

Construção de um compilador de Lua para Parrot Virtual Machine usando Objective Caml

Guilherme Pacheco de Oliveira

`guilherme.061@gmail.com`

Faculdade de Computação
Universidade Federal de Uberlândia

22 de novembro de 2016

Lista de Figuras

2.1	Instalando e testando LUA	9
2.2	Instalando e testando OCaml	10
2.3	Instalando e testando Parrot	11

Lista de Tabelas

Lista de Listagens

2.1	Output Simples em Parrot Assembly Language	11
2.2	Output Simples em Parrot Intermediate Representation	11
3.1	Programa nano 01 em Ruby	13
3.2	Programa nano 01 em PIR	14
4.1	Programa nano 01 em Lua	15
4.2	Programa nano 01 em PASM	15
4.3	Programa nano 02 em Lua	15
4.4	Programa nano 02 em PASM	15
4.5	Programa nano 03 em Lua	16
4.6	Programa nano 03 em PASM	16
4.7	Programa nano 04 em Lua	16
4.8	Programa nano 04 em PASM	16
4.9	Programa nano 05 em Lua	16
4.10	Programa nano 05 em PASM	16
4.11	Programa nano 06 em Lua	16
4.12	Programa nano 06 em PASM	17
4.13	Programa nano 07 em Lua	17
4.14	Programa nano 07 em PASM	17
4.15	Programa nano 08 em Lua	17
4.16	Programa nano 08 em PASM	18
4.17	Programa nano 09 em Lua	18
4.18	Programa nano 09 em PASM	18
4.19	Programa nano 10 em Lua	19
4.20	Programa nano 10 em PASM	19
4.21	Programa nano 11 em Lua	19
4.22	Programa nano 11 em PASM	20
4.23	Programa nano 12 em Lua	20
4.24	Programa nano 12 em PASM	20
4.25	Programa micro 01 em Lua	21
4.26	Programa Micro 01 em PASM	21
4.27	Programa micro 02 em Lua	22
4.28	Programa Micro 02 em PASM	22
4.29	Programa micro 03 em Lua	23
4.30	Programa Micro 03 em PASM	23
4.31	Programa micro 04 em Lua	24
4.32	Programa Micro 04 em PASM	24
4.33	Programa micro 05 em Lua	25
4.34	Programa Micro 05 em PASM	26
4.35	Programa micro 06 em Lua	27
4.36	Programa Micro 06 em PASM	27

4.37	Programa micro 07 em Lua	28
4.38	Programa Micro 07 em PASM	29
4.39	Programa micro 08 em Lua	29
4.40	Programa Micro 08 em PASM	30
4.41	Programa micro 09 em Lua	30
4.42	Programa Micro 09 em PASM	31
4.43	Programa micro 10 em Lua	32
4.44	Programa Micro 10 em PASM	32
4.45	Programa micro 11 em Lua	33
4.46	Programa Micro 11 em PASM	33
5.1	Automato reconhecedor da linguagem descrita	39
5.2	Arquivo do Reconhecedor léxico de Lua	43
5.3	Carregador	45
5.4	Teste de Tokens	46
6.1	Arquivo Sintático	50
6.2	Arquivo Léxico	51
6.3	Arvore Sintatica	52
6.4	Regras da linguagem Lua no Menhir	55
6.5	Arvore Sintática	61
6.6	Makefile	63

Sumário

Lista de Figuras	2
Lista de Tabelas	3
1 Introdução	8
2 Instalação dos componentes	9
2.1 Homebrew	9
2.2 Lua	9
2.2.1 Instalação e Teste	9
2.2.2 Informações sobre a linguagem Lua	10
2.3 Ocaml	10
2.3.1 Instalação e Teste	10
2.3.2 Informações sobre a linguagem OCaml	10
2.4 Parrot Virtual Machine	10
2.4.1 Instalação e Teste	10
2.4.2 Informações sobre a Parrot Virtual Machine	11
2.4.3 Parrot Assembly Language (PASM)	12
2.4.4 Parrot Intermediate Representation (PIR)	12
3 Compilação de Código Ruby para Parrot Intermediate Representation (PIR)	13
4 Códigos LUA e Parrot Assembly (PASM)	15
4.0.1 Nano Programas	15
4.0.2 Micro Programas	21
5 Analisador Léxico	35
5.1 Análise Léxica	35
5.2 Analisador Manual	35
5.2.1 Linguagem a ser interpretada	35
5.2.2 Autômato reconhecedor da linguagem	36
5.2.3 Implementação	38
5.3 Analisador com Ocamllexer	42
5.3.1 Convenções Léxicas da Linguagem Lua	42
5.3.2 Implementação	43
5.3.3 Testes	46
5.3.4 Futuras Correções	49

6	Analizador Sintático	50
6.1	Parser Preditivo	50
6.1.1	Linguagem Analisada	50
6.1.2	Arquivo Sintatico	50
6.1.3	Arquivo Léxico	51
6.1.4	Gerador da Arvore Sintática	52
6.1.5	Testes	54
6.2	Análise Sintática com Menhir	54
6.2.1	Arquivo Ocamlinit	54
6.2.2	Tipos de Parser	54
6.2.3	Análise Sintática Linguagem Lua	55
6.2.4	Futuras correções	64
6.2.5	Testes	64
7	Referências	65

Capítulo 1

Introdução

Este documento tem como intenção documentar todo o processo de construção de um compilador para a linguagem de programação Lua para a máquina virtual Parrot utilizando Objective OCaml.

A intenção é cobrir desde os primeiros passos, como instalação das ferramentas até conceitos de compiladores, implementações em OCaml e detalhes técnicos. Dessa forma, espera-se que um leitor seja capaz de replicar os resultados.

O Sistema Operacional utilizado é OS X El Capitan 10.11.6

Capítulo 2

Instalação dos componentes

2.1 Homebrew

Homebrew é um gerenciador de pacotes para Mac OS X, escrito em Ruby, e é responsável por instalar pacotes nos diretórios adequados e fazer adequadamente a configuração desses pacotes, instalá-lo facilita todo o processo de instalação dos componentes necessários.

Para instalar o homebrew basta digitar no terminal:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2.2 Lua

2.2.1 Instalação e Teste

Para instalar Lua através do homebrew, basta digitar no terminal:

```
$ brew install lua
```

Resultado:

Figura 2.1: *Instalando e testando LUA*

```
oliveira:lua oliveira$ brew install lua
=> Downloading https://homebrew.bintray.com/bottles/lua-5.2.4_3.el_capitan.bott
##### 100.0%
=> Pouring lua-5.2.4_3.el_capitan.bottle.tar.gz
=> Caveats
Please be aware due to the way Luarocks is designed any binaries installed
via Luarocks-5.2 AND 5.1 will overwrite each other in /usr/local/bin.

This is, for now, unavoidable. If this is troublesome for you, you can build
rocks with the '--tree=' command to a special, non-conflicting location and
then add that to your 'SPATH'.
=> Summary
📦 /usr/local/Cellar/lua/5.2.4_3: 143 files, 697.3K
oliveira:lua oliveira$ lua
Lua 5.2.4 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> print("Hello World")
Hello World
> ^D
oliveira:lua oliveira$
```

2.2.2 Informações sobre a linguagem Lua

A principal referência para Lua é a documentação em seu site oficial [1]. Lua é uma linguagem de programação de extensão, projetada para dar suporte à outras linguagens de programação procedimental e planejada para ser usada como uma linguagem de script leve e facilmente embarcável, é implementada em C.

2.3 Ocaml

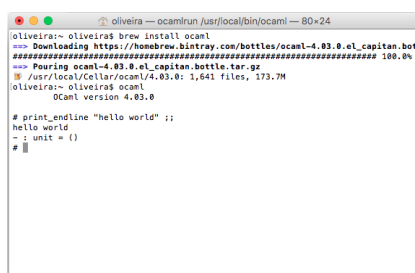
2.3.1 Instalação e Teste

Novamente através do homebrew, basta digitar:

```
$ brew install ocaml
```

Resultado:

Figura 2.2: *Instalando e testando OCaml*



2.3.2 Informações sobre a linguagem OCaml

A documentação oficial do OCaml [2] possui manuais, licenças, documentos e algumas dicas sobre como programar adequadamente na linguagem. OCaml é uma linguagem de programação funcional, imperativa e orientada à objetos.

2.4 Parrot Virtual Machine

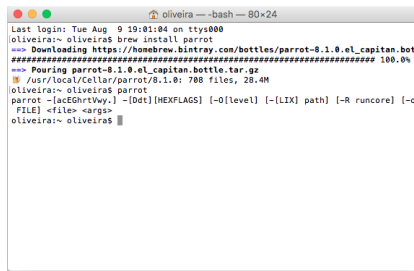
2.4.1 Instalação e Teste

Digitar no Terminal:

```
$ brew install parrot
```

Resultado:

Figura 2.3: Instalando e testando Parrot



```

Last login: Tue Aug 9 19:01:04 on ttys000
oliveira:~ oliveiras$ brew install parrot
=> Downloading https://homebrew.bintray.com/bottles/parrot-8.1.0.o1_capitan.bot
##### 100.0%
=> Pouring parrot-8.1.0.o1_capitan.bottle.tar.gz
=> /usr/local/Cellar/parrot/8.1.0: 700 files, 20.4M
oliveira:~ oliveiras$ parrot
parrot - [acEhrtVwy.] - [Dot] [HEXFLAGS] [-O [level]] [-L [LIX] path] [-R runcore] [-o
FILE] <file> <args>
oliveira:~ oliveiras$

```

2.4.2 Informações sobre a Parrot Virtual Machine

A máquina virtual Parrot é utilizada principalmente para linguagens dinâmicas como Perl, Python, Ruby e PHP, seu design foi originalmente feito para trabalhar com a versão 6 de Perl, mas seu uso foi expandido como uma máquina virtual dinâmica e de propósito geral, apta a lidar com qualquer linguagem de programação de alto nível. [3]

Parrot pode ser programada em diversas linguagens, os dois mais utilizados são: Parrot Assembly Language (PASM): É a linguagem de mais baixo nível utilizada pela Parrot, muito similar a um assembly tradicional. Parrot Intermediate Representation (PIR): De mais alto nível que PASM, também um pouco mais fácil de se utilizar e mais utilizada.

Fazendo alguns testes com PASM e PIR:

Listagem 2.1: Output Simples em Parrot Assembly Language

```

1 say "Here are the news about Parrots."
2 end

```

Para executar o código:

```
$ parrot news.pasm
```

Listagem 2.2: Output Simples em Parrot Intermediate Representation

```

1 .sub main :main
2   print "No parrots were involved in an accident on the M1 today...\n"
3 .end

```

Para executar o código:

```
$ parrot hello.pir
```

Os arquivos PASM e PIR são convertidos para Parrot Bytecode (PBC) e somente então são executados pela máquina virtual, é possível obter o arquivo .pbc através comando:

```
$ parrot -o output.pbc input.pasm
```

De acordo com a documentação oficial, o Compilador Intermediário de Parrot é capaz de traduzir códigos PIR para PASM através do comando:

```
$ parrot -o output.pasm input.pir
```

Mas essa execução resultou em um código bytecode invés do assembly.

Apesar da documentação oficial enfatizar que PIR é mais utilizado e mais recomendado para o desenvolvimento de compiladores para Parrot, o alvo será a linguagem Assembly PASM.

2.4.3 Parrot Assembly Language (PASM)

A linguagem PASM é muito similar a um assembly tradicional, com exceção do fato de que algumas instruções permitem o acesso a algumas funções dinâmicas de alto nível do sistema Parrot.

