PCS2056

Linguagens e Compiladores

Construção do compilador de Lua para Bytecode

Professor: Ricardo Luis de Azevedo Rocha

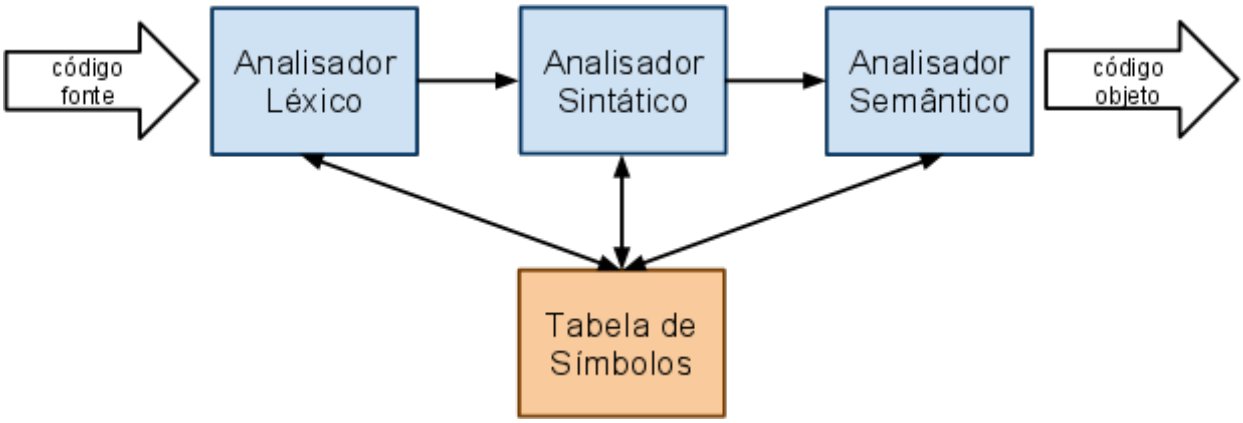
Grupo:
Filipe Morgado Simões de Campos 5694101
Rafael Barbolo Lopes 5691262

1. Introdução

Este relatório descreve o desenvolvimento parcial de um compilador de Lua para Bytecode (JVM). Este compilador foi construído com objetivo didático e, por isso, requisitos de velocidade e otimização de código-fonte foram desconsiderados. Ele é um compilador escrito em C, que compila a linguagem Lua e gera o código objeto que é representado por Bytecode, que roda na máquina virtual Java (JVM).

O compilador foi desenvolvido inteiramente em ambiente Unix, no sistema operacional Mac OS X, sendo que os recursos computacionais necessários para a sua execução são simples e pouco custosos. O compilador é dirigido por sintaxe e executado em um único passo, com analisador sintático baseado em Autômato de Pilha Estruturado (APE). O código do compilador pode ser acessado pelo link que segue:
https://github.com/barbolo/educational_compiler/tree/master/PROVA.

Sua arquitetura geral pode ser observada na imagem a seguir:



O desenvolvimento do compilador foi dividido nas seguintes fases:

1. Projeto e implementação do analisador léxico
2. Projeto e implementação do analisador sintático
3. Projeto e implementação do analisador semântico e gerador de código

Na primeira fase, que se refere ao analisador léxico, foram implementados os *tokens* e o transdutor que faz parsing do código fonte de entrada. Na segunda fase, que se refere ao analisador sintático, foi implementado um reconhecedor da linguagem Lua baseado no Autômato de Pilha Estruturado visto em aula.

Por fim, na terceira e última fase, foram projetadas e implementadas as ações semânticas e suas integrações com os analisadores léxico e sintático, permitindo que o código objeto fosse

gerado pelo compilador.

2. Especificação da linguagem fonte: Lua

2.1 Lua em BNF estendida sem precedência de operadores

```
trecho ::= {comando [';']} [ultimocomando[';']]
bloco ::= trecho
comando ::= listavar '=' listaexp |
      chamadadefuncao |
      do bloco end |
      while exp do bloco end |
      repeat bloco until exp |
      if exp then bloco {elseif exp then bloco} [else bloco] end |
      for Nome '=' exp ',' exp [',' exp] do bloco end |
      for listadenomes in listaexp do bloco end |
      function nomedafuncao corpodafuncao |
      local function Nome corpodafuncao |
      local listadenomes ['=' listaexp]
ultimocomando ::= return [listaexp] | break
nomedafuncao ::= Nome {'.' Nome} [':' Nome]
listavar ::= var {',' var}
var ::= Nome | expprefixo '[' exp ']' | expprefixo '.' Nome
listadenomes ::= Nome {',' Nome}
listaexp ::= {exp ','} exp
exp ::= nil | false | true | Numero | Cadeia | '...' | funcao |
      expprefixo | construtortabela | exp opbin exp | opunaria exp
expprefixo ::= var | chamadadefuncao | '(' exp ')'
chamadadefuncao ::= expprefixo args | expprefixo ':' Nome args
args ::= '(' [listaexp] ')' | construtortabela | Cadeia
funcao ::= function corpodafuncao
corpodafuncao ::= '(' [listapar] ')' bloco end
listapar ::= listadenomes [',' '...'] | '...'
construtortabela ::= '{' [listadecampos] '}'
listadecampos ::= campo {separadoresdecampos campo} [separadordecampos]
campo ::= '[' exp ']' '=' exp | Nome '=' exp | exp
separadordecampos ::=  ',' | ';'
opbin ::= '+' | '-' | '*' | '/' | '^' | '%' | '..' |
      '<' | '<=' | '>' | '>=' | '==' | '~=' |
      and | or
opunaria ::= '-' | 'not' | '#'
```

2.2 Lua em descrição de Wirth

```
trecho = { comando [ ";" ] } [ ultimocomando [ ";" ] ].
bloco = trecho.
comando =   listavar "=" listaexp |
```

```
                  chamadadefuncao |
                  "do" bloco "end" |
                  "while" exp "do" bloco "end" |
                  "repeat" bloco "until" exp |
                  "if" exp "then" bloco {"elseif" exp "then" bloco} ["else"
bloco] "end" |
                  "for" Nome ({ "," Nome } "in" listaexp | "=" exp "," exp
[ "," exp ]) "do" bloco "end" |
                  "function" nomedafuncao corpodafuncao |
                  "local" "function" Nome corpodafuncao |
                  "local" listadenomes [ "=" listaexp ].
ultimocomando = "return" [ listaexp ] | "break".
nomedafuncao = Nome { "." Nome } [ ":" Nome ].
listavar = var { "," var }.
var = Nome | expprefixo "[" exp "]" | expprefixo "." Nome.
listadenomes = Nome { "," Nome }.
listaexp = { exp "," } exp.
exp = "nil" | "false" | "true" | Numero | Cadeia | "..." | funcao |
expprefixo | construtortabela | exp opbin exp | opunaria exp.
expprefixo = var | chamadadefuncao | "(" exp ")".
chamadadefuncao = expprefixo args | expprefixo ":" Nome args.
args = "(" [ listaexp ] ")" | construtortabela | Cadeia.
funcao = "function" corpodafuncao.
corpodafuncao = "(" [ listapar ] ")" bloco "end".
listapar = listadenomes [ "," "..." ] | "...".
construtortabela = "{" [listadecampos] "}".
listadecampos = campo { separadoresdecampos campo } [ separadoresdecampos ].
campo = "[" exp "]" "=" exp | Nome "=" exp | exp.
separadoresdecampos = "," | ";".
opbin = "+" | "-" | "*" | "/" | "^" | "%" | ".." |
                  "<" | "<=" | ">" | ">=" | "==" | "~=" |
                  "and" | "or".
opunaria = "-" | "not" | "#".
```

