Universidade Federal de Alagoas

Compiladores
C--

Paulo Bernardo
pbaf@ic.ufal.br

Ricardo Alves
ras@ic.ufal.br

Dezembro, 2019.2

Sumário

1 Introdução

A linguagem C-- tem como base as linguagens C e Python, C-- admite escopo global e não admite cast. Não possui coerção (conversão implicita de tipos) é estáticamente tipada e sua compatibilidade de tipos é feita por nome. C-- não é orientada a objetos e é case-sensitive.

2 Estrutura geral de um programa

Um programa escrito deve possuir:

- Todas as funcões devem ser declaradas antes de poderem ser utilizadas e o modo de passagem de parâmetros é por cópia. A declaração de uma função deve ser iniciada pelo seu tipo de retorno, seguido pelo seu identificador (começando em letra maiúscula), após isso a lista de parâmetros dever ser dada entre parênteses e seus itens separados por vírgula, seu corpo dever estar contido dentro de chaves. Cada item dos parâmetros deve conter apenas o tipo da variável e o seu respectivo identificador. Funções sem o **return** retornam o valor padrão de cada tipo.

  Exemplo:

  ```
  returnType Identifier(typeA paramA, typeB paramB,...) {
      ...
      code
      ...
  }
  ```

- Um ponto de início de execução que se dá pela função inicial **Main** com tipo de retorno obrigatório do tipo int.

  ```
  int Main() {
      ...
      code
      ...
      return 0;
  }
  ```

3 Tipos de dados e nomes

3.1 Definições de variáveis globais/locais

Variáveis globais devem ser declaradas fora de funções/procedimentos.

3.2 Palavras reservadas

Lista de palavras reservadas: *and*, *bool*, *char*, *during*, *else*, *false*, *float*, *from*, *get*, *if*, *increment*, *int*, *not*, *or*, *print*, *return*, *string*, *to*, *true*, *void*.

3.3 Identificadores

Tem o tamanho máximo de 31 caracteres e as seguintes regras:

- Todas as variáveis devem iniciar com letra minúscula e funções/procedimentos devem ser iniciadas com letra maiúscula.
- Os caracteres, a partir do segundo, podem ser letras, números ou underline.
- Não é permitido o uso de espaços.
- Não é permitido o uso de palavras reservadas como identificador.

3.4 Comentários

Apenas comentários por linha, identificados por '#', podendo ser introduzidos em qualquer posição da linha.

3.5 Inteiro

Deve ser identificado pela palavra reservada *int* e representa um número inteiro de 64 bits. Seus literais são uma sequência de números inteiros.
Exemplo:

```
int integer;
```

3.6 Ponto Flutuante

Deve ser identificado pela palavra reservada *float* e representa um número real de 64 bits. Seus literais são da forma:

```
[0-9]+\.([0-9]+)
```

Exemplo:

```
float floating_point;
```

3.7 Caractere

Deve ser identificado pela palavra reservada *char* e representa 1 byte que guarda um valor de 0 a 127 referente a seu símbolo na tabela ASCII. Ao atribuir um caractere à uma variável, o caractere deve estar contido dentro de apóstrofos. Ao representar o caractere (') deve-se utilizar o caractere de escape (\).

Exemplo:

```
char character;
char a = 'a';
char b = '\''
char c = '\\';
```

3.8 Arranjos Unidimensionais

Um vetor deve ter seu tipo seguido pelo seu tamanho especificado entre colchetes e logo após o seu identificador. Seus literais dependem do tipo utilizado e devem ser inclusos dentro de colchetes seperados por vírgula.
Exemplo:

```
<Type>[size] identifier;

<Type>[size] identifier = [<values>];

<Type>[size] Identifier(<parameters>) {}
```

Onde *size* é opcional quando for utilizada para definir o tipo de retorno de uma função ou como tipo de um parâmetro, mas quando utilizada deve ser uma variável inteira ou uma constante inteira.

Os elementos de um arranjo podem ser acessados da seguinte maneira:

```
identifier[position];
```

Se uma quantidade de elementos num arranjo for diferente da que foi declarada, duas coisas podem acontecer:

- Quantidade menor do que o declarado:
  - O restante do arranjo é preenchido com os valores padrões do tipo do arranjo.
- Quantidade maior do que o declarado:
  - Erro.

