Universidade Federal de Alagoas - UFAL
Instituto de Computação - IC
Curso de Ciência da Computação

Prof. Alcino Dall'Igna Júnior

Compiladores: Linguagem DL

Especificação

Derecky Costa da F. Andrade
Larissa Santos da Silva

Maceió, 19 de novembro de 2019

Conteúdo

1 Introdução

Este documento especifica a linguagem de programação DL. Esta linguagem terá implementado seu analisador léxico e sintático na linguagem Java, seguindo as especificações deste documento.

2 Estrutura Geral de um programa

A linguagem DL trata-se de uma linguagem de programação procedural, projetada para ser analisada em passo único, admitindo coerção implícita de alguns tipos compatíveis. Usa-se palavras reservadas em inglês. DL trata-se de uma linguagem sem sensibilidade (Secção 4) à caixa e fortemente tipada (Secção 5). DL não faz tratamento de erros em detecção de tipos.

O programa em DL inicia-se com a palavra `pgm`, e termina com a palavra `end_pgm`, desta forma, tudo que estiver escrito além de `end_pgm` será ignorado pelo compilador.

O bloco de instruções principal é definido pela palavra reservada `main` seguidos de um par de chaves {}, estas delimitam o escopo do bloco principal, variáveis globais e funções devem ser declaradas acima deste, admitindo escopo global.

Quanto à declaração de funções (ver seção 8.1), estas devem ser declarados através da palavra reservada `func` seguida do nome da função, parênteses e seu parâmetro, o escopo da função é delimitado por chaves {}.

As variáveis globais são inicializadas na inicialização do programa, as locais, quando o bloco de instrução for chamado.

Um programa DL tem a seguinte estrutura:

```
pgm
    <variaveis>
    <funcoes>
    main{
        <instrucoes>;
    }
end_pgm
```

Vale ressaltar que, apesar de a rotina `main` estar limitada por chaves, está não é uma função e a mesma deve vir ao final das declarações de variáveis e funções.

3 Escopo

O escopo da linguagem DL é estático. As variáveis declaradas fora das funções, têm o escopo global enquanto parâmetros de funções por sua vez, têm como escopo todo o bloco da função. Variáveis declaradas dentro de blocos, têm como escopo todo o bloco em que a declaração foi feita.

4 Nomes

Os identificadores em DL não são sensíveis a capitalização, além de **não possuir limite de caracteres**; Podem ser compostos por *letras*, *dígitos*, desde que seguindo as seguintes regras:

- A sequência não pode ser uma palavra reservada (ver seção 4.1);
  ex.: `pgm`, `end_pgm`, `main`
- Todo identificador deve ser iniciado por uma letra, portanto não é possível iniciar com dígito;
  ex.: não válido → `23abc, 12ab, 12`
- Não podem possuir caracteres especiais, como: @, !, ?, etc.;
  ex.: `ab@12, a!_2`

Onde Lê-se *dígitos*, entende-se por elementos pertencentes ao conjunto {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ao qual representaremos por [0-9].

Onde lê-se *letras*, entende-se por elementos pertencentes ao conjunto {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}, ao qual representaremos por [a-z], o caso das maiúsculas {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}, representaremos por [A-Z]. Apesar de não ser sensível a capitalização faz-se necessário o uso deste conjunto para análise.

4.1 Palavras Reservadas

Palavras especiais da linguagem são palavras reservadas, consequentemente não poderão ser usadas como identificador. A Tabela 1 mostra as palavras que compõem este conjunto.

`pgm`	`end_pgm`	`main`	`func`
`string`	`int`	`real`	`char`
`if`	`else`	`switch`	`case`
`or`	`and`	`not`	`bool`
`array`	`while`	`to`	`mod`
`repeat`	`from`	`default`	`div`
`break`	`continue`		

Tabela 1: Palavras reservadas

4.2 Identificadores

São os nomes dos símbolos definidos pelo programador que podem ser modificados e reusados, os valores destes estão sujeitos às regras de escopo da linguagem. São caracterizados por qualquer palavra iniciada por uma letra, seguida de letras e números, tanto espaços em branco quanto palavras reservadas não podem ser usados como identificadores. Não é permitido operador ou símbolo especial.

