Especificação mínima da linguagem

1 - Características Gerais

Um programa escrito na linguagem é composto por um único módulo, devendo conter um função nomeada com a palavra-chave *main*, cujo é a principal função do programa, responsável por iniciar a execução do mesmo. Esta função deve retornar um inteiro, 0 quando executada com sucesso, ou 1 quando ocorre algum erro na execução. Abaixo é exemplificado a estrutura geral de um programa escrito na linguagem:

```
# Exemplo de estrutura geral
module <nome do módulo> {
        <declaração de variáveis>
        <declaração de funções>

        fun int main() {
                <intruções>;
                return 0;
        }
}
```

A delimitação de escopo na linguagem é feita com os símbolos chave aberta e fechada, ({) e (}), usados para iniciar e encerrar o escopo, respectivamente. Além disso, comentários são definidos usando o símbolo hash (#). Por fim, é usado ponto e vírgula (;) para indicar o término de uma instrução dentro de um módulo.

1.1 - Nomes

Os nomes que podem ser definidos de acordo com a linguagem iniciam por caractere, seguido de caractere ou dígitos, e vice-versa. Um exemplo seria *n0me*. Um exemplo cujo não seria aceito pela linguagem seria *1lha*. Os nomes são limitados a 32 caracteres e são *case-sensitive*, não aceitando caracteres especiais. Os nomes podem ser usados para nomear variáveis e funções, sendo chamados de identificadores.

❖ 1.1.1 - Palavras reservadas

Palavra-chave	Ação
empty	Definição de tipo vazio
int	Definição de tipo inteiro
float	Definição de tipo ponto flutuante
char	Definição de tipo caractere
string	Definição de tipo para string
bool	Definição de tipo lógico
true	Valor lógico para “verdadeiro”
false	Valor lógico para “falso”
null	Definição de valor inicial nulo
if	Definição de estrutura condicional
elif	Definição de estrutura condicional mais abrangente
else	Definição de estrutura condicional com segunda via
while	Definição de estrutura de repetição com controle lógico
repeater	Definição de estrutura de repetição controlada por contador
module	Definição de módulo
fun	Definição de funções
return	Retorno de uma função
main	Definição da principal função do programa
read	Instrução para receber informações de entrada
show	Instrução para imprimir informações de saída

1.2 - Tipos de dados

Na linguagem, a vinculação é estática e fortemente tipada. Ou seja, quando é dado um tipo a uma variável na declaração, este tipo fica vinculado durante toda a sua vida.

- Inteiro: tipo de 32 bits definido pela palavra-chave ***int***;
- Ponto flutuante: tipo de 32 bits definido pela palavra-chave ***float***;
- Caractere: definido pela palavra-chave ***char*** com base na tabela ASCII;

- String: cadeia de caracteres definida pela palavra-chave ***string***, suportando 256 caracteres;
- Booleano: definido pela palavra-chave ***bool***, assumindo valores true ou false;
- Array unidimensional: definido pelo tipo seguido do nome da variável.

No caso de arrays, podemos especificar o tamanho dentro dos colchetes, sempre positivo maior que zero, ou definir um array sem tamanho fixo, omitindo valor dentro dos colchetes.

1.2.1 - Constantes literais

- Inteiro: um literal inteiro aceita dígitos de 0 a 9, podendo ter o sinal de negativo antes (-);
- Ponto flutuante: as condições de literais inteiros se aplicam, sendo que deve-se utilizar o ponto (.) para indicar as casas decimais após os dígitos;
- Caractere: um caractere literal é válido entre aspas simples;
- String: uma string literal pode ter qualquer símbolo dentro de aspas duplas;
- Booleano: um literal booleano assume os valores *true* ou *false*;
- Array: num array literal seus valores devem ser colocados dentro de colchetes e devem seguir, unicamente, o tipo definido no array. Caso especificado o tamanho, deve ser inicializado com a quantidade de dados equivalente.

Nos casos onde queira inicializar um tipo com valor nulo, usasse a constante *null*.

1.2.2 - Valores padrão

- Inteiro: inicializado com valor 0 (zero);
- Ponto flutuante: inicializado com valor 0.0 (zero);
- Caractere: inicializado com o valor ‘ (vazio no ASCII);
- String: inicializado com o valor “” (palavra vazia);
- Booleano: inicializado com o valor *false* (falso);
- Array: inicializado com o valor padrão do tipo usado.

1.2.3 - Operações com tipos

- Inteiro: atribuição, aritmética, relacional;
- Ponto flutuante: atribuição, aritmética, relacional;
- Caractere: atribuição, concatenação;
- String: atribuição, concatenação, relacional;
- Booleano: atribuição, lógico;
- Array: atribuição, concatenação de mesmo tipo.

