Alan D. Barroso, Kenji Sakata Jr

Definição da Linguagem

Universidade de São Paulo
Escola Politécnica

Linguagens e Compiladores
PCS2056

São Paulo
2013

1 Descrição informal da linguagem

A linguagem de alto nível criada para a construção deste compilador foi baseada nas linguagens de programção imperativas C e Pascal. Nesta seção iremos explicar informalmente as principais estruturas sintáticas reconhecidas pelo compilador.

A estrutura básica de um programa em nossa linguagem consiste de seis partes principais:

- Declarações de constantes
- Declarações de tipos
- Declarações de estruturas
- Declarações de variáveis globais
- Declarações de funções
- Programa principal

Cada uma dessas partes, com excessão do programa principal, não são obrigatórias e podem ser omitidas. No entanto, elas devem seguir a exata ordem indicada acima.

1.1 Declaração de constantes

As constantes do programa são declaradas da seguinte maneira:

```
1 const nome_da_variavel = valor_da_constante;
```

A constante poder ser um número (inteiro ou ponto flutuante), true, false ou um caracter envolto em apóstrofes.

1.2 Declarações de tipos

Os tipos definidos pelo usuário são definidos da seguinte maneira:

```
1 typedef tipo novo_nome;
```

Os novos tipos só podem ser definidos em cima dos tipos básicos da linguagem (bool, int, float, char) ou tipos já definidos acima dele.

1.3 Declaração de estruturas

As estruturas são os agregados heterogêneos da linguagem. Nessa linguagem, uma vez declarada uma estrutura, ela é considerada como um tipo a ser usado no resto do programa. Elas são declaradas da seguinte maneira:

```
1 struct nome_da_estrutura begin
2   declaracao_variavel1;
3   declaracao_variavel2;
4   ...
5   declaracao_variavelN;
6 end struct
```

Uma estrutura deve conter pelo menos uma variável. As declarações de variáveis seguem as mesmas regras que as declarações de variáveis globais.

1.4 Declarações de variáveis globais

As declarações de variáveis podem ser de dois tipos, variáveis simples ou agregados homogêneos. As variáveis simples seguem a seguinte estrutura:

```
1   tipo indentificador_variavel;
```

Já os agregados homogêneos:

```
1 tipo indentificador_variavel[inteiro];
```

É possível adicionar quantas vezes forem necessárias o párticula [inteiro], criando assim matrizes e não somente vetores.

Os tipos das variáveis podem ser: os tipos básicos da linguagem, os tipos definidos pelo programador ou as estruturas definidas anteriormente. As variáveis definidas nesse trecho são compartilhadas com todo resto do programa.

1.5 Definições de funções

As funções nessa linguagem são definidas como abaixo:

```
function tipo nome_funcao (tipo param1, tipo byref param2,
    ... tipo paramN) begin
  %
    Bloco interno
  %
end function
```

Os tipos das funções podem ser do mesmo tipo que os das variáveis além de um tipo a mais, o tipo void. Funções que possuem um tipo diferente do tipo void devem obrigatoriamente possuir um retorno. Funções do tipo void também podem possuir retorno, mas esse retorno deve ser obrigatóriamente vazio.

A estrutura interna de uma função segue a mesma estrutura que o programa principal, portanto, seus detalhes serão explicados a seguir.

1.6 Programa Principal

Primeiramente, o compilador aceita programas compostos por uma sequência de declarações de variáveis e, em seguida, uma sequência de comandos. O escopo do programa inicia-se com a sequência das palavras-chaves program seguida de begin. O programa acaba com as palavras end program.Tanto para as declarações de variáveis quanto para os comandos, o separador é o ponto e vírgula.

```
program begin
  %
    Espaco destinado a declaracao de variaveis.
  %
  declaracao_var1; declaracao_var2; ... declaracao_varN;

  %
    Espaco destinado a declaracao de comandos.
  %
  comando1; comando2; ... comandoN;
end program
```

A declaração de variáveis segue o mesmo estilo que o das variáveis globais.

Os comandos podem ser divididos em seis tipos:

- Comando de atribuição
- Comando de chamada de função

- Comando condicional
- Comando iterativo
- Comando de entrada
- Comando de saída
- Comando de retorno

1.6.1 Comando de Atribuição

O comando de atribuição associa o valor de uma expressaão a uma variável, explicitada à esquerda do comando. Note que a variável pode ser uma variável escalar, um ponteiro ou um vetor.