Parrot é uma maquina virtual baseada em registradores, há um número ilimitado de registradores que não precisam ser instanciados antes de serem utilizados, a maquina virtual se certifica de criar os registradores de acordo com a sua necessidade, tal como fazer a reutilização e se livrar de registradores que não estão mais sendo utilizados, todos os registradores começam com o símbolo "\$" e existem 4 tipos de dados, cada um com suas regras:

Strings: Registradores de strings começam com um S, por exemplo: "\$S10"

Inteiros: Registradores de inteiros começam com um I, por exemplo: "\$I10"

Número: Registradores de números de ponto flutuante, começam com a letra N, por exemplo: "\$N10"

PMC: São tipos de dados utilizados em orientação a objetos, podem ser utilizados para guardar vários tipos de dados, começam com a letra P, por exemplo: "\$P10"

Para mais referências sobre PASM, consultar [4], [7] para os opcodes e [8] para exemplos.

2.4.4 Parrot Intermediate Representation (PIR)

A maior dos compiladores possuem como alvo o PIR, inclusive o que será utilizado para estudar qual o comportamento um compilador deve ter ao gerar o assembly. A própria máquina virtual Parrot possui um módulo intermediário capaz de interpretar a linguagem PIR e gerar o bytecode ou o próprio assembly (PASM), além disso, existem compiladores capazes de realizar a mesma tarefa.

PIR é de nível mais alto que assembly mas ainda muito próximo do nível de máquina, o principal benefício é a facilidade em programar em PIR em comparação com a programação em PASM, além disso, ela foi feita para compiladores de linguagens de alto nível gerarem código PIR para trabalhar com a maquina Parrot. Mais informações sobre PIR e sua sintaxe podem ser encontradas em [5].

Capítulo 3

Compilação de Código Ruby para Parrot Intermediate Representation (PIR)

O compilador que será utilizado será o Cardinal [6], é um compilador da linguagem Ruby para a máquina virtual Parrot capaz de gerar código o código intermediário (PIR) como saída.

A documentação do compilador é simples e clara, para baixar o compilador basta digitar no terminal:

```
$ git clone git://github.com/parrot/cardinal.git
```

Entre as várias opções de instalação, é possível fazê-la utilizando do próprio parrot, para isso basta entrar na pasta onde foi baixado o Cardinal e digitar:

```
$ winxed setup.winxed build
```

Para compilar é necessário estar na pasta de instalação e o comando é:

```
$ parrot cardinal.pbc [arquivo].rb
```

Sendo o arquivo o diretório do arquivo Ruby que se deseja executar, para gerar o PIR o comando é:

```
$ parrot cardinal.pbc -o [output].pir --target=pir [arquivo].rb
```

Sendo output o diretório onde será salvo o arquivo PIR.

Exemplo

Listagem 3.1: Programa nano 01 em Ruby

```
1 # modulo minimo
```

Compilação do Código Ruby:

```
$ parrot /Users/oliveira/cardinal/cardinal/cardinal.pbc -o
../parrot/nano01.pir --target=pir nano01.rb
```

Listagem 3.2: Programa nano 01 em PIR

```
1
2 .HLL "cardinal"
3
4 .namespace []
5 .sub "_block1000" :load :main :anon :subid("10_1471301651.1019")
6     .param pmc param_1002 :optional :named("!BLOCK")
7     .param int has_param_1002 :opt_flag
8 .annotate 'file', "nano01.rb"
9 .annotate 'line', 0
10    .const 'Sub' $P1004 = "11_1471301651.1019"
11    capture_lex $P1004
12 .annotate 'line', 1
13     if has_param_1002, optparam_13
14     new $P100, "Undef"
15     set param_1002, $P100
16 optparam_13:
17     .lex "!BLOCK", param_1002
18     .return ()
19 .end
20
21
22 .HLL "cardinal"
23
24 .namespace []
25 .sub "" :load :init :subid("post12") :outer("10_1471301651.1019")
26 .annotate 'file', "nano01.rb"
27 .annotate 'line', 0
28     .const 'Sub' $P1001 = "10_1471301651.1019"
29     .local pmc block
30     set block, $P1001
31 .end
32
33
34 .HLL "parrot"
35
36 .namespace []
37 .sub "_block1003" :init :load :anon :subid("11_1471301651.1019") :outer("
    10_1471301651.1019")
38 .annotate 'file', "nano01.rb"
39 .annotate 'line', 0
40 $P0 = compreg "cardinal"
41 unless null $P0 goto have_cardinal
42 load_bytecode "cardinal.pbc"
43 have_cardinal:
44     .return ()
45 .end
```

A compilação dos programas Ruby não foi bem sucedida para todos os programas, além disso, programas que utilizavam a linha de código a seguir, que é utilizada para pegar dados do usuário, compilavam mas não funcionavam na máquina virtual.

```
input = gets.chomp
```

Capítulo 4

Códigos LUA e Parrot Assembly (PASM)

Os códigos PASM dessa seção foram feitos manualmente, não foram utilizados compiladores para esse fim.

4.0.1 Nano Programas

Nano 01

Listagem 4.1: Programa nano 01 em Lua

```
1 -- Listagem 1: Modulo minimo que caracteriza um programa
```

Listagem 4.2: Programa nano 01 em PASM

```
1 # Modulo Minimo
2 end
```

Nano 02

Listagem 4.3: Programa nano 02 em Lua

```
1 -- Listagem 2: Declaracao de uma variavel
2
3 -- Em Lua, declaracao de variaveis limitam apenas seu escopo
4 -- As variaveis podem ser local ou global
5 -- local: local x = 10 - precisam ser inicializadas
6 -- global: x = 10      - nao precisam ser inicializadas
7 -- local x            e um programa aceito em lua (declaracao de uma
    variavel local)
8 -- x                  nao e um programa aceito em lua
```

Listagem 4.4: Programa nano 02 em PASM

```
1 # Declarando uma variavel
2
3 end
```

Nano 03

Listagem 4.5: Programa nano 03 em Lua

```
1 -- Atribuicao de um inteiro a uma variavel
2 n = 1
```

Listagem 4.6: Programa nano 03 em PASM

```
1 # Atribuição de um inteiro a uma variavel
2
3 set I1, 1
4 end
```

Nano 04

Listagem 4.7: Programa nano 04 em Lua

```
1 -- Atribuicao de uma soma de inteiros a uma variavel
2 n = 1 + 2
```

Listagem 4.8: Programa nano 04 em PASM

```
1 # Atribuição de uma soma de inteiros a uma variavel
2 set I1, 1
3 set I2, 2
4 add I3, I1, I2
5 end
```

Nano 05

Listagem 4.9: Programa nano 05 em Lua

```
1 -- Inclusao do comando de impressao
2 n = 2
3 print(n)
```

Listagem 4.10: Programa nano 05 em PASM

```
1 # Inclusão do comando de impressão
2 set I1, 2
3 print I1
4 print "\n"
5
6 end
```

Saída:

2

Nano 06

Listagem 4.11: Programa nano 06 em Lua

```
1 -- Listagem 6: Atribuicao de uma subtracao de inteiros a uma variavel
```


4.0

```
2
3 n = 1 - 2
4 print (n)
```

Listagem 4.12: Programa nano 06 em PASM

```
1 # Atribuição de uma subtração de inteiros a uma variável
2 set I1, 1
3 set I2, 2
4 sub I3, I1, I2
5
6 print I3
7 print "\n"
8
9 end
```

Saída:

-1

Nano 07

Listagem 4.13: Programa nano 07 em Lua

```
1 -- Listagem 7: Inclusao do comando condicional
2 n = 1
3 if (n == 1)
4 then
5     print (n)
6 end
```

Listagem 4.14: Programa nano 07 em PASM

```
1 # Inclusão do comando condicional
2
3 set     I1, 1 # atribuição
4 eq      I1, 1, VERDADEIRO
5 branch  FIM
6
7 VERDADEIRO:
8 print   I1
9 print   "\n"
10
11 FIM:
12 end
```

Saída:

1

Nano 08

Listagem 4.15: Programa nano 08 em Lua

```
1 -- Listagem 8: Inclusao do comando condicional com parte senao
2
```

```

3 n = 1
4 if(n == 1)
5 then
6   print(n)
7 else
8   print("0")
9 end

```

Listagem 4.16: Programa nano 08 em PASM

```

1 # Inclusão do comando condicional senão
2
3 set      I1, 1
4 eq      I1, 1, VERDADEIRO
5 print    "0\n"
6 branch   FIM
7
8 VERDADEIRO:
9 print    I1
10 print    "\n"
11
12 FIM:
13 end

```

Saída:

1

Nano 09

Listagem 4.17: Programa nano 09 em Lua

```

1 -- Listagem 9: Atribuicao de duas operacoes aritmeticas sobre inteiros a
   uma variavel
2
3 n = 1 + 1 / 2
4 if (n == 1)
5 then
6   print(n)
7 else
8   print("0")
9 end

```

Listagem 4.18: Programa nano 09 em PASM

```

1 # Atribuição de duas operações aritmeticas sobre inteiros a uma variável
2
3 set      I1, 1
4 set      I2, 2
5 div      I3, I1, I2
6 add      I4, I1, I3
7
8 eq      I4, 1, VERDADEIRO
9 print    "0\n"
10 branch   FIM
11
12 VERDADEIRO:

```

4.0

```
13 print    I4
14 print    "\n"
15
16 FIM:
17 end
```

Saída:

1

Nano 10

Listagem 4.19: Programa nano 10 em Lua

```
1 -- Listagem 10: Atribuicao de duas variaveis inteiras
2 n = 1
3 m = 2
4
5 if (n == m)
6 then
7   print(n)
8 else
9   print("0")
10 end
```

Listagem 4.20: Programa nano 10 em PASM

```
1 # Atribuição de duas variáveis inteiras
2
3 set    I1, 1
4 set    I2, 2
5
6 eq I1, I2, VERDADEIRO
7 print  "0\n"
8 branch FIM
9
10 VERDADEIRO:
11 print  I1
12 print  "\n"
13
14 FIM:
15 end
```

Saída:

0

Nano 11

Listagem 4.21: Programa nano 11 em Lua

```
1 -- Listagem 11: Introducao do comando de repeticao enquanto
2 n = 1
3 m = 2
4 x = 5
5
```

```

6 while(x > n)
7 do
8   n = n + m
9   print(n)
10 end

```

Listagem 4.22: Programa nano 11 em PASM

```

1 # Introdução do comando de repetição enquanto
2
3 set      I1, 1 # n
4 set      I2, 2 # m
5 set      I3, 5 # x
6
7 TESTE:
8 gt       I3, I1, LOOP # gt = greater then
9 branch   FIM
10
11 LOOP:
12 add      I1, I1, I2
13 print    I1
14 print    "\n"
15 branch   TESTE
16
17 FIM:
18 end

```

Saída:

```

3
5

```

Nano 12

Listagem 4.23: Programa nano 12 em Lua

```

1 -- Listagem 12: Comando condicional aninhado em um comando de repeticao
2 n = 1
3 m = 2
4 x = 5
5
6 while(x > n)
7 do
8   if(n == m)
9   then
10    print(n)
11   else
12    print("0")
13   end
14   x = x - 1
15 end

```

Listagem 4.24: Programa nano 12 em PASM

```

1 # Comando condicional aninhado com um de repeticao
2
3 set      I1, 1

```

4.0

```
4 set      I2, 2
5 set      I3, 5
6
7 TESTE_ENQUANTO:
8 gt       I3, I1, LOOP
9 branch   FIM
10
11 LOOP:
12 eq       I1, I2, VERDADEIRO
13 print    "0\n"
14 branch   POS_CONDICIONAL
15
16 VERDADEIRO:
17 print    I1
18 print    "\n"
19
20 POS_CONDICIONAL:
21 dec      I3                # decrementa I3 (x)
22 branch   TESTE_ENQUANTO
23
24 FIM:
25 end
```

Saída:

```
0
0
0
0
```

4.0.2 Micro Programas

Micro 01

Listagem 4.25: Programa micro 01 em Lua

```
1 -- Listagem 13: Converte graus Celsius para Fahrenheit
2
3 -- [[ Funcao: Ler uma temperatura em graus Celsius e apresenta-la
   convertida em graus Fahrenheit. A formula de conversao e : F=(9*C+160)
   / 5, sendo F a temperatura em Fahrenheit e C a temperatura em Celsius.
   --]]
4
5 print("Tabela de Conversão: Celsius -> Fahrenheit")
6 print("Digite a Temperatura em Celsius: ")
7 cel = io.read("*number")
8 far = (9*cel+160)/5
9 print("A nova temperatura é:", far)
```

Listagem 4.26: Programa Micro 01 em PASM

```
1 # Converte graus Celsius para Fahrenheit
2 .loadlib 'io_ops'          # Para fazer IO
3
4 set      S1, "Tabela de Conversao: Celsius -> Fahrenheit\n"
5 set      S2, "Digite a Temperatura em Celsius: "
6 set      S3, "A nova temperatura e: "
```

```

7 set      S4, " graus F."
8
9 print    S1
10 print   S2
11 read    S10, 5
12 set     I1, S10
13
14 mul     I1, I1, 9
15 add     I1, I1, 160
16 div     I1, I1, 5
17
18 print    S3
19 print    I1
20 print    S4
21 print    "\n"
22
23 end

```

Tabela de Conversao: Celsius -> Fahrenheit
 Digite a Temperatura em Celsius: 20
 A nova temperatura e: 68 graus F.