2.3 Gramática reduzida (em Wirth)

```
bloco = { (((Nome | ((Nome | "(" exp ")")   { "[" exp "]" | "." Nome | ("(" [
{ exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia) | ":" Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "="
exp | Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp |
exp) } [ "," | ";" ] ] "}") | Cadeia)}) "[" exp "]" | ((Nome | "(" exp ")")
{ "[" exp "]" | "." Nome | ("(" [ { exp "," } exp ] ")" | ("{" [ ("["
exp "]" "=" exp | Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp |
Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia) | ":" Nome ("(" [ {
exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia)}) "." Nome) { "," (Nome | ((Nome | "(" exp ")")   { "[" exp "]" |
"." Nome | ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp |
```

```
Nome "=" exp | exp) { "," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) }
[ "," | ";" ] ] "}") | Cadeia) | ":" Nome ("(" [ { exp "," } exp ] ")" | ("{"
[ ("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("[" exp "]" "="
exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia)}) "[" exp "]" |
((Nome | "(" exp ")")   { "[" exp "]" | "." Nome | ("(" [ { exp "," } exp ]
")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("["
exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia) | ":"
Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "="
exp | exp) { "," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," |
";" ] ] "}") | Cadeia)}) "." Nome) }) "=" { exp "," } exp | (((Nome | "("
exp ")")   { "[" exp "]" | "." Nome | ("(" [ { exp "," } exp ] ")" | ("{" [
("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("[" exp "]" "=" exp
| Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia) | ":" Nome ("(" [ {
exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia)}) ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp |
Nome "=" exp | exp) { "," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) }
[ "," | ";" ] ] "}") | Cadeia) | ((Nome | "(" exp ")")   { "[" exp "]" | "."
Nome | ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "="
exp | exp) { "," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," |
";" ] ] "}") | Cadeia) | ":" Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("["
exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("[" exp "]" "=" exp |
Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia)}) ":" Nome ("(" [ {
exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia)) | "do" bloco "end" | "while" exp "do" bloco "end" | "repeat"
bloco "until" exp | "if" exp "then" bloco {"elseif" exp "then" bloco} ["else"
bloco] "end" | "for" Nome ({ "," Nome } "in" { exp "," } exp | "=" exp ","
exp [ "," exp ]) "do" bloco "end" | "function" Nome { "." Nome } [ ":" Nome ]
"(" [ Nome { "," Nome } [ "," "..." ] | "..." ] ")" bloco "end" |
"local" "function" Nome "(" [ Nome { "," Nome } [ "," "..." ] | "..." ] ")"
bloco "end" | "local" Nome { "," Nome } [ "=" { exp "," } exp ]) [ ";" ] } [
("return" [ { exp "," } exp ] | "break") [ ";" ] ].

exp = ("nil" | "false" | "true" | Numero | Cadeia | "..." | ("function" "(" [
Nome { "," Nome } [ "," "..." ] | "..." ] ")" bloco "end") | ((Nome | "("
exp ")")   { "[" exp "]" | "." Nome | ("(" [ { exp "," } exp ] ")" | ("{" [
("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("[" exp "]" "=" exp
| Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia) | ":" Nome ("(" [ {
exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia)}) | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";")
("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}") | {"-" |
"not" | "#"} ("nil" | "false" | "true" | Numero | Cadeia | "..." |
("function" "(" [ Nome { "," Nome } [ "," "..." ] | "..." ] ")" bloco "end") |
((Nome | "(" exp ")")   { "[" exp "]" | "." Nome | ("(" [ { exp "," } exp ]
")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) { "," | ";") ("["
exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}") | Cadeia) | ":"
Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "="
```

```
exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," |
";" ] ] "}") | Cadeia)}) | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp)
{ ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ]
"}"))) {("+" | "-" | "*" | "/" | "^" | "%" | ".." | "<" | "<=" | ">" | ">=" |
"==" | "~=" | "and" | "or") ("nil" | "false" | "true" | Numero | Cadeia |
"..." | ("function" "(" [ Nome { "," Nome } [ "," "..." ] | "..." ] ")"
bloco "end") | ((Nome | "(" exp ")")   { "[" exp "]" | "." Nome | ("(" [ {
exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia) | ":" Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "="
exp | Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp |
exp) } [ "," | ";" ] ] "}") | Cadeia)}) | ("{" [ ("[" exp "]" "=" exp |
Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) }
[ "," | ";" ] ] "}") | {"-" | "not" | "#"} ("nil" | "false" | "true" | Numero
| Cadeia | "..." | ("function" "(" [ Nome { "," Nome } [ "," "..." ] | "..." ]
")" bloco "end") | ((Nome | "(" exp ")")   { "[" exp "]" | "." Nome | ("(" [
{ exp "," } exp ] ")" | ("{" [ ("[" exp "]" "=" exp | Nome "=" exp | exp) {
("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) } [ "," | ";" ] ] "}")
| Cadeia) | ":" Nome ("(" [ { exp "," } exp ] ")" | ("{" [ ("[" exp "]" "="
exp | Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp |
exp) } [ "," | ";" ] ] "}") | Cadeia)}) | ("{" [ ("[" exp "]" "=" exp |
Nome "=" exp | exp) { ("," | ";") ("[" exp "]" "=" exp | Nome "=" exp | exp) }
[ "," | ";" ] ] "}"))) }.
```

2.4 Exemplo de programa

```
function perfeitos(n)
   cont=0
   x=0
   print('Os numeros perfeitos sao ')
   repeat
      x=x+1
      soma=0
      for i=1,(x-1) do
         if math.mod(x,i)==0 then soma=soma+i;
         end
      end
      if soma == x then
         print(x)
         cont = cont+1
      end
   until cont==n
   print('Pressione qualquer tecla para finalizar...')
end
```

2.5 Lista de palavras reservadas

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

2.6 Lista de símbolos

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

3. Projeto e implementação do analisador léxico

O analisador léxico é um módulo responsável pela leitura do(s) arquivo(s) onde o código fonte está localizado; ele realiza o *parsing* desse arquivo, extraindo o que são chamados de *tokens* (átomos).

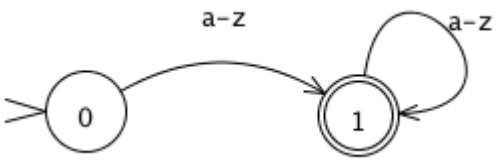
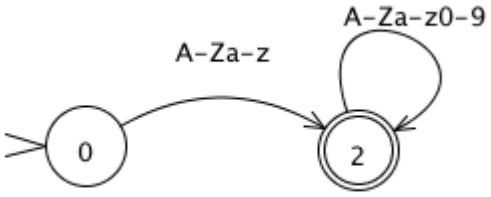
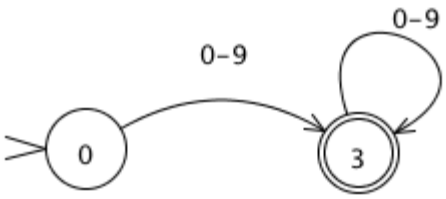
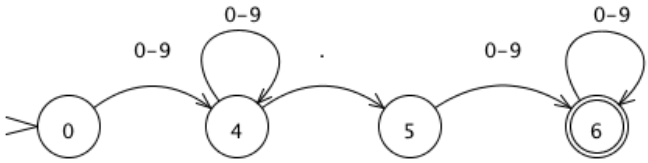
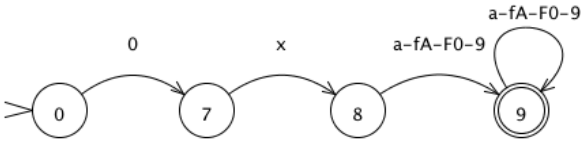
O analisador léxico lê cada um dos caracteres do código fonte e realiza agrupamento deles em *tokens*, que são classificados em tipos de informações importantes do programa que deve ser compilado/interpretado (um *token* pode ser, por exemplo, uma palavra reservada ou um identificador de variável). Durante esse processo, o analisador pode realizar conversões desses *tokens* (por exemplo, conversão para tipo numérico), e também elimina caracteres que não fazem parte do programa (espaços em branco, comentários, etc).