A avaliação das expressões segue as convenções usuais, sendo efetuada da esquerda para a direita. As expressões podem ser tanto booleanas quanto aritiméticas.

No caso das expressões aritiméticas, por definição, as potenciações possuem precedência sobre os produtos e divisões, e estes precedência sobre as somas e subtrações. Note que é possível alterar a prioridade de tais precedências graças ao uso de parênteses. As expressões aritiméticas aceitam números, identificadores de variáveis e o valor de retorno de chamadas de função.

Analogamente, no caso das expressões booleanas, a operação lógica "e"tem prioridade sobre a operação lógica "ou". As expressões booleanas aceitam tanto booleanos puros (true ou false) quanto o resultado de comparações entre expressões aritiméticas.

1.6.2 Comando de chamada de função

Funções podem ser consideradas como sub-programas, que recebem um conjunto de parâmetros e que são chamados pelo programa principal para executar uma dada ação. Há dois tipos de função implementadas via mesma estrutura sintática: rotinas e funções. Rotinas têm tipo void e executam seus comandos sem a necessidade de retornar algo no fim de sua execução. Já as funções são tipadas, como por exemplo int ou char, portanto, necessitam de pelo menos um return dentro de seu bloco principal.

Tanto as funções quanto as rotinas são chamadas através do nome da função requisitada, seguido dos parâmetros que devem ser passados para sua execução entre parênteses.

```
nome_funcao(parametro1 , parametro2, ..., parametroN);
```

1.6.3 Comando condicional

Refere-se à possibilidade de realizar um salto condicional segundo o resultado de uma expressão booleana.

Além das operações lógicas de "e"e "ou", as expressões booleanas consideram a possibilidade de realizar comparações lógicas entre partículas comparativas, através dos operadores "=="(igual a) ou "!="(diferente de). Tais partículas são ou booleanos ou o resultado de uma comparação entre expressões aritiméticas, efetuadas através dos operadores ">"(maior que), "<"(menor que), ">="(maior ou igual a) e "<="(menor ou igual a).

Há duas estruturas sintáticas possíveis para o comando condicional: a simples, na qual somente são executados os comandos referente à expressão booleana após a palavra reservada ïffor verdadeira, e a composta. Neste último caso, caso a comparação seja verdadeira, o comando que se encontra entre as palavras ẗhenë ëlseëerá executado. Caso contrário, o comando após o ëlseëerá executado.

```
if expressao_booleana then
  %
    Bloco interno
  %
else
  %
    Bloco interno
  %
end if
```

O bloco interno dos comando condicionais são equivalentes são aos blocos internos de funções e do programa principal.

1.6.4 Comando Iterativo

Este comando testa a expressão booleana após a palavra reservada ẅhileẗpara decidir se irá realizar os comandos que segue a palavra “do”. Esta ação solicitada será executada repetidamente até que a condição de teste não mais seja atendida.

```
while expressao_booleana do
  %
    Bloco interno
  %
```

```
5 end while
```

1.6.5 Comandos de Entrada e Saída

Os comandos de entrada e saída promovem, respectivamente, a entrada e saída de dados com relação a um meio externo. O comando de leitura captura dados e preenche o valor de uma variável, especificada após a palavra s̈can:. Já o comando de impressão permite a impressão do resultado de uma expressão.

```
1 scan variavel;
2
3 print expressao;
```

2 Exemplo de programa escrito na linguagem definida

```
function int fatorial_recursivo(int n) begin
  int result;

  if (n <=1)  then
    result = 1;
  else
    result = n * fatorial_recursivo(n - 1);
  end if

  return result;
end funcion

function int fatorial_iterativo(int n) begin
  int fatorial;

  fatorial = 1;
  while (n>0) do
      fatorial = fatorial * n;
      n = n - 1;
  end while

  return fatorial;
end function

program begin
  int  fatorial_10_recursivo;

  fatorial_10_recursivo = fatorial_recursivo(10);

  print fatorial_10_recursivo;
  print fatorial_iterativo(10);
end program
```

