ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Avenida Professor Luciano Gualberto, travessa 3 nº 158 CEP 05508-900 São Paulo SP
Telefone: (011) 818-5583 Fax (011) 818-5294

Departamento de Engenharia de Computação e Sistemas Digitais

PCS2056 - Linguagens e Compiladores
Compilador FM

Felipe Giunte Yoshida Nº USP 4978231
Mariana Ramos Franco Nº USP 5179364

30 de Novembro de 2009

Conteúdo

1 Introdução

Este projeto foi desenvolvido para a disciplina PCS2056 - Linguagens e Compiladores. Seu objetivo é a construção de um compilador que transforma uma linguagem imperativa chamada FM, que será definida a seguir, em código MVN. O programa foi desenvolvido em Java e consiste em quatro etapas cujas dependências podem ser observadas na figura 1.

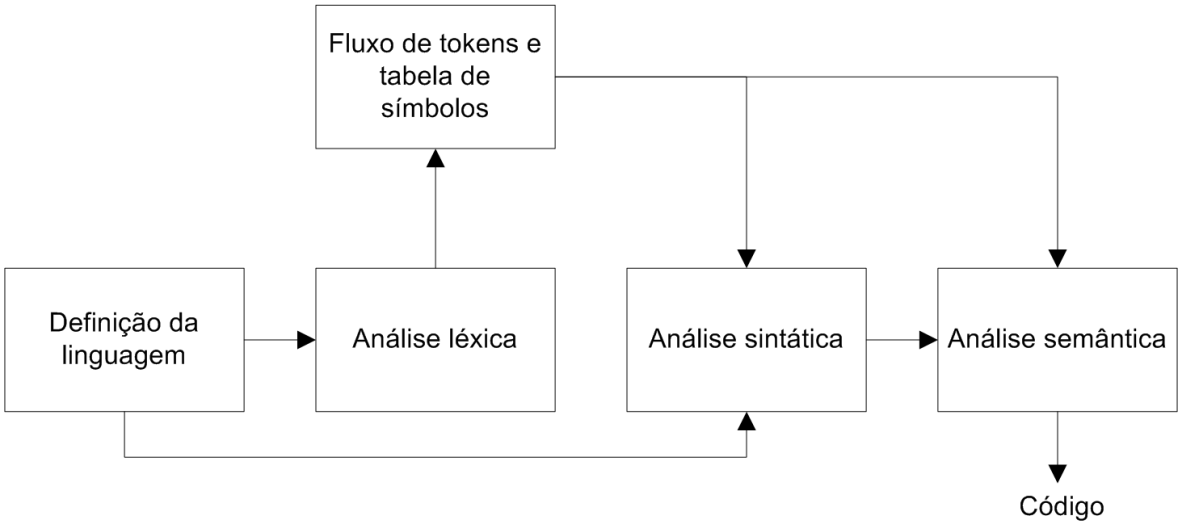

Figura 1: Etapas do projeto do compilador

Definição da linguagem Baseada na linguagem C, a linguagem FM é basicamente a tradução da linguagem C do inglês ao português com algumas simplificações.

Analisador léxico A partir da linguagem, o analisador léxico permite a divisão do código de entrada do compilador em diversos tokens, formando um Fluxo de tokens. Além disso, ele inicia uma tabela com dados sobre as variáveis do programa, chamada de tabela de símbolos.

Analisador sintático A etapa do analisador sintático pode ser dividida em duas partes. Na primeira, temos a montagem no autômato de pilha estruturado a partir da definição da linguagem. Na segunda, este autômato recebe o fluxo de tokens e a tabela de símbolos, checando se o código é reconhecido pela linguagem.

Analisador semântico O analisador semântico age juntamente do reconhecimento do código feito pelo analisador sintático. Com o reconhecimento dos tokens, instruções MVN são geradas no arquivo de saída.

2 Definição da Linguagem

A linguagem adotada no projeto corresponde à uma simplificação da linguagem de programação C, onde os comandos em inglês (IF, ELSE, WHILE, ...) foram traduzidos

para o português (SE, SENAO, ENQUANTO, ...).

Além disso, os seguintes recursos foram retirados da linguagem:

- comandos: for, do while, switch/case
- ponteiros
- tipos: short/long, signed/unsigned, float/double
- alguns operadores unários
- alguns operadores binários
- union, enum
- include, typedef
- etc...

Desta maneira, a linguagem adotada (que chamaremos de linguagem FM) aceita programas formados por uma seqüência de comandos. Os comando aceitos pela linguagem são os seguintes:

- Registro
- Declaração de variáveis
- Atribuição de valor à uma variável
- Comando iterativo: enquanto
- Comando condicional: se/senao
- Chamada de função
- Comando de entrada de dados
- Comando de saída de dados

Abaixo encontra-se a descrição formal da linguagem através da notação BNF e da notação de Wirth.

2.1 Definição Formal da Linguagem – Notação BNF

```
<programa> ::= <cabeçalho> <principal>

<cabeçalho> ::= <cabeçalho> <estrut> | <estrut> | E

<estrut>::= estrut { <seq_declarações> }

<seq_declarações> ::= <seq_declarações> <declaração> | <declaração>
```

```
<principal> ::= <seq_funções> principal ( <declara_parâmetros> )
                { <seq_comandos> }

<seq_funções> ::= <seq_funções> <função> | <função> | E

<função> ::= <tipo> <identificador> ( <declara_parâmetros> ) {
             <seq_comandos> retorno <retorno> ;}

<tipo> ::= inteiro | caracteres | booleano

<retorno> :: = <variável> | <expressão> | <numero>
               | <string> | <booleano>

<declara_parâmetros> ::= <declara_parâmetros> , <parâmetro>
                         | <parâmetro> | E

<parâmetro> ::= <tipo> <identificador> <vetor>

<identificador> ::= <identificador> <letra>
                    | <identificador> <numero> | <letra>

<vetor> ::= [ <número> ] | E

<seq_comandos> ::= <seq_comandos> <comando> | <comando> | E

<comando> ::= <declaração> | <atribuição> | <iterativo> | <condicional>
              | <chamada_função> | <entrada> | <saída> | E

<declaração> ::= <parâmetro> ;

<atribuição> ::= <variável> = <expressão> ;
                 | <variável> = <string> ;
                 | <variável> = <booleano> ;

<variável> ::= <identificador> <vetor_exp>

<vetor_exp> ::= <vetor> | [ <expressão> ]

<expressão> ::= <expressão> + <termo> | <expressão> - <termo> | <termo>

<termo> ::= <termo> * <fator> | <termo> / <fator> | <fator>

<fator> ::= <variável> | <valor_função> | <numero> | ( <expressão> )

<valor_função> ::= <identificador> ( <lista_parâmetros> )

<lista_parâmetros> ::= <expressão> | <string> | <booleano>
                       | <lista_parâmetros> , <expressão> | E

<string> := <string> <letra>
          | <string> <digito>
          | <letra> | <digito>

<booleano> := true | false
```

```
<iterativo> ::= enquanto ( <condição> ) { <seq_comandos> }

<condição> ::= <expressão> <comparador> <expressão>

<comparador> ::= > | < | >= | <= | == | !=

<condicional> ::= se ( <condição> ) { <seq_comandos> }
                | se ( <condição> ) { <seq_comandos> }
                  senao { <seq_comandos> }

<chamada_função> ::= <valor_função> ;

<entrada> ::= entrada = <variável> ;

<saida> ::= saida = <expressão> ; | saida = <string> ;

<numero> ::= <numero> <digito> | <digito>

<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
    p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F |
    G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
    X | Y | Z

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.2 Definição Formal da Linguagem – Notação de Wirth

```
programa = cabeçalho principal .

cabeçalho = { estrut } .

estrut = "estrut" "{" declaração { declaração } "}" .

principal = seq_funções "principal" "(" declara_parâmetros ")"
            "{" seq_comandos "}" .

seq_funções = { função } .

