|  |  |
| --- | --- |
|  | ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO  Departamento de Engenharia de Computação e Sistemas Digitais |

**Compilador Hopper**

**PCS2056 – Linguagens e Compiladores**

Nathalia Sautchuk Patrício 5432596

Bruno Umeda Grisi 5438011

Índice

[1. Introdução 4](#_Toc247172979)

[2. Definição da Linguagem 5](#_Toc247172980)

[2.1 Recursos da Linguagem 5](#_Toc247172981)

[2.1.1 Estrutura do programa 5](#_Toc247172982)

[2.1.2 Variáveis simples 5](#_Toc247172983)

[2.1.3 Variáveis indexadas – vetor e matriz 5](#_Toc247172984)

[2.1.4 Comandos de declaração de variáveis 5](#_Toc247172985)

[2.1.5 Comandos de atribuição 6](#_Toc247172986)

[2.1.6 Comandos de entrada 6](#_Toc247172987)

[2.1.7 Comandos de saída 6](#_Toc247172988)

[2.1.8 Comandos condicionais 6](#_Toc247172989)

[2.1.9 Comandos iterativos 6](#_Toc247172990)

[2.1.10 Expressões aritméticas 6](#_Toc247172991)

[2.1.11 Expressões booleanas 7](#_Toc247172992)

[2.2 Notação BNF 7](#_Toc247172993)

[2.3 Notação Wirth 9](#_Toc247172994)

[3. Leitor de Máquina de Estados 11](#_Toc247172995)

[4. Análise Léxica 12](#_Toc247172996)

[5. Análise Sintática 15](#_Toc247172997)

[5.1 Submáquina Atribuição 15](#_Toc247172998)

[5.2 Submáquina Condição 15](#_Toc247172999)

[5.3 Submáquina Condicional 16](#_Toc247173000)

[5.4 Submáquina Declaração 17](#_Toc247173001)

[5.5 Submáquina Entrada 17](#_Toc247173002)

[5.6 Submáquina Exp\_Booleana 18](#_Toc247173003)

[5.7 Submáquina Expressão 18](#_Toc247173004)

[5.8 Submáquina Fator 19](#_Toc247173005)

[5.9 Submáquina Função 19](#_Toc247173006)

[5.10 Submáquina Identificador 20](#_Toc247173007)

[5.11 Submáquina Iteração 21](#_Toc247173008)

[5.12 Submáquina Programa 22](#_Toc247173009)

[5.13 Submáquina Saída 23](#_Toc247173010)

[5.14 Submáquina Termo 23](#_Toc247173011)

[6. Análise Semântica 24](#_Toc247173012)

[7. Bibliografia 25](#_Toc247173013)

# Introdução

Neste projeto tem-se por objetivo modelar e construir um compilador batizado de Hopper em homenagem a Grace Hopper, considerada a primeira pessoa a construir um compilador.

O projeto está dividido em quatro partes principais:

1. **Definição da linguagem:** para que se possa executar um programa, este deve ser escrito em uma linguagem compreensível ao compilador. Portanto, nesta fase, será definida uma gramática e quais serão os comandos aceitos e suas respectivas funções. A linguagem foi batizada com o nome de Grace.
2. **Analisador Léxico:** definida a linguagem, o analisador léxico é a parte responsável por receber os caracteres do arquivo fonte e agrupá-los em pequenos grupos (tokens).
3. **Analisador Sintático:** este módulo é responsável por verificar se o código analisado está gramaticalmente correto.
4. **Analisador Semântico:** este módulo tem como funções principais analisar restrições quanto à utilização dos identificadores , verificar a compatibilidade de tipos, efetuar a tradução do programa e gerar o código-objeto.

**Obs.:** Por se tratar de uma *Gramática Livre de Contexto*, a análise sintática não é suficiente para validar o código escrito na linguagem proposta. Portanto, as ações semânticas se fazem necessárias.

O produto gerado pelo compilador é um arquivo, com extensão **.mvn**, com o código MVN correspondente ao código de entrada, escrito na linguagem especificada na seção seguinte.

# Definição da Linguagem

A linguagem **Horae** elaborada possui as especificações determinadas em aula. Assim, os principais componentes da mesma são:

* estrutura do programa
* declaração de variáveis
* variáveis simples dos tipos *caracter*, *booleano*, *inteiro* e *ponto flutuante*
* variáveis indexadas – vetor e matriz
* comandos de atribuição
* comandos de entrada
* comandos de saída
* comandos condicionais
* comandos iterativos
* expressões aritméticas
* expressões booleanas

A seguir, faremos uma descrição funcional da linguagem elencando e detalhando os seus principais recursos e restrições de acordo com o que foi projetado.

