UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

COMPILADORES - 2017.2

Hapais: especificação mínima da linguagem e expressões regulares

Autor 1
Dayvson SALES

Autor 2
Warley VITAL

May 31, 2018

Chapter 1

Especificação mínima da linguagem

Este documento tem como objetivo apresentar a especificação da linguagem Hapais, uma linguagem moderna, estaticamente tipada, bonita, sensível ao caso, com coerção explícita e de fácil aprendizado, desenvolvida com base na linguagem Elixir e C, seguindo a especificação da disciplina de Compiladores - IC/UFAL, semestre 2017.2.

Cada programa escrito em Hapais precisa estar dentro de um módulo, definido pela palavra reservada `defmod` <*nomeModulo*> `do...endmod`. A linguagem utiliza ponto e vírgula (`;`) para indicar a terminação de uma instrução, e terminadores `do...end`; e o `endmod` apenas para indicar fim do módulo. Obrigatoriamente, todo programa em Hapais deve conter a função *main*. Assim, a estrutura básica da linguagem é:

```
defmod <nome do modulo> do
    <variáveis globais>
    def <retorno> <nome da função> (<lista de parâmetros>) do
      <declaração de variáveis/instruções>
        return <valor>;
    end

    def void main() do
      <instruções>
    end
endmod
```

Não é possível adicionar módulo dentro de módulo.

Comentários são definidos por # e toda linha seguida antes da primeira quebra de linha será comentada e ignorada da avaliação.

A linguagem possui coerção implícita em algumas partes.

1.1 Nomes, vinculações, escopos e tipos de dados

1.1.1 Nomes

Aqui trataremos da especificação dos nomes na linguagem Hapais.

Os nomes possíveis começam por `letras` seguidas de letras ou dígitos. Por exemplo, `haPa1s` é um nome válido, enquanto que `1haPais` não é.

Os nomes são limitados a 32 caracteres.

Palavras reservadas

As seguintes palavras na Tabela 1.1 são reservadas e não podem, de maneira alguma, serem utilizadas como identificadores ou nomes.

KEYWORD	AÇÃO
void	Definição de tipo de retorno vazio
int	Definição de tipo inteiro
real	Definição de tipo real
char	Definição de tipo caractere
bool	Definição de tipo lógico
true	Valor lógico
false	Valor lógico
str	Definição de tipo cadeira de caracteres
defmod	Definição de módulo
endmod	Terminador para definição de módulo
def	Definição de estruturas
end	Terminador para definição de estruturas
do	Terminador para definição de estruturas
return	Definição do retorno de uma função
when	Definição de estrutura condicional
otherwise	Definição de estrutura condicional com segunda via
until	Def. de estrutura repetitiva com controle lógico
rep	Def. de estrutura repetitiva controlada por contador
print	Instrução para imprimir informações na saída
read	Instrução para receber informações da entrada
main	Função principal de um programa ou módulo

Table 1.1: Palavras reservadas

1.1.2 Tipos de dados

Os tipos válidos em Hapais são:

- Inteiro de 16 bits definido pela palavra-chave `int`;
- Ponto flutuante de 32 bits definido por `real`;
- Caractere definido por `char` seguindo a tabela ASCII;
- Booleano definido por `bool`, assumindo `true` ou `false`;
- String (cadeia de caracteres, máximo de 2$\hat{1}$6 = 65.536 caracteres) definida por `str`;
- Arrays unidimensionais definidos pelo tipo seguindo de colchetes (i.e., `int[]`).

No caso de arrays, podemos especificar o tamanho dentro dos colchetes, ou definir um array sem tamanho fixo (que por padrão começa com 10) omitindo valor dentro dos colchetes. Caso seja passado valor, este deve ser positivo maior que zero.

void

O void é utilizado para indicar que uma função não irá ter retorno.

Constantes literais

Todos os tipos definidos possuem constantes literais.

