ->
53      printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
54      None
55
56 let parse s =
57   let lexbuf = Lexing.from_string s in
58   let ast = parse_com_erro lexbuf in
59   ast
60
```

```
let parse_arq nome =
  let ic = open_in nome in
  let lexbuf = Lexing.from_channel ic in
  let result = parse_com_erro lexbuf in
  let _ = close_in ic in
  match result with
  | Some ast -> ast
  | None -> failwith "A analise sintatica falhou"
```

5.2 Execução

Para compilar os arquivos do projeto, primeiro gere:

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

Depois modifique o arquivo sintatico.msg com as suas mensagens de erro. Agora basta gerar o arquivo erroSint.ml que contém as mensagens de erro:

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

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

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

Entre no Ocaml:

```
rlwrap ocaml
```

Entre com o arquivo e extensao dentro do Ocaml:

```
parse_arq "nome_arquivo.extensao"
```

5.3 Análise sintática dos programas micro

Listagem 5.4: astMicro01

```
Ast.programa option =
Some
 (Programa
   [Funcao
     {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
      fn_corpo =
       [CmdDeclaracao ((VarSimples "cel", TipoFloat), Some (ExpFloat 0.));
        CmdDeclaracao ((VarSimples "far", TipoFloat), Some (ExpFloat 0.));
        CmdPrint [ExpStr "Tabela de conversao: Celsius -> Fahrenheit\n"];
        CmdPrint [ExpStr "Digite a temperatura em Celsius: "];
        CmdInput (VarSimples "cel", (None, TipoFloat));
        CmdAtrib (VarSimples "far",
         ExpOperB (Div,
```

```
14            ExpOperB (Soma,
15             ExpOperB (Mult, ExpFloat 9., ExpVar (VarSimples "cel")),
16             ExpFloat 160.),
17            ExpFloat 5.));
18          CmdPrint
19           [ExpStr "A nova temperatura e: "; ExpVar (VarSimples "far");
20            ExpStr " F"]]};
21     Cmd (CmdChmd (ExpChmd (VarSimples "main", []))))]
```

Listagem 5.5: astMicro02

```
1 Ast.programa option =
2 Some
3  (Programa
4    [Funcao
5      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6       fn_corpo =
7        [CmdPrint [ExpStr "Digite o primeiro numero: "];
8         CmdInputDeclaracao ((VarSimples "num1", TipoInt), (None, TipoInt))
              ;
9         CmdPrint [ExpStr "Digite o segundo numero: "];
10        CmdInputDeclaracao ((VarSimples "num2", TipoInt), (None, TipoInt))
              ;
11        CmdIf
12         (ExpComp
13           (Maior, ExpVar (VarSimples "num1"), ExpVar (VarSimples "num2"))
                 ,
14          [CmdPrint
15            [ExpStr "\nO primeiro numero"; ExpVar (VarSimples "num1");
16             ExpStr "e maior que o segundo"; ExpVar (VarSimples "num2")]],
17          Some
18           (CmdElse
19             [CmdPrint
20               [ExpStr "O segundo numero"; ExpVar (VarSimples "num2");
21                ExpStr "e maior que o primeiro"; ExpVar (VarSimples "num1")
                     ]]))]};
22     Cmd (CmdChmd (ExpChmd (VarSimples "main", []))))]
```

Listagem 5.6: astMicro03

```
1 Ast.programa option =
2 Some
3  (Programa
4    [Funcao
5      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6       fn_corpo =
7        [CmdPrint [ExpStr "Digite um numero:  "];
8         CmdInputDeclaracao ((VarSimples "numero", TipoInt), (None, TipoInt
              ));
9         CmdIf
10         (ExpComp (MaiorIgual, ExpVar (VarSimples "numero"), ExpInt 100),
11         [CmdIf
12           (ExpComp (MenorIgual, ExpVar (VarSimples "numero"), ExpInt 200)
                 ,
13           [CmdPrint [ExpStr "\n numero entre 100 e 200"]],
14           Some (CmdElse [CmdPrint [ExpStr "\n numero maior que 200"]]))],
15          Some (CmdElse [CmdPrint [ExpStr "\n numero menor que 100"]]));
16         CmdReturn None]};
17     Cmd (CmdChmd (ExpChmd (VarSimples "main", []))))]
```

Listagem 5.7: astMicro04

```
Ast.programa option =
Some
  (Programa
    [Funcao
      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
       fn_corpo =
        [CmdDeclaracao ((VarSimples "intervalo", TipoInt), Some (ExpInt 0))
           ;
         CmdDeclaracao ((VarSimples "x", TipoInt), None);
         CmdDeclaracao ((VarSimples "num", TipoInt), None);
         CmdFor (VarSimples "x", (ExpInt 0, ExpInt 5),
          [CmdPrint [ExpStr "Digite o numero: "];
           CmdInput (VarSimples "num", (None, TipoInt));
           CmdIf (ExpComp (MaiorIgual, ExpVar (VarSimples "num"), ExpInt
              10),
            [CmdIf
              (ExpComp (MenorIgual, ExpVar (VarSimples "num"), ExpInt 150),
              [CmdAtrib (VarSimples "intervalo",
                ExpOperB (Soma, ExpVar (VarSimples "intervalo"), ExpInt 1))
                  ],
              None)],
            None)]);
         CmdPrint
          [ExpStr "Ao total, foram digitados ";
           ExpVar (VarSimples "intervalo");
           ExpStr " numeros no intervalo entre 10 e 150\n"];
         CmdReturn None]};
     Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.8: astMicro05

```
Ast.programa option =
Some
  (Programa
    [Funcao
      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
       fn_corpo =
        [CmdDeclaracao ((VarSimples "x", TipoInt), None);
         CmdDeclaracao ((VarSimples "m", TipoInt), Some (ExpInt 0));
         CmdDeclaracao ((VarSimples "h", TipoInt), Some (ExpInt 0));
         CmdDeclaracao ((VarSimples "nome", TipoStr), Some (ExpStr ""));
         CmdDeclaracao ((VarSimples "sexo", TipoStr), Some (ExpStr ""));
         CmdFor (VarSimples "x", (ExpInt 0, ExpInt 1),
          [CmdInput (VarSimples "nome",
            (Some (ExpStr "Digite o nome: "), TipoStr));
           CmdInput (VarSimples "sexo",
            (Some (ExpStr "H - Homem ou M - Mulher: "), TipoStr));
           CmdIf
            (ExpComp (Equivalente, ExpVar (VarSimples "sexo"), ExpChar 'H')
              ,
            [CmdAtrib (VarSimples "h",
              ExpOperB (Soma, ExpVar (VarSimples "h"), ExpInt 1))],
            Some
             (CmdIf
               (ExpComp (Equivalente, ExpVar (VarSimples "sexo"), ExpChar '
                  M'),
               [CmdAtrib (VarSimples "m",
                 ExpOperB (Soma, ExpVar (VarSimples "m"), ExpInt 1))],
```

```
26                Some (CmdElse [CmdPrint [ExpStr "Sexo so pode ser H ou M!\n"
                     ]])))))]);
27          CmdPrint
28           [ExpStr "\nForam inseridos "; ExpVar (VarSimples "h");
29            ExpStr " homens"];
30          CmdPrint
31           [ExpStr "\nForam inseridas "; ExpVar (VarSimples "m");
32            ExpStr " mulheres"];
33          CmdReturn None]};
34      Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.9: astMicro06

```
1  Ast.programa option =
2  Some
3   (Programa
4     [Funcao
5       {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6        fn_corpo =
7         [CmdInputDeclaracao ((VarSimples "numero", TipoInt),
8           (Some (ExpStr "Digite um numero de 1 a 5: "), TipoInt));
9          CmdIf (ExpComp (Equivalente, ExpVar (VarSimples "numero"), ExpInt
                1),
10          [CmdPrint [ExpStr "Um"]],
11          Some
12           (CmdIf
13             (ExpComp (Equivalente, ExpVar (VarSimples "numero"), ExpInt 2)
                 ,
14             [CmdPrint [ExpStr "Dois"]],
15             Some
16              (CmdIf
17                (ExpComp (Equivalente, ExpVar (VarSimples "numero"), ExpInt
                      3),
18                [CmdPrint [ExpStr "Tres"]],
19                Some
20                 (CmdIf
21                   (ExpComp
22                     (Equivalente, ExpVar (VarSimples "numero"), ExpInt 4),
23                   [CmdPrint [ExpStr "Quatro"]],
24                   Some
25                    (CmdIf
26                      (ExpComp
27                        (Equivalente, ExpVar (VarSimples "numero"), ExpInt
                             5),
28                      [CmdPrint [ExpStr "Cinco"]],
29                      Some (CmdElse [CmdPrint [ExpStr "Numero Invalido!!!"
                           ]])))))))))));
30          CmdReturn None]};
31      Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.10: astMicro07

```
1  Ast.programa option =
2  Some
3   (Programa
4     [Funcao
5       {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6        fn_corpo =
7         [CmdDeclaracao ((VarSimples "numero", TipoInt), Some (ExpInt 0));
```

```
8          CmdDeclaracao ((VarSimples "programa", TipoInt), Some (ExpInt 1));
9          CmdDeclaracao ((VarSimples "opc", TipoStr), Some (ExpStr ""));
10         CmdWhile
11          (ExpComp (Equivalente, ExpVar (VarSimples "programa"), ExpInt 1),
12          [CmdInput (VarSimples "numero",
13            (Some (ExpStr "Digite um numero: "), TipoInt));
14           CmdIf (ExpComp (Maior, ExpVar (VarSimples "numero"), ExpInt 0),
15            [CmdPrint [ExpStr "Positivo"]],
16            Some
17             (CmdElse
18               [CmdIf
19                 (ExpComp
20                   (Equivalente, ExpVar (VarSimples "numero"), ExpInt 0),
21                 [CmdPrint [ExpStr "O numero e igual a 0"]],
22                 Some
23                  (CmdIf
24                    (ExpComp (Menor, ExpVar (VarSimples "numero"), ExpInt
                          0),
25                    [CmdPrint [ExpStr "Negativo"]], None)))]));
26          CmdInput (VarSimples "opc",
27           (Some (ExpStr "Deseja Finalizar? (S/N) :"), TipoStr));
28          CmdIf
29           (ExpComp (Equivalente, ExpVar (VarSimples "opc"), ExpStr "S"),
30           [CmdAtrib (VarSimples "programa", ExpInt 0)], None)]);
31         CmdReturn None]};
32     Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.11: astMicro08

```
1 Ast.programa option =
2 Some
3  (Programa
4    [Funcao
5      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6       fn_corpo =
7        [CmdDeclaracao ((VarSimples "numero", TipoInt), Some (ExpInt 1));
8         CmdWhile
9          (ExpOperB (Orlog,
10           ExpComp (Menor, ExpVar (VarSimples "numero"), ExpInt 0),
11           ExpComp (Maior, ExpVar (VarSimples "numero"), ExpInt 0)),
12         [CmdInput (VarSimples "numero",
13           (Some (ExpStr "Digite um numero: "), TipoInt));
14          CmdIf (ExpComp (Maior, ExpVar (VarSimples "numero"), ExpInt 10),
15           [CmdPrint [ExpStr "Numero maior que 10"]],
16           Some (CmdElse [CmdPrint [ExpStr "Numero menor que 10"]]))]);
17         CmdReturn None]};
18     Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.12: astMicro09

```
1 Ast.programa option =
2 Some
3  (Programa
4    [Funcao
5      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6       fn_corpo =
7        [CmdDeclaracao ((VarSimples "novopreco", TipoFloat), None);
8         CmdInputDeclaracao ((VarSimples "preco", TipoFloat),
9          (Some (ExpStr "Digite o preco: "), TipoFloat));
```

```
10          CmdInput (VarSimples "venda",
11           (Some (ExpStr "Digite a venda: "), TipoFloat));
12          CmdIf
13           (ExpOperB (Orlog,
14             ExpComp (Menor, ExpVar (VarSimples "venda"), ExpInt 500),
15             ExpComp (Menor, ExpVar (VarSimples "preco"), ExpInt 30)),
16           [CmdAtrib (VarSimples "novopreco",
17             ExpOperB (Soma, ExpVar (VarSimples "preco"),
18              ExpOperB (Mult, ExpOperB (Div, ExpInt 10, ExpInt 100),
19               ExpVar (VarSimples "preco")))))],
20           Some
21            (CmdIf
22              (ExpOperB (Orlog,
23                ExpOperB (ANDlog,
24                  ExpComp (MaiorIgual, ExpVar (VarSimples "venda"), ExpInt
                        500),
25                  ExpComp (Menor, ExpVar (VarSimples "venda"), ExpInt 1200)),
26                ExpOperB (ANDlog,
27                  ExpComp (MaiorIgual, ExpVar (VarSimples "preco"), ExpInt
                        30),
28                  ExpComp (Menor, ExpVar (VarSimples "preco"), ExpInt 80))),
29              [CmdAtrib (VarSimples "novopreco",
30                ExpOperB (Soma, ExpVar (VarSimples "preco"),
31                 ExpOperB (Mult, ExpOperB (Div, ExpInt 15, ExpInt 100),
32                  ExpVar (VarSimples "preco")))))],
33              Some
34               (CmdIf
35                 (ExpOperB (Orlog,
36                   ExpComp
37                    (MaiorIgual, ExpVar (VarSimples "venda"), ExpInt 1200),
38                   ExpComp (MaiorIgual, ExpVar (VarSimples "preco"), ExpInt
                          80)),
39                 [CmdAtrib (VarSimples "novopreco",
40                   ExpOperB (Sub, ExpVar (VarSimples "preco"),
41                    ExpOperB (Mult, ExpOperB (Div, ExpInt 20, ExpInt 100),
42                     ExpVar (VarSimples "preco")))))],
43                 None)))));
44          CmdPrint [ExpStr "O novo preco e:"; ExpVar (VarSimples "novopreco"
               )];
45          CmdReturn None]};
46      Cmd (CmdChmd (ExpChmd (VarSimples "main", []))))]
```

