|  |  |
| --- | --- |
|  | 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](#_Toc247322757)

[2. Definição da Linguagem 5](#_Toc247322758)

[2.1 Recursos da Linguagem 5](#_Toc247322759)

[2.1.1 Estrutura do programa 5](#_Toc247322760)

[2.1.2 Variáveis simples 5](#_Toc247322761)

[2.1.3 Variáveis indexadas – vetor e matriz 5](#_Toc247322762)

[2.1.4 Comandos de declaração de variáveis 6](#_Toc247322763)

[2.1.5 Comandos de atribuição 6](#_Toc247322764)

[2.1.6 Comandos de entrada 6](#_Toc247322765)

[2.1.7 Comandos de saída 6](#_Toc247322766)

[2.1.8 Comandos condicionais 6](#_Toc247322767)

[2.1.9 Comandos iterativos 6](#_Toc247322768)

[2.1.10 Expressões aritméticas 7](#_Toc247322769)

[2.1.11 Expressões booleanas 7](#_Toc247322770)

[2.2 Notação BNF 7](#_Toc247322771)

[2.3 Notação Wirth 9](#_Toc247322772)

[3. Leitor de Máquina de Estados 11](#_Toc247322773)

[4. Análise Léxica 12](#_Toc247322774)

[5. Análise Sintática 15](#_Toc247322775)

[5.1 Submáquina Atribuição 15](#_Toc247322776)

[5.2 Submáquina Condição 15](#_Toc247322777)

[5.3 Submáquina Condicional 16](#_Toc247322778)

[5.4 Submáquina Declaração 17](#_Toc247322779)

[5.5 Submáquina Entrada 17](#_Toc247322780)

[5.6 Submáquina Exp\_Booleana 18](#_Toc247322781)

[5.7 Submáquina Expressão 18](#_Toc247322782)

[5.8 Submáquina Fator 19](#_Toc247322783)

[5.9 Submáquina Função 19](#_Toc247322784)

[5.10 Submáquina Identificador 20](#_Toc247322785)

[5.11 Submáquina Iteração 21](#_Toc247322786)

[5.12 Submáquina Programa 22](#_Toc247322787)

[5.13 Submáquina Saída 23](#_Toc247322788)

[5.14 Submáquina Termo 23](#_Toc247322789)

[6. Análise Semântica 24](#_Toc247322790)

[6.1 Código-Objeto: MVN 25](#_Toc247322791)

[6.2 Projeto das Ações Semânticas 26](#_Toc247322792)

[6.3 Ações Semânticas 27](#_Toc247322793)

[6.3.1 Submáquina Programa 27](#_Toc247322794)

[6.3.2 Submáquina Declaração 28](#_Toc247322795)

[6.3.3 Submáquina Atribuição 29](#_Toc247322796)

[7. Classes Auxiliares 30](#_Toc247322797)

[7.1 Estrutura 30](#_Toc247322798)

[7.1.1 Classe Pilha 30](#_Toc247322799)

[**Métodos da Classe:** 30](#_Toc247322800)

[7.2 Semântico 30](#_Toc247322801)

[7.2.1 Classe Memória 30](#_Toc247322802)

[**Atributos da Classe:** 30](#_Toc247322803)

[**Métodos da Classe:** 31](#_Toc247322804)

[7.2.2 Classe PosiçãoDeMemória 31](#_Toc247322805)

[**Atributos da Classe:** 31](#_Toc247322806)

[**Métodos da Classe:** 32](#_Toc247322807)

[8. Bibliografia 33](#_Toc247322808)