Micro 02

Listagem 4.27: Programa micro 02 em Lua

```

1 -- Listagem 14: Ler dois inteiros e decide qual e maior
2
3 --[[ Funcao : Escrever um algoritmo que leia dois valores inteiro
   distintos e informe qual e o maior --]]
4
5 print("Escreva o primeiro número:")
6 num1 = io.read("*number")
7 print("Escreva o segundo número:")
8 num2 = io.read("*number")
9
10 if(num1 > num2)
11 then
12   print("O primeiro número", num1,"é maior que o segundo",num2)
13 else
14   print("O segundo número", num2, "é maior que o primeiro", num1)
15 end

```

Listagem 4.28: Programa Micro 02 em PASM

```

1 # Ler dois inteiros e decidir qual e maior
2 .loadlib 'io_ops'
3
4 set      S1, "Digite o primeiro numero: "
5 set      S2, "Digite o segundo numero: "
6 set      S3, "o primeiro numero"
7 set      S4, "o segundo numero"
8 set      S5, " e maior que "
9
10 print    S1
11 read    S10, 3
12 set     I1, S10
13 print    S2

```

4.0

```
14 read      S11, 3
15 set       I2, S11
16
17 gt        I1, I2, VERDADEIRO
18 print     S4
19 print     S5
20 print     S3
21 print     "\n"
22 branch    FIM
23
24 VERDADEIRO:
25 print     S3
26 print     S5
27 print     S4
28 print     "\n"
29
30 FIM:
31 end
```

```
Digite o primeiro numero: 10
Digite o segundo numero: 20
o segundo numero e maior que o primeiro numero
Digite o primeiro numero: 20
Digite o segundo numero: 10
o primeiro numero e maior que o segundo numero
```

Micro 03

Listagem 4.29: Programa micro 03 em Lua

```
1 -- Le um numero e verifica se ele esta entre 100 e 200
2 --[[ Funcao: Faca um algoritmo que receba um numero e diga se este numero
   esta no intervalo entre 100 e 200 --]]
3
4 print("Digite um número:")
5 numero = io.read("*number")
6
7 if(numero >= 100)
8 then
9     if(numero <= 200)
10 then
11     print("O número está no intervalo entre 100 e 200")
12 else
13     print("O número não está no intervalo entre 100 e 200")
14 end
15 else
16     print("O número não está no intervalo entre 100 e 200")
17 end
```

Listagem 4.30: Programa Micro 03 em PASM

```
1 # Le um numero e verifica se ele esta entre 100 e 200
2 .loadlib 'io_ops'
3
4 set      S1, "Digite um numero: "
5 set      S2, "O numero esta no intervalo entre 100 e 200\n"
6 set      S3, "O numero nao esta no intervalo entre 100 e 200\n"
7
```

```

8 print      S1
9 read      S10, 3
10 set      I1, S10
11
12 ge      I1, 100, MAIOR_QUE_100
13 branch   NAO_ESTA_NO_INTERVALO
14
15 MAIOR_QUE_100:
16 le      I1, 200, MENOR_QUE_200
17
18 NAO_ESTA_NO_INTERVALO:
19 print    S3
20 branch   FIM
21
22 MENOR_QUE_200:
23 print    S2
24
25 FIM:
26 end

```

```

Digite um numero: 5
O numero nao esta no intervalo entre 100 e 200

Digite um numero: 150
O numero esta no intervalo entre 100 e 200

Digite um numero: 201
O numero nao esta no intervalo entre 100 e 200

```

Micro 04

Listagem 4.31: Programa micro 04 em Lua

```

1 -- Listagem 16: Le numeros e informa quais estao entre 10 e 150
2
3 --[[ Função: Ler 5 numeros e ao final informar quantos numeros estao no
   intervalo entre 10 (inclusive) e 150(inclusive) --]]
4
5 intervalo = 0
6
7 for x=1,5,1
8 do
9   print("Digite um número")
10  num = io.read("*number")
11  if(num >= 10)
12  then
13    if(num <= 150)
14    then
15      intervalo = intervalo + 1
16    end
17  end
18 end
19
20 print("Ao total, foram digitados",intervalo,"números no intervalo entre 10
   e 150")

```

Listagem 4.32: Programa Micro 04 em PASM

4.0

```
1 # Le numeros e informa quais estao entre 10 e 150
2 .loadlib 'io_ops'
3
4 set      S1, "Digite um numero: "
5 set      S2, "Ao total foram digitados "
6 set      S3, " numeros no intervalo entre 10 e 150."
7
8 set      I1, 1
9 set      I2, 0
10
11 LOOP_TESTE:
12 le       I1, 5, INICIO_LOOP
13 branch   FIM
14
15 INICIO_LOOP:
16 print    S1
17 read     S10, 3
18 set      I10, S10
19
20 ge       I10, 10, MAIOR_QUE_10
21 branch   FIM_LOOP
22
23 MAIOR_QUE_10:
24 le       I10, 150, MENOR_QUE_150
25 branch   FIM_LOOP
26
27 MENOR_QUE_150:
28 inc      I2
29
30 FIM_LOOP:
31 inc      I1
32 branch   LOOP_TESTE
33
34
35 FIM:
36 print    S2
37 print    I2
38 print    S3
39 print    "\n"
40 end
```

```
Digite um numero: 50
Digite um numero: 50
Digite um numero: 50
Digite um numero: 50
Digite um numero: 50
Ao total foram digitados 5 numeros no intervalo entre 10 e 150.

Digite um numero: 02
Digite um numero: 03
Digite um numero: 25
Digite um numero: 60
Digite um numero: 160
Ao total foram digitados 2 numeros no intervalo entre 10 e 150.
```

Micro 05

```

1 -- Listagem 17: Le strings e caracteres
2 --[[ Funcao: Escrever um algoritmo que leia o nome e o sexo de 56 pessoas
   e informe o nome e se ela e homem ou mulher. No final informe o total
   de homens e mulheres --]]
3
4 h = 0
5 m = 0
6 for x=1,5,1
7 do
8   print("Digite o nome: ")
9   nome = io.read()
10  print("H - Homem ou M - Mulher")
11  sexo = io.read()
12  if(sexo == 'H') then h = h + 1
13  elseif (sexo == 'M') then m = m + 1
14  else print("Sexo só pode ser H ou M!")
15  end
16 end
17
18 print("Foram inseridos",h,"homens")
19 print("Foram inseridas",m,"mulheres")

```

Listagem 4.34: Programa Micro 05 em PASM

```

1 # Le strings e caracteres
2 .loadlib 'io_ops'
3
4 set      S2, "H - Homem ou M - Mulher: "
5 set      S3, "Sexo so pode ser H ou M!\n"
6 set      S4, "Foram inseridos "
7 set      S5, "Foram inseridas "
8 set      S6, " homens"
9 set      S7, " mulheres"
10
11 set      I1, 1                # x
12 set      I2, 0                # homens
13 set      I3, 0                # mulheres
14
15 LOOP_TESTE:
16 le       I1, 5, INICIO_LOOP
17 branch   FIM
18
19 INICIO_LOOP:
20 print    S2
21 read     S11, 2
22
23 eq       S11, "H\n", HOMEM
24 eq       S11, "M\n", MULHER
25
26 print    S3
27 branch   FIM_LOOP
28
29 HOMEM:
30 inc      I2
31 branch   FIM_LOOP
32
33 MULHER:
34 inc      I3
35

```

```

36 FIM_LOOP:
37 inc      I1
38 branch   LOOP_TESTE
39
40 FIM:
41 print     S4
42 print     I2
43 print     S6
44 print     "\n"
45
46 print     S5
47 print     I3
48 print     S7
49 print     "\n"
50 end

```

```

H - Homem ou M - Mulher: H
H - Homem ou M - Mulher: M
H - Homem ou M - Mulher: H
H - Homem ou M - Mulher: M
H - Homem ou M - Mulher: M
Foram inseridos 2 homens
Foram inseridas 3 mulheres

```

Micro 06

Listagem 4.35: Programa micro 06 em Lua

```

1 -- Escreve um numero lido por extenso
2
3 --[[ Funcao: Faça um algoritmo que leia um número de 1 a 5 e o escreva por
   extenso. Caso o usuario digite um numero que nao esteja nesse
   intervalo, exibir mensagem: numero invalido --]]
4
5 print("Digite um número de 1 a 5")
6 numero = io.read("*number")
7 if(numero == 1) then print("Um")
8 elseif (numero == 2) then print("Dois")
9 elseif (numero == 3) then print("Três")
10 elseif (numero == 4) then print("Quatro")
11 elseif (numero == 5) then print("Cinco")
12 else print("Número Invalido!!!")
13 end

```

Listagem 4.36: Programa Micro 06 em PASM

```

1 # Escrever um numero por extenso
2 .loadlib 'io_ops'
3
4 print      "Digite um numero de 1 a 5: "
5 read      S1, 2
6 set       I1, S1
7
8 eq        I1, 1, UM
9 eq        I1, 2, DOIS
10 eq       I1, 3, TRES
11 eq       I1, 4, QUATRO
12 eq       I1, 5, CINCO

```

```

13
14 print      "Numero invalido!!!"
15 branch    FIM
16
17 CINCO:
18 print      "Cinco"
19 branch    FIM
20
21 QUATRO:
22 print      "Quatro"
23 branch    FIM
24
25 TRES:
26 print      "Tres"
27 branch    FIM
28
29 DOIS:
30 print      "Dois"
31 branch    FIM
32
33 UM:
34 print      "Um"
35
36 FIM:
37 print      "\n"
38 end

```

Digite um numero de 1 a 5: 3
Tres

Micro 07

Listagem 4.37: Programa micro 07 em Lua

```

1 -- Listagem 19: Decide se os numeros sao positivos, zeros ou negativos
2
3 --[[ Funcao: Faca um algoritmo que receba N numeros e mostre positivo,
   negativo ou zero para cada número --]]
4
5 programa = 1
6 while (programa == 1)
7 do
8   print("Digite um numero: ")
9   numero = io.read()
10  numero = tonumber(numero)
11
12  if(numero > 0)
13  then print("Positivo")
14  elseif(numero == 0)
15  then print("O número é igual a 0")
16  elseif(numero < 0)
17  then print("Negativo")
18  end
19
20
21  print("Deseja Finalizar? (S/N) ")
22  opc = io.read("*line")
23

