UNIVERSIDADE FEDERAL DE ALAGOAS - UFAL
COMPILADORES 2018.2
INSTITUTO DE COMPUTAÇÃO - IC

IAGO BARBOZA DE SOUZA

Especificação da linguagem de programação HI

Maceió
Março de 2019

1. Introdução

A Linguagem de programação HI tem o propósito de ser utilizada na implementação de programas simples, com algoritmos simples, funcionando como uma linguagem interessante para se aprender os conceitos mais básicos de programação tais como operações aritméticas e estruturas condicionais.

2. Características gerais

- Cobre uma classe de aplicações simples que utilizam algoritmos simples;
- Adota um paradigma de programação imperativa;
- Linguagem compilada.

Uma vez que HI é uma linguagem de programação estática, não há um tratamento de erro de detecção de tipo, isso implica na sua confiabilidade.

2.1 Palavras reservadas

Em HI existem palavras reservadas que são: *none, int, float, char, string, boolean, array, while, repeater, print e read.*

3. Forma geral de um programa em HI

Um programa em HI é um conjunto de instruções definidas de forma sequencial e imperativa, obedecendo às especificações da linguagem, ou seja, à sua sintaxe estrutural de comandos. Essas instruções são escritas em um arquivo de texto cuja a extensão é “.hi”, como por exemplo “helloworld.hi” e “fibonacci.hi”.

Todos os programas em HI são compostos por funções, sendo a função *main() a* primeira a ser executada. Ela é uma função sem retorno e que possui a seguinte sintaxe:

```
main() none {
}
```

Para sinalizar o final de um comando, deve-se utilizar o **;** e para a delimitação de escopo deve-se utilizar **{** no início do escopo e **}** no final do escopo.
A linguagem HI permite o uso de comentários de linha, que podem ser definidos utilizando-se **$** na linha a ser comentada mas não suporta comentários do tipo bloco.

3.1 Identificador

Os identificadores em HI seguem as seguintes regras:

- Iniciam-se obrigatoriamente com um letra minuscula;
- O tamanho é de 16 caracteres;
- É vetada a utilização de espaços em branco;
- Não podem usar palavras reservadas.

Tudo é passado por referência não permitindo efeito colateral.

4. Especificação de tipos

A linguagem HI é estaticamente tipada, ou seja, no momento da definição da declaração de uma variável, o tipo dela é obrigatoriamente necessário, ficando associado à variável durante todo o ciclo de vida dela. Os tipos de variáveis suportadas pela linguagem HI são:

- **int** - Inteiro, identifica a variável como um númer, tendo seus literais expressos em uma sequência de dígitos decimais;
  int inteiro;
- **float** - Ponto flutuante, Identifica a variável como um ponto flutuante, tendo seus literais expressos em uma sequência de dígitos decimais seguidos por um ponto e os demais dígitos decimais;
- **char** - Caractere;
- **string** - Cadeia de caracteres;
- **boolean** - Booleano;
- **none** - Tipo de função sem retorno;
- **tipo_da_variável** variável[**tamanho_do_arranjo(Deve ser uma contante)**] - Arranjos Unidimensionais.

4.1. Operações suportadas por cada tipo

Tipo	**Operação Suportada**
int	*atribuição, aritméticas, relacionais*
float	*atribuição, aritméticas, relacionais*
char	*atribuição, relacionais*
string	*atribuição, relacionais, concatenação*
boolean	*atribuição, relacionais, lógicos*
array	*atribuição*

4.2. Coerção

A linguagem é estaticamente tipada, não aceitando coerção entre variáveis de tipos diferentes. As verificações de compatibilidade de tipos serão feitas estaticamente, com isso, deve-se aumentar a detecção de erros. As operações que envolvem os tipos *int* e *float* devem ter como resultado um tipo float. Exemplo:

```
Int a = 2;
float b = 2;
float x ;
x = a + b;
```

4.3. Valores Default

Os valores default atribuídos à cada variável declarada são:

TIPO	**Valor Default**
Int	0
float	0.0
Array	NULL
char	NULL
string	NULL
Boolean	False

5. Operadores

5.1. Aritméticos

+	*adição*
-	*subtração*
*	*multiplicação*
/	*divisão*
^	*exponenciação*
-	*unário negativo*

5.2. Relacionais

>	*maior que*
<	*menor que*
==	*igual*
!=	*diferente de*
>=	*maior ou igual*
<=	*menor ou igual*

