

MicroPascal - Parte 1

Universidade Católica de Pernambuco

Disciplina: Construção de Compiladores

Professor: Robson Lins

Projeto de Compiladores – Primeira Parte

O objetivo desta primeira parte do projeto é praticar técnicas de construção de analisadores léxicos e sintáticos. Para isso, os estudantes devem criar as duas primeiras etapas de um compilador para uma versão simplificada da linguagem Pascal, que será intitulada "micro-Pascal". As demais etapas do compilador ficarão para a próxima entrega, na segunda parte da disciplina.

O projeto pode ser desenvolvido individualmente ou em grupo de até 5 (cinco) estudantes usando a linguagem C/C++. O projeto deverá ser apresentado em data a ser combinada com o professor. Na apresentação, serão feitas perguntas individuais, então todos devem estar preparados!

As seções a seguir detalham os requisitos para cada parte do projeto.

1. Analisador Léxico (Lexer)

A subseção 1.1 detalha os tipos de tokens que deverão ser reconhecidos e retornados pelo analisador léxico. Para certos tokens, o analisador léxico deverá retornar não apenas o tipo, mas também alguma informação complementar (ex.: o lexema), para auxiliar as etapas subsequentes do compilador.

1.1 Tokens

Na definição dos tokens, assumimos a definição de *letra* e *dígito* (assim sublinhados) como indicado abaixo (atenção: eles não são tokens, esses nomes apenas serão usados para representar as expressões dadas):

```
letra  ::= [a-zA-Z_]
dígito ::= [0-9]
```

Segue abaixo a descrição dos tipos de tokens da linguagem:

- **Identificador**, dado pela expressão:

```
letra (letra | dígito)*
```

- **Operadores relacionais** (recomendo tratar como tokens diferentes):

```
< | > | <= | >= | = (igual) |  
<> (diferente)
```

- **Operadores lógico-aritméticos** (recomendo tratar como tokens diferentes):

```
+ | - | * | / (divisão de reais)  
| div (divisão de inteiros)  
| and | or | not
```

- **Operador de atribuição:**

```
:=
```

- **Símbolos especiais** (devem ser tokens diferentes):

```
) | ( | , | ; | . | :
```

- **Palavras reservadas** (devem ser tokens diferentes):

```
program | if | then | else | while | do  
| repeat
```

```
| until | integer | real | char | begin  
| end  
| write | var
```

Os seguintes operadores, também, são palavras reservadas:

```
div | and | or | not
```

- **Valor inteiro literal:**

```
digito+
```

- **Real literal:**

```
digito* . digito+
```

- **Caractere (char) literal:**

```
'(letra | digito | \n | \t)'
```

- Atenção para as aspas simples delimitando o token!
- Exemplos de lexemas válidos: 'a', 'c', '0', '9', '\n' e '\t'.
- O lexema '\n' representa quebra de linha, enquanto '\t' representa tabulação (como em C).

Em todos os casos, considere que micro-Pascal é sensível a maiúsculas e minúsculas. Portanto, um lexema "begin" representa a palavra-reservada, mas o lexema "Begin" não é a palavra reservada – ele representa um identificador. Isso porque, como você pode conferir acima, todas as palavras reservadas são escritas estritamente em letras minúsculas.

A linguagem considera como caracteres irrelevantes (brancos) o caractere de espaço e os seguintes caracteres especiais: `\n` (quebra de linha), `\t` (tabulação) e `\r` (retorno de cursor, usado antes da quebra de linha no Windows). Esses caracteres, quando aparecerem entre os lexemas, devem ser ignorados.

O lexer deverá retornar mensagem de erro caso leia um caractere que não case com o início de nenhum dos tokens. O formato das mensagens de erro é esse:

```
Erro léxico no caracter [x]
```

2. Analisador Sintático (Parser)

A subseção 2.1 apresenta a gramática abstrata da linguagem micro-Pascal. Vale lembrar que, por ser a gramática abstrata, ela provavelmente não é adequada para ser usada diretamente (como

gramática concreta) na construção do parser. Caberá ao grupo fazer modificações na gramática onde for necessário.

2.1. Gramática Abstrata

A linguagem tem uma estrutura de blocos aninhados (como em C ou em Java) e está descrita na notação BNF (Backus-Naur Form) acrescida do operador regular `*` (para indicar zero ou mais ocorrências, como em expressões regulares). Onde aparecer um trecho entre chaves, significa que o trecho vai ser tratado com o operador.

