Universidade Federal Rural de Pernambuco
Disciplina: Compiladores
Professor: Pablo Azevedo Sampaio
Semestre: 2016.2
Última alteração: 23/09/2016

Projeto de Compiladores – Primeira Parte

O projeto será dividido em **duas partes**. Juntas, as duas partes formarão um compilador para uma linguagem nova que chamaremos de **Doxa**. Ela é inspirada em várias linguagens, mas com muitas simplificações e algumas novidades.

O projeto pode ser desenvolvido individualmente ou em dupla usando, preferencialmente, Java. Converse com o professor se desejar usar outra linguagem.

A primeira parte do projeto, tratada neste documento, consiste em construir os **analisadores léxico e sintático** da linguagem. Apenas o *lexer* (analisador léxico) pode ser desenvolvido com ferramenta (como o JFlex), porém há *requisitos extras*, neste caso. O *parser* (ou analisador sintático) deve ser desenvolvido diretamente.

Esta é a estrutura deste documento, cujas seções principais detalham os requisitos para os dois módulos:

1. Analisador Léxico (Lexer)

A próxima subseção detalha os tipos de tokens que deverão ser reconhecidos e retornados pelo *lexer* de **Doxa**.

Tokens

Segue abaixo a descrição dos tipos de tokens da linguagem:

- Um **identificador** (*string* que serve para dar nomes às variáveis e funções) é dado por esta expressão regular:
  `[a-z]` (**`[a-zA-Z_]`** | **`[0-9]`**)*

  Observação: Lexemas deste token só podem iniciar com letra minúscula. Depois, podem apresentar letras de qualquer caso, underline ou dígitos.

- Operadores relacionais:
  `<` | **`>`** | **`<=`** | **`>=`** | **`=`** | **`<>`**

- Operadores lógico-aritméticos (alguns são palavras reservadas também):
  `+` | **`-`** | **`*`** | **`/`** | **`%`** | **`and`** | **`or`** | **`not`**

- Operador de atribuição:
  `:=`

- Símbolos especiais:
  `)` | **`(`** | **`,`** | **`;`** | **`{`** | **`}`**

- Palavras-chave reservadas:
  `if` | **`else`** | **`while`** | **`return`** | **`float`** | **`char`** | **`void`** | **`prnt`** | **`int`** | **`and`** | **`or`** | **`not`** | **`proc`** | **`var`**

- Valores inteiros literais:
  `[0-9]+`

- Valores reais (de ponto flutuante) literais:
  `[0-9]*.[0-9]+` | **`[0-9]+.[0-9]*`**

- Valores caracteres literais:
  `'` (**`[0-9]`** | **`[a-zA-Z]`** | **`\n`** | **`\t`** | | **`:`** | **`(`** | **`)`** | **`,`**) **`'`**

  Observações:
  - Lexemas deste token iniciam e terminam com aspas simples. No meio pode ter um dígito ou uma letra ou \n ou \t ou um espaço.

- Exemplos de lexemas válidos: `'a'`, `'c'`, `'0'`, `'9'`, `'\n'`, `'\t'` e `' '` (este último é o caractere gerado pela barra de espaços do teclado).
- Atenção: tal como em C, o lexema `'\n'` é formado exatamente pelos quatro caracteres visíveis aqui (apesar de representar quebra de linha na linguagem Doxa).

Outros Requisitos

Doxa é **sensível a maiúsculas e minúsculas**. Portanto, um lexema "if" representa a *palavra-reservada*, mas o lexema "iF" não é a palavra reservada – ele representa um *identificador*. (Veja que todas as palavras reservadas são escritas em letras minúsculas).

A linguagem considera como **caracteres irrelevantes** (brancos) o caractere de espaço (ASCII decimal 32) e os seguintes caracteres especiais: quebra de linha (ASCII decimal 10), tabulação (ASCII decimal 9) e retorno de cursor (ASCII decimal 13, usado antes da quebra de linha no Windows). Quando aparecerem na entrada, esses caracteres devem ser ignorados (não formando um lexema de nenhum token).

O *lexer* deverá retornar **mensagem de erro** caso leia um caractere que não case com o início de nenhum dos tokens. No mínimo, a mensagem deve informar o caractere que causou o erro.

Requisito Extra

Quem usar uma ferramenta (como **lex** ou **JFlex**) para gerar o *lexer* tem os seguintes **requisitos extras**:

1) indicar a linha e a coluna onde ocorreu um erro léxico;
2) aceitar comentários.

