Universidade Federal de Alagoas - UFAL
Instituto de Computação - IC

Jadson Crislan Santos Costa
Samuel Lima da Silva

Compiladores

2020.1
Universidade Federal de Alagoas - UFAL
Instituto de Computação - IC

Sumário

1. Estrutura geral de um programa

A linguagem admite escopo global e escopo local de variáveis. As variáveis são declaradas ao especificar seu tipo e seu identificador em qualquer parte do programa. Subprogramas são declarados ao se declarar seu tipo de retorno, seu identificador e sua lista de parâmetros. A execução de um programa se inicia no subprograma de identificador 'main'. Uma instrução sempre deve terminar com o caractere **;**. Blocos de comandos são iniciados por **{** e terminados por **}**. Ao longo deste texto os caracteres especiais e palavras reservadas da linguagem estão em negrito.

2. Nomes

Nomes são sensíveis à caixa. Nomes não possuem limite de caracteres. Nomes podem conter qualquer caractere alfabético, numérico e o caractere _. Nomes só podem começar com caractere alfabético ou _.

A expressão regular para aceitar um nome:

‘[_a-zA-Z][_a-zA-Z0-9]*’

3. Palavras reservadas

Palavras reservadas da linguagem não podem ser usadas como nome. As palavras reservadas são: **if, else, while, for, break, return, int, float, char, void, string, bool, and, or, true, false**.

Os caracteres especiais da linguagem são: **; , + - * ( ) { } [ ] ! ‘ ’ “ ” = > < ^ %**

4. Variáveis

Variáveis possuem um nome, valor e endereço na memória. Variáveis podem ser declaradas dentro de subprogramas ou fora deles. Não é permitido mascaramento de variáveis.

Várias variáveis podem ser declaradas em uma mesma instrução. Mais detalhes sobre inicialização na seção 7.

A forma de declaração de uma variável é:

tipo nome **;**

Ou na forma de declarar várias variáveis com o mesmo tipo de uma única vez:

tipo nome1 , nome2 **,** nome3 **,** … **,** nomeN **;**

Ou na forma de declarar uma única variável com um valor de inicialização:

tipo nome **=** valor_de_inicialização **;**

forma de inicializar várias variáveis numa mesma instrução:

tipo nome1 **=** valor_de_inicialização **,** nome2 **=** valor_de_inicialização**,** … **,** nomeN **=** valor_de_inicialização**;**

Mais detalhes sobre tipos na seção 7. Mais detalhes sobre valores de inicialização na seção 7. A forma de declaração de uma variável do tipo array é detalhada na seção 7.

5. Escopo

6.1 Escopo Local

A linguagem admite escopo local de variáveis. Variáveis declaradas dentro de subprogramas possuem escopo local.

6.2 Blocos

A linguagem permite o uso de blocos para definir novos escopos locais dentro de um escopo estático existente, assim, blocos podem ser aninhados. Blocos sempre são limitados por **{** e **}**.

6.3 Escopo global

Variáveis declaradas fora de subprogramas possuem escopo global. Variáveis com escopo global são visíveis a todos os subprogramas no programa.

6.4 Ordem de declaração

A declaração de uma variável dentro de um subprograma pode aparecer em qualquer ponto que uma instrução possa ocorrer. Entretanto, seu escopo começa em sua declaração e termina no fim do bloco ao qual pertence.

7. Tipos de dados

7. 1 Tipos numéricos

- **int**: Número inteiro.
- **float**: Número de ponto flutuante.

Tipos numéricos só suportam números na base decimal.
A expressão regular para constantes literais do tipo **int** é:

‘{Dig}{Dig}*’

Onde Dig é expandido para [0-9].

A expressão regular para constantes literais do tipo float é:

‘({Dig})({Dig}*)\.({Dig}*)’

A constante literal deve estar na base decimal.

Variáveis de tipo numéricos podem ser inicializadas com constantes literais de seu tipo.

7. 2 Tipos Caracteres

- **char**: 1 único caractere. Admite qualquer caractere ASCII.
- **string**: Cadeia de caracteres. Caracteres da string podem ser indexados utilizando **[**índice**]**. Onde índice deve ser uma constante literal do tipo **int**, uma variável do tipo **int** ou uma expressão aritmética com resultado do tipo **int**. O primeiro caractere da string tem índice 0 e assim por diante. Mais detalhes nas seções 8 e 13.
  A expressão regular para formar constantes literais do tipo char e string é:
  - **char:** \’[^\’]\’
  - **string:** \”([^\”]*)\”

7. 3 Tipos Booleanos

- **bool**: Booleano.
  As constantes literais do tipo bool são **true** ou **false**.

