UNIVERSIDADE FEDERAL DE SANTA CATARINA

CONSTRUÇÃO DE COMPILADORES

MANUAL DA LINGUAGEM DE PROGRAMAÇÃO

PATRICK HOECKLER

ARARANGUÁ

OUTUBRO - 2019

Sumário

1. Corpo básico do programa

Todo programa escrito para esta linguagem deve começar com a palavra *program* (exceto por comentários que podem ser escritos em qualquer parte, inclusive antes de todo o resto), seguida de um *identificador*, que pode ser um nome qualquer para o programa, e de um símbolo ponto e vírgula.

Em seguida podem ser adicionados quatro blocos de código: a declaração dos procedimentos; declaração das constantes; declaração das variáveis; e o corpo do código. Esses blocos devem ser escritos nessa mesma ordem. Por fim é necessário um ponto para indicar o fim do programa.

A seguir serão descritos os detalhes de cada bloco e da adição de comentários.

1.1 Comentários

Existem dois tipos de comentários, aqueles que espandem toda a linha em que são escritos e aqueles que espande um bloco de tamanho qualquer cercado por símbolos delimitadores. Para fazer um comentário de linha basta adicionar uma barra dupla (//) e todo texto a partir dessa barra até o fim da linha será ignorado.

Já os comentários de bloco são iniciados com uma barra seguida de uma estrela (/*) e terminados com uma estrela seguida de uma barra (*/), todo texto entre esses dois conjuntos de símbolos é ignorado.

Abaixo estão alguns exemplos de comentários válidos para esta linguagem.

```
//comentario de linha
x = y + 1;
/* comentario de bloco
  expandindo mais de uma linha
*/
x = y / 2; /*os dois comandos serão executados*/ x = y * 3;
```

1.2 Declaração de procedimentos

Essa parte do código é opcional e pode ser ignorada. Para cada procedimento a ser declarado é preciso iniciar com a palavra *procedure*, em seguida um *identificador* qualquer para nomear esse procedimento.

Depois do identificador é necessário os parâmetros desse procedimento, caso não haja nenhum não é necessário escrever nada. Caso haja parâmetros, será preciso escrever os mesmos entre parênteses seguindo o mesmo formato usado para declarar variáveis, ou seja:

```
identificador1, identificador2, … , identificadorN : tipo;
```

É possível repetir isso várias vezes dentro dos parênteses para definir quantos parâmetros forem necessários.

Depois da declaração dos parâmetros vem um bloco de declaração de variáveis, da mesma maneira como existe um bloco desses para o programa em si, a seção 1.4 explica os detalhes sobre esse bloco.

Em seguida também existe um bloco de corpo de código, que segue o mesmo padrão do bloco usado no programa. Esse bloco é descrito na seção 1.5.

Depois disto basta adicionar um ponto e vírgula e a declaração deste procedimento foi feita. É possível repetir este processo para declarar quantos procedimentos forem necessários. O código abaixo mostra um exemplo para declaração de procedimentos.

```
//procedimento com parametros
procedure proc1 (int1, int2 : integer; ch1 : char)
  //declaracao de variaveis
  //corpo de codigo
;
//procedimento sem parametros
procedure proc2
  //declaracao de variaveis
  //corpo de codigo
;
```

1.3 Declaração de constantes

Essa parte do código é opcional e pode ser ignorada. Se o bloco não for ignorado será preciso iniciar o mesmo com a palavra *const*. Para adicionar uma constante basta seguir o formato abaixo.

```
identificador = tipo;
```

É possível repetir esse comando para adicionar múltiplas constantes. A palavra *const* não deve ser repetida para adicionar mais constantes, a mesma só deve aparecer uma vez no início do

bloco (se ela aparecer é necessário declarar pelo menos uma constante). A seguir está um exemplo de declaração de constantes.

```
const
  const1 = integer;
  const2 = real;
  const3 = string;
```