## Recursos da Linguagem

### Estrutura do programa

O programa se inicia com a palavra chave *program* e termina com a palavra chave *end*. Todas as declarações, de variáveis ou funções, e comandos estarão localizados entre o *program* e o *end*.

### Variáveis simples

A nossa linguagem aceita quatro tipos diferentes de variáveis: inteiros (int), booleano (boolean), ponto flutuante (float) e caracter (string).

As operações aritméticas possíveis com as variáveis do tipo inteiro são: adição, subtração, multiplicação, divisão e módulo (%). Vale-se ressaltar que a precedência dos operadores de multiplicação, divisão e módulo (%) sobre os demais operadores é respeitada.

### Variáveis indexadas – vetor e matriz

As duas estruturas projetadas são vetores e matrizes. O objetivo é que se crie uma forma de acesso de leitura e escrita aos dados contidos nestas estruturas. Neste caso, a quantidade de memória alocada para cada estrutura deve ser especificada em sua declaração. Por exemplo:

* Vetor com 5 números 🡪 int[5];
* Matriz 2x2 de números 🡪 int[2][2];

### Comandos de declaração de variáveis

A declaração da variável é feita especificando o tipo da variável, seguida de uma cadeia de caracteres que será seu identificador e o caracter “;” apontando o final da declaração. Não é possível a declaração de variáveis distintas com o mesmo identificador.

### Comandos de atribuição

São comandos utilizados para a alteração do valor de uma variável, seja este vindo de uma outra variável ou através do resultado de expressões. Na linguagem definida o comando de atribuição ocorre através de um identificador da variável que receberá o valor, seguido de um sinal de “=”, a expressão que irá definir o novo valor da variável destino, seguido do caracter “;” para indicar o fim do comando.

### Comandos de entrada

O comando de entrada implementado aqui é o input que lê um byte e guarda esse valor no endereço fornecido como argumento do comando.

### Comandos de saída

O comando de saída aqui projetado é o output que imprime na saída um valor ou um resultado armazenado em uma variável (local de memória) acessado pelo programa.

### Comandos condicionais

O comando condicional projetado é do tipo if-then-else-endif. Ele testa uma condição, e caso a mesma seja verdadeira, executa o bloco de comandos definidos entre as palavras chaves then e else. Caso contrário, executará o bloco de comandos entre as palavras chaves else e endif. A forma if-then-endif também é permitida.

### Comandos iterativos

Este tipo de comando permite ao usuário executar um bloco de comandos repetidamente enquanto uma condição testada for verdadeira. Na linguagem Grace existem dois comandos iterativos, while e for:

* o while executará o bloco de comandos contidos entre as palavras chaves do e endwhile enquanto a condição testada for verdadeira e
* o for executará o bloco de comandos contidos entre as palavras chaves beginfor e endfor enquanto a condição testada for verdadeira.

### Expressões aritméticas

A linguagem possui ainda suporte a expressões aritméticas, que podem ser realizadas em atribuições e condições. Pode-se utilizar parênteses para mudar a precedência dos operadores. Os operadores são +, -, \*, / e %.

### Expressões booleanas

As expressões booleanas são utilizadas na condição dos comandos de iteração e de condição, para determinar a ação a ser executada. Contém os operadores and, or, xor, not, ==, <>, <, >, <= e >=. Pode-se utilizar parênteses para mudar a precedência dos operadores.

## Notação BNF

Abaixo temos a linguagem em notação BNF:

<programa> ::= program <funções> <comandos> end

<comandos> ::= <comando>; | <comandos> <comando>; | ε

<comando> ::= <declaração> | <atribuição> | <entrada> | <saída> | <iteração>

| <condicional>

<declaração> ::= <tipo> <identificador>; | <tipo> <vetor>; | <tipo> <matriz>;

<tipo> ::= int | float | string | boolean

<identificador> ::= <letra> | <identificador><letra> | <identificador><digitos>

<vetor> ::= <identificador> [<digitos>]

<matriz> ::= <identificador> [<digitos>][<digitos>]

<letra> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q

| R | S | T | U | V | Y | X | W | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | y | x | w | z

<digitos> ::= <digito> | <digitos><digito>

| <digitos><digito>.<digitos><digito>

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

<atribuição> ::= <identificador> = <expressão> | <vetor> = <expressão>

| <matriz> = <expressão>

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

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

<fator> ::= <identificador> | <vetor> | <booleano> | <chamada função>

| ( <expressão> ) | - <expressão> | <matriz> | <digitos>

<booleano> ::= true | false

<condição> ::= <expressão> <op booleano> <expressão> | <exp booleana>

| not <exp booleana>

<op booleano> ::= < | > | >= | <= | == | <>

<exp booleana> ::= <condição> <op lógico> <condição> | <booleano> |(<condição>)

<op lógico> ::= and | or | xor | not

<entrada> ::= input <vetor> | input <identificador> | input <matriz>

<saída> ::= output <expressão> | output <cadeia>

<cadeia> ::= '<caracteres>'