7. 4 Tipos array

- O tipo array é definido ao se colocar **[ ]** logo após e junto do nome variável. O tamanho do array é definido entre os **[ ]**. O tamanho do array pode ser uma expressão aritmética que terá seu valor avaliado em tempo de execução ou uma constante literal do tipo **int** com valor não negativo. Arrays podem ser definidos como se segue:
  - tipo nome **[** tamanho **] ;**
  - tipo nome **[** expressão aritmética **] ;**
  - tipo nome **[ ] = {** elementos **} ;**
    Os elementos podem ser constantes literais do tipo especificado separadas por **,** ou variáveis do tipo especificado separadas por **,** . O tamanho do array é inferido pela quantidade de elementos. arrays são indexados por números inteiros a partir do 0. Elementos do array são acessados pelo seu índice usando a forma:
    nome**[**índice**]**

7. 5 Outros tipos

- **void**: Tipo usado somente para indicar que o subprograma não possui retorno, é um procedimento

7. 6 Operações com os tipos

As operações para os tipos suportados são as seguintes:

Tipo	Operações
int	Aritméticas e relacionais.
float	Aritméticas e relacionais.
char	Relacionais de igualdade e diferença e concatenação, nesse caso o resultado é do tipo **string**
string	Relacionais de igualdade e diferença e concatenação.
bool	Relacionais de igualdade e diferença, negação, conjunção, disjunção.

8. Expressões e atribuição

Expressões na linguagem são composições de operadores, constantes literais e nomes.

8.1 Expressões aritméticas

Todos os operadores aritméticos binários são infixos. Os operadores aritméticos são: **+**(adição), **-**(subtração), ***(multiplicação), **/**(divisão), **%**(resto).

As expressões aritméticas são compostas com variáveis do tipo **int** ou float e por constantes literais desses tipos e com operadores aritméticos. O tipo do resultado depende da coerção feita nos operandos. Ver seção 8 e 13.

Os operadores unários são prefixo: **+**,**-** também são operadores binário.

8.2 Expressões relacionais e lógicas

Todos os operadores relacionais e o booleanos binários são infixos. Os operadores relacionais são: **>**(maior que), **<**(menor que), **>=**(maior igual que), **<=**(menor igual que), **==**(igualdade), **!=**(desigualdade).

Os operadores lógicos são: **!**(negação), **and**(disjunção), **or** (conjunção), **^**(disjunção exclusiva).

Expressões relacionais são compostas com variáveis do tipos **int**, **float**, **bool** ou constantes literais desses tipos e com operadores relacionais. O resultado sempre é do tipo **bool**.

Expressões lógicas são compostas com variáveis do tipo **bool** ou constantes literais desse tipo e com operadores lógicos.

Os operadores unários são prefixos: **!**.

8.3 Expressões com caracteres e cadeia de caracteres

A linguagem só possui um único operador de caracteres e strings: **++**. Denominado operador de concatenação. É um operador binário. Permite concatenar operando de qualquer tipo para formar uma string.

Ex.:

true ++ “c” concatena para “truec”.

2++”a” concatena para “2a”.

8.4 Precedência e associatividade

A associatividade, precedência dos operadores e tipos suportados dos operandos são mostrados na tabela:

Operador	**Associatividade**	**Número de Precedência**	**Tipos suportados**
++	esquerda	1	**char**, **string**
or	esquerda	2	**bool**
^	esquerda	3	**bool**
and	esquerda	3	**bool**
==	esquerda	4	**int**, **float**, **char**, **string**, **bool**
!=	esquerda	4	**int**, **float**, **char**, **string**, **bool**
>	esquerda	5	**int, float**
<	esquerda	5	**int**, **float**
>=	esquerda	5	**int**, **float**
<=	esquerda	5	**int**, **float**
+	esquerda	6	**int**, f**loat**
-	esquerda	6	**int**, **float**
*	esquerda	7	**int**, **float**
/	esquerda	7	**int**, **float**
%	esquerda	7	**int**
!	direita	8	**bool**
-(unário)	direita	8	**int**, **float**

+(unário)	direita	8	**int**, **float**

Quanto maior o número de precedência maior a precedência do operador.

A ordem de associatividade e precedência também é modificada pelo uso de parênteses.

8.5 Avaliação em curto-circuito

Apenas os operadores **and**, **or** são avaliados em curto-circuito.

8.6 Sobrecarga de operadores

- Os operadores **+**, **-**, ***, **/** são sobrecarregados para permitir operações entre operandos do tipo **int** e **float**.

