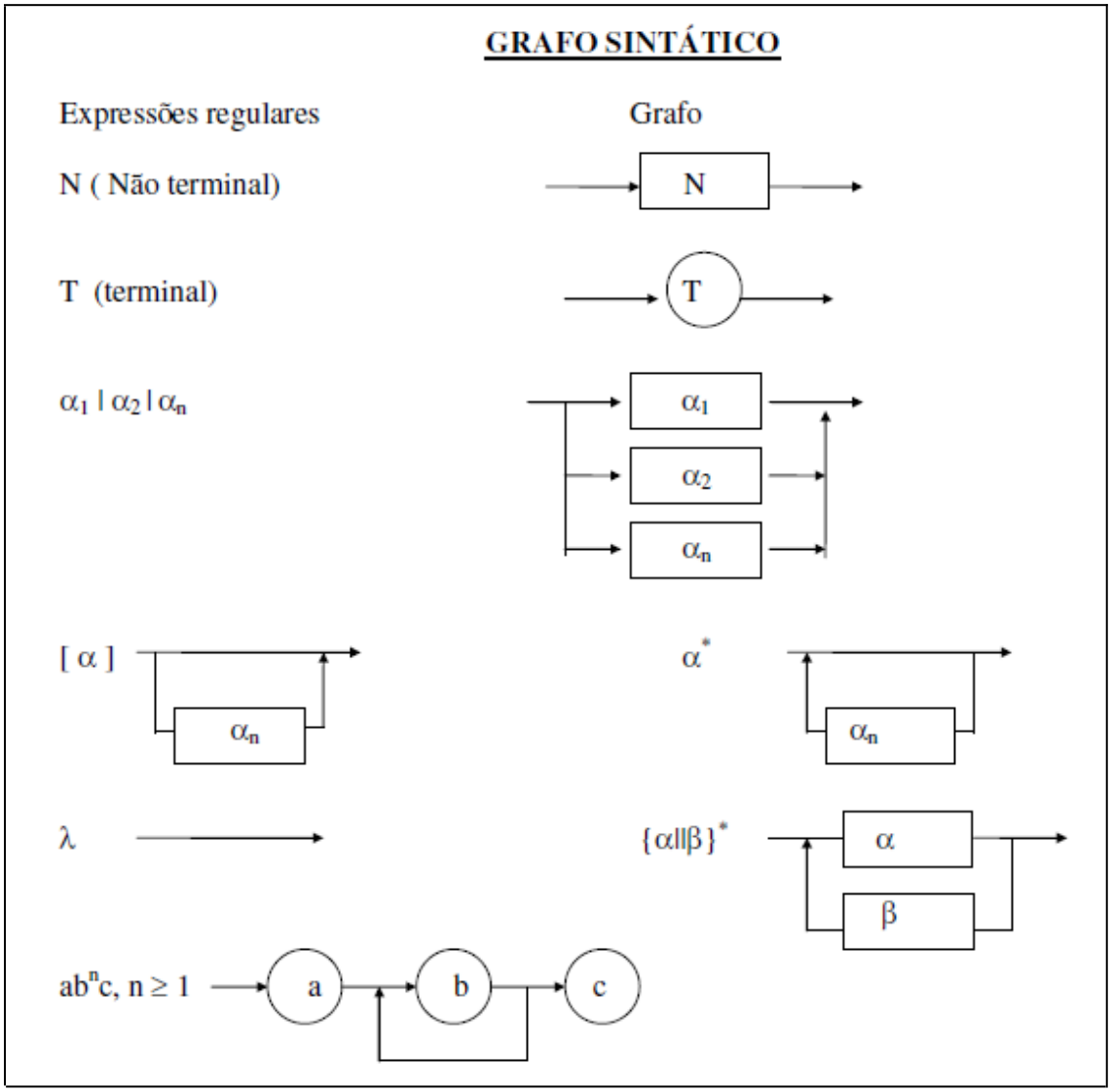
Implementação de um Analisador Léxico

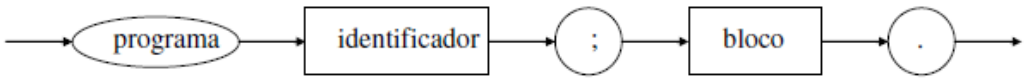
1 Grafo Sintático

Representação gráfica de elementos de uma linguagem de programação.