# 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 "]".

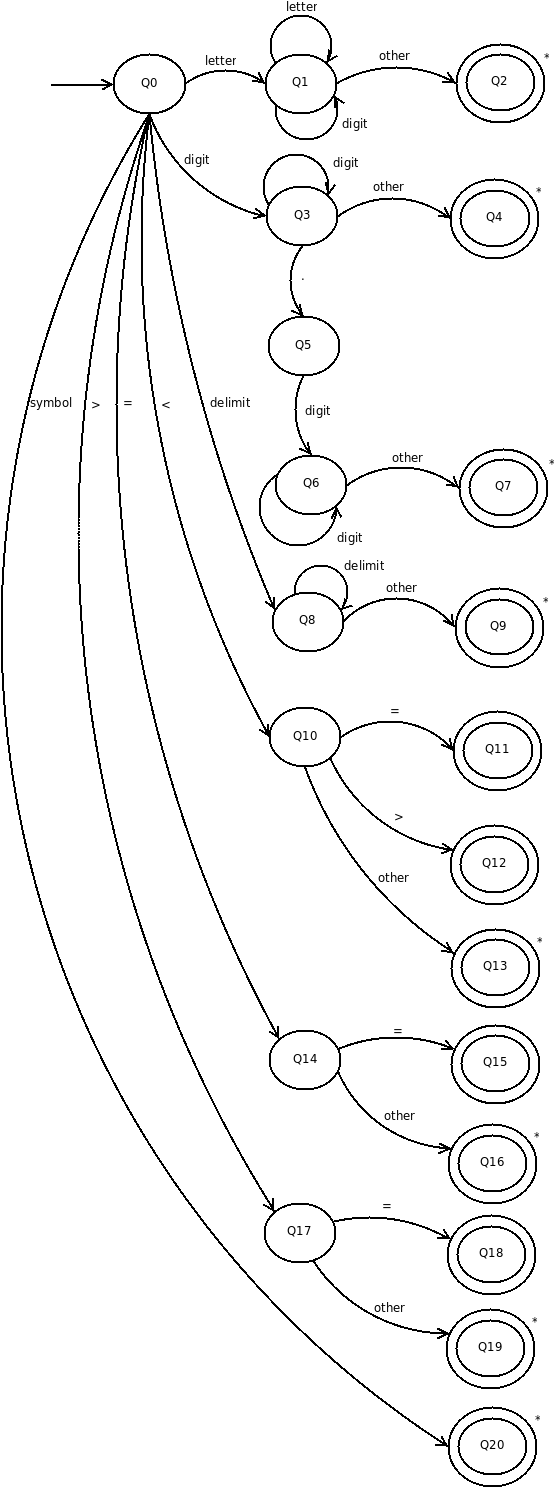
Para cada uma das transições da máquina está associada uma **ação**. As ações possíveis são:

* 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.

Assim, 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.

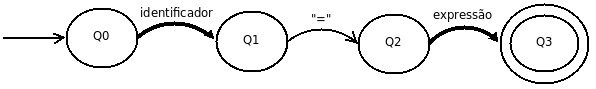


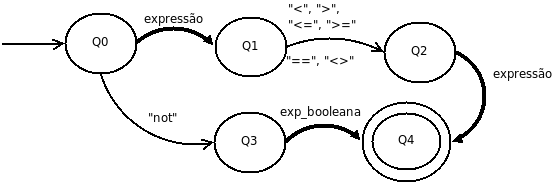
Figura : Submáquina Atribuição

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: não faz nada.

## 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.

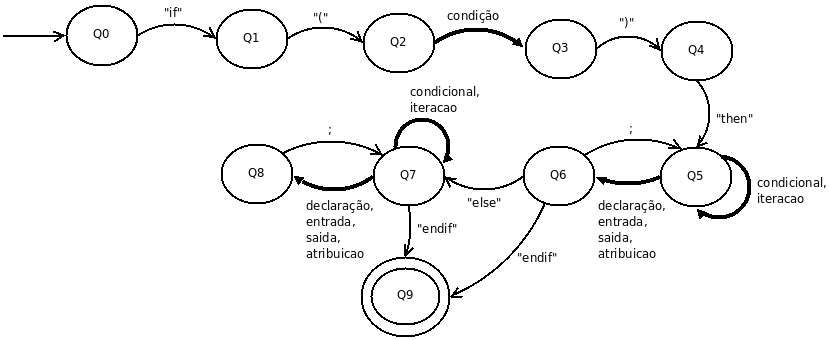
Figura : Submáquina Condição

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: não faz nada.