Listagem 5.13: astMicro10

```
1 Ast.programa option =
2 Some
3  (Programa
4    [Funcao
5      {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6       fn_corpo =
7        [CmdInputDeclaracao ((VarSimples "numero", TipoInt),
8          (Some (ExpStr "Digite um numero: "), TipoInt));
9         CmdDeclaracao ((VarSimples "fat", TipoInt),
10          Some
11           (ExpChmd (VarSimples "fatorial", [ExpVar (VarSimples "numero")])
                ));
12         CmdPrint [ExpStr "O fatorial eh"; ExpVar (VarSimples "fat")];
13         CmdReturn None]};
14     Funcao
```

```
15      {fn_nome = VarSimples "fatorial"; fn_tiporet = TipoInt;
16       fn_formais = [(VarSimples "n", TipoInt)];
17       fn_corpo =
18        [CmdIf (ExpComp (MenorIgual, ExpVar (VarSimples "n"), ExpInt 0),
19          [CmdReturn (Some (ExpInt 1))],
20          Some
21           (CmdElse
22             [CmdReturn
23               (Some
24                 (ExpOperB (Mult, ExpVar (VarSimples "n"),
25                   ExpChmd (VarSimples "fatorial",
26                    [ExpOperB (Sub, ExpVar (VarSimples "n"), ExpInt 1)])))
                     ]))]};
27     Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Listagem 5.14: astMicro11

```
1  Ast.programa option =
2  Some
3   (Programa
4     [Funcao
5       {fn_nome = VarSimples "main"; fn_tiporet = TipoNone; fn_formais = [];
6        fn_corpo =
7         [CmdInputDeclaracao ((VarSimples "numero", TipoInt),
8           (Some (ExpStr "Digite um numero: "), TipoInt));
9          CmdDeclaracao ((VarSimples "x", TipoInt),
10          Some
11           (ExpChmd (VarSimples "verifica", [ExpVar (VarSimples "numero")])
                ));
12         CmdIf (ExpComp (Equivalente, ExpVar (VarSimples "x"), ExpInt 1),
13          [CmdPrint [ExpStr "Positivo"]],
14          Some
15           (CmdIf (ExpComp (Equivalente, ExpVar (VarSimples "x"), ExpInt 0)
                  ,
16             [CmdPrint [ExpStr "zero"]],
17             Some (CmdElse [CmdPrint [ExpStr "Negativo"]])))));
18         CmdReturn None]};
19     Funcao
20      {fn_nome = VarSimples "verifica"; fn_tiporet = TipoInt;
21       fn_formais = [(VarSimples "n", TipoInt)];
22       fn_corpo =
23        [CmdDeclaracao ((VarSimples "res", TipoInt), None);
24         CmdIf (ExpComp (Maior, ExpVar (VarSimples "n"), ExpInt 0),
25          [CmdAtrib (VarSimples "res", ExpInt 1)],
26          Some
27           (CmdIf (ExpComp (Menor, ExpVar (VarSimples "n"), ExpInt 0),
28             [CmdAtrib (VarSimples "res", ExpInt (-1))],
29             Some (CmdElse [CmdAtrib (VarSimples "res", ExpInt 0)])))));
30         CmdReturn (Some (ExpVar (VarSimples "res")))]};
31     Cmd (CmdChmd (ExpChmd (VarSimples "main", [])))])
```

Capítulo 6

Análise semantica

O analisador semântico dará signicado as instruções, além de ocorrer a validação de diversas regras da linguagem. A análise semântica percorre a árvore sintática relaciona os identicadores com seus dependentes de acordo com a estrutura hierárquica. Essa etapa também captura informações sobre o programa fonte para que as fases subsequentes gerar o código objeto, um importante componente da analise semântica é a vericação de tipos, nela o compilador verica se cada operador recebe os operandos permitidos e especicados na linguagem fonte. Outra ação realizada na análise semântica é diferenciar os escopos entre global e local, para utilizar corretamente os elementos no código.

6.1 Execução

Para execução, digite no terminal:

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

E então abra o Ocaml:

```
rlwrap ocaml
```

6.2 Codigos

Algumas alterações foram feitas nos arquivos passados:

Listagem 6.1: sintatico.mly

```
1 %{
2 open Lexing
3 open Ast
4 open Sast
5 %}
6
7 %token <int * Lexing.position>      LITINT
```

```
%token <float * Lexing.position>    LITFLOAT
%token <char * Lexing.position>     LITCHAR
%token <string * Lexing.position>   LITSTRING
%token <bool * Lexing.position>   LITBOOL
%token <string * Lexing.position>   ID

/* Tipos */
%token <Lexing.position> INTEIRO
%token <Lexing.position> FLOAT
%token <Lexing.position> CHAR
%token <Lexing.position> STRING
%token <Lexing.position> BOOL
%token <Lexing.position> NONE

/* símbolos */
%token <Lexing.position> VIRG
%token <Lexing.position> DPONTOS
%token <Lexing.position> SETA
%token <Lexing.position> APAR
%token <Lexing.position> FPAR
%token <Lexing.position> ATRIB

/* palavras reservadas */
%token <Lexing.position> IF
%token <Lexing.position> ELIF
%token <Lexing.position> ELSE
%token <Lexing.position> WHILE
%token <Lexing.position> FOR
%token <Lexing.position> IN
%token <Lexing.position> RANGE
%token <Lexing.position> PRINT
%token <Lexing.position> INPUT
%token <Lexing.position> NOT
%token <Lexing.position> ELOG
%token <Lexing.position> OULOG
%token <Lexing.position> DEF
%token <Lexing.position> RETURN

/* operadores relacionais */
%token <Lexing.position>  EQUIVALENTE
%token  <Lexing.position> NAOEQUIVALENTE
%token  <Lexing.position> MENOR
%token  <Lexing.position> MAIOR
%token  <Lexing.position> MENORIGUAL
%token  <Lexing.position> MAIORIGUAL

/* operadores aritmeticos */
%token <Lexing.position> SOMA
%token <Lexing.position> SUB
%token <Lexing.position> MULT
%token <Lexing.position> DIV
%token <Lexing.position> MOD

/* leiaute */
%token  INDENTA
%token  DEDENTA
%token  NOVALINHA

/* eof */
```

```
%token EOF

/* DEFINIÇÃO DA ASSOCIATIVIDADE */
%left NOT
%left ELOG
%left OULOG

%left EQUIVALENTE
%left NAOEQUIVALENTE
%left MENOR
%left MAIOR
%left MENORIGUAL
%left MAIORIGUAL

%left SOMA
%left SUB
%left MULT
%left DIV
%left MOD

%start <Sast.expressao Ast.programa> programa

%%

programa: ins=instrucao*
          EOF
          { Programa ins }

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

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

parametro: nome=variavel DPONTOS t=tipo { (nome, t) }

comandos: cmd=comando+ { cmd }

comando: c=comando_instancia        { c }
       | c=comando_declara          { c }
       | c=comando_atribuicao       { c }
       | c=comando_input_dec_atr    { c }
       | c=comando_input_atr        { c }
       | c=comando_print            { c }
       | c=comando_se               { c }
       | c=comando_while            { c }
       | c=comando_for_dec          { c }
```

```
126         | c=comando_for               { c }
127         | c=comando_chamada           { c }
128         | c=comando_retorno           { c }
129
130 comando_instancia: v=parametro NOVALINHA {
131   CmdDeclara (v)
132 }
133
134 comando_declara: p=parametro ATRIB e=expressao NOVALINHA {
135   CmdDeclaraAtrib (p, e)
136 }
137
138 comando_atribuicao: v=variavel ATRIB e=expressao NOVALINHA {
139   CmdAtrib (v, e)
140 }
141
142 comando_input_dec_atr: p=parametro ATRIB t=tipo APAR INPUT APAR e=
      separated_list(VIRG, expressao) FPAR FPAR NOVALINHA {
143   CmdInputDecAtr (p,e,t)
144 }
145
146 comando_input_atr: v=variavel ATRIB t=tipo APAR INPUT APAR e=
      separated_list(VIRG, expressao) FPAR FPAR NOVALINHA {
147   CmdInputAtr (v,e,t)
148 }
149
150 comando_print: PRINT APAR args=separated_list(VIRG, expressao) FPAR
      NOVALINHA {
151   CmdPrint ( args )
152 }
153
154 comando_se: IF cond=expressao DPONTOS NOVALINHA INDENTA entao=comandos
      DEDENTA cmd1=option(comando_se2) {
155   CmdIf (cond, entao, cmd1)
156 }
157
158 comando_se2: ELSE DPONTOS NOVALINHA INDENTA cmd2=comandos DEDENTA { cmd2 }
159            | ELIF cond1=expressao DPONTOS NOVALINHA INDENTA entao1=
                 comandos DEDENTA cmd1=option(comando_se2) {
160                [CmdIf (cond1, entao1, cmd1)]
161              }
162
163 comando_while: WHILE cond=expressao DPONTOS NOVALINHA INDENTA cmd=comandos
       DEDENTA {
164   CmdWhile (cond, cmd)
165 }
166
167 comando_for_dec: FOR p=parametro IN RANGE APAR n1=expressao VIRG n2=
      expressao FPAR DPONTOS NOVALINHA INDENTA cmd=comandos DEDENTA {
168   CmdFor_Dec (p, (n1, n2), cmd)
169 }
170
171 comando_for: FOR v=variavel IN RANGE APAR n1=expressao VIRG n2=expressao
      FPAR DPONTOS NOVALINHA INDENTA cmd=comandos DEDENTA {
172   CmdFor (v, (n1, n2), cmd)
173 }
174
175 comando_chamada: exp=expressao NOVALINHA {
176   CmdExprs exp
```

```
177 }
178
179 comando_retorno: RETURN e=option(expressao) NOVALINHA { CmdReturn e }
180
181 chamada : nome=ID APAR args=separated_list(VIRG, expressao) FPAR { ExpChmd
        (nome, args) }
182
183 tipo: INTEIRO { TipoInt }
184     | STRING  { TipoStr }
185     | BOOL    { TipoBool }
186     | CHAR    { TipoChar }
187     | FLOAT   { TipoFloat }
188     | NONE    { TipoNone }
189
190 expressao:
191       f=chamada   {          f }
192     | v=variavel  { ExpVar   v }
193     | i=LITINT    { ExpInt   i }
194     | c=LITCHAR   { ExpChar  c }
195     | f=LITFLOAT  { ExpFloat f }
196     | s=LITSTRING { ExpStr   s }
197     | b=LITBOOL   { ExpBool  b }
198     | l=expressao op=oper r=expressao { ExpOperB( op,l,r) }
199     | l=expressao op=oper_un { ExpOperU( op,l) }
200     | APAR e=expressao FPAR { e }
201
202 %inline oper:
203       pos = ELOG              { (Elog,pos)   }
204     | pos = OULOG             { (Oulog,pos)  }
205     | pos = SOMA              { (Mais,pos)   }
206     | pos = SUB               { (Sub,pos)    }
207     | pos = MULT              { (Mult,pos)   }
208     | pos = DIV               { (Div,pos)    }
209     | pos = MOD               { (Mod,pos)    }
210     | pos = EQUIVALENTE       { (Igual, pos) }
211     | pos = NAOEQUIVALENTE    { (Difer, pos) }
212     | pos = MENOR             { (Menor, pos) }
213     | pos = MAIOR             { (Maior, pos) }
214     | pos = MENORIGUAL        { (MenorIgual, pos) }
215     | pos = MAIORIGUAL        { (MaiorIgual, pos) }
216
217 %inline oper_un:
218     | pos = NOT { ((Not, pos)) }
219
220 variavel:
221     | x = ID {VarSimples x }
```

Listagem 6.2: ast.ml