Um inteiro é um tipo de 16 bits que aceita dígitos entre 0 a 9, que possuem o sinal de menos (-) caso sejam negativo. Exemplos:

```
1 int a = 12;
2 int b = -12;
```

Um literal booleano assume os valores `true` ou `false`. Caracteres literais são válidos entre aspas simples, e uma literal string pode ter qualquer símbolo dentro de aspas duplas, como em: `str haPais = "haPais"`. Ponto flutuante possuem 32 bits e deve-se utilizar o ponto (.) para indicar as casas decimais após os dígitos. Exemplo: `real var = 12.4;`

Um array pode ser inicializado; seus valores devem ser colocados dentro de colchetes e devem seguir, unicamente, o tipo definido no array. Caso especificado o tamanho, deve conter, obrigatoriamente, esses espaços definidos. Exemplo: `int[4] a = [1, 2, 3, 4];`

Existem também as constantes `EOF` e `nil` que indicam fim de arquivo e valor nulo, respectivamente. Ambas não podem ser usadas do lado esquerdo de uma expressão.

A constante literal do tipo bool assume valores `true` ou `false`, representando verdadeiro ou falso, respectivamente.

Valores padrão

Na Tabela 1.2 é indicado quais os valores cada tipo tem na declaração de uma variável.

TIPO	VALOR
int	inicializado com o valor 0 (zero)
real	inicializado com o valor 0.0 (zero)
char	inicializado com o valor ’ (vazio no ASCII)
str	inicializado com o valor "" (palavra vazia)
bool	inicializado com o valor **true** (verdadeiro)
array	inicializado com o valor padrão do seu tipo

Table 1.2: Valores padrões

Operações com tipos

A Tabela 1.3 mostra as operações para cada tipo.

TIPO	OPERAÇÃO
inteiro	Atribuição, aritmética, relacional
real	Atribuição, aritmética, relacional
char	Atribuição, concatenação
bool	Atribuição, lógico, relacional
str	Atribuição, concatenação, relacional
array	Atribuição, concatenação de mesmo tipo

Table 1.3: Tipos e suas operações

O operador `$` é utilizado para concatenação e gera uma cadeia de caracteres, realizando uma coerção implícita quando for uma concatenação entre cadeia de caracteres e um tipo numérico ou booleano. Caso seja feita uma concatenação entre dois vetores unidimensionais de mesmo tipo, será gerado um novo array contendo a união entre os dois. Caso o tipo seja diferente ou não seja um array, um erro em execução será lançado (a coerção não será utilizada neste caso). Alguns dos casos comentados são mostrados na Tabela 1.4.

Entrada	Saída
`"a" $ 1`	`"a1"`
`"a" $ true`	`"atrue"`
`[1,2,3,4] $ [1,2]`	`[1,2,3,4,1,2]`
`[1,2] $ true`	**erro** em tempo de compilação
`[1,2] $ [true, true]`	**erro** em tempo de compilação

Table 1.4: Tipos e concatenações

Coerção de tipo

Salvo os casos mostrados na subseção 1.1.2, a linguagem permite apenas a coerção explícita, que é feita usando do tipo desejado entre parênteses seguido do valor desejado, como em `int a = int(3.14)`, onde `a` assume o valor 3. É possível fazer o *casting* (coerção) de tipos menores para maiores e vice versa; isto é, de `int` para `real`, ou o contrário como mostrado anteriormente. Uma atribuição `int z = 3.14` resulta em erro de compilação.

1.1.3 Conjunto de operadores

Aritmético

- + : Soma de operandos
- - : Subtração de operandos
- * : Multiplicação de operandos
- / : Divisão de operandos
- - : Negação de valores int ou real (unário)

Relacional

- > : Maior que
- >= : Maior ou igual que
- < : Menor que
- <= : Menor ou igual que
- == : Igualdade entre operandos
- != : Diferença entre operandos

Lógico

- ~: Negação
- && : Conjunção
- || : Disjunção

A Tabela 1.5 mostra a ordem de precedência (da maior para menor) e a associatividade de cada operador.