função = tipo identificador "(" declara_parâmetros ")"
         "{" seq_comandos "retorno" retorno ";" "}" .

retorno = variável | expressão | numero | string  | booleano

tipo = "inteiro" | "caracteres" | "booleano" .

declara_parâmetros = [ parâmetro {"," parâmetro} ] .

parâmetro = tipo identificador vetor .

identificador = letra { letra | digito } .

vetor = [ "[" numero "]" ] .
```

```
seq_comandos = { comando } .

comando = [ declaração | atribuição | iterativo
            | condicional | chamada_função | entrada | saida ] .

declaração = parâmetro ";" .

atribuição = variável "=" ( expressão | string | booleano )";" .

variável = identificador vetor_exp .

vetor_exp = vetor | "[" expressão "]" .

expressão = termo { ( "+" | "-" ) termo } .

termo = fator { ( "*" | "/" ) fator} .

fator = variável | valor_função | numero | "(" expressão ")" .

valor_função = identificador "(" lista_parâmetros ")" .

lista_parâmetros =  [ expressão {"," expressão } ] .

string  = ( letra | digito ) { letra | digito } .

booleano = "true" | "false" .

iterativo = "enquanto" "(" condição ")" "{" seq_comandos "}" .

condição = expressão comparador expressão .

comparador = ">" | "<" | ">=" | "<=" | "==" | "!=" .

condicional = "se" "(" condição ")" "{" seq_comandos "}"
              [ "senão" "{" seq_comandos "}" ] .

chamada_função = valor_função ";"

entrada = "entrada" "=" variável ";" .

saida = "saida" "=" ( expressão | string ) ";" .

numero = digito { digito } .

letra = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
    | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
    "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "
   H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S"
    | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" .

digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

2.3 Simplificação da Gramática

Para permitir a implementação do autômato de pilha estruturado, método que será utilizado para a construção do analisador sintático, a descrição em notação de Wirth mostrada anteriormente foi reduzida, agrupando os não-terminais essenciais à linguagem.

```
programa = { "estrut"
           "{" tipo identificador [ "[" numero "]" ] ";"
           { tipo identificador [ "[" numero "]" ] ";" } "}" }
           { tipo identificador
           "(" [ tipo identificador [ "[" numero "]" ]
           { "," tipo identificador [ "[" numero "]" ] } ] ")"
           "{" {comando}
           "retorno" ( identificador [ "[" ( numero | expressao ) "]" ]
           | expressao | numero | string
           | booleano ) ";" "}" }
           "principal" "(" [ tipo identificador [ "[" numero "]" ]
           { "," tipo identificador [ "[" numero "]" ] } ] ")"
           "{" {comando} "}" .


comando = [
          tipo identificador [ "[" numero "]" ] ";"
          |
          identificador [ "[" ( numero | expressao ) "]" ]  "="
          ( expressao | string | booleano ) ";"
          |
          "enquanto"
          "(" expressao (">" | "<" | ">=" | "<=" | "==" | "!=")
          expressao ")" "{" { comando } "}"
          |
          "se" "(" expressao (">" | "<" | ">=" | "<=" | "==" | "!=")
          expressao ")" "{" { comando } "}"
          [ "senao" "{" { comando } "}" ]
          |
          identificador "(" [ expressao {"," expressao} ] ")" ";"
          |
          "entrada" "=" identificador
          [ "[" ( numero | expressao ) "]" ] ";"
          |
          "saida" "=" ( expressao | string ) ";"
          ] .


expressao = ( identificador [ "[" ( numero | expressao ) "]" ]
            | identificador "(" [ expressao  { "," expressao } ] ")"
            | numero | "(" expressao ")" )
            { ( "*" | "/" )
            ( identificador [ "[" ( numero | expressao ) "]" ]
            | identificador "(" [ expressao {"," expressao } ] ")"
            | numero | "(" expressao ")" ) }
            { ( "+" | "-" )
            ( identificador [ "[" ( numero | expressao ) "]" ]
            | identificador "(" [ expressao {"," expressao } ] ")"
            | numero | "(" expressao ")" )
```

```
{ ( "*" | "/" )
( identificador [ "[" ( numero | expressao ) "]" ]
| identificador "(" [ expressao {"," expressao } ] ")"
| numero | "(" expressao ")" ) } } .
```

3 Descrição do analisador léxico

A função desta etapa do compilador é receber o código fonte como entrada, dividi-lo em tokens e preencher parte da tabela de símbolos. Para esta funcionalidade, implementamos o autômato finito representado na figura 2, que recebe cada caractere do código fonte e cada vez que volta ao estado inicial, pode escrever um token na estrutura de fluxo de tokens e pode também colocar uma entrada na tabela de símbolos.

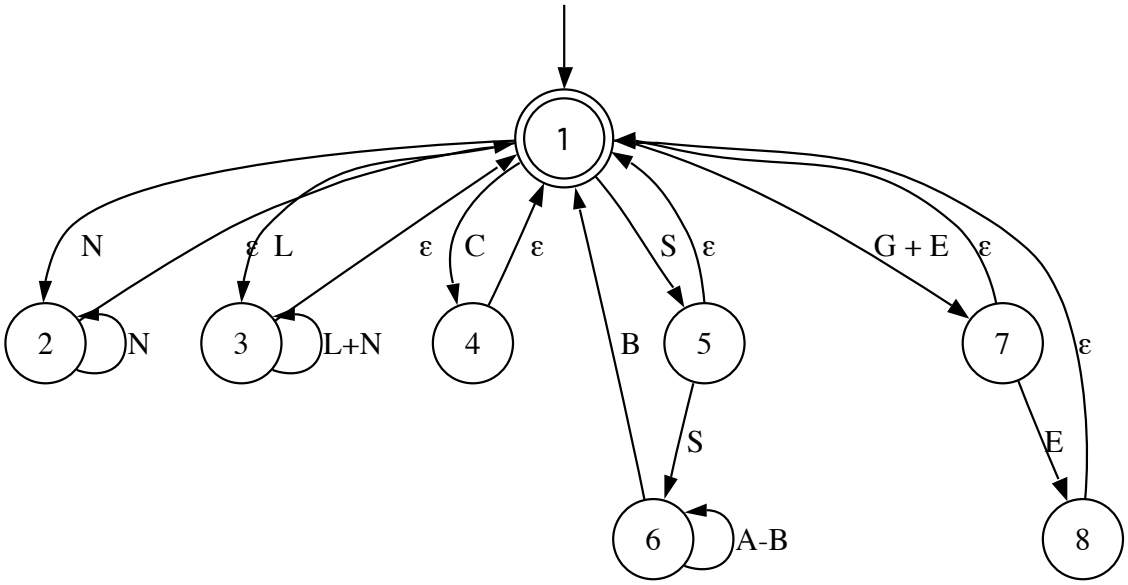

Figura 2: Autômato finito equivalente à gramática definida

Estados:
1 - Inicial
2 - Número
3 - Seqüência de caracteres
4 - Caracter especial
5 - Divisão
6 - Comentário
7 - Comparação ou atribuição
8 - Comparação

Entradas:
A - Qualquer caracter
B - Nova linha
C - Caracteres especiais (. , ; + - * ( ) [ ] { })
E - Igual
(=) G - Maior ou menor (< >)
L - Letra [A-z]
N - Número [0-9]
S - Barra (/)

Dividimos a implementação do analisador léxico em duas etapas: a primeira monta o autômato em uma estrutura interna de acordo com um arquivo XML e o segundo simula tal autômato a partir do código fonte.