5.3. Lógicos

not	*negação*
and	*conjunção*
or	*disjunção*

5.4. Concatenação de cadeias de caracteres

++	*concatenação*

5.5. Precedência e Associatividade

A tabela abaixo define a ordem de precedência da mais alta para a mais baixa e as regras de associatividade dos operadores:

Operador	**Comentário**	**Associatividade**
-	Menos unário	*direita para esquerda*
^	Multiplicativo	*direita para esquerda*
* /	Multiplicativo	*esquerda para direita*
+ -	Aditivos	*esquerda para direita*
< <= > >=	Comparativos	*não associativos*
== !=	Igualdade	*esquerda para direita*
and	Conjunção	*esquerda para direita*

or	Disjunção	*esquerda para direita*
not	Negação	*esquerda para direita*
=	Atribuição	*direita para esquerda*

5.6 Operadores multiplicativos e aditivos

Quando esses operadores são utilizados em variáveis do mesmo tipo o valor resultante desta operação **deve** ser armazenado em uma variável de mesmo tipo.

Caso esses operadores sejam utilizados em variáveis de tipos diferentes (int e float) o valor resultante será do tipo float. Não sendo tratados outros tipos.

5.7 Operadores comparativos e igualdade

Estes operadores não são associativos e o “valor” resultante é uma variável do tipo bool.

5.8 Operadores de negação, conjunção e disjunção

Após instrução com tais operadores, o valor resultante é do tipo bool e é definido pelo valor da sua utilização na lógica booleana.

5.9 Operador de concatenação

No operador de concatenação as operações devem ser do tipo char e string, obtida concatenando-se ambas as variáveis.

6. Instruções

Cada linha de instrução só é terminada com a presença de " # ".

6.1. Estrutura condicional de uma e duas vias

if
if / else

A instrução *if* define um escopo acessível apenas se uma condição for satisfeita. Essa condição é definida através de uma expressão cujo o resultado deve ser um *boolean*. Caso o valor do *boolean* seja true, o escopo será executado, caso contrário, o escopo não será executado.

```
if(expressão){
        comando 1#
        comando 2#
        …
        comando N#
  }
```

No caso de duas vias, devemos definir 2 escopos. O primeiro é definido pelo *if* e será acessível apenas se a condição definida pela expressão for satisfeita. O *else* define o segundo escopo, que será executado apenas se a condição estabelecida pela expressão do *if* não for satisfeita.

```
if(expressão){
        comando 1#
        comando 2#
}
```

```
else{
        Comando 3#
}
```

6.2. Estrutura iterativa com controle lógico

while

A instrução *while* define um escopo que será executado N vezes enquanto uma expressão definida for satisfeita, ou seja, enquanto a expressão gerar um *boolean* com valor true.

```
while(expressão){
        comando 1#
        comando 2#
                …
        comando N#
  }
```

6.3. Estrutura iterativa controlada por contador

repeater

A instrução *repeater* pode ser utilizada para a repetição de um bloco (escopo) de instruções, tomando como base um contador.

```
repeater(i, range(start, end)){
        comando 1#
        comando 2#
        …
```

```
        comando N#
    }
```

6.4. Entrada e saída

print
read

A instrução *print* é utilizada para imprimir uma mensagem na tela. Essa mensagem pode ser uma cadeia de caracteres (string) ou uma variável (nesse caso será impresso o conteúdo da variável), logo essa instrução pode receber qualquer um desse elementos como parâmetro.

A instrução *read* é utilizada para a leitura de dados no console, cujos a inserção é feita pelo usuário. Ela recebe como parâmetro o tipo da variável, seguido do seu identificador.

```
print(“hello world”)#
string message = “hello world”#
print(message)#

read(string, message)#
```

7. Atribuição

A atribuição é feita através do operador =, seguindo as regras de associatividade definidas, ou seja, com associatividade da direita para esquerda.

```
a = b# (O valor da variável b será atribuído à variável a)
```

8. Funções

Na linguagem HI, as funções devem obrigatoriamente ser declaradas antes da sua utilização ou chamada. Funções aninhadas não são permitidas, ou seja, cada função deve ser implementada de forma separada de qualquer outra função.