❖ 1.2.4 - Coerção

Na linguagem a coerção acontece de forma automática entre os tipos *int* e *float*. Na situação contrária, apenas a parte inteira é atribuída a variável.

1.3 - Conjunto de operadores

❖ 1.3.1 - Aritmético

- ➢ +: soma de operandos;
- ➢ -: subtração de operandos;
- ➢ *: multiplicação de operandos;
- ➢ /: divisão de operandos;
- ➢ ^: exponenciação de operandos;
- ➢ ~ : negação de valores int ou float (unário).

❖ 1.3.2 - Relacional

- ➢ >: maior que;
- ➢ <: menor que;
- ➢ >=: maior ou igual que;
- ➢ <=: menor ou igual que;
- ➢ ==: igualdade entre operandos;
- ➢ !=: diferença entre operandos.

❖ 1.3.3 - Lógico

- ➢ ~: negação;
- ➢ &&: conjunção (“e” lógico);
- ➢ ||: disjunção (“ou” lógico).

❖ 1.3.4 - Concatenação

- ➢ ++: concatenação.

Na tabela abaixo, temos a ordem de precedência (crescente) e associatividade dos operadores acima. Se uma expressão possui parênteses, será avaliado primeiro o que estiver dentro dos parênteses.

Operador	Associatividade
^	Direita para a esquerda
- (unário negativo) ~	Direita para a esquerda
* /	Esquerda para direita
+ -	Esquerda para direita
++	Direita para esquerda
< > <= >=	Esquerda para direita

== !=	Esquerda para direita
&&	Esquerda para direita
||	Esquerda para direita
=	Direita para esquerda

1.4 - Escopo

- Na linguagem, o escopo é léxico, o que significa que tudo que for definido dentro de um escopo ficará visível para os escopos que a contém. O tempo de vida de uma variável definida será até o fim do escopo que a definiu.

1.5 - Ambiente de referenciamento

- Como o escopo é léxico, todas as variáveis definidas em um escopo, estão disponíveis em todos os outros escopos que contêm.

2 - Instruções

2.1 - Atribuição

As atribuições são feitas através de um comando utilizando o operador =. Do lado esquerdo fica a variável e do lado direito o valor a ser atribuído a ela. As atribuições só poderão ser feitas para tipos iguais, caso o tipo de valor a ser atribuído à variável seja diferente do tipo inicialmente dado a mesma, ocorrerá um erro de compilação.

Exemplo de atribuição: int a = 5;

2.2 - Instruções condicionais e iterativas

Nessas instruções, as condições avaliadas sempre terão um resultado booleano, indicando qual direcionamento o programa deve ter.

- **2.2.1 - Instrução condicional de uma via**

Para realizar uma instrução condicional de uma via a linguagem usa a palavra-chave *if* seguida de parênteses, com a condição desejada dentro dos parênteses. Pode haver mais de uma condição envolvida, para isso é necessário o uso dos operadores lógicos para combinar as expressões. De forma geral, a instrução condicional de uma via tem a seguinte forma:

```
if (<condição>) {
        <instruções>;
}
```

Exemplo:

```
if (i < 5) {
        print(“i menor que 5);
}
```

❖ 2.2.2 - Instrução condicional de duas vias

Seguindo a mesma lógica do tópico anterior, podemos usar instrução condicional de duas vias utilizando a palavra-chave *if* com a palavra-chave *else* em seguida. De forma Geral, teremos uma instrução condicional de duas vias tem a seguinte forma:

```
if (<condição>) {
        <instruções>;
} else {
        <instruções>;
}
```

Exemplo:

```
if (i < 5) {
        print(“i menor que 5);
} else {
        print(“i maior ou igual a 5);
}
```

❖ 2.2.3 - Instrução condicional mais abrangente

Caso a verificação precise ser mais abrangente ou exija várias validações, podemos usar a palavra-chave *elif*, e aninhar as validações a serem feitas. De forma geral, teremos uma instrução condicional mais abrangente da seguinte forma:

```
if (<condição>) {
        <instruções>;
} elif (<condição>) {
        <instruções>;
} else {
        <instruções>;
}
```

Exemplo:

```
if (i < 5) {
        print(“i menor que 5);
} elif (i > 5) {
        print(“i maior ou igual a 5);
} else {
        print(“i é igual a 5);
}
```

- **2.2.4 - Instrução iterativa com controle lógico**

Para instrução iterativa com controle lógico, usamos a palavra chave *while*, que recebe uma expressão lógica e a executa até ela tornar-se falsa. De forma geral, uma instrução iterativa com controle lógico tem a seguinte forma:

```
while (<condição>) {
        <instruções>;
}
```

Exemplo:

```
while (i < 5) {
        print(“i menor que 5”);
        i = i + 1;
}
```

- **2.2.4 - Instrução iterativa controlada por contador**

Para instrução iterativa controlada por contador, utilizamos a palavra-chave *repeater*, seguida de parênteses que contém a condição e o contador. De forma geral, uma instrução iterativa controlada por contador tem a seguinte forma:

```
repeater (<contador>; <limite>; <passo>) {
        <instruções>;
}
```

Exemplo:

```
int i;
repeater (i = 0; i < 5; i + 1) {
        print(“i menor que 5);
}
```

2.3 - Forma de controle

Não há operadores como *break* e *continue* como na linguagem C, para controlar a estrutura de repetição.

