

---

# APS LÓGICA DA COMPUTAÇÃO

LINGUAGEM HUCA++



Hugo Ebert Carl



## DESCRIÇÃO

A linguagem HuCa++ foi criada com a finalidade principal de diminuir as linhas de código em um algoritmo tornando a compreensão mais simples e a vida do programador mais fácil para programas que exigem uma complexidade mais básica em aplicações menores. A ideia central é ser uma linguagem de programação modular que deve ser construída em blocos, porém possui uma certa flexibilidade pensando no conforto do programador. Huca++ é uma linguagem livre de contexto (LLC), que possui todas as funcionalidades básicas de uma linguagem de programação comum, condicionais, loops, funções, variáveis (sem tipagem).

# TOKENS DA LINGUAGEM

Pensando na praticidade do programador que programa em várias linguagens a estrutura da linguagem se assemelha bastante às linguagens mais usadas no mercado como C, Java ou Php. Para a fácil compreensão é possível fazer analogias aos tokens mais conhecidos entre os desenvolvedores.

- while = @
- if = ?
- else = ?.
- def, function = BLOCK ( Ao invés do uso de parênteses para blocos a linguagem usa < > )
- print() = show<>
- input() = read<>
- . = concat
- != not
- {exemplo} ==> exemplo <=
- <, >, <=, >= = less, greater, less\_eq, greater\_eq
- == = eqto
- ; = |
- \$var = \_var
- Alguns tokens comuns como return, +, -, \*, /, True, False, String, comentários (php) foram mantidos.

# EBNF

```
<digit> ::= "0" | "1" | ... | "9";

<int_num> ::= <digit>, {<digit>};

<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" | ... | "z";

<text> ::= <letter>, {<letter>};

<string> ::= "", <text>, "";

<bool_val> ::= true | false;

<input> ::= "read", "<", ">";

<declarator> ::= ("_", <text>)?, (",")?;

<variable_assign> ::= "_", <text>, "=", <relexp>, "|";

<program_statement> ::= "#INIT" , {<statement>}, "#END";

<compound_statement> ::= "=>", {<statement>}, "<=";

<statement> ::= <iteration_statement>
              | <condition_statement>
              | "|" finish statement
              | <compound_statement>
              | "return" <relexp>, "|"
              | <variable_assign>
              | <print_statement>
              | <function_definition>
              | <function_call>, "|";
```

```
<function_definition> ::= "BLOCK", <text>, "<", {<declarator>}, ">", <compound_statement>;

<function_call> ::= <text>, "<", {<declarator> | (relexp, (",")?)}, ">";

<iteration_statement> ::= "@", "<", {<relexp>}, ">", <compound_statement>;

<print_statement> ::= "show", <relexp>, "|";

<condition_statement> ::= "?", "<", {<relexp>}, ">", <compound_statement>, ("?.", <compound_statement>)?;

<relexp> ::= <exp> | <exp>, ("less" | "less_eq" | "greater" | "greater_eq" | "eqto"), <exp>;

<exp> ::= <term>, {("&+" | "- " | "or" | "concat"), <term>};

<term> ::= <factor>, {("*" | "/" | "and"), <factor>};

<factor> ::= <int_num> | ("+" | "- " | "not"), <factor> | "(", <exp>, ")" | <bool_val> | <input> | <string> | <function_call>;
```



# COMPILAÇÃO

Para compilar a linguagem foram feitas as etapas de pré-processamento, análise léxica, sintática e semântica. Para a análise léxica e sintática foi utilizada uma biblioteca em python chamada ply, entretanto para a compilação completa da HuCa++ foi usado o compilador de php construído ao longo do semestre, realizando todas as etapas da compilação seguindo a estrutura EBNF apresentada no slide anterior.

# DEMONSTRAÇÃO

```
#INIT  
  
_ss=read<>|show (6+2)/_ss|show 1+3+5/3| _var=4|  
  
#END|
```

O código simples acima pode ser programado em apenas uma linha para simplificar alguns tipos de código o programador tem a opção de dividir todas as ações por |.

# DEMONSTRAÇÃO

```
#INIT

BLOCK ola<_x> =>
  ? <_x eqto 0>=>
    return 1|
  <= ?. =>
    _y = ola<_x - 1> + 3|
    show _y||
    return _y|
  <=

show ola<3>|

#END
```

```
def ola(y):
    if y == 0:
        return 1
    else:
        z = ola(y-1) + 3
        print(ola(y-1) + 3)
        return z

print(ola(3))
```

Saída esperada:

```
4
4
7
4
4
7
10
10
```

O exemplo mostra o equivalente em python da criação de uma função resursiva (bloco limitado por => <=) Demonstrando que a linguagem pode funcionar de forma semelhante.



**FIM |**