Implementação de um Analisador Léxico

Disciplina: Compiladores

Data: 2016/02

1 Forma Normal de Backus (BNF)

Outra maneira de representar as regras de produção.
1) $\rightarrow$ é substituído por ::=

$w \rightarrow s \equiv w ::= s$

2) Os nós não terminais são palavras entre < >.
A notação BNF é usada para definir gramáticas com as características de que o lado esquerdo de cada regra é composta por um único símbolo não terminal.

Ex.

G = ({S, M, N}, {x,y}, P, S)	BNF
P = { S $\rightarrow$ x S $\rightarrow$ M M $\rightarrow$ MN N $\rightarrow$ y M $\rightarrow$ xy }	G = ({<S>, <M>, <N>}, {x,y}, P, <S>) <S> ::= x | <M> <M> ::= <M> <N> | xy <N> ::= y

Os símbolos <, >, ::= não fazem parte da linguagem!

2. Especificação de uma Linguagem Simplificada de Programação

2.1 Descrição BNF da Linguagem TINY

<PROGRAMA> ::= <DECL_SEQUENCIA>

<DECL_SEQUENCIA> ::= <DECL_SEQUENCIA> **;** <DECLARACAO> | <DECLARACAO>

<DECLARACAO> ::= <COND_DECL> | <REPET_DECL> | <ATRIB_DECL> |

<LEIT_DECL> | <ESCR_DECL>

<COND_DECL> ::= **if** <EXP> **then** <DECL_SEQUENCIA |

if <EXP> **then** <DECL_SEQUENCIA> **else** <DECL_SEQUENCIA> **end**

<REPET_DECL> ::= **repeat** <DECL_SEQUENCIA> **until** <EXP>

<ATRIB_DECL> ::= **identificador :=** <EXP>

<LEIT_DECL> ::= **read identificador**

<ESCR_DECL> ::= **write** <EXP>

<EXP> ::= <EXP_SIMPLES> <COMP_OP> <EXP_SIMPLES> | <EXP_SIMPLES>

<COMP_OP> ::= **<** | **=**

<EXP_SIMPLES> ::= <EXP_SIMPLES> <SOMA> <TERMO> | <TERMO>

<SOMA> ::= **+** | **-**

<TERMO> ::= <TERMO> <MULT> <FATOR> | <FATOR>

<MULT> ::= ***** | **/**

<FATOR> ::= **(**EXP**) | número | identificador**

COMENTÁRIOS

Uma vez que os comentários servem apenas como documentação do código fonte, ao realizar a compilação deste código faz-se necessário eliminar todo o conteúdo entre seus delimitadores

{ }

3. Palavras Reservadas e Tokens da Linguagem TINY

Tokens da linguagem TINY	
Lexema	**TOKEN**
If	IF
then	THEN
else	ELSE
end	END
repeat	REPEAT
until	UNTIL
read	READ
write	WRITE
+	PLUS
-	MINUS
*****	TIMES
/	DIV
=	EQUAL
<	LESS
(	LBRACKET
)	RBRACKET
;	DOTCOMA
:=	ATRIB
número (1 ou mais dígitos)	NUM
Identificador (1 ou mais letras)	ID

4. Exemplo de Programa

```
{programa de exemplo
  no linguagem TINY –
   computa o fatorial
}
read x;              {entrada de um inteiro}
if 0 < x then       {não calcula se x <= 0}
    fact := 1;
    repeat
       fact := fact * x;
       x := x - 10
     until x = 0;
write fact          {saída do fatorial de x}
end
```

5. Analisador Léxico

O analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequência de *token*s que o analisador sintático utiliza.

Os Algoritmos do Analisador Léxico

Uma vez defina a estrutura de dados do analisador léxico, é possível descrever seu algoritmo básico. No nível mais alto de abstração, o funcionamento do analisador léxico pode ser definido pelo algoritmo:

```
Algoritmo Analisador Léxico (Nível 0)
Inicio
      Abre arquivo fonte
      Enquanto não acabou o arquivo fonte
      Faça {
            Trata Comentário e Consome espaços
                  Pega Token
                  Coloca Token na Lista de Tokens
            }
       Fecha arquivo fonte
 Fim
```