```

```

24  if(opc == "S")
25  then programa = 0
26  end
27 end

```

Listagem 4.38: Programa Micro 07 em PASM

```

1  # Decide se os numeros sao positivos, zeros ou negativos
2  .loadlib 'io_ops'
3
4  LOOP:
5  print      "Digite um numero: "
6  read      S1, 3
7  set       I1, S1
8
9  # Testar se e maior que 0
10 gt        I1, 0, POSITIVO
11 eq        I1, 0, ZERO
12 lt        I1, 0, NEGATIVO
13
14 POSITIVO:
15 print      "Positivo!\n"
16 branch    FINALIZAR
17
18 ZERO:
19 print      "Zero!\n"
20 branch    FINALIZAR
21
22 NEGATIVO:
23 print      "Negativo!\n"
24
25 # Parte de DESEJA FINALIZAR?
26 FINALIZAR:
27 print      "Deseja finalizar? (S/N): "
28 read      S10, 2
29 eq        S10, "S\n", FIM
30 branch    LOOP
31
32 FIM:
33 end

```

```

Digite um numero: 5
Positivo!
Deseja finalizar? (S/N): N
Digite um numero: -5
Negativo!
Deseja finalizar? (S/N): N
Digite um numero: 0
Zero!
Deseja finalizar? (S/N): S

```

Micro 08

Listagem 4.39: Programa micro 08 em Lua

```

1  -- Listagem 20: Decide se um numero e maior ou menor que 10
2
3  numero = 1

```

```

4 while(numero ~= 0)
5 do
6   print("Escreva um numero: ")
7   numero = tonumber(io.read())
8
9   if(numero > 10)
10    then print("O numero",numero,"e maior que 10")
11    else print("O numero",numero,"e menor que 10")
12    end
13 end

```

Listagem 4.40: Programa Micro 08 em PASM

```

1 # Decide se um número é maior ou menor que 10
2 .loadlib 'io_ops'
3
4 set      I1, 1                # variavel numero
5
6 TESTE_LOOP:
7 ne      I1, 0, LOOP
8 branch  FIM
9
10 LOOP:
11 print   "Digite um numero: "
12 read    S10, 3
13 set     I1, S10
14
15 gt      I1, 10, MAIOR
16 print   "O numero "
17 print   I1
18 print   " e menor que 10.\n"
19 branch  TESTE_LOOP
20
21 MAIOR:
22 print   "O numero "
23 print   I1
24 print   " e maior que 10.\n"
25 branch  TESTE_LOOP
26
27 FIM:
28 end

```

```

Digite um numero: 50
O numero 50 e maior que 10.
Digite um numero: 5
O numero 5 e menor que 10.
Digite um numero: 0
O numero 0 e menor que 10.

```

Micro 09

Listagem 4.41: Programa micro 09 em Lua

```

1 -- Listagem 21: Calculo de Precos
2
3 print("Digite o preco: ")
4 preco = tonumber(io.read())
5 print("Digite a venda: ")

```

4.0

```
6 venda = tonumber(io.read())
7
8 if ((venda < 500) or (preco < 30))
9 then novo_preco = preco + (10/100 * preco)
10 elseif ((venda >= 500 and venda < 1200) or (preco >= 30 and preco < 80))
11 then novo_preco = preco + (15/100 * preco)
12 elseif (venda >= 1200 or preco >= 80)
13 then novo_preco = preco - (20/100 * preco)
14 end
15
16 print("O novo preco e: ", novo_preco)
```

Listagem 4.42: Programa Micro 09 em PASM

```
1 # Calculo de precos
2 .loadlib 'io_ops'
3
4
5 print      "Digite o preco (max. 2 digitos): "
6 read      S1, 3
7 set       N1, S1
8 print      "Digite a venda (max. 4 digitos): "
9 read      S1, 5
10 set      N2, S1
11
12 lt       N2, 500, AUMENTAR_10_PORCENTO
13 ge      N1, 30, FALSO1
14
15 AUMENTAR_10_PORCENTO:
16 mul      N3, 10, N1
17 div      N3, N3, 100
18 add      N3, N3, N1
19 branch   FIM
20
21 FALSO1:
22 lt       N2, 500, SEGUNDO_TESTE
23 lt       N2, 1200, AUMENTAR_15_PORCENTO
24 SEGUNDO_TESTE:
25 lt       N1, 30, FALSO2
26 ge      N1, 80, FALSO2
27
28 AUMENTAR_15_PORCENTO:
29 mul      N3, N1, 15
30 div      N3, N3, 100
31 add      N3, N3, N1
32 branch   FIM
33
34 FALSO2:
35 ge      N2, 1200, DIMINUIR_20_PORCENTO
36 lt      N1, 80, FIM
37
38 DIMINUIR_20_PORCENTO:
39 mul      N3, 20, N1
40 div      N3, N3, 100
41 sub      N3, N1, N3
42
43 FIM:
44 print      "O novo preco e: "
45 print      N3
```

```

46 print          "\n"
47 end

```

```

Digite o preco: 10
Digite a venda: 10
O novo preco e: 11

Digite o preco: 40
Digite a venda: 600
O novo preco e: 46

Digite o preco: 90
Digite a venda: 1500
O novo preco e: 72

```

Micro 10

Listagem 4.43: Programa micro 10 em Lua

```

1 --Listagem 22: Calcula o fatorial de um numero
2
3 --[[ Funcao: recebe um numero e calcula recursivamente o fatorial desse nú
    mero --]]
4
5 function fatorial(n)
6     if(n <= 0)
7     then return 1
8     else return (n* fatorial(n-1))
9     end
10 end
11
12 print("Digite um numero: ")
13 numero = tonumber(io.read())
14 fat = fatorial(numero)
15
16 print("O fatorial de", numero, "e: ", fat)

```

Listagem 4.44: Programa Micro 10 em PASM

```

1 # Calcula o fatorial de um numero
2 .loadlib 'io_ops'
3
4 print          "Digite um numero: "
5 read          S1, 2
6 set           I1, S1
7 set           I10, S1
8
9 branch        FATORIAL
10 RETURN:
11 print         "O fatorial de "
12 print         I1
13 print         " e: "
14 print         I10
15 print         "\n"
16
17 end
18
19

```


4.0

```
20 FATORIAL:
21 set      I11, I10
22 dec      I11
23
24 TESTE:
25 eq        I11, 0, RETURN
26 mul       I10, I10, I11
27 dec       I11
28 branch    TESTE
```

```
Digite um numero: 5
O fatorial de 5 e: 120
```

Micro 11

Listagem 4.45: Programa micro 11 em Lua

```
1 -- Listagem 23: Decide se um numero e positivo, zero ou negativo com o
  auxilio de uma funcao.
2
3 --[[ Funcao: recebe um numero e verifica se o numero e positivo, nulo ou
  negativo com o auxilio de uma funcao --]]
4
5 function verifica(n)
6     if(n > 0)
7     then res = 1
8     elseif (n < 0)
9     then res = -1
10    else res = 0
11    end
12
13    return res
14 end
15
16 print("Escreva um numero: ")
17 numero = tonumber(io.read())
18 x = verifica(numero)
19
20 if(x==1)
21 then print("Numero positivo")
22 elseif(x==0)
23 then print("Zero")
24 else print("Numero negativo")
25 end
```

Listagem 4.46: Programa Micro 11 em PASM

```
1 # Decide se um numero e positivo, zero ou negativo com auxilio de uma
  subrotina
2 .loadlib 'io_ops'
3
4 print      "Digite um numero: "
5 read       S1, 3
6 set        I1, S1
7
8 set        I2, 0                # variavel que tera o resultado
9 branch     VERIFICA
10 RETORNO:
```

```
11
12 eq      I2, 1,  POSITIVO
13 eq      I2, 0,  ZERO
14 print   "Negativo\n"
15 branch  FIM
16
17 ZERO:
18 print   "Zero\n"
19 branch  FIM
20
21 POSITIVO:
22 print   "Positivo\n"
23
24 FIM:
25 end
26
27 VERIFICA:
28 gt      I1, 0,  MAIOR
29 lt      I1, 0,  MENOR
30 branch  FIM_SUB
31
32 MENOR:
33 set     I2, -1
34 branch  FIM_SUB
35
36 MAIOR:
37 set     I2, 1
38
39 FIM_SUB:
40 branch  RETORNO
```

Digite um numero: 5
Positivo

Digite um numero: -5
Negativo

Digite um numero: 0
Zero

Capítulo 5

Analizador Léxico

5.1 Análise Léxica

A Análise Léxica tem como principal objetivo facilitar o entedimento do programa para as análises subsequentes do compilador. Essa análise poderia ser feita juntamente com a análise sintática, mas é feita separada por motivos de eficiência, modularização (facilita manutenção e alterações futuras) e por tradição, as linguagens geralmente são criadas com módulos separados para análise léxica e sintática.

As expressões para a análise geralmente são escritas em expressões regulares, assim, os analisadores léxicos são criados como um automato finito determinístico.

Um Analisador Léxico tem como input uma string correspondente a todo o código digitado, essa string é separada em uma lista de caracteres que subsequentemente é separada em tokens. Após a separação em tokens, é trabalho do analisador verificar a correteude léxica do código bem como rotular corretamente cada token reconhecido.

5.2 Analisador Manual

5.2.1 Linguagem a ser interpretada

A linguagem a ser interpretada é uma linguagem simples que reconhece algumas palavras reservadas básicas de todas as linguagens de programação, como: print, if, then, else e comandos como atribuição, soma, subtração e multiplicação, além de reconhecer identificadores e números inteiros, todas as linhas devem ser terminadas com um ponto e vírgula, um exemplo de um programa nessa linguagem:

```
b := 2;
a := 1 + b;
print (a * b);
if1 := a - 2;
if2 := b + 3;
if if1 > 0
then print (if1);
```

```
else print(if2);
```

5.2.2 Autômato reconhecedor da linguagem

O autômato capaz de interpretar a linguagem é um autômato do tipo $M = (ALF, Q, d, q_0, F)$, onde F é o alfabeto de símbolos de entrada, Q são os estados possíveis do autômato, d é a função de transição, q_0 é o estado inicial e F é o conjunto de todos os estados finais do autômato.

ALF = o alfabeto de entrada é qualquer conjunto de palavras sobre o conjunto ASCII 2

Q = inicio, p, pr, pri, prin, print, i, if, t, th, the, then, e, el, els, else, identificador, inteiro, abre-parentese, fecha-parentese, comparador, operador, dois-pontos, atribuicao, ponto-virgula, estado-morto

q_0 = inicio

F = print, if, then, else, identificador, inteiro, abre-parentese, fecha-parentese, comparador, operador, atribuicao, ponto-virgula

d = A função de transição será descrita na forma: (estado atual, simbolo lido, proximo estado), todos as transições que não forem listadas dessa forma levam ao estado morto que invalida a palavra lida.