```
1  open Lexing
2
3  type identificador = string
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
14  and 'expr decfn = {
15    fn_nome:    variavel;
16    fn_tiporet: tipo;
17    fn_formais: (variavel * tipo) list;
18    fn_corpo:  'expr comandos
19  }
20
21  and tipo =
22      TipoInt
23    | TipoStr
24    | TipoBool
25    | TipoChar
26    | TipoFloat
27    | TipoNone
28
29  and 'expr comando =
30      CmdDeclara of parametro
31    | CmdDeclaraAtrib of parametro * ('expr )
32    | CmdAtrib  of variavel * ('expr )
33    | CmdInputDecAtr of parametro * ('expr  list) * tipo
34    | CmdInputAtr of variavel * ('expr  list) * tipo
35    | CmdPrint  of ('expr ) list
36    | CmdIf     of ('expr ) * ('expr comandos) * (('expr comandos) option)
37    | CmdReturn of ('expr ) option
38    | CmdWhile  of ('expr ) * ('expr comandos)
39    | CmdFor    of variavel * (('expr ) * ('expr )) * ('expr comandos)
40    | CmdFor_Dec of parametro * (('expr ) * ('expr )) * ('expr comandos)
41    | CmdExprs   of ('expr )
42
43  and variaveis = variavel list
44
45  and parametro = variavel * tipo
46
47  and variavel = VarSimples of identificador pos
48
49  and operador =
50      Mais
51    | Sub
52    | Mult
53    | Div
54    | Mod
55    | Elog
56    | Oulog
57    | Maior
58    | Menor
59    | MaiorIgual
60    | MenorIgual
61    | Igual
62    | Difer
63
64  and operador_unario =
65        Not
```

Listagem 6.3: sast.ml

```
1  open Ast
```

```
2
3 type expressao =
4   | ExpVar   of variavel
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_unario pos * expressao
12  | ExpChmd  of identificador pos * (expressao expressoes)
```

Listagem 6.4: tast.ml

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

Listagem 6.5: semantico.ml

```
1 module Amb = Ambiente
2 module A = Ast
3 module S = Sast
4 module T = Tast
5
6 let posicao exp = let open S in
7   match exp with
8   | ExpVar v -> (match v with
9       | A.VarSimples            (_,pos)         -> pos
10    )
11  | ExpInt                      (_,pos)         -> pos
12  | ExpStr                      (_,pos)         -> pos
13  | ExpChar                     (_,pos)         -> pos
14  | ExpBool                     (_,pos)         -> pos
15  | ExpFloat                    (_,pos)         -> pos
16  | ExpOperB                    ((_,pos),_,_)   -> pos
17  | ExpChmd                  ((_,pos), _)     -> pos
18
19 type classe_op = Aritmetico | Relacional | Logico | Cadeia
20
21 let classifica op =
22   let open A in
23   match op with
24     Oulog
25   | Elog  -> Logico
26   | Menor
27   | Maior
28   | Igual
29   | MaiorIgual
```

```
30    | MenorIgual
31    | Difer -> Relacional
32    | Mais
33    | Sub
34    | Mult
35    | Mod
36    | Div -> Aritmetico
37
38  let msg_erro_pos pos msg =
39    let open Lexing in
40    let lin = pos.pos_lnum
41    and col = pos.pos_cnum - pos.pos_bol - 1 in
42    Printf.sprintf "Semantico -> linha %d, coluna %d: %s" lin col msg
43
44  let msg_erro nome msg =
45    let pos = snd nome in
46    msg_erro_pos pos msg
47
48  let insere_declaracao_var amb dec =
49    let open A in
50      match dec with
51           (nome, tipo) ->  Amb.insere_local amb (fst nome) tipo
52
53  let nome_tipo t =
54    let open A in
55      match t with
56        TipoInt     -> "inteiro"
57      | TipoStr     -> "string"
58      | TipoBool    -> "bool"
59      | TipoNone    -> "none"
60      | TipoChar    -> "char"
61      | TipoFloat   -> "float"
62
63  let mesmo_tipo pos msg tinf tdec =
64    if tinf <> tdec
65    then
66      let msg = Printf.sprintf msg (nome_tipo tinf) (nome_tipo tdec) in
67      failwith (msg_erro_pos pos msg)
68
69  let rec infere_exp amb exp =
70    match exp with
71      S.ExpInt n      -> (T.ExpInt    (fst n, A.TipoInt),       A.TipoInt)
72    | S.ExpStr s      -> (T.ExpStr    (fst s, A.TipoStr),       A.TipoStr)
73    | S.ExpBool b     -> (T.ExpBool   (fst b, A.TipoBool),      A.TipoBool)
74    | S.ExpChar c     -> (T.ExpChar   (fst c, A.TipoChar),      A.TipoChar)
75    | S.ExpFloat f    -> (T.ExpFloat  (fst f, A.TipoFloat),     A.TipoFloat)
76    | S.ExpVar v      ->
77      (match v with
78         A.VarSimples nome ->
79         let id = fst nome in
80           (try (match (Amb.busca amb id) with
81                 | Amb.EntVar tipo -> (T.ExpVar (A.VarSimples nome, tipo),
                       tipo)
82                 | Amb.EntFun _ ->
83                   let msg = "nome de funcao usado como nome de variavel: "
                         ^ id in
84                    failwith (msg_erro nome msg)
85              )
86           with Not_found ->
```

```
87                  let msg = "A variavel " ^ id ^ " nao foi declarada" in
88                  failwith (msg_erro nome msg)
89          )
90       | _ -> failwith "infere_exp: não implementado"
91      )
92      (* FAZER ExpOperU e verifica aritmetico operU*)
93    | S.ExpOperB (op, esq, dir) ->
94      let (esq, tesq) = infere_exp amb esq
95      and (dir, tdir) = infere_exp amb dir in
96
97      let verifica_aritmetico () =
98        (match tesq with
99          A.TipoFloat
100          | A.TipoInt ->
101          let _ = mesmo_tipo (snd op)
102                      "O operando esquerdo eh do tipo %s mas o direito eh
                           do tipo %s"
103                      tesq tdir
104          in tesq (* O tipo da expressão aritmética como um todo *)
105
106        | t -> let msg = "um operador aritmetico nao pode ser usado com o
             tipo " ^
107                          (nome_tipo t)
108          in failwith (msg_erro_pos (snd op) msg)
109        )
110
111      and verifica_relacional () =
112        (match tesq with
113           A.TipoInt
114         | A.TipoFloat
115         | A.TipoChar
116         | A.TipoStr ->
117          let _ = mesmo_tipo (snd op)
118                    "O operando esquerdo eh do tipo %s mas o direito eh do
                         tipo %s"
119                    tesq tdir
120          in A.TipoBool (* O tipo da expressão relacional é sempre booleano
                *)
121
122        | t -> let msg = "um operador relacional nao pode ser usado com o
             tipo " ^
123                          (nome_tipo t)
124          in failwith (msg_erro_pos (snd op) msg)
125        )
126
127      and verifica_logico () =
128        (match tesq with
129           A.TipoBool ->
130          let _ = mesmo_tipo (snd op)
131                    "O operando esquerdo eh do tipo %s mas o direito eh do
                         tipo %s"
132                    tesq tdir
133          in A.TipoBool (* O tipo da expressão lógica é sempre booleano *)
134
135        | t -> let msg = "um operador logico nao pode ser usado com o tipo
             " ^
136                          (nome_tipo t)
137              in failwith (msg_erro_pos (snd op) msg)
138        )
```

```
139     and verifica_cadeia () =
140       (match tesq with
141          A.TipoStr ->
142          let _ = mesmo_tipo (snd op)
143                    "O operando esquerdo eh do tipo %s mas o direito eh do
                          tipo %s"
144                    tesq tdir
145          in A.TipoStr (* O tipo da expressão relacional é sempre string *)
146        | A.TipoChar ->
147          let _ = mesmo_tipo (snd op)
148                    "O operando esquerdo eh do tipo %s mas o direito eh do
                          tipo %s"
149                    tesq tdir
150          in A.TipoStr (* O tipo da expressão relacional é sempre string *)
151
152        | t -> let msg = "um operador relacional nao pode ser usado com o
              tipo " ^
153                          (nome_tipo t)
154               in failwith (msg_erro_pos (snd op) msg)
155       )
156
157     in
158     let op = fst op in
159     let tinf = (match (classifica op) with
160           Aritmetico -> verifica_aritmetico ()
161         | Relacional -> verifica_relacional ()
162         | Logico -> verifica_logico ()
163         | Cadeia -> verifica_cadeia ()
164       )
165     in
166       (T.ExpOperB ((op,tinf), (esq, tesq), (dir, tdir)), tinf)
167
168   | S.ExpChmd(nome, args) ->
169      let rec verifica_parametros ags ps fs =
170         match (ags, ps, fs) with
171           (a::ags), (p::ps), (f::fs) ->
172              let _ = mesmo_tipo (posicao a)
173                        "O parametro eh do tipo %s mas deveria ser do tipo %s
                              " p f
174              in verifica_parametros ags ps fs
175         | [], [], [] -> ()
176         | _ -> failwith (msg_erro nome "Numero incorreto de parametros")
177      in
178      let id = fst nome in
179      try
180        begin
181          let open Amb in
182
183          match (Amb.busca amb id) with
184          (* verifica se 'nome' está associada a uma função *)
185            Amb.EntFun {tipo_fn; formais} ->
186            (* Infere o tipo de cada um dos argumentos *)
187            let argst = List.map (infere_exp amb) args
188            (* Obtem o tipo de cada parâmetro formal *)
189            and tipos_formais = List.map snd formais in
190            (* Verifica se o tipo de cada argumento confere com o tipo
                  declarado *)
191            (* do parâmetro formal correspondente.
                                                  *)
```

```
192             let _ = verifica_parametros args (List.map snd argst)
                    tipos_formais
193              in (T.ExpChmd (id, (List.map fst argst), tipo_fn), tipo_fn)
194           | Amb.EntVar _ -> (* Se estiver associada a uma variável, falhe
                 *)
195             let msg = id ^ " eh uma variavel e nao uma funcao" in
196             failwith (msg_erro nome msg)
197         end
198       with Not_found ->
199         let msg = "Nao existe a funcao de nome " ^ id in
200         failwith (msg_erro nome msg)
201
202 (* FAZER função para inferir tipo *)
203
204 let  infere_var amb exp =
205   match exp with
206         A.VarSimples nome ->(*MESMA COISA DO INFERE EXP, verificar com
              professor se é assim que faz*)
207         let id = fst nome in
208           (try (match (Amb.busca amb id) with
209                 | Amb.EntVar tipo -> (A.VarSimples nome, tipo)
210                 | Amb.EntFun _ ->
211                   let msg = "nome de funcao usado como nome de variavel: "
                          ^ id in
212                    failwith (msg_erro nome msg)
213              )
214            with Not_found ->
215                   let msg = "A variavel " ^ id ^ " nao foi declarada" in
216                   failwith (msg_erro nome msg)
217           )
218
219 let rec verifica_cmd amb tiporet cmd =
220   let open A in
221   match cmd with
222     CmdReturn exp ->
223     (match exp with
224      (* Se a função não retornar nada, verifica se ela foi declarada como
             void *)
225        None ->
226        let _ = mesmo_tipo (Lexing.dummy_pos)
227                    "O tipo retornado eh %s mas foi declarado como %s"
228                    TipoNone tiporet
229        in CmdReturn None
230      | Some e ->
231        (* Verifica se o tipo inferido para a expressão de retorno confere
              com o *)
232        (* tipo declarado para a função.
                                                          *)
233            let (e1,tinf) = infere_exp amb e in
234            let _ = mesmo_tipo (posicao e)
235                               "O tipo retornado eh %s mas foi declarado
                                     como %s"
236                               tinf tiporet
237            in CmdReturn (Some e1)
238       )
239   | CmdIf (teste, entao, senao) ->
240     let (teste1,tinf) = infere_exp amb teste in
241     (* O tipo inferido para a expressão 'teste' do condicional deve ser
          booleano *)
```