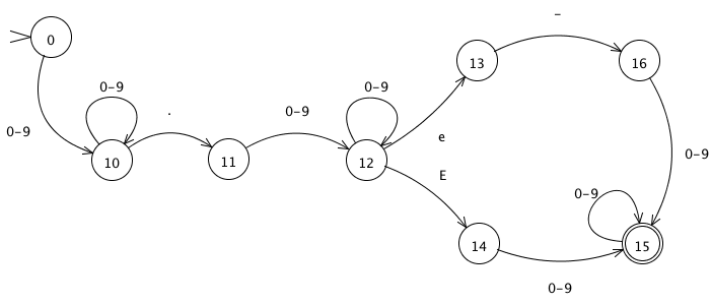
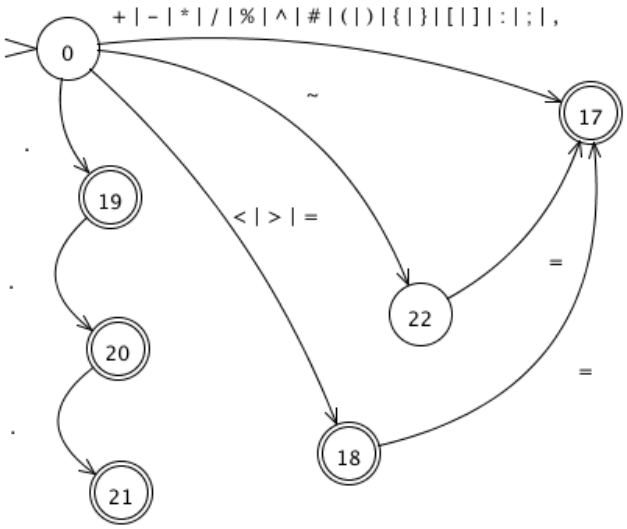
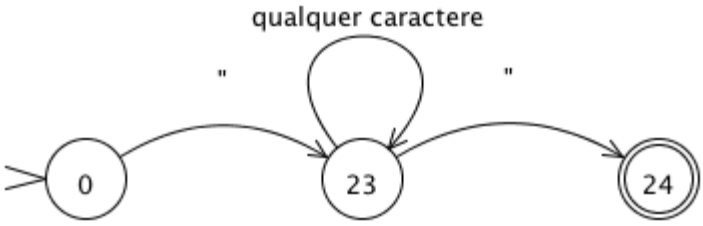
Também é papel do analisador léxico relatar erros durante sua execução, tratar macros, e criar e preencher estruturas de dado que guardam referência de localização de cada *token* no código fonte.

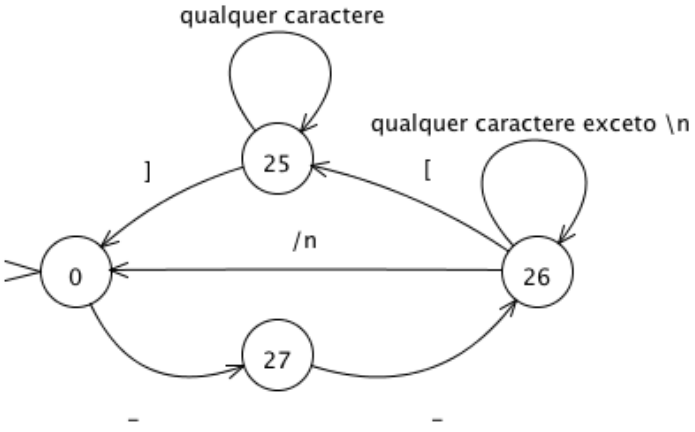
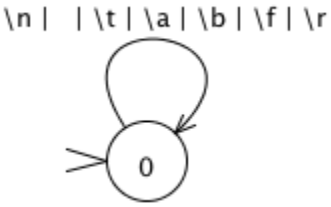
3.1 Átomos e autômatos finitos

Os átomos da linguagem e seus autômatos finitos estão definidos na tabela abaixo:

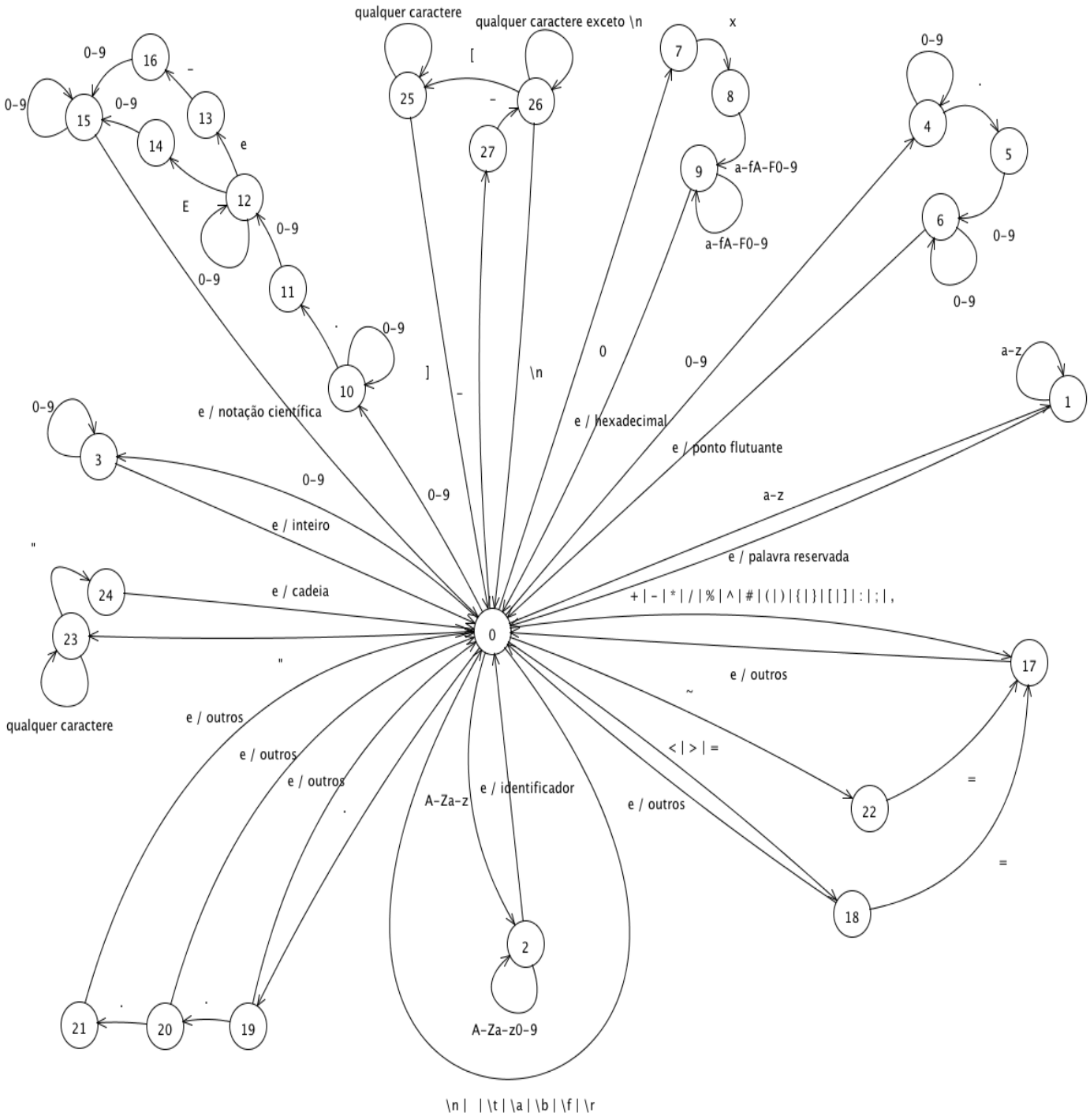
Classe do Token (Átomo)	**Expressão Regular**	**Autômato**

