UPA

Gabriel Lopes dos Santos
33465

Lucas Gutierrez Villas Boas
33938

Paulo Eduardo Paes Salomon
33934

Pedro Felipe Dominguite
33713

Resumo

O presente artigo visa apresentar uma proposta de linguagem de programação e a criação de um analisador léxico e sintático da linguagem para a disciplina de Compiladores (ECOM06) do curso de Engenharia de Computação da Universidade Federal de Itajubá, ministrada pela professora Thatyana de Faria Piola Seraphim. A linguagem supracitada tem fins didáticos, propondo melhorias quanto ao Portugol (ou Português Estruturado), frequentemente utilizado em ciclos introdutórios de computação, e, também, uma maior aproximação da linguagem C, que, em geral, é ensinada em sequência nos cursos técnicos e superiores de computação e informática.

Parte I
Definição Teórica da Linguagem

1 Introdução

A linguagem de programação UPA surgiu a partir da ideia de aperfeiçoar a leitura de código, tornar mais intuitiva a declaração e atribuição de variáveis e o uso de comandos básicos de linguagens de programação, tais como laços de repetição e estruturas condicionais em relação à forma como outras linguagens utilizadas nos cursos introdutórios de programação o fazem.

Boa parte das linguagens de programação adotadas para a prática de ensino de programação no Brasil possuem estruturas truncadas e, por vezes, não difíceis, mas dispendiosas de serem escritas. Muitas delas, como o conhecido *Portugol*, não implementa comandos, atalhos ou sobrecargas já presentes, há um bom tempo, em linguagens como C, C++, Java e Python.

Além de propor um intermédio entre as linguagens citadas acima e uma melhor aprendizagem da estrutura de um código de programa, através da boa legibilidade e facilidade da manipulação da linguagem, a linguagem UPA é escrita em português e suporta o uso de algumas estruturas de dados mais comuns e utilizadas na programação, o que provê, já de antemão, uma boa inserção do jovem programador no universo computacional.

1.1 Linguagens Regulares

Tais linguagens são dadas pelos símbolos sob o alfabeto Σ, cuja sequência é obtida através de um conjunto de regras:

α^n: repetição do símbolo α n vezes. Se α for um conjunto, tal regra delimita uma palavra de tamanho n, composta pelos símbolos desse conjunto.

Por exemplo, sendo α um símbolo, α^2: $\alpha\alpha$. Caso $\alpha = \{a, b\}$, α^3 pode ser dado como aaa, aab, aba, abb, baa, bab, bba ou bbb.

α^*: possibilidade de repetição de símbolos de nenhuma a infinitas vezes, formando palavras de tamanho zero a quaisquer outros tamanhos.

Por exemplo, sendo α um símbolo, α^*: $\alpha\alpha\alpha$. Caso $\alpha = \{a, b\}$, α^* pode ser dado como a, b, ab, ba, aaa, abbaa...

α^+: possibilidade de repetição, obrigatoriamente diferente de zero, de quaisquer tamanhos de palavras, desde que cada símbolo de um conjunto seja expresso pelo menos uma vez.

1.2 Autômatos

Um autômato é um modelo reconhecedor de entradas, que atualiza seus estados conforme as mesmas. Um autômato finito possui um número finito de estados e, consequentemente, um número finito de transições.

O estado inicial é dado por q_0, que é o estado em que o autômato se encontra antes de ler o primeiro símbolo.

Caso uma entrada resulte exclusivamente em um estado seguinte, o autômato é denominado Determinístico. Se uma mesma entrada pode transicionar para estados diferentes, dá-se o nome de Não-Determinístico ao autômato.

Os autômatos finitos são modelos mais simples do modelo geral de reconhecedores, com suas principais particularidades:

1. Não possuem memória auxiliar;
2. Utilizam da fita de entrada para efetuar a leitura, de maneira que esta possui o tamanho da cadeia de entrada;
3. Leitura feita em apenas um sentido (da esquerda para a direita).

1.2.1 Autômato Finito Determinístico

Um autômato finito determinístico (AFD) pode ser definido como uma quíntupla:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Em que Q é o conjunto finito de estados, Σ é o alfabeto de entrada, δ é uma função de transição, q_0 é o estado inicial e F é o conjunto de estados finais, possuindo pelo menos um estado.

A função de transição δ associa pares da maneira (estado atual, entrada $\rightarrow$ próximo estado) ou $\delta(q_0, \alpha) = q_1$

Quando ocorre o esgotamento da cadeia de entrada, ou seja, todos os símbolos já foram lidos, deve-se avaliar o estado em que se encontra o autômato: se for um estado final (ou de aceitação), então o autômato reconheceu a cadeia, ou seja, a cadeia de entrada é válida. Caso o estado não seja um estado de aceitação, então a cadeia não é aceita, não sendo pertencente à linguagem definida pelo autômato.

Graficamente, um autômato pode ser dado por:

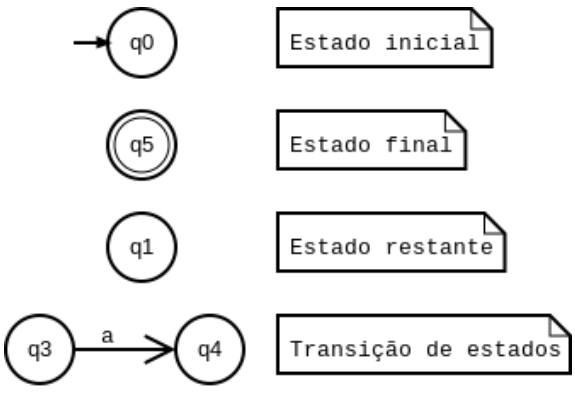

Figura 1: Estados de um autômato.

2 Alfabeto

O alfabeto, conjunto de todos os símbolos válidos desta linguagem, é dado por Σ = {*letra*, *numero*, *operador*, *operador de comparação*, *separador*}

Os conjuntos de símbolos são:
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, Y, Z}
numero = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
operador = {+, -, *, /, =, !}
operador de comparação = {>, < }
separador = {., :, „ , ;, '}. O conjunto separador é formado por: ponto, dois pontos, vírgula, espaço, ponto e vírgula e apóstrofo.

3 Variáveis

As variáveis nesta linguagem de programação podem ser dos tipos: *caracter*, *inteiro*, *real*, *VF* ou *constante*. A atribuição de valores para uma variável é dada essencialmente pelo operador "=", unicamente, seguido do valor ao qual se deseja atribuir à variável. A linguagem aceita apenas atribuições do mesmo tipo ao qual ela foi declarada. Desse modo, variáveis do tipo caractere, por exemplo, só podem receber caracteres e, jamais, um número inteiro ou um número de ponto flutuante.