Expressão regular: `[A-Za-z][A-Za-z0-9]*`

Exemplo	Validação
aba	Válido
AbA	Válido
ANDA1	Válido
1aba	Não válido
_aba	Não válido
ds:ds	Não válido
or	Não válido
OR	Não válido

Tabela 2: Identificadores válidos e não válidos

4.3 Símbolos Especiais

São os caracteres com significado na linguagem, segue abaixo os símbolos especiais desta linguagem:

Símbolo	Significado
[]	Referência de tipo array
()	Delimita os parâmetros de uma função e define a ordem na precedência de operações
{}	Agrupa blocos de instruções
,	Separa variáveis ou parâmetros de função
;	Termina instrução

Tabela 3: Símbolos especiais

4.4 Operadores

São símbolos que desencadeiam uma ação, podendo ser unários ou binários.

**	Unário	~	Unário	>=	Binário
-	Binário	*	Binário	>	Binário
=	Binário	==	Binário	<=	Binário
+	Binário	/	Binário	<	Binário
<>	Binário				

Tabela 4: Operadores suportados

5 Tipos e Estruturas de dados

5.1 Forma de declaração

```
1  <tipo> <identificador1>,..., <identificadorN>;
2  <tipo> func <identificador>(<parametros>) {}
3  <tipo> <identificador>[<tamanho>];
```

5.2 Compatibilidade

Esta linguagem suporta compatibilidade por estrutura.

5.3 Constantes literais

Constante literal ou simplesmente literal, é um valor terminal, número, caractere ou string que poderá estar associado a uma variável ou constante simbólica, geralmente usado como: argumento de uma função; operador numa operação aritmética ou lógica. Um literal sempre representa o mesmo valor, são valores colocados diretamente no código, como o número 5, o caractere 'D' ou a string "Olá Mundo".

Literais numéricos podem ser representados numa variedade de formatos (decimal, hexadecimal, binário, ponto flutuante, octal, etc). Esta linguagem não oferece suporte aos inteiros Hexadecimais, octais e binários.

5.4 Tipos de dados primitivos

Os tipos primitivos que a linguagem DL suporta são:

5.4.1 Inteiro

É um número na base (10).

- Deve conter apenas dígitos 0-9
- Expressão regular: `[0-9]+`
- Não pode começar com zero, exceto no caso que o inteiro seja o próprio zero.
- Não pode conter o ponto decimal.
- Não pode conter vírgulas ou espaços.
- Pode ser precedido pelo unário negativo "$\sim$"
- Declaração: **`int`** `meuinteiro;`
- Exemplos de decimais inteiros válidos: `0 5 127  1002 65535`
- Exemplos de decimais inteiros inválidos: `32,76 1.2 1 27 032 3A`

5.4.2 Ponto Flutuante

Podem ser representados em vários formatos para expressar diferentes variações. O qualificador literal `"f"` força o compilador a tratar o valor como ponto flutuante e precisa ser inserido pelo programador explicitamente.

- Deve conter dígitos 0-9
- Pode conter um ponto decimal
- É permitido o qualificador literal `"f"`, forçando o compilador a tratá-lo como real
- Não pode começar com zero, a menos que o zero seja seguido de um ponto decimal.
- Pode usar a notação `"e"` para expressar valores exponenciais (`e`$\pm$`n`=10^n)
- Não pode conter vírgulas ou espaços
- Pode ser precedido pelo unário negativo "$\sim$"
- Expressão regular: `f?[:digit:]+.[:digit:]{+}([E|e][+|-]?[:digit:]+)?`
- Declaração: **`real`** `meureal;`
- Exemplos de pontos flutuantes válidos: `2.21e-5 10.22 48e+8 0.5 f10`
- Exemplos de pontos flutuantes inválidos: `02.42 f22 0x5eA`

5.4.3 Caractere

- Deve estar entre apóstrofo (aspas simples)
- Pode conter qualquer caractere imprimível
- Expressão regular: `[^]`
- Exemplos de caracteres válidos: `'r', 'R', \n, '@', '2', ' '(espaço)`
- Exemplos de caracteres inválidos: `me,`