3.1 Montagem do autômato finito

3.1.1 XML

Nesta etapa, o programa lê um arquivo XML como indicado na figura 3. Nele temos uma tag *gramatica* que engloba todo o arquivo. Dentro desta tag, há um ou mais estados identificados com a tag *estado*. Dentro dos estados há as tags *id*, que é um número para identificar o estado, a tag *final*, para informar se o estado pode finalizar um token, a tag *tipo* para classificar o token de acordo com a o id mostrado na tabela 1 quando o fim do mesmo é identificado e zero ou mais transições, identificadas pelas tags *transicao*. Cada transição tem um ou mais caracteres que indicam quais entradas que ativam a transição, identificados pela tag *entradas* e o respectivo estado que é ativado.

```
<gramatica>
     <estado>
          <id>0</id>
          <final>false</final>
          <tipo>0</tipo>
          <transicao>
               <entradas>0123456789</entradas>
               <proximo>1</proximo>
          </transicao>
          [Outras transições]
     </estado>
     [Outros estados]
</gramatica>
```

Figura 3: Estrutura do autômato descrito em XML

Tag *tipo*	Significado
0	Estado Inicial
1	Número
2	Seqüência de caracteres
3	Caractere especial
4	Comentário
5	String
6	Reservado

Tabela 1: Relação entre a identificação da tag *tipo* do XML e o tipo do estado

No exemplo da figura 3, é mostrado apenas um estado, inicial, que não finaliza tokens. Ao receber um número, há uma transição para o estado 1, que não é mostrado.

3.1.2 Estrutura do autômato

Podemos ver na figura 4 o diagrama de classes da estrutura em qual o autômato do arquivo XML é transformado. Temos basicamente três classes: AFD (autômatos), Estado e Transição. Iremos descrever a partir de agora cada uma destas classes, seus métodos e atributos. Como os *gets* e *sets* não necessitam de explicação, serão ignorados.

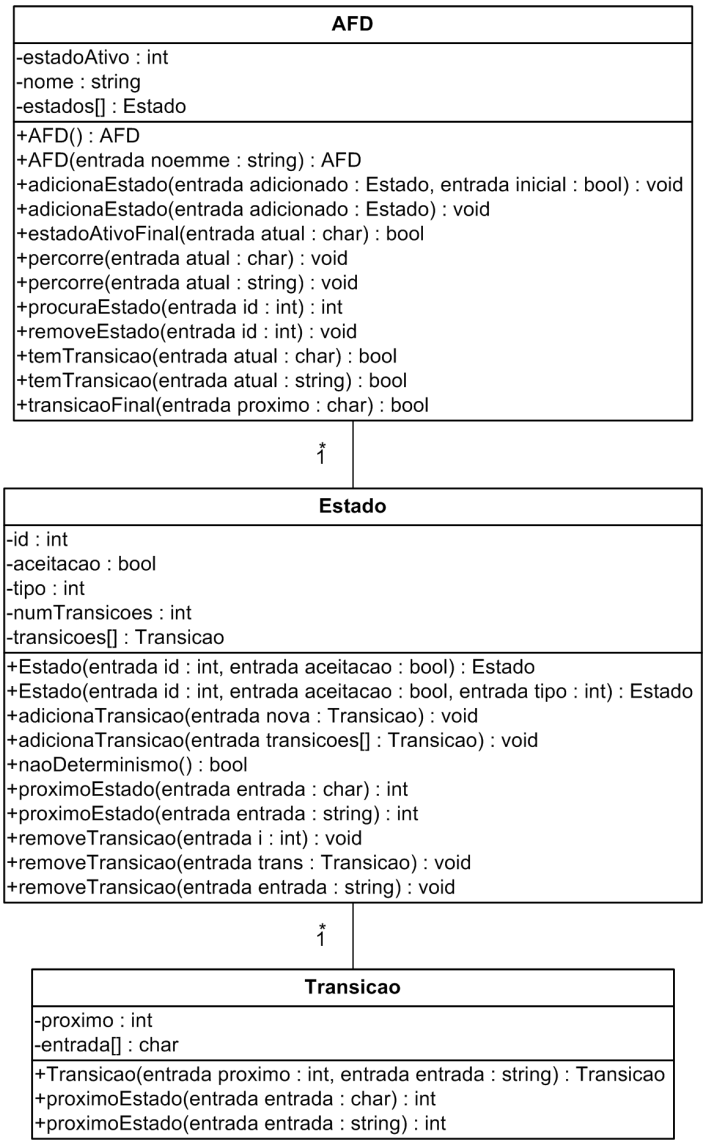

Figura 4: Diarama de classes do autômato

AFD

Atributos Um autômato, como esperado, tem uma série de estados, armazenados no vetor *estados*. A variável *estadoAtivo* indica qual deles está ativo naquele momento e *nome* armazena o nome do autômato.

Métodos Os dois métodos *adicionaEstado* obviamente adicionam um estado na máquina. A diferença é que um define o estado como ativo se ele é considerado inicial, e o outro, caso a entrada *inicial* seja verdadeira. *estadoAtivoFinal* verifica se o estado atual é final ou não. *removeEstado* remove um estado do autômato. O método *procuraEstado* retorna o id do vetor estados equivalente ao estado com o id definido dentro da classe Estado. Já o método *temTransicao*, retorna um booleano que indica se há uma transição válida do estado ativo para a o caractere de entrada. O método *percorre* é um dos mais importantes. Ele é usado na simulação do autômato, onde o caractere de entrada faz a transição para o próximo estado automaticamente e altera todas as variáveis internas do autômato. Finalmente, o método *transicaoFinal* indica se o estado ativo emite um token ou não. Os métodos *temTransicao* e *pecorre* têm duas versões, pois os autômatos são usados tanto nesta etapa de tokens, de caractere em caractere, ou mais à frente, com strings.

Estado

Atributos Cada estado é identificado por seu *id*. Há também a indicação se ele emite um token ou não, pela variável *aceitacao*. A variável *tipo* foi descrita anteriormente na tabela 1, e indica qual o tipo de token a partir deste estado. *numTransicoes* guarda o número de transições que saem deste estado e *transicoes* é um vetor de transições.

Métodos Há basicamente quatro métodos relevantes nesta classe. O primeiro, *adicionaTransicao*, adiciona uma transição ao estado. Já o método *proximoEstado* retorna o id do próximo estado quando o caractere de entrada é recebido. *naoDeterminismo* verifica se o estado tem apenas uma transição para cada entrada ou mais. Finalmente, *removeTransicao* elimina uma transicao do estado.

Transicao

Atributos Uma transição é composta pelo id do estado que ficará ativo (*proximo*) quando alguma dos caracteres do vetor *entrada* é recebido.

Métodos O método *proximoEstado* retorna o id do estado que é ativado com o caractere de entrada ou -1 caso não haja transição válida.

3.2 Simulação do autômato finito

Para a simulação do autômato, tomamos como o mesmo já definido na estrutura descrita no item anterior. Como vimos, o método *percorre* é o que realiza esta função com a ajuda dos outros métodos já descritos. Portanto, focaremos nesta etapa a descrição das estruturas que recebem os dados vindos da máquina.

3.2.1 Estrutura do fluxo de tokens

FluxoTokens

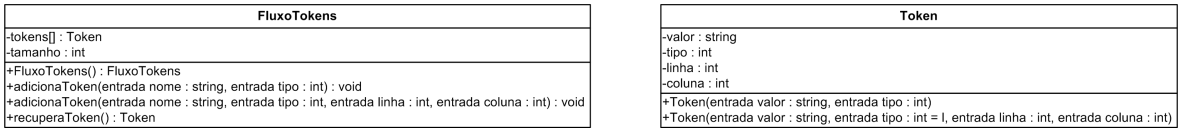

Figura 5: Diarama de classes do fluxo de tokens

Atributos Um vetor de tokens é armazenado na variável *tokens*, onde seu tamanho é armazenado na variável *tamanho*.

Métodos O método *adicionaToken* adiciona um token no vetor de tokens, e o método recuperaToken, devolve o próximo token da lista e o elimina.

Token

Atributos A variável *valor* guarda o nome das variáveis representadas pelos tokens. A variável *tipo* estabelece uma relação numérica com a tabela de símbolos e a variável. *linha* e *coluna* armazenam a posição do tokenno código.

Métodos Não há métodos a serem descritos.