1.4 Declaração de variáveis

Se não for desejado declarar variáveis essa parte pode ser ignorada sem escrever nenhum comando. Caso contrário, basta iniciar o bloco com a palavra *declaravariaveis* (será preciso declarar pelo menos uma variável nesse caso). Para declarar variáveis de um tipo qualquer basta seguir o formato a seguir.

```
identificador1, identificador2, … , identificadorN : tipo;
```

Esse comando pode ser repetido quantas vezes for necessário. A seguir está um exemplo de um bloco de declaração de variáveis.

```
declaravariaveis
  real1, real2, real3 : integer;
  st1, st2 : string;
```

Os identificadores tanto para variáveis quanto para procedimentos podem ser nomeados usando os símbolos alfanuméricos (a-z e 0-9), com diferenciação entre maiúsculas e minúsculas, e também o símbolo sublinhado (underscore). Entretanto, não é permitido usar um número como primeiro caractere do nome de um identificador.

1.5 Corpo de código

Essa parte não pode ser ignorada, é necessário começar este bloco com a palavra *begin* e terminá-lo com *end*. Tudo que estiver entre essas duas palavras são comandos para serem executados, não tem um limite para o número de comandos, todo comando deve terminar com um ponto e vírgula (inclusive um comando vazio) e o bloco precisa de pelo menos um comando. Abaixo estão dois exemplos de corpo de código.

```
//bloco que nao faz nada
begin
  ; //comando vazio
end
//bloco com alguns comandos nao vazios
begin
  x = 5 * 2;
  x = x + 1;
  y = x * 2;
end
```

Na seção 3 são apresentados os tipos de comando que podem ser colocados dentro deste bloco.

1.6 Erros Léxicos

Os erros léxicos só ocorrem quando o autômato para num estado que não é estado final (referir-se ao documento do autômato para a descrição dos estados). Sendo assim, existem oito erros diferentes que são possíveis – a seguir serão explicados esses erros.

- ERR_PLUS: ocorre quando existem mais de um sinal de + simultâneamente.
- ERR_MINUS: ocorre quando existem mais de um sinal de - simultâneamente.
- ERR_DOT: ocorre quando um sinal de + ou – é seguido de um ponto, mas esse ponto não é seguido de nenhum digito indicando um número real. Por exemplo:

```
//diferentes atribuições corretas de um numero real
x = 3.0;
x = 3.;
x = .5;
x = +3.1;
x = -2.5;
x = +4.;
x = -.5;

//diferentes atribuições que geram o erro ERR_DOT
x = +.;
x = -.;
```

- ERR_EX: uso de caracteres inválidos – a tabela abaixo mostra o conjunto desses caracteres.

Valor do byte	Equivalente na tabela ASCII
1 – 9	caracteres de controle
11 – 12	
14 – 31	
33	!
35	#
37	%
38	&
63	?
64	@
92	\
94	^
96	`
123	{
124	|
125	}
126	~
127	DEL
128 – 255	N/A

- ERR_STR: não fechar um literal de string que já foi aberto, ou seja, não colocar aspas duplas até o fim da linha atual para terminar o literal.
- ERR_CHAR: não fechar um literal de char que já foi aberto, ou seja, não colocar aspas simples até o fim da linha atual para terminar o literal.

- ERR_ESCS: utilizar uma sequência de escape inexistente dentro de um literal de string. As sequências de escape são iniciadas com uma barra invertida (\) seguidas de um caractere de escape. As possíveis sequências de escape estão na tabela abaixo (referir ao documento do autômato para mais detalhes).

Sequência	Byte resultante	Byte em ASCII
\0	0	NUL
\a	7	BEL
\b	8	BS
\t	9	TAB
\n	10	EOL
\v	11	VT
\f	12	FF
\r	13	CR
\e	27	ESC
\”	34	”
\’	38	’
\\	92	\

- ERR_ESCC: utilizar uma sequência de escape inexistente dentro de um literal de char.