<caracteres> :: = <caracter> | <caracteres><caracter>

<caracter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | Y | X | W | Z| | \_ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | - | \* | / | = | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | y | x | w | z

<iteração> :: = while (<condição>) do <comandos> endwhile

| for (<atribuição>|<declaração>;<condição>;<atribuição>) beginfor <comandos> endfor

<condicional> ::= if (<condição>) then <comandos> endif

| if (<condição>) then <comandos> else <comandos> endif

<funções> ::= <função> | <funções><função> | ε

<função> ::= function <tipo> <identificador>(<declara parâmetros>) beginfunction <comandos> <retorno função> endfunction

<declara parâmetros> ::= <declaração> | <declara parâmetros>, <declaração>| ε

<chamada função> ::= <identificador>(<parâmetros>);

<parâmetros> ::= <expressão> | <parâmetros>, <expressão> | ε

<retorno função> ::= return <expressão>;

## Notação Wirth

A partir da notação BNF, criamos a descrição em notação de Wirth abaixo:

programa = "program" {função} {comando} "end".

comando = (declaração ";" |atribuição ";" | entrada ";" | saida ";"

| iteração | condicional ).

declaração = tipo (identificador | vetor | matriz).

tipo = ("int" | "float" | "string" | "boolean").

identificador = letra {letra | digito}.

vetor = identificador "[" digito {digito} "]".

matriz = identificador "[" digito {digito} "][" digito {digito} "]".

letra= ("A" | ... | "Z" | "a" | ... | "z") .

digitos= digito {digito} ["." digito {digito}].

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

atribuição = ( identificador | vetor | matriz ) "=" expressão.

expressão = termo {("+"|"-") termo}.

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

fator = ( identificador | vetor | matriz | booleano | chamada\_função

| "(" expressão ")" | "-" expressão | digitos).

booleano = ("true" | "false").

condição = (expressão op\_booleano expressão) | ["not"] exp\_booleana .

op\_booleano = "<" | ">" | "==" | "<>" | ">=" | "<=".

exp\_booleana = (condição op\_lógico condição) | booleano

| "(" condição ")" .

op\_lógico = "and" | "or" | "xor" | "not" .

entrada = "input" (vetor | identificador | matriz ).

saida = "output" (expressão | cadeia ).

cadeia = "'" caracter {caracter} "'".

caracter = ("A" | ... | "Z" | "0" | ... | "9" | " " | "+" | "-" | "\_" | "/" | "\*" | "=" | "a" | ... | "z" ) .

iteração = "while" "(" condição ")" "do" { comando } "endwhile"

| "for" "(" (atribuição | declaração) condição atribuição ")" "beginfor" {comando} "endfor".

condicional = "if" "(" condição ")" "then" { comando } [ "else" { comando }] "endif".

função = "function" tipo identificador "(" [declaracao { "," declaracao}] ")" "beginfunction" {comando} "return" expressão ";" "endfunction".

chamada\_função = identificador "(" [expressão { "," expressão}] ")".

# Leitor de Máquina de Estados

Com o objetivo de facilitar a implementação do compilador, foi implementado um leitor de máquina de estados em linguagem Java.

Foi feita uma classe que representa uma transição da máquina, que possui os seguintes atributos:

* estadoAtual: é um inteiro que representa o estado em que a máquina está no momento;
* proximoEstado: é um inteiro que representa para qual estado a máquina deve transitar;
* simbolo: é uma string que representa o símbolo que faz a máquina transitar do estadoAtual para o proximoEstado;
* acao: é uma string que representa a ação a ser tomada quando ocorre a transição;

Também existe uma classe que executa a máquina de estados, transitando os estados. Essa classe possui como atributos:

* tabelaTransicoes: é um ArrayList de transições da máquina que se quer executar;
* tabelaEstadosAceitacao: é uma ArrayList que contêm os estados de aceitação da máquina de estados;
* estadoAtual: é um inteiro que representa o estado em que a máquina está no momento;
* estadoInicial: é um inteiro que representa o estado inicial da máquina;
* nome: é uma string que guarda o nome da máquina e
* reset: é um booleano usado para dizer se a máquina deve ser resetada (ir para o estado inicial) após chegar em um estado de aceitação.

Essa classe possui o método “transita” que recebe como parâmetro uma string que representa o valor lido (nesse caso de um arquivo). Essa função pega o valor lido e, como sabe em qual estado a máquina está, compara com os símbolos que a máquina pode receber nesse estado. Achando o resultado, faz a transição para o próximo estado descrito na tabela. Caso esse estado seja um estado de aceitação, o estadoAtual da máquina é setada para o estadoInicial da máquina. No caso de não ser o estadoInicial passa a ser o estado para o qual a máquina fez a transição. O método devolve a ação a ser executada.