Palavra reservada	[a-z][a-z]*	
Identificador (Nome)	[A-Za-z][A-Za-z0-9]*	
Número inteiro	[0-9][0-9]*	
Número ponto flutuante	[0-9]+\.[0-9]+	
Número hexadecimal	0x[a-fA-F0-9]+	

Número em notação científica	[0-9]+\.[0-9]+((e-)|(E))[0-9]+	
Outros itens léxicos	[+]|[-]|[*]|[/][%]|[\^]|[#]|(==)|(~=)|(<=)|(>=)|[<]|[>]|[=]|[\(]|[\)]|[\{]|[\}]|[\[]|[\]]|[;]|[:]|[,]|[\.]|(\.\.)|(\.\.\.)	
Cadeia	".*"	

Comentário	(--[^\n]*\n) | (--[.*])	
Espaçador	(\n| |\t|\a|\b|\f|\r)+	

3.2 Transdutor



O transdutor foi abstraído como uma tabela de transições. Essa tabela possui 4 colunas: estado atual, entrada, próximo estado, saída. A implementação desta tabela foi feita através de uma matriz de uma estrutura de dados que guarda próximo estado e saída. Desta forma, é

possível acessar próximo estado e saída conhecendo apenas o estado atual e a entrada.

É importante deixar claro que a tabela de transições se apresentou como um meio para implementar o transdutor com flexibilidade. Como a linguagem de programação do código fonte ainda não foi definida, é importante que o código seja flexível para aceitar novas regras de transições do transdutor.

O transdutor é uma instância que tem um método que consome uma entrada e atualiza seu estado atual, retornando o tipo de token que está sendo lido pelo analisador léxico. Para que este método funcione, é necessário conhecer o caractere de entrada atual e o seguinte (*lookahead*), de maneira que transições com cadeia vazia sejam resolvidas e não sejam retornados resultados inválidos. Este método é chamado pelo analisador léxico enquanto está lendo um token e analisando seu tipo e valor.

4.5 Considerações sobre a implementação

4.5.1 Transdutor

O transdutor foi abstraído como uma tabela de transições. Essa tabela possui 4 colunas: estado atual, entrada, próximo estado, saída. A implementação desta tabela foi feita através de uma matriz de uma estrutura de dados que guarda próximo estado e saída. Desta forma, é possível acessar próximo estado e saída conhecendo apenas o estado atual e a entrada.

É importante deixar claro que a tabela de transições se apresentou como um meio para implementar o transdutor com flexibilidade. Como a linguagem de programação do código fonte ainda não foi definida, é importante que o código seja flexível para aceitar novas regras de transições do transdutor.

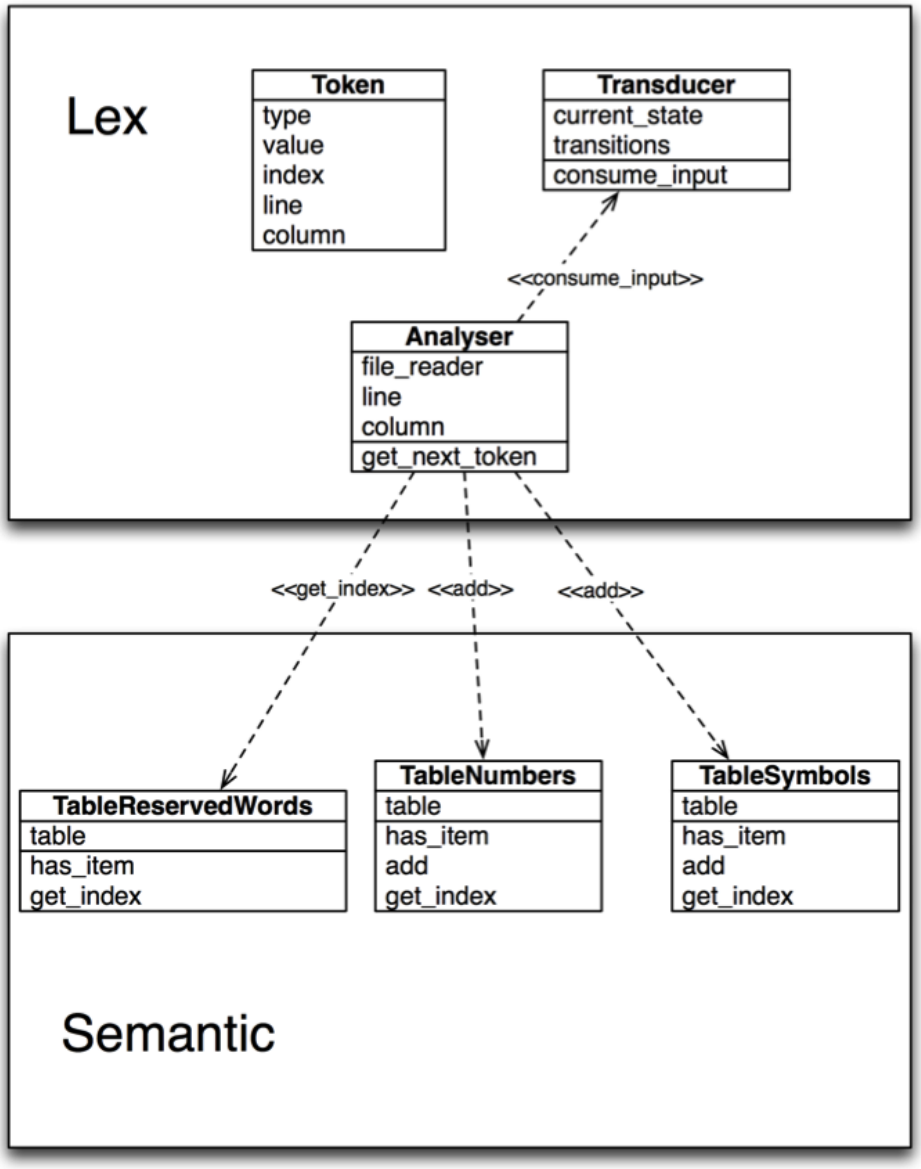
O transdutor é uma instância que tem um método que consome uma entrada e atualiza seu estado atual, retornando o tipo de token que está sendo lido pelo analisador léxico. Para que este método funcione, é necessário conhecer o caractere de entrada atual e o seguinte (*lookahead*), de maneira que transições com cadeia vazia sejam resolvidas e não sejam retornados resultados inválidos. Este método é chamado pelo analisador léxico enquanto está lendo um token e analisando seu tipo e valor.

4.5.2 Módulos em C

O analisador léxico, bem como todo o compilador que será desenvolvido, foi implementado utilizando-se a linguagem de programação C. Após uma etapa inicial de estudos teóricos, que foram apresentados nas primeiras questões e portanto não serão reintroduzidos aqui, prosseguimos com o seu desenvolvimento e testes para validar seu funcionamento. Logo, cabe apresentar a estrutura do compilador até então, descrevendo seu funcionamento e os testes realizados.

