ilp2018 Documentation

Release 0.6

Lucas Nanni

Oct 15, 2018

Conteúdo:

CHAPTER 1

Introdução

Grace é uma linguagem de programação projetada especificamente para a disciplina de Implementação de Linguagens de Programação (ILP) do Departamento de Informática (DIN) da Universidade Estadual de Maringá (UEM). A linguagem possui apenas fins didáticos e não se compromete a entregar funcionalidades encontradas em linguagens de programação de propósito geral.

O nome da linguagem é uma homenagem à Grace M. Hopper. Arquivos de código fonte escritos em Grace devem possuir a extensão `.grc`.

1.1 Características da linguagem

- Imperativa;
- Fortemente tipada;
- Declaração explícita de variáveis;
- Vinculação estática de tipos;
- Sistema de escopo estático (léxico);
- Sensível à caixa (case-sensitive);

1.2 Sistema de Tipos

A linguagem possui um sistema de tipos com duas classes: tipos primitivos e tipos agregados.

1.2.1 Tipos primitivos

Os tipos primitivos são números inteiros, valores lógicos e strings, representados respectivamente pelos tipos `int`, `bool` e `string`.

1.2.2 Tipos Agregados (arranjo)

O tipo agregado é um arranjo de algum tipo primitivo. Dessa forma, podemos ter as variantes: arranjo de inteiros, arranjo de lógicos e arranjo de strings.

CHAPTER 2

Especificação Léxica

2.1 Identificadores

Chamamos de identificador qualquer nome criado pelo usuário da linguagem. Os identificadores seguem a mesma regra de formação da linguagem C:

Devem iniciar com uma letra (minúscula ou maiúscula) ou um subtraço seguido de letras, subtraços ou dígitos entre 0 e 9.

Um identificador será expresso pelo símbolo id nas especificações sintáticas.

2.2 Literais

Daremos o nome de literal a todo valor fixado no código. A linguagem possui representação de literais para seus três tipos primitivos.

2.2.1 Números

Os literais numéricos devem ser representados na base decimal e podem conter qualquer combinação de dígitos entre 0 e 9. Os números negativos não serão processados na fase léxica, mas sim na sintática e semântica. Dessa forma, o número -42, por exemplo, consiste de dois lexemas: "-" e "42", e serão tratados como uma operação aritmética nas análises sintática e semântica.

2.2.2 Strings

Os literais string possuem a mesma regra de formação definida pela linguagem C.

Exemplo: `"isso é \"uma\" string!\n"`

2.2.3 Lógicos

Os literais lógicos verdadeiro e falso são representados pelos lexemas `true` e `false` respectivamente.

2.3 Comentários

A linguagem possui apenas comentários de linha:

- Começam com `//` e seguem até o final da linha.

De forma geral, os comentários podem conter qualquer tipo de símbolo, inclusive os não permitidos pela linguagem. Os comentários devem ser processados corretamente pelo analisador léxico e em seguida descartados.

2.4 Palavras reservadas e símbolos

- Palavras reservadas: `bool` `def` `else` `false` `for` `if` `int` `read` `return` `skip` `stop` `string` `true` `var` `while` `write`
- Símbolos: `(` `)` `[` `]` `{` `}` `,` `;` `+` `-` `*` `/` `%` `==` `!=` `>` `>=` `<` `<=` `||` `&&` `!` `=` `+=` `-=` `*=` `/=` `%=` `?` `:`

CHAPTER 3

Especificação Sintática

3.1 Programa

Um programa consiste de uma sequência não vazia de declarações de variáveis e subprogramas.

```
programa ::= dec {dec}
```

3.2 Variáveis

Existem dois tipos de variáveis: as simples e as agregadas. Variáveis simples suportam apenas um único valor de um determinado tipo primitivo em um determinado momento. Variáveis agregadas são de tipos agregados (arranjos) e suportam mais de um valor de um mesmo tipo em um determinado momento.

3.2.1 Declaração de Variáveis

```
decVar ::= 'var' listaSpecVars ':' tipo ';'
listaSpecVars ::= specVar {',' specVar}
specVar ::= specVarSimples | specVarSimplesIni |
            specVarArranjo | specVarArranjoIni
```

Exemplo:

```
var a, b = 3, c = 2 + b: int;
var str1, str2 = "String 2": string;
var i, j = true: bool;
var x, v[10], z[3] = {1, 5, 8}: int;
```

Observe que a declaração de variáveis é indicada pela palavra reservada var. Observe também que múltiplas variáveis podem ser declaradas de uma vez e que elas podem ser inicializadas durante a declaração.