# Análise Léxica

O componente do compilador responsável pela análise léxica é o analisador léxico. A sua função principal é subdividir o código-fonte a ser compilado em elementos identificáveis, denominados tokens, que serão consumidos pelo analisador sintático mais tarde. Pode-se resumir as suas funções às seguintes:

1. Extração e classificação de tokens;
2. Tratamento de identificadores;
3. Identificação de palavras reservadas.

Uma vez implementado um leitor de máquina de estados, a idéia foi descrever o analisador léxico em uma máquina de estados e executá-la. A máquina está representada pela Figura 1.

O asterisco (\*), visto na figura, significa que deve-se retroceder um caracter.

O “delimit” pode ser um símbolo de branco, um símbolo de tab ou símbolo de pular linha, ou seja, caracteres de controle.

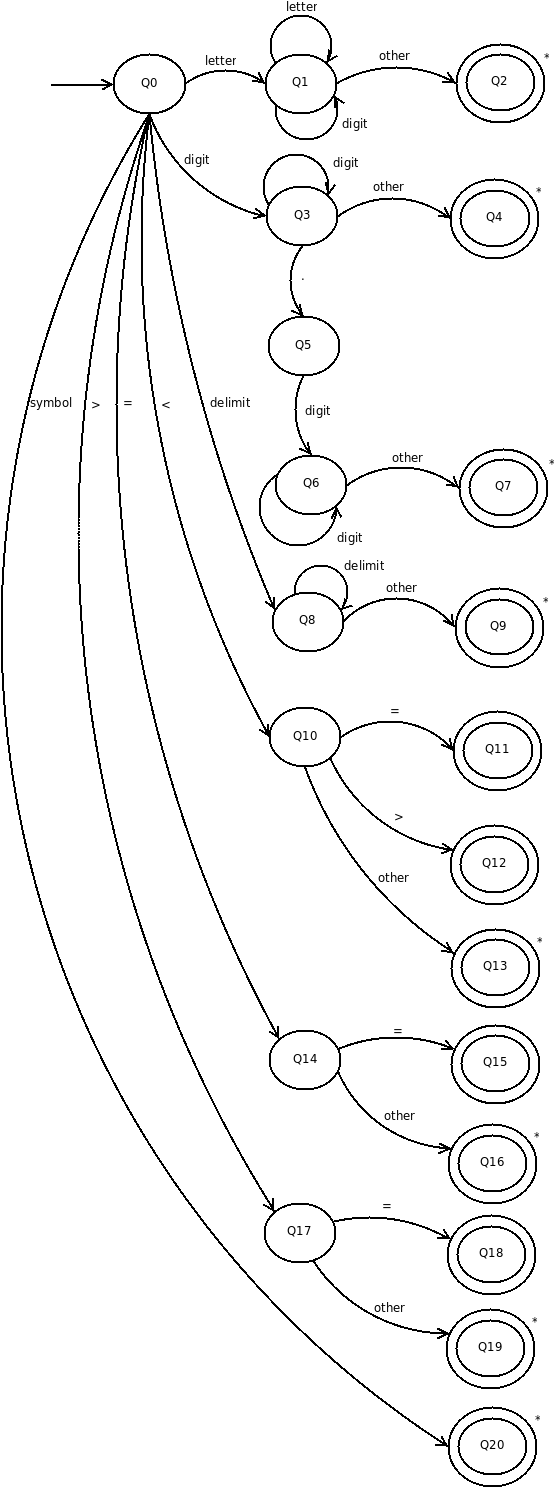
O “symbol” pode ser um dos seguintes símbolos: "+", "-", "\*", "/", "%", ";", ",", "(", ")", "{", "}", "[" e "]".

Para cada uma das transições da máquina está associada uma **ação**. As ações possíveis são:

* empilha: concatena o valor lido com o que está na pilha;
* desempilha: pega o valor que está na pilha e coloca na tabela de símbolos, setando o tipo do símbolo como “identificador”;
* desempilhaNumero: pega o valor que está na pilha e coloca na tabela de símbolos, setando o tipo do símbolo como “numero”;
* ignora: simplesmente não faz nada (usado no caso da leitura de caracteres de controle que são desprezados) e
* colocaTabelaSimbolos: concatena o valor lido com o que está na pilha e coloca na tabela de símbolos.