Não fechar o comentário de bloco não resulta em erro léxico pois isso será considerado como um bloco até o fim do arquivo.

2. Tipos de dados

Esta linguagem aceita quatro tipos de dados: integer, real, char e string. A seguir esses tipos serão descritos.

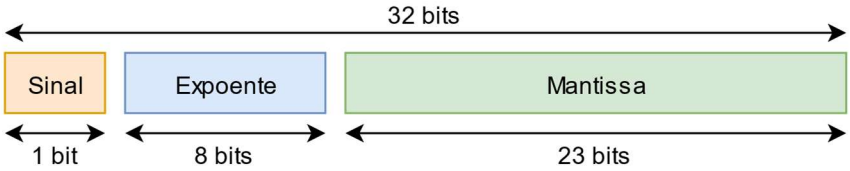
2.1 integer

Dados desse tipo representam números inteiros. Para armazenar o seu valor são utilizados 4 bytes (32 bits) e o complemento de 2, isso indica que o intervalo de valores possíveis é de -2^{31} até $2^{31} - 1$.

Um literal desse tipo é qualquer número escrito sem nenhum ponto decimal e sem nenhum caractere adicional junto com o mesmo (exceto o caracter hífen indicando número negativo). Por exemplo, são considerados integers: 123, 0, -45; enquanto que não são integers: ‘123’ (inválido), “123” (string), 123.0 (float).

2.2 real

Esse tipo de dado descreve números reais com ponto flutuante e possui tamanho de 4 bytes. Os valores são representados no formato binary32 do padrão IEEE754 aonde os 32 bits são organizados conforme a imagem abaixo.



No padrão IEEE 754, dependendo do valor da mantissa e do expoente é possível ter números de diferentes categorias conforme mostrado na tabela a seguir.

Expoente (em hexadecimal)	**Categoria**	
	Mantissa = 0	**Mantissa ≠ 0**
0x00	Zero	Número subnormal
0x01 até 0xFE	Número normal	
0xFF	±infinito	NaN

Se os bits do número forem representados por $(b_0, b_1, \dots, b_{30}, b_{31})$, aonde os bits da mantissa vão de b_0 até b_{22}, do expoente de b_{23} até b_{30} e o do sinal é o bit b_{31}, então, o valor do número da categoria subnormal pode ser obtido via a equação abaixo.

$$\text{Valor} = (-1)^{\text{sinal}} 2^{-12} \left(\sum_{i=1}^{23} b_{23-i}\, 2^{-i} \right)$$

Já para os números normais, o valor é obtido pela seguinte equação.

$$\text{Valor} = (-1)^{\text{sinal}} 2^{\text{expoente}-127} \left(1 + \sum_{i=1}^{23} b_{23-i}\, 2^{-i} \right)$$

Literais do tipo real não devem ter caracteres especiais no início ou fim do número, precisam apenas ser um número positivo ou negativo (com um hífen na frente) que inclui um ponto decimal separador. Abaixo estão alguns exemplos do uso de literais do tipo real.

```
//diferentes atribuições corretas de um numero real
x = 3.0;  //3.0
x = 3.;   //3.0
x = .5;   //0.5
x = +3.1; //3.1
x = -2.5; //-2.5
x = +4.;  //4.0
x = -.5;  //-0.5
```

2.3 char

São representados por apenas 1 byte, que permite o intervalo de 0 até 255. Serve para indicar um único caractere da tabela ASCII, mas também pode representar um número inteiro dentro de seu intervalo.

Uma variável do tipo char pode ser atribuida um número inteiro dentro de seu intervalo, entretanto, um literal escrito como número inteiro (ex: 420) é do tipo integer. Literais do tipo char precisam ser cercados por aspas simples, dentro das aspas simples podem conter um ou mais de um caractere. São considerados literais do tipo char, por exemplo: ‘a’, ‘8’, ‘ ’, ‘_’, ‘ab’, ‘8zc0’; enquanto que não são considerados char: “a” (string), 2 (integer), a (identificador).