3.2.2 Estrutura da tabela de símbolos

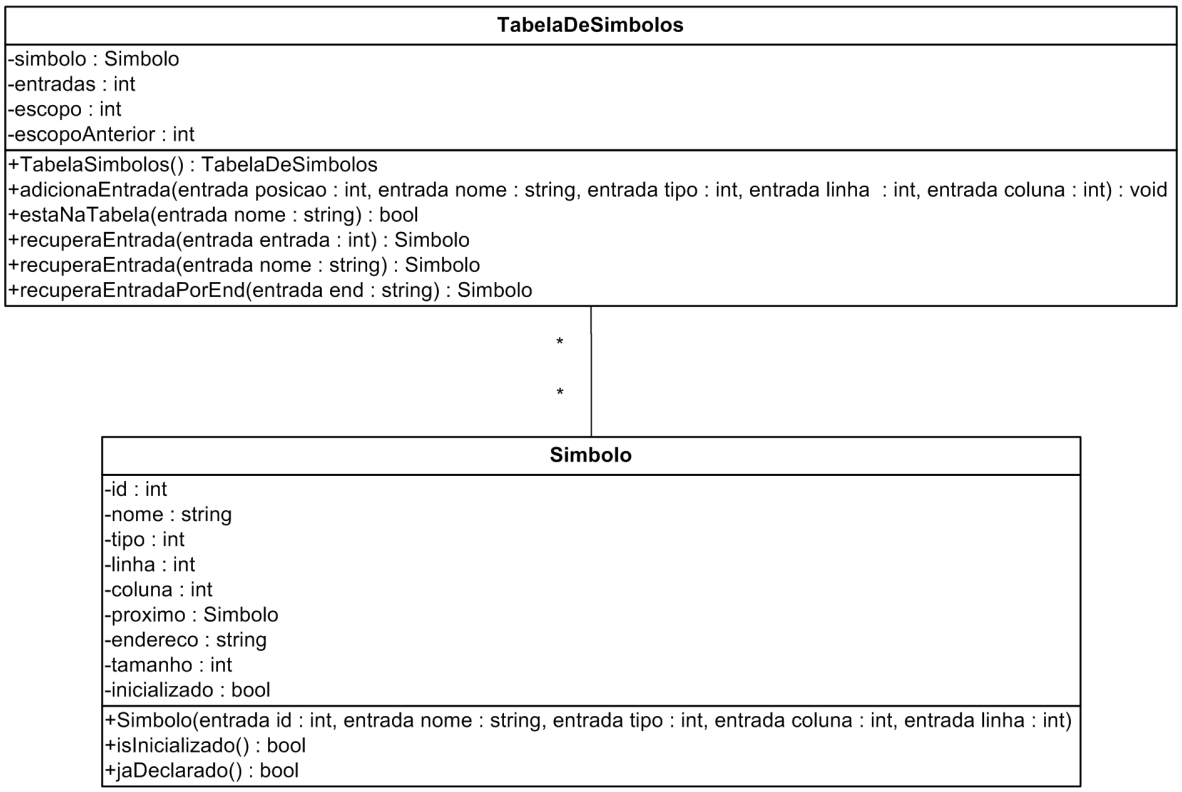

Figura 6: Diarama de classes da tabela de símbolos

TabelaDeSimbolos

Atributos Apenas o primeiro simbolo da tabela é armazenado na variável *simbolo*, onde seu tamanho é armazenado na variável *entradas*. Posteriormente, na parte da análise sintática as variáveis *escopo* identificrão o escopo ao qual a tabela pertence e *escopoAnterior* a tabela vinculada ao escopo anterior da atual.

Métodos O método *adicionaEntrada* adiciona um símbolo na tabela de símbolos. *estaNaTabela* verifica se um dado símbolo está na tabela. As diversas instâncias de *recuperaEntrada* recuperam um símbolo da tabela com diferentes valores de procura.

Simbolo

Atributos *Id* é a ligação numérica de um símbolo com o fluxo de tokens. *Nome* é um identificador para o símbolo, *tipo* é uma classificação do mesmo tipo da tabela 1. *Linha* e *coluna* indicam onde cada token está dentro do código fonte, e próximo é o próximo token da tabela. *proximo* é o próximo símbolo da tabela, cujo tamanho é indicado por *tamanho*. *inicializado* e *endereco* serão usados mais adiante. O primeiro indica se a váriavel já foi alocada no código, e a segunda indica seu endereço.

Métodos *isInicializado* retorna se a variável já alocada ou não, e *jaDeclarado* indica se ela já está na tabela.

3.3 Classes especiais

Definimos também algumas classes especiais, responsáveis por guardar dados e facilitar uma rápida modificação em dados caso necessário. Temos duas classes, *TiposLexico* e *PalvrasReservadas*. A primeira apenas guarda os dados da tabela 1. Já a segunda, explicitada na figura 7, contém as palavras da figura 8. Internamente, o vetor *palavras* contém estas palavras e o método *resevada* indica se a seqüência de caracteres de entrada está nessa lista.

3.4 Teste

Para testar esta etapa do compilador, criamos um pequeno código que é capaz de testar todas as funcionalidades da mesma. Ele é apresentado na figura 9. Nele testamos todos os tipos identificáveis do analisador léxico: seqüência de caracteres, números, caracteres especiais, comparadores, atribuidores, comentários, atribuidores aritméticos e um caractere inválido. O resultado, obtido a partir do log desta etapa, pode ser visto logo abaixo.

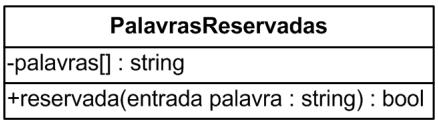

Figura 7: Diarama de classes da lista de palavras

Palavras
estrut
principal
retorno
inteiro
caracteres
booleano
entrada
saida
se
senao
enquanto
verdadeiro
falso

Figura 8: Palavras reservadas

```
principal () {

inteiro var1;
inteiro var2;

var1 = 0;
var2 = 3;

funcao();

enquanto (var1 < (var2+1) ){
var1 = var1 + 1;
}

var1 = (var1 * 5) / var1 + 3;
saida = var1;
}
```

Figura 9: Código de teste

3.4.1 Resultado

```
Token: principal, id: 6
Token: (, id: 3
Token: ), id: 3
Token: {, id: 3
Token: inteiro, id: 6
Token: var1, id: 2
Token: ;, id: 3
Token: inteiro, id: 6
Token: var2, id: 2
```

```
Token: ;, id: 3
Token: var1, id: 2
Token: =, id: 3
Token: 0, id: 1
Token: ;, id: 3
Token: var2, id: 2
Token: =, id: 3
Token: 3, id: 1
Token: ;, id: 3
Token: funcao, id: 2
Token: (, id: 3
Token: ), id: 3
Token: ;, id: 3
Token: enquanto, id: 6
Token: (, id: 3
Token: var1, id: 2
Token: <, id: 3
Token: (, id: 3
Token: var2, id: 2
Token: +, id: 3
Token: 1, id: 1
Token: ), id: 3
Token: ), id: 3
Token: {, id: 3
Token: var1, id: 2
Token: =, id: 3
Token: var1, id: 2
Token: +, id: 3
Token: 1, id: 1
Token: ;, id: 3
Token: }, id: 3
Token: var1, id: 2
Token: =, id: 3
Token: (, id: 3
Token: var1, id: 2
Token: *, id: 3
Token: 5, id: 1
Token: ), id: 3
Token: /, id: 3
Token: var1, id: 2
Token: +, id: 3
Token: 3, id: 1
Token: ;, id: 3
Token: saida, id: 6
Token: =, id: 3
Token: var1, id: 2
Token: ;, id: 3
Token: }, id: 3
```

4 Projeto do analisador sintático

O analisador sintático consiste de um autômato de pilha estruturado criado automaticamente por um meta-analisador a partir da descrição simplificada da gramática na notação de Wirth apresentada no item 2.3.