notacoes/program_example

3 Descrição da linguagem em BNF

```
<programa> ::= <declaracoes_constante> <definicoes_tipo> <
    definicoes_struct> <declaracoes_var> <declaracoes_func>
    program begin <declaracoes_var> <comandos> end program

<declaracoes_constante> ::= <declaracao_constante>
            | <declaracao_constante> <declaracoes_constante>

<declaracao_constante> ::= const <identificador> = <constante
    > ;
            | epsilon

<identificador> ::= <letra> | <letra><letra_dig>

<constante> ::= <numero>
       | <booleano>
       | <char>

<numero> ::= <inteiro>
       | <inteiro>.<inteiro>
       | .<inteiro>

<booleano> ::= true | false

<char> ::= '<letra_dig>' | '\<letra_dig>'

<definicoes_tipo> ::= <definicao_tipo>
          | <definicao_tipo> <definicoes_tipo>

<definicao_tipo> ::= typedef <identificador> <tipo> ;
          | epsilon

<tipo> ::= bool | int | char | float | <identificador>

<definicoes_struct> ::= <definicao_struct>
          | <definicao_struct> <definicoes_struct>

<definicao_struct> ::= struct <identificador> begin <
    declaracao_var> <declaracoes_var> end struct
          | epsilon

<declaracoes_var> ::= <declaracao_var>
          | <declaracao_var> <declaracoes_var>

<declaracao_var> ::= <tipo> <vars> ;
```

```
41          | epsilon
42
43 <vars> ::= <declaracao_array>
44     | <declaracao_array>, <vars>
45
46 <declaracao_array> ::= <identificador>
47          | <declaracao_array>[<inteiro>]
48
49 <inteiro> ::= <digito>
50       | <digito><inteiro>
51
52 <declaracoes_func> ::= <declaracao_func>
53          | <declaracao_func> <declaracoes_func>
54
55 <declaracao_func> ::= function <tipo_funcao> <identificador>
       (<declaracoes_parametro>) begin <declaracoes_var> <
       comandos> end function
56
57 <tipo_funcao> ::= void | <tipo>
58
59 <declaracoes_parametro> ::= <declaracao_parametro>
60             | epsilon
61
62 <declaracao_parametro> ::= <tipo> <declaracao_array>
63             | <tipo> byref <declaracao_array>
64             | <tipo> <declaracao_array>, <
       declaracao_parametro>
65             | <tipo> byref <declaracao_array>, <
       declaracao_parametro>
66
67 <comandos> ::= <comando> <comandos>
68       | epsilon
69
70 <comando> ::= <comando_atribuicao>
71       | <comando_condicional>
72       | <comando_iterativo>
73       | <comando_entrada>
74       | <comando_saida>
75       | <comando_retorno>
76
77 <comando_atribuicao> ::= <var> <operador_atribuicao> <ou>;
78             | <expressao>;
79
80 <operador_atribuicao> ::= =
81             | +=
82             | -=
83             | *=
84             | /=
85
86 <var> ::= <array>
87     | <array>.<var>
```

```
88
89  <array> ::= <identificador>
90      | <identificador>[<expressao>]
91
92  <expressao> ::= <ou>
93        | epsilon
94
95  <ou> ::= <e>
96      | <e> or <ou>
97
98  <e> ::= <comparacao>
99    | <e> and <comparacao>
100
101 <comparacao_booleana> ::= <soma>
102             | <soma> <operador_comparacao> <soma>
103
104 <operador_comparacao> ::= ==
105             | !=
106             | <
107             | >
108             | <=
109             | >=
110
111 <soma> ::= <multiplicacao>
112     | <multiplicacao> + <soma>
113     | <multiplicacao> - <soma>
114
115 <multiplicacao> ::= <potenciacao>
116         | <potenciacao> * <multiplicacao>
117         | <potenciacao> / <multiplicacao>
118
119 <potenciacao> ::= <operacao_unitaria>
120         | <operacao_unitaria> ^ <potenciacao>
121
122 <operacao_unitaria> ::= <particula>
123           | - <particula>
124           | ! <particula>
125
126 <particula> ::= (expressao_aritmerica)
127       | <constante>
128       | <var>
129       | <chamada_funcao>
130
131 <chamada_funcao> ::= <identificador>(<parametros>)
132
133 <parametros> ::= <parametro>
134         | epsilon
135
136 <parametro> ::= <expressao>
137       | <expressao>, <parametro>
138
```

```
139 <comando_condicional> ::= if(<expressao>) then <
    declaracoes_var> <comandos> end if
140             | if(<expressao>) then <declaracoes_var> <
    comandos> else <comandos> end if
141
142 <comando_iterativo> ::= while(<expressao>) do <comandos> end
    while
143
144 <comando_entrada> ::= scan <lista_enderecos>;
145
146 <lista_enderecos> ::= <var>
147           | <var>, <lista_enderecos>
148
149 <comando_saida> ::= print <parametro>;
150
151
152 <comando_retorno> ::= return <expressao>;
153
154 <letra> ::= _ | 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 |
      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
155
156 <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
157
158 <letra_dig> ::= <letra>
159       | <digito>
160       | <letra><letra_dig>
161       | <dig><letra_dig>
```