3.1 Caracter

O tipo caracter define todo e qualquer caractere possível de ser representado pelos teclados em formato ABNT.

```
1 caracter c1 = '~';
2 caracter c2 = 'y';
3 caracter c3 = '7';
```

3.2 Inteiro

Variáveis do tipo inteiro são aqueles capazes de armazenar números inteiros compreendidos na faixa que vai de -9223372036854775808 a +9223372036854775807.

```
1 inteiro i1 = 7;
2 inteiro i2 = -65486156132;
3 inteiro i3 = 0;
```

3.3 Real

Variáveis do tipo real são aquelas capazes de armazenar valores do tipo ponto flutuante compreendidos da faixa que vai de $2,2 * 10^{-308}$ até $1,8 * 10^{+308}$.

```
1 real r1 = 1.0;
2 real r2 = -999999.8888888888;
3 real r3 = 7575.7575;
```

3.4 VF

As variáveis do tipo *VF* só podem assumir dois valores: *VERDADEIRO, FALSO.*

```
1 VF nomeVariavel = FALSO;
```

3.5 Constante

Constantes são variáveis que não terão seu valor modificado durante a execução do código.

```
constante inteiro nomeConstante = 10;
```

4 Comandos, Operadores Básicos, Instruções e Estruturas de Dados

Esta seção compreende alguns comandos essenciais da linguagem , tais como os de **atribuição**, de **entrada** e **saída** de dados, **condicionais** e de **repetição**, alguns operadores **relacionais**, **estrutura de dados** já implementadas e, ainda, instruções de **entrada** e **saída** de dados. Através dos exemplos abaixo, é possível compreender, também, a sintaxe da linguagem proposta.

4.1 Comandos

4.1.1 Se/Senão

Ambos *se* e *senão* são comandos condicionais, utilizados para tomada de decisões ao longo da lógica do programa.

O comandos *se* indica uma condição a ser analisada. Caso esta seja satisfeita, o trecho de código presente no escopo (entre chaves, portanto) será executado. Assim, um comando *se* tem seu escopo definido por colchetes. O formato de uma verificação **se** é:

```
se (expressao condicional) {
        //codigo a ser executado
}
```

O comando *senao* existe apenas após um *se* e tem seu escopo executado apenas se a expressão condicional do *se* que o antencede for falsa. Um comando *senao* tem seu escopo definido por colchetes. O formato de uma verificação **senao** é:

```
se (condicao1) {
        //codigo a ser executado
} senao {
        //codigo a ser executado se "condicao1" for falsa
}
```

4.1.2 Ss (*Senãose*)

O comando **ss** possui como intuito facilitar a concatenação de vários Ses e Senãos. Concatenação condicional sem o uso do **ss**.

```
se (condicao1) {
        //codigo a ser executado se "condicao1"
} senao se(condicao2) {
        //codigo a ser executado se "condicao2"
} senao {
        //codigo a ser executado se e "condicao1" e "condicao2" nao forem verdade
}
```

Concatenação condicional com o uso do **ss**.

```
se (condicao1) {
        //codigo a ser executado se "condicao1"
} ss(condicao2) {
        //codigo a ser executado se "condicao2"
}
```

4.1.3 Para/Para Cada

Ambos são comandos para realização de laços ou ciclos num programa. Por meio deles é possível repetir trechos de códigos e instruções.

para

No comando *para* utilizamos uma variável auxiliar que funciona como contador de repetições. Para isso, é preciso determinar o valor inicial, final e o passo dessa variável auxiliar. Por passo, entende-se o valor do qual a variável auxiliar é alterada a cada vez que a repetição é executada até que ela atinja (ou ultrapasse) o valor final. Um comando *para* tem seu escopo definido por colchetes. O formato de um laço **para** é:

```
1 inteiro i; //variavel auxiliar
2 para (i = 0; i < 10; i++) { //i = i + 1;
3         //codigo a ser executado 10 vezes
4 }
```

paracada

O comando *paracada* é uma espécie de *para* automatizado, em que o passo da variável auxiliar é necessariamente 1. Fornece-se uma variável auxiliar e um elemento pelo qual a variável irá iterar, isto é, irá percorrer. Esse elemento pode ser, por exemplo, uma *lista*, estrutura que será apresentada mais adiante. Um comando *paracada* tem seu escopo definido por colchetes. O formato de um laço **paracada** é:

```
1 inteiro i; //variavel auxiliar
2 lista inteiro [10] numeros; //lista de 10 inteiros
3 paracada (i : numeros) { //i = i + 1 ocorre implicitamente
4         //codigo a ser executado 10 vezes
5 }
```

4.1.4 Enquanto

O comando *enquanto* consiste de um laço (ou instrução de repetição) que é realizado sempre que uma condição de verificação é satisfeita. Um comando *enquanto* tem seu escopo definido por colchetes. O formato de um laço **enquanto** é:

```
1 enquanto (condicao) {
2         //codigo a ser executado ate que "condicao" seja satisfeita
3 }
```

4.1.5 Escolha / Caso

A estrutura **escolha / caso** é usada de forma a condensar uma cadeia de vários *se* e *senao* encadeados. O conteúdo de uma variável escolhida é comparado com um valor pré-definido, e caso a comparação seja verdadeira o código subsequente é executado. É tipicamente usado para criação de *menus*.

```
1  inteiro variavelEscolhida  = 3;
2  inteiro aux = 0;
3  escolha (variavelEscolhida) {
4         caso (2)
5                 aux = aux + 2;
6         caso (3)
7                 aux = aux + 3;
8  }
```

4.2 Instruções

4.2.1 Imprima

A instrução **imprima** é responsável por exibir informações na tela (*terminal*). É usada para criar interfaces pseudo-gráficas e melhorar a experiência de interação com o usuário.

```
inteiro resultado;
inteiro a;
inteiro b;
a = 10;
b = 20;
resultado = a + b;
imprima("O resultado final e:" + resultado + ".");
/*saida na tela:
O resultado final e 30.*/
```

4.2.2 Leia

A instrução leia é responsável por coletar informações da entrada principal (*teclado*). É usada para possibilitar a interação usuário-software.

```
inteiro resultado;
inteiro a;
inteiro b;
a = leia(inteiro, "Entre com o valor de a: ");
b = leia(inteiro, "Entre com o valor de b: ");
resultado = a + b;
```

4.3 Estrutura de Dados

4.3.1 Enumerado

Enumerado é um tipo de dado definido pelo usuário que define uma variável que vai receber apenas um conjunto restrito de valores.

```
enumerado bool = {falso = 0, verdade = 1};
```

4.3.2 Lista

A lista é uma coleção homogênea dos tipos inteiro, caractere ou VF, organizada no formato 1 linha e M colunas.

```
lista caractere [10] nomeLista;
```

4.3.3 Tabela

A tabela é uma coleção homogênea dos tipos inteiro, caractere ou VF, organizada no formato N linhas e M colunas.

```
tabela inteiro [10,10] nomeLista;
```

4.3.4 Estrutura

Esta declaração é capaz de criar um novo tipo de variável, de acordo com especificações. Para acesso dos tipos internos, deve-se usar ponto (.).

```
estrutura novoTipo {
        lista inteiro [5] lista1;
        lista caractere [5] lista2;
}
```

O código acima cria um novo tipo de variável, chamado *novoTipo*, o qual possui duas listas: uma de inteiro e uma de caracteres.