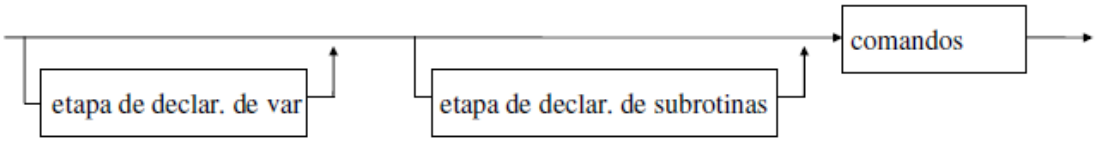


Desenvolvimento de um subconjunto do Pascal e seu grafo sintático

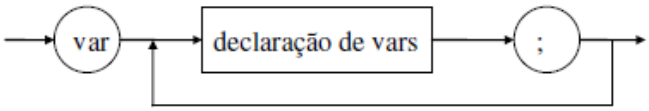
<programa> ::= programa <identificador> ; <bloco> .



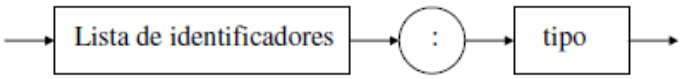
<bloco> ::= [<etapa de declar. de var.>] [<etapa de declar. de subrotinas>] <comandos>



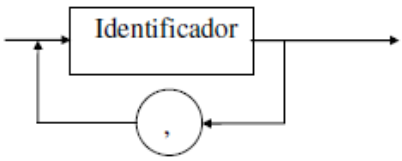
<etapa de declar. de var.> ::= var <declaração de vars> ; { <declararação de vars> ; }



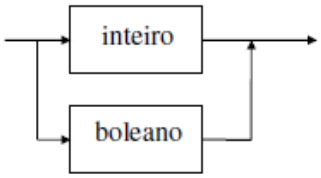
<declaração de variáveis> ::= <lista de identificadores> : <tipo>



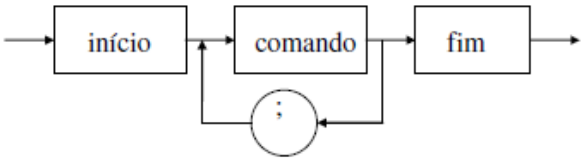
<lista de identificadores> ::= <identificador> {, <identificador>}



<tipo> ::= inteiro / booleano



<comandos> ::= início <comando> {; comando} fim



2. Forma Normal de Backus (BNF)

Outra maneira de representar as regras de produção.
1) → é substituído por ::=
  w → s ≡ 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 → x S → M M → MN N → y M → 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!

3. Especificação de uma Linguagem Simplificada de Programação

3.1 Descrição BNF da Linguagem Simplificada

A linguagem que será definida representa uma linguagem de programação estruturada semelhante à linguagem de programação estruturada PASCAL. Esta linguagem receberá o nome de **LPD** (Linguagem de Programação Didática). O compilador a ser desenvolvido receberá o nome de **CSD** (Compilador Simplificado Didático). Os símbolos não terminais de nossa linguagem serão mnemônicos colocados entre parênteses angulares < e >, sendo os símbolos terminais colocados em negrito.

Descrição BNF da Linguagem Simplificada

<programa>::= **programa** <identificador> **;** <bloco> **.**

<bloco>::= [<etapa de declaração de variáveis>]
[<etapa de declaração de sub-rotinas>]
<comandos>

DECLARAÇÕES

<etapa de declaração de variáveis>::= **var** <declaração de variáveis> **;**
{<declaração de variáveis>**;**}

<declaração de variáveis>::= <identificador> {**,** <identificador>} **:** <tipo>

<tipo> ::= (**inteiro** | **booleano**)

<etapa de declaração de sub-rotinas> ::= (<declaração de procedimento>**;**|
<declaração de função>**;**)
{<declaração de procedimento>**;**|
<declaração de função>**;**}

<declaração de procedimento> ::= **procedimento** <identificador>**;**
<bloco>

<declaração de função> ::= **funcao** <identificador>**: <**tipo>**;**
<bloco>

COMANDOS

<comandos>::= **inicio**
<comando>{**;**<comando>}[**;**]
fim

<comando>::= (<atribuição_chprocedimento>|
<comando condicional> |
<comando enquanto> |
<comando leitura> |
<comando escrita> |
<comandos>)