## 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.

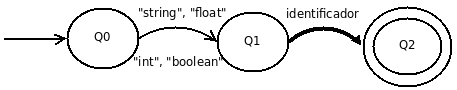
Figura : Submáquina Condicional

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: não faz nada.

## 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.

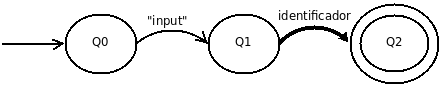
Figura : Submáquina Declaração

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: não faz nada.

## 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.

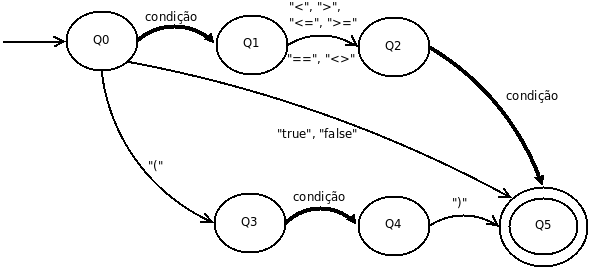
Figura : Submáquina Entrada

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: não faz nada.

## 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.

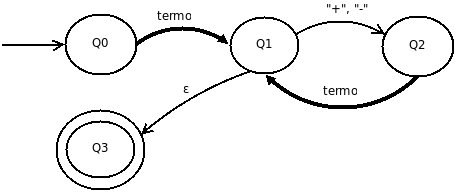
Figura : Submáquina Exp\_Booleana

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: não faz nada.

## 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.

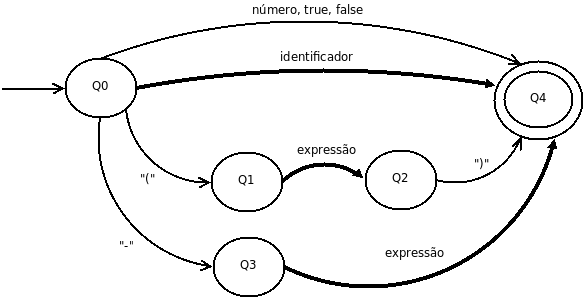
Figura : Submáquina Expressão

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: não faz nada.

## 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:

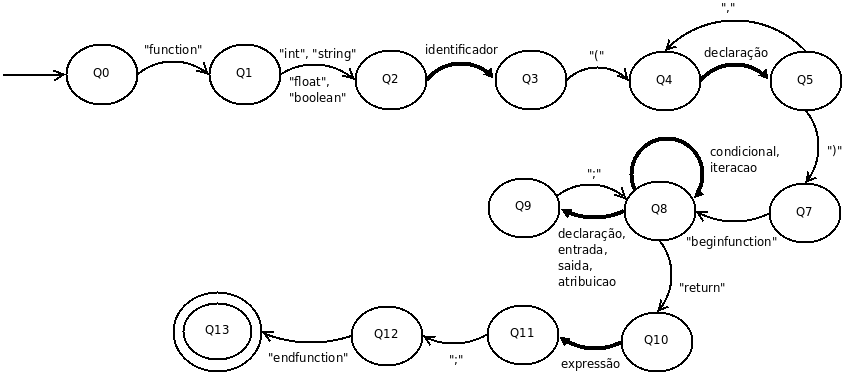
* chamaExpressao: não consome o token e passa para a submáquina expressão;
* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* ignora: não faz nada.