Na declaração de arranjos, o tamanho da estrutura deve ser especificada como um literal numérico.

O tipo *string*

Dados do tipo string não são indexados como na maioria das linguagens. O objetivo da existência do tipo string na linguagem é apenas fornecer uma maneira de apresentar (escrever) mensagens na tela.

No momento da declaração de uma variável do tipo string, é possível indicar a quantidade de memória que será reservada a ela. Por exemplo, nas declarações

```
var palavra: string[32]; // memória reservada para 32 caracteres.
var texto: string; // memória reservada para 256 caracteres.
var nome = "Fulano": string; // memória reservada para 6 caracteres.
var titulo = "Meu programa": string[64]; // memória reservada para 64 caracteres.
```

a variável `palavra` foi declarada como uma string capaz de armazenar 32 caracteres. Caso não seja informado o tamanho da string, como na declaração da variável `texto`, serão reservados 256 caracteres. Quando a variável é inicializada na declaração, será alocado o maior espaço entre o suficiente para a string ou o especificado na declaração.

3.3 Subprogramas (procedimentos e funções)

A definição de procedimentos e funções possui uma sintaxe comum, exceto pela ausência do tipo de retorno para procedimentos. Não há separação entre declaração e definição de subprogramas, isto é, o subprograma deve ser definido durante sua própria declaração.

3.3.1 Declaração de Subprogramas

```
decSub ::= decProc | decFunc
```

Declaração de Procedimento

```
decProc ::= 'def' id '(' [listaParâmetros] ')' bloco
```

Exemplo

```
def proc(y: int) {
    if (y < 0) {
        return;
    }
    x = 2 * y; // x é global!
}
```

Declaração de Função

```
decFunc ::= 'def' id '(' [listaParâmetros] ')' ':' tipo bloco
```

Exemplo

```
def func(x[], y: int; z: bool): int {
    a = x[y-1]: int;
    return a + 1;
}
```

Lista de Parâmetros

```
listaParâmetros ::= specParams {';' specParams}
specParams ::= param {',' param} ':' tipo
param ::= id | id '[' ']'
```

Parâmetros de tipo inteiro ou lógico são passados naturalmente por cópia e parâmetros de tipo arranjo ou string são passados naturalmente por referência.

Declaração Aninhada de Subprogramas

A sintaxe de declaração de subprogramas permite que eles sejam declarados de maneira aninhada, como exemplificado a seguir:

```
def adicionar(v[]: int; n: int; x: int) {
    var i: int;
    def soma(a: int): int {
        return a + x;
    }

    for (i=0; i<n; i+=1) {
        v[i] = soma(v[i]);
    }
}
```

Observe o funcionamento do escopo local permitindo que o parâmetro `x` do procedimento `adicionar` seja utilizado dentro da função `soma`.

3.4 Comandos

Existem duas classes de comandos: os comandos simples e os blocos de comando.

```
comando ::= cmdSimples | bloco
```

A seguir são especificados os comandos simples:

3.4.1 Atribuição

```
cmdAtrib ::= atrib ';'
atrib ::= variável ('='|'+='|'-='|'*='|'/='|'%=') expressão
```

O comando de atribuição avalia o valor da expressão e o armazena na variável.

As atribuições compostas devem ser traduzidas da seguinte maneira:

```
var X= expressão -> var = var X expressão
```

3.4.2 Condicional If

```
cmdIf ::= 'if' '(' expressão ')' comando ['else' comando]
```

A estrutura condicional if é executada verificando o resultado da expressão de teste. Se ela resultar no valor true, apenas o primeiro comando será executado. Se a expressão resultar no valor false, caso a estrutura else esteja presente, apenas o segundo comando será executado.

3.4.3 Laço While

```
cmdWhile ::= 'while' '(' expressão ')' comando
```

O laço while inicia verificando o resultado da expressão de teste. Caso o valor seja true, o comando do seu corpo é executado e o laço volta a testar o valor da expressão de teste para a próxima iteração. Caso o valor seja false, a execução do laço é interrompida.

3.4.4 Laço For

```
cmdFor ::= 'for' '(' atrib-ini ';' expressão ';' atrib-passo ')' comando
```

O laço for inicia executando a atribuição de inicialização. A partir daí, antes de cada iteração, o resultado da expressão de teste é verificado. Se ele for true, o comando corpo é executado e a atribuição de passo é executada em seguida, reiniciando o processo. Se antes de qualquer iteração o valor resultado pela expressão de teste for false, a execução do laço é interrompida.