O acesso aos dados da lista se dá por:

```
novoTipo variavel1;
inteiro a;
caractere b;
a = variavel1.lista1[0];
b = variavel1.lista2[4];
```

4.4 Operadores

A linguagem conta com operadores relacionais, ou operadores conhecidos como "de comparação". Estes são: *maior* (>), *menor* (<), *menor ou igual* (<=), *maior ou igual* (>=), *igual* (==) e *diferente* (!=). Podemos incluir aqui, também, o operador de *negação* (!).

Os demais operadores lógicos podem sem analisados nas subseções abaixo.

4.4.1 E

E é um operador lógico cuja saída segue na tabela abaixo.

A	B	Saída
0	0	FALSO
0	1	FALSO
1	0	FALSO
1	1	VERDADE

Tabela 1: Tabela verdade operador *E*

Tipicamente é usado em testes condicionais.

```
1 se (condicao1 E condicao2) {
2         //codigo a ser executado;
3 }
```

4.4.2 OU

OU é um operador lógico cuja saída segue na tabela abaixo.

A	B	Saída
0	0	FALSO
0	1	VERDADE
1	0	VERDADE
1	1	VERDADE

Tabela 2: Tabela verdade operador *E*

Tipicamente é usado em testes condicionais.

```
1 se (condicao1 OU condicao2) {
2         //codigo a ser executado;
3 }
```

4.4.3 Resto

O operador **resto** retorna o resto da divisão cujo numerador e denominador são inteiros. Exemplo: 11 dividido por 3; Q = 3 e R = 2.

```
1 inteiro n = 11;
2 inteiro d = 3;
3 inteiro r = 0;
4 r = n resto d; //r recebe 2
```

5 Funções

Funções são formas de modularizar um código, evitando redundâncias de linhas de comandos sucessivos. Elas reúnem o bloco de código desejado "para fora"do ciclo de execução do programa. Assim, quando a lógica implementada pela função necessita ser executada, basta que se chame a função pelo seu nome, passando os atributos requeridos pela lógica implementada. Todo programa nessa linguagem deve estar contido dentro da função chamada **principal**. Como exceção, essa

função não possui tipo de retorno e não recebe nenhum parâmetro. As demais funções criadas pelo programador deve possuir um tipo de retorno e os atributos-parâmetros dessa função, se necessários.

Os tipos de retorno podem ser: *caracter*, *inteiro*, *real*, *VF* ou *vazio*. Isso significa que, após a execução da lógica, a função deve retornar um valor correspondente ao seu tipo de retorno ou, no caso de uma função *vazio*, não retorna nada.

Assim, toda função deve ser definida na seguinte ordem, com os seguintes elementos: **tipo de retorno**, **nome**, **parâmetros** (se necessário) e **escopo**. Ao final da lógica implementada, deve-se utilizar o comando **retorne** seguido do valor desejado a ser retornado. Lembre-se que para funções do tipo *vazio*, o comando não é utilizado.

```
1  inteiro quadrado (inteiro n) {
2          retorne n*n;
3  }
4
5  vazio imprimaCincoVezes (caracter c) {
6          inteiro i;
7          para (i = 0; i < 5; i++)
8          imprima(c);
9  }
```

5.1 Função Principal

Todo programa executado por esta linguagem deve ser descrito dentro de uma função **principal**.

```
1  principal () {
2          inteiro a;
3          inteiro b;
4      a = 55;
5
6      se (a < 55) {
7          b = a - 55;
8      } senao {
9          b = a * 2;
10     }
11 }
```

6 Palavras Reservadas

constante	VF	se	senao
ss	para	paracada	enquanto
escolha	caso	imprima	leia
lista	tabela	estrutura	E
OU	resto	inteiro	caractere
real	retorne	vazio	

Tabela 3: Tabela de palavras reservadas

7 Autômatos de Comandos

7.1 Se/Senão

Os autômatos dessas expressões são:

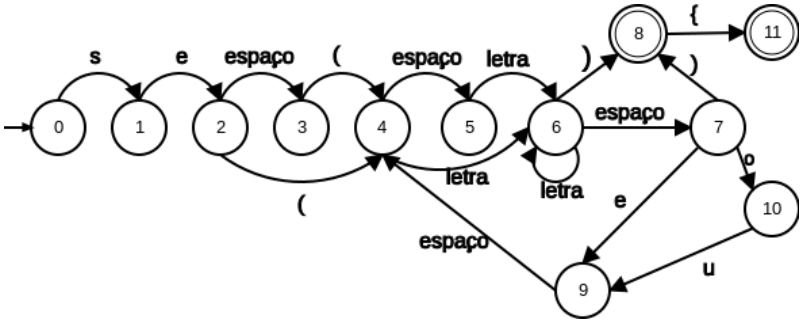

Figura 2: Autômato para comando *se*

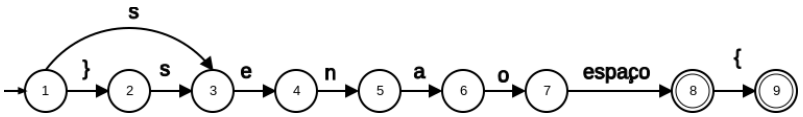

Figura 3: Autômato para comando *senao*

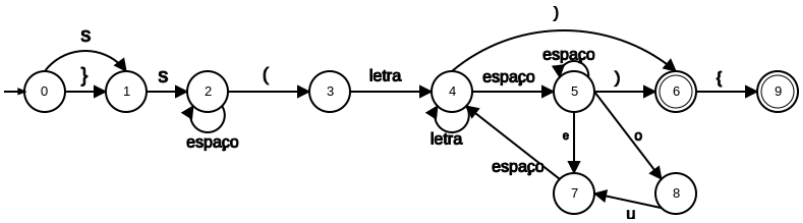

Figura 4: Autômato para comando *ss*

7.2 Para/Para cada

Os autômatos dessas expressões são:

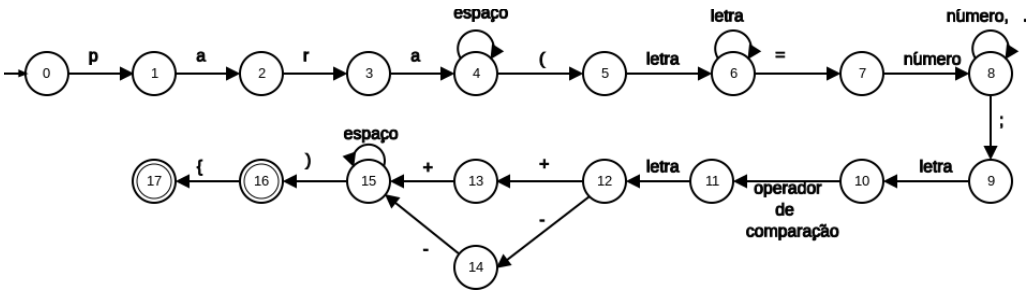

Figura 5: Autômato do comando *para*

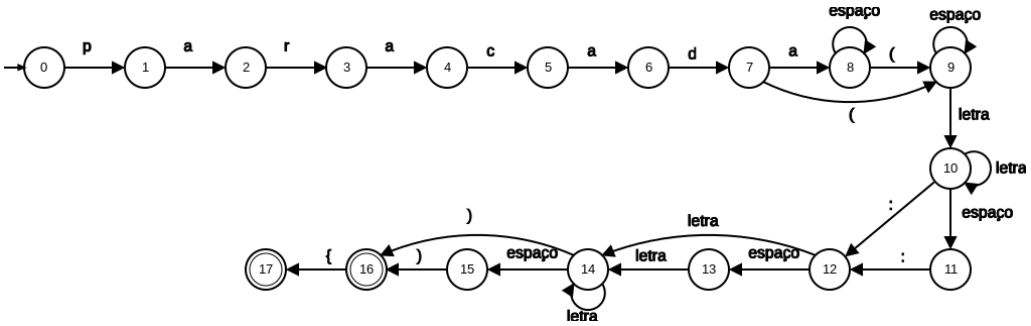

Figura 6: Autômato do comando *paracada*

7.3 Enquanto

O autômato da expressão *enquanto* é:

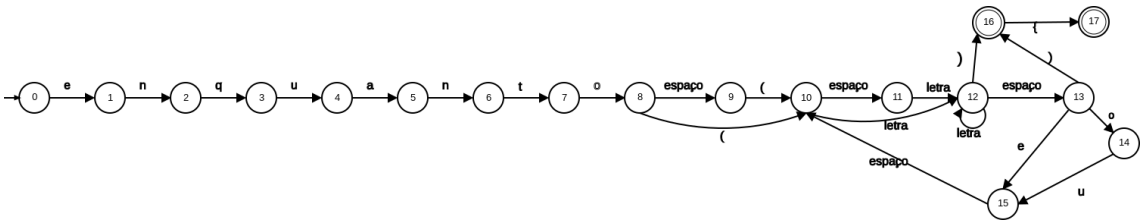

Figura 7: Autômato do comando *enquanto*

7.4 Escolha/Caso

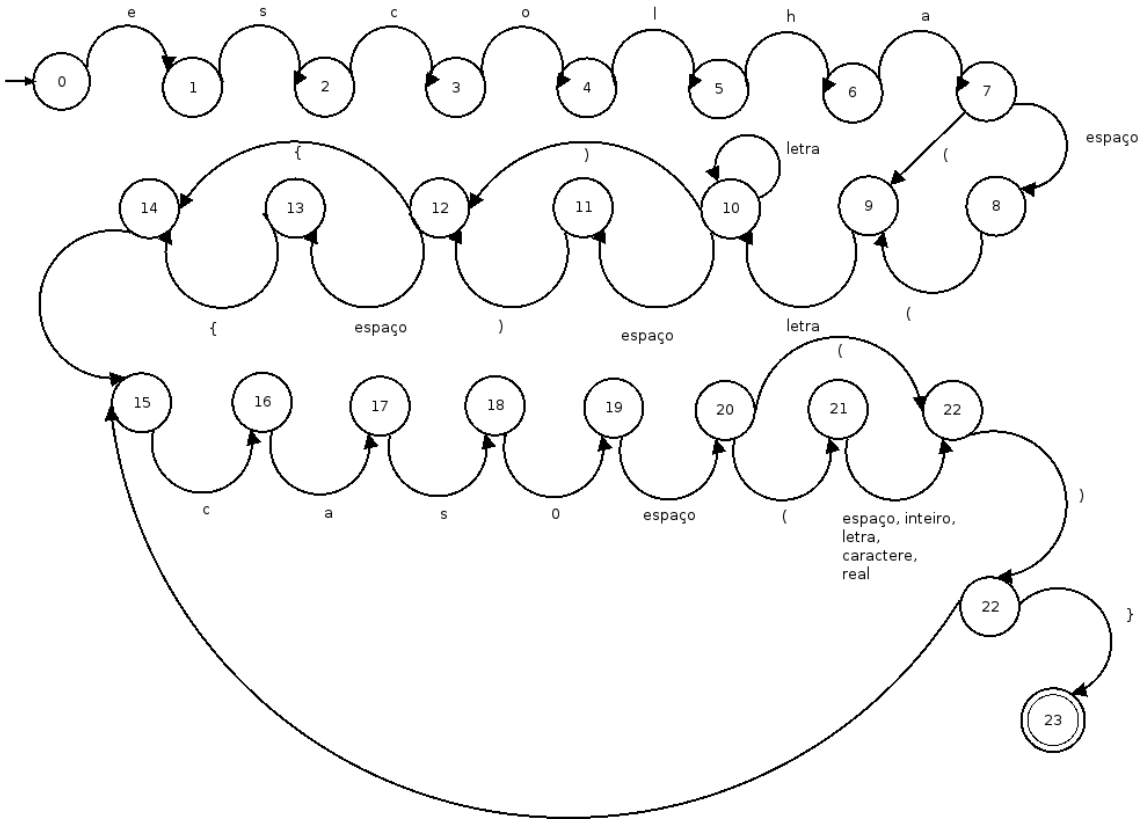

Figura 8: Autômato do comando *escolha/caso*

8 Gramática Livre de Contexto

8.1 Se/Senão

A estrutura da linguagem livre de contexto para esse comando é:

declaracao → *se (expressao)*
declaracao → *se (expressao) declaracao*
declaracao → *se (expressao) declaracao senao*
declaracao → *se (expressao) declaracao senao declaracao*
expressao → *expressao E expressao*
expressao → *expessao OU expressao*
expressao → *termo comparador termo*

comparador → > | < | >= | <=
termo → termo operador fator
operador → + | - | * | /
fator → *número* | *letra*