## 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.

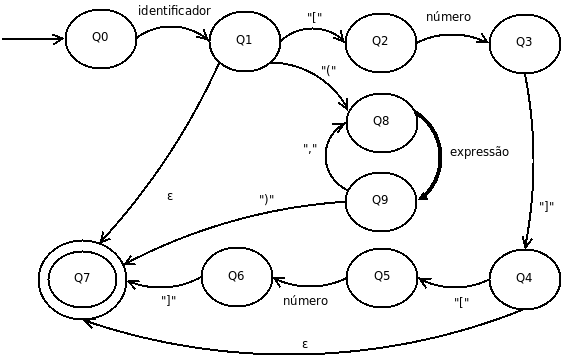
Para cada uma das transições da submáquina está associada uma ação que são:

* 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: não faz nada.

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.

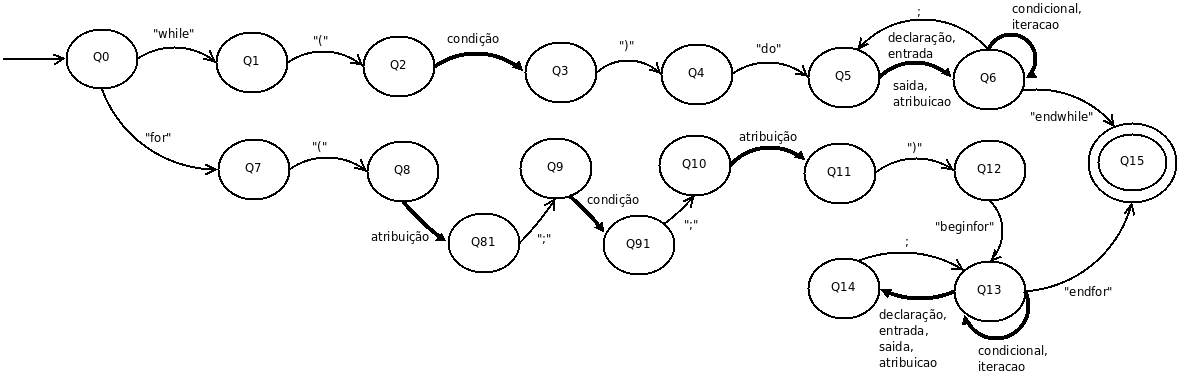
Figura : Submáquina Identificador

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: não faz nada.

## 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:

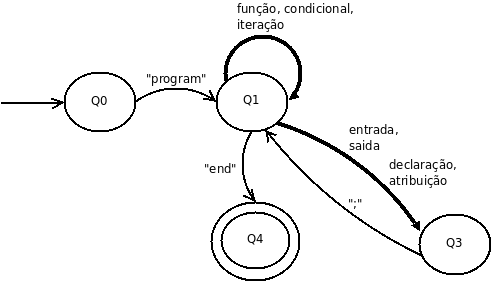
* chamaCondicao: não consome o token e passa para a submáquina condição;
* chamaCondicional: não consome o token e passa para a submáquina condicional;
* chamaIteracao: não consome o token e passa para a submáquina iteração;
* chamaIdentificador: não consome o token e passa para a submáquina identificador;
* 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: não faz nada.

## 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.

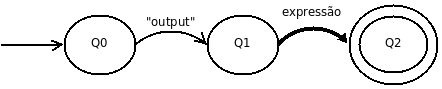
Para cada uma das transições da submáquina está associada uma ação que são:

* 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: não faz nada.

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.

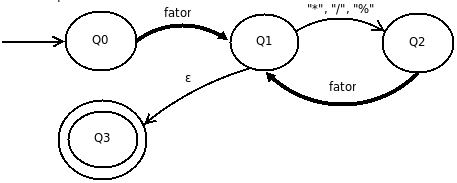
Figura : Submáquina Saída

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: não faz nada.

## 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: não faz nada.

# Análise Semântica

A análise semântica é um dos módulos principais constituintes de um compilador, uma vez que são determinadas as ações semânticas responsáveis pelo tratamento da dependência de contexto da linguagem e pela geração do código-objeto.