4.1 Meta-analisador

Para facilitar o processo de obtenção do autômato de pilha estruturado e de suas submáquinas, referentes aos não-terminais de nossa linguagem, foi criado um meta-analisador que, a partir da gramática na notação de Wirth, automatiza a geração do reconhecedor sintático. O desenvolvimento do meta-analisador seguiu o método apresentado em [3] .

Assim, no desenvolvimento do meta-analisador, procuramos utilizar o mesmo esquema criado para gerar o autômato na parte léxica do projeto, ou seja, primeiro descrever o autômato em XML e depois monta-lo na estrutura utilizada para sua representação dentro do compilador.

Algumas modificações foram feitas no XML de entrada e na estrutura utilizada para representar o autômato.

4.2 Montagem do meta-analisador

4.2.1 XML

Para conseguir representar o autômato de pilha estruturado representativo da gramática de Wirth, constituído das submáquinas WIRTH e EXPR, utilizamos a estrutura XML apresentada à seguir:

```
<gramatica>
     <nterminal>
        <nome>WIRTH</nome>
        <estado>
          <id>0</id>
          <final>false</final>
          <transicao>
              <entradas>NTERM</entradas>
              <proximo>2</proximo>
          </transicao>
          [Outras transições]
        </estado>
        [Outros estados]
     </nterminal>
     [Outros não-terminais]
</gramatica>
```

Assim, através dessa estrutura, definimos no arquivo **metacompilador.xml** as duas submáquinas WIRTH e EXPR, responsável pelo reconhecimento das gramáticas em notação de Wirth.

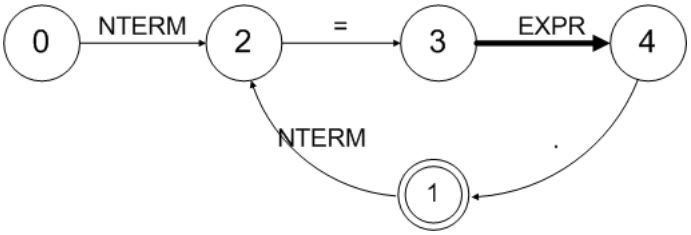

Figura 10: Submáquina WIRTH

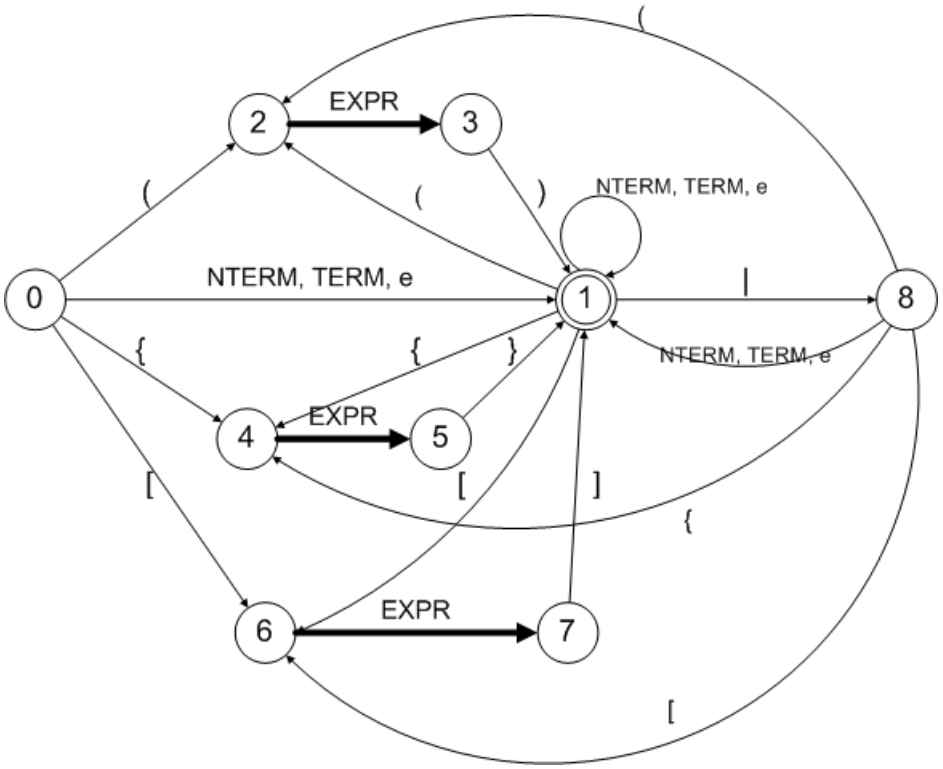

Figura 11: Submáquina EXPR

4.2.2 Estrutura do autômato

Além das estruturas já apresentadas no item 3.1.2, criamos uma classe **APE** (Autômato de Pilha Estruturado), que armazena a lista de submáquinas descritas pela classe **AFD** (Autômato Finito Determinístico).

Assim, a clase **MontaMetaAPE**, a partir do XML mostrado no item anterior, cria para cada não-terminal (WIRTH e EXPR) uma submáquina através da classe **AFD**, e ao final, estas submáquinas são adicionadas à um objeto da classe **APE**.

4.2.3 Léxico

Antes de simular o autômato de pilha estruturado da gramática de Wirth, é necessário tratar o arquivo de entrada **gramatica.txt**, que contêm a descrição da nossa linguagem.

Assim, usando o mesmo método utilizado no item 3, trocamos o autômato responsável por identificar o tipo de cada token (TERM, NTERM, ESPECIAL), pelo apresentado na figura 12 e descrito no arquivo **metalexico.xml**.

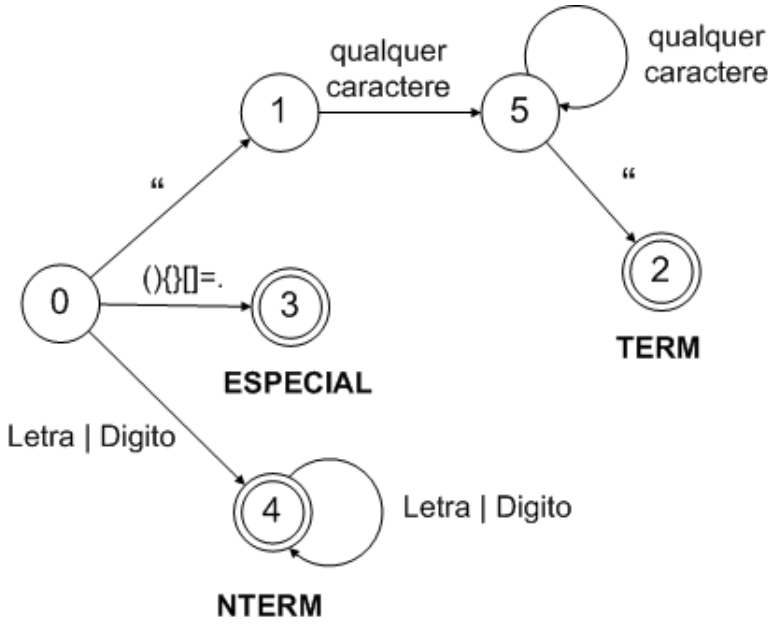

Figura 12: Autômato finito usado na parte léxica do meta-analisador

4.3 Funcionamento do meta-analisador

A classe **MetaCompilador**, através do método **executa**, é responsável por efetuar as ações do meta-analisador.

Assim, primeiramente, é montado o meta-analisador através da classe **MontaMetaAPE**, como já apresentado no item 4.2.

Depois disso, executa-se o léxico (item 4.2.3) que lê o arquivo de entrada **gramatica.txt** para criar o fluxo de tokens.

Esse fluxo de tokens é então usado através da classe **PercorreMetaAPE** para criar o autômato de pilha estruturado da nossa linguagem. Para isso, a classe **PercorreMetaAPE** implementa as ações semânticas apresentadas em [3].

Em seguida, o objeto da classe **APE** obtido é minimizado, retirando as transições em vazio e eliminando não determinismos através do seu método **minimiza**.

Obtemos, assim, o autômato de pilha estruturado da linguagem FM.