(inicio, 'p', p)

(inicio, 'i', i)

(inicio, 't', t)

(inicio, 'e', e)

(inicio, '(', abre-parentese)

(inicio, ')', fecha-parentese)

(inicio, '>', comparador)

(inicio, '+', '-', '*', operador)

(inicio, ':', dois-pontos)

(inicio, ';', ponto-virgula)

(inicio, '0'-'9', inteiro)

(inicio, 'a'-'z', 'A'-'Z' ou '_', identificador)

(p, 'r', pr)

(p, 'a'-'z' exceto 'r' ou 'A'-'Z' ou '0'-'9', identificador)

(pr, 'i', pri)

(pr, 'a'-'z' exceto 'i' ou 'A'-'Z' ou '0'-'9', identificador)

(pri, 'n', prin)

(pri, 'a'-'z' exceto 'n' ou 'A'-'Z' ou '0'-'9', identificador)

(prin, 't', print)

(print, 'a'-'z' exceto 't' ou 'A'-'Z' ou '0'-'9', identificador)

(print, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(i, 'f', if)

(i, 'a'-'z' exceto 'f', 'A'-'Z' ou '0'-'9', identificador)

(if, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(e, 'l', el)

(e, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(el, 's', els)

(el, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(els, 'e', else)

(els, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(else, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(t, 'h', th)

(t, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(th, 'e', the)

(th, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(the, 'n', then)

(the, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(then, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(identificador, 'a'-'z' ou 'A'-'Z' ou '0'-'9', identificador)

(inteiro, '0'-'9', inteiro)

(dois-pontos, '=', atribuicao)

(atribuicao, -, estado-morto)

(abre-parentese, -, estado-morto)

(fecha-parentese, -, estado-morto)

(comparador, -, estado-morto)

(operador, -, estado-morto)

(ponto-virgula, -, estado-morto)

5.2.3 Implementação

Convenções

Para facilitar a escrita e o entendimento do código, os estados descritos acima foram codificados como números inteiros da seguinte forma:

Nome do Estado	Inteiro Correspondente
inicio	0
p	1
pr	2
pri	3
prin	4
print	5
i	6
if	7
t	8
th	9
the	10
then	11
e	12
el	13
els	14
else	15
identificador	16
inteiro	17
abre-parentese	18
fecha-parentese	19
comparador	20
operador	21
dois-pontos	22
ponto-virgula	23
branco	24
atribuicao	25
estado-morto	-1

Código do Autômato

Abaixo se encontram as modificações feitas no código já fornecido do analisador léxico para que ele pudesse reconhecer a linguagem descrita mais acima:

Listagem 5.1: Automato reconhecedor da linguagem descrita

```

1  type token =
2  | If
3  | Then
4  | Else
5  | AbreParentese
6  | FechaParentese
7  | Comparador of string
8  | Operador of string
9  | Atribuicao
10 | PontoVirgula
11 | Id of string
12 | Int of string
13 | Print
14 | Branco
15 | EOF
16
17 let lexico (str:entrada) =
18   let trans (e:estado) (c:simbolo) =
19     match (e,c) with
20     | (0, 'p') -> 1
21     | (0, 'i') -> 6
22     | (0, 't') -> 8
23     | (0, 'e') -> 12
24     | (0, '(') -> 18
25     | (0, ')') -> 19
26     | (0, '>') -> 20
27     | (0, '+') -> 21
28     | (0, '-') -> 21
29     | (0, '*') -> 21
30     | (0, ':') -> 22
31     | (0, ';') -> 23
32     | (0, _) when eh_letra c -> 16
33     | (0, _) when eh_digito c -> 17
34     | (0, _) when eh_branco c -> 24
35     | (0, _) ->
36       failwith ("Erro lexico: caracter desconhecido " ^ Char.escaped c)
37
38   | (1, 'r') -> 2
39   | (1, _) when eh_letra c || eh_digito c -> 16
40
41   | (2, 'i') -> 3
42   | (2, _) when eh_letra c || eh_digito c -> 16
43
44   | (3, 'n') -> 4
45   | (3, _) when eh_letra c || eh_digito c -> 16
46
47   | (4, 't') -> 5
48   | (4, _) when eh_letra c || eh_digito c -> 16
49
50   | (5, _) when eh_letra c || eh_digito c -> 16
51
52   | (6, 'f') -> 7

```

```

53
54 | (7, _) when eh_letra c || eh_digito c -> 16
55
56 | (8, 'h') -> 9
57 | (8, _) when eh_letra c || eh_digito c -> 16
58
59 | (9, 'e') -> 10
60 | (9, _) when eh_letra c || eh_digito c -> 16
61
62 | (10, 'n') -> 11
63 | (10, _) when eh_letra c || eh_digito c -> 16
64
65 | (11, _) when eh_letra c || eh_digito c -> 16
66
67 | (12, 'l') -> 13
68 | (12, _) when eh_letra c || eh_digito c -> 16
69
70 | (13, 's') -> 14
71 | (13, _) when eh_letra c || eh_digito c -> 16
72
73 | (14, 'e') -> 15
74 | (14, _) when eh_letra c || eh_digito c -> 16
75
76 | (15, _) when eh_letra c || eh_digito c -> 16
77
78 | (16, _) when eh_letra c || eh_digito c -> 16
79
80 | (17, _) when eh_digito c -> 17
81
82 | (22, '=') -> 25
83
84 | (24, _) when eh_branco c -> 24
85 | _ -> estado_morto
86 and rotulo e str =
87 match e with
88 | 7 -> If
89 | 5 -> Print
90 | 11 -> Then
91 | 15 -> Else
92 | 18 -> AbreParentese
93 | 19 -> FechaParentese
94 | 20 -> Comparador str
95 | 21 -> Operador str
96 | 25 -> Atribuicao
97 | 23 -> PontoVirgula
98 | 1
99 | 2
100 | 3
101 | 4
102 | 6
103 | 8
104 | 9
105 | 10
106 | 12
107 | 13
108 | 14
109 | 16 -> Id str
110 | 17 -> Int str
111 | 24 -> Branco

```



```
112 | _ -> failwith ("Erro lexico: sequencia desconhecida " ^ str)
```

Reconhecimento do Código

Para demonstrar o funcionamento do autômato, será feita a entrada de um exemplo de programa, que foi especificado anteriormente e está novamente listado abaixo:

```
b := 2;
a := 1 + b;
print (a * b);
if1 := a - 2;
if2 := b + 3;
if if1 > 0
then print(if1);
else print(if2);
```

Para iniciar o Ocaml, ler o arquivo e interpretar um código, os seguintes comandos devem ser especificados no terminal:

```
rlwrap ocaml
#use "dfalexer.ml";;
lexico "comando";;

exit 0;;
```

Para testar o exemplo, será feita a entrada do programa como uma única string:

```
#use "dfalexer.ml";;
lexico "b := 2;\na := 1 + b;\nprint (a * b);\nif1 := a - 2;\nif2 := b + 3;\nif if1 > 0\nthen print(if1);\nelse print(if2);";;

- : token list =
[Id "b"; Branco; Atribuicao; Branco; Int "2"; PontoVirgula; Branco; Id "a"
;
Branco; Atribuicao; Branco; Int "1"; Branco; Operador "+"; Branco; Id "b"
;
PontoVirgula; Branco; Print; Branco; AbreParentese; Id "a"; Branco;
Operador "*"; Branco; Id "b"; FechaParentese; PontoVirgula; Branco;
Id "if1"; Branco; Atribuicao; Branco; Id "a"; Branco; Operador "-";
Branco;
Int "2"; PontoVirgula; Branco; Id "if2"; Branco; Atribuicao; Branco;
Id "b"; Branco; Operador "+"; Branco; Int "3"; PontoVirgula; Branco; If;
Branco; Id "if1"; Branco; Comparador ">"; Branco; Int "0"; Branco; Then;
Branco; Print; AbreParentese; Id "if1"; FechaParentese; PontoVirgula;
Branco; Else; Branco; Print; AbreParentese; Id "if2"; FechaParentese;
PontoVirgula; EOF]
```

Para verificar a corretude do automato, testarei também com um exemplo negativo, por exemplo, colocando um @ no meio da string de teste:

```
#use "dfalexer.ml";;
lexico "b := 2;\na := 1 @ + b;\nprint (a * b);\nif1 := a - 2;\nif2 := b + 3;\nif if1 > 0\nthen print(if1);\nelse print(if2);";;

Exception: Failure "Erro lexico: caracter desconhecido @".
```

5.3 Analisador com Ocamllexer

5.3.1 Convenções Léxicas da Linguagem Lua

Na documentação oficial da Linguagem Lua, as convenções léxicas podem ser encontradas na Seção 3.1.

Lua ignora espaços brancos, novas linhas e comentários entre elementos léxicos (tokens).

Nomes (identificadores) em Lua podem ser strings de letras, dígitos e underscore, não podendo começar com um dígito. Identificadores são utilizados para rotular variáveis, tabelas e rótulos.

A Linguagem possui as seguintes palavras reservadas:

and	break	do	else	elseif	end
false	for	function	goto	if	in
local	nil	not	or	repeat	return
then	true	until	while		

Lua é caso sensível, and é uma palavra reservada, porém AND, And, aNd, etc... podem ser utilizadas como identificadores. Por convenção, identificadores que começam com underscore e são seguidos por letras maiúsculas (como _VERSION) são variáveis utilizadas pela linguagem

São outros tokens reconhecidos pela linguagem:

+	-	*	/	%	^	#
==	~=	<=	>=	<	>	=
()	{	}	[]	::
;	:	,	

Strings podem ser limitadas por aspas simples ou duplas e pode conter qualquer caractere de escape contido no C:

```
'\a', '\b', '\f', '\n', '\r', '\t', '\v', '\\', '\"' e '\'
```

Strings em Lua também podem ser criadas utilizando uma notação de colchetes, em n níveis, as seguintes strings são aceitas em lua:

```
[[teste]] [= [teste]=] == [teste]==] ...
```

Uma constante numérica pode ser escrita com uma parte fracionária opcional, marcada por 'e' ou 'E'. Lua também aceita constantes hexadecimais, que começam com '0x' ou '0X', e aceitam um complemento binário que vem após um 'p' ou 'P', as seguintes constantes numéricas são aceitas:

```
3      3.0      3.1416      314.16e-2      0.31416E1
0xff   0x0.1E   0xA23p-4    0X1.921FB54442D18P+1
```

Um comentário começa com `--` fora de uma string, se após os dois hífens não vier um colchete, o comentário é de uma linha, caso contrário, o comentário segue até encontrar um colchete fechando o comentário.

5.3.2 Implementação

A implementação foi feita utilizando Ocaml Lex, uma ferramenta própria para desenvolvimento de analisadores léxicos utilizando autômatos finitos, o código implementado está a seguir.

Listagem 5.2: Arquivo do Reconhecedor léxico de Lua

```

1 {
2   open Lexing
3   open Printf
4
5   let incr_num_linha lexbuf =
6     let pos = lexbuf.lex_curr_p in
7     lexbuf.lex_curr_p <- { pos with
8       pos_lnum = pos.pos_lnum + 1;
9       pos_bol = pos.pos_cnum;
10    }
11
12   let msg_erro lexbuf c =
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 "%d:%d: caracter desconhecido %c" lin col c
17
18   type tokens = ABREPARENTESE
19               | FECHAPARENTESE
20               | ABRECOLCHETE
21               | FECHACOLCHETE
22               | ABRECHAVES
23               | FECHACHAVES
24               | DOISPONTOS
25               | DOISDOISPONTOS
26
27               | PONTOEVIRGULA
28               | VIRGULA
29               | QUADRADO
30
31               | PONTO
32               | PONTOPONTO
33               | PONTOPONTOPONTO
34
35               | ATRIBUICAO
36               | OPERADOR of string
37               | COMPARADOR of string
38
39               | AND
40               | BREAK
41               | DO
42               | ELSE
43               | ELSEIF
44               | END
45               | FALSE