- **<**, **>**, **<=**, **<=**, **==**, **!=** e **^** são sobrecarregados para permitir operações entre operandos do tipo **int**, **float**. Também são sobrecarregados para permitir operações entre operandos do tipo **bool**, porém os operandos envolvidos na expressão devem ser do tipo **bool**.
  and e or só aceitam operandos do tipo **bool**.

- **++** aceita sobrecarga para permitir operações entre operandos do tipo **char**, **string**, **float**, **int**, **bool**.

8.7 Conversões de tipos

A linguagem permite coerção do tipo **char** para **string** numa operação de concatenação. Permite coerção do tipo **int** para o tipo **float** em uma operação aritmética. Conversões explícitas são feitas através de:

(tipo**)**nome_da_variavel.

8.8 Atribuição

A atribuição tratada como uma instrução.
A forma geral de uma atribuição é:

nome **=** valor **;**

Onde nome é o identificador da variável que vai ter valor atribuído. valor tem que ser uma expressão do mesmo tipo de nome.

9. Estruturas de controle de fluxo

9.1 Estruturas de seleção

A linguagem admite estrutura de seleção de duas vias. O tipo da expressão que controla a seleção é bool. Logo, a expressão deve ter a forma de uma expressão relacional ou lógica. A expressão de controle é sempre colocada entre parênteses. Ver seção 8 para mais detalhes.

A forma geral da estrutura de seleção é:

if (expressão de controle) {
    lista de comandos
} else {
    lista de comandos
}

Onde **if (** expressão de controle **)** é a parte que controla a seleção. Se a expressão de controle tiver valor **true** a lista de comando entre **{** e **}** é executada. Caso contrário, a lista de comando entre **else {** e **}** é executada. O uso de seletores aninhados é permitido. O bloco else é opcional.

9.2 Estruturas de iteração

a. Controle lógico

A linguagem admite interação controlada por controle lógico usando a palavra reservada 'while'.

O tipo da expressão que controla a iteração é bool. Então, a expressão deve ter a forma de uma expressão relacional ou lógica. Ver seção 8 para mais detalhes.

A forma geral da estrutura de iteração com controle lógico é:

while (expressão de controle) {
    lista de comandos
}

Onde **while (expressão de controle)** é onde o teste de repetição acontece. Enquanto a **expressão de controle** tiver valor true a lista de comando entre **{** e **}** é executada e se repete sempre que **}** é alcançado. A repetição só para quando **expressão de controle** for **false**.

b. Controle por contador

A linguagem admite iteração controlada por contador usando a palavra reservada **for**. A forma geral da estrutura é:

for (var_inc**,** var_inicio **,** var_limite **,**var_step) **{**

lista de comandos

}

São usadas quatro variáveis para controlar a repetição. Onde var_inc é variável a ser incrementada a cada repetição, var_inicio é a variável com o valor inicial da primeira interação, var_limite é a variável com valor da última da iteração e var_step é a variável com o valor a ser incrementado na variavel de controle a cada repetição. As quatro variáveis devem ser do mesmo tipo e devem ter sido declaradas antes da aparição da estrutura. Os tipos suportados para essas variáveis são **int**. As variáveis não podem ter seu valor alterado enquanto a repetição estiver ocorrendo dentro do corpo da estrutura. Quando a repetição inicia var_inc é atribuída o valor de var_inicio. A repetição é encerrada assim que a variável a ser incrementada atinge o valor de var_limite, sendo que var_inc assume o valor de var_limite na última interação.

9.3 Desvio incondicional

Quando uma instrução **break** é executada dentro de um loop, o loop em questão é encerrado.

10. Subprogramas

A linguagem admite o uso de subprogramas. Subprogramas podem ser funções ou procedimentos.

10.1 Declaração de um subprograma

A forma geral de declaração de um subprograma é:
tipo identificador **(** lista de parâmetros **) {**
    comandos
}

Onde tipo é o tipo de retorno do subprograma. Os tipos podem ser qualquer um especificados na seção 7. Quando um subprograma possui um tipo diferente de void, um dos comandos do subprograma deve ser um return.

A forma geral de um return é:
return valor **;**

Onde valor é uma expressão com o mesmo tipo de retorno do subprograma. o identificador deve ser único em todo o programa. Quando o valor a ser retornado é um array, na declaração de tipo do subprograma deve ser colocado [].

Ex.:
tipo **[ ]** identificador **(** lista de parâmetros formais **) {**
    comandos
}

Indica que um array de tipo é retornado pelo subporgrama.