A relação entre os módulos Semântico e Sintático ocorre exatamente nas transições de estados das submaquinas definidas pelo Sintático. Assim, para cada transição de uma submáquina, é possível verificar se as regras gramaticais *dependentes de contexto* são obedecidas.

As metas principais destas regras verificadas neste projeto são:

* Manutenção da Tabela de Símbolos: em todas as declarações contidas no programa-fonte, o TOKEN é modificado na Tabela de Símbolos, setando os atributos *categoria* (neste caso, a categoria é *variável*)e *declarado* (passa a ser *true*);
* Associação de Atributos aos Símbolos: os atributos associados aos Tokens lidos são Código, Nome, Tipo (*identificador*, *número*, *palavraReservada*), Categoria (*variável*, *vetor*, *parâmetro* ou *função*), *declarado* (*true* ou *false –* default);

Com relação às regras gramaticais, deve-se realizar as seguintes verificações:

* Verifica se os Tokens do tipo *Identificador* já foram declarados: o atributo *declarado* já está definido com o valor *true*.

**Obs.:** Neste projeto, esta verificação foi feita utilizando o artifício de um atributo indicativo para declaração, no entanto, poderia ser verificado na Tabela de Símbolos a existência do Token.

* Verificação das expressões aritméticas e booleanas;
* Verifica se os Tokens do tipo *Identificador* declarados como *int* ou *boolean* são utilizados em expressões aritméticas e booleanas, respectivamente;
* Verifica se, nas chamadas de função, os parâmetros passados estão corretos de acordo com a assinatura da função declarada. Assim, tanto a quantidade de parâmetros passados e o tipo de cada parâmetro devem estar coerentes na ordem em que foram definidos na assinatura da função;
* Verifica se foram atribuídas expressões do tipo correto em conformidade com o tipo do *Identificador* que recebe seu valor.

**[RESTRIÇÃO DO PROJETO]**

**Obs.:** O compilador Hopper não foi projetado para aceitar funções sem retorno definidos pela palavra reservada VOID. Assim, toda função declarada deve possuir um tipo associado (no caso, *int* ou *boolean*).

Os comandos aceitos pela linguagem para a qual o compilador está sendo construído estão listados abaixo:

* Comando de Declaração (variável simples, vetor ou função)
* Comando de Atribuição (aritmética ou booleana de variável simples ou vetor)
* Comando WHILE
* Comando IF-THEN-ELSE-ENDIF
* Comando de Chamada de função
* Comando de Entrada de dados
* Comando de Saída de dados

Primeiramente, foi necessária a construção de classes que simulassem uma memória da máquina MVN. Para implementar esse conceito, foram criadas as classes *Memória* e *PosicaoMemoria*, explicadas na seção seguinte de Classes Auxiliares.

A seguir, será detalhada a maneira como cada um dos comandos anteriores em alto nível foi construído por instruções reconhecidas pela MVN.

Por fim, a geração de código fundamentou-se no conceito de *Rótulos* para identificar cada posição de memória a ser ocupada pelo programa em compilação.

## Código-Objeto: MVN

Na Tabela abaixo, são apresentadas as instruções da MVN:

|  |  |  |
| --- | --- | --- |
| Código (hexa) | Instrução (4 bits) | Operando (12 bits) |
| 0 | Desvio incondicional | Endereço de desvio |
| 1 | Desvio se acumulador é zero | Endereço de desvio |
| 2 | Desvio se acumulador é negativo | Endereço de desvio |
| 3 | Deposita uma constante no acumulador | Constante de 12 bits |
| 4 | Soma | Endereço da parcela |
| 5 | Subtração | Endereço do subtraendo |
| 6 | Multiplicação | Endereço do multiplicador |
| 7 | Divisão | Endereço do divisor |
| 8 | Memória para acumulador | Endereço do dado de origem |
| 9 | Acumulador para memória | Endereço de destino |
| A | Desvio de subprograma (função) | Endereço do subprograma |
| B | Retorno de subprograma (função) | Endereço do resultado |
| C | Parada | Endereço de desvio |
| D | Entrada | Dispositivo de E/S |
| E | Saída | Dispositivo de E/S |
| F | Chamada de supervisor | Constante |