3 - Entrada

Para entrada de dados utilizamos a palavra-chave *read*. Uma instrução de entrada de dados tem a seguinte forma:

```
read(<variáveis>);
```

Exemplo:

```
int a;
float b;

...

read(a, b);
```

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

4 - Saída

Para saída de dados utilizamos a palavra-chave *show*. A instrução de saída, deve conter como parâmetro um string literal, ou uma variável, ou ainda conter uma variável e um string literal, e vice-versa. Para exibir um valor de uma ou mais variáveis com uma string literal, pode-se formatar a saída especificando o tipo de dado esperado na string literal. Usamos %d para *int*, %f para *float*, %c para *char* e %s para *string*. No caso do tipo *float*, pode-se dizer quantas casa decimais após o ponto devem ser exibidas, usando %._f, onde em _ deve-se colocar a quantidade desejada. Também há a opção de usar o operador de concatenação no caso de *char* e *string*. Abaixo temos alguns exemplos:

```
show(<variáveis> ou <string literal>);
```

Exemplo:

```
int a = 5;
float b = 1.5;
string c = “cadeira de caracteres”;
string d = “ é legal!”
...
show(a , b , c, d) ;
ou
show(“Esta é a mensagem de saída!”);
ou
show(“O valor da variável é: %d”, a);
ou
show(%.2f, b);
ou
show(c ++ d);
```

5 - Módulos

Um módulo é definido pela palavra-chave *module*. Assim, de forma geral, a definição de um módulo tem a seguinte forma:

```
module <nome_módulo> {
        <declaração de variáveis>
        <funções>
}
```

Exemplo:

```
module example {
        int a = 5;

        fun showValue() {
                show(a);
        }
}
```

6 - Funções

Na linguagem toda função deve ser declarada antes de ser usada, essa declaração tem que ser feita e implementada antes da função principal *main*. Não é possível criar funções aninhadas. Para definir funções usamos a palavra-chave *fun*. Quando uma função não tem tipo de retorno definido, então é assumido que não há retorno na função, não sendo necessário usar a palavra-chave *return* no fim da função. Numa função, os parâmetros são passados por referência. De forma geral, a definição de uma função tem o seguinte formato:

```
fun <tipo_retorno> <nome_função>(<(opcional) parâmetros>) {
        <instruções>;
        return <tipo_retorno>;
}
```

Exemplos:

```
module HelloWorld {
        fun int main() {
                show("Hello World =)");
                return 0;
        }
}
```

```
module Fibonacci {
        fun fib(int limit) {
                int [limit +1] array;
                array[ 0 ] = 0;
                array[ 1 ] = 1;
                int i = 0;

                when(limit <= 0) {
                        show("0");
                }

                if(i < limit) {
                        when(i > 2) {
                                array[ i ] = array[i -1] + array[i -2];
                        }
                        show(array[ i ]);

                        when(i != limit - 1) {
                                show(", ");
                        }
```

```
                        i = i + 1;
                }
        }

        fun int main() {
                int limit = read();
                fib(limit);
        }
}
```

```
module ShellSort {
        fun int[] shellSort(int[] values) {
                int c;
                int j;
                int n = length(values);
                int h = 1;

                if(h < n) {
                        h = h * 3 + 1;
                }
                h = h / 3;

                if(n > 2) {
                        repeater(int i = h, i < n, i + 1) {
                                c = values[i];
                                j = i;

                                if(j >= h && values[j - h] > c) {
                                        values[j] = values[j - h];
                                        j = j - h;
                                }

                                values[j] = c;
                        }

                        h = h / 2;
                }

                return values;
        }

        fun int main() {
                int[] values;

                if(read() ! = EOF) {
                        values[] = lastValueRead();
                }
```

```
                int[] array = shellsort(values);

                repeater(int i = 0, i < len(values), i + 1) {
                        show(array[i]);
                        show("\n");
                }
        }
}
```

Especificação dos tokens

A linguagem de programação que irá ser utilizada para produção dos analisadores sintático e léxico será o Java.