2.4 string

Esse tipo de dado é usado para representar uma sequência de caracteres de um tamanho qualquer (exceto por literais). Os literais do tipo string correspondem a qualquer sequência de tamanho menor que 509 caracteres cercados por aspas duplas. Exemplos de literais do tipo string são: “abc”, “5 patinhos”, “2!.;o”.

2.5 array

Arrays representam um conjunto ordenado bidimensional de valores de um mesmo tipo, ou seja, são matrizes de tamanho arbitrário aonde cada elemento é um dado de um dos quatro tipos vistos anteriormente (integer, real, char, string). Para declarar uma variável de tipo array deve-se seguir o formato abaixo.

```
identificador : array [ inteiro .. inteiro ] of tipo
```

No formato acima, a palavra *inteiro* representa um número inteiro qualquer que determinará o número de linhas e colunas do array respectivamente, e o tipo pode ser qualquer um menos o tipo array.

Abaixo estão alguns exemplos da declaração de arrays.

```
//quatro exemplos da declaracao de arrays
//nao é preciso separar nada com espaços
vecChar: array [1..4] of char
vecInteger : array [1.. 10] of integer
vecString: array [1 ..4] of string
veReal : array [2 .. 5] of real
```

3. Comandos

Os comandos devem ser colocados apenas dentro de um corpo de código e representam a parte da manipulação dos dados do programa. Os possíveis comandos serão descritos a seguir.

3.1 Comandos de repetição

3.1.1 Comando while

Esse comando permite que alguns comandos sejam repetidos enquanto uma dada condição for verdadeira. A sintaxe para este comando é a seguinte.

```
while [ condição ] do begin comandos end
```

Na sintaxe acima a condição pode ser o valor de uma variável, um literal ou um resultado de uma operação (seção 4 mostra os operadores dessa linguagem). Já a palavra comandos entre *begin* e *end* representa uma quantidade qualquer de comandos que serão executados dentro desse loop enquanto a condição for verdadeira, podendo ser até outros comandos while.

Abaixo é mostrado um exemplo básico do uso do comando while.

```
while [x < y] do
begin
  x = x + 1;
end
```

3.1.2 Comando repeat

Esse comando repete uma quantidade de comandos até que uma condição seja verdadeira, este os executará pelo menos uma vez independente se a condição for verdadeira ou falsa inicialmente. A sintaxe desse comando é mostrada abaixo.

```
repeat comandos until [ condição ]
```

As palavras comandos e condição representam o mesmo que elas representam para o comando while. Um exemplo do uso deste comando é visto a seguir.

```
repeat
  x = x + 1;
until [x > 0]
```

3.1.3 Comando for

Similarmente ao while, esse comando repetirá um conjunto de comandos enquanto uma condição for verdadeira. A diferença é que antes de começar o loop, é inicializada uma variável para algum valor qualquer. A seguir está a sintaxe deste comando.

```
for [ identificador = expressão ] to [ condição ] do begin comandos end
```

Na sintaxe acima as palavras condição e comandos significam o mesmo do que para os comandos while e repeat. Já a palavra expressão representa o valor inicial que será dado para o *identificador*, esse valor pode ser simplesmente um literal, uma variável, ou o resultado de uma expressão completa (envolvendo operadores).

Um exemplo para esse comando é mostrado a seguir.

```
for [ i = 0 ] to [ i < 10 ] do
begin
  x = x * 2;
  i = i + 1;
end
```

3.2 Comando if else

Esse é um comando de decisão, ou seja, é um comando que executará um conjunto de comandos caso uma condição for verdadeira e executará outro conjunto caso contrário. A sintaxe desse comando é a seguinte.

```
if [ condição ] then begin comandos end else begin comandos end
```

Na sintaxe acima, as palavras comandos e condição representam o mesmo do que o comando while. Se a condição for verdadeira os comandos entre o *begin* e *end* do *if* serão executados, senão será os comandos do *else*. A parte do *else* pode ser omitida caso não se deseje fazer alguma coisa se a condição for falsa.