4.4 Reconhecedor de pilha determinístico

Uma vez construído o reconhecedor, inicia-se o reconhecimento do código de entrada. Novamente é usado a estrutura APE, que desta vez contém três submáquinas que podem ser vistas nas figuras 14, 15 e 16. Uma vez que as submáquinas já não têm não-determinismos e transições vazias, o processo de reconhecimento é direto.

Para cada token recebido da análise léxica, um *lookahead* é feito para se determinar se há uma transição válida na submáquina ou em alguma outra submáquina que possa ser derivada desta. A cada passo no reconhecimento, dados são passados para a análise semântica, que será descrita a seguir, e que é responsável pela geração de código.

5 Descrição do analisador semântico

Como visto na seção anterior, tokens são passados desde a a etapa do reconhecedor para a análise semântica, a fim de se gerar código. Nesta etapa, uma série de pilhas são usadas para manter registro do estado atual da geração de código.

Dentre as estruturas mais importantes usadas nesta etapa, podemos citar:

- Vetor de escopos: responsável por manter um índice dos diferentes escopos (blocos) do código
- Tabela de símbolos: usada novamente para manter registro das variáveis e sua declaração
- Pilha de tokens: mantém um registro dos últimos tokens recebidos, a fim de se manter o contexto
- Pilhas de declarações e instruções: guardam as instruções que serão usadas para gerar o código
- Pilha de operadores e operandos: usado na submáquina de expressão

5.1 MVN

O código objeto se dará na linguagem MVN. Podemos observar na figura 13 o mnemônicos disponíveis para a linguagem.

5.2 Submáquina programa

A submáquina programa é responsável pela criação de código de registro, funções e do corpo principal do programa.

5.3 Submáquina comando

As funções definidas para a máquina comando são a atribuição de valores, a definição de variáveis, comparações, entrada e saída de dados e chamadas de funções.

5.4 Submáquina expressão

A submáquina expressão funciona basicamente com duas pilhas, uma de operadores e uma de operandos.

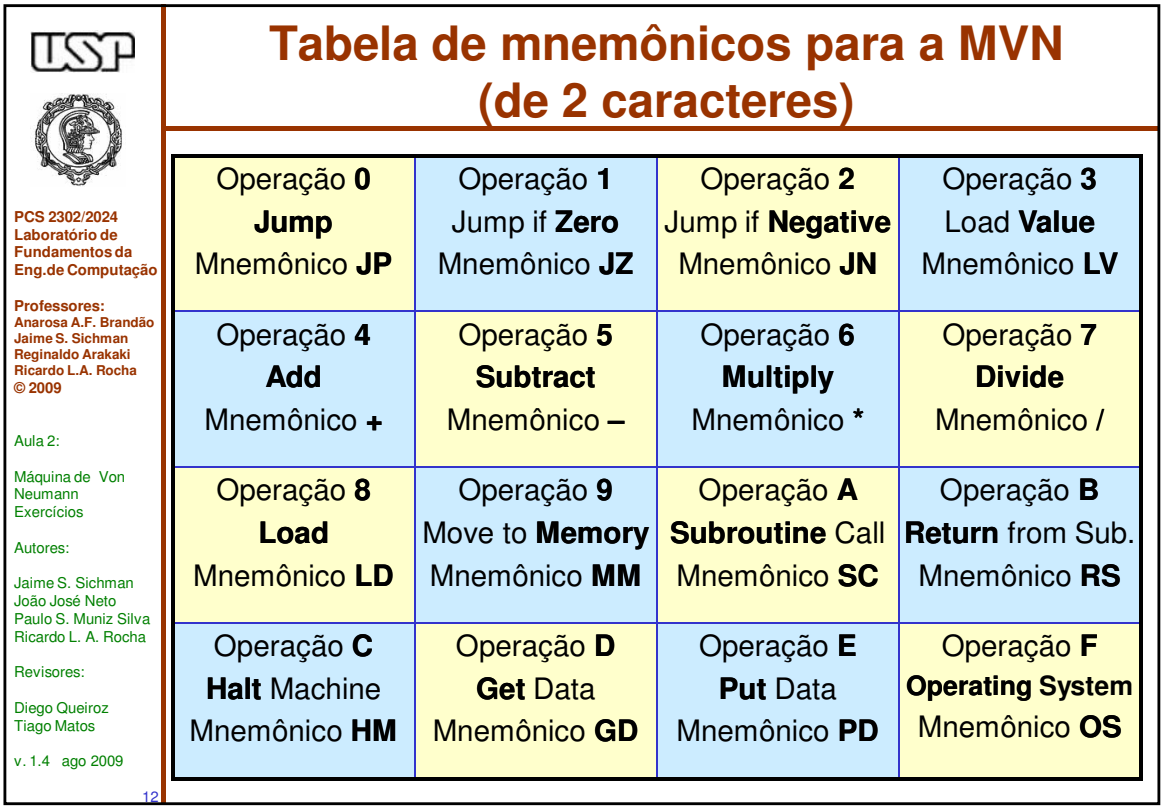

Figura 13: Mnemônicos da MVN

6 Conclusões

6.1 Limitações

Infelizmente, devido à restrições de tempo, algumas funções não puderam ser implementadas na etapa final do projeto, a semântica. Elas estão listadas à seguir:

- Registro
- Vetores
- Uso de parâmetros
- Expressões booleanas
- *Strings*
- Caracteres

Em todos os casos, o código chega a ser aceito, já que o reconhecedor está funcional. Porém, nehum código é gerado.

6.2 Testes

Para se testar todas as funções implementadas, realizamos um teste final com o seguinte código:

```
// exemplo de funcao
inteiro funcao(){
inteiro param;
inteiro var1;

param = 0;
var1 = 15;

saida = var1;

enquanto ( param <= 2) {
param = param + 1;
saida = param;
}
retorno param;
}

// rotina de inicio do programa
principal () {

// declaracao de variavel para tipo int e boolean
inteiro var1;
inteiro var2;
booleano bol;

// atribuicao de valor com verificacao de tipo
bol = verdadeiro;
var1 = 0;
var2 = 3;

// nao eh possivel atribuir o valor de retorno da funcao para uma variavel
funcao();

// ifs e whiles aninhados e operacoes de comparacao com o valor de expressoes
enquanto ( var1 < ( var2 * 2) ){
var1 = var1 + 1;

// if sem else
se ( (var1 - 2) > 0 ){
saida = var1;
}
}
```

```
// if com else
se (var1 == var2){
saida = var1 + var2;

}senao{
var1 = var2;
saida = var1;
}

// expressoes fazem verificacao de tipo
var1 = (var1 * 5) / var1 + 3;

// operacao de saida
saida = var1;
}
```