O programa foi estruturando em dois grupos para essa fase do compilador. O primeiro dele,

representa o analisador léxico e suas estruturas auxiliares que são o foco dessa atividade. Já o segundo, reserva espaço para a etapa futura, o analisador semântico, que pôde ser contemplado com algumas estruturas que também são utilizadas pelo analisador léxico. A interação entre esses módulos ficará mais clara no decorrer da descrição do funcionamento do sistema.



O primeiro grupo é composto por uma dupla de arquivos que representa um token (token.h e token.c), uma outra dupla para implementar o transdutor (transducer.h e transducer.c) e, por último, a dupla de arquivos que coordena o processo do analisador léxico (analyser.h e analyser.c).

No arquivo token.h foi definida a estrutura de dados de um token e todos os tipos possíveis a

que este poderia pertencer. Além dessa estrutura e dos tipos de tokens que são utilizados por todo o analisador léxico, foi declarada a variável global que representa o token no compilador.

Para o arquivo token.c restou a implementação da função *token_type_name* que é responsável por retornar o nome do tipo do token que está sendo processado pelo analisador léxico. Ela foi utilizada na impressão desse nome na saída do programa.

O arquivo transducer.h e o arquivo transducer.c já tiveram seu funcionamento explicado no item 4.5.1. Vale apenas acrescentar que seu funcionamento é solicitado pelo analyser através da função *transducer_consume_input* em que são passados os valores do caractere atual e do próximo (*lookahead*) e a função retorna o código do tipo do token com base na consulta da matriz *transducer_transitions* que representa o transdutor final.

O arquivo analyser.h contém apenas a declaração da função *get_next_token* que foi implementada no arquivo analyser.c. Essa função é responsável pelo funcionamento da parte do compilador responsável pela análise léxica como já foi explicado na questão 8. Durante esse procedimento, é feito o uso da função *read_next_char,* que retorna os caracteres do arquivo que contém o código fonte, e das estruturas anteriormente indicadas do grupo semântico para a consulta e adição de valores nas tabelas semânticas.

O segundo grupo, contém as tabelas semânticas que estão implementadas nos arquivos tables.h, tables.c, symboltable.h e symboltable.c. Elas são: tabela de símbolos, tabela de números, tabela de palavra reservadas e a tabela de caracteres especiais.

Uma biblioteca foi criada para representar uma string, strings.h, Apêndice XI, e strings.c, Apêndice XII. A utilizamos apenas para comparar se uma string é igual a outra através da função *strcmp*, e também para retornar uma cópia de uma string com a função *strcpy*.

Por fim, a última biblioteca realiza a leitura dos caracteres dos arquivos, reader.h e reader.c. Vale comentar que essa possui uma estrutura representando uma cabeça de leitura que possui os caracteres atual, anterior e futuro (lookahead), a posição (linha e coluna) do caractere atual e o ponteiro para o arquivo sendo lido.

Para consolidar o desenvolvimento dessa etapa do compilador, testamos a saída do léxico para o seguinte código de entrada:

```
b
b
b == a = 10120 + 15;
> >=
< <=

3
```

```
bc = 10

d == ; 29123 7

10

10

15

bc

bc

=

for

else
```

A saída encontrada, que também era a esperada, foi:

Padrão da saída para cada token:

valor (indicie na tabela semântica) :: nome do tipo (identificador do tipo) :: linha , coluna

```
Token:    b      (0)  ::  identifier     (2)  ::  line  1, column  1
Token:    b      (0)  ::  identifier     (2)  ::  line  2, column  1
Token:    b      (0)  ::  identifier     (2)  ::  line  3, column  1
Token:    ==     (0)  ::  special char   (4)  ::  line  3, column  3
Token:    a      (1)  ::  identifier     (2)  ::  line  3, column  6
Token:    =      (1)  ::  special char   (4)  ::  line  3, column  8
Token:    10120  (0)  ::  integer number (3)  ::  line  3, column  10
Token:    +      (2)  ::  special char   (4)  ::  line  3, column  16
Token:    15     (1)  ::  integer number (3)  ::  line  3, column  18
Token:    ;      (3)  ::  special char   (4)  ::  line  3, column  20
Token:    >      (4)  ::  special char   (4)  ::  line  4, column  1
Token:    >=     (5)  ::  special char   (4)  ::  line  4, column  3
Token:    <      (6)  ::  special char   (4)  ::  line  5, column  1
Token:    <=     (7)  ::  special char   (4)  ::  line  5, column  3
Token:    3      (2)  ::  integer number (3)  ::  line  7, column  1
```

Token:	**bc**	**(2)**	**::**	**identifier**	**(2)**	**::**	**line**	**11, column**	**1**
Token:	**=**	**(1)**	**::**	**special char**	**(4)**	**::**	**line**	**11, column**	**4**
Token:	**10**	**(3)**	**::**	**integer number**	**(3)**	**::**	**line**	**11, column**	**6**
Token:	**d**	**(3)**	**::**	**identifier**	**(2)**	**::**	**line**	**13, column**	**1**
Token:	**==**	**(0)**	**::**	**special char**	**(4)**	**::**	**line**	**13, column**	**3**
Token:	**;**	**(3)**	**::**	**special char**	**(4)**	**::**	**line**	**13, column**	**6**
Token:	**29123**	**(4)**	**::**	**integer number**	**(3)**	**::**	**line**	**13, column**	**8**
Token:	**7**	**(5)**	**::**	**integer number**	**(3)**	**::**	**line**	**13, column**	**14**
Token:	**10**	**(3)**	**::**	**integer number**	**(3)**	**::**	**line**	**15, column**	**1**
Token:	**10**	**(3)**	**::**	**integer number**	**(3)**	**::**	**line**	**17, column**	**1**
Token:	**15**	**(1)**	**::**	**integer number**	**(3)**	**::**	**line**	**19, column**	**1**
Token:	**bc**	**(2)**	**::**	**identifier**	**(2)**	**::**	**line**	**21, column**	**1**
Token:	**bc**	**(2)**	**::**	**identifier**	**(2)**	**::**	**line**	**23, column**	**1**
Token:	**=**	**(1)**	**::**	**special char**	**(4)**	**::**	**line**	**25, column**	**1**
Token:	**for**	**(1)**	**::**	**reserved word**	**(1)**	**::**	**line**	**27, column**	**1**
Token:	**else**	**(2)**	**::**	**reserved word**	**(1)**	**::**	**line**	**29, column**	**1**

4. Projeto e implementação do analisador sintático

Como segundo e principal bloco do compilador dirigido por sintaxe desse projeto, temos o analisador sintático. Podemos dizer que ele é o principal bloco, pois é responsável por todo o gerenciamento e chamadas das rotinas existentes no compilador, tanto do léxico quanto do semântico. Sua função principal está relacionada à análise e verificação da sequência dos átomos provindo do analisador léxico quando este lê o código fonte. Assim é de sua responsabilidade verificar se esses átomos estão de acordo com a sintaxe da linguagem a ser compilada.

Em teoria, o produto dessa etapa de compilação é a árvore sintática. Porém, para a implementação, o que de fato ocorre é que não existe a necessidade de se sintetizar toda essa árvore para então passarmos para a etapa do analisador semântico, tudo isso pode ser feito de uma vez só. Ou seja, geram-se “pedaços” da árvore necessários para que as ações semânticas corretas possam ser chamadas e a geração de código efetuada com sucesso.