```
242     let _ = mesmo_tipo (posicao teste)
243               "O teste do if deveria ser do tipo %s e nao %s"
244               TipoBool tinf in
245     (* Verifica a validade de cada comando do bloco 'então' *)
246     let entao1 = List.map (verifica_cmd amb tiporet) entao in
247     (* Verifica a validade de cada comando do bloco 'senão', se houver *)
248     let senao1 =
249         match senao with
250           None -> None
251         | Some bloco -> Some (List.map (verifica_cmd amb tiporet) bloco)
252      in
253      CmdIf (teste1, entao1, senao1)
254
255   | CmdAtrib (elem, exp) ->
256     (* Infere o tipo da expressão no lado direito da atribuição *)
257     let (exp,  tdir) = infere_exp amb exp
258     (* Faz o mesmo para o lado esquerdo *)
259     and (elem1, tesq) = infere_var amb elem in
260     (* Os dois tipos devem ser iguais *)
261     let nome_elem = match elem with A.VarSimples a -> a in
262     let _ = mesmo_tipo (snd nome_elem)
263                        "Atribuicao com tipos diferentes: %s = %s" tesq
                              tdir
264     in CmdAtrib (elem1, exp)
265
266   | CmdExprs exp ->
267      let (exp,tinf) = infere_exp amb exp in
268      CmdExprs exp
269
270   | CmdDeclaraAtrib((elem,tipo), exp) ->
271     let var = match elem with A.VarSimples a -> a in
272     let _ = insere_declaracao_var amb (var,tipo) in
273     (* Infere o tipo da expressão no lado direito da atribuição *)
274     let (exp,  tdir) = infere_exp amb exp
275     (* Faz o mesmo para o lado esquerdo *)
276     and (elem1, tesq) = infere_var amb elem in
277     (* Os dois tipos devem ser iguais *)
278     let nome_elem = match elem with A.VarSimples a -> a in
279     let _ = mesmo_tipo (snd nome_elem)
280                        "Atribuicao com tipos diferentes: %s = %s" tesq
                              tdir
281     in CmdDeclaraAtrib ((elem1,tipo), exp)
282
283   | CmdPrint exps ->
284     (* Verifica o tipo de cada argumento da função 'entrada' *)
285     let exps = List.map (infere_exp amb) exps in
286     CmdPrint (List.map fst exps)
287
288   | CmdDeclara(elem,tipo) ->
289     let var = match elem with A.VarSimples a -> a in
290     let _ = insere_declaracao_var amb (var,tipo) in
291     CmdDeclara (elem,tipo)
292
293   |CmdInputDecAtr ((elem,tipo), exps, tipof) ->
294     let var = match elem with A.VarSimples a -> a in
295     let _ = insere_declaracao_var amb (var,tipo) in
296     (* Infere o tipo da expressão no lado direito da atribuição *)
297     let exps = List.map (infere_exp amb) exps in
298     let exps = List.map (fst) exps in
```

```
299     (* Faz o mesmo para o lado esquerdo *)
300     let (elem1, tesq) = infere_var amb elem in
301     (* Os dois tipos devem ser iguais *)
302     let _ = mesmo_tipo (snd var)
303                         "Atribuicao com tipos diferentes: %s = %s" tipo
                               tipof
304     in CmdInputDecAtr ((elem1,tipo), exps, tipof)
305
306   |CmdInputAtr (elem, exps, tipof) ->
307     let var = match elem with A.VarSimples a -> a in
308     (* Infere o tipo da expressão no lado direito da atribuição *)
309     let exps = List.map (infere_exp amb) exps in
310     let exps = List.map (fst) exps in
311     (* Faz o mesmo para o lado esquerdo *)
312     let (elem1, tesq) = infere_var amb elem in
313     (* Os dois tipos devem ser iguais *)
314     let _ = mesmo_tipo (snd var)
315                         "Atribuicao com tipos diferentes: %s = %s" tesq
                               tipof
316     in CmdInputAtr (elem1, exps, tipof)
317
318   | CmdWhile (teste, entao) ->
319     let (teste1,tinf) = infere_exp amb teste in
320     (* O tipo inferido para a expressão 'teste' do condicional deve ser
           booleano *)
321     let _ = mesmo_tipo (posicao teste)
322               "O teste do while deveria ser do tipo %s e nao %s"
323               TipoBool tinf in
324     (* Verifica a validade de cada comando do bloco 'então' *)
325     let entao1 = List.map (verifica_cmd amb tiporet) entao in
326      CmdWhile (teste1, entao1)
327
328   | CmdFor (variavel, (de, para), entao) ->
329     let (var1, tesq) = infere_var amb variavel in
330     let pos = posicao (ExpVar variavel) in
331     let (de1,tde) = infere_exp amb de and
332     (para1,tpara) = infere_exp amb para in
333     (* O tipo inferido para a expressão 'teste' do condicional deve ser
           booleano *)
334     let _ = mesmo_tipo (pos)
335               "A variavel do for deveria ser do tipo %s e nao %s"
336               TipoInt tesq in
337     let _ = mesmo_tipo (pos)
338               "A inicializacao do for deveria ser do tipo %s e nao %s"
339               TipoInt tde in
340     let _ = mesmo_tipo (pos)
341               "A finalizacao do for deveria ser do tipo %s e nao %s"
342               TipoInt tpara in
343     (* Verifica a validade de cada comando do bloco 'então' *)
344     let entao1 = List.map (verifica_cmd amb tiporet) entao in
345      CmdFor (var1, (de1, para1), entao1)
346
347   | CmdFor_Dec ((variavel,tipo), (de, para), entao) ->
348     let var = match variavel with A.VarSimples a -> a in
349     let _ = insere_declaracao_var amb (var,tipo) in
350     let (var1, tesq) = infere_var amb variavel in
351     let pos = posicao (ExpVar variavel) in
352     let (de1,tde) = infere_exp amb de and
353     (para1,tpara) = infere_exp amb para in
```

```
354     (* O tipo inferido para a expressão 'teste' do condicional deve ser
           booleano *)
355     let _ = mesmo_tipo (pos)
356               "A variavel do for deveria ser do tipo %s e nao %s"
357               TipoInt tesq in
358     let _ = mesmo_tipo (pos)
359               "A inicializacao do for deveria ser do tipo %s e nao %s"
360               TipoInt tde in
361     let _ = mesmo_tipo (pos)
362               "A finalizacao do for deveria ser do tipo %s e nao %s"
363               TipoInt tpara in
364     (* Verifica a validade de cada comando do bloco 'então' *)
365     let entao1 = List.map (verifica_cmd amb tiporet) entao in
366      CmdFor_Dec ((var1,tipo), (de1, para1), entao1)
367
368 (*and verifica_fun amb ast =
369   let open A in
370   match ast with
371     | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
372     (* Estende o ambiente global, adicionando um ambiente local *)
373     let ambfn = Amb.novo_escopo amb in
374     (* Insere os parâmetros no novo ambiente *)
375     let insere_parametro (v,t) = Amb.insere_param ambfn (fst v) t in
376     let _ = List.iter insere_parametro fn_formais in
377     (* Insere as variáveis locais no novo ambiente *)
378     let insere_local = function
379         (DecVar (v,t)) -> Amb.insere_local ambfn (fst v)  t in
380     let _ = List.iter insere_local fn_locais in
381     (* Verifica cada comando presente no corpo da função usando o novo
           ambiente *)
382     let corpo_tipado = List.map (verifica_cmd ambfn fn_tiporet) fn_corpo
           in
383       Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo = corpo_tipado}
384     | Cmd _ -> failwith "Instrucao invalida"*)
385
386 and verifica_fun amb ast =
387   let open A in
388   match ast with
389   | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
390     (* Estende o ambiente global, adicionando um ambiente local *)
391     let ambfn = Amb.novo_escopo amb in
392     (* Insere os parâmetros no novo ambiente *)
393     let insere_parametro (v,t) = Amb.insere_param ambfn (fst v) t in
394     let fn_formaisn = (List.map
395     (fun fn_formal -> (match fn_formal with (A.VarSimples a, tipo) -> (a,
           tipo)) )
396      fn_formais) in
397     let _ = List.iter insere_parametro fn_formaisn in
398     (* Verifica cada comando presente no corpo da função usando o novo
           ambiente *)
399     let corpo_tipado = List.map (verifica_cmd ambfn fn_tiporet) fn_corpo
           in
400       Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo = corpo_tipado}
401   | Cmd _ -> failwith "Instrucao invalida"
402
403 let rec verifica_dup xs =
404   match xs with
405     [] -> []
406   | (nome,t)::xs ->
```

```
407     let id = fst nome in
408     if (List.for_all (fun (n,t) -> (fst n) <> id) xs)
409     then (id, t) :: verifica_dup xs
410     else let msg = "Parametro duplicado " ^ id in
411       failwith (msg_erro nome msg)
412
413 let insere_declaracao_fun amb dec =
414   let open A in
415     match dec with
416       | Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
417         (* Verifica se não há parâmetros duplicados *)
418         let fn_formaisn = (List.map
419     (fun fn_formal -> (match fn_formal with (A.VarSimples a, tipo) -> (a,
          tipo)) )
420      fn_formais) in
421         let formais = verifica_dup fn_formaisn in
422         let fn_nome = (match fn_nome with (A.VarSimples (a, tipo)) -> (a,
             tipo)) in
423         let nome = fst fn_nome in
424         Amb.insere_fun amb nome formais fn_tiporet
425       | Cmd _ -> failwith "Instrucao invalida"
426
427
428 (* Lista de cabeçalhos das funções pré definidas *)
429 let fn_predefs = let open A in [
430    ("entrada", [("x", TipoInt); ("y", TipoInt)], TipoNone);
431    ("saida",   [("x", TipoInt); ("y", TipoInt)], TipoNone)
432 ]
433
434 (* insere as funções pré definidas no ambiente global *)
435 let declara_predefinidas amb =
436   List.iter (fun (n,ps,tr) -> Amb.insere_fun amb n ps tr) fn_predefs
437
438 let semantico ast =
439   let amb_global = Amb.novo_amb [] in
440   let _ = declara_predefinidas amb_global in
441   let A.Programa instr = ast in
442   let decs_funs = List.filter(fun x ->
443     (match x with
444     | A.Funcao _ -> true
445     | _          -> false)) instr in
446     let _ = List.iter (insere_declaracao_fun amb_global) decs_funs in
447       let decs_funs = List.map (verifica_fun amb_global) decs_funs in
448       (A.Programa decs_funs, amb_global)
```

Listagem 6.6: .ocamlinit

```
1  let () =
2    try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
3    with Not_found -> ()
4  ;;
5
6  #use "topfind";;
7  #require "menhirLib";;
8  #directory "_build";;
9  #load "sintatico.cmo";;
10 #load "lexico.cmo";;
11 #load "ast.cmo";;
12 #load "sast.cmo";;
```

```
#load "tast.cmo";;
#load "tabsimb.cmo";;
#load "ambiente.cmo";;
#load "semantico.cmo";;
#load "semanticoTest.cmo";;

open Ast
open Ambiente
open SemanticoTest
```

6.3 Compilação e execução

Gerando mensagens de erro:

```
menhir -v --list-errors sintatico.mly > sintatico.messages
menhir -v --list-errors sintatico.mly --compile-errors sintatico.messages
    > fnmes.ml
```

Compilando o arquivo de teste, digite no terminal:

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

Enfim entre no ocaml e chame o verificador de tipos para obter árvore anotada com os tipos, digite:

```
# verifica_tipos "exemplos/micro10.py";;
```

6.4 Codigo teste

Usado o codigo micro10.py para teste do semântico.

Listagem 6.7: micro10.txt

```
# verifica_tipos "micro10.py";;
- : Tast.expressao Ast.programa * Ambiente.t =
(Programa
  [Funcao
    {fn_nome =
      VarSimples
       ("main",
        {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 4});
     fn_tiporet = TipoNone; fn_formais = [];
     fn_corpo =
      [CmdInputDecAtr
        ((VarSimples
           ("numero",
            {Lexing.pos_fname = ""; pos_lnum = 2; pos_bol = 20;
             pos_cnum = 24}),
          TipoInt),
        [Tast.ExpStr ("Digite um numero: ", TipoStr)], TipoInt);
```