O código de saída pode ser observado abaixo.

```
@ /0
JP INICIO
NUM_0 K /0 ; 0
NUM_5 K /5 ; 5
NUM_2 K /2 ; 2
NUM_1 K /1 ; 1
VAR_0 K /00 ; param
VAR_1 K /00 ; var1
NUM_3 K /3 ; 3
VAR_2 K /00 ; var1
VAR_3 K /00 ; var2
FUNC_0 JP /00
LD NUM_0 ; VAR_0 = NUM_0
MM VAR_0
LD NUM_5 ; VAR_1 = NUM_5
MM VAR_1
LD VAR_1 ; TEMP_0 = VAR_1
MM TEMP_0
LD TEMP_0
MM ARG_INT
SC PRINT_INT
LD RESULT_ASC
PD /100 ; SAIDA = TEMP_0
LD NEW_LINE
PD /100
LOOP_0 LD /00
LD VAR_0 ; TEMP_0 = VAR_0
MM TEMP_0
```

```
LD NUM_2 ; TEMP_1 = NUM_2
MM TEMP_1
LD TEMP_1 ; WHILE ( TEMP_0 <= TEMP_1 )
-TEMP_0
JN ENDLOOP_0
LD VAR_0 ; TEMP_0 = VAR_0 + NUM_1
+ NUM_1
MM TEMP_0
LD TEMP_0 ; VAR_0 = TEMP_0
MM VAR_0
JP LOOP_0
ENDLOOP_0 LD /00 ; END WHILE
LD VAR_0 ; TEMP_0 = VAR_0
MM TEMP_0
LD TEMP_0
MM ARG_INT
SC PRINT_INT
LD RESULT_ASC
PD /100 ; SAIDA = TEMP_0
LD NEW_LINE
PD /100
LD VAR_0 ; RETURN VAR_0
RS FUNC_0
INICIO LD /00
LD NUM_0 ; VAR_2 = NUM_0
MM VAR_2
LD NUM_3 ; VAR_3 = NUM_3
MM VAR_3
SC FUNC_0
LOOP_1 LD /00
LD VAR_2 ; TEMP_0 = VAR_2
MM TEMP_0
LD VAR_3 ; TEMP_2 = VAR_3 + NUM_1
+ NUM_1
MM TEMP_2
LD TEMP_2 ; TEMP_1 = TEMP_2
MM TEMP_1
LD TEMP_1 ; WHILE ( TEMP_0 < TEMP_1 )
-TEMP_0
JN ENDLOOP_1
JZ ENDLOOP_1
LD VAR_2 ; TEMP_0 = VAR_2 + NUM_1
+ NUM_1
MM TEMP_0
LD TEMP_0 ; VAR_2 = TEMP_0
MM VAR_2
```

```
JP LOOP_1
ENDLOOP_1 LD /00 ; END WHILE
IF_0 LD /00
LD VAR_2 ; TEMP_0 = VAR_2
MM TEMP_0
LD VAR_3 ; TEMP_1 = VAR_3
MM TEMP_1
LD TEMP_0 ; IF ( TEMP_0 == TEMP_1 )
-TEMP_1
JZ IF_EQ_0
JP ELSE_0
IF_EQ_0 LD /00
LD VAR_2 ; TEMP_1 = VAR_2 + VAR_3
+ VAR_3
MM TEMP_1
LD TEMP_1 ; TEMP_0 = TEMP_1
MM TEMP_0
LD TEMP_0
MM ARG_INT
SC PRINT_INT
LD RESULT_ASC
PD /100 ; SAIDA = TEMP_0
LD NEW_LINE
PD /100
JP ENDIF_0
ELSE_0 LD /00 ; ELSE
LD VAR_3 ; VAR_2 = VAR_3
MM VAR_2
LD VAR_2 ; TEMP_0 = VAR_2
MM TEMP_0
LD TEMP_0
MM ARG_INT
SC PRINT_INT
LD RESULT_ASC
PD /100 ; SAIDA = TEMP_0
LD NEW_LINE
PD /100
ENDIF_0 LD /00 ; END IF
LD VAR_2 ; TEMP_0 = VAR_2 * NUM_5
* NUM_5
MM TEMP_0
LD TEMP_0 ; TEMP_1 = TEMP_0 / VAR_2
/ VAR_2
MM TEMP_1
LD TEMP_1 ; TEMP_2 = TEMP_1 + NUM_3
+ NUM_3
```

```
MM TEMP_2
LD TEMP_2 ; VAR_2 = TEMP_2
MM VAR_2
LD VAR_2 ; TEMP_0 = VAR_2
MM TEMP_0
LD TEMP_0
MM ARG_INT
SC PRINT_INT
LD RESULT_ASC
PD /100 ; SAIDA = TEMP_0
LD NEW_LINE
PD /100
FIM HM /00
TEMP_0 K /00
TEMP_1 K /00
TEMP_2 K /00
TEMP_3 K /00
TEMP_4 K /00
TEMP_5 K /00
TEMP_6 K /00
TEMP_7 K /00
TEMP_8 K /00
TEMP_9 K /00
TEMP_10 K /00
TEMP_11 K /00
TEMP_12 K /00
TEMP_13 K /00
TEMP_14 K /00
TEMP_15 K /00
TEMP_16 K /00
TEMP_17 K /00
TEMP_18 K /00
TEMP_19 K /00
TEMP_20 K /00
PRINT_INT JP /00
LD ARG_INT ; CONVERT INT TO ASC
+ OFFSET
MM RESULT_ASC
LD STRING_1
PD /100 ;
LD STRING_2
PD /100 ;
LD STRING_3
PD /100 ;
LD STRING_4
PD /100 ;
```

```
LD STRING_5
PD /100 ;
LD STRING_6
PD /100 ;
LD STRING_7
PD /100 ;
LD STRING_6
PD /100 ;
RS PRINT_INT
ARG_INT K /00 ; ARG INT
RESULT_ASC K /00 ; RESULT ASC
OFFSET K /30 ; OFFSET
STRING_1 K /53 ; S
STRING_2 K /41 ; A
STRING_3 K /49 ; I
STRING_4 K /44 ; D
STRING_5 K /41 ; A
STRING_6 K /20 ;
STRING_7 K /3d ; =
NEW_LINE K /0a ; NEW LINE
```

Referências

[1] Neto, J. J.: *Introdução à compilação*, 1ª edição, LTC, 1987

[2] Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman J. D.: *Compiladores: Princípios, técnicas e ferramentas*, 2ª edição, Pearson, 2008

[3] Neto, J. J., Parientem C. B., Leonardi F.: *Compiler Construction: A Pedagogical Approach*. Proceedings of the ICIE. Buenos Aires. 1999. [disponível para download em www.pcs.usp.br/ lta]