```

```

46         | FOR
47         | FUNCTION
48         | GOTO
49         | IF
50         | IN
51         | LOCAL
52         | NIL
53         | NOT
54         | OR
55         | REPEAT
56         | RETURN
57         | THEN
58         | TRUE
59         | UNTIL
60         | WHILE
61
62         | INT of int
63         | STRING of string
64         | ID of string
65         | EOF
66 }
67
68 let digito = ['0' - '9']
69 let inteiro = digito+
70
71 let letra = ['a' - 'z' 'A' - 'Z']
72 let identificador = letra ( letra | digito | '_' ) *
73
74 let brancos = [' ' '\t'] +
75 let novalinha = '\r' | '\n' | "\r\n"
76
77 let comentario = "-- " [ ^ '\r' '\n' ] *
78
79 rule token = parse
80   brancos      { token lexbuf }
81 | novalinha    { incr_num_linha lexbuf; token lexbuf }
82 | comentario  { token lexbuf }
83 | "--["      { comentario_bloco 0 lexbuf }
84 | '('         { ABREPARENTESE }
85 | ')'         { FECHAPARENTESE }
86 | '['         { ABRECOLCHETE }
87 | ']'         { FECHACOLCHETE }
88 | '{'         { ABRECHAVES }
89 | '}'         { FECHACHAVES }
90
91 | '+'         { OPERADOR "+" }
92 | '-'         { OPERADOR "-" }
93 | '*'         { OPERADOR "*" }
94 | '/'         { OPERADOR "/" }
95 | '%'         { OPERADOR "%" }
96 | '^'         { OPERADOR "^" }
97
98 | "=="        { COMPARADOR "==" }
99 | "~="        { COMPARADOR "~=" }
100 | ">="        { COMPARADOR ">=" }
101 | "<="        { COMPARADOR "<=" }
102 | '>'        { COMPARADOR ">" }
103 | '<'        { COMPARADOR "<" }
104

```

```

105 | '#'           { QUADRADO }
106 | ':'           { DOISPONTOS }
107 | "::"         { DOISDOISPONTOS }
108 | ';'           { PONTOEVIRGULA }
109 | ','           { VIRGULA }
110 | '.'           { PONTO }
111 | ".."          { PONTOPONTO }
112 | "..."       { PONTOPONTOPONTO }
113
114 | "="          { ATRIBUICAO }
115 | inteiro as num { let numero = int_of_string num in
116 |               INT numero }
117 | "and"         { AND }
118 | "break"       { BREAK }
119 | "do"          { DO }
120 | "else"        { ELSE }
121 | "elseif"      { ELSEIF }
122 | "end"         { END }
123 | "false"       { FALSE }
124 | "for"         { FOR }
125 | "function"    { FUNCTION }
126 | "goto"        { GOTO }
127 | "if"          { IF }
128 | "in"          { IN }
129 | "local"       { LOCAL }
130 | "nil"         { NIL }
131 | "not"         { NOT }
132 | "or"          { OR }
133 | "repeat"      { REPEAT }
134 | "return"      { RETURN }
135 | "then"        { THEN }
136 | "true"        { TRUE }
137 | "until"       { UNTIL }
138 | "while"       { WHILE }
139
140 | identificador as id { ID id }
141 | '"'          { let buffer = Buffer.create 1 in
142 |               let str = leia_string buffer lexbuf in
143 |               STRING str }
144 | _ as c { failwith (msg_erro lexbuf c) }
145 | eof    { EOF }
146 and comentario_bloco n = parse
147   "]" ]" { if n=0 then token lexbuf
148 |         else comentario_bloco (n-1) lexbuf }
149 | "--[" [ { comentario_bloco (n+1) lexbuf }
150 | _        { comentario_bloco n lexbuf }
151 | eof      { failwith "Comentário não fechado" }
152 and leia_string buffer = parse
153   '"' { Buffer.contents buffer}
154 | "\\t" { Buffer.add_char buffer '\t'; leia_string buffer lexbuf }
155 | "\\n" { Buffer.add_char buffer '\n'; leia_string buffer lexbuf }
156 | '\\\| '"' { Buffer.add_char buffer '\\\|'; leia_string buffer lexbuf }
157 | '\\\| '\\\|' { Buffer.add_char buffer '\\\|'; leia_string buffer lexbuf }
158 | _ as c { Buffer.add_char buffer c; leia_string buffer lexbuf }
159 | eof    { failwith "A string não foi fechada"}

```

Além disso, é utilizado o arquivo carregador, fornecido em aula, para que se possa testar corretamente o analisador.

Listagem 5.3: Carregador

```

1 #load "lualexer.cmo";;
2
3 let rec tokens lexbuf =
4   let tok = Lualexer.token lexbuf in
5   match tok with
6   | Lualexer.EOF -> [Lualexer.EOF]
7   | _ -> tok :: tokens lexbuf
8 ;;
9
10 let lexico str =
11   let lexbuf = Lexing.from_string str in
12   tokens lexbuf
13 ;;
14
15 let lex arq =
16   let ic = open_in arq in
17   let lexbuf = Lexing.from_channel ic in
18   let toks = tokens lexbuf in
19   let _ = close_in ic in
20   toks

```

5.3.3 Testes

Para compilar e executar corretamente o Analisador Léxico os seguintes comandos devem ser utilizados:

```

ocamllex lualexer.mll
ocamlc lualexer.ml

rlwrap ocaml
#use "carregador.ml";;

```

Para testar todos os tokens, foi escrito o seguinte arquivo de teste:

Listagem 5.4: Teste de Tokens

```

1 --[[
2 comentario
3 de mais
4 de uma
5 linha
6 ]]
7
8 --[[
9 --[[
10 comentario
11 aninhado
12 ]]
13 ]]
14
15
16 -- OPERADORES
17 -- soma
18 2 + 2

```

5.3

```
19 -- subtracao
20 2 - 2
21 -- multiplicacao
22 2 * 2
23 -- divisao
24 2 / 2
25 -- resto de divisao
26 2 % 2
27 -- exponenciacao
28 2 ^ 2
29
30 -- COMPARADORES
31 -- igualdade
32 2 == 2
33 -- menor igual
34 2 <= 2
35 -- maior igual
36 2 >= 2
37 -- menor
38 2 < 2
39 -- maior
40 2 > 2
41
42
43 -- inteiro
44 1234
45 23908423
46 123124
47 1032490
48 324932
49 3294
50
51 -- identificadores
52 identificador
53 m
54 n
55 i
56 j
57 k
58 l
59
60 -- outros tokens em lua
61 #
62 :
63 ::
64 ;
65 ,
66 .
67 ..
68 ...
69
70 -- abre e fecha parentese
71 (
72 )
73 [
74 ]
75 {
76 }
77
```

```

78 -- atribuicao
79 =
80
81 -- palavras reservadas
82 and
83 break
84 do
85 else
86 elseif
87 end
88 false
89 for
90 function
91 goto
92 if
93 in
94 local
95 nil
96 not
97 or
98 repeat
99 return
100 then
101 true
102 until
103 while
104
105 -- strings
106 "hello world"
107 "helloworld123"
108 "hello\\world"
109 "hello\"aworld"
110 "hello\nworld"
111 "hello\
112 world"
113
114 -- invalidos
115
116 -- _identificador
117
118 -- @

```

Que fornece o seguinte resultado:

```

# lex "teste.lua";
- : Lualexer.tokens list =
[Lualexer.INT 2; Lualexer.OPERADOR "+"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.OPERADOR "-"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.OPERADOR "*"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.OPERADOR "/"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.OPERADOR "%"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.OPERADOR "^"; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.COMPARADOR "==" ; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.COMPARADOR "<=" ; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.COMPARADOR ">=" ; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.COMPARADOR "<" ; Lualexer.INT 2; Lualexer.INT 2;
 Lualexer.COMPARADOR ">" ; Lualexer.INT 2; Lualexer.INT 1234;
 Lualexer.INT 23908423; Lualexer.INT 123124; Lualexer.INT 1032490;
 Lualexer.INT 324932; Lualexer.INT 3294; Lualexer.ID "identificador";

```



```

Lualexer.ID "m"; Lualexer.ID "n"; Lualexer.ID "i"; Lualexer.ID "j";
Lualexer.ID "k"; Lualexer.ID "l"; Lualexer.QUADRADO; Lualexer.DOISPONTOS;
Lualexer.DOISDOISPONTOS; Lualexer.PONTOEVIRGULA; Lualexer.VIRGULA;
Lualexer.PONTO; Lualexer.PONTOPONTO; Lualexer.PONTOPONTOPONTO;
Lualexer.ABREPARENTESE; Lualexer.FECHAPARENTESE; Lualexer.ABRECOLCHETE;
Lualexer.FECHACOLCHETE; Lualexer.ABRECHAVES; Lualexer.FECHACHAVES;
Lualexer.ATRIBUICAO; Lualexer.AND; Lualexer.BREAK; Lualexer.DO;
Lualexer.ELSE; Lualexer.ELSEIF; Lualexer.END; Lualexer.FALSE; Lualexer.
    FOR;
Lualexer.FUNCTION; Lualexer.GOTO; Lualexer.IF; Lualexer.IN; Lualexer.
    LOCAL;
Lualexer.NIL; Lualexer.NOT; Lualexer.OR; Lualexer.REPEAT; Lualexer.RETURN
    ;
Lualexer.THEN; Lualexer.TRUE; Lualexer.UNTIL; Lualexer.WHILE;
Lualexer.STRING "hello world"; Lualexer.STRING "helloworld123";
Lualexer.STRING "hello\\world"; Lualexer.STRING "hello\\"aworld";
Lualexer.STRING "hello\\nworld"; Lualexer.STRING "hello\\\\nworld";
Lualexer.EOF]

```

5.3.4 Futuras Correções

O Analisador Léxico ainda não está totalmente de acordo com as regras da linguagem Lua e ainda não fornece algumas informações importantes para o processo de correção do código.

- O Analisador implementado é capaz de reconhecer comentários de múltiplas linhas aninhados, desde que haja um fechamento correspondente à cada abertura. Isso não é suportado na linguagem Lua.
- Quando há um erro Léxico, há um erro ao apontar onde exatamente está o erro, o Analisador desconsidera os comentários e não conta as linhas que ele ocupa.
- A Linguagem Lua oferece declaração de strings em níveis, começando com [= e terminando em |=], a quantidade de "=" indica o nível da declaração, isso também não é suportado atualmente pelo analisador.

Capítulo 6

Analizador Sintático

Um analisador sintático tem como tarefa recombinar os tokens produzidos pela análise léxica e gerar uma estrutura que reflete a estrutura da linguagem, essa estrutura é a árvore sintática. Além disso, na etapa de Análise Sintática, qualquer texto que viole as regras sintáticas da linguagem deve ser rejeitado.

As linguagens adotadas por linguagens de programação são linguagens livres de contexto, que são interpretadas por autômatos com pilha

6.1 Parser Preditivo

O parser preditivo é uma implementação para Analisadores Sintáticos que utiliza do método recursivo descendente, pelo método LL1, isto é, a entrada será lida da esquerda para a direita, a árvore será construída com precedência mais à esquerda e apenas um token da entrada será analisado por vez.