```
18        CmdDeclaraAtrib
19         ((VarSimples
20            ("fat",
21             {Lexing.pos_fname = ""; pos_lnum = 3; pos_bol = 70;
22              pos_cnum = 74}),
23           TipoInt),
24          Tast.ExpChmd ("fatorial",
25           [Tast.ExpVar
26             (VarSimples
27               ("numero",
28                {Lexing.pos_fname = ""; pos_lnum = 3; pos_bol = 70;
29                 pos_cnum = 93}),
30              TipoInt)],
31           TipoInt));
32        CmdPrint
33         [Tast.ExpStr ("O fatorial eh", TipoStr);
34          Tast.ExpVar
35           (VarSimples
36             ("fat",
37              {Lexing.pos_fname = ""; pos_lnum = 4; pos_bol = 101;
38               pos_cnum = 128}),
39            TipoInt)];
40        CmdReturn None]};
41    Funcao
42     {fn_nome =
43       VarSimples
44        ("fatorial",
45         {Lexing.pos_fname = ""; pos_lnum = 7; pos_bol = 145; pos_cnum =
              149});
46      fn_tiporet = TipoInt;
47      fn_formais =
48       [(VarSimples
49          ("n",
50           {Lexing.pos_fname = ""; pos_lnum = 7; pos_bol = 145;
51            pos_cnum = 158}),
52         TipoInt)];
53      fn_corpo =
54       [CmdIf
55         (Tast.ExpOperB ((MenorIgual, TipoBool),
56           (Tast.ExpVar
57             (VarSimples
58               ("n",
59                {Lexing.pos_fname = ""; pos_lnum = 8; pos_bol = 174;
60                 pos_cnum = 181}),
61              TipoInt),
62            TipoInt),
63           (Tast.ExpInt (0, TipoInt), TipoInt)),
64         [CmdReturn (Some (Tast.ExpInt (1, TipoInt)))],
65         Some
66          [CmdReturn
67            (Some
68              (Tast.ExpOperB ((Mult, TipoInt),
69                (Tast.ExpVar
70                  (VarSimples
71                    ("n",
72                     {Lexing.pos_fname = ""; pos_lnum = 11; pos_bol = 216;
73                      pos_cnum = 231}),
74                   TipoInt),
75                 TipoInt),
```

```
76                (Tast.ExpChmd ("fatorial",
77                  [Tast.ExpOperB ((Sub, TipoInt),
78                    (Tast.ExpVar
79                      (VarSimples
80                        ("n",
81                         {Lexing.pos_fname = ""; pos_lnum = 11; pos_bol =
                              216;
82                          pos_cnum = 244}),
83                       TipoInt),
84                     TipoInt),
85                    (Tast.ExpInt (1, TipoInt), TipoInt))],
86                  TipoInt),
87                 TipoInt))))])]}],
88  <abstr>)
```

Capítulo 7

Interprete

Os nano e micro-programas podem ser executados sem a necessidade de gerar um código de máquina, para isso é necessário o interpretador.

7.1 Códigos

Listagem 7.1: ambInterp.ml

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

```
32
33 let insere_fun amb nome params resultado corpo =
34   let ef = EntFun { tipo_fn = resultado;
35                     formais = params;
36                     corpo = corpo }
37   in Tab.insere amb.ambv nome ef
```

Listagem 7.2: 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  (* ExpVar - (VarSimples, identificador) *)
9  let obtem_nome_tipo_var exp amb = let open T in
10   match exp with
11     | A.VarSimples (nome,_) ->
12       let tipo = (match (Amb.busca amb nome) with
13       | Amb.EntVar (tipo, v) ->  tipo
14       |  _ -> failwith "obtem_nome_tipo_var: nao eh variavel"
15       ) in      (nome,tipo)
16
17 (* ExpInt *)
18 let pega_int exp =
19   match exp with
20   |  T.ExpInt (i,_) -> i
21   | _ -> failwith "pega_int: nao eh inteiro"
22
23 (* ExpStr *)
24 let pega_string exp =
25   match exp with
26   |  T.ExpStr (s,_) -> s
27   | _ -> failwith "pega_string: nao eh string"
28
29 (* ExpChar *)
30 let pega_char exp =
31   match exp with
32   |  T.ExpChar (i,_) -> i
33   | _ -> failwith "pega_char: nao eh caracter"
34
35 (* ExpBool *)
36 let pega_bool exp =
37   match exp with
38   |  T.ExpBool (b,_) -> b
39   | _ -> failwith "pega_bool: nao eh booleano"
40
41 (* ExpFloat *)
42 let pega_float exp =
43   match exp with
44   |  T.ExpFloat (i,_) -> i
45   | _ -> failwith "pega_float: nao eh float"
46
47 type classe_op = Aritmetico | Relacional | Logico
48
49 (* operador *)
50 let classifica op =
```

```
51    let open A in
52    match op with
53      Oulog
54    | Elog -> Logico
55    | Menor
56    | Maior
57    | MenorIgual
58    | MaiorIgual
59    | Igual
60    | Difer -> Relacional
61    | Mais
62    | Sub
63    | Mult
64    | Mod
65    | Div -> Aritmetico
66
67  (* operador_unario *)
68  let classificaun op =
69    let open A in
70    match op with
71      | Not -> Logico
72
73  let rec interpreta_exp amb exp =
74  let open A in
75  let open T in
76
77    match exp with
78    | ExpNone
79    | ExpInt _
80    | ExpStr _
81    | ExpChar _
82    | ExpFloat _
83    | ExpBool _    -> exp
84    | ExpVar (v,t) ->
85      let (id,tipo) = obtem_nome_tipo_var v amb  in
86      (match (Amb.busca amb id) with
87       | Amb.EntVar (tipo, v) ->
88         (match v with
89          | None -> failwith ("variável nao inicializada: " ^ id)
90          | Some valor -> valor
91         )
92       |  _ -> failwith "interpreta_exp: expvar"
93      )
94    | ExpOperB ((op,top), (esq, tesq), (dir,tdir)) ->
95      let  vesq = interpreta_exp amb esq
96      and vdir = interpreta_exp amb dir in
97
98      let interpreta_aritmetico () =
99        (match tesq with
100        | TipoInt ->
101          (match op with
102           | Mais  -> ExpInt (pega_int vesq + pega_int vdir, top)
103           | Sub -> ExpInt (pega_int vesq - pega_int vdir, top)
104           | Mult  -> ExpInt (pega_int vesq * pega_int vdir, top)
105           | Div   -> ExpInt (pega_int vesq / pega_int vdir, top)
106           | Mod   -> ExpInt (pega_int vesq mod pega_int vdir, top)
107           | _ -> failwith "interpreta_aritmetico"
108          )
109        | TipoFloat ->
```

```
110           (match op with
111            | Mais  -> ExpFloat (pega_float vesq +. pega_float vdir, top)
112            | Sub -> ExpFloat (pega_float vesq -. pega_float vdir, top)
113            | Mult  -> ExpFloat (pega_float vesq *. pega_float vdir, top)
114            | Div   -> ExpFloat (pega_float vesq /. pega_float vdir, top)
115            | _ -> failwith "interpreta_aritmetico"
116           )
117         | _ -> failwith "interpreta_aritmetico"
118       )
119
120     and interpreta_relacional () =
121       (match tesq with
122        | TipoInt ->
123           (match op with
124            | Menor -> ExpBool (pega_int vesq < pega_int vdir, top)
125            | MenorIgual -> ExpBool (pega_int vesq <= pega_int vdir, top)
126            | Maior  -> ExpBool (pega_int vesq > pega_int vdir, top)
127            | MaiorIgual  -> ExpBool (pega_int vesq >= pega_int vdir, top)
128            | Igual   -> ExpBool (pega_int vesq == pega_int vdir, top)
129            | Difer   -> ExpBool (pega_int vesq != pega_int vdir, top)
130            | _ -> failwith "interpreta_relacional int"
131           )
132        | TipoFloat ->
133           (match op with
134            | Menor -> ExpBool (pega_float vesq < pega_float vdir, top)
135            | MenorIgual -> ExpBool (pega_float vesq <= pega_float vdir, top
                 )
136            | Maior  -> ExpBool (pega_float vesq > pega_float vdir, top)
137            | MaiorIgual  -> ExpBool (pega_float vesq >= pega_float vdir,
                  top)
138            | Igual   -> ExpBool (pega_float vesq == pega_float vdir, top)
139            | Difer   -> ExpBool (pega_float vesq != pega_float vdir, top)
140            | _ -> failwith "interpreta_relacional float"
141           )
142        | TipoStr ->
143           (match op with
144            | Menor -> ExpBool (pega_string vesq < pega_string vdir, top)
145            | Maior  -> ExpBool (pega_string vesq > pega_string vdir, top)
146            | MenorIgual -> ExpBool  ((pega_string vesq < pega_string vdir)
                  || not(pega_string vesq <> pega_string vdir), top)
147            | MaiorIgual  -> ExpBool ((pega_string vesq > pega_string vdir)
                  || not(pega_string vesq <> pega_string vdir), top)
148            | Igual   -> ExpBool (not(pega_string vesq <> pega_string vdir),
                   top)
149            | Difer   -> ExpBool (pega_string vesq <> pega_string vdir, top)
150            | _ -> failwith "interpreta_relacional string"
151           )
152        | TipoChar ->
153           (match op with
154            | Menor -> ExpBool (pega_char vesq < pega_char vdir, top)
155            | Maior  -> ExpBool (pega_char vesq > pega_char vdir, top)
156            | MenorIgual -> ExpBool  ((pega_char vesq < pega_char vdir) ||
                  not(pega_char vesq <> pega_char vdir), top)
157            | MaiorIgual  -> ExpBool ((pega_char vesq > pega_char vdir) ||
                  not(pega_char vesq <> pega_char vdir), top)
158            | Igual   -> ExpBool (not(pega_char vesq <> pega_char vdir), top
                 )
159            | Difer   -> ExpBool (pega_char vesq <> pega_char vdir, top)
160            | _ -> failwith "interpreta_relacional char"
```

```
161           )
162         | _ ->  failwith "interpreta_relacion| ExpVar (v,tipo) ->al"
163       )
164
165     and interpreta_logico () =
166       (match tesq with
167        | TipoBool ->
168          (match op with
169           | Oulog -> ExpBool (pega_bool vesq || pega_bool vdir, top)
170           | Elog -> ExpBool (pega_bool vesq && pega_bool vdir, top)
171           | _ ->  failwith "interpreta_logico bool"
172          )
173        | _ ->  failwith "interpreta_logico"
174       )
175       in
176     let valor = (match (classifica op) with
177           Aritmetico -> interpreta_aritmetico ()
178         | Logico -> interpreta_logico ()
179         | Relacional -> interpreta_relacional ()
180       )
181     in
182       valor
183   | ExpOperU ((op,top), (esq, tesq)) ->
184     let vesq = interpreta_exp amb esq in
185
186     let interpreta_logico () =
187       (match tesq with
188        | TipoBool ->
189          (match op with
190           | Not ->let vvesq = pega_bool vesq in
191           if vvesq then  ExpBool (false , top)
192           else
193            ExpBool (true, top)
194           | _ -> failwith "interpreta_logico bool"
195          )
196        | _ ->  failwith "interpreta_logico"
197       )
198
199     in
200     let valor = (match (classificaun op) with
201         | Logico -> interpreta_logico ()
202         | _ -> failwith "Classifica unário: não implementado"
203       )
204     in
205       valor
206
207   | ExpChmd (id, args, tipo) ->
208     let open Amb in
209     ( match (Amb.busca amb id) with
210       | Amb.EntFun {tipo_fn; formais; corpo} ->
211           (* Interpreta cada um dos argumentos *)
212           let vargs = List.map (interpreta_exp amb) args in
213           (* Associa os argumentos aos parâmetros formais *)
214           let vformais = List.map2 (fun (n,t) v -> (n, t, Some v))
                formais vargs
215           in interpreta_fun amb id vformais corpo
216       | _ -> failwith "interpreta_exp: expchamada"
217     )
218
```

```
219 and interpreta_fun amb fn_nome fn_formais fn_corpo =
220   let open A in
221  (* Estende o ambiente global, adicionando um ambiente local *)
222   let ambfn = Amb.novo_escopo amb
223   in
224   (* Associa os argumentos aos parâmetros e insere no novo ambiente *)
225   let insere_parametro (n,t,v) = Amb.insere_param ambfn n t v in
226   let _ = List.iter insere_parametro fn_formais in
227     (* Interpreta cada comando presente no corpo da função usando o novo
228        ambiente *)
229   try
230     let _ = List.iter (interpreta_cmd ambfn) fn_corpo in T.ExpNone
231     with
232        Valor_de_retorno expret -> expret
233
234 and interpreta_cmd amb cmd =
235   let open A in
236   let open T in
237   match cmd with
238     CmdReturn exp ->
239     (* Levantar uma exceção foi necessária pois, pela semântica do comando
            de
240         retorno, sempre que ele for encontrado em uma função, a computação
241         deve parar retornando o valor indicado, sem realizar os demais
               comandos.
242     *)
243     (match exp with
244      (* Se a função não retornar nada, verifica se ela foi declarada como
            void *)
245        None -> raise (Valor_de_retorno ExpNone)
246      | Some e ->
247        (* Avalia a expressão e retorne o resultado *)
248        let e1 = interpreta_exp amb e in
249        raise (Valor_de_retorno e1)
250     )
251   | CmdIf (teste, entao, senao) ->
252     let teste1 = interpreta_exp amb teste in
253    (match teste1 with
254        ExpBool (true,_) ->
255        (* Interpreta cada comando do bloco 'então' *)
256        List.iter (interpreta_cmd amb) entao
257      | _ ->
258        (* Interpreta cada comando do bloco 'senão', se houver *)
259        (match senao with
260           None -> ()
261         | Some bloco -> List.iter (interpreta_cmd amb) bloco
262        )
263     )
264
265   | CmdAtrib (elem, exp) ->
266     (* Interpreta o lado direito da atribuição *)
267     let exp = interpreta_exp amb exp
268     (* Faz o mesmo para o lado esquerdo *)
269     and (elem1,tipo) = obtem_nome_tipo_var elem amb in
270     Amb.atualiza_var amb elem1 tipo (Some exp)
271
272   | CmdDeclaraAtrib((elem,tipo), exp) ->
273     let var = match elem with A.VarSimples a -> fst a in
274     let exp = interpreta_exp amb exp in
```

```
275     Amb.insere_local amb var tipo (Some exp)
276
277   | CmdDeclara(elem,tipo) ->
278     let var = match elem with A.VarSimples a -> fst a in
279     Amb.insere_local amb var tipo None
280
281   | CmdExprs exp -> ignore( interpreta_exp amb exp)
282
283   | CmdInputAtr (elem, exps, tipo) ->
284     let _ =  interpreta_cmd amb (CmdPrint exps) in
285     let nome = match elem with A.VarSimples var -> fst var in
286       let valor =
287         (match tipo with
288          | A.TipoInt    -> T.ExpInt    (read_int (),   tipo)
289          | A.TipoStr -> T.ExpStr (read_line (),  tipo)
290          | A.TipoFloat  -> T.ExpFloat  (read_float (), tipo)
291          | A.TipoChar   ->let str = (read_line ()).[0] in T.ExpChar   (str
                , tipo)
292          | _ -> failwith "leia_var: nao implementado"
293         )
294       in  Amb.atualiza_var amb nome tipo (Some valor)
295
296   | CmdInputDecAtr ((elem,_), exps, tipo) ->
297     let _ =  interpreta_cmd amb (CmdPrint exps) in
298     let nome = match elem with A.VarSimples var -> fst var in
299       let valor =
300         (match tipo with
301          | A.TipoInt    -> T.ExpInt    (read_int (),   tipo)
302          | A.TipoStr -> T.ExpStr (read_line (),  tipo)
303          | A.TipoFloat  -> T.ExpFloat  (read_float (), tipo)
304          | A.TipoChar   ->let str = (read_line ()).[0] in T.ExpChar   (str
                , tipo)
305          | _ -> failwith "leia_var: nao implementado"
306         )
307       in  Amb.insere_local amb nome tipo (Some valor)
308
309   | CmdPrint exps ->
310     (* Interpreta cada argumento da função 'saida' *)
311     let exps = List.map (interpreta_exp amb) exps in
312     let imprima exp =
313       (match exp with
314        | T.ExpInt (n,_) ->      let _ = print_int n in print_string " "
315        | T.ExpStr (s,_) -> let _ = print_string s in print_string " "
316        | T.ExpFloat (f,_) -> let _ = print_float  f in print_string " "
317        | T.ExpChar (c,_) -> let _ = print_char  c in print_string " "
318        | T.ExpBool (b,_) ->
319          let _ = print_string (if b then "true" else "false")
320          in print_string " "
321        | _ -> failwith "imprima: nao implementado"
322       ) in
323     let _ = List.iter imprima exps in
324     print_newline ()
325
326
327   | CmdWhile (teste, doit) ->
328      let teste1 = interpreta_exp amb teste in
329    (match teste1 with
330        ExpBool (true,_) ->
331        (* Interpreta uma iteraçao comando do corpo do while *)
```

```
332         let _ = List.iter (interpreta_cmd amb) doit in
333         (* interpreta recursivamente as possíveis demais iterações do
               comando *)
334          interpreta_cmd amb (CmdWhile (teste, doit))
335         | _ -> ()
336     )
337
338   | CmdFor (variavel, (inicio, fim), doit) ->
339   (* Interpreta o For como uma atribuição seguida de um while,
340    que ao final do corpo tem uma operação de incremento na variável de
          iteração *)
341
342     (* incializa variável de iteração *)
343     let _ = interpreta_cmd amb (CmdAtrib (variavel, inicio)) in
344
345     (* monta artificialmente o comando de incremento *)
346     let inc = CmdAtrib (variavel, (ExpOperB((Mais, TipoInt ),
347     (ExpVar (variavel, TipoInt),TipoInt), (ExpInt (1, TipoInt),TipoInt) ))
          ) in
348
349     (* adiciona esse incremento ao final do corpo *)
350     let novocorpo = List.append doit [inc] in
351
352     (* cria o teste (variavel_de_iteração < fim) *)
353     let teste = (ExpOperB((Menor, TipoInt ),(ExpVar (variavel, TipoInt),
          TipoInt),(fim,TipoInt) )) in
354
355     (* relança o comando while para a funçao interpretá-lo*)
356     interpreta_cmd amb (CmdWhile (teste, novocorpo))
357
358   | CmdFor_Dec ((variavel,tipo), (inicio, fim), doit) ->
359   (* Interpreta o For como uma atribuição seguida de um while,
360    que ao final do corpo tem uma operação de incremento na variável de
          iteração *)
361
362     (* incializa variável de iteração *)
363     let _ = interpreta_cmd amb (CmdAtrib (variavel, inicio)) in
364
365     (* monta artificialmente o comando de incremento *)
366     let inc = CmdAtrib (variavel, (ExpOperB((Mais, TipoInt ),
367     (ExpVar (variavel, TipoInt),TipoInt), (ExpInt (1, TipoInt),TipoInt) ))
          ) in
368
369     (* adiciona esse incremento ao final do corpo *)
370     let novocorpo = List.append doit [inc] in
371
372     (* cria o teste (variavel_de_iteração < fim) *)
373     let teste = (ExpOperB((Menor, TipoInt ),(ExpVar (variavel, TipoInt),
          TipoInt),(fim,TipoInt) )) in
374
375     (* relança o comando while para a funçao interpretá-lo*)
376     interpreta_cmd amb (CmdWhile (teste, novocorpo))
377
378 let insere_declaracao_fun amb dec =
379   let open A in
380     match dec with
381         {fn_nome; fn_tiporet; fn_formais;  fn_corpo} ->
382          let nome = match fn_nome with A.VarSimples nome -> fst nome in
383          let formais = List.map (fun (A.VarSimples n,t) ->  ((fst n), t))
```

```
            fn_formais in
384         Amb.insere_fun amb nome formais fn_tiporet fn_corpo
385
386
387 (* Lista de cabeçalhos das funções pré definidas *)
388 let fn_predefs = let open A in [
389    ("entrada", [("x", TipoInt); ("y", TipoInt)], TipoNone);
390    (* ("entradaln", [("x", TipoInt); ("y", TipoInt)], TipoVoid); *)
391    ("saida",   [("x", TipoInt); ("y", TipoInt)], TipoNone);
392    (* ("saidaln",   [("x", TipoInt); ("y", TipoInt)], TipoVoid) *)
393 ]
394
395 (* insere as funções pré definidas no ambiente global *)
396 let declara_predefinidas amb =
397   List.iter (fun (n,ps,c) -> Amb.insere_fun amb n ps c []) fn_predefs
398
399 let interprete ast =
400   (* cria ambiente global inicialmente vazio *)
401   let amb_global = Amb.novo_amb [] in
402   let _ = declara_predefinidas amb_global in
403   let (A.Programa  corpo) = ast in
404   (* Interpreta a função principal *)
405   let resultado = List.iter (
406     fun instr -> (match instr with
407         A.Funcao f -> insere_declaracao_fun amb_global f
408       | A.Cmd c -> interpreta_cmd amb_global c
409   )
410   ) corpo in
411   resultado
```