Os tokens mais simples (sinais, operadores e palavras-chave diversos) aparecem na gramática abaixo em sua forma literal (o próprio lexema). Às vezes aparecem entre aspas para evitar erros de interpretação. Já os tokens mais complexos (como o token para identificadores e os tokens para os valores literais) aparecem com um nome todo em letras maiúsculas.

```
<programa>      ::= program IDENTIFICADOR  
; <secao_var> <bloco> .  
  
<secao_var>      ::= var {<decl_var>}*  
  
<decl_var>       ::= IDENTIFICADOR {,  
IDENTIFICADOR}* : <tipo> ;
```

```

<tipo>          ::= integer | real | char

<bloco>         ::= begin <lista_comandos>
end

<lista_comandos> ::= {<comando>}*

<comando>       ::= <bloco> ;
                   | <atribuicao>
                   | <iteracao>
                   | <decisao>
                   | <escrita>

<atribuicao>     ::= IDENTIFICADOR ":="
<expressao> ;

<iteracao>      ::= while <expressao> do
<comando>
                   | repeat <comando> until
<expressao> ;

<decisao>       ::= if <expressao> then
<comando>
                   | if <expressao> then
<comando> else <comando>

<escrita>       ::= write ( <expressao> )
;

<expressao>     ::= <expressao> +
<expressao>
                   | <expressao> *
<expressao>
                   | <expressao> /

```

```

<expressao>
    | <expressao> div
<expressao>
    | <expressao> and
<expressao>
    | <expressao> or
<expressao>
    | <expressao> "="
<expressao>
    | <expressao> "<"
<expressao>
    | <expressao> "<="
<expressao>
    | <expressao> "<"
<expressao>
    | <expressao> ">="
<expressao>
    | <expressao> ">"
<expressao>
    | <expr_basica>

<expr_basica> ::= ( <expressao> )
    | not <expressao>
    | INTEIRO_LITERAL
    | REAL_LITERAL
    | CHAR_LITERAL
    | IDENTIFICADOR

```

2.2 Informações adicionais

A análise sintática deve considerar todos os operadores como associativos à esquerda,

agrupados nos seguintes níveis de precedência (do mais alto para o mais baixo):

1. `*`, `/`, `div`
2. `+`, `-`
3. `=`, `<>`, `<`, `>`, `<=`, `>=`
4. `or`, `and`

Por fim, como último requisito, a análise sintática deverá emitir mensagem de erro sempre que for necessário. O formato para as mensagens é:

```
Erro de sintaxe no token [lexema]
```

Apêndice: Exemplos de Código

Neste apêndice, apresentamos alguns exemplos de códigos-fonte válidos na linguagem micro-Pascal, para ajudar os alunos que não conhecem a linguagem Pascal.

Exemplo 1: Programa para testar se um número *n* é par ou ímpar

Veja que a linguagem micro-Pascal possui os mesmos comandos de C (com semântica igual ou parecida), mas usa uma sintaxe diferente. Algumas diferenças:

- Os programas têm um nome ("TestaParidade", neste exemplo)
- Declarações de variáveis acontecem na seção que começa com "var"
- Nas declarações de variáveis, o nome vem antes do tipo
- A divisão entre inteiros é feita com o operador "div" (o operador "/" existe em micro-Pascal, mas é para divisão entre valores reais)
- A condição de teste do "if" (e dos comandos de laço) não precisa vir entre parênteses e a condição termina na palavra reservada "then"
- A atribuição usa dois-pontos seguido de igual
- O teste de igualdade usa apenas um sinal de igual

```

program TestaParidade;
var
  n : integer;
begin
  n := 2 * 13 + 5; //número a ser testado
  if n = (n div 2)*2 then
    write('p');
  else
    write('i');
end.

```

Exemplo 2: Programa para somar os n primeiros números ímpares

Este programa não tem muitas novidades em relação ao anterior. Alguns detalhes que merecem ser destacados:

- A declaração de várias variáveis em uma só linha, assim como em C, é permitida em micro-Pascal
- Os blocos começam com "begin" (equivalente ao "{" de C) e terminam com "end" (equivalente ao "}" de C)
- Quando termina um bloco dentro do programa, coloca-se um ponto-e-vírgula depois do "end"
- Quando termina o bloco principal do programa, coloca-se um ponto depois do "end"

```
program SomaImpares;
var
  n : integer;
  i, proximoImpar, soma : integer;
begin
  n := 4; //tamanho da sequência a ser
somada
  i := 0;
  soma := 0;
  while i < n do
  begin
    proximoImpar := 2*i + 1;
    soma := soma + proximoImpar;
    i := i + 1;
  end;
  write(soma);
```

```
write('\n');  
end.
```