OPERADOR	ASSOCIATIVIDADE
- ~	Unária
* /	Esquerda para direita
+ -	Esquerda para direita
$	Direita para esquerda
< > <= >=	Esquerda para direita
== !=	Esquerda para direita
&&	Esquerda para direita
||	Esquerda para direita

Table 1.5: Operadores e sua associatividade

Se uma expressão possui parênteses, será avaliado primeiro o que estiver dentro dos parênteses.

1.1.4 Vinculações e Inferência de Tipo

Hapais é uma linguagem fortemente tipada, a inferência ocorrerá em tempo de compilação e sua vinculação é estática.

1.1.5 Escopo

Em Hapais o escopo é léxico, o que significa que tudo que for definido dentro de um escopo ficará visível para os escopos que a contém. Observe o exemplo:

```
1
2  defmod Teste do
3    int c;
4    def void main() do
5        int a;
6          a = 10;
7          when (a == 10) do
8            int b = 10;
9            print("a == 10");
10             print(b);
11             c = a + b;
12             print(c);
13         end
14     end
15 endmod
```

Nesse exemplo, a variável C ficará no escopo do módulo, que é visível pelo escopo da função main. A variável `int a` e `int c` estarão visíveis no escopo do condicional `when` e assim poderão ser utilizadas, enquanto `int b` será visível apenas em `when`.

O tempo de vida de uma variável definida será até o fim do escopo que a definiu.

1.1.6 Ambiente de Referenciamento

Como Hapais utiliza o escopo léxico, todas as variáveis definidas em um escopo, estão disponíveis em todos os outros escopos que contêm. Qualquer variável em um escopo aninhado cujo nome coincide com uma variável do alcance circundante irá somar essa variável externa. Em outras palavras, a variável dentro do escopo aninhado esconde temporariamente a variável do alcance circundante, mas não a afeta de forma alguma.

1.2 Expressões e sentenças de atribuição, estruturas de controle de nível de declaração

1.2.1 Atribuição

A linguagem Hapais utiliza o símbolo = para indicar uma atribuição. O elemento do lado esquerdo indica a variável a ser atribuída e no lado direito indica o valor. É necessário verificar o tipo da variável para atribuir o valor, do contrário, ocorrerá erro de compilação ou execução. A atribuição será uma instrução. Exemplo:

```
1 int a;
2 a = 10;
3 # ou
4 int a = 10;
```

1.2.2 Instruções condicionais e iterativas

Instrução condicional de uma via

A linguagem utiliza o termo `when` para indicar um condicional de uma via. Aqui é necessário ter operações lógicas ou alguma variável booleana; pode-se utilizar grupos, combinando com os operadores lógicos (`&&`, `||`, `~`).

De forma geral, o `when` tem a forma:

```
when (<expressão>) do
    <corpo>
end
```

Exemplo:

```
when (i < 10) do
    print("i menor que 10");
end
```

Instrução condicional de duas vias

Assim como em uma via, o `when` pode ser utilizado como duas vias, combinado com a palavra reservada `otherwise`. De forma geral, temos:

```
when (<expressão>) do
    <corpo>
otherwise
    <corpo>
end
```

Exemplo:

```
when (i < 10) do
    print("i menor que 10");
otherwise
    print("i maior que 10");
end
```

Instrução iterativa com controle lógico

Para instruções iterativas de controle lógico, Hapais possui a palavra-chave `until`, que recebe uma expressão lógica e executa o código até esta expressão se tornar falsa. De forma geral, temos:

```
until (<expressão>) do
    <corpo>
end
```

É possível usar de um controle pós-teste para a instrução de iteração `until` ao inverter a ordem dos terminadores, como é mostrado no exemplo abaixo:

```
do
    <corpo>
until (<expressão>) end
```