A análise sintática engloba, em geral, as seguinte funções :

- Identificação de sentenças;
- Detecção de erros de sintaxe;
- Recuperação de erros;
- Correção de erros;
- Montagem da árvore abstrata da sentença;
- Comando da ativação do analisador léxico;
- Comando do modo de operação do analisador léxico;
- Ativação de rotinas da análise referente às dependências de contexto da

linguagem;
- Ativação de rotinas de análise semântica;
- Ativação de rotinas de síntese do código objeto.

O analisador sintático foi construído tomando como base a já apresentada e discutida gramática reduzida da linguagem.
O passo seguinte constituiu em utilizar o gerador de autômados criado por Hugo Baraúna e Fábio Yamate, localizado em: http://radiant-fire-72.heroku.com/. Para tal, deve-se colocar a gramática reduzida em notação de Wirth na entrada do gerador, e esse deve produzir como saída os autômatos já otimizados que representam cada submáquina do APE que reconhece a linguagem descrita pela gramática dada. Tais autômatos nos são transmitidos na forma de XML com as transições correspondentes às máquinas que o APE deve utilizar.
Abaixo, seguem as transições e os autômatos gerados para cada submáquina.

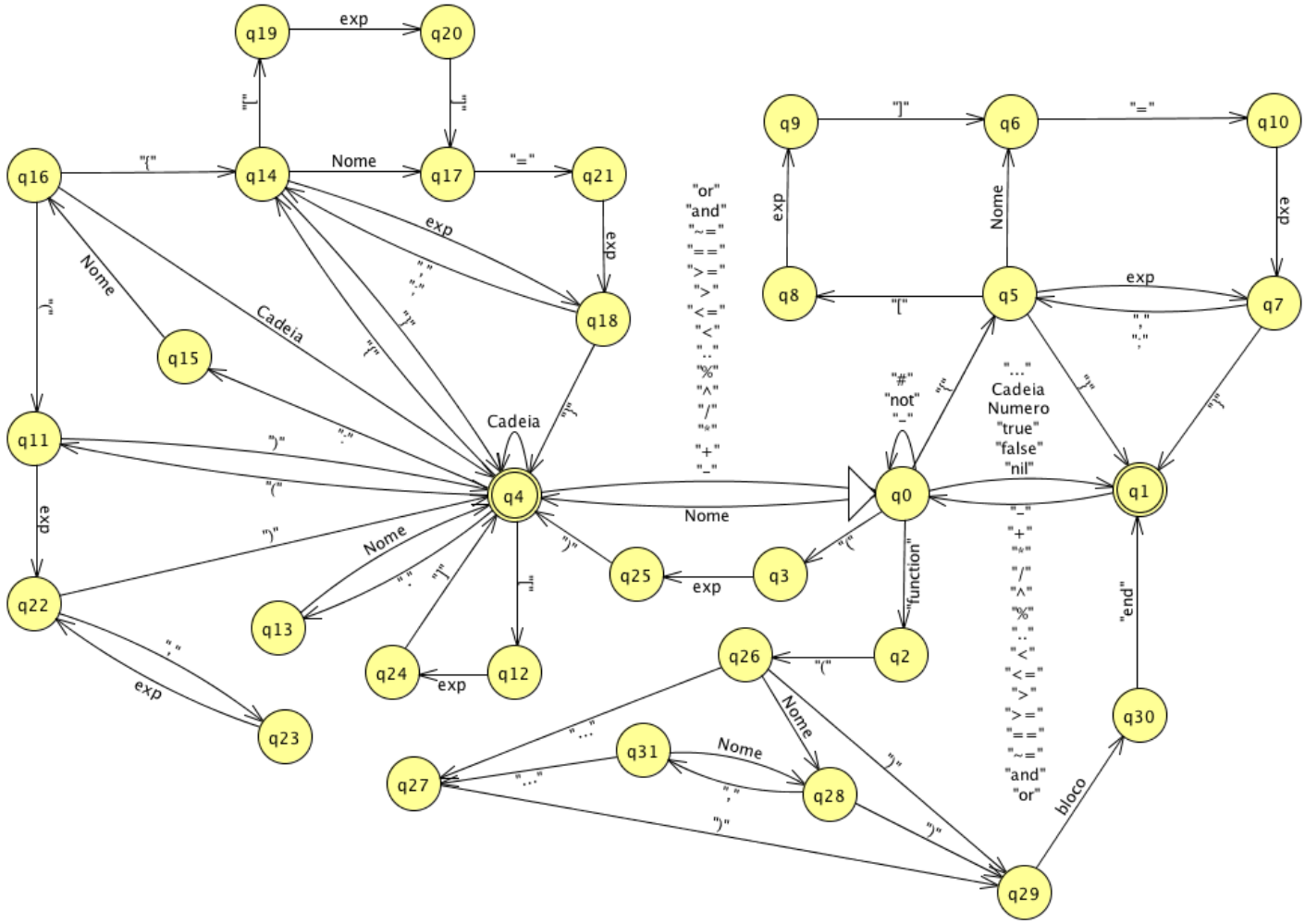
Máquina **Exp**:

Transições:

initial: 0 final: 1, 4 (0, "nil") -> 1 (0, "false") -> 1 (0, "true") -> 1 (0, Numero) -> 1 (0, Cadeia) -> 1 (0, "...") -> 1 (0, "function") -> 2 (0, "(") -> 3 (0, Nome) -> 4 (0, "{") -> 5 (0, "-") -> 0 (0, "not") -> 0 (0, "#") -> 0 (1, "-") -> 0 (1, "+") -> 0 (1, "*") -> 0 (1, "/") -> 0 (1, "^") -> 0 (1, "%") -> 0 (1, "..") -> 0 (1, "<") -> 0 (1, "<=") -> 0 (1, ">") -> 0 (1, ">=") -> 0 (1, "==") -> 0 (1, "~=") -> 0 (1, "and") -> 0 (1, "or") -> 0	(4, "==") -> 0 (4, "~=") -> 0 (4, "and") -> 0 (4, "or") -> 0 (5, Nome) -> 6 (5, exp) -> 7 (5, "[") -> 8 (5, "}") -> 1 (6, "=") -> 10 (7, ",") -> 5 (7, ";") -> 5 (7, "}") -> 1 (8, exp) -> 9 (9, "]") -> 6 (10, exp) -> 7 (11, ")") -> 4 (11, exp) -> 22 (12, exp) -> 24 (13, Nome) -> 4 (14, Nome) -> 17 (14, exp) -> 18 (14, "[") -> 19 (14, "}") -> 4 (15, Nome) -> 16 (16, Cadeia) -> 4 (16, "(") -> 11 (16, "{") -> 14 (17, "=") -> 21 (18, ",") -> 14 (18, ";") -> 14

(2, "(") -> 26 (3, exp) -> 25 (4, Cadeia) -> 4 (4, "(") -> 11 (4, "[") -> 12 (4, ".") -> 13 (4, "{") -> 14 (4, ":") -> 15 (4, "-") -> 0 (4, "+") -> 0 (4, "*") -> 0 (4, "/") -> 0 (4, "^") -> 0 (4, "%") -> 0 (4, "..") -> 0 (4, "<") -> 0 (4, "<=") -> 0 (4, ">") -> 0 (4, ">=") -> 0	(18, "}") -> 4 (19, exp) -> 20 (20, "]") -> 17 (21, exp) -> 18 (22, ",") -> 23 (22, ")") -> 4 (23, exp) -> 22 (24, "]") -> 4 (25, ")") -> 4 (26, "...") -> 27 (26, Nome) -> 28 (26, ")") -> 29 (27, ")") -> 29 (28, ",") -> 31 (28, ")") -> 29 (29, bloco) -> 30 (30, "end") -> 1 (31, "...") -> 27 (31, Nome) -> 28

Autômato:



Máquina **Bloco**

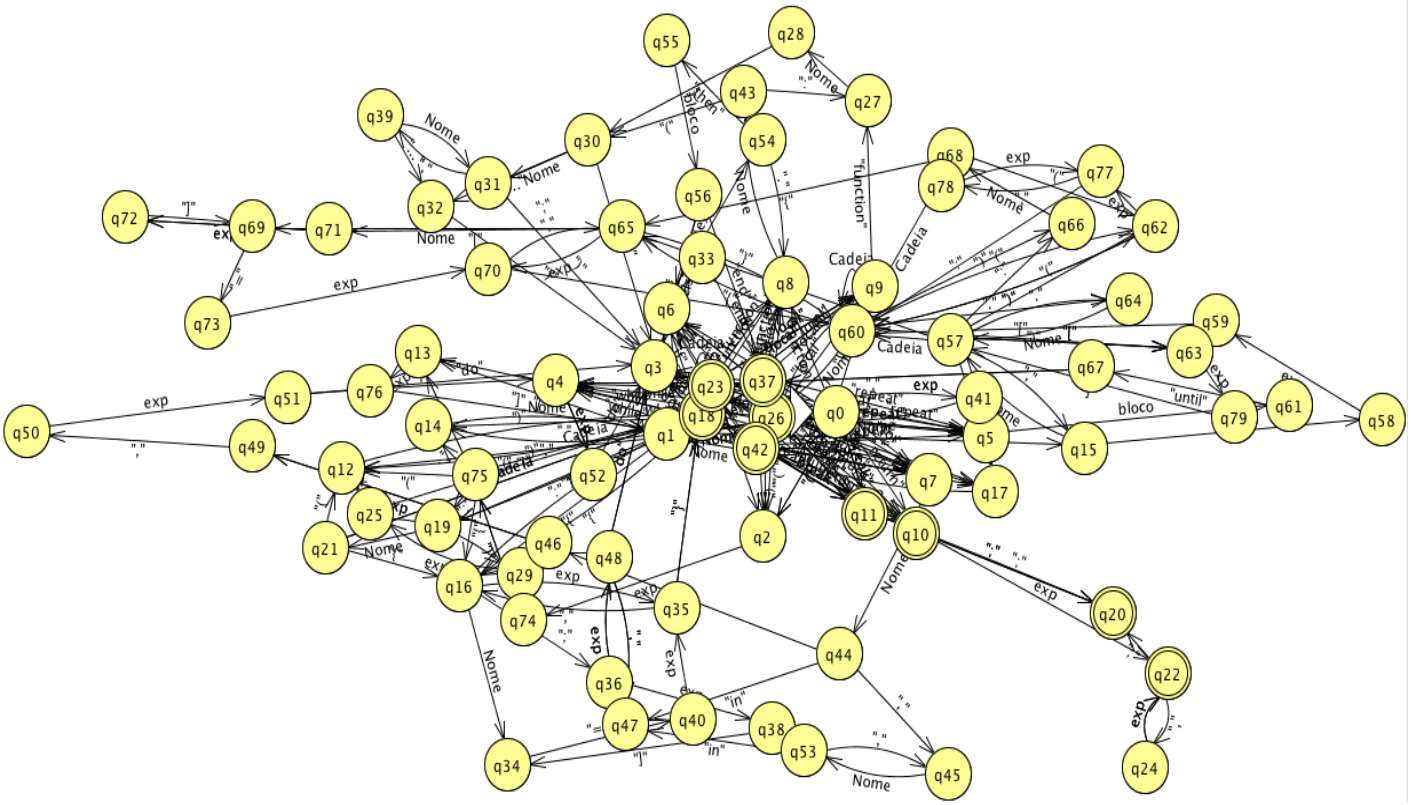
Transições:

initial: 0 final: 10, 11, 18, 20, 22, 23, 26, 37, 42 (0, Nome) -> 1 (0, "(") -> 2 (0, "do") -> 3 (0, "while") -> 4 (0, "repeat") -> 5 (0, "if") -> 6 (0, "for") -> 7 (0, "function") -> 8 (0, "local") -> 9 (0, "return") -> 10 (0, "break") -> 11 (1, "(") -> 12 (1, "[") -> 13	(35, ",") -> 16 (35, ";") -> 16 (35, "}") -> 18 (36, exp) -> 38 (37, Nome) -> 1 (37, "(") -> 2 (37, ";") -> 23 (37, "do") -> 3 (37, "while") -> 4 (37, "repeat") -> 5 (37, "if") -> 6 (37, "for") -> 7 (37, "function") -> 8 (37, "local") -> 9 (37, "return") -> 10

(1, ".") -> 14
(1, ",") -> 15
(1, "{") -> 16
(1, "=") -> 17
(1, Cadeia) -> 18
(1, ":") -> 19
(2, exp) -> 74
(3, bloco) -> 33
(4, exp) -> 52
(5, bloco) -> 61
(6, exp) -> 54
(7, Nome) -> 44
(8, Nome) -> 43
(9, Nome) -> 26
(9, "function") -> 27
(10, exp) -> 22
(10, ";") -> 20
(11, ";") -> 20
(12, exp) -> 25
(12, ")") -> 18
(13, exp) -> 76
(14, Nome) -> 1
(15, Nome) -> 57
(15, "(") -> 58
(16, Nome) -> 34
(16, exp) -> 35
(16, "[") -> 36
(16, "}") -> 18
(17, exp) -> 42
(18, Nome) -> 1
(18, "(") -> 12
(18, "[") -> 13
(18, ".") -> 14
(18, "{") -> 16
(18, ";") -> 23
(18, Cadeia) -> 18
(18, ":") -> 19
(18, "do") -> 3
(18, "while") -> 4
(18, "repeat") -> 5
(18, "if") -> 6
(18, "for") -> 7
(18, "function") -> 8
(18, "local") -> 9
(18, "return") -> 10
(18, "break") -> 11
(19, Nome) -> 21
(21, "(") -> 12
(21, "{") -> 16
(21, Cadeia) -> 18
(37, "break") -> 11
(38, "]") -> 34
(39, Nome) -> 31
(39, "...") -> 32
(40, exp) -> 35
(41, Nome) -> 26
(42, Nome) -> 1
(42, "(") -> 2
(42, ",") -> 17
(42, ";") -> 23
(42, "do") -> 3
(42, "while") -> 4
(42, "repeat") -> 5
(42, "if") -> 6
(42, "for") -> 7
(42, "function") -> 8
(42, "local") -> 9
(42, "return") -> 10
(42, "break") -> 11
(43, "(") -> 30
(43, ".") -> 8
(43, ":") -> 27
(44, ",") -> 45
(44, "=") -> 46
(44, "in") -> 47
(45, Nome) -> 53
(46, exp) -> 49
(47, exp) -> 48
(48, ",") -> 47
(48, "do") -> 3
(49, ",") -> 50
(50, exp) -> 51
(51, ",") -> 4
(51, "do") -> 3
(52, "do") -> 3
(53, ",") -> 45
(53, "in") -> 47
(54, "then") -> 55
(55, bloco) -> 56
(56, "end") -> 37
(56, "elseif") -> 6
(56, "else") -> 3
(57, "(") -> 62
(57, "[") -> 63
(57, ".") -> 64
(57, ",") -> 15
(57, "{") -> 65
(57, "=") -> 17
(57, Cadeia) -> 60
(57, ":") -> 66