Listagem 7.3: .ocamlinit

```
1  let () =
2    try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
3    with Not_found -> ()
4  ;;
5
6  #use "topfind";;
7  #require "menhirLib";;
8  #directory "_build";;
9  #load "sintatico.cmo";;
10 #load "lexico.cmo";;
11 #load "ast.cmo";;
12 #load "sast.cmo";;
13 #load "tast.cmo";;
14 #load "tabsimb.cmo";;
15 #load "ambiente.cmo";;
16 #load "semantico.cmo";;
17 #load "ambInterp.cmo";;
18 #load "interprete.cmo";;
19 #load "interpreteTest.cmo";;
20
21 open Ast
22 open AmbInterp
23 open InterpreteTest
```

7.2 Compilação e execução

Para gerar as mensagens de erro, digite:

```
> menhir -v --list-errors sintatico.mly > sintatico.messages
> menhir -v --list-errors sintatico.mly --compile-errors sintatico.
    messages > fnmes.ml
```

Para compilar o arquivo de teste, digite no terminal:

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

Enfim, dentro do ocaml, para chamar o interprete digite:

```
# interprete "exemplos/micro10.py";;
```

Capítulo 8

Representação intermediária

A linguagem intermediária, comumente usada em compiladores comerciais, utilizam uma representação intermediária entre o código fonte e código de máquina. Com o objetivo de facilitar a implementação de otimizações do código, dado que ela é independente da máquina que está efetuando o processo de compilação.

Um tipo popular de linguagem intermediária é conhecido como código de três endereços. Neste tipo de código uma sentença típica tem a forma **X := A op B**, onde *X, A* e *B* são operandos e *op* uma operação qualquer. O nome é relativo ao fato de, para cada instrução, ser possível utilizar no máximo 3 operandos.

8.1 Codigos

Listagem 8.1: Codigo.ml

```
1  open Ast
2  open Tast
3
4  type endereco =
5       Nome of string
6     | ConstInt of int
7     | ConstFloat of float
8     | ConstStr of string
9     | ConstChar of char
10    | ConstBool of bool
11    | Temp of int
12
13 and instrucao =
14      AtribBin of endereco * endereco * opBin * endereco  (* x = y op z *)
15    | Copia of endereco * endereco                        (* x = y      *)
16    | Goto of instrucao                                   (* goto L     *)
17    | If of endereco *  instrucao                         (* if x goto L *)
18    | IfFalse of endereco * instrucao                     (* ifFalse x goto
          L *)
19    | IfRelgoto of endereco * opRel * endereco * instrucao
20                                                          (* if x oprel y goto
                                                               L *)
21    | Call of string * (endereco * Ast.tipo) list * Ast.tipo (* call p,[(x,
```

```
        t)],t *)
22    | Recebe of string * Ast.tipo
23    | Local of string * Ast.tipo
24    | Global of string * Ast.tipo
25    | CallFn of endereco * string *  (endereco * Ast.tipo) list * Ast.tipo
            (* x = call p,n,t *)
26    | Return of endereco option
27    | BeginFun of string * int * int      (* beginFun p,nparam, nlocais *)
28    | EndFun
29    | Rotulo of string
30
31 and opBin = Ast.operador * Ast.tipo
32
33 and opRel = Ast.operador * Ast.tipo
```