A linguagem **Doxa** permitirá dois tipos de **comentários**. Um deles é o comentário de linha, que inicia por "******" e ignora tudo que tiver até o final da linha (semelhante ao "//" de C). O outro tipo é o comentário multi-linha, que inicia com "**>>**" e ignora tudo que tiver até o primeiro "**<<**" (semelhante aos comentários entre "/*" e "*/", de C).

2. Analisador Sintático (Parser)

A seguir, apresentamos a **gramática abstrata** da linguagem **Doxa**. Ela não é adequada para ser usada diretamente (como *gramática concreta*) na construção do *parser*. Caberá ao grupo fazer as modificações necessárias.

Gramática Abstrata

A linguagem tem uma estrutura de blocos aninhados (como em C) e está descrita na notação **EBNF** que conta com os operadores de expressões regulares. Parênteses são usados para agrupar o trecho no qual um operador regular (como *) é usado.

Os tokens mais simples (sinais, operadores e palavras-chave) são representados pelo próprio lexema entre aspas. Já os tokens mais complexos (com vários lexemas possíveis) aparecem com um nome todo em letras maiúsculas. O não-terminal inicial da gramática é **programa**.

```
programa = ( decl_global )*

decl_global = decl_variavel
            | decl_funcao

decl_variavel = "var" lista_idents "-" tipo ";"

lista_idents = IDENTIFICADOR ("," IDENTIFICADOR)*

tipo  = "int" | "char" | "float"

decl_funcao = "proc" nome_args "-" tipo bloco
            | "proc" nome_args          bloco

nome_args = ( IDENTIFICADOR "(" param_formais ")" )+

param_formais = IDENTIFICADOR "-" tipo ( "," IDENTIFICADOR "-" tipo )*
              | ε

bloco = "{" lista_comandos "}"

lista_comandos = (comando)*

comando = decl_variavel
        | atribuicao
        | iteracao
        | decisao
        | escrita
        | retorno
        | bloco
        | chamada_func_cmd

atribuicao = IDENTIFICADOR ":=" expressao ";"

iteracao = "while" "(" expressao ")" comando

decisao = "if" "(" expressao ")" comando "else" comando
        | "if" "(" expressao ")" comando

escrita = "prnt" "(" lista_exprs ")" ";"
```

Lembrar:
(**r**)+ : um ou mais
(**r**)* : zero ou mais

```
chamada_func_cmd = chamada_func ";"

retorno = "return" expressao ";"

chamada_func = ( IDENTIFICADOR "(" lista_exprs ")" )+

lista_exprs = ε
            | expressao ("," expressao)*

expressao = expressao "+" expressao
          | expressao "-" expressao
          | expressao "*" expressao
          | expressao "/" expressao
          | expressao "%" 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" expr_basica
            | "-" expr_basica
            | INT_LITERAL
            | CHAR_LITERAL
            | FLOAT_LITERAL
            | IDENTIFICADOR
            | chamada_func
```

Outros Requisitos

A análise sintática deverá **emitir mensagem de erro** sempre que encontrar algo inesperado no código dado como entrada. No mínimo, a mensagem deve informar o *token* onde ocorreu o erro. Se o *lexer* for feito com alguma ferramenta, o *parser* deve **informar linha e coluna** também.

Considere, ainda, que todos os operadores binários são associativos **à esquerda** e que os operadores obedecem aos seguintes **níveis de precedência**, do maior (1) para o menor (5):

1. **`not , -`** (menos unário)
2. **`*, /,`** `%` (resto da divisão)
3. **`+, -`** (menos binário)
4. **`=, <>, <, >, <=, >=`**
5. **`or, and`**

Observação: Para facilitar o desenvolvimento, pode tratar os operadores como recursivos à direita.

Apêndice: Exemplos de Código

Neste apêndice, apresentamos alguns exemplos de códigos-fonte válidos na linguagem **Doxa**, para ajudá-los a entender. Comparando com C, algumas características de **Doxa** são:

- Assim como em C, "main" indica a função principal, porém, em **Doxa**, não precisa especificar o tipo de retorno de uma função que não retorna valor.
- A declaração de variáveis começa com a palavra reservada "var", e o tipo é especificado no final, separado por "-" (menos).
- Há um comando padrão para imprimir, que é similar a chamar uma função de nome "prnt