notacoes/BNF.txt

4 Descrição da linguagem em EBNF

```
1  programa = {declaracao_constante} {definicao_tipo} {
       definicao_struct} {declaracao_var} {declaracao_func} "
       program" "begin" {declaracao_var} {comando} "end" "program
       ".
2
3  declaracao_constante = "const" identificador "=" constante ";
       ".
4
5  identificador = letra {letra_dig}.
6
7  constante = numero | booleano | char.
8
9  numero = inteiro | ([inteiro] "." inteiro).
10
11 booleano = "true" | "false".
12
13 char = "'" (letra_dig | "\" letra_dig) "'".
14
15 definicao_tipo = "typedef" identificador tipo ";"
16
17 tipo = "bool" | "int" | "char" | "float" | identificador.
18
19 definicao_struct = "struct" identificador "begin"
       declaracao_var {declaracao_var} "end" "struct"
20
21 declaracao_var = tipo vars ";".
22
23 vars = declaracao_array ["," declaracao_array].
24
25 declaracao_array = identificador {"[" inteiro "]"}.
26
27 inteiro = digito {digito}.
28
29 declaracao_func = "function" tipo_funcao identificador "(" [
       declaracoes_parametro] ")" "begin" {declaracao_var} {
       comando} "end" "function".
30
31 tipo_funcao = "void" | tipo.
32
33 declaracoes_parametro = declaracao_parametro {","
       declaracao_parametro}.
34
35 declaracao_parametro = tipo ["byref"] declaracao_array.
36
```

```
37 comando = comando_atribuicao
38     | comando_condicional
39     | comando_iterativo
40     | comando_entrada
41     | comando_saida
42     | comando_retorno.
43
44 comando_atribuicao = (var operador_atribuicao expressao | [
       expressao]) ";".
45
46 operador_atribuicao = "=" | "+=" | "-=" | "*=" | "/=".
47
48 var = array {"." array}.
49
50 array = identificador {"[" expressao "]"}.
51
52 expressao = e {"or" e}.
53
54 e = comparacao {"and" comparacao}.
55
56 comparacao = soma {operador_comparacao soma}.
57
58 operador_comparacao = "==" | "!=" | "<" | ">" | "<=" | ">=".
59
60 soma = multiplicacao {("+" | "-") multiplicacao}.
61
62 multiplicacao = potenciacao {("*" | "/") potenciacao}.
63
64 potenciacao = operacao_unitaria {"^" operacao_unitaria}.
65
66 operacao_unitaria = operador_unitario particula.
67
68 operador_unitario = "!" | "-".
69
70 particula = ("(" expressao ")") | constante | var |
       chamada_funcao.
71
72 chamada_funcao = identificador "(" [parametros] ")".
73
74 parametros = expressao {"," expressao}.
75
76 comando_condicional = "if" "(" [expressao] ")" "then" {
       declaracao_var} {comando} ["else" {declaracao_var} {
       comando}] "end" "if".
77
78 comando_iterativo = "while" "(" [expressao] ")" "do" {
       declaracao_var} {comando} "end" "while".
79
80 comando_entrada = "scan" var {"," var} ";".
81
82 comando_saida = print parametros ";".
```

```
83
84 comando_retorno = "return" [expressao] ";".
85
86 letra = "_" | "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" | "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".
87
88 digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
     | "9".
89
90 letra_dig = (letra | digito) {letra_dig}.
```

notacoes/wirth.txt