8.2 Para

A estrutura para esse comando é:

declaracao → para (letra = número; letra comparador número; letra = operadorsimples operadorsimples)
declaracao → para (letra = número; letra comparador número; letra = numero operador numero)
comparador → > | < | >= | <=
operadorsimples → + | -
operador → *operadorsimples* | * | /

Parte II
Implementação do Analisador Léxico e Sintático

9 Ferramentas Utilizadas

Para a implementação do analisador léxico, foi utilizada a ferramenta FLEX e para o analisador sintático, a ferramenta BISON. Devido à disponibilidade, boa documentação e facilidade de uso, essas ferramentas foram preferidas dentre as demais opções (Gold Parser, Regex, JavaCC, JFlex).

10 Particularidades da Implementação

10.1 Árvore Sintática Abstrata

Para o desenvolvimento dos analisadores léxico e sintático, utilizou-se uma estrutura chamada árvore sintática abstrata. É uma estrutura bastante similar à árvore de derivação, que possui um nó para cada regra de derivação. Ao escrever uma gramática, muitas regras existem somente para auxiliar a construção das regras e não possuem significado para o programa. É para isso que serve a árvore sintática abstrata, nela é possível omitir estas regras que não são relevantes para o significado sintático do programa.

Nesse tipo de árvore, existem vários tipos de nós para diferentes funcionalidades. Existem nós para estruturas condicionais, estruturas de repetição, todos tipos de operadores aritiméticos e condicionais, etc.

Uma vez que a árvore sintática abstrata é criada, é possível escrever funções que "caminham" pela árvore e resolvem as operações ou executam os comandos, de acordo com o tipo de nó em questão, e no final retorna um valor resultante.

10.2 Analisador Sintático

No código do analisador sintático, a variável yyval foi implementada como do tipo "union". Desta forma ela pode armazenar tipos de dados diferentes sobre cada token.

```
1 %union {
2     struct ast *a;
3     double d;
4     struct symbol *s;
5     struct symlist *sl;
```

```
6       int fn;
7       char *st;
8   }
```

- A varíavel `a` do tipo `struct ast*` aponta um nó da árvore sintática abstrata.
- A variável `d` do tipo `double` serve para armazenar os valores numéricos do programa.
- A variável `s` do tipo `struct symbol*` aponta para um símbolo e `struct symlist* sl` para uma lista de símbolos.
- A variável `fn` do tipo `int` armazena um número que corresponde a qual tipo de operador ou função embutida da linguagem foi reconhecido, como explicado nas seções 12.2 12.4.
- A variável `st` do tipo `char*` aponta para uma cadeias de caracteres.

Os tokens a serem utilizados e suas respectivas variáveis que representam seus valores foram declarados da seguinte forma:

```
1  %token <d> NUMBER
2  %token <s> NAME
3  %token <fn> FUNC
4  %token <st> STRING
5
6  %token IF ELSE WHILE DO LET PRINCIPAL FOR RESTO LEIA
```

A linha 6 declara tokens que não armazenam valores.

A precedência dos operadores foi definida da seguinte forma:

```
1  %nonassoc <fn> CMP
2  %right '='
3  %left '+' '-'
4  %left '*' '/'
5  %nonassoc '|' UMINUS
```

Os operadores de comparação (<, >, >=, <=, !=, ==) não são associativos e tem a precedência mais baixa.

Os operadores `'+'` e `'-'` são associativos à esquerda e com precedência menor do que os operadores `'*'` e `'/'` que também são associativos à esquerda. Isso significa que as contas de multiplicação e divisão serão realizadas primeiro e só depois as de adição e subtração serão realizadas.

Os operadores `'|'` (módulo) e UMINUS (menos unário) tem a maior precedência e não são associativos.

Os demais símbolos são declarados:

```
1  %type <a> exp stmt list explist function
2  %type <sl> symlist
3
4  %start principal
```

os simbolos `exp, stmt, list, explist, function` tem o seu valor associado a uma variável que armazena um nó da árvore sintática abstrata. O símbolo `principal` é o símbolo que começa a gramática.

10.3 Arquivo de Cabeçalho

Foi criado um arquivo de cabeçalho (.h) para definições de variáveis, enumerações, estruturas e funções utilizadas nos códigos do analisador léxico e sintático. Este arquivo de cabeçalho foi incluído no código do analisador léxico e sintático.

Algumas das componentes relevantes para o trabalho serão demonstradas a seguir.

Tabela de símbolos:

```
1  struct symbol {
2      char *name;
3      double value;
4      struct ast *func;
5      struct symlist *syms;
6  };
7
8  #define NHASH 9997
9  struct symbol symtab[NHASH];
```

A tabela de símbolos é um vetor da estrutura `symbol`. Essa estrutura contém uma variáveis que apontam para o nome do símbolo, um valor real, um nó da arvore sintática e uma lista de símbolos.

Lista de símbolos:

```
1  struct symlist {
2      struct symbol *sym;
3      struct symlist *next;
4  };
```

Esta lista de símbolos será útil mais tarde para a declaração de funções (seção 13).

Enumeração para representar as funções embutidas:

```
1  enum bifs {
2      B_sqrt = 1,
3      B_exp,
4      B_log,
5      B_print
6  };
```

Árvore sintática abstrata:

```
1  struct ast {
2      int nodetype;
3      struct ast *l;
4      struct ast *r;
5  };
```

10.4 Funções Auxiliares Utilizadas

Foram criadas diversas funções auxiliares para realizar tarefas durante a fase de análise léxica e sintática. Algumas das mais relevantes tiveram a sua declaração especificada e logo abaixo uma explicação da função:

`struct symbol* lookup(char*)`
Retorna um endereço na tabela de símbolos a partir do nome de um símbolo. Caso o símbolo não exista na tabela, uma nova entrada é criada para ele. Para determintar a posição no vetor, ela utiliza a função `symhash(char*)` que retorna um número a partir de uma string. Caso a posição da tabela já esteja ocupada por um símbolo, a função escaneia linearmente até achar uma posição livre. Caso a tabela esteja cheia a função é abortada e uma condição de erro é retornada.

`double eval(struct ast*)`
Percorre a árvore, avalia a expressão, retornando o valor resultante dela.