5.4.4 Cadeia de caractere ou String

- Deve estar entre aspas duplas
- Aceita qualquer conjunto de caracteres entre aspas duplas
- Deve começar e terminar na mesma linha
- Expressão regular: `"[^"]*"`
- Declaração: **`string`** `meustring;`
- Exemplos de strings válidos: `"MM", "Nasa", "PC", "A", "sew121@&[]"`
- Exemplos de strings inválidos: `2"w", "Ola, "¨¨`

5.4.5 Lógico

É o tipo booleano, com dois únicos possivéis valores, **`true, false`**.

- Declaração: **`bool`** `meubooleano;`

5.4.6 Arranjos unidimensionais

Arranjos são variáveis que podem armazenar muitos valores do mesmo tipo, os valores individuais, chamados elementos, são armazenados sequencialmente e são identificados pelo arranjo unicamente por um índice.

- Pode conter qualquer número de elementos
- Elementos têm que ser do mesmo tipo
- Os índices têm que ser do tipo inteiro
- O índice do primeiro elemento é zero
- Os índices não podem ser valores inteiros negativos
- Quando passados como parâmetros de função, não se explicita o tamanho do arranjo
- Para variáveis, o tamanho do arranjo tem que ser explicitado na sua declaração
- Declaração: **`<tipo>`** `meuarray[<tamanho>];`

- Exemplos:
```
int meuint[12];
real meureal[8];
bool meubool[112];
```

As constantes são como variáveis, sendo a única diferença é que o seu valor não pode ser modificado pelo programa uma vez definido.

5.5 Equivalência de tipos

Esta linguagem é estaticamente tipada, toda a verificação de compatibilidade de tipos será feita estaticamente. Não admite constante com nome, apenas constantes literais dos tipos são admissíveis.

- Os tipos primitivos usam equivalência de nomes
- Os arranjos são equivalentes de forma estrutural

5.5.1 Coerções admitidas

As seguintes coerções são válidas quando as variáveis são inicializadas:

- char para int
- int para char
- int para real
- char para string
- Exemplo:
```
int meuint = v;
char meuchar = 22;
real meureal = 10;
string meureal = 'h';
```

5.5.2 Conversão de tipo explícita (cast)

- char para int
- int para char
- int para real (perde-se a parte fracionária)
- real para int
- char para string

- Exemplo:

```
int meuint = (int)v;
char meuchar = (char)22;
real meureal = (real)10;
int meuint = (int)10.2;
string meureal = (string)'h';
```

6 Atribuição e Expressões

Atribuição é uma instrução feita com operador "=". É realizado a atribuição da expressão à direita para a variável à esquerda do mesmo.

6.1 Expressões aritméticas, relacionais e lógicas

Os tipos das operações são definidos por expressão. Segue abaixo, a lista de operadores disponíveis.

- Aritméticos

Símbolo	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
**	Exponencial
$\sim$	Unário negativo
div	Divisão inteira
mod	Resto de divisão

Tabela 5: Operadores aritméticos

- Relacional

Símbolo	Relação
$>$	Maior que
$<$	Menor que
$==$	Igual a
$<>$	Diferente de
$>=$	Maior ou igual
$<=$	Menor ou igual

Tabela 6: Operadores relacionais

- Lógicos

Símbolo	Operação
and	Conjunção
or	Disjunção
not	Negação

Tabela 7: Operadores lógicos

6.2 Precedência e Associatividade

Na tabela a seguir os operadores agrupados na mesma seção têm a mesma precedência, as subsequentes secções têm precedência mais baixa, a associatividade também pode ser observada. Quando expressões são formadas por múltiplos operadores, a precedência determina a ordem de avaliação, quando dois operadores possuem a mesma precedência, a associatividade determina a ordem de avaliação.