Para as operações de **Entrada** e **Saída**, devem ser definidos os dispositivos de acordo com a lógica a seguir:

|  |  |  |
| --- | --- | --- |
| **OP** | **Tipo** | **Dispositivo** |

**OP** D (entrada) ou E (saída)

**Tipo** Tipos de dispositivos:

0 = Teclado

1 = Monitor

2 = Impressora

3 = Disco

**Dispositivo** Identificação do dispositivo. Pode-se ter vários tipos de dispositivos, ou unidades lógicas (LU). No caso do disco, um arquivo é considerado uma unidade lógica. Pode-se ter, portanto, até 16 tipos de dispositivos e, cada um, pode ter até 256 unidades lógicas.

## Projeto das Ações Semânticas

Com o objetivo de projetar as ações semânticas, serão descritas as funções de geração de código para cada ***operação*** do compilador Hopper.

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

**Comparação Igual (varA == varB)**

1. (ACC = )
2. ACC := MEM(varA)

**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

Para simplificar as expressões em notação de Wirth, serão realizadas as seguintes alterações:

* Supressão das aspas que envolvem não-terminais
* Palavras reservadas serão sublinhadas
* *Identificador* será denotado por **I**
* *Número* será denotado por **N**
* *Comando* será denotado por **C**
* *Expressão* será denotado por **E**
* Abre e fecha parênteses serão denotados, respectivamente, por

## Ações Semânticas

Uma vez definidas as funções para geração de código MVN a serem executadas pelas ações semânticas, resta projetar sua lógica de acordo com as transições convenientes das submáquinas implementadas no Analisador Sintático.

Assim, para cada submáquina do Analisador Sintático, deve-se definir as ações semânticas apropriadas com a finalidade de integrar os dois módulos do compilador. Por comodidade, serão utilizadas as formas tabulares de representação.

**Obs.:** A representação tabular descreve as ações semânticas convenientes às transições da submáquina.

### Submáquina Programa

Segue a Tabela de Transições desta submáquina com a inclusão das ações semânticas para o átomo Programa:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | program | funcao | condicional | iteracao | entrada | saida | declaracao | atribuicao | ; | end | ação de finalização |
| Q0 | Q1/1 |  |  |  |  |  |  |  |  |  | 11 |
| Q1 |  | Q1/2 | Q1/3 | Q1/4 |  |  |  |  |  | Q4/10 | 12 |
| Q3 |  |  |  |  | Q3/5 | Q3/6 | Q3/7 | Q3/8 | Q1/9 |  | 13 |
| * Q4 |  |  |  |  |  |  |  |  |  |  | 14 |

As ações semânticas foram construídas da seguinte forma:

|  |  |  |  |
| --- | --- | --- | --- |
| Ações Semânticas | Átomo Associado | Código Gerado | Outras Ações |
| 1 | program | (ver detalhes) | (ver detalhes) |
| 2 | funcao | - | - |
| 3 | condicional | - | - |
| 4 | iteracao | - | - |
| 5 | entrada | - | - |
| 6 | saida | - | - |
| 7 | declaracao | - | - |
| 8 | atribuicao | - | - |
| 9 | ; | - | - |
| 10 | end | (ver detalhes) | (ver detahes) |
| 11 | (nenhum) | - | mensagem de erro |
| 12 | (nenhum) | - | mensagem de erro |
| 13 | (nenhum) | - | mensagem de erro |
| 14 | (nenhum) | - | finalização da compilação |