A forma geral da lista de parâmetros formais é:
tipo1 nome1 **,** tipo2 nome2 **,** tipo3 nome3 **,** …**,** tipoN nomeN

Onde tipo i, com i entre 1 e N, é o tipo do parâmetro nome i. Os tipos podem ser qualquer tipo especificado na seção 7, com exceção de void.

Quando o subprograma possui tipo void, a presença de um comando return não é obrigatória e o subprograma não retorna valor algum.

A forma de iniciar a execução de um subprograma é:
identificador **(** lista de parâmetros reais **) ;**

Onde identificador deve ser o mesmo na declaração do subprograma. A lista de parâmetros reais deve conter todos os parâmetros formais informados na declaração. A correspondência entre os parâmetros formais e reais é posicional e os tipos dos parâmetros formais devem ser respeitados pelos parâmetros reais.

A forma da lista de parâmetros reais é:

nome1 **,** nome2 **,** nome3 **,** … **,** nomeN

Subprogramas não podem ser declarados dentro de outros subprogramas.

10.2 Modo de passagem de parâmetros

O modo adotado pela linguagem é sempre em modo de entrada. Os parâmetros reais são sempre passados por valor.

11. Instruções de entrada e saída

A instrução de leitura da entrada padrão é realizada especificando as variáveis em que serão atribuídos os valores lidos.

input (nome1 **,** nome2 **,**... **,** nomeN **);**

Um erro é lançado caso a variável nomei, i entre 1 e N, não seja do mesmo tipo informado pelo usuário.

‘ Na instrução de saída padrão deve ser especificado o texto formatado e as variáveis ou constantes literais que vão ser substituídas.

output (“%2d %.2f …%f **“,** nome1**,** nome2**,** …**,** nomeN**);**

A expressão entre “ ” é o que define a formatação e sempre deve vir antes de qualquer nome. Um erro é lançado caso a variável nomei, i entre 1 e N, não seja do mesmo tipo informado na expressão ou não esteja presente na expressão. A ordem entre a formatação e a aparição dos nomes é posicional.

12. Comentários

A linguagem admite comentários de linha usando **//**. Comentários que envolvam uma ou mais linhas iniciam com **/*** e terminam com ***/**. Não é permitido comentário de múltiplas linhas dentro de comentário de múltiplas linhas.

13. Especificação dos tokens

A linguagem em que os analisadores léxico e sintático serão implementados será C++.

Enumeração com as categorias dos tokens:

```
enum  Category {
CtrlIf = 1, CtrlElse, LoopWhile, LoopFor, Break,
Return, Input, Output, Integer, Float, Char, Void,String, Boolean,
Parenth1, Parenth2, Braces1,Braces2, Bracket1, Bracket2, OpAdd,
OpMinus, OpDiv, OpMult, OpRem, OpConcat, OpGreater, OpLesser, OpGEqual,
OpLEqual, OpEqual, OpDiff, OpBinXor,OpNot, OpAnd, OpOr, Assign, Comma,
Terminator, Error, Id, CteInt, CteFloat, CteChar, CteString, CteBool, Eof };
```

A categoria Error é usada para indicar um erro no analisador léxico, o lexema formado até aquele ponto é retornado. A categoria Eof é usada para indicar que chegou no fim de arquivo.

13.1 Tabela de categoria de tokens

Lexema	**Categoria**
if	CtrlIf
else	CtrlElse
while	LoopWhile
for	LoopFor
break	Break
return	Return
input	Input
output	Output
int	Integer
float	Float

char	Char
void	Void
string	String
bool	Boolean
(	Parenth1
)	Parenth2
{	Braces1
}	Braces2
[	Bracket1
]	Bracket2
+	OpAdd
-	OpMinus
/	OpDiv
*	OpMult
%	OpRem
++	OpConcat
>	OpGreater
<	OpLesser
>=	OpGEqual
<=	OpLEqual
==	OpEqual
!=	OpDiff
^	OpBinXor
!	OpNot
and	OpAnd
or	OpOr
=	Assign

,	Comma
;	Terminator

13.2 Tabela de expressões regulares

Expressões regulares auxiliares

- dig = ‘[1-9]’
- Dig = ‘[0-9]’

Não Terminal	**Categoria**
[_a-zA-Z][_a-zA-Z0-9]*	Id
{Dig}{Dig}*	CteInt
({Dig})({Dig}*)\.({Dig}*)	CteFloat
\’[^\’]\’ \”([^\”]*)\”	CteChar
\’[^\’]\’ \”([^\”]*)\”	CteString
(true)|(false)	CteBool