Desse modo, sempre haverá pelo menos uma iteração para laços do tipo.
Exemplo com pré-teste:

```
until(i > 10) do
    print("i continua maior que 10");
    i = i - 1;
end
```

Exemplo com pós-teste:

```
i = 10
do
    print(i + " ");
    i = i - 1;
until (i != 0) end
# saída : 10 9 8 7 6 5 4 3 2 1
```

Instrução iterativa controlada por contador

Diferente do `while`, comummente usado em outras linguagens, temos o `rep`, que pode ser controlada por um contador com ou sem passo. Caso o último argumento (passo) seja omitido, por padrão, teremos execução com passo 1 (um). De forma geral, temos:

```
rep (<variável de controle>, <expressão lógica>, <passo>) do
    <corpo>
end
```

Exemplo 1:

```
rep(i : int = 0, i < len(valores), i = i + 1) do
    print(arr[i]);
end
```

Exemplo 2:

```
rep(i : int = 0, i < len(valores)) do
    print(arr[i]);
end
```

1.2.3 Forma de controle

Infelizmente, não há operadores como break e continue como em C para controlar a estrutura de repetição. Também não há instrução de jump como o goto.

1.3 Entrada

Como toda linguagem, temos comandos de entrada e saída de dados. Para entrada, utilizamos o comando `read`. O `read` pode receber uma lista de variáveis, que irá cada uma ser associada ao valor lido ou pode-se omitir, sendo assim lido um valor apenas, por padrão.

```
read(<(opcional) variaveis>);
```

Por exemplo:

```
int a;
real b;

read(a, b); # associariamos o valor lido para a e depois a b
```

Ou poderiamos fazer:

```
int a = read();
real b = read();
```

O último valor lido pela `read` fica disponível em uma função chamada `lastValueRead()`.

1.4 Saída

Para apresentação dos dados ao usuário, utilizamos o comando `print`. Este é um comando com muitas utilidades e formatos, aceita formatação, permitindo mais de uma variável/constante literal. Por exemplo:

```
  int a, b, c;
  print(a, b, c);
  print("variavel a: " + a + "variavel b: " + b + "variável c: " + c);
```

É possível formatar o tamanho do campo, utilizando a expressão \%<tamanho> antes da concatenação da variável. Por exemplo:

```
print("%2" + a);
```

Para pontos flutuantes esta formatação se torna para as casas decimais, e caso omitido, será duas casas por padrão.

1.5 Funções

Para definir funções usa-se a seguinte nomenclatura:

```
def <tipo_retorno> <nome_funcao> (<lista_parametros>) do
    <corpo>
    return <tipo_retorno> # em caso de tipo void void, não se usa return
end
```

A lista de parâmetro deve especificar qual tipo de parâmetro será passado e sempre é necessário especificar a quantidade de parâmetro, não sendo possível passar múltiplos parâmetros como argumento (igual a Java, com os ..., por exemplo). Exemplos de funções e da linguagem como um todo são mostrados abaixo.

1.6 Exemplos

Primeiro exemplo temos o clássico **Alô, mundo!**.

```
# Alô mundo!

defmod AloMundo do
  def void main() do
    print("Alô mundo!");
  end
endmod
```

Em seguida é mostrado como a série Fibonnaci pode ser implementada para um dado `limite`.

```
# Fibonacci
defmod Fibonacci do
  def void fib(limite : int) do
    int[limite+1] arr;
    arr[0] = 0;
    arr[1] = 1;
    int i = 0;

  when(limite <= 0) do
      print("0");
    end

    until(i < limite) do
      when (i > 2) do
```

```
        arr[i] = arr[i-1] + arr[i-2];
      end

      print(arr[i]);

      when(i != limite-1) do
        print(", ");
      end

      i = i + 1;
    end
  end


  def void main() do
    int limite = read();
    fib(limite);
  end

endmod
```

