



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



Prof. Alcino Dall'Igna Júnior
Trabalho requisitado no semestre de 2019.2

Compiladores: Linguagem DL
Especificação

Derecky Costa da F. Andrade
Larissa Santos da Silva

Maceió, 19 de novembro de 2019

Conteúdo

1	Introdução	3
2	Estrutura Geral de um programa	4
3	Escopo	4
4	Nomes	4
4.1	Palavras Reservadas	5
4.2	Identificadores	5
4.3	Símbolos Especiais	6
4.4	Operadores	6
4.4.1	Concatenação	6
5	Tipos e Estruturas de dados	6
5.1	Forma de declaração	6
5.2	Compatibilidade	6
5.3	Constantes literais	7
5.4	Tipos de dados primitivos	7
5.4.1	Inteiro	7
5.4.2	Ponto Flutuante	9
5.4.3	Caractere	10
5.4.4	Cadeia de caractere ou String	11
5.4.5	Lógico	11
5.4.6	Arranjos unidimensionais	11
5.5	Equivalência de tipos	11
5.5.1	Coerções admitidas	12
5.5.2	Conversão de tipo explícita (cast)	12
6	Atribuição e Expressões	12
6.1	Expressões aritméticas, relacionais e lógicas	12
6.2	Precedência e Associatividade	13
7	Sintaxe e exemplo de estruturas de controle	13
7.1	Estrutura condicional de uma e duas vias	13
7.1.1	Semântica	14
7.2	Estrutura iterativa com controle lógico	14
7.2.1	Semântica	14
7.3	Estrutura iterativa controlada por contador com passo igual a um caso omitido	15
7.3.1	Semântica	15
7.4	Desvios incondicionais	15
7.4.1	Semântica	15
8	Subprogramas	16
8.1	Funções	16
8.1.1	Procedimento	16
9	Comentário	16

10 Print()	16
11 Programas exemplos	17
11.1 Alô mundo	17
11.2 Fibonacci	17
11.3 ShellSort	18
12 Especificação dos tokens	18

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 à caixa (Secção 4). 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 secção 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:

```
1 pgm
2   <variaveis>
3   <funcoes>
4   main{
5       <instrucoes>;
6   }
7 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 secção 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 \rightarrow `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, conseqüentemente não poderão ser usadas como identificador. A Tabela 1 mostra as palavras que compõem este conjunto.

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

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.

**	/	>=	<
-	*	>	=
==	<=	+	<>
and	or	not	

Tabela 4: Operadores suportados

4.4.1 Concatenação

O operador + entre um operador do tipo char ou string e outro operador, realiza a operação de concatenação, resultando em uma cadeia de caracteres. A seguir exemplos:

- char com int: 'd' + 3 => "d3"
- string com bool: "verde" + true => "verdetrue"
- char com float: 'a' + 5.6 => "a5.6"

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 nome.

Tipo	Intervalo	Exemplo
Inteiro	[-2.147.483.648, +2.147.483.647]	-1, 0, 10
Ponto flutuante	[-3.4e+38, +3.4e+38]	-1.5, 4.3
Booleano	[true, false]	true, false
Caractere	[0,127]	a,b,c, @,!
String	—	Vamos passar nesse período!
Array unidimensional	—	[1,2,3], [1.0,2.0,3.0], [true, true, false]

Tabela 5: Constantes literais por tipos de dados

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. ver Tabela 5.

5.4 Tipos de dados primitivos

Os tipos primitivos que a linguagem DL suporta são:

5.4.1 Inteiro

- Expressão regular: [0-9]+
- Declaração: **int** meuinteiro;
- Exemplos de decimais inteiros válidos: 0 5 127 -1002 65535
- Exemplos de decimais inteiros inválidos: 32,76 1.32 3A

A constante literal do inteiro não pode conter o ponto decimal, vírgulas ou espaços.

O tipo inteiro admite as seguintes operações:

- Adição e subtração (+ ou -)

Ex.:

```

1 int a, b, c, d;
2 a = 3;
3 b = 4;
4 c = a + b;
5 d = b - a;
6 main{
7     print(c + "\n" + d);
8 }

```

Obtendo o seguinte resultado como saída:


```
1 7
2 1
```

- Inverso aditivo (—)

Ex.:

```
1 int a = -10;
```

- Multiplicação e divisão (* e /)

Ex.:

```
1 int a, b, c, d;
2 a = 10;
3 b = 5;
4 c = a * b;
5 d = a / b;
6 main {
7     print(c + "\n" + d);
8 }
```

Obtendo o seguinte resultado como saída:

```
1 50
2 2
```

- exponenciação (**)

Ex.:

```
1 int a, b, c;
2 a = 2;
3 b = 5;
4 c = a ** b;
5 main {
6     print(c);
7 }
```

Obtendo o seguinte resultado como saída:

```
1 32
```

- módulo (mod)

Ex.:

```
1 int a, b, c, d;
2 a = 10;
3 b = 3;
4 c = a mod b;
5 d = b mod a;
6 main {
7     print(c + "\n" + d);
8 }
```

Obtendo o seguinte resultado

```
1 1
2 3
```

Este tipo suporta as operações relacionais.

5.4.2 Ponto Flutuante

- Expressão regular: `[[:digit:]]+.[[:digit:]]*([E|e][+|-]?[[:digit:]]+)?f?`
- Declaração: **real** meureal;
- Exemplos de pontos flutuantes válidos: `2.21e-5 -10.22 48e+8 0.5 10f`
- Exemplos de pontos flutuantes inválidos: `02.42 f22 0x5eA`

A constante literal do ponto flutuante pode conter o ponto decimal, o qualificador literal "f", e não pode conter vírgulas ou espaços. É possível usar a notação "e" para expressar valores exponenciais ($e \pm n = 10^n$). O tipo ponto flutuante admite as seguintes operações:

- Adição e subtração (+ ou -)

Ex.:

```
1 real a, b, c, d;
2 a = 3.4;
3 b = 4.6;
4 c = a + b;
5 d = b - a;
6 main{
7     print(c + "\n" + d);
8 }
```

Obtendo o seguinte resultado como saída:

```
1 8
2 1.2
```

- Inverso aditivo (-)

Ex.:

```
1 real a = -10f;
```

- Multiplicação e divisão (* e /)

Ex.:

```
1 real a, b;
2 a = 10f;
3 b = 5f;
4 c = a * b;
5 d = a / b;
6 main {
7     print(c + "\n" + d);
8 }
```

Obtendo o seguinte resultado como saída:

```
1 50.0
2 2.0
```

- exponenciação (**)

Ex.:

```
1 real a,b,c;
2 a = 2.0;
3 b = 5.0;
4 c = a ** b;
5 main {
6     print(c);
7 }
```

Obtendo o seguinte resultado como saída:

```
1 32.0
```

- módulo (mod)

Ex.:

```
1 real a, b, c, d;
2 a = 10f;
3 b = 3f;
4 c = a mod b;
5 d = b mod a;
6 main {
7     print(c + "\n" + d);
8 }
```

Obtendo o seguinte resultado

```
1 1
2 3
```

Também existe disponibilidade deste tipo para operadores relacionais.

5.4.3 Caractere

- Expressão regular: [^]
- Exemplos de caracteres válidos: 'r', 'R', '\n', '@', '2', ' ' (espaço)
- Exemplos de caracteres inválidos: me, '1 , a' , "a"

A constante literal deve estar entre apóstrofo (aspas simples) e pode conter qualquer caractere imprimível. Admite operações com operadores relacionais.

5.4.4 Cadeia de caractere ou String

- Expressão regular: "[^"]*"
- Declaração: **string** meustring;
- Exemplos de strings válidos: "MM", "Nasa", "PC", "A", "sew121@&[]"
- Exemplos de strings inválidos: 2"w", "Ola, ""

A constante literal deste tipo requisita de aspas e deve estar contida na mesma linha. Admite operações com operadores relacionais.

5.4.5 Lógico

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

- Declaração: **bool** meuboleano;

Permite operações com os operadores de igualdade e desigualdade.

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];
```

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 nominal

5.5.1 Coerções admitidas

As seguintes coerções são válidas quando as variáveis são inicializadas:

- char para int
- int para real
- char para string
- Exemplo:

```
int meuint = '5';  
real meureal = 10;  
string minhastring = 'h';
```

5.5.2 Conversão de tipo explícita (cast)

- char para int
- 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 do valor 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
- Relacional
- Lógicos

A avaliação em curto-circuito não é admitida.

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

Tabela 6: Operadores aritméticos

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

Tabela 7: Operadores relacionais

Símbolo	Operação
and	Conjunção
or	Disjunção
not	Negação

Tabela 8: Operadores lógicos

6.2 Precedência e Associatividade

Na tabela a seguir os operadores agrupados na mesma seção têm a mesma precedência, as subseqüentes 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.