3.4.5 Interrupção do laço

```
cmdStop ::= 'stop' ';'
```

O comando stop interrompe o laço mais próximo que o cerca. Ele só pode aparecer dentro do corpo de comandos de repetição while e for.

3.4.6 Salto de iteração do laço

```
cmdSkip ::= 'skip' ';'
```

O comando skip salta para a próxima iteração do laço mais próximo que o cerca, ignorando a execução dos comandos que o seguem dentro deste laço. Ele só pode aparecer dentro do corpo de comandos de repetição while e for.

3.4.7 Retorno de subprograma

```
cmdReturn ::= 'return' [expressão] ';'
```

O comando return encerra a execução do subprograma que o cerca retornando o valor resultado pela expressão. A expressão de retorno de uma função deve resultar em um valor do mesmo tipo para o qual a função foi definida. Funções devem obrigatoriamente conter pelo menos um comando return. Já procedimentos podem ou não conter comandos return. Caso o tenham, eles devem retornar nada: return; Como o programa principal é definido por meio de uma função, ele deve conter pelo menos um comando return e o valor retornado deve ser um número inteiro.

3.4.8 Chamada de procedimento

```
cmdChamadaProc ::= id '(' [expressão {',' expressão}] ')' ';'
```

Como a chamada de procedimentos não resulta em um valor, é necessário um comando para sua execução. A chamada de funções possui sintaxe semelhante, exceto por não ser um comando, e sim uma expressão.

3.4.9 Entrada Read

```
cmdRead ::= 'read' variável ';'
```

3.4.10 Saída Write

```
cmdWrite ::= 'write' expressão {',' expressão} ';'
```

3.4.11 Bloco de Comandos

Um bloco é uma sequência de (nenhuma ou várias) declarações de subprogramas e variáveis seguida de uma sequência de (nenhum ou vários) comandos. Um bloco é circundado por chaves `{ }`.

```
bloco ::= '{' {dec} {comando} '}'
```

3.5 Expressão

Uma expressão pode conter valores dos três tipos definidos (inteiros, lógicos e strings), uso de variáveis, chamadas de função e outras expressões. Uma expressão pode estar cercada por parênteses e se relacionar a outras expressões por meio dos seguintes operadores:

Table 1: Tabela de Operadores

Precedência	Operador	Descrição	Associatividade
1	-	Negativo Unário	À direita
	!	Não lógico	
2	*, /, %	Multiplicação, divisão e resto	À esquerda
3	+, -	Adição e subtração	
4	<, <=	Operadores relacionais < e respectivamente	
	>, >=	Operadores relacionais > e respectivamente	
5	==, !=	Operadores relacionais = e respectivamente	
6	&&	E lógico	
7	||	OU lógico	
8	? :	Condicional ternário	À direita

O operador condicional ternário é formado da seguinte maneira:

```
opTern ::= expressão-teste '?' expressão-então ':' expressão-senão
```

A expressão teste é avaliada. Se o resultado for `true`, a expressão-então é resultada, caso contrário, a expressão-senão é resultada. Dessa forma, o resultado desse operador é sempre uma expressão. O operador pode ser utilizado assim:

```
x = a > 0 ? a * 2 : a + 1;
```

O operador condicional ternário terá associatividade à direita, ilustrado no exemplo abaixo, onde a expressão `b > 0 ? a / b : a + b` é uma expressão-senão, como em C e C++, ao invés de tratar a expressão `a > 0 ? a * 2 : b > 0` como expressão-teste, como em PHP.

```
x = a > 0 ? a * 2 : b > 0 ? a / b : a + b;
x = a > 0 ? a * 2 : (b > 0 ? a / b : a + b); // Associação à Direita (C, C++)
x = (a > 0 ? a * 2 : b > 0) ? a / b : a + b; // Associação à Esquerda (PHP)
```

3.5.1 Uso de variável

Como o uso de uma variável resulta no valor armazenado pela variável, todo uso de variável é uma expressão. Variáveis simples são usadas por meio do identificador (nome) associado a ela e variáveis compostas (arranjo) são usadas por meio do identificador e a posição numérica do elemento acessado.

```
variável ::= id | id '[' expressão ']'
```

Observe que a sintaxe do uso de variável não impede que uma variável simples seja utilizada como arranjo. Essa associação deve ser verificada na etapa de análise semântica.

CHAPTER 4

Especificação Semântica

4.1 Programa

Um programa consiste de uma sequência de declarações. A última declaração deve ser obrigatoriamente a da rotina principal, pela qual se dará o início da execução do programa. Essa declaração deve ser de uma função chamada `main` com retorno do tipo `int`. O valor retornado por essa função representa o resultado da execução do programa, onde `0` significa que a execução foi bem sucedida.