1 - Categorias dos Tokens

```
public enum TokensCategories {

        tID, tEMPTY, tINT, tFLOAT, tCHAR, tSTRING, tBOOL, tMODULE, tFUN,
tRETURN, tMAIN, tREAD, tSHOW, tIF, tELIF, tELSE, tWHILE, tREPEATER,
tTRUE, tFALSE, tCTEINT, tCTEFLOAT, tCTECHAR, tCTESTRING, tSPTR, tCO,
tSCO, tOP, tCP, tOB, tCB, tOPA, tOPM, tOPU, tATR, tREL1, tREL2, tAND, tOR,
tNOT, tCONCAT, tNULL;

}
```

2 - Descrição dos tokens

Token	Descrição	Token	Descrição
tID	token para um identificador;	tCTEFLOAT	token para uma constante float;
tEMPTY	token para o tipo vazio;	tCTECHAR	token para uma constante caractere;
tINT	token para o tipo int;	tCTESTRING	token para uma constante cadeia de caractere;
tFLOAT	token para o tipo float;	tSPTR	token para um separador;
tCHAR	token para o tipo char;	tCO	token para dois pontos;
tSTRING	token para o tipo string;	tSCO	token para ponto e vírgula;
tBOOL	token para o tipo booleano;	tOP	token para abertura de parênteses;

tMODULE	token para uma definição de módulo;	tCP	token para fechamento de parênteses;
tFUN	token para a definição de uma função;	tOB	token para abertura de colchetes;
tRETURN	token para a um retorno de uma função;	tCB	token para fechamento de colchetes;
tMAIN	token para a definição da função principal;	tOPA	token para operador aritmético;
tREAD	token para comando de leitura;	tOPM	token para operador multiplicativo;
tSHOW	token para comando de exibição de dados;	tOPU	token para um operador unário;
tIF	token para uma estrutura condicional de uma via;	tATR	token para uma atribuição;
tELIF	token para uma estrutura condicional mais abrangente;	tREL1	token para um operador relacional;
tELSE	token para uma estrutura condicional de duas vias;	tREL2	token para um operador relacional;
tWHILE	token para uma estrutura de repetição com controle lógico;	tAND	token para o operador lógico AND;
tREPEATER	token para uma estrutura de repetição controlada por contador;	tOR	token para o operador lógico OR;
tTRUE	token para constante boleana true;	tNOT	token para o operador lógico NOT;
tFALSE	token para constante boleana false;	tCONCAT	token para o operador de concatenação;
tCTEINT	token para uma constante inteira;	tNULL	token para definição de valor inicial nulo;

3 - Expressões regulares auxiliares

digit = ’0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ’9’;
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’;
symbol = ‘ ‘ | ‘.’ | ‘,’ | ‘:’ | ‘;’ | ‘!’ | ‘?’ | ‘+‘ | ‘-’ | ‘*‘ | ‘/’ | ‘\’ | ‘_’ | ‘<‘ | ‘>’ | ‘=’ | ‘(‘ | ‘)’ | ‘[‘ | ‘]’ | ‘{‘ | ‘}’ | ‘’’ | ‘“‘ | ‘@’ | ‘%’ | ‘&’ | ‘$’ | ‘^’ | ‘|’;

4 - Expressões para os lexemas

tID	‘\’[{letter}{letter|digit}*]‘’
tEMPTY	‘empty’
tINT	‘int’
tFLOAT	‘float’
tCHAR	‘char’
tSTRING	‘string’
tBOOL	‘bool’
tMODULE	‘module’
tFUN	‘fun’
tRETURN	‘return’
tMAIN	‘main’
tREAD	‘read’
tSHOW	‘show’
tIF	‘if’
tELIF	‘elif’
tELSE	‘else’
tWHEN	‘when’
tREPEATER	‘repeater’
tTRUE	‘true’
tFALSE	‘false’
tCTEINT	‘{‘digit’}*’
tCTEFLOAT	‘{‘digit’}*’.’{digit}+’
tCTECHAR	‘\’[{letter}{digit}{symbol}]?\‘’
tCTESTRING	‘\”{tCTECHAR}*\”'
tSPTR	‘,’

tCO	‘:’
tSCO	‘;’
tOP	‘(’
tCP	‘)’
tOB	‘[‘
tCB	‘]’
tOPA	‘+’ | ‘-’
tOPM	‘*’ | ‘/’
tOPE	‘^’
tOPU	‘-’
tATR	‘=’
tREL1	‘<’ | ‘>’ | ‘<=’ | ‘>=’
tREL2	‘==’ | ‘!=’
tAND	‘&&’
tOR	‘||’
tNOT	‘~’
tCONCAT	‘++’
tNULL	‘null’
comentário	‘#’