7 Sintaxe e exemplo de estruturas de controle

Esses comandos oferecem instruções para tomada de decisão. Estas obedecem 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>

```

Operador	Descrição	Associatividade
()	Expressão em parêntesis	Dentro para fora
[]	Descritor de tamanho de arranjo	
-	Unário negativo	Direita para esquerda
not	NOT lógico	
**	Exponencial	
* / mod div	Multiplicação, divisão, módulo, divisão inteira	Esquerda para direita
+ -	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 9: Precedência e associatividade de operadores

```

7  }
8
9  switch <variavel> {
10     case <valor>:
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>) {
2          <instrucoes>
3      }

```

7.2.1 Semântica

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

O tipo de `expressao1` depende da declaração do identificador, enquanto o tipo de `expressao2` e `expressao3` devem ser inteiros. Esta estrutura iterativa irá executar um conjunto de instruções, enquanto o identificador varia de `expressao1` até `expressao2`, com um passo opcional declarado na `expressao3`. Para cada iteração em que `identificador ≤ expressao2`, o bloco de instruções será executado.

7.4 Desvios incondicionais

```
1 repeat cont from i to 10 step 2{
2   if(<condicao>){
3     continue;
4   }
5   if(<condicao>){
6     break;
7   }
8   <instrucoes>
9 }
10
11
12 while(i < 10){
13   if(<condicao>){
14     continue;
15   }
16   if(<condicao>){
17     break;
18   }
19   i+=2;
20 }
```

No exemplo acima, caso a condição do `if` seja `true`, o programa executará `continue` ou `break`. Caso execute `continue` o programa pulará para a próxima iteração, no caso do `break` o programa sairá do laço. Caso os `if` não tenham condição verdadeira, ele executará as instruções seguintes a esses blocos de código.

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:

```
1 func <tipoDeRetorno> <identificador ([<tipoDoParametro><
    identificadorDoParametro]*>) {
2     <conj.Instrucoes>
3     return <valorDeRetorno>
4 }
```

Observações:

- Dada uma função, esta poderá ter nenhum ou n parâmetros, no caso de mais de um parâmetros, estes devem ser separados por vírgula.
- O retorno de funções (`return`) não necessariamente deve vir no final da função, podendo vir em qualquer momento nas instruções.
- Uma função de tipo `void` é uma função sem retorno.
- Os parâmetros de funções na linguagem DL são do tipo de entrada, ou seja, seu valor não é alterado durante a execução da função.

Exemplos:

```
1 func int soma(int a, int b) {
2     int c = a + b;
3     return c;
4 }
```

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 Print()

A função `print` é uma função que imprime na tela o valor desejado. seguem exemplos de uso e saída da função `print`:

```

1  int a = 10;
2  real b = 10.0f;
3  int c[2];
4  c[0] = a;
5  c[1] = 3;
6  real d[2];
7  d[0] = b;
8  d[1] = 0.0;
9  main{
10     print(a);
11     print(b);
12     print(c);
13     print(d);
14 }

```

Obtendo como saída:

```

1  10
2  10.0
3  [10,3]
4  [10.0, 0.0]

```

11 Programas exemplos

Seguem abaixo alguns exemplos de programas em DL:

11.1 Alô mundo

```

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

```

11.2 Fibonacci

```

1  pgm
2  func void fibonacci(int n){
3      int prev, atual, temp, count;
4      prev = 0;
5      atual = 1;
6      count = 0;
7
8      while(atual < n){
9          if(count == 0){
10             print(prev + ", ");
11             count = count + 1;
12         }
13         else {
14             print(atual + ", ");
15             temp = prev + atual;
16             prev = atual;
17             atual = temp;

```

```

18     }
19 }
20 }
21
22 main{
23     int n;
24     read(n);
25     fibonacci(n);
26 }
27
28 end_pgm

```

11.3 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] = h;
21         }
22     }
23     return v;
24 }
25
26
27 main{
28     vet = [2, 3, 4, 0, 9, 7, 8, 1, 5, 6];
29
30     int x;
31     print(vet);
32     vet = shellsort(vet, 10);
33     print(vet);
34 }
35 end_pgm

```

12 Especificação dos tokens

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

Num.	Lexema	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	DLELSE	(i:"else")
9	while	DLWHILE	(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	DLTRUE	(i:"true")
17	false	DLFALSE	(i:"false")
18	print	PRINT	(i:"print")
19	func	FUNC	(i:"func")
20	step	STEP	(i:"step")
21	identifier	IDENTIFIER	[[alpha]][[: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	' "'
51	void	DLVOID	(i:"void")
52	null	DLNULL	(i:"null")
53	read	READ	(i:"read")
54	lit_int	LIT_INT	[0-9]+
55	lit_char	LIT_CHAR	[^]
56	lit_string	LIT_STRING	"[^"]*"
57	lit_bool	LIT_BOOL	
58	lit_real	LIT_REAL	[[:digit:]]+.[[:digit:]]*([E e][+ -]?[[:digit:]]+)?f?