(22, ",") -> 24 (22, ";") -> 20 (23, Nome) -> 1 (23, "(") -> 2 (23, "do") -> 3 (23, "while") -> 4 (23, "repeat") -> 5 (23, "if") -> 6 (23, "for") -> 7 (23, "function") -> 8 (23, "local") -> 9 (23, "return") -> 10 (23, "break") -> 11 (24, exp) -> 22 (25, ")") -> 18 (25, ",") -> 29 (26, Nome) -> 1 (26, "(") -> 2 (26, ",") -> 41 (26, "=") -> 17 (26, ";") -> 23 (26, "do") -> 3 (26, "while") -> 4 (26, "repeat") -> 5 (26, "if") -> 6 (26, "for") -> 7 (26, "function") -> 8 (26, "local") -> 9 (26, "return") -> 10 (26, "break") -> 11 (27, Nome) -> 28 (28, "(") -> 30 (29, exp) -> 25 (30, Nome) -> 31 (30, ")") -> 3 (30, "...") -> 32 (31, ")") -> 3 (31, ",") -> 39 (32, ")") -> 3 (33, "end") -> 37 (34, "=") -> 40	(58, exp) -> 59 (59, ")") -> 60 (60, "(") -> 62 (60, "[") -> 63 (60, ".") -> 64 (60, "{") -> 65 (60, Cadeia) -> 60 (60, ":") -> 66 (61, "until") -> 67 (62, exp) -> 77 (62, ")") -> 60 (63, exp) -> 79 (64, Nome) -> 57 (65, Nome) -> 69 (65, exp) -> 70 (65, "[") -> 71 (65, "}") -> 60 (66, Nome) -> 68 (67, exp) -> 37 (68, "(") -> 62 (68, "{") -> 65 (68, Cadeia) -> 60 (69, "=") -> 73 (70, ",") -> 65 (70, ";") -> 65 (70, "}") -> 60 (71, exp) -> 72 (72, "]") -> 69 (73, exp) -> 70 (74, ")") -> 75 (75, "(") -> 12 (75, "[") -> 13 (75, ".") -> 14 (75, "{") -> 16 (75, Cadeia) -> 18 (75, ":") -> 19 (76, "]") -> 1 (77, ")") -> 60 (77, ",") -> 78 (78, exp) -> 77 (79, "]") -> 57

Autômato:



Para a implementação do analisador sintático, foram desenvolvidos três conjuntos (arquivo .c e arquivo .h) de arquivos, sendo eles:

- syntatic: nesses arquivos residem a estrutura lógica central do compilador. É aqui que foi implementada a rotina responsável pelo gerenciamento e pela chamada das outras rotinas dos grandes blocos do compilador. Seu funcionamento será explicado abaixo;
- machine: nesses arquivos encontra-se a estrutura de dados que representa cada submáquina do APE de nosso compilador e também uma rotina para a inicialização das transições dessas máquinas;
- ape: aqui, foi desenvolvida a estrutura de dados que representa o autômato de pilha estruturado bem como as funções necessárias para seu funcionamento;
- ape_stack: esses arquivos existem para implementar a pilha utilizada no APE. Logo, existe a estrutura de dados responsável por tal representação e as rotinas de *push* e *pop*.

A rotina inicial do nosso compilador e responsável por uma série de chamadas preparatórias até que se chegue na rotina do analisador sintático. Elas são, *init_reader* que inicia o leitor do arquivo com o código fonte; *initialize_writer* que inicializa a funcionalidade de escrita que será utilizada para escrever o código de máquina após a compilação; *initialize_transitions* que inicia as transições do transdutor utilizado no analisar léxico; *initialize_semantic* que inicia o analisador semântico; *init_semantic_actions* que é responsável por inicializar as ações semânticas; *init_ape_machines* que inicia os APE utilizados pelo sintático e por fim a função

run_sintatic é chamada pra dar início à rotina mestre do sintático.

Após ser chamada, a rotina *run_sintatic* começa requisitando o primeiro átomo do analisador léxico. Após esse passo, entra-se em um loop que executa as seguintes ações:

- *ape_consume_token*, essa rotina é responsável por fazer funcionar o APE responsável pela análise sintática do código fonte, bem como as chamadas das ações semânticas necessárias em cada transição dos autômatos das submáquinas do APE. Essa função também alertará o compilador da ocorrência de erro sintático, caso ela ocorra, a execução do compilador é interrompida;
- checa-se a ocorrência de erro semântico, caso ela seja verdadeira a execução do compilador é interrompida;
- requisita-se o próximo átomo do analisador léxico.

Por fim, é realizada uma etapa de checagem para verificar se o APE está em seu estado final e se a pilha do APE está vazia. Se sim, chama-se a função do módulo semântico *write_variables* que realiza a escrita do código referente à área de dados como as variáveis declaradas durante o código, as constante utilizadas e as variáveis temporárias. Essas informações são recuperadas consultando as tabelas criadas durante a compilação do código fonte.

5. Projeto e implementação do analisador semântico e gerador de código

Para implementar o analisador semântico, foram necessárias estruturas de dados especiais. Tanto elas quanto as ações semânticas serão especificadas nos itens a seguir.

5.1 Estruturas de dados

Tabela de símbolos

A tabela de símbolos possui os campos:
- nome do identificador;
- tipo do identificador (rótulo ou variável);
- valor (pode ser inteiro, char ou booleano).

Além disso, cada tabela de símbolos usada no compilador pode guardar referência para uma tabela de símbolos pai ou filha, de maneira que escopos de variáveis e rótulos possam ser criados no analisador semântico.

Esta estrutura é responsável por gerenciar os símbolos declarados no código fonte, sejam eles nomes de funções (rótulos) ou de variáveis. Por este motivo, existem métodos que permitem acessar tais tabelas, inserindo ou atualizando dados. Também existe um método chamado "create_new_scope" que cria uma nova tabela de símbolos, ligando-a à atual por meio do relacionamento de pai e filho.

Tabela de palavras reservadas

Esta estrutura é uma tabela que guarda a lista de palavras reservadas da linguagem. Ao inicializar o compilador, esta tabela é inicializada recebendo os valores de todas as palavras reservadas da linguagem. Após a inicialização, seu conteúdo é constante e ela é utilizada apenas para leitura. Conforme os tokens são consumidos, esta tabela é utilizada para verificar se um token lido representa um identificador ou uma palavra reservada.

Tabela de constantes

No código fonte podem ser declaradas constantes do tipo inteiro, booleano ou char. Quando o analisador léxico lê um token de um destes tipos (por exemplo, 10 para inteiro), este token é adicionado na tabela de constantes que cria um id para cada constante reconhecida no código fonte.

Pilhas de operandos e operadores

As pilhas de operadores e operandos são pilhas comuns que também possuem um método que retorna o elemento do topo sem removê-lo. Também são oferecidos métodos para verificar o tamanho da pilha e se ela está vazia. Estas duas pilhas são utilizadas durante a compilação de expressões.

Variáveis complementares

Foram utilizadas variáveis complementares que permitiram a declaração de variáveis temporárias (usadas, por exemplo, durante a compilação de uma expressão) e a declaração de rótulos da linguagem em comandos de laços ou condicionais. Algumas dessas variáveis também permitiram que certas ações semânticas fossem executadas, por exemplo, guardando resultados de operações ou endereços de retornos. Como exemplos destas variáveis, citam-se CONTATEMP e CONTAIF, que são contadores de variáveis temporárias e rótulos gerados durante a compilação.

Registro de Ativação (Stack Frame)

O Registro de Ativação consiste de uma pilha que guarda o ambiente de execução do compilador. Ela é utilizada, por exemplo, quando uma função é chamada. Neste momento, o ambiente de execução (variáveis, pilhas e tabelas) é empilhado na Stack Frame e um novo ambiente de execução é criado, evitando que conflitos entre endereços de variáveis, tipos ou declarações ocorram.

5.2 Ações semânticas

Devido ao tempo escasso para desenvolvimento deste compilador, não foram implementadas suas ações semânticas.