6.1.1 Linguagem Analisada

A linguagem a ser analisada pelo Analisador Sintático implementado por um Parser Preditivo é da seguinte forma:

Símbolos não Terminais = S, X, Y, Z

Símbolos Terminais = a, b, c, d, e, f

Símbolo Inicial = S

Regras de Produção = $\{S \rightarrow XYZ; X \rightarrow aXb; X \rightarrow ; Y \rightarrow cYZcX; Y \rightarrow d; Z \rightarrow eZYe; Z \rightarrow f\}$

6.1.2 Arquivo Sintático

Listagem 6.1: Arquivo Sintático

```

1 type tokens = A
2           | B
3           | C
4           | D
5           | E
6           | F
7           | VAZIO
8           | EOF

```

6.1.3 Arquivo Léxico

Listagem 6.2: Arquivo Léxico

```

1 {
2   open Lexing
3   open Printf
4   open Sintatico
5
6
7   let incr_num_linha lexbuf =
8     let pos = lexbuf.lex_curr_p in
9     lexbuf.lex_curr_p <- { pos with
10       pos_lnum = pos.pos_lnum + 1;
11       pos_bol = pos.pos_cnum;
12     }
13
14   let msg_erro lexbuf c =
15     let pos = lexbuf.lex_curr_p in
16     let lin = pos.pos_lnum
17     and col = pos.pos_cnum - pos.pos_bol - 1 in
18     sprintf "%d-%d: caracter desconhecido %c" lin col c
19
20
21 }
22
23 let digito = ['0' - '9']
24 let inteiro = digito+
25
26 let letra = ['a' - 'z' 'A' - 'Z']
27 let identificador = letra ( letra | digito | '_' ) *
28
29 let brancos = [ ' ' '\t' ] +
30 let novalinha = '\r' | '\n' | "\r\n"
31
32 let comentario = "//" [ ^ '\r' '\n' ] *
33
34 rule token = parse
35   brancos      { token lexbuf }
36 | novalinha    { incr_num_linha lexbuf; token lexbuf }
37 | comentario  { token lexbuf }
38 | "/" *       { comentario_bloco 0 lexbuf }
39 | 'a'         { A }
40 | 'b'         { B }
41 | 'c'         { C }
42 | "d"         { D }
43 | "e"         { E }
44 | "f"         { F }
45 | _ as c     { failwith (msg_erro lexbuf c) }

```

```

46 | eof          { EOF }
47 and comentario_bloco n = parse
48   "*" / "      { if n=0 then token lexbuf
49                 else comentario_bloco (n-1) lexbuf }
50 | "/" * "      { comentario_bloco (n+1) lexbuf }
51 | _            { comentario_bloco n lexbuf }
52 | eof          { failwith "Comentário não fechado" }

```

6.1.4 Gerador da Arvore Sintática

Listagem 6.3: Arvore Sintatica

```

1 (* Parser preditivo *)
2 #load "lexico.cmo";;
3 open Sintatico;;
4
5 type regra = REGRA_S_XYZ of regra * regra * regra
6             | REGRA_X_AXB of tokens * regra * tokens
7             | REGRA_Y_CYZCX of tokens * regra * regra * tokens * regra
8             | REGRA_Y_D of tokens
9             | REGRA_Z_EZYE of tokens * regra * regra * tokens
10            | REGRA_Z_F of tokens
11            | REGRA_X_VAZIO
12
13 type comando = If of expressao * comando * comando
14              | Bloco of comando list
15              | Print of expressao
16 and expressao = Igual of tokens * tokens
17
18 let tk = ref EOF (* variável global para o token atual *)
19 let lexbuf = ref (Lexing.from_string "")
20
21 (* lê o próximo token *)
22 let prox () = tk := Lexico.token !lexbuf
23
24 let to_str tk =
25   match tk with
26   | A -> "a"
27   | B -> "b"
28   | C -> "c"
29   | D -> "d"
30   | E -> "e"
31   | F -> "f"
32   | EOF -> "eof"
33
34 let erro esp =
35   let msg = Printf.sprintf "Erro: esperava %s mas encontrei %s"
36                             esp (to_str !tk)
37   in
38   failwith msg
39
40 let consome t = if (!tk == t) then prox() else erro (to_str t)
41
42 let rec ntS () =
43   match !tk with
44   | A -> let expx = ntX () in
45          let expy = ntY () in
46          let expz = ntZ () in

```

```

47         REGRA_S_XYZ (expx, expy, expz)
48     | C      ->
49         let expx = ntX() in
50         let expy = ntY() in
51         let expz = ntZ() in
52         REGRA_S_XYZ (expx, expy, expz)
53     | D      ->
54         let expx = ntX() in
55         let expy = ntY() in
56         let expz = ntZ() in
57         REGRA_S_XYZ (expx, expy, expz)
58     | _ -> erro "a, c ou d"
59 and ntX () =
60     match !tk with
61     A -> let _ = consome A in
62         let cmd = ntX() in
63         let _ = consome B in
64         REGRA_X_AXB (A, cmd, B)
65     | _ -> REGRA_X_VAZIO
66 and ntY () =
67     match !tk with
68     C -> let _ = consome C in
69         let cmdy = ntY() in
70         let cmdz = ntZ() in
71         let _ = consome C in
72         let cmdx = ntX() in
73         REGRA_Y_CYZCX (C, cmdy, cmdz, C, cmdx)
74     | D -> let _ = consome D in
75         REGRA_Y_D D
76     | _ -> erro "c ou d"
77 and ntZ () =
78     match !tk with
79     E -> let _ = consome E in
80         let cmdz = ntZ() in
81         let cmdy = ntY() in
82         let _ = consome E in
83         REGRA_Z_EZYE (E, cmdz, cmdy, E)
84     | F -> let _ = consome F in
85         REGRA_Z_F F
86     | _ -> erro "e ou f"
87
88 let parser str =
89     lexbuf := Lexing.from_string str;
90     prox (); (* inicializa o token *)
91     let arv = ntS () in
92     match !tk with
93     EOF -> let _ = Printf.printf "Ok!\n" in arv
94     | _ -> erro "fim da entrada"
95
96
97 let teste () =
98     let entrada =
99         "abcdfcf"
100     in
101     parser entrada

```

6.1.5 Testes

Teste com a palavra "abcdcf":

```
- : regra =
  REGRA_S_XYZ (REGRA_X_AXB (A, REGRA_X_VAZIO, B),
    REGRA_Y_CYZCX (C, REGRA_Y_D D, REGRA_Z_F F, C, REGRA_X_VAZIO), REGRA_Z_F
    F)
```

Teste com a palavra "bd", uma palavra que não é reconhecida na linguagem:

```
Exception: Failure "Erro: esperava a, c ou d mas encontrei b".
```

6.2 Análise Sintática com Menhir

Menhir é um gerador de Parsers LR(1) para Linguagem de Programação OCaml. Para instalar o Menhir basta entrar com o comando do homebrew:

```
brew install opam
brew install menhir
```

Além disso, também é utilizado menhirLib, ocamlfind e ocamlbuild para compilar todos os arquivos necessários, para obter esses arquivos o comando é:

```
opam init
opam install menhirLib
opam install ocamlfind
opam install ocamlbuild
eval `opam config env`
```

6.2.1 Arquivo Ocamlinit

O Arquivo Ocamlinit é um arquivo que deve ser nomeado como ".ocamlinit" e ele é executado toda vez que o Ocaml é inicializado. Ele é útil para importar automaticamente todas as dependências necessárias.

6.2.2 Tipos de Parser

Um parser analisa uma sequência de entrada e verifica se a entrada está de acordo com a linguagem descrita.

LL(1)

O Parser LL(1) é um analisador sintático descendente que lê entrada da esquerda para a direita e gerar uma derivação mais à esquerda, por isso é chamado de LL (left-left do inglês). O "1" significa que o parser utiliza apenas um token por vez.

LR(1)

O Parser LR(1) também utiliza apenas um token para a predição, lê a entrada da esquerda para a direita e produz uma derivação mais a direita, por isso o nome LR (left-right do inglês).

6.2.3 Análise Sintática Linguagem Lua

Listagem 6.4: Regras da linguagem Lua no Menhir

```

1  %{
2      open Ast
3  %}
4
5  %token ABREPARENTESE
6  %token FECHAPARENTESE
7  %token ABRECOLCHETE
8  %token FECHACOLCHETE
9  %token ABRECHAVES
10 %token FECHACHAVES
11 %token DOISPONTOS
12 %token DOISDOISPONTOS
13
14 %token PONTOEVIRGULA
15 %token VIRGULA
16 %token QUADRADO
17
18 %token PONTO
19 %token PONTOPONTO
20 %token PONTOPONTOPONTO
21
22 %token ATRIBUICAO
23 %token SOMA
24 %token SUBTRACAO
25 %token MULTIPLICACAO
26 %token DIVISAO
27 %token MODULO
28 %token EXPONENCIACAO
29 %token IGUALDADE
30 %token DIFERENTE
31 %token MENORIGUAL
32 %token MAIORIGUAL
33 %token MENOR
34 %token MAIOR
35
36 %token AND
37 %token BREAK
38 %token DO
39 %token ELSE
40 %token ELSEIF
41 %token END
42 %token FALSE
43 %token FOR
44 %token FUNCTION
45 %token GOTO
46 %token IF
47 %token IN

```

```

48 %token LOCAL
49 %token NIL
50 %token NOT
51 %token OR
52 %token REPEAT
53 %token RETURN
54 %token THEN
55 %token TRUE
56 %token UNTIL
57 %token WHILE
58
59 %token <int> INT
60 %token <string> STRING
61 %token <string> ID
62 %token <float> FLOAT
63 %token EOF
64
65 %start <Ast.programa> chunk
66
67 %%
68
69 chunk:
70   | b=block EOF { Programa(b) }
                                                    (*
              OK *)
71   ;
72
73 block:
74   | s=stat* r=retstat? { Bloco(s,r) }
                                                    (* OK *)
75   ;
76
77 stat:
78   | PONTOEVIRGULA { PontoeVirgula }
                                                    (* OK
              *)
79   | vl=varlist ATRIBUICAO el=explist { Atribuicao(vl, el) }
                                                    (* OK *)
80   | f=functioncall { StatFunctionCall(f) }
                                                    (* OK *)
81   | l=label { StatLabel(l) }
              (* OK *)
82   | BREAK { Break }
              (* OK *)
83   | GOTO i=ID { Goto(i) }
              (* OK *)
84   | DO b=block END { StatBloco(b) }
                                                    (* OK
              *)
85   | WHILE e=exp DO b=block END { While(e,b) }
                                                    (* OK *)
86   | REPEAT b=block UNTIL e=exp { Repeat(b,e) }
                                                    (* OK *)
87   | IF e=exp THEN b=block el=elseif_rule* es=else_block_rule? END { If(e,b
              ,el,es) }
              (* OK *)

```


6.2

```

88 | FOR i=ID ATRIBUICAO e=exp VIRGULA ec=exp c=comma_exp_rule? DO b=block
    END { For(i,e,ec,c,b) } (* OK *)
89 | FOR nl=namelist IN el=explist DO b=block END { Forlist(nl,el,b) }
    (* OK *)
90 | FUNCTION fn=funcname fb=funcbody { FunctionDefinition(fn,fb) }
    (* OK *)
91 | LOCAL n=namelist a=atribuicao_explist_rule? { Local(n,a) }
    (* OK *)
92 ;
93
94 (* AUXILIARES PARA A REGRA STAT *)
95 elseif_rule:
96 | ELSEIF e=exp THEN b=block { Elseif(e,b) }
    (* OK *)
97 ;
98
99 else_block_rule:
100 | ELSE b=block { Else(b) }
    (* OK *)
101 ;
102
103 comma_exp_rule:
104 | VIRGULA e=exp { Virgula(e) }
    (*
    OK *)
105 ;
106 atribuicao_explist_rule:
107 | ATRIBUICAO e=explist { Atribuicao(e) }
    (* OK *)
108 ;
109 (* ----- *)
110
111 retstat:
112 | RETURN e=explist? PONTOEVIRGULA? { Retorno (e) }
    (* OK *)
113 ;
114
115 label:
116 | DOISDOISPONTOS i=ID DOISDOISPONTOS { Label(i) }
    (* OK *)
117 ;
118
119 funcname:
120 | i=ID p=ponto_id_rule* d=doispontos_id_rule? { FuncName (i, p, d) }
    (* OK *)
121 ;
122
123 (* auxiliares para funcname *)
124
125 ponto_id_rule:
126 | PONTO i=ID { Ponto(i) }
    (* OK *)
127 ;
128 doispontos_id_rule:
129 | DOISPONTOS i=ID { Doispontos(i) }
    (* OK *)
130 ;