O tipo do retorno da função é opcional. Caso não exista, a função não retorna nada. Caso um tipo de retorno seja definido, a função deve retornar um valor do tipo utilizando o comando *return.*

```
tipo_do_retorno(opcional) nome_da_função(tipo_do_parametro1 parametro1, tipo_do_parametro2 parametro2, …, tipo_do_parametroN parametroN) {
        comando 1;
        comando 2;
        …
        comando N;
        return valor_do_tipo_do_retorno;
}
```

9. Exemplos

9.1. Hello World

```
main() {
     print(“Hello World”)#
}
```

9.2. Sequência de Fibonacci

```
fib(n) {
 int a = 0#
 int b = 1#
 int tmp#
  if(n == 0) {
   print("0")#
 }
 else {
   let output = "0, 1"#
   while (n > 0) {
     n--#
     tmp = a + b#
     a = b#
     b = tmp#
     output = output ++ ", " ++ tmp#
   }
    print(output)#
 }
}
```

9.3. Shell Sort

```
shellSort(nums){
  int gap = 1#
  int n = length(nums)#
  while(gap < n) {
    gap = gap * 3 + 1#
  }
  gap = h / 3#
  while (gap > 0){
    repeater(int i, range(gap, n)) {
      int c = nums[i]#
      int j = i#
      while (j >= gap && nums[j - gap] > c) {
        nums[j] = nums[j - gap]#
        j = j - gap#
      }
      nums[j] = c#
    }
    gap = gap / 2#
  }
```

Especificação dos Tokens da linguagem de programação HI

1. Especificação da linguagem de implementação

Considerando os requisitos necessários para a implementação dos analisadores léxico e sintático da linguagem HI, a linguagem Java foi a selecionada para a implementação, uma vez que fornecesse recursos suficientes para que o desenvolvimento possa ser realizado.

2. Enumeração com as categorias dos tokens

public enum TokenCategory{

main, **id**, **tNone**, **tInt**, **tFloat**, **tChar**, **tString**, **tLogic**, **scopeBegin**, **scopeEnd**, **paramBegin**, **paramEnd**, **arrayBegin**, **arrayEnd**, **term**, **comma**, **semicolon**, **numIntConst**, **decNumConst**, **logicConst**, **charConst**, **stringConst**, **rwRead**, **rwPrint**, **rwIf**, **rwElse**, **rwRepeater**, **rwRange, rwWhile**, **rwReturn**, **attributiveOp**, **andLogicOp**, **orLogicOp**, **negLogicOp**, **addArithOp**, **multArithOp**, **expArithOp**, **unNegOp**, **eqRelOp**, **ineqRelOp**, **concOp**;

}

3. Expressões regulares

3.1 - Expressões regulares auxiliares

letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | t'' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

symbol = ' ' | '.' | ',' | ':' | ';' | '?' | '!' | '+' | '-' | '*' | '\' | '/' | '_' | '%' | '@' | '&' | '#' | '$' | '<' | '>' | '=' | '(' | ')' | '[' | ']' | '{' | '}' | '|' | '"' | '"' ;

boolean = 'true' | 'false'

dot = '\.'

sqm = '\''

qm = '\"'

3.2 - Lexemas

Main:

main = 'main';

Identificador:

id = '({letter}|_)({letter}|_|{digit}|_)*'

Tipos primitivos:

tNone = 'none';
tInt = 'int';
tFloat = 'float';
tBoolean = 'boolean';
tChar = 'chart';
tString = 'string';

Delimitadores

Escopo:

escBegin = '{';
escEnd = '}';

Parâmetros:

paramBegin = '(';
paramEnd = ')';

Array:

arrayBegin = '[';
arrayEnd = ']';

Comentários:

comment = '$';

Terminador:

term = '#';

Separadores:

comma = ',';
semicolon = ';';

Constantes de tipos:

numIntConst = '({digit})({digit})*'
decNumConst = '({digit})({digit})*({dot})({digit})({digit})*'
logicConst = '({boolean})'
charConst = '({sqm})({letter}|{digit}|{symbol})({sqm})'
constString = ' \".*\" '

Palavras reservadas de comando de entrada ou saída:

rwRead = 'read';
rwPrint = 'print';

Palavras reservadas de comando de iteração ou seleção:

rwIf = 'if';
rwElse = 'else';
rwRepeater = 'repeater';
rwWhile = 'while';

Palavra reservada de retorno de função:

rwReturn = 'return';

Palavra reservada de comando de contador:

rwRange = 'range';

Operador atributivo:

attributiveOp = '=';

Operadores lógicos:

andLogicOp = 'and';
orLogicOp = 'or';
negLogicOp = 'not';

Operadores aritméticos:

addArithOp = '+' | '-';
multArithOp = '*';
expArithOp = '^';

Operador unário:

unNegOp = '-';

Operadores relacionais:

ineqRelOp = '<' | '>' | '<=' | '>=';

eqRelOp = '==' | '~=';

Operador de concatenação:

concOp = '++';