Listagem 8.2: Cod3End.ml

```
1  open Printf
2
3  open Ast
4  open Tast
5  open Codigo
6
7  let conta_temp = ref 0
8  let conta_rotulos = ref (Hashtbl.create 5)
9
10 let zera_contadores () =
11   begin
12     conta_temp := 0;
13     conta_rotulos := Hashtbl.create 5
14   end
15
16 let novo_temp () =
17    let numero = !conta_temp in
18    let _ = incr conta_temp in
19    Temp numero
20
21 let novo_rotulo prefixo =
22   if Hashtbl.mem !conta_rotulos prefixo
23   then
24      let numero = Hashtbl.find !conta_rotulos prefixo in
25      let _ = Hashtbl.replace !conta_rotulos prefixo (numero + 1) in
26      Rotulo (prefixo ^ (string_of_int numero))
27   else
28      let _ = Hashtbl.add !conta_rotulos prefixo 1 in
29      Rotulo (prefixo ^ "0")
30
31 let endr_to_str = function
32    | Nome s -> s
33    | ConstInt n -> string_of_int n
34    | ConstStr n -> n
35    | ConstChar n -> String.make 1 n
36    | ConstFloat n -> string_of_float n
37    | ConstBool b -> string_of_bool b
38    | Temp n  -> "t" ^ string_of_int n
39
40 let tipo_to_str t =
41     match t with
42       TipoInt -> "inteiro"
```

```
43      | TipoFloat -> "float"
44      | TipoStr -> "string"
45      | TipoChar -> "char"
46      | TipoBool -> "bool"
47      | TipoNone -> "void"
48
49  let op_to_str op =
50    match op with
51    | Mais  -> "+"
52    | Sub -> "-"
53    | Mult  -> "*"
54    | Div   -> "/"
55    | Mod   -> "%"
56    | Menor -> "<"
57    | Maior -> ">"
58    | MaiorIgual -> ">="
59    | MenorIgual -> "<="
60    | Igual -> "="
61    | Difer -> "!="
62    | Elog     -> "&&"
63    | Oulog    -> "||"
64
65  let rec args_to_str ats =
66     match ats with
67     | [] -> ""
68     | [(a,t)] ->
69       let str = sprintf "(%s,%s)" (endr_to_str a) (tipo_to_str t) in
70       str
71     | (a,t) :: ats ->
72       let str = sprintf "(%s,%s)" (endr_to_str a) (tipo_to_str t) in
73       str ^ ", " ^ args_to_str ats
74
75  let rec escreve_cod3 c =
76    match c with
77    | AtribBin (x,y,op,z) ->
78        sprintf "%s := %s %s %s\n" (endr_to_str x)
79                                   (endr_to_str y) (op_to_str (fst op)) (
                                      endr_to_str z)
80    | Copia (x,y) ->
81        sprintf "%s := %s\n" (endr_to_str x) (endr_to_str y)
82    | Goto l ->
83        sprintf "goto %s\n" (escreve_cod3 l)
84    | If (x,l) ->
85        sprintf "if %s goto %s\n" (endr_to_str x) (escreve_cod3 l)
86    | IfFalse (x,l) ->
87        sprintf "ifFalse %s goto %s\n" (endr_to_str x) (escreve_cod3 l)
88    | IfRelgoto (x,oprel,y,l) ->
89        sprintf "if %s %s %s goto %s\n" (endr_to_str x) (op_to_str (fst
            oprel))
90                                        (endr_to_str y) (escreve_cod3 l)
91    | Call (p,ats,t) -> sprintf "call %s(%s): %s\n" p (args_to_str ats) (
        tipo_to_str t)
92    | Recebe (x,t) -> sprintf "recebe %s,%s\n" x (tipo_to_str t)
93    | Local (x,t)  -> sprintf "local %s,%s\n" x (tipo_to_str t)
94    | Global (x,t) -> sprintf "global %s,%s\n" x (tipo_to_str t)
95    | CallFn (x,p,ats,t) ->
96        sprintf "%s := call %s(%s): %s\n" (endr_to_str x) p (args_to_str ats
            ) (tipo_to_str t)
97    | Return x ->
```

```
98      (match x with
99         None   -> "return\n"
100       | Some x -> sprintf "return %s\n" (endr_to_str x) )
101   | BeginFun (id,np,nl) -> sprintf "beginFun %s(%d,%d)\n" id np nl
102   | EndFun -> "endFun\n\n"
103   | Rotulo r -> sprintf "%s: " r
104
105
106 let rec escreve_codigo cod =
107   match cod with
108   | [] -> printf "\n"
109   | c::cs -> printf "%s" (escreve_cod3 c);
110              escreve_codigo cs
111
112 let pega_tipo exp =
113   match exp with
114   | ExpInt (n, t) -> t
115   | ExpVar (v, t) -> t
116   | ExpStr  (s, t) -> t
117   | ExpBool (b, t) -> t
118   | ExpFloat (f, t) -> t
119   | ExpOperB ((op,t),_,_) -> t
120   | ExpChmd (id, args, t) -> t
121   | _ -> failwith "pega_tipo: não implementado"
122
123 let rec traduz_exp exp =
124   match exp with
125   | ExpInt (n, TipoInt) ->
126      let t = novo_temp () in
127     (t, [Copia (t, ConstInt n)])
128
129   | ExpVar (v, tipo) ->
130     (match v with
131        VarSimples nome ->
132        let id = fst nome in
133        ((Nome id), [])
134     )
135
136   | ExpStr (n, TipoStr) ->
137     let t = novo_temp () in
138    (t, [Copia (t, ConstStr n)])
139
140   | ExpOperB (op, exp1, exp2) ->
141     let (endr1, codigo1) = let (e1,t1) = exp1 in traduz_exp e1
142     and (endr2, codigo2) = let (e2,t2) = exp2 in traduz_exp e2
143     and t = novo_temp () in
144     let codigo = codigo1 @ codigo2 @ [AtribBin (t, endr1, op, endr2)] in
145     (t, codigo)
146
147   | ExpChmd (id, args, tipo_fn) ->
148       let (enderecos, codigos) = List.split (List.map traduz_exp args) in
149       let tipos = List.map pega_tipo args in
150       let endr_tipos = List.combine enderecos tipos
151       and t = novo_temp () in
152       let codigo = (List.concat codigos) @
153                    [CallFn (t, id, endr_tipos, tipo_fn)]
154       in
155         (t, codigo)
156   | _ -> failwith "traduz_exp: não implementado"
```

```
157
158
159 let rec traduz_cmd cmd isInFun =
160
161   match cmd with
162
163   | CmdReturn exp ->
164     (match exp with
165      | None -> [Return None]
166      | Some e ->
167        let (endr_exp, codigo_exp) = traduz_exp e in
168        codigo_exp @ [Return (Some endr_exp)]
169     )
170
171   | CmdAtrib (elem, ExpInt (n, TipoInt)) ->
172     let (endr_elem, codigo_elem) = traduz_exp (ExpVar (elem, TipoInt))
173     in codigo_elem @ [Copia (endr_elem, ConstInt n)]
174
175   | CmdAtrib (elem, ExpStr (n, TipoStr)) ->
176     let (endr_elem, codigo_elem) = traduz_exp (ExpVar (elem, TipoStr))
177     in codigo_elem @ [Copia (endr_elem, ConstStr n)]
178
179   | CmdAtrib (elem, ExpFloat (n, TipoFloat)) ->
180     let (endr_elem, codigo_elem) = traduz_exp (ExpVar (elem, TipoFloat))
181     in codigo_elem @ [Copia (endr_elem, ConstFloat n)]
182
183   | CmdAtrib (elem, ExpChar (n, TipoChar)) ->
184     let (endr_elem, codigo_elem) = traduz_exp (ExpVar (elem, TipoChar))
185     in codigo_elem @ [Copia (endr_elem, ConstChar n)]
186
187   | CmdAtrib (elem, exp) ->
188     let (endr_exp, codigo_exp) = traduz_exp exp
189     and (endr_elem, codigo_elem) = traduz_exp (ExpVar (elem, TipoNone))
        in
190     let codigo = codigo_exp @ codigo_elem @ [Copia (endr_elem, endr_exp)]
191     in codigo
192
193   | CmdDeclara (var,tipo) -> let id = match var with Ast.VarSimples id ->
       fst id in
194     if isInFun then
195       [Local (id,tipo)]
196     else [Global (id,tipo)]
197
198   | CmdDeclaraAtrib ((var,tipo),exp) ->
199     let id = match var with Ast.VarSimples id -> fst id in let dec = if
        isInFun then
200       [Local (id,tipo)]
201     else [Global (id,tipo)] in
202     let (endr_exp, codigo_exp) = traduz_exp exp
203     and (endr_elem, codigo_elem) = traduz_exp (ExpVar (var, TipoNone))  in
204     let codigo = dec @ codigo_exp @ codigo_elem @ [Copia (endr_elem,
        endr_exp)]
205     in codigo
206
207   | CmdInputAtr (arg, saida, tipo)->
208       let (enderecosp, codigosp) = List.split (List.map traduz_exp saida)
          in
209       let tipos = List.map pega_tipo saida in
210       let endr_tipos = List.combine enderecosp tipos in
```

```
211       let (enderecos, codigos) = traduz_exp (ExpVar (arg, TipoNone)) in
212         (List.concat codigosp) @
213         [Call ("print", endr_tipos, TipoNone)] @
214         codigos @
215         [Call ("read", [(enderecos,tipo)], TipoNone)]
216
217   | CmdInputDecAtr ((arg,tipo), saida, tipof)->
218       let id = match arg with Ast.VarSimples id -> fst id in
219       let dec = if isInFun then
220       [Local (id,tipo)]
221       else [Global (id,tipo)] in
222       let (enderecosp, codigosp) = List.split (List.map traduz_exp saida)
             in
223       let tipos = List.map pega_tipo saida in
224       let endr_tipos = List.combine enderecosp tipos in
225       let (enderecos, codigos) = traduz_exp (ExpVar (arg, TipoNone)) in
226         dec @
227         (List.concat codigosp) @
228         [Call ("print", endr_tipos, TipoNone)] @
229         codigos @
230         [Call ("read", [(enderecos,tipo)], TipoNone)]
231
232   | CmdIf (teste, entao, senao) ->
233     let (endr_teste, codigo_teste) = traduz_exp teste
234     and codigo_entao = traduz_cmds entao isInFun
235     and rotulo_falso = novo_rotulo "L" in
236     (match senao with
237         | None -> codigo_teste @
238                   [IfFalse (endr_teste, rotulo_falso)] @
239                   codigo_entao @
240                   [rotulo_falso]
241         | Some cmds ->
242           let codigo_senao = traduz_cmds cmds isInFun
243           and rotulo_fim = novo_rotulo "L" in
244               codigo_teste @
245               [IfFalse (endr_teste, rotulo_falso)] @
246               codigo_entao @
247               [Goto rotulo_fim] @
248               [rotulo_falso] @ codigo_senao @
249               [rotulo_fim]
250     )
251
252   | CmdWhile (teste, doit) ->
253     let (endr_teste, codigo_teste) = traduz_exp teste
254     and codigo_doit = traduz_cmds doit isInFun
255     and rotulo_inicio = novo_rotulo "W"
256     and rotulo_fim = novo_rotulo "W"  in
257       [rotulo_inicio] @
258       codigo_teste @
259       [IfFalse (endr_teste, rotulo_fim)] @
260       codigo_doit @
261       [Goto rotulo_inicio] @
262       [rotulo_fim]
263
264   | CmdFor_Dec ((v,tipo), (inicio, fim), doit) ->
265     let id = match v with Ast.VarSimples id -> fst id in
266     let dec = if isInFun then [Local (id,tipo)]
267     else [Global (id,tipo)] in
268     let (endr_teste, codigo_teste) = traduz_exp (ExpOperB ((Menor, TipoInt
```

```
          ),
269       (ExpVar (v,TipoInt),TipoInt), (fim,TipoInt))) in
270     let codigo_atrib = traduz_cmds [(CmdAtrib (v, inicio))] isInFun
271     and codigo_inc = traduz_cmds [(CmdAtrib (v, (ExpOperB ((Mais, TipoInt)
          ,
272       (ExpVar (v,TipoInt),TipoInt), (ExpInt (1, TipoInt),TipoInt))))))]
            isInFun
273     and codigo_doit = traduz_cmds doit isInFun
274     and rotulo_inicio = novo_rotulo "W"
275     and rotulo_fim = novo_rotulo "W"  in
276       dec @
277       codigo_atrib @
278       [rotulo_inicio] @
279       codigo_teste @
280       [IfFalse (endr_teste, rotulo_fim)] @
281       codigo_doit @
282       codigo_inc @
283       [Goto rotulo_inicio] @
284       [rotulo_fim]
285
286   | CmdFor (v, (inicio, fim), doit) ->
287     let (endr_teste, codigo_teste) = traduz_exp (ExpOperB ((Menor, TipoInt
          ),
288       (ExpVar (v,TipoInt),TipoInt), (fim,TipoInt))) in
289     let codigo_atrib = traduz_cmds [(CmdAtrib (v, inicio))] isInFun
290     and codigo_inc = traduz_cmds [(CmdAtrib (v, (ExpOperB ((Mais, TipoInt)
          ,
291       (ExpVar (v,TipoInt),TipoInt), (ExpInt (1, TipoInt),TipoInt))))))]
            isInFun
292     and codigo_doit = traduz_cmds doit isInFun
293     and rotulo_inicio = novo_rotulo "W"
294     and rotulo_fim = novo_rotulo "W"  in
295       codigo_atrib @
296       [rotulo_inicio] @
297       codigo_teste @
298       [IfFalse (endr_teste, rotulo_fim)] @
299       codigo_doit @
300       codigo_inc @
301       [Goto rotulo_inicio] @
302       [rotulo_fim]
303       | CmdExprs (ExpChmd (id, args, tipo_fn)) ->
304       let (enderecos, codigos) = List.split (List.map traduz_exp args) in
305       let tipos = List.map pega_tipo args in
306       let endr_tipos = List.combine enderecos tipos in
307       (List.concat codigos) @
308       [Call (id, endr_tipos, tipo_fn)]
309
310   | CmdExprs _ -> []
311
312   | CmdPrint args ->
313       let (enderecos, codigos) = List.split (List.map traduz_exp args) in
314       let tipos = List.map pega_tipo args in
315       let endr_tipos = List.combine enderecos tipos in
316       (List.concat codigos) @
317       [Call ("print", endr_tipos, TipoNone)]
318
319 and traduz_cmds cmds isInFun =
320     match cmds with
321     | [] -> []
```

```
322     | cmd :: cmds ->
323       let codigo = traduz_cmd cmd isInFun in
324       codigo @ traduz_cmds cmds isInFun
325
326 let conta_locais corpo =
327     List.length (List.filter (fun cmd ->
328       match cmd with
329         | Local (_,_) -> true
330         | _ -> false
331    ) corpo )
332
333 let traduz_fun ast isInFun =
334   match ast with
335     Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
336     let fn_nome = (match fn_nome with ( VarSimples (a, tipo)) -> (a,tipo))
            in
337     let nome = fst fn_nome in
338     let formais = List.map (fun ( VarSimples n,t) ->  Recebe (fst n, t) )
          fn_formais in
339     let nformais = List.length fn_formais in
340     let corpo = traduz_cmds fn_corpo isInFun in
341     let nlocais = conta_locais corpo in
342     [BeginFun (nome, nformais, nlocais)] @ formais @ corpo @ [EndFun]
343     | _ -> failwith "traduz_fun: não implementado"
344
345 let tradutor ast_tipada =
346   let _ = zera_contadores () in
347   let (Ast.Programa(instr)) = ast_tipada in
348   let funs_trad = List.map (fun ins -> (match ins with
349   | Funcao _ -> traduz_fun ins true
350   | Cmd cmd  -> traduz_cmd cmd false )) instr in
351   (List.concat funs_trad)
```