Por fim, mostramos como a técnica *shell sort* pode ser implementada.

```
# Shellsort

defmod Shellsort do

  def int[] shellsort(valores : int[]) do

    int n = len(valores); # retorna tamanho do array valores, aqui nao eh comando
    para os tokens
    int h = 1;

    until(h < n) do
      h = h * 3 + 1;
    end

    h = h / 3;

    int c;
    int j;

    until(n > 2) do

      rep (i : int = h, i < n, i = i + 1) do
        c = valores[i];
        j = i;

        until(j >= h && valores[j - h] > c) do
                valores[j] = valores[j - h];
                j = j - h;
        end

        valores[j] = c;
      end

      h = h / 2;

    end

    return valores;
  end

  def void main() do

    int[] valores;

    until(read() != EOF) do
      valores[] = lastValueRead();
    end

    int[] arr = shellsort(valores);
```

```
49
50      rep(i : int = 0, i < len(valores), i = i + 1) do
51        print(arr[i]);
52        print("\n");
53      end
54
55    end
56
57  endmod
```

Chapter 2

Especificação dos tokens

A linguagem de programação que irá ser utilizada para produção dos analisadores sintático e léxico será Java, na versão 8.

2.1 Tokens

Utilizando a enumeração do Java, temos as seguintes categorias:

```
1
2  public enum CategoriaToken {
3
4      TK_ID,
5      TK_COMMENT,
6
7      TK_DEFMOD,
8      TK_DEF,
9      TK_DO,
10     TK_END,
11     TK_ENDMOD,
12
13     TK_UNTIL,
14     TK_REP,
15     TK_WHEN,
16     TK_RETURN,
17     TK_OTRWISE,
18
19     TK_VOID,
20     TK_INT,
21     TK_REAL,
22     TK_STR,
23     TK_BOOL,
24     TK_CHAR,
25
26     TK_CTEINT,
27     TK_CTEREAL,
28     TK_CTESTR,
29     TK_CTECHAR,
30
31     TK_SPTOR,
32     TK_DPTS,
33     TK_PVGL,
34     TK_READ,
35     TK_PRINT,
36     TK_LTVREAD,
37
38     TK_ABPAR,
39     TK_FCPAR,
40     TK_ABCOL,
41     TK_FCCOL,
42     TK_TRUE,
43     TK_FALSE,
44     TK_NIL,
45     TK_OPA,
46     TK_OPM,
```

```
47      TK_ATR,
48      TK_REL,
49      TK_REL2,
50      TK_AND,
51      TK_OR,
52      TK_NOT,
53
54      TK_CONCAT,
55
56      TK_EOF
57  }
```

2.2 Descrição dos tokens

A Tabela 2.1 abaixo mostra a descrição de cada token.

TK_ID	Token para identificador
TK_DEFMOD	Definir o início de um módulo
TK_DEF	Definir uma função
TK_DO	Primeiro terminador
TK_END	Terminador final
TK_ENDMOD	Indicar fim do módulo
TK_UNTIL	Para a estrutura de repetição until
TK_REP	Para a estrutura de repetição rep
TK_WHEN	Para o condicional when
TK_RETURN	Para o comando de retorno
TK_OTRWISE	Caso contrário (otherwise)
TK_VOID	Tipo void
TK_INT	Tipo int
TK_REAL	Tipo real
TK_STR	Tipo str
TK_BOOL	Tipo bool
TK_CHAR	Tipo char
TK_CTEINT	Constante inteira
TK_CTEREAL	Constante real
TK_CTESTR	Constante cadeira de caractere
TK_CTECHAR	Constante caractere
TK_SPTOR	Separador da repetição (,)
TK_DPTS	Dois pontos
TK_PVGL	Ponto e vírgula
TK_READ	Comando de leitura read
TK_PRINT	Comando de escrita print
TK_LTVREAD	Comando para receber o último valor lido (lastValueRead)
TK_ABPAR	Abre parênteses
TK_FCPAR	Fecha parênteses
TK_ABCOL	Abre colchetes
TK_FCCOL	Fecha colchetes
TK_TRUE	Constante booleana true
TK_FALSE	Constante booleana false
TK_NIL	Indicar valor nulo
TK_OPA	Operador aritmético
TK_OPM	Operador multiplicativo
TK_OPU	Operador unário
TK_ATR	Atribuição
TK_REL1	Relacionais (maior, menor, maior igual, menor igual)
TK_REL2	Relacionais (igual, diferente)
TK_AND	Operador lógico AND
TK_OR	Operador lógico OR
TK_NOT	Operador lógico NOT
TK_CONCAT	Concatenação
TK_EOF	Fim de arquivo