```

```

131  (* ----- *)
132
133 varlist:
134   | v1=var v=virgula_var_rule* { Varlist(v1, v) }
135                                     (* OK *)
136   ;
137   (* auxiliares var list *)
138   virgula_var_rule:
139   | VIRGULA v=var { Variavel(v) }
140                                     (* OK
141                                     *)
142   ;
143   (* ----- *)
144   var:
145   | i=ID { Identificador(i) }
146                                     (* OK *)
147   | p=prefixexp ABRECOLCHETE e=exp FECHACOLCHETE { VarCol(p,e) }
148                                     (* OK *)
149   | p=prefixexp PONTO i=ID { SeparadoPonto(p,i) }
150                                     (* OK *)
151   ;
152   namelist:
153   | i=ID vi=virgula_id_rule* { NameList(i, vi) }
154                                     (* OK *)
155   ;
156   (* namelist auxiliares *)
157   virgula_id_rule:
158   | VIRGULA i=ID { Name(i) }
159                                     (* OK *)
160   ;
161   (* ----- *)
162   explist:
163   | e=exp e2=virgula_exp_rule* { Explist(e, e2) }
164                                     (* OK *)
165   ;
166   (* auxiliar para explist *)
167   virgula_exp_rule:
168   | VIRGULA e=exp { Expression(e) }
169                                     (* OK *)
170   ;
171   (* ----- *)
172   exp:
173   | NIL { Nil }
174                                     (* OK *)
175   | FALSE { False }
176                                     (* OK
177                                     *)
178   | TRUE { True }
179                                     (*
180                                     OK *)

```

6.2

```

171 | i=INT { Int(i) }
                                           (* OK
                                           *)
172 | f=FLOAT { Float(f) }
                                           (* OK *)
173 | s=STRING { String(s) }
                                           (* OK *)
174 | PONTOPONTOPONTO { Pontopontoponto }
                                           (* OK *)
175 | f=functiondef { FunctionDef(f) }
                                           (* OK *)
176 | p=prefixexp { ExpPrefixExp(p) }
                                           (* OK *)
177 | t=tableconstructor { TableConstructor(t) }
                                           (* OK *)
178 | e1=exp b=binop e2=exp { ExpBinop(b,e1,e2) }
                                           (* OK *)
179 | u=unop e=exp { ExpUnop(u,e) }
                                           (* OK *)
180 ;
181
182 prefixexp:
183 | v=var { PrefixExpVar (v) }
                                           (* OK *)
184 | f=functioncall { PrefixExpFunctionCall(f) }
                                           (* OK *)
185 | ABREPARENTESE e=exp FECHAPARENTESE { PrefixExpParentese(e) }
                                           (* OK *)
186 ;
187
188 functioncall:
189 | p=prefixexp a=args { FunctionCallPA(p, a) }
                                           (* OK *)
190 | p=prefixexp DOISPONTOS i=ID a=args { PrefixoDoisPontosIdArgs(p,i,a) }
                                           (* OK *)
191 ;
192
193 args:
194 | ABREPARENTESE e=explist? FECHAPARENTESE { ArgsExp(e) }
                                           (* OK *)
195 | t=tableconstructor { TableConstructor(t) }
                                           (* OK *)
196 | s=STRING { Args(s) }
                                           (* OK *)
197 ;
198
199 functiondef:
200 | FUNCTION f=funcbody { FunctionDef(f) }
                                           (* OK *)
201 ;
202
203 funcbody:
204 | ABREPARENTESE p=parlist? FECHAPARENTESE b=block END { FuncBody (p,b) }
                                           (* OK *)
205 ;
206
207 parlist:
208 | n=namelist v=virgula_tres_pontos_rule? { NameListVirgula(n,v) }
                                           (* OK *)

```

```

209 | PONTOPONTOPONTO { Pontopontoponto }
                                           (* OK *)
210 ;
211 (* parlist auxiliar *)
212 virgula_tres_pontos_rule:
213 | VIRGULA PONTOPONTOPONTO { VirgulaPPP }
                                           (* OK *)
214 ;
215 (* ----- *)
216
217 tableconstructor:
218 | ABRECHAVES f=fieldlist? FECHACHAVES { FieldList(f) }
                                           (* OK *)
219 ;
220
221 fieldlist:
222 | f=field ffr=fieldsep_field_rule* fs=fieldsep? { FieldLists(f, ffr, fs)
    }
    (* OK *)
223 ;
224 (* auxiliar fieldlist *)
225 fieldsep_field_rule:
226 | fs=fieldsep f=field { FieldSepField(fs, f) }
    (* OK *)
227 ;
228 (* ----- *)
229
230 field:
231 | ABRECOLCHETE e=exp FECHACOLCHETE ATRIBUICAO e2=exp { Campo1(e, e2) }
    (* OK *)
232 | i=ID ATRIBUICAO e=exp { Campo2(i,e) }
    (* OK *)
233 | e=exp { Campo3(e) }
    (* OK *)
234 ;
235
236 fieldsep:
237 | VIRGULA { Virgula }
    (* OK *)
238 | PONTOEVIRGULA { PontoEVirgula }
    (* OK *)
239 ;
240
241 binop:
242 | SOMA { Soma }
    (*
    OK *)
243 | SUBTRACAO { Subtracao }
    (* OK *)
244 | MULTIPLICACAO { Multiplicacao }
    (* OK *)
245 | DIVISAO { Divisao }
    (* OK *)
246 | EXPONENCIACAO { Exponenciacao }
    (* OK *)
247 | MODULO { Modulo }
    (* OK
    *)
248 | PONTOPONTO { Pontoponto }
    (* OK *)

```

```

249 | MENOR { Menor }
                                     (* OK
                                     *)
250 | MENORIGUAL { MenorIgual }
                                     (* OK *)
251 | MAIOR { Maior }
                                     (* OK
                                     *)
252 | MAIORIGUAL { MaiorIgual }
                                     (* OK *)
253 | IGUALDADE { Igualdade }
                                     (* OK *)
254 | DIFERENTE { Diferente }
                                     (* OK *)
255 | AND { And }
                                     (* OK *)
256 | OR { Or }
                                     (* OK *)
257 ;
258
259 unop:
260 | SUBTRACAO { Decremento }
                                     (* OK *)
261 | NOT { Not }
                                     (* OK *)
262 | QUADRADO { Quadrado }
                                     (* OK *)
263 ;

```

Listagem 6.5: Arvore Sintática

```

1 (* The type of the abstract syntax tree (AST). *)
2 type programa = Programa of bloco
3 and bloco = Bloco of stat_list * retstat_option
4
5 and stat_list = stat list
6 and stat =
7     | Stat of string
8     | PontoeVirgula
9     | Break
10    | Atribuicao of varlist * explist
11    | StatFunctionCall of functioncall
12    | FunctionDefinition of funcname * funcbody
13    | If of exp * bloco * elseif list * else_r option
14    | StatBloco of bloco
15    | StatLabel of label
16    | Goto of string
17    | Local of namelist * atribuicao_explist_rule option
18    | While of exp * bloco
19    | Repeat of bloco * exp
20    | Forlist of namelist * explist * bloco
21    | For of string * exp * exp * comma_exp_rule option * bloco
22
23 and elseif = Elseif of exp * bloco
24 and else_r = Else of bloco
25 and label = Label of string

```

```

26 and atribuicao_explist_rule = Atribuicao of explist
27 and comma_exp_rule = Virgula of exp
28
29 and varlist = Varlist of var * variavel list
30 and variavel = Variavel of var
31 and var =
32     | Identificador of string
33     | SeparadoPonto of prefixexp * string
34     | VarCol of prefixexp * exp
35
36 and funcname = FuncName of string * ponto_id_rule list *
    doispontos_id_rule option
37
38 and ponto_id_rule = Ponto of string
39 and doispontos_id_rule = Doispontos of string
40
41 and namelist = NameList of string * name list
42 and name = Name of string
43 and virgula_tres_pontos_rule = VirgulaPPP
44
45
46 and explist = Explist of exp * expaux list
47 and expaux = Expression of exp
48 and exp =
49     | Nil
50     | True
51     | False
52     | Int of int
53     | Float of float
54     | String of string
55     | ExpVar of var
56     | ExpFunctioncall of functioncall
57     | ExpPrefixExp of prefixexp
58     | ExpAExpF of exp
59     | Exp of string
60     | ExpBinop of binop * exp * exp
61     | ExpUnop of unop * exp
62     | Pontopontoponto
63     | FunctionDef of functiondef
64     | TableConstructor of tableconstructor
65
66 and functiondef = FunctionDef of funcbody
67 and funcbody = FuncBody of parlist option * bloco
68 and parlist =
69     | NameListVirgula of namelist * virgula_tres_pontos_rule option
70     | Pontopontoponto
71
72 and prefixexp =
73     | PrefixExpVar of var
74     | PrefixExpFunctionCall of functioncall
75     | PrefixExpParentese of exp
76
77 and functioncall =
78     | FunctionCallPA of prefixexp * args
79     | PrefixoDoisPontosIdArgs of prefixexp * string * args
80
81 and args =
82     | ArgsExp of explist option
83     | Args of string

```

```

84         | TableConstructor of tableconstructor
85
86 and tableconstructor = FieldList of fieldlist option
87 and fieldlist = FieldLists of field * fieldsep_field_rule list * fieldsep
    option
88 and fieldsep_field_rule = FieldSepField of fieldsep*field
89 and fieldsep =
90     | Campo1 of exp * exp
91     | Campo2 of string * exp
92     | Campo3 of exp
93 and fieldsep =
94     | Virgula
95     | PontoEVirgula
96
97 and retstat_option = retstat option
98 and retstat =
99     | Retorno of explist option
100
101 and binop =
102     | Soma
103     | Subtracao
104     | Multiplicacao
105     | Divisao
106     | Exponenciacao
107     | Modulo
108     | Pontoponto
109     | Maior
110     | MaiorIgual
111     | Menor
112     | MenorIgual
113     | Igualdade
114     | Diferente
115     | And
116     | Or
117
118 and unop =
119     | Decremento
120     | Not
121     | Quadrado

```

Arquivo Makefile com os comandos necessários para compilar os arquivos e gerar as mensagens de erro:

Listagem 6.6: Makefile

```

1 erros:
2   menhir -v --list-errors sintatico.mly > sintatico_erros.msg
3
4 env:
5   eval `opam config env`
6
7 msg:
8   menhir -v --list-errors sintatico.mly --compile-errors sintatico.msg >
    erroSint.ml
9
10 build:
11   ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib sintaticoTest.byte

```

Erros: gera os erros possíveis do automato e fornece o espaço para colocar as devidas mensagens de erro que vai ser demonstrada no compilador.

Env: configura o ambiente ocaml para que a compilação seja bem sucedida

Msg: Comando menhir para usar o arquivo de mensagens e gerar um arquivo Ocaml que relaciona uma falha com a sua respectiva mensagem

Build: Comando Ocaml que vai compilar o arquivo teste e todas as dependências necessárias.

6.2.4 Futuras correções

Geração da Arvore

A geração da árvore está muito "poluída", idealmente ela deve ser mais simples para facilitar as próximas etapas do compilador.

Mensagens de Erro

As mensagens de erro devem ser significativas para o programador, elas devem ser reformuladas.

6.2.5 Testes

Para realizar os testes com o analisador da linguagem Lua é necessário entrar com os comandos:

```
make build
rlwrap ocaml
# parser_arq "diretorio";;
```


Capítulo 7

Referências

- [1] Documentação Lua - <https://www.lua.org/docs.html>
- [2] Documentação OCaml - <https://ocaml.org/docs/>
- [3] Wikibooks, Parrot Virtual Machine - https://en.wikibooks.org/wiki/Parrot_Virtual_Machine
- [4] Wikibooks, PASM Reference - https://en.wikibooks.org/wiki/Parrot_Virtual_Machine/PASM_Reference
- [5] Wikibooks, Parrot Intermediate Representation (PIR) - https://en.wikibooks.org/wiki/Parrot_Virtual_Machine/Parrot_Intermediate_Representation
- [6] Cardinal, Github - <https://github.com/parrot/cardinal/>
- [7] Opcodes de PASM - <http://docs.parrot.org/parrot/latest/html/ops.html>
- [8] Parrot Documentation, Exemplos de PASM - <http://parrot.org/dev/examples/pasm>
- [9] Basics of Copiler Design - Torben Ægidius Mogensen