`void treefree(struct ast*)`
Deleta uma árvore sintática abstrata e libera a memória por ela ocupada.

`void dodef(struct symbol*, struct symlist*, struct ast*)`
Define uma função.

Todas as funções abaixo criam um nó da árvore sintática abstrata. Cada função serve para criar um tipo de nó diferente:

`struct ast* newast(int nodetype, struct ast *l, struct ast *r)`
Cria um nó "genérico"

`struct ast *newcmp(int cmptype, struct ast *l, struct ast *r)`
Cria um nó que representa uma operação de comparação.

`struct ast *newfunc(int functype, struct ast *l)`
Cria um nó de função.

`struct ast *newcall(struct symbol *s, struct ast *l)`
Cria um nó de chamada de função.

`struct ast *newref(struct symbol *s)`
Cria um nó que referencia um símbolo.

`struct ast *newasgn(struct symbol *s, struct ast *v)`
Cria um nó da operação de atribuição.

`struct ast *newasgnname(struct symbol *s)`
Cria um nó que será usado para ler um valor de entrada.

`struct ast *newnum(double)`
Cria um nó de número.

`struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr)`
Cria um nó de uma estrutura condicional.

`struct ast *newflowfor(int nodetype, struct ast *ctrl, struct ast *cond, struct ast *range, struct ast *todo)`
Cria um nó de um loop do tipo "for".

`struct ast *newstring(char *s)`
Cria um nó do tipo "string".

11 Diferenças da Linguagem Proposta e Implementada

Alguns aspectos da linguagem UPA foram modificados durante a etapa da implementação e outros não foram implementados principalmente devido a dificuldades enfrentadas neste processo:

- A função `imprima` foi mudada para `escreva` porém a funcionalidade permanece a mesma
- As estruturas `ss, paracada, escolha/caso, enumerado, lista, tabela, estrutura` não foram implementadas

12 Reconhecimento de Tokens

12.1 Operadores de Único Caracter

Todos os operadores de um único caracter foram agrupados em só uma regra. A ação atribuida a essa regra simplesmente retorna o primeiro caracter da variável yytext, que corresponde ao caracter reconhecido. Para facilitar a implementação, os caracteres separadores (Ponto e vírgula, parênteses e colchetes) também foram incluídos na regra.

```
1 "+"  |
2 "-"  |
3 "*"  |
4 "/"  |
5 "="  |
6 "|"  |
```

```
7  ","  |
8  ";"  |
9  "("  |
10 "{"  |
11 "}"  |
12 ")"  |
13 "%"    { return yytext[0]; }
```

12.2 Operadores de Comparação e Lógicos

Cada um dos operadores de comparação é reconhecido através de sua própria regra. O token retornado para todos operadores de comparação é o token CMP. O que diferencia cada um dos operadores é o valor da variável `yyval.fn`, que assume um valor diferente para cada um dos operadores, de acordo com o código. Para facilitar a implementação, os operadores lógicos E e OU foram reconhecidos através deste mesmo trecho de código.

```
1 ">"      { yylval.fn = 1; return CMP; }
2 "<"      { yylval.fn = 2; return CMP; }
3 "!="     { yylval.fn = 3; return CMP; }
4 "=="     { yylval.fn = 4; return CMP; }
5 ">="     { yylval.fn = 5; return CMP; }
6 "<="     { yylval.fn = 6; return CMP; }
7 "E"      { yylval.fn = 7; return CMP; }
8 "OU"     { yylval.fn = 8; return CMP; }
```

12.3 Palavras Reservadas

O código para o reconhecimento das palavras reservadas da linguagem simplesmente retorna um token correspondente para cada uma das palavras.

```
1 "se"         { return IF; }
2 "senao"      { return ELSE; }
3 "enquanto"   { return WHILE; }
4 "funcao"     { return LET; }
5 "principal"  { return PRINCIPAL; }
6 "para"       { return FOR; }
7 "resto"      { return RESTO; }
8 "leia"       { return LEIA; }
```

12.4 Funções Embutidas na Linguagem

As funções embutidas na linguagem são as funções `escreva` e `leia`. A função `escreva`, como o próprio nome já diz, escreve na tela. A variável `yyval.fn` recebe um valor que identifica a própria função. O token FUNC é retornado ao reconhecer a função escreva. O reconhecimento da função `leia` é feito através da palavra reservada "leia" como mostrado na seção 12.3. O token retornado é o LEIA.

```
1 "escreva" { yylval.fn = B_print; return FUNC; }
```

12.5 Reconhecimento de Nomes e Strings

Ao reconhecer uma cadeia em que começa com letras maiúsculas ou minúsculas, seguidas de letras ou números, o analisador sabe que se trata de um nome, que representa um identificador de variáveis, funções, etc. O token NAME é retornado. A função auxiliar `lookup` é chamada, onde o parâmetro é a própria cadeia identificadora. A função devolve uma estrutura do tipo `symbol` já preenchida e ela é armazenada na variável `yylval.s`.

A expressão regular que reconhece strings, aceita cadeias que começam e terminam com aspas duplas (‘‘) e podem conter qualquer tipo de caractere entre as aspas menos o próprio ‘‘. A variável st recebe o valor da string digitada incluindo as aspas e o token STRING é retornado.

```
1 [a-zA-Z][a-zA-Z0-9]*   { yylval.s = lookup(yytext); return NAME; }
2 ["][^"]*["] { yylval.st = yytext; return STRING;}
```

12.6 Reconhecimento de Números

```
1 EXP       ([Ee][-+]?[0-9]+)
2 [0-9]+"."[0-9]*{EXP}? |
3 "."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }
```

A linha 1 do código acima é uma definição que foi colocada na primeira seção do programa LEX, e serve para reconhecer a parte do expoente de um número ponto flutuante. Esta parte deve obrigatoriamente começar com a letra 'e' (maiúscula ou minúscula), seguida de um sinal opcional e seguida de um número inteiro.

As linhas 2 e 3 se referem a expressão regular para reconhecer o tipo número. A expressão pode ser quebrada em duas partes. As duas partes estão conectadas pelo operador `'|'` (OU). Se uma das partes for reconhecida, isso já é condição suficiente para a expressão regular aceitar a entrada.

A primeira parte, que se localiza na linha 2, aceita qualquer cadeia que comece com uma sequência de 1 ou mais dígitos (de 0 a 9), seguida de um ponto, depois por mais uma sequência de 0 ou mais digitos, e finalmente uma parte exponencial opcional.

A segunda parte aceita cadeias que podem opcionalmente começar com um ponto, uma sequência de 1 ou mais dígitos e a parte exponencial opcional.

Caso a cadeia seja aceita pela expressão regular, o valor é armazenado na variável `yylval.d` que armazena o valor entrado em ponto flutuante e o token NUMBER é retornado.

12.7 Reconhecimento de Comentários e Espaços em Branco

```
1 "//".*
2 [/][*][^*]*[*]+([^*/][^*]*[*]+)*[/] { }
3 [ \t]
4 \\\n
5 "\n"
6 .       { yyerror("Mystery character %c\n", *yytext); }
```

As linhas 1 a 5 do código acima representam expressões que serão ignoradas pelo compilador, por isso não há nenhuma ação associada a elas.

A linha 1 contém uma expressão que reconhece um comentário de uma linha. A cadeia correspondente deve começar obrigatoriamente por `'//'` e em seguida pode ter qualquer tipo de caractere.

A linha 2 contém uma expressão que reconhece comentários de múltiplas linhas. Obrigatoriamente o comentário deve começar com `'\*'` e terminar com `'*\'`, tudo o que tiver dentro será ignorado.

A expressão da linha 3 ignora espaços em branco e caracteres de tabulação e as linhas 4 e 5 ignoram o caractere `'\n'` que é a quebra de linha.

A última linha reconhece qualquer caractere que não foi reconhecido, e retorna uma condição de erro.

13 Regras Utilizadas

A primeira regra reconhece as estruturas condicionais e de repetição. As funções auxiliares `newflow` e `newflowfor` criam os nós respectivos para cada tipo de estrutura. A variável que armazena o valor do símbolo `stmt` recebe o nó criado pela função auxiliar.

```
1  stmt: IF '(' exp ')' '{' list '}'
2  { $$ = newflow('I', $3, $6, NULL); }
3
4  | IF '(' exp ')' '{' list '}' ELSE '{' list '}'
5  { $$ = newflow('I', $3, $6, $10);  }
6
7  | WHILE '(' exp ')' '{' list '}'
8  { $$ = newflow('W', $3, $6, NULL); }
9
10 | FOR '(' exp ';' exp ';' exp ')' '{' list '}'
11 { $$ = newflowfor('Y', $3, $5, $7, $10); }
12 ;
```

A segunda regra reconhece "listas" de declarações. A primeira linha da regra diz que o símbolo `list` pode ser substituído por um símbolo vazio, neste caso o valor recebido é nulo. A segunda linha da regra diz que um símbolo `list` pode ser substituído por um símbolo `stmt` (Que é uma estrutura condicional ou de repetição) seguido por um outro símbolo do tipo `list`. A terceira linha permite o símbolo `list` ser substituído por um símbolo `exp` (Que representa uma expressão) seguido de um ponto e vírgula e por último um símbolo `list`.

A combinação da primeira linha, onde o símbolo `list` pode ser substituído com um símbolo vazio, com a segunda e terceira linhas, onde sempre há um símbolo `list` no final, permite criar uma estrutura recursiva. Graças a essa recursividade, é possível reconhecer várias estruturas condicionais, de repetição e expressões terminadas por ponto e vírgula seguidas umas das outras.

A ação realizada na segunda e terceira linha da regra é a criação de um novo nó para a árvore sintática abstrata.

```
1  list: { $$ = NULL; }
2  | stmt list { if ($2 == NULL)
3                 $$ = $1;
4               else
5                 $$ = newast('L', $1, $2);
6             }
7  | exp ';' list { if ($3 == NULL)
8                   $$ = $1;
9                 else
10                  $$ = newast('L', $1, $3);
11              }
12 ;
```

A terceira regra reconhece expressões. As 7 primeiras linhas reconhecem operações com expressões, onde um novo nó da árvore é criado. O tipo do nó é o operador em questão e os ramos são os operandos (outras expressões). Como a precedência dos operadores foi definida anteriormente, não há problemas de ambiguidade na gramática. Foi utilizada a recursividade nas regras, de forma a garantir que expressões com várias operações sejam reconhecidas. A oitava linha da regra garante que toda expressão que está entre parênteses seja avaliada primeiro e a nona linha faz com que o menos unário seja a operação de mais alta precedência.

Depois a décima linha permite que o símbolo `exp` seja substituído por um token do tipo NUMBER e um nó do tipo número é criado na árvore. As demais linhas permite que expressões sejam atribuições e chamadas de funções, incluindo a função embutida `leia` (seção 12.4). É importante ressaltar também que o símbolo `exp` pode ser substituído por um token do tipo NOME, permitindo assim criar expressões onde se fazem operações com variáveis criadas pelo usuário.

```
1  exp: exp CMP exp          { $$ = newcmp($2, $1, $3); }
2  | exp '+' exp             { $$ = newast('+', $1,$3); }
3  | exp '-' exp             { $$ = newast('-', $1,$3); }
4  | exp '*' exp             { $$ = newast('*', $1,$3); }
5  | exp '/' exp             { $$ = newast('/', $1,$3); }
6  | exp RESTO exp           { $$ = newast('%', $1,$3); }
7  | '|' exp                 { $$ = newast('|', $2, NULL); }
8  | '(' exp ')'             { $$ = $2; }
9  | '-' exp %prec UMINUS    { $$ = newast('M', $2, NULL); }
10 | NUMBER                  { $$ = newnum($1); }
11 | FUNC '(' STRING  ')'    { $$ = newstring($3); }
12 | FUNC '(' explist ')'    { $$ = newfunc($1, $3); }
13 | NAME                    { $$ = newref($1); }
14 | NAME '=' exp            { $$ = newasgn($1, $3); }
15 | NAME '(' explist ')'    { $$ = newcall($1, $3); }
16 | LEIA '(' NAME ')'       { $$ = newasgnname($3); }
17 ;
```

A quarta e quinta regra criam uma lista de expressões e uma lista de símbolos respectivamente. cada elemento desta lista é separado por uma vírgula (,). A função apropriada para criar o nó da árvore é chamada. O símbolo `explist` apareceu em duas linhas da regra anterior e ele representa uma lista de expressões que são os parâmetros para a chamada da função. Já para a declaração de uma função, uma lista de símbolos é necessária, como será explicado na próxima regra.

```
1  explist: exp
2  | exp ',' explist  { $$ = newast('L', $1, $3); }
3  ;
4  symlist: NAME       { $$ = newsymlist($1, NULL); }
```

```
5 | NAME ',' symlist { $$ = newsymlist($1, $3); }
6 ;
```