3.9 Cadeia de Caracteres

Deve ser identificado pela palavra reservada *string* e representa uma sequência de caracteres. Ao atribuir uma string à uma variável string, a string deve estar contida dentro de aspas.
Ao representar o caractere (") dentro da string deve-se utilizar o caractere de escape (\).
Exemplo:

```
string str;
string a = "string";
string b = "str \" ing";
```

3.10 Booleano

Deve ser identificado pela palavra reservada "bool"e representa somente 2 possíveis valores: true e false.
Exemplo:

```
bool boolean;
```

3.11 Declaração de variáveis

A declaração de variáveis deve seguir o seguinte formato:
Para inteiro, caractere, booleano, string e ponto flutuante.

```
<Type> identifierA;
<Type> indentifierB = value;
```

Para arranjos unidimensionais:

```
<Type>[size] identifierA;
<Type>[size] identifierB = [value1, ...];
```

3.12 Operaçoes de cada tipo

Tabela 1:

Tipo	Operações
int	Atribuição, Aritiméticos, Relacionais
float	Atribuição, Aritiméticos*, Relacionais
char	Atribuição, Relacionais, Concatenação
string	Atribuição, Relacionais, Concatenação
bool	Atribuição, Relacionais**, Lógicas

* operador de resto não incluso
** somente os operadores '==' e '!=' são aceitos.

3.13 Valores padrão

Tabela 2:

Tipo	Valor
int	0
float	0.0
char	' ' (0)
string	null
bool	false

4 Conjunto de Operadores

4.1 Aritimeticos

Tabela 3:

Operação	Símbolo
Soma	+
Subtração	-
Unário Negativo	-
Multiplicação	*
Divisão	/

4.2 Relacionais

Tabela 4:

Operação	Símbolo
Igual	==
Menor que	<
Maior que	>
Menor ou igual	<=
Maior ou igual	>=
Diferente	!=

4.3 Lógicos

Tabela 5:

Operação	Símbolo
Negação Unária	not
Conjunção	and
Disjunção	or

4.4 Concatenação

É formada pelo operador "&", que ao ser aplicado concatena os dois lados da operação se os dois lados forem cadeia de caracteres.
Exemplo:

```
string str = string1 & string2;
```

4.5 Tipo das operações

É definido de operador a operador a partir dos tipos dos operandos. Os operandos de uma operação devem obrigatoriamente ser do mesmo tipo.

4.6 Precedência e Associatividade

Quanto menor o valor, maior sua precedência.

Tabela 6:

Operador	Associatividade	Precedência
- (unário)	Direita	1
/, *	Esquerda	2
+, -	Esquerda	3
==, !=	Nenhuma	4
<, >, <=, >=	Nenhuma	5
not	Direita	6
and	Esquerda	7
or	Esquerda	8
&	Nenhuma	9

Parênteses podem alterar a precedência.

5 Instruções

5.1 Atribuição

Definida pelo operador '=', sendo o lado esquerdo o identificador a receber o valor e o direito o valor ou expressão a ser atribuído.
Os dois lados devem ser do mesmo tipo.

5.2 Estrutura condicional de uma e duas vias

Definida da seguinte forma:

```
if expressao_logica {
    ...
} else {
    ...
}
```

5.3 Estrutura iterativa com controle lógico

Definida da seguinte forma:

```
during expressao_logica {
    ...
```

```
}
```

5.4 Estrutura iterativa controlada por contador

Definida da seguinte forma:

```
from id = expressao_aritmetica to expressao_aritimetica increment expressao_aritmetica {
    ...
}
```

5.5 Entrada e saída

- Entrada: É feita por meio da instrução *get()* que recebe como argumentos as variáveis que irão receber a entrada. O tipo da entrada deve ser igual ao tipo da variável, caso não seja, ocorre erro.
  Exemplo:

```
<Type> identifierA;
<Type> identifierB;
get(identifierA, identifierB);
```

- Saída: É feita por meio da instrução *print()*. Ele recebe alguns argumentos, onde o primeiro é uma string podendo ter caracteres e a formatação da saída das variáveis passadas nos próximos argumentos, todos separados por vírgula.