A gramática da linguagem permite a aninhação de comandos if–else, entretanto esses comandos tem que estar inteiramente dentro do corpo do if ou else, ou seja, entre o *begin* e o *end* dos mesmos. Sendo assim, não é possível um if com mais de um else, mas é possível o uso de um comando if–else dentro da área de comandos de um outro if ou else.

A seguir está um exemplo para o uso deste comando.

```
if [ x + y = 0 ] then
begin
  x = y - x;
end
else
begin
  y = x - y;
end
```

3.3 Comando read

É usado para receber input do teclado do usuário e armazenar o valor em variáveis. A sintaxe para este comando é:

```
read (identificador1, identificador2, … , identificadorN)
```

Os valores digitados pelo usuário são armazenados nas variáveis passadas para o comando. Um exemplo de uso desse comando é: `read(int1, y, z)`

3.4 Comando write

Esse comando imprime os valores passados na tela do usuário, sua sintaxe é a seguinte.

```
write (expressão1, expressão2, … , expressãoN)
```

O comando write recebe um número qualquer de expressões, que podem ser literais, variáveis, e expressões complexas envolvendo operadores. Um exemplo da utilização desse comando é: `write("Jeff", 3 * 5, x).`

OBS: não é usado nenhum literal especial para escrever mensagens no comando write, os únicos literais dessa linguagem são do tipo: integer, real, string e char. Não é preciso nenhum literal adicional só para este comando – para escrever mensagens só é necessário usar um literal string.

3.5 Comando chamaprocedure

A função desse comando é executar um procedimento definido no início do programa. A sintaxe do comando é:

```
chamaprocedure identificador parâmetros
```

No formato acima, a palavra parâmetros representa os parâmetros que o procedimento precisa para executar sua tarefa, ou seja, o programador deverá passar os mesmos de acordo com a declaração feita no bloco de declaração de procedimentos. Abaixo segue um exemplo do uso desse comando.

```
//DECLARACAO DE PROCEDIMENTOS
//procedimento com parametros
procedure soma (int1, int2 : integer;)
begin
  write(int1 + int2);
end;
//procedimento sem parametros
procedure hello
begin
 write("Hello World");
end;

//CORPO DE CODIGO DO PROGRAMA
begin
  //executa procedimento com parametros
  chamaprocedure soma(2, 5);

  //executa procedimento sem parametros
  chamaprocedure hello;
end
```

3.6 Expressões em geral

Por último, é possível colocar no corpo de código apenas expressões sem envolver nenhum dos comandos vistos anteriormente. Essas expressões servem para, por exemplo, incrementar o valor de uma variável, multiplicar dois valores e armazenar numa variável, e qualquer outra coisa que envolva manipulação dos dados. Na próxima seção serão descritos os operadores dessa linguagem e o que eles fazem.

4. Operadores

Supondo que X e Y sejam dois valores válidos na linguagem, a tabela abaixo mostra os operadores da mesma.

Categoria	Operador	Descrição
Operadores Aritméticos	X = Y	Atribuição
	X + Y	Soma
	X - Y	Subtração
	X * Y	Multiplicação
	X / Y	Divisão
Operadores Lógicos	X or Y	Ou lógico
	X and Y	E lógico
Operadores de Comparação	X = Y	Igualdade
	X <> Y	Diferença
	X < Y	Menor que
	X > Y	Maior que
	X <= Y	Menor igual
	X >= Y	Maior igual

Os operadores de atribuição e de igualdade podem ser usados em qualquer tipo de dado, entretanto, os outros operadores só podem ser usados em integer, real e char.