Operador	Descrição	Associatividade
`()`	Expressão em parêntesis	Dentro para fora
`[]`	Descritor de tamanho de arranjo	
∼	Unário negativo	Direita para esquerda
`not`	NOT lógico	
`**`	Exponencial	
`* /`	Multiplicação, divisão, módulo, divisão	Esquerda para direita
`mod`	inteira	
`div`		
`+ -`	Soma, subtração	Esquerda para direita
`< <=`	Menor que, Menor que ou igual	Esquerda para direita
`> >=`	Maior que, Maior que ou igual	
`== <>`	Igual, Não igual	Esquerda para a direita
`and`	AND lógico	Esquerda para a direita
`or`	OU lógico	

Tabela 8: Precedência e associatividade de operadores

7 Sintaxe e exemplo de estruturas de controle

Esses comandos oferecem instruções para tomada de decisão. Estas obedencem a condição que representa um valor lógico, como true ou false.

7.1 Estrutura condicional de uma e duas vias

```
1            if (<condicao>) { <instrucoes> }
2
3            if (<condicao>) {
4                <instrucoes>
5            } else {
6                <instrucoes>
7            }
```

```
8
9           switch <variavel> {
10              case <condicao>:
11                  <instrucoes>;
12              default:
13                  <instrucoes>
14          }
```

7.1.1 Semântica

Para a instrução `if`: se condição for verdadeira executa bloco de instruções, caso contrário o bloco de instruções associado ao `else` subsequente ao `if` será executado.O bloco de instruções só será executado se a condição não for verdadeira.

Para a instrução `switch`: o valor da `variável` é avaliado em todas as condições `case`, se nenhuma das condições `case` for satisfeita, o bloco de instruções associado ao `default` é executado. O comando termina a sua execução quando encontra um `case` com condição verdadeira.

7.2 Estrutura iterativa com controle lógico

Esse tipo de comando permite a execução de instruções até que uma dada condição seja satisfeita.

```
1           while (<condicao\_bool>) {
2               <instrucoes>
3           }
```

7.2.1 Semântica

Se condição booleana for verdadeira (true), o conjunto de instruções é executado, esse processo será repetido até que a condição booleana seja falsa.

7.3 Estrutura iterativa controlada por contador com passo igual a um caso omitido

```
1       repeat <identificador> from <expressao1> to <expressao2> [
           step <expressao3>]? {
2           <instrucoes>
3       }
```

7.3.1 Semântica

Esse comando vai executar um conjunto de instruções enquanto o valor da `expressao1` for igual ou menor ao valor da `expressao2`, onde *expressao1* é uma expressão que retorna um valor resultante de uma operação aritmética, `expressao1` é incrementada em uma unidade ou num valor definido na `expressao3`. O valor da *expressao1* é pré-avaliado e armazenado numa variável temporária definida pelo compilador.

7.4 Desvios incondicionais

```
repeat cont from i to 10 step 2{
    <instrucoes>
    continue;
}

while(i < 10){
    i+=2;
    continue;
}
```

7.4.1 Semântica

Em uma estrutura iterativa é possível usar comandos para desviar a execução do programa. Com o comando **break**, a execução do bloco de repetição é cancelada, independente do valor da condição booleana. Enquanto ao usar a instrução **continue**, a iteração atual do bloco é cancelada e iniciado o próximo passo.

8 Subprogramas

8.1 Funções

Funções só poderão ser declaradas no escopo global, ou seja,funções não podem ser aninhadas. A declaração de uma função é feita da seguinte forma:

```
func <tipoDeRetorno> <identificador(<tipoDoParametro><
   identificadorDoParametro>) {
    <conj.Instrucoes>
    return <valorDeRetorno>
}
```

Observações:

- Dada uma função, esta poderá ter nenhum ou `n` parâmetros, no caso de mais de um parâmetros, estes devem ser separados por vírgula.
- Uma função de tipo void é uma função sem retorno.

Exemplos:

```
func int soma(int a, int b){
    int c = a + b;
    return c;
}
```

8.1.1 Procedimento

Funções sem retorno se comportam como procedimentos, desta forma, DL é uma linguagem que dá suporte a procedimentos.