Listagem 8.3: cod3endTest.ml

```
1  open Printf
2  open Lexing
3
4  open Ast
5  exception Erro_Sintatico of string
6
7  module S = MenhirLib.General (* Streams *)
8  module I = Sintatico.MenhirInterpreter
9
10 open Semantico
11 open Codigo
12 open Cod3End
13
14 let message =
15   fun s ->
16     match s with
17     | 0 ->
18         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
19     | 1 ->
20         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
21     | 34 ->
22         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
23     | 35 ->
24         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
```

```
25      | 36 ->
26          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
27      | 72 ->
28          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
29      | 47 ->
30          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
31      | 48 ->
32          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
33      | 49 ->
34          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
35      | 51 ->
36          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
37      | 52 ->
38          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
39      | 55 ->
40          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
41      | 56 ->
42          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
43      | 57 ->
44          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
45      | 58 ->
46          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
47      | 61 ->
48          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
49      | 62 ->
50          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
51      | 63 ->
52          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
53      | 64 ->
54          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
55      | 73 ->
56          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
57      | 74 ->
58          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
59      | 95 ->
60          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
61      | 89 ->
62          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
63      | 97 ->
64          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
65      | 98 ->
66          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
67      | 99 ->
68          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
69      | 65 ->
70          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
71      | 66 ->
72          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
73      | 53 ->
74          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
75      | 67 ->
76          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
77      | 68 ->
78          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
79      | 59 ->
80          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
81      | 60 ->
82          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
83      | 42 ->
```

```
84          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
85      | 41 ->
86          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
87      | 70 ->
88          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
89      | 75 ->
90          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
91      | 77 ->
92          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
93      | 76 ->
94          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
95      | 105 ->
96          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
97      | 84 ->
98          "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
99      | 43 ->
100         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
101     | 85 ->
102         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
103     | 86 ->
104         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
105     | 45 ->
106         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
107     | 46 ->
108         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
109     | 102 ->
110         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
111     | 103 ->
112         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
113     | 81 ->
114         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
115     | 3 ->
116         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
117     | 2 ->
118         "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
119     | 6 ->
120         "estado 6: esperava um tipo. Exemplo:\n   x : inteiro;\n"
121     | 7 ->
122         "estado 7: esperava a definicao de um campo. Exemplo:\n   i:
               registro\n         parte_real: inteiro;\n         parte_imag:
               inteiro;\n      fim registro;\n      "
123     | 8 ->
124         "estado 8: esperava ':'. Exemplo:\n   x: inteiro;\n   "
125     | 9 ->
126         "estado 9: esperava um tipo. Exemplo:\n   x: inteiro;\n"
127     | 25 ->
128         "estado 25: esperva um ';'.\n"
129     | 26 ->
130         "estado 26: uma declaracao foi encontrada. Para continuar era\n
               esperado uma outra declara\195\167\195\163o ou a palavra '
               inicio'.\n"
131     | 29 ->
132         "estado 29: espera a palavra 'registro'. Exemplo:\n   i: registro\
               n         parte_real: inteiro;\n         parte_imag: inteiro;\n
                     fim registro;\n"
133     | 31 ->
134         "estado 31: esperava um ';'. \n"
135     | 107 ->
136         "estado 107: uma declaracao foi encontrada. Para continuar era\n
```

```
                esperado uma outra declara\195\167\195\163o ou a palavra '
              inicio'.\n"
137     | 13 ->
138         "estado 13: esperava um '['. Exemplo:\n   arranjo [1..10] de
              inteiro;\n"
139     | 14 ->
140         "estado 14: esperava os limites do vetor. Exemplo:\n   arranjo
              [1..10] de inteiro;\n"
141     | 15 ->
142         "estado 15: esperava '..'. Exemplo:\n   1 .. 10\n"
143     | 16 ->
144         "estado 16: esperava um numero inteiro. Exemplo:\n   1 .. 10\n"
145     | 18 ->
146         "estado 18: esperava um ']'. Exemplo\n   arranjo [1..10] de
              inteiro;\n"
147     | 19 ->
148         "estado 19: esperava a palavra reservada 'de'. Exemplo:\n
              arranjo [1..10] de inteiro;\n"
149     | 20 ->
150         "estado 20: esperava um tipo. Exemplo\n   arranjo [1..10] de
              inteiro;\n"
151     | _ ->
152         raise Not_found
153
154 let posicao lexbuf =
155     let pos = lexbuf.lex_curr_p in
156     let lin = pos.pos_lnum
157     and col = pos.pos_cnum - pos.pos_bol - 1 in
158     sprintf "linha %d, coluna %d" lin col
159
160 (* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
       checkpoint *)
161
162 let pilha checkpoint =
163   match checkpoint with
164   | I.HandlingError amb -> I.stack amb
165   | _ -> assert false (* Isso não pode acontecer *)
166
167 let estado checkpoint : int =
168   match Lazy.force (pilha checkpoint) with
169   | S.Nil -> (* O parser está no estado inicial *)
170       0
171   | S.Cons (I.Element (s, _, _, _), _) ->
172       I.number s
173
174 let sucesso v = Some v
175
176 let falha lexbuf (checkpoint : (Sast.expressao Ast.programa) I.checkpoint)
       =
177   let estado_atual = estado checkpoint in
178   let msg = message estado_atual in
179   raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
180                                         (Lexing.lexeme_start lexbuf) msg))
181
182 let loop lexbuf resultado =
183   let fornecedor = I.lexer_lexbuf_to_supplier (Lexico.token (Lexico.
         cria_estado ())) lexbuf in
184   I.loop_handle sucesso (falha lexbuf) fornecedor resultado
185
```

```
186
187 let parse_com_erro lexbuf =
188   try
189     Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.lex_curr_p))
190   with
191   | Lexico.Erro_lexico msg ->
192      printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
193      None
194   | Erro_Sintatico msg ->
195      printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
196      None
197
198 let parse s =
199   let lexbuf = Lexing.from_string s in
200   let ast = parse_com_erro lexbuf in
201   ast
202
203 let parse_arq nome =
204   let ic = open_in nome in
205   let lexbuf = Lexing.from_channel ic in
206   let ast = parse_com_erro lexbuf in
207   let _ = close_in ic in
208   ast
209
210 let verifica_tipos nome =
211   let ast = parse_arq nome in
212   match ast with
213     Some (Some ast) -> semantico ast
214   | _ -> failwith "Nada a fazer!\n"
215
216 let traduz nome =
217   let (arv,tab) = verifica_tipos nome in
218   tradutor arv
219
220 let imprime_traducao cod =
221    let _ = printf "\n" in
222    escreve_codigo cod
223
224 let gera_nano num =
225    let cod = traduz ("exemplos/nano" ^ string_of_int(num) ^ ".py")
226     in imprime_traducao cod
227
228 let gera_micro num =
229    let cod = traduz ("exemplos/micro" ^ string_of_int(num) ^ ".py")
230     in imprime_traducao cod
```

Listagem 8.4: .ocamlinit

```
1 let () =
2   try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
3   with Not_found -> ()
4 ;;
5
6 #use "topfind";;
7 #require "menhirLib";;
8 #directory "_build";;
9 #load "sintatico.cmo";;
10 #load "lexico.cmo";;
11 #load "ast.cmo";;
```

```
#load "sast.cmo";;
#load "tast.cmo";;
#load "tabsimb.cmo";;
#load "ambiente.cmo";;
#load "semantico.cmo";;
#load "Codigo.cmo";;
#load "Cod3End.cmo";;
#load "cod3endTest.cmo";;

open Ast
open Ambiente
open Codigo
open Cod3End
open Cod3endTest
```

8.2 Compilação e execução

Para compilar, digite no terminal:

```
ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib cod3endTest.byte
```

Para usar, entre no ocaml: rlwrap ocaml Para gerar a representação intermediária de um programa, entre no ocaml e digite:

```
# traduz "exemplos/micro10.py";;
```

Se quiser ver em um formato mais legível:

```
# let cod = traduz "exemplos/micro10.py" in imprime_traducao cod;;
```

8.3 Exemplos

O exemplo usado foi o micro10.py:

Listagem 8.5: micro10.py

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

def main() -> None:
    numero:int = int(input("Digite um numero: "))
    fat:int = fatorial(numero)
    print("O fatorial eh", fat)
    return

main()
```

Usando o modo:

```
# traduz "exemplos/micro10.py";;
```

Listagem 8.6: micro10.py

```
[BeginFun ("fatorial", 1, 0); Recebe ("n", TipoInt);
 Copia (Temp 0, ConstInt 0);
 AtribBin (Temp 1, Nome "n", (MenorIgual, TipoBool), Temp 0);
 IfFalse (Temp 1, Rotulo "L0"); Copia (Temp 2, ConstInt 1);
 Return (Some (Temp 2)); Goto (Rotulo "L1"); Rotulo "L0";
 Copia (Temp 3, ConstInt 1);
 AtribBin (Temp 4, Nome "n", (Sub, TipoInt), Temp 3);
 CallFn (Temp 5, "fatorial", [(Temp 4, TipoInt)], TipoInt);
 AtribBin (Temp 6, Nome "n", (Mult, TipoInt), Temp 5);
 Return (Some (Temp 6)); Rotulo "L1"; EndFun; BeginFun ("main", 0, 2);
 Local ("numero", TipoInt); Copia (Temp 7, ConstStr "Digite um numero: ");
 Call ("print", [(Temp 7, TipoStr)], TipoNone);
 Call ("read", [(Nome "numero", TipoInt)], TipoNone); Local ("fat",
     TipoInt);
 CallFn (Temp 8, "fatorial", [(Nome "numero", TipoInt)], TipoInt);
 Copia (Nome "fat", Temp 8); Copia (Temp 9, ConstStr "O fatorial eh");
 Call ("print", [(Temp 9, TipoStr); (Nome "fat", TipoInt)], TipoNone);
 Return None; EndFun; Call ("main", [], TipoNone)]
```

Usando o modo mais legível:

```
# let cod = traduz "exemplos/micro10.py" in imprime_traducao cod;;
```

Listagem 8.7: micro10.py

```
beginFun fatorial(1,0)
recebe n,inteiro
t0 := 0
t1 := n <= t0
ifFalse t1 goto L0:
t2 := 1
return t2
goto L1:
L0: t3 := 1
t4 := n - t3
t5 := call fatorial((t4,inteiro)): inteiro
t6 := n * t5
return t6
L1: endFun

beginFun main(0,2)
local numero,inteiro
t7 := Digite um numero:
call print((t7,string)): void
call read((numero,inteiro)): void
local fat,inteiro
t8 := call fatorial((numero,inteiro)): inteiro
fat := t8
t9 := O fatorial eh
call print((t9,string), (fat,inteiro)): void
return
endFun

call main(): void
```

```
30
31 - : unit = ()
```

Capítulo 9

Referências

Python

Dalvik

Smali

OCaml