**Ação 1:** Tratamento de PROGRAM (início do programa)

1. (defino a área do programa – neste projeto, foi decidido arbitrariamente pela posição 0 da memória)
2. GERA\_CODIGO 🡪 *ESPACO* @ =0 ; area do programa
3. (para cada constante numérica utilizada pelo código-fonte, deve-se alocar um espaço de memória com seu respectivo valor)
4. (para cada uma das variáveis temporárias utilizadas no processamento do código-fonte, deve-se alocar um espaço de memória)
5. GERA\_CODIGO 🡪 *ESPACO* HM /0
6. (encerra o processamento do código-fonte)

**[RESTRIÇÃO DO PROJETO]**

**Obs.:** Note que, por simplicidade, não é possível declarar funções externas ao programa principal *PROGRAM*. O compilador implementado deve iniciar obrigatoriamente pela palavra reservada *PROGRAMA* e necessariamente terminar pela palavra reservada *END*.

**[DETALHE DO PROJETO]**

**Obs.:** Por comodidade, as mensagens de erro para o Semântico foram definidas apenas por “*Erro no semântico!*”.

**Ação 10:** Tratamento de END (fim do programa)

1. (para cada um dos identificadores declarados como variáveis, deve-se alocar um espaço de memória)
2. (para cada constante numérica utilizada pelo código-fonte, deve-se alocar um espaço de memória com seu respectivo valor)
3. (para cada uma das variáveis temporárias utilizadas no processamento do código-fonte, deve-se alocar um espaço de memória)
4. GERA\_CODIGO 🡪 HM /0
5. (encerra o processamento do código-fonte)

### Submáquina Declaração

Segue a Tabela de Transições desta submáquina com a inclusão das ações semânticas para Declaração (de variáveis):

|  |  |  |
| --- | --- | --- |
|  | int | boolean | I |
| Q0 | Q1/1 |  |
| Q1 |  | Q2/2 |
| * Q2 |  |  |

As ações semânticas foram construídas de acordo com o seguinte roteiro:

1. Empilha( pilha de operandos, identificador encontrado)

### Submáquina Atribuição

Segue a Tabela de Transições desta submáquina com a inclusão das ações semânticas para o comando Atribuição:

|  |  |  |  |
| --- | --- | --- | --- |
|  | I | = | E |
| Q0 | Q1/1 |  |  |
| Q1 |  | Q2/2 |  |
| Q2 |  |  | Q3/3 |
| * Q3 |  |  |  |

As ações semânticas foram construídas de acordo com o seguinte roteiro:

1. Empilha( pilha de operandos, identificador encontrado)

# Classes Auxiliares

## Estrutura

Este *package* implementa classes auxiliares de estrutura de dados, tais como: Pilha.

### Classe Pilha

Implementa uma estrutura de dados de pilha.

**Métodos da Classe:**

* void empilha(Object elemento): empilha o elemento passado como parâmetro;
* Object desempilha(): desempilha o elemento do topo, retornando um elemento do tipo Object.
* boolean pilhaVazia(): verifica se a pilha está vazia, retornando TRUE se estiver, e FALSE, em caso contrário.
* void esvaziaPilha(): esvazia a pilha, resetando o apontador do topo.

## Semântico

No mesmo *package* do semântico, foram implementadas classes para simplificar a manipulação da memória da MVN.

### Classe Memória

A classe memória representa a memória da máquina MVN. Consiste apenas em uma estrutura de lista ligada, onde cada elemento desta lista corresponde a uma posição da memória. No caso, estamos utilizando como elemento desta lista uma instância da classe PosicaoMemoria, a ser descrita no próximo item.

Este classe possui métodos que foram implementados para que a inserção/remoção de novos blocos de memória fosse facilitada. Estes métodos permitem a inserção de um bloco em determinada posição, deslocando todos os demais da lista. Também permite a definição de um apontador que referencia determinada posição da memória para que os próximos elementos possam ser adicionados antes ou depois desta posição.