<atribuição_chprocedimento>::= (<comando atribuicao>|
<chamada de procedimento>)

<comando atribuicao>::= <identificador> **:=** <expressão>

<chamada de procedimento>::= <identificador>

<comando condicional>::= **se** <expressão>
entao <comando>
[**senao** <comando>]

<comando enquanto> ::= **enquanto** <expressão> **faca** <comando>

<comando leitura> ::= **leia (** <identificador> **)**

<comando escrita> ::= **escreva (** <identificador> **)**

EXPRESSÕES
<expressão>::= <expressão simples> [<operador relacional><expressão simples>]

<operador relacional>::= (**<>** | **=** | **<** | **<=** | **>** | **>=**)

<expressão simples> ::= [**+** | **-**] <termo> {(**+** | **-** | **ou**) <termo> }

<termo>::= <fator> {(***** | **div** | **e**) <fator>}

<fator> ::= (<variável> |
<número> |
<chamada de função> |
(<expressão>**)** | **verdadeiro** | **falso nao** <fator>)

<variável> ::= <identificador>

<chamada de função> ::= <identificador >

NÚMEROS E IDENTIFICADORES

<identificador> ::= <letra> {<letra> | <dígito> }

<número> ::= <dígito> {<dígito>}

<dígito> ::= (**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**)

<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)

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.

delimitadores : **{ }**

Exemplo de um programa na linguagem **LPD**

```
programa test ;
 var v , i , max, juro : inteiro ;
 inicio
      enquanto v <> 0 faca
      início
          leia ( v ) ;        {leia o valor inicial}
          leia ( juro ) ;     {leia a taxa de juros }
          leia ( max ) ;    { Leia o periodo }
          valor := 1 ;
          i := 1 ;
          enquanto i <= Max faca
           inicio
              valor := valor * ( 1 + juro ) ;
              i := i + 1 ;
          fim
          escreva(valor);
      fim
 fim.
```

4. 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.

4.1) Objetivo:

Implementar o analisador léxico para a linguagem descrita na BNF da seção 3.1.

O analisador léxico receberá como entrada o nome do arquivo a ser analisado e este deverá ser passado como um parâmetro de entrada (a exemplo do faz o gcc do Linux) via linha de comando.

Exemplo: $ gcc <nome do arquivo.extensão>

Portanto o compilador (analisador léxico) implementado deverá poder executar arquivos (programas) passados pelo usuário e não um único arquivo lido no programa.

O arquivo texto (código fonte da linguagem) e retornará uma lista encadeada com os *tokens* reconhecidos da linguagem na forma de uma lista de tuplas:

<NOME_DO_TOKEN, LINHA>

Exemplo 1:

Programa (arquivo texto)de entrada: teste1.p

```
1. { exemplo de programa }
2.
3. leia ( x ) ;
4. ...
```

Entrada: <meu_compilador> teste1.p

Saída (1) Com sucesso:):

<LEIA, 3 >
<ABREPAR, 3>
<ID, 3>
<FECHAPAR, 3>
<POTVIRG, 3>

Exemplo 2:

Programa (arquivo texto)de entrada: teste2.p

Entrada: <meu_compilador> teste2.p

```
1.  lei@ ( x ) ;
2.  soma := x + 5.0 ;
3.  escreva ( soma ) ;
```

Indicar o tipo e localização (linha) do erro:
2.1) Identificador inválido;
2.2) Número inválido.

Saída (2) Com falha:

Erro linha 1: Identificador inválido!
Erro linha 4: Número inválido!

Lista de tokens reconhecidos com sucesso:
<ABREPAR, 1>
<ID, 1>
<FECHAPAR, 1>
<POTVIRG, 1>
<ID, 2>
<ATRIB, 2>
<ID, 2>
<SOMA, 2>
<POTVIRG, 2>
<ESCREVA, 3>
<ABREPAR, 3>
<ID, 3>
<FECHAPAR, 3>
<POTVIRG, 3>

Data final para apresentação: 18/09/2019

Trabalho Individual pode ser implementado em qualquer linguagem

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.
```

Referências

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

Compiladores Princípios e Práticas. Kenneth C. Louden.

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