Na tentativa de aproximar o algoritmo acima de um código executável, são feitos refinamentos sucessivos do mesmo. Durante este processo, surgem novos procedimentos, que são refinados na medida do necessário.

```
 Algoritmo Analisador Léxico (Nível 1)
 Def. token: TipoToken
 Inicio
    Abre arquivo fonte
    Ler(caracter)
    Enquanto não acabou o arquivo fonte
    Faça {Enquanto ((caracter = "{") ou
                        (caracter = espaço)) e
                         (não acabou o arquivo fonte)
            Faça { Se caracter = "{"
                    Então {Enquanto (caracter ≠ "}" ) e
                                (não acabou o arquivo fonte)
                      Faça Ler(caracter)
                      Ler(caracter)}
                    Enquanto (caracter = espaço) e
                            (não acabou o arquivo fonte)
                    Faça Ler(caracter)
                   }
            se caracter <> fim de arquivo
            então {Pega Token
                    Insere Lista}
             }
    Fecha arquivo fonte
Fim.
```

```
Algoritmo Pega Token
Inicio
   Se caracter é digito
        Então Trata Digito
        Senão Se caracter é letra
                Então Trata Identificador e Palavra Reservada
                Senão Se caracter = ":"
                        Então Trata Atribuição
                        Senão Se caracter ∈ {+,-,*}
                                Então Trata Operador Aritmético
                                Senão Se caracter ∈ {<,>,=}
                                        EntãoTrataOperadorRelacional
                                        Senão Se caracter ∈ {; ,",", (, ), .}
                                                Então Trata Pontuação
                                                Senão ERRO
Fim.
```

```
Algoritmo Trata Dígito
Def num : Palavra
Inicio
    num ← caracter
    Ler(caracter)
    Enquanto caracter é dígito
       Faça {
                num ← num + caracter
                Ler(caracter)
              }
    token.símbolo ← snúmero
    token.lexema ← num
Fim.
```

```
Algoritmo Trata Identificador e Palavra Reservada
Def id: Palavra
Inicio
    id ← caracter
    Ler(caracter)
    Enquanto caracter é letra ou dígito ou "_"
        Faça { id ← id + caracter
               Ler(caracter)
             }
    token.lexema ← id
    caso
        id = "programa" : token.símbolo ← sprograma
        id = "se" : token.símbolo ← sse
        id = "entao" : token.símbolo ← sentao
        id = "senao" : token.símbolo ← ssenao
        id = "enquanto" : token.símbolo ← senquanto
        id = "faca" : token.símbolo ← sfaca
        id = "início" : token.símbolo ← sinício
        id = "fim" : token.símbolo ← sfim
        id = "escreva" : token.símbolo ← sescreva
        id = "leia" :token.símbolo ← sleia
        id = "var" : token.símbolo ← svar
        id = "inteiro" : token.símbolo ← sinteiro
        id = "booleano" : token.símbolo ← sbooleano
        id = "verdadeiro" : token.símbolo ← sverdadeiro
        id = "falso" : token.símbolo ← sfalso
        id = "procedimento" : token.símbolo ← sprocedimento
        id = "funcao" : token.símbolo ← sfuncao
        id = "div" : token.símbolo ← sdiv
        id = "e" : token.símbolo ← se
        id = "ou" : token.símbolo ← sou
        id = "nao" : token.símbolo ← snao
        senão : token.símbolo ← sidentificador
Fim.
```

6. Trabalho Prático

6.1) Objetivo:

Implementar o analisador léxico para a linguagem TINY descrita na BNF da seção 2.

O analisador léxico receberá como entrada um arquivo no formato texto (código fonte da linguagem TINY) e retornará uma lista com os *tokens* reconhecidos da linguagem.

Por exemplo:

Entrada:

```
{ exemplo de programa na linguagem TINY}

read x ;
read y ;

soma := x + y ;

write soma
```

Saída:

1) Com sucesso:

<READ><ID><DOTCOMA><READ><ID><DOTCOMA><ID><ATRIB><ID><PLUS><ID>
<DOTCOMA><WRITE><ID>

2) Com falha:
   Indicar o tipo e localização (linha e coluna) da falha:
   2.1) Identificador inválido;
   2.2) Número inválido;
   2.3) comando ou sintaxe inválida

```
{ exemplo de programa na linguagem TINY}

r@ed x ;
read y ;

soma := x + y ;

write soma
```

Erro linha 3, coluna 2: comando ou sintaxe inválida!

Data de entrega: a definir

Trabalho Individual

Referências

Compiladores Princípios e Práticas. Kenneth C. Louden.

Compiladores. Princípios, Técnicas e Ferramentas. Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman.

Implementação de Linguagens de Programação: Compiladores. Ana Maria de Alencar Price e Simão Sirineo Toscani