**Atributos da Classe:**

* Pilha pilhaInserirAntes: corresponde a uma pilha que armazena os rótulos marcadores. É utilizado no caso em que é necessária a utilização de um novo rótulo marcador quando já existir um outro rótulo em uso, ou seja, quando existem comandos dentro de comandos que utilizam esta idéia de rótulo marcador.
* String rotuloMarcador: guarda o rótulo atual que está sendo utilizado como referência para a inserção de novos elementos.
* Boolean inserirAntes: flag que indica se existe algum rótulo sendo utilizado como marcador.

**Métodos da Classe:**

* incrementarInserirAntes(String rotulo): seta o rótulo que está sendo passado como parâmetro para ser utilizado como referência para inserção dos próximos rótulos. Se já existir algum rótulo sendo utilizado, o rótulo atual é colocado na pilha e o rótulo do parâmetro passa a ser a referência.
* decrementarInserirAntes(): retira um rótulo que está sendo utilizado como referência. Se existir mais algum rótulo na pilha, este é removido da pilha e passa a ser o rótulo marcador.
* inserir(PosicaoMemoria posição): insere o objeto posição na memória, de acordo com a existência ou não de um rótulo marcador. Caso exista, é chamado o método inserirAntes. Caso contrário, o objeto é inserido ao final da lista (da memória).
* inserirAntes(PosicaoMemoria posição): insere o objeto na posição anterior ao rótulo marcador. É chamado apenas pelo método inserir.
* imprimir(): imprime o conteúdo da memória.
* imprimirMVNes(): imprime o conteúdo da memória já no formato correto para ser utilizado como entrada na máquina MVN. Deve ser utilizado apenas após os endereços dos operandos terem sido resolvidos.
* getEnderecoDeRotulo(String rotulo): procura na memória por algum elemento que possua um rótulo igual ao que está sendo passado como parâmetro. Caso encontre, retorna o endereço correspondente ao rótulo. Caso contrário, retorna nulo. Deve ser chamado apenas após os endereços dos operandos terem sido resolvidos.
* remover(PosicaoMemoria posicaoMemoria): remove um elemento da memória de acordo com a informação contida no objeto posicaoMemoria passado como parâmetro.

### Classe PosiçãoDeMemória

Esta classe representa um elemento na memória, ou seja, uma posição da memória. Em cada elemento da memória existe uma instância desta classe, contendo as informações necessárias para a geração de código e resolução de endereços.

**Atributos da Classe:**

* int endereço: contém o endereço em hexadecimal da posição da memória após terem sido resolvidos os endereços.
* int operação: contém a operação MVN da instrução corresponde à instância desta PosicaoMemoria.
* String rotuloOperando: corresponde ao rótulo do operando. Para formação de uma instrução MVN são necessários um operador e um operando. O rótuloOperando corresponde ao rótulo do operando (um endereço de memória) sendo referenciado. Se este for uma constante, não será necessária a resolução do endereço do operando.
* int operando: contém o operando da instrução, em hexadecimal.
* String rotulo: contém o rótulo desta posição da memória.

**Métodos da Classe:**

* setEndereco(String endereço): seta o endereço da posição da memória. A entrada é uma string de um valor hexadecimal. Internamente ao código é realizada a conversão deste valor hexadecimal para inteiro e é feito seu armazenamento no atributo endereco.
* setOperacao(String operação): seta o operador da instrução. A entrada em hexadecimal é convertida para inteiro e armazenada no atributo operação.
* setRotuloOperando(String rotulo): seta o rótulo do operando.
* setOperando(String operando): operação de set para o atributo operando. Internamente ao método é feita a conversão da string operando para inteiro.
* setRotulo(String rotulo): operação de set para o atributo rotulo.
* Operações de get: retornam os referidos atributos de cada um dos métodos.
* toString(): imprime os principais campos desta classe e seus conteúdos.

# 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.