Para realizar o desenvolvimento do analisador léxico foi preciso atentar para a linguagem definida. Com isso foi criada a tabela de palavras reservadas:

|  |  |  |  |
| --- | --- | --- | --- |
| **Palavras reservadas** | | | |
| 1 | program | 25 | false |
| 2 | end | 26 | function |
| 3 | int | 27 | beginfunction |
| 4 | float | 28 | endfunction |
| 5 | string | 29 | + |
| 6 | boolean | 30 | - |
| 7 | input | 31 | \* |
| 8 | output | 32 | / |
| 9 | while | 33 | % |
| 10 | endwhile | 34 | ; |
| 11 | do | 35 | = |
| 12 | for | 36 | , |
| 13 | beginfor | 37 | ( |
| 14 | endfor | 38 | ) |
| 15 | if | 39 | { |
| 16 | then | 40 | } |
| 17 | else | 41 | [ |
| 18 | endif | 42 | ] |
| 19 | and | 43 | <> |
| 20 | or | 44 | > |
| 21 | xor | 45 | < |
| 22 | not | 46 | <= |
| 23 | return | 47 | >= |
| 24 | true | 48 | == |

Figura : Máquina de Estados do Analisador Léxico

# Análise Sintática

A função principal do analisador sintático é promover a análise da seqüência com que os átomos componentes do texto-fonte se apresentam e, a partir disso, montar sua árvore de sintaxe, com base na gramática da linguagem-fonte.

O método de construção do analisador sintático usado foi o de autômato de pilha estruturado, aplicando-se sobre a notação de Wirth da gramática definida. Após isso, as submáquinas foram simplificadas, retirando recursividades à esquerda e transições em vazio, além de juntar máquinas. No final, o analisador e reconhecedor sintático ficou definido por 14 submáquinas. Nas subseções abaixo as submáquinas são explicadas em detalhes.

## Submáquina Atribuição

A submáquina atribuição reconhece a atribuição de um valor de uma expressão em uma variável. Para isso, ela chama outras duas submáquinas: identificador e expressão. Na figura 2, é mostrada a submáquina atribuição.

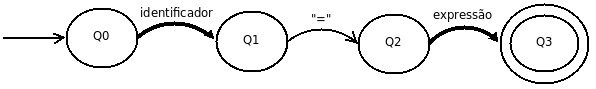


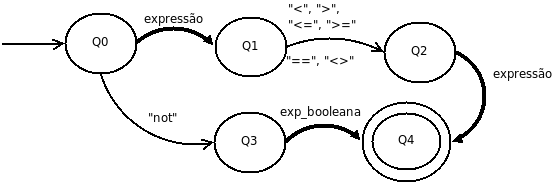
Figura : Submáquina Atribuição

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* chamaExpressao: não consome o token e passa para a submáquina expressão;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Condição

A submáquina condição reconhece uma condição que pode estar dentro de um bloco de iteração ou de condicional. Ela chama outras duas submáquinas: exp\_booleana e expressão. Na figura 3, é mostrada a submáquina condição.

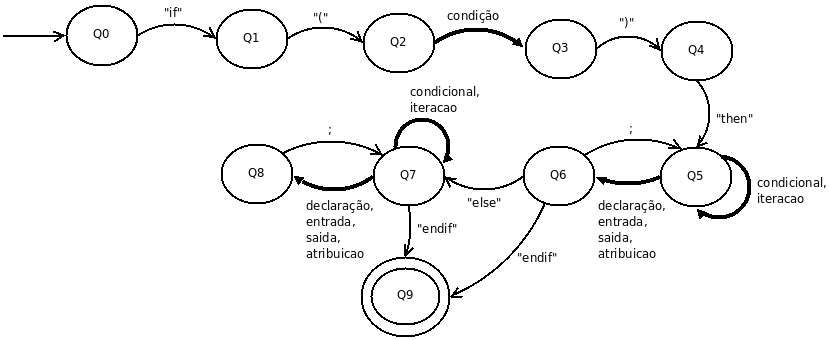
Figura : Submáquina Condição

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaExpBooleana: não consome o token e passa para a submáquina exp\_booleana;
* chamaExpressao: não consome o token e passa para a submáquina expressão;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Condicional

A submáquina condicional reconhece um bloco condicional do tipo “if-else-endif”. Ela chama outras sete submáquinas: condição, declaração, entrada, saída, atribuição, condicional e iteração. Na figura 4, é mostrada a submáquina condicional.

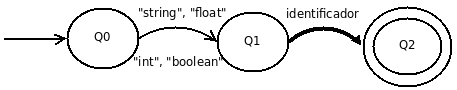
Figura : Submáquina Condicional

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaCondicao: não consome o token e passa para a submáquina condição;
* chamaCondicional: não consome o token e passa para a submáquina condicional;
* chamaIteracao: não consome o token e passa para a submáquina iteração;
* chamaDeclaracao: não consome o token e passa para a submáquina declaração;
* chamaEntrada: não consome o token e passa para a submáquina entrada;
* chamaSaida: não consome o token e passa para a submáquina saída;
* chamaAtribuicao: não consome o token e passa para a submáquina atribuição;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Declaração

A submáquina declaração reconhece um bloco de declaração de uma variável, podendo ser de quatro tipos: int, float, boolean e string. Essa declaração não pode ser atribuída, ou seja, primeiro deve-se declarar a variável e só depois pode atribuir um valor a ela, não é possível fazer as duas coisas ao mesmo tempo. Ela chama a submáquina identificador. Na figura 5, é mostrada a submáquina declaração.