Todas as declarações realizadas no programa (fora de qualquer subprograma) estão dentro do escopo global.

4.2 Declaração

Declaração de variáveis, funções e procedimentos são responsáveis por adicionar os símbolos envolvidos e suas vinculações na tabela de símbolos.

Caso a declaração de uma variável considere sua inicialização, o tipo da expressão de inicialização deve ser o mesmo declarado para a variável.

4.3 Comandos

4.3.1 If

- A expressão condicional do comando `if` deve resultar em um valor do tipo lógico.

4.3.2 While

- A expressão condicional do comando `while` deve resultar em um valor do tipo lógico.

4.3.3 For

- As atribuições de inicialização e passo devem ser analisadas como um comando de atribuição normal.
- A expressão condicional deve resultar em um valor do tipo lógico.

4.3.4 Stop

- O comando `stop` deve estar cercado (diretamente ou indiretamente) por um comando de repetição (while ou for).

4.3.5 Skip

- O comando `skip` deve estar cercado (diretamente ou indiretamente) por um comando repetição (while ou for).

4.3.6 Return

- Caso apareça dentro de uma função, o tipo da expressão de retorno deve ser o mesmo do retorno declarado da função. Caso apareça dentro de um procedimento, o comando return não pode ter expressão.

4.3.7 Read

- A variável utilizada no comando read deve estar declarada e visível no escopo atual.

4.3.8 Write

- Não há análise especial para o comando write.

4.3.9 Chamada de Procedimento

- O procedimento chamado deve estar declarado e visível no escopo atual.
- O número de argumentos fornecidos deve ser o mesmo da declaração do procedimento.
- Os argumentos fornecidos devem ter a mesma ordem de tipo utilizada na declaração do procedimento.

4.3.10 Atribuição

- O lado esquerdo da atribuição deve ser uma variável declarada (simples ou acesso de arranjo) e visível no escopo atual.
- O lado direito deve ser uma expressão com tipo igual ao da variável do lado esquerdo da atribuição.

4.3.11 Bloco

- Define um novo escopo estático. O escopo é criado no início do bloco e finalizado no término do bloco.

4.4 Expressões

4.4.1 Aritmética (+, −, *, /, %, neg)

- O(s) operando(s) devem ser do tipo inteiro. O tipo resultante é inteiro.

4.4.2 Relacional (>, >=, <, <=)

- Os operandos devem ser do tipo inteiro. O tipo resultante é lógico.

4.4.3 Igualdade (==, !=)

- Os operandos devem ser do mesmo tipo primitivo. O tipo resultante é lógico.

4.4.4 Lógica (&&, ||, !)

- O(s) operando(s) devem ser do tipo lógico. O tipo resultante é lógico.

4.4.5 Ternária

- A expressão condicional deve resultar um valor do tipo lógico.
- As expressões então e senão devem possuir o mesmo tipo.
- O tipo resultante da expressão ternária é o mesmo tipo da expressão-então.

4.4.6 Uso de variável

- A variável deve estar declarada e visível no escopo atual. O tipo resultante é o tipo declarado da variável.
- Para variáveis agregadas, a expressão que resulta no índice a ser acessado deve ser do tipo inteiro.

4.4.7 Chamada de função

- Análise análoga à chamada de procedimento.
- O tipo resultante é igual ao tipo declarado da função.

CHAPTER 5

Exemplos

5.1 Bubble-Sort

```
var v[10]: int;

// Procedimento de ordenação por troca
// Observe como um parâmetro de arranjo é declarado

def bubblesort(v[]: int; n: int) {
    var i=0, j: int;
    var trocou = true: bool;

    while (i < n-1 && trocou) {
        trocou = false;
        for (j=0; j<(n-i-1); j+=1) {
            if (v[j] > v[j+1]) {
                var aux = v[j]: int;
                v[j] = v[j+1];
                v[j+1] = aux;
                trocou = true;
            }
        }
        i += 1;
    }
}

def main(): int {
    var i: int;

    write "Digite os valores do arranjo:\n";

    for (i=0; i<10; i+=1) {
        write "A[", i, "] = ";
        read v[i];
```

(continues on next page)

(continued from previous page)

```
    }

    bubblesort(v, 10);

    write "Arranjo ordenado:\nA = ";

    for (i=0; i<10; i+=1) {
        write v[i], " ";
    }

}
```