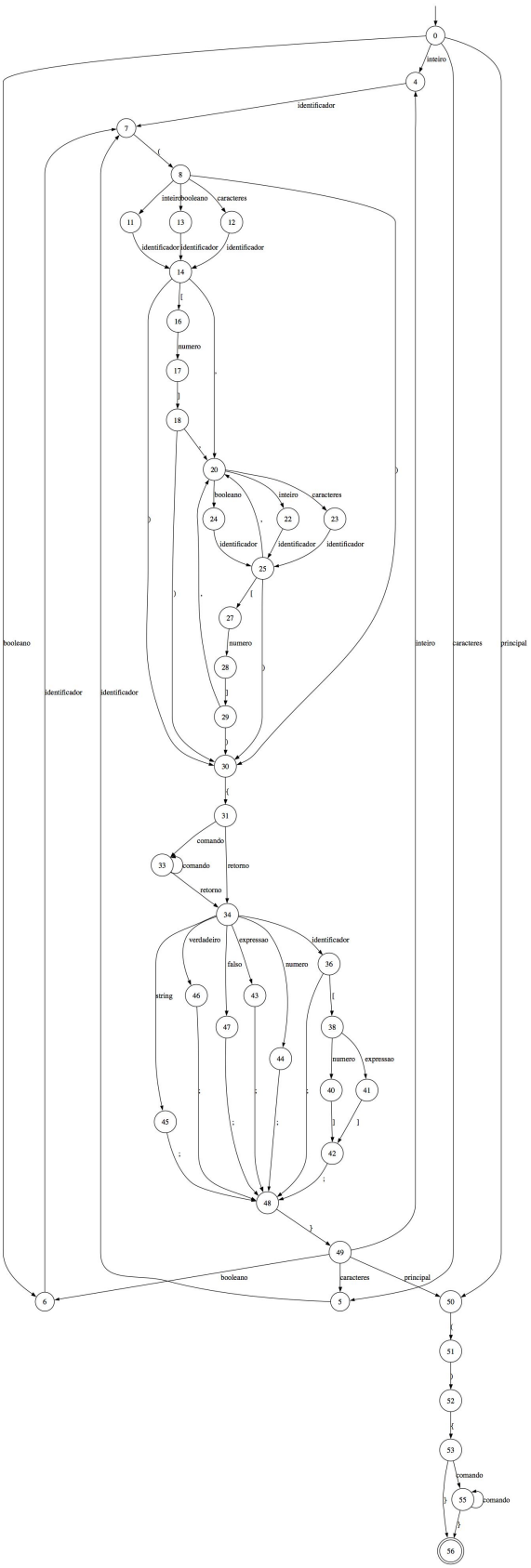

Figura 14: Autômato finito equivalente à submáquina programa

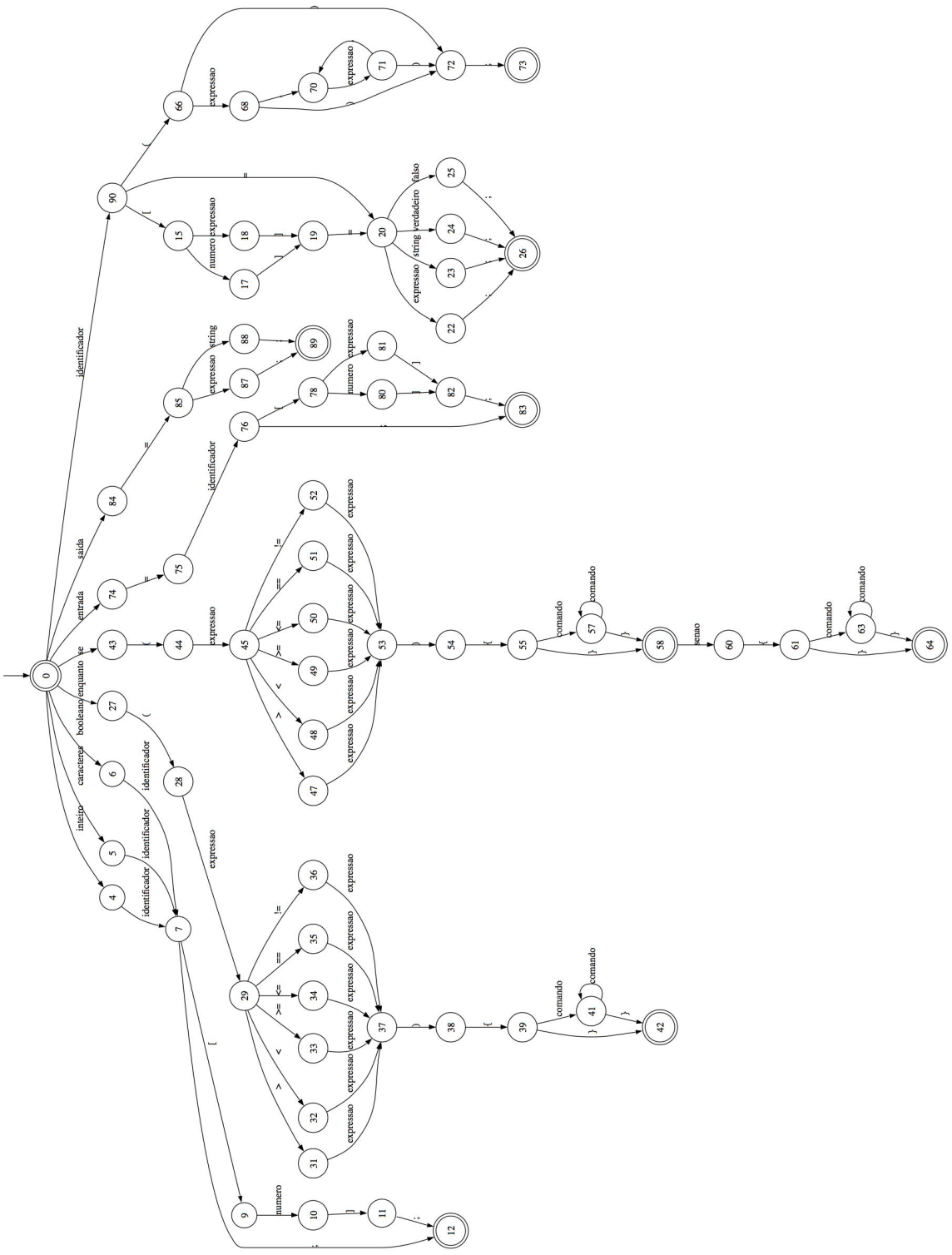

Figura 15: Autômato finito equivalente à submáquina expressão

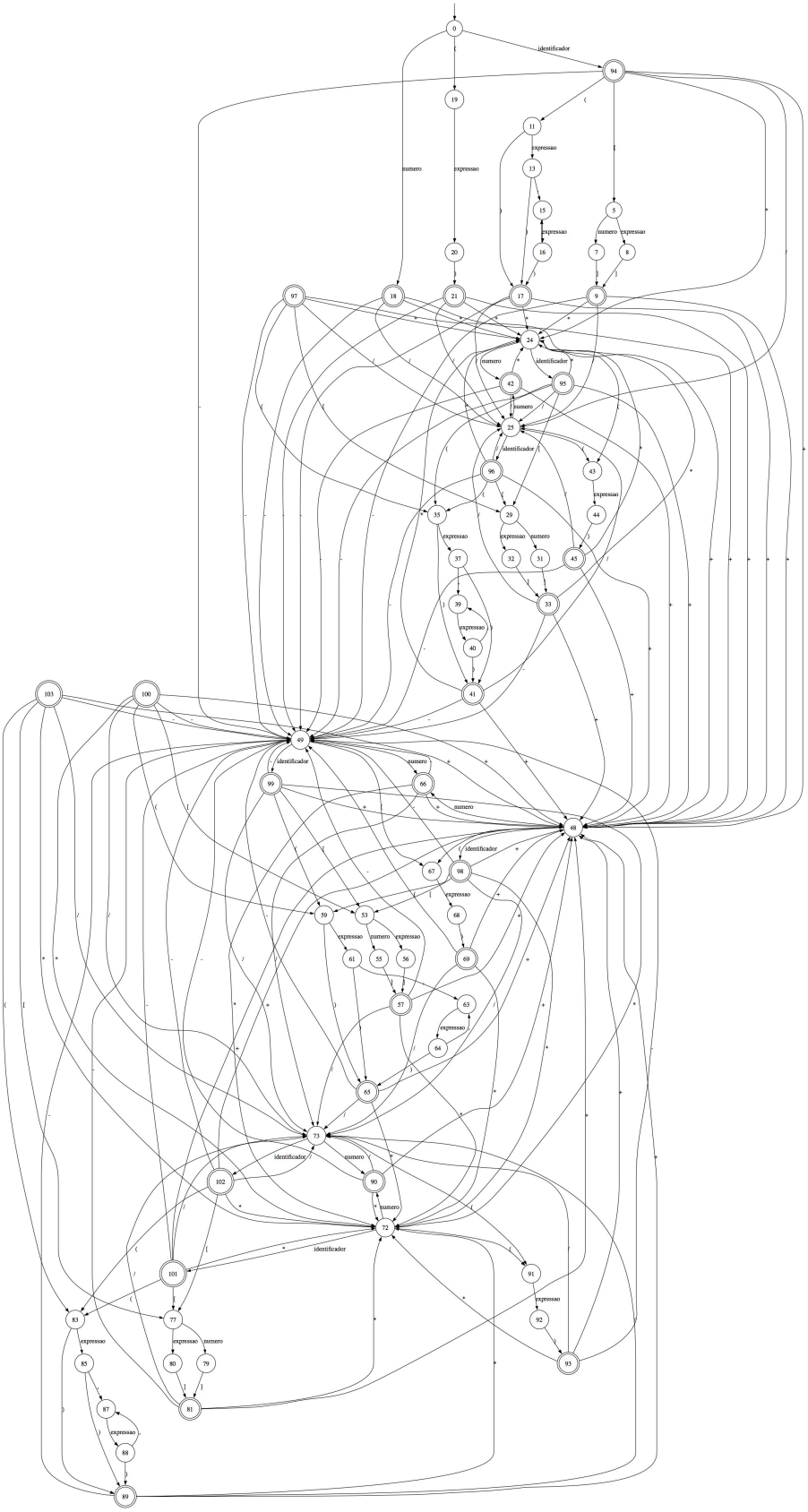

Figura 16: Autômato finito equivalente à submáquina comando