  A formatação deve seguir algumas regras:

Tabela 7:

Tipo	Formatação	Formatação de Tamanho	*
int	$d	$*d	* Quantidade de dígitos que irão aparecer na saída
float	$f	$*f	* Quantidade de casas decimais
char	$c	–	–
bool	$b	–	–
string	$s	–	–

Exemplos:

```
print("$s", "Hello World");
print("$d - $d", 2, intVar);
print("$2f $4d", floatVar, intVar);
```

6 Programas Exemplos

6.1 Hello World

```
int Main() {

    print("Hello World");

    return 0;
}
```

6.2 Fibonacci

```
void Fibonacci(int value) {

    int a = 1;
    int b = 1;
    int next;

    if value == 0 {
        return;
    }

    during next <= value {

        print("$d, ", a);
        next = a + b;
        a = b;
        b = next;

    }

    print("$d\n", a);

    return;
}

int Main() {

    int value;

    get(value);

    Fibonacci(value);

    return 0;
}
```

6.3 Shell Sort

```
int[] ShellSort(int[] values, int size) {

    int i;
    int j;
    int num;
    int gap = 1;

    during gap < size {
        gap = gap * 3 + 1;
    }

    during gap > 0 {
        from i = gap to size - 1 increment 1 {
            num = values[i];
            j = i;
            int k = j - gap;
            during (j > gap - 1) and (num <= values[k]) {
                values[j] = values[k];
                j = j - gap;
                k = j;
            }
            values[j] = num;
        }
        gap = gap / 3;
    }

    return values;
}

int Main() {

    int i;
    print("Enter the length of array:");
    int size;
    get(size);
    int[size] values;
    print("Enter the array");

    from i = 0 to size - 1 increment 1 {
        get(values[i]);
        print("$d ", values[i]);
    }

    values = ShellSort(values, size);
```

```
    from i = 0 to size - 1 increment 1 {
        print("$d ", values[i]);
    }
}
```

7 Especificação da Linguagem de Programação

Os analisadores léxico e sintático da linguagem serão implementados em Java, utilizado o analisador descendente preditivo recursivo.

8 Especificação dos Tokens

8.1 Lista de Tokens

A lista de tokens é definida por:

```
public enum CategoryList {

    Tunknown, Tmain, Tint, Tfloat, Tstring, Tbool,
    Tchar, Tvoid, Tcomma, Tif, Telse, Tduring, Tfrom, Tto,
    Tincrement, TsemiCol, TbegBrac, TendBrac, TbegSqBrac,
    TendSqBrac, TbegCrBrac, TendCrBrac, Tprint, Tget, Treturn,
    TopConc, TopAnd, TopOr, TopNot, TopAtr, TopEq, TopDif,
    TopAdd, TopSub, TopDiv, TopMult, TopMod, TopLowThen,
    TopLowThnE, TopGreThen, TopGreThnE, TfuncId, TnameId,
    TcteInt, TcteFloat, TcteString, TcteBool, TcteChar, TEOF

}
```

9 Especificação das Expressões Regulares

9.1 Expressões Regulares Auxiliares

```
letter = ‘[a-zA-Z]’
letterLow = ‘[a-z]’
letterHig = ‘[A-Z]’
digit = ‘[0-9]’
symbol = ‘[ /\\!@#$%&*()_\-=+\[\]{}><?.;:,"’]’
```