9 Comentário

Os comentários são ignorados pelo compilador. Esta linguagem suporta apenas comentário de linha, todos os caracteres da linha serão ignorados após o símbolo //.

```
1         //  um comentario
```

10 Programas exemplos

Seguem abaixo alguns exemplos de programas em DL:

10.1 Alô mundo

```
1  pgm
2    main{
3      print("Alo mundo!");
4    }
5  end_pgm
```

10.2 Fibonacci

```
1  pgm
2    int valor;
3    func void fibo(int n){
4      int prev, atual, temp, i;
5      prev = 0;
6      atual = 1
7      i = 0;
8      repeat i from 0 to n {
9
10       if(i==0){ print(i + ", "); }
11       else{
12         if(i < 2){print(atual + ", ");}
13         else{
14           temp = atual;
15           atual = atual + prev;
16           prev = temp;
17           if(atual <= n){
18             print(atual + ", ");
19           }
20         }
21       }
22     }
23   }
24   main{
25     int n;
26     read(n);
27     fibo(n);
28   }
29
30 end_pgm
```

10.3 ShellSort

```
1  pgm
2    int vet[10];
3    int n;
4    int func shellsort(int v[], int tam){
5
6      int i, j, h;
```

```
7       int gap = 1;
8       while(gap < tam){
9         gap = 3*gap+1;
10      }
11      while(gap > 1){
12        gap = gap / 3;
13        repeat i from gap to size{
14          h = v[i];
15          j = i;
16          while(j >= gap and h < v[j-gap]){
17            vet[j] = v[j-gap];
18            j = j-gap;
19          }
20          v[j] = value;
21        }
22      }
23     return v;
24
25    }
26
27    main{
28     vet = [2, 3, 4, 0, 9, 7, 8, 1, 5, 6];
29     vet = shellsort(vet, 10);
30    }
31  end_pgm
```

11 Especificação dos tokens

A lista a seguir lista todos os tokens com as suas respectivas categorias simbólicas:

Num.	Token	Categoria Simbólica	Expressão Regular
0	pgm	PGM	`(i:"pgm")`
1	int	DLINT	`(i:"int")`
2	real	DLREAL	`(i:"real")`
3	string	DLSTRING	`(i:"string")`
4	char	DLCHAR	`(i:"char")`
5	bool	DLBOOL	`(i:"bool")`
6	array	DLARRAY	`(i:"array")`
7	if	DLIF	`(i:"if")`
8	else	ELSE	`(i:"else")`
9	while	WHILE	`(i:"while")`
10	return	DLRETURN	`(i:"return")`
11	from	FROM	`(i:"from")`
12	repeat	REPEAT	`(i:"repeat")`
13	main	MAIN	`(i:"main")`
14	end_pgm	ENDPGM	`(i:"end_pgm")`
15	to	TO	`(i:"to")`
16	true	TRUE	`(i:"true")`
17	false	FALSE	`(i:"false")`
18	print	PRINT	`(i:"print")`
19	func	FUNC	`(i:"func")`
20	step	STEP	`(i:"step")`
21	identifier	IDENTIFIER	`[letra][:alnum:]*`

22	==	EQ	`==`
23	~	UNARY	`"~"`
24	*	MULT	`"*"`
25	**	POW	`"**"`
26	+	PLUS	`"+"`
27	-	MINUS	`-"`
28	mod	MOD	`(i:"mod")`
29	div	DIV	`(i:"div")`
30	or	OR	`(i:"or")`
31	not	NOT	`(i:"not")`
32	and	AND	`(i:"and")`
33	<>	DIFF	`"<>"`
34	<	SMALLER	`"<"`
35	<=	SMALLERE	`"<="`
36	>	GREATER	`">"`
37	>=	GREATERE	`">="`
38	//	COMMENT	`"//"`
39	=	ASSIGN	`"="`
40	]	squareEnd	`"]"`
41	[	squareBeg	`"["`
42	/	DIVIDE	`"/"`
43	)	parenthEnd	`")"`
44	(	parenthBeg	`"("`
45	}	keyEnd	`"}"`
46	{	keyBeg	`"{"`
47	:	POINTS	`":"`
48	;	SEMICOLON	`";"`
49	,	COMMA	`","`
50	"	DOUBLEQUOTES	`'"'`

12 Conclusão

13 Referências bibliográficas