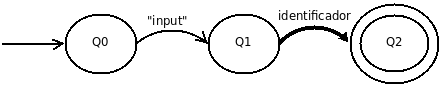
Figura : Submáquina Declaração

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Entrada

A submáquina entrada reconhece um bloco de entrada. A entrada de dados se dá por dispositivos como, por exemplo, o teclado. Essa submáquina chama a submáquina identificador. Na figura 6, é mostrada a submáquina entrada.

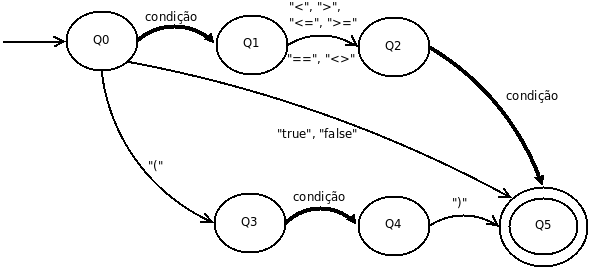
Figura : Submáquina Entrada

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Exp\_Booleana

A submáquina exp\_booleana reconhece uma expressão booleana. Essa submáquina chama a submáquina condição. Na figura 7, é mostrada a submáquina exp\_booleana.

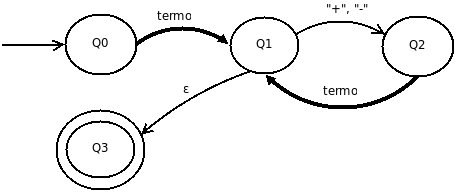
Figura : Submáquina Exp\_Booleana

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaCondicao: não consome o token e passa para a submáquina condicao;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Expressão

A submáquina expressão começa o reconhecimento de uma expressão matemática ou booleana. O reconhecimento de uma expressão é feito em conjunto por três submáquinas: expressão, termo e fator. Essa divisão existe para respeitar a precedência dos operadores aritméticos. Essa submáquina chama a submáquina termo. Na figura 8, é mostrada a submáquina expressão.

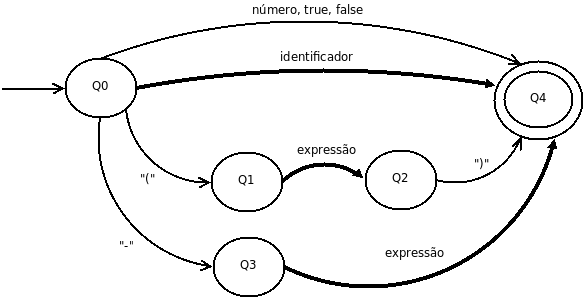
Figura : Submáquina Expressão

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaTermo: não consome o token e passa para a submáquina termo;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Fator

A submáquina fator é a que finaliza o reconhecimento de uma expressão matemática ou booleana. Essa submáquina chama duas outras submáquinas: expressão e identificador. Na figura 9, é mostrada a submáquina fator.

Figura : Submáquina Fator

Para cada uma das transições da submáquina está associada uma ação que são:

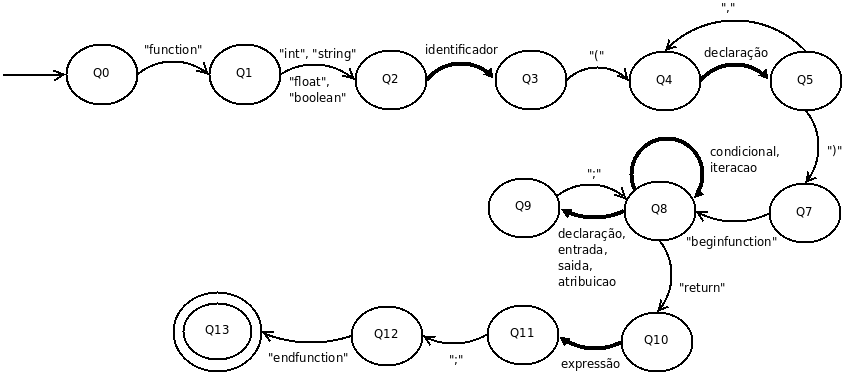
* chamaExpressao: não consome o token e passa para a submáquina expressão;
* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Função

A submáquina função reconhece um bloco de uma função, que inicia com “funtion” e finaliza com um “endfunction”. Ela chama outras oito submáquinas: identificador, expressao, declaração, entrada, saída, atribuição, condicional e iteração. Na figura 10, é mostrada a submáquina função.