9.2 Lexemas

```
Main:
    Tmain = ‘Main’
```

```
Identificador:
    TnameId = ‘(letterLow)(letter|digit|_)*{31}’
    TfuncId = ‘(letterHig)(letter|digit|_)*{31}’

Tipos Primitivos:
    Tint = ‘int’
    Tfloat = ‘float’
    Tchar = ‘char’
    Tstring = ‘string’
    Tbool = ‘bool’
    Tvoid = ‘void’

Delimitadores:

    Escopo:
        TbegCrBrac = ‘\{’
        TendCrBrac = ‘\}’

    Parâmetros:
        TbegBrac = ‘\(’
        TendBrac = ‘\)’

    Array:
        TbegSqBrac = ‘\[’
        TendSqBrac = ‘\]’

    Finalizador:
        TsemiCol = ‘;’

    Separador:
        Tcomma = ‘,’

Definiçoes de tipos:
    TcteInt      = ‘digit+’
    TcteFloat    = ‘digit+\.(digit+)?’
    TcteBool     = ‘(true|false)’
    TcteChar     = ‘’(letter|digit|symbol)’’
    TcteString   = ‘"(letter|digit|symbol)*"’

Palavras reservadas de fluxo:
    Tif = ‘if’
    Telse = ‘else’
    Tfrom = ‘from’
    Tduring = ‘during’
    Tto = ‘to’
    Tincrement = ‘increment’
    Treturn = ‘return’
```

```
Operadores lógicos:
    TopAnd = ‘and’
    TopOr = ‘or’
    TopNot = ‘not’

Operadores Aritiméticos:
    TopAdd = ‘(+)’
    TopSub = ‘(-)’
    TopMult = ‘(*)’
    TopDiv = ‘(/)’
    TopMod = ‘(%)’

Operadores relacionais:
    TopEq = ‘(==)’
    TopDif = ‘(!=)’
    TopGreThen = ‘(>)’
    TopGreThnE = ‘(>=)’
    TopLowThen = ‘(<)’
    TopLowThnE = ‘(<=)’

Operador de Concatenação:
    TopConc = ‘&’

Fim de Arquivo
    TEOF = EOF
```

10 Especificação da Gramática da Linguagem

Será especificado a seguir o tipo de analisador sintático a ser construído e sua gramática.

10.1 Analisador Sintático

O analisador sintático escolhido para C-- foi o analisador descendente preditivo recursivo, implementado na linguagem Java.

10.2 Gramática

```
S           = DeclFun S
            | DeclVar S
            | ’int’ Main

DeclFun     = FuncType DeclArrOpc ’funcId’ OpParCl OpSentCl

FuncType    = VarType
            | ’void’
```

```
VarType     = 'int'
            | 'float'
            | 'string'
            | 'bool'
            | 'char'

DeclArrOpc  = '[' ArrSizeOpc ']'
            | EPSILON

ArrSizeOpc  = ArrSizeObg
            | EPSILON

ArrSizeObg  = 'nameId'
            | 'cteInt'

OpParCl     = '(' Param ')'

Param       = ParamList
            | EPSILON

ParamList   = ParVar ',' ParamList
            | ParVar

ParVar      = VarType DeclArrOpc 'nameId'

OpSentCl    = '{' Sent '}'

Sent        = DeclVar Sent
            | FunCall Sent
            | Commands Sent
            | Var Atr ';' Sent
            | Return ';' Sent
            | EPSILON

DeclVar     = VarType DeclArrObg 'nameId' ';'
            | VarType DeclArrObg 'nameId' '=' Ec ';'
            | VarType DeclArrObg 'nameId' '=' '[' EcList ']' ';'

DeclArrObg  = '[' ArrSizeObg ']'
            | EPSILON

FunCall     = 'funcId' OpParCallCl ';'

OpParCallCl = '(' ParCall ')'

ParCall     = ParCallList
            | EPSILON
```

```
ParCallList = Ec ',' ParCallList
            | Ec

Commands    = PrintComm
            | GetComm
            | IfComm
            | FromComm
            | DuringComm

PrintComm   = 'printComm' '(' 'cteString' PrintOpt ')' ';'

PrintOpt    = ',' ParCallList
            | EPSILON

GetComm     = 'get' '(' GetVarList ')' ';'

GetVarList  = Var ',' GetVarList
            | Var

Var         = 'nameId' DeclArrObg

IfComm      = 'if' Eb OpSentCl ElseComm

ElseComm    = 'else' OpSentCl
            | EPSILON

FromComm    = 'from' Var '=' Ea 'to' Ea 'increment' Ea OpSentCl

DuringComm  = 'during' Eb OpSentCl

Atr         = '=' Ec
            | '=' '[' EcList ']'
            | EPSILON

EcList      = EcListValues
            | EPSILON

EcListValues= Ec EcListR

EcListR     = ',' EcListValues
            | EPSILON

Return      = 'return' Ec
            | 'return' '[' EcList ']'
            | 'return'

Main        = 'Main' OpParCl OpSentCl
```

```
Ec          = Ec '&' Eb
            | Eb

Eb          = Eb 'opOr' Tb
            | Tb

Tb          = Fb Tbr

Tb          = Tb 'opAnd' Fb
            | Fb

Fb          = Fb 'RelLtGt' Fc
            | 'opNot' Fb
            | 'cteBool'
            | Fc

Fc          = 'cteString'
            | 'cteChar'
            | Ra

Ra          = Ra 'opEq' Ea
            | Ea

Ea          = Ea 'AddSub' Ta
            | Ta

Ta          = Ta 'Mult' Fa
            | Fa

Fa          = '(' Ec ')'
            | 'UnNeg' Fa
            | Var
            | FunCall
            | 'cteInt'
            | 'cteFloat'
```