Table 2.1: Tabela das expressões regulares

2.3 Expressões regulares auxiliares

A tabela 2.2 mostra a relação das expressões auxiliares juntamente com sua descrição.

d = ['0'-'9']	Expressão de dígitos de 0 a 9
nZ = ['1'-'9']	Expressão de dígitos de 1 a 9
leMin = ['a'-'z']	Expressão de letras minúsculas de a-z
leMa = ['A'-'Z']	Expressão de letras minúsculas de A-Z
simb = ' ' | '_' | '.' | ',' | ':' | ';' | '!'	Expressão para os símbolos
simb = '+' | '-' | '*'| '?' | '<' | '>' | '='	Expressão para os símbolos (cont...)
simb = '(' | ')' | '[' | ']' | '{' | '}'	Expressão para os símbolos (cont...)
simb = ''' | '"' | '@' | '%' | '^'	Expressão para os símbolos (cont...)
spcm = (' '|#)	Expressão para os espaços e comentários

Table 2.2: Nomes das expressões e expressões auxiliares

2.4 Expressões para os lexemas

A tabela 2.3 mostra a relação dos tokens juntamente com sua expressão regular.

TK_ID	`(leMin|leMa)(leMin|leMa|d)*`
TK_DEFMOD	`'defmod'`
TK_DEF	`'def'`
TK_DO	`'do'`
TK_END	`'end'`
TK_ENDMOD	`'endmod'`
TK_UNTIL	`'until'`
TK_REP	`'rep'`
TK_WHEN	`'when'`
TK_RETURN	`'return'`
TK_OTRWISE	`'otherwise'`
TK_VOID	`'void'`
TK_INT	`'int'`
TK_REAL	`'real'`
TK_STR	`'str'`
TK_BOOL	`'bool'`
TK_CHAR	`'char'`
TK_CTEINT	`(nZd*|d)`
TK_CTEREAL	`(((nZd*|d)'.'d+)|(nZd*|d)+'.')`
TK_CTESTR	`(d|leMa|leMin|simb)*`
TK_CTECHAR	`(d|leMa|leMin|simb)`
TK_SPTOR	`','`
TK_DPTS	`':'`
TK_PVGL	`';'`
TK_READ	`'read'`
TK_PRINT	`'print'`
TK_LTVREAD	`'lastValueRead'`
TK_ABPAR	`'('`
TK_FCPAR	`')'`
TK_ABCOL	`'['`
TK_FCCOL	`']'`
TK_TRUE	`'true'`
TK_FALSE	`'false'`
TK_NIL	`'nil'`
TK_OPA	`('+'|'-')`
TK_OPM	`('*'|'\')`
TK_OPU	`-`
TK_ATR	`'='`
TK_REL1	`('<'|'>'|'<='|'>=')`
TK_REL2	`('=='|'!=')`
TK_AND	`'&&'`
TK_OR	`'||'`
TK_NOT	`'~'`
TK_CONCAT	`'$'`
TK_EOF	'EOF'

Table 2.3: Tokens e suas respectivas expressões regulares