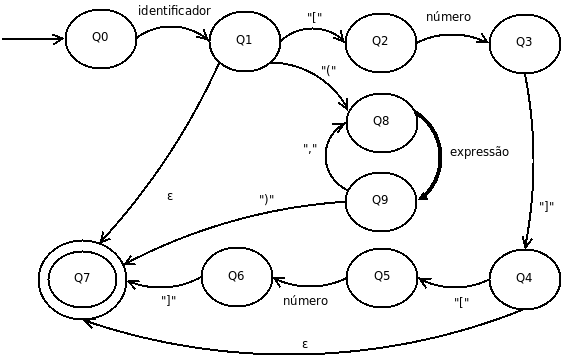
Para cada uma das transições da submáquina está associada uma ação que são:

* chamaCondicional: não consome o token e passa para a submáquina condicional;
* chamaExpressao: não consome o token e passa para a submáquina expressao;
* chamaIteracao: não consome o token e passa para a submáquina iteração;
* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* chamaDeclaracao: não consome o token e passa para a submáquina declaração;
* chamaEntrada: não consome o token e passa para a submáquina entrada;
* chamaSaida: não consome o token e passa para a submáquina saída;
* chamaAtribuicao: não consome o token e passa para a submáquina atribuição;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

Figura : Submáquina Função

## Submáquina Identificador

A submáquina identificador reconhece vários tipos de bloco: variável simples, vetor, matriz e chamada de função. Ela chama a submáquina expressão. Na figura 11, é mostrada a submáquina identificador.

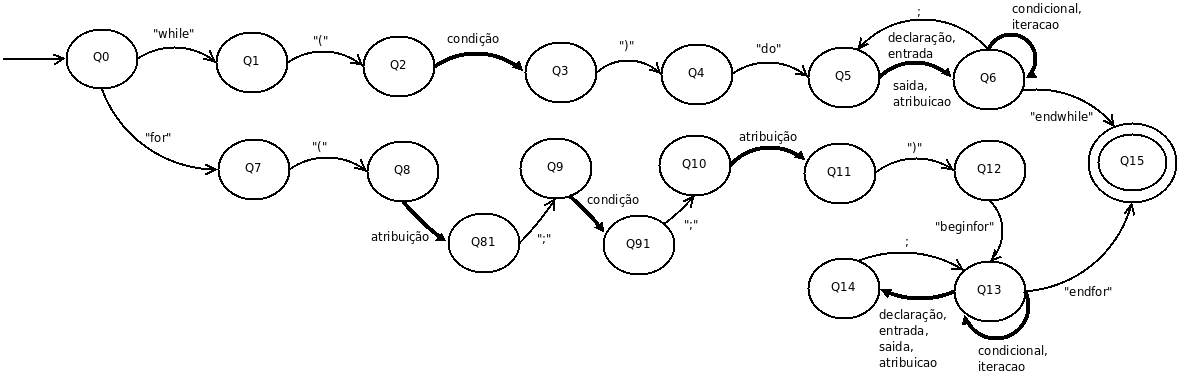
Figura : Submáquina Identificador

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaExpressao: não consome o token e passa para a submáquina expressao;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Iteração

A submáquina iteração reconhece dois tipos de bloco de iteração: “while-do-endwhile” e “for-beginfor-endfor”. Na figura 12, é mostrada a submáquina iteração.

Figura : Submáquina Iteração

Para cada uma das transições da submáquina está associada uma ação que são:

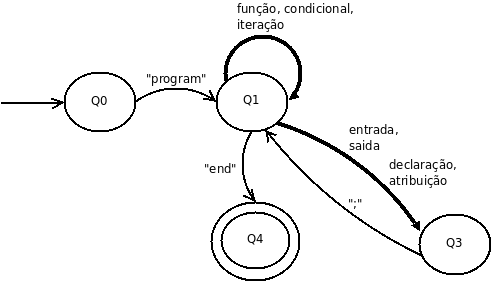
* chamaCondicao: não consome o token e passa para a submáquina condição;
* chamaCondicional: não consome o token e passa para a submáquina condicional;
* chamaIteracao: não consome o token e passa para a submáquina iteração;
* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* chamaDeclaracao: não consome o token e passa para a submáquina declaração;
* chamaEntrada: não consome o token e passa para a submáquina entrada;
* chamaSaida: não consome o token e passa para a submáquina saída;
* chamaAtribuicao: não consome o token e passa para a submáquina atribuição;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Programa

A submáquina programa reconhece a estrutura de um programa (“program-end”). Essa submáquina é a primeira a ser chamada pelo sintático e a que passa os tokens para as demais submáquinas. Na figura 13, é mostrada a submáquina programa.

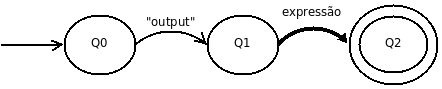
Para cada uma das transições da submáquina está associada uma ação que são:

* chamaFuncao: não consome o token e passa para a submáquina função;
* chamaCondicional: não consome o token e passa para a submáquina condicional;
* chamaIteracao: não consome o token e passa para a submáquina iteração;
* chamaDeclaracao: não consome o token e passa para a submáquina declaração;
* chamaEntrada: não consome o token e passa para a submáquina entrada;
* chamaSaida: não consome o token e passa para a submáquina saída;
* chamaAtribuicao: não consome o token e passa para a submáquina atribuição;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

Figura : Submáquina Programa

## Submáquina Saída

A submáquina saída reconhece um bloco de saída. A saída de dados se dá por dispositivos como, por exemplo, o monitor. Na figura 14, é mostrada a submáquina saída.

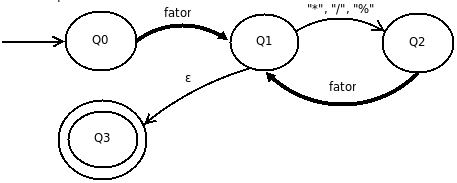
Figura : Submáquina Saída

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaExpressao: não consome o token e passa para a submáquina expressao;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

## Submáquina Termo

A submáquina termo continua o reconhecimento de uma expressão matemática ou booleana. O reconhecimento de uma expressão é feito em conjunto por três submáquinas: expressão, termo e fator. Essa divisão existe para respeitar a precedência dos operadores aritméticos. Na figura 15, é mostrada a submáquina termo.

Figura : Submáquina Termo

Para cada uma das transições da submáquina está associada uma ação que são:

* chamaFator: não consome o token e passa para a submáquina fator;
* ignora: isso vai mudar depois, vai chamar uma ação dosemântico

**Obs.:** Em determinadas transições de estado, é assinalada uma observação no token, através de uma chamada do método:

**tabelaTokens.getToken(posicaoAtual).setObservacao(***observação***)**

As observações podem ser:

* **SCAN**
* **ATRIBUICAO**
* **IF**
* **FIM\_IF\_PARAMETROS**
* **DIVISOR**
* **FIM\_PARAMETROS**
* **FIM\_WHILE**
* **FIM\_WHILE\_PARAMETROS**
* **FIM\_IF**
* **FIM\_IF\_PARAMETROS**
* **FIM\_ELSE**
* **FIM\_**
* **FIM\_GENERICO**

Essas observações são úteis ao analisador semântico e à rotina de geração de código. Além disso, não se faz necessária a construção da árvore sintática, ao se utilizar este artifício.

# Análise Semântica

Os blocos básicos para *Comparação* em linguagem MVN são:

**Comparação Igual (varA == varB)**

1. ACC := MEM(varA)
2. ACC := ACC – MEM(varB)
3. Desvio para 6 se ACC é zero
4. ACC := 0
5. Desvio para 7
6. ACC := 1
7. Fim da comparação

**Comparação Diferente (varA != varB)**

1. ACC := MEM(varA)
2. ACC := ACC – MEM(varB)
3. Desvio para 5 se ACC é zero
4. ACC := 1
5. Fim da comparação

**Comparação Maior (varA > varB)**

1. ACC := MEM(varB)
2. ACC := ACC – MEM(varA)
3. Desvio para 6 se ACC é negativo
4. ACC := 0
5. Desvio para 7
6. ACC := 1
7. Fim da comparação

**Comparação Menor (varA < varB)**

1. ACC := MEM(varA)
2. ACC := ACC – MEM(varB)
3. Desvio para 6 se ACC é negativo
4. ACC := 0
5. Desvio para 7
6. ACC := 1
7. Fim da comparação

**Comparação Maior ou Igual (varA >= varB)**

1. ACC := MEM(varB)
2. ACC := ACC – MEM(varA)
3. Desvio para 7 se ACC é negativo
4. Desvio para 7 se ACC é zero
5. ACC := 0
6. Desvio para 8
7. ACC := 1
8. Fim da comparação

**Comparação Menor ou Igual (varA <= varB)**

1. ACC := MEM(varA)
2. ACC := ACC – MEM(varB)
3. Desvio para 7 se ACC é negativo
4. Desvio para 7 se ACC é zero
5. ACC := 0
6. Desvio para 8
7. ACC := 1
8. Fim da comparação

# Bibliografia

[1] JOSÉ NETO, J., Introdução à compilação – Livros técnicos e científicos, Editora S.A. Rio de Janeiro, 1987.

[2] AHO, A. V. & SETHI, R. & ULLMAN, J. D. & LAM, MONICA S., Compilers. Principles, Techniques and tools, Editora Pearson Education, 2ª Edição, 2006.