10.3 Gramática LL(1)

```
S           = 'int' SR
            | NotIntType Decl S
            | 'void' DeclFun S

SR          = 'Main' '(' Param ')' OpSentCl
            | Decl S
```

```
Param       = ParamList
            | EPSILON

ParamList   = ParVar ParamListR

ParVar      = VarType OptFnArray 'nameId'

VarType     = 'int'
            | 'float'
            | 'string'
            | 'bool'
            | 'char'

OptFnArray  = '[' OptFnArrayR
            | EPSILON

OptFnArrayR = ArrSize ']'
            | ']'

ArrSize     = 'nameId'
            | 'cteInt'

ParamListR  = ',' ParamList
            | EPSILON

OpSentCl    = '{' Sent '}'

Sent        = VarType OptArray DeclVar Sent
            | FunCall ';' Sent
            | Commands Sent
            | Var Atr ';' Sent
            | 'return' ReturnParam ';' Sent
            | EPSILON

OptArray    = '[' ArrSize ']'
            | EPSILON

DeclVar     = 'nameId' Atr ';'

Atr         = '=' AtrR
            | EPSILON

AtrR        = Ec
            | '[' EcList ']'

EcList      = EcListValues
            | EPSILON
```

```
EcListValues= Ec EcListR

EcListR     = ',' EcListValues
            | EPSILON

FunCall     = 'funcId' '(' ParCall ')'

ParCall     = ParCallList
            | EPSILON

ParCallList = Ec ParCalListR

ParCalListR = ',' ParCallList
            | EPSILON

Commands    = 'printComm' '(' 'cteString' PrintOpt ')' ';'
            | 'get' '(' GetVarList ')' ';'
            | 'if' Eb OpSentCl ElseComm
            | 'from' Var '=' Ea 'to' Ea 'increment' Ea OpSentCl
            | 'during' Eb OpSentCl

PrintOpt    = ',' ParCallList
            | EPSILON

GetVarList  = Var GetVarListR

Var         = 'nameId' OptArray

GetVarListR = ',' GetVarList
            | EPSILON

ElseComm    = 'else' OpSentCl
            | EPSILON

ReturnParam = Ec
            | '[' EcList ']'
            | EPSILON

NotIntType  =  'float'
            | 'string'
            | 'bool'
            | 'char'

Decl        = '[' DeclR
            | DeclAux

DeclR       = ']' DeclFun
            | ArrSize ']' DeclAux
```

```
DeclFun     = 'funcId' '(' Param ')' OpSentCl

DeclAux     = DeclFun
            | DeclVar

Ec          = Eb Ecr

Ecr         = '&' Eb Ecr
            | EPSILON

Eb          = Tb Ebr

Ebr         = 'opOr' Tb Ebr
            | EPSILON

Tb          = Fb Tbr

Tbr         = 'opAnd' Fb Tbr
            | EPSILON

Fb          = Fc Fbr
            | 'opNot' Fb
            | 'cteBool'

Fbr         = 'RelLtGt' Fc Fbr
            | EPSILON

Fc          = 'cteString'
            | 'cteChar'
            | Ra

Ra          = Ea Rar

Rar         = 'opEq' Ea Rar
            | EPSILON

Ea          = Ta Ear

Ear         = 'AddSub' Ta Ear
            | EPSILON

Ta          = Fa Tar

Tar         = 'Mult' Fa Tar
            | EPSILON

Fa          = '(' Ec ')'
```

```
| 'UnNeg' Fa
| Var
| FunCall
| 'cteInt'
| 'CteFloat'
```