A sexta regra trata de uma declaração de função. Essa regra é recursiva a esquerda e a ação dela é chamar uma função auxiliar que irá criar a definição da função a partir do valor do token NAME (Que irá identificar a função), do valor do símbolo `symlist` (lista de símbolos que armazenarão os parâmetros) e do valor do símbolo `list` (uma lista de declarações, que são as ações a serem realizadas pela função).

```
1 function:
2 | function LET NAME '(' symlist ')' '{' list '}'
3 {dodef($3, $5, $8);}
4 ;
```

Finalmente a sétima e última regra executa a função principal do programa. Ela avalia a árvore sintática abstrata que está contida no símbolo `list` (lista de declarações) desta forma executando cada um dos comandos do programa e por final deleta a árvore, liberando o espaço de memória por ela ocupado.

```
1  principal:
2  | function PRINCIPAL '(' ')' '{' list '}' {
3    if($1 == NULL) {
4      if(debug)
5        dumpast($6, 0);
6      eval($6);
7      treefree($6);
8    } else {
9      if(debug)
10       dumpast($6, 0);
11     eval($6);
12     treefree($6);
13   }
14 }
15 ;
```

14 Exemplo de Árvore de Derivação

Será exemplificada a árvore de derivação do seguinte código:

```
1 principal(){
2   escreva((5-4)*8+4/2);
3 }
```

Ao passar pelo analisador léxico, a sequência de tokens retornadas é a seguinte: `PRINCIPAL '(' ')' '{' FUNC '(' '(' NUMBER '-' NUMBER ')' '*' NUMBER '+' NUMBER '/' NUMBER ')' ';' '}'`.

A derivação a partir das regras gramaticais da linguagem é a seguinte:

```
PRINCIPAL ⇒ PRINCIPAL '(' ')' '{' list '}'
          ⇒ PRINCIPAL '(' ')' '{' exp ';' list '}'
          ⇒ PRINCIPAL '(' ')' '{' exp ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' explist ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' exp '+' exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' exp '+' exp '/' exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' exp '*' exp '+' exp '/' exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' '(' exp ')' '*' exp '+' exp '/'
            exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' '(' exp '-' exp ')' '*' exp '+' exp
            '/' exp ')' ';' '}'
          ⇒ PRINCIPAL '(' ')' '{' FUNC '(' '(' exp '-' exp ')' '*' exp '+' exp '/'
            NUMBER ')' ';' '}'
```

```
⇒ PRINCIPAL ‘(’ ‘)’ ‘{’ FUNC ‘(’ ‘(’ exp ‘-’ exp ‘)’ ‘*’ exp ‘+’
  NUMBER ‘/’ NUMBER ‘)’ ‘;’ ‘}’
⇒ PRINCIPAL ‘(’ ‘)’ ‘{’ FUNC ‘(’ ‘(’ exp ‘-’ exp ‘)’ ‘*’
  NUMBER ‘+’ NUMBER ‘/’ NUMBER ‘)’ ‘;’ ‘}’
⇒ PRINCIPAL ‘(’ ‘)’ ‘{’ FUNC ‘(’ ‘(’ exp ‘-’ NUMBER ‘)’ *
  NUMBER + NUMBER ‘/’ NUMBER ‘)’ ‘;’ ‘}’
⇒ PRINCIPAL ‘(’ ‘)’ ‘{’ FUNC ‘(’ ‘(’ NUMBER ‘-’ NUMBER ‘)’ ‘*’
  NUMBER ‘+’ NUMBER ‘/’ NUMBER ‘)’ ‘;’ ‘}’
```

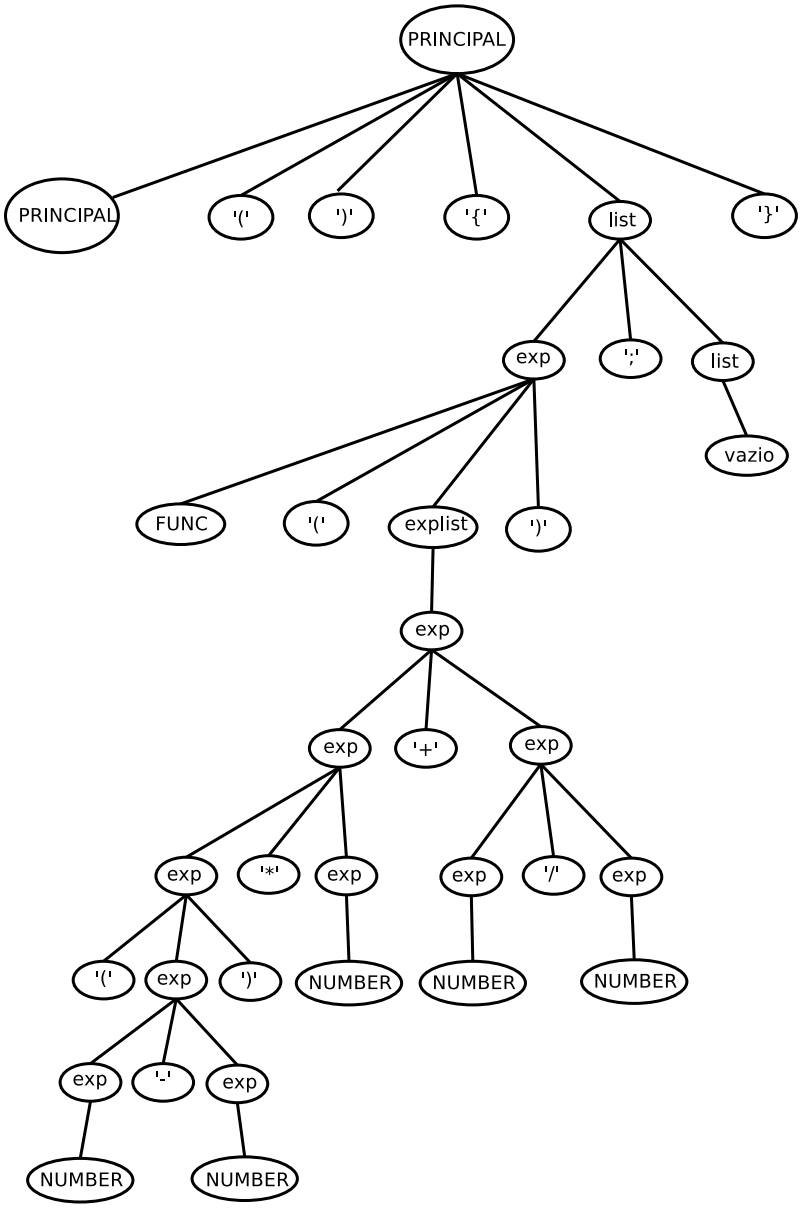

Figura 9: Árvore de Derivação

Referências

[1] John Levine. *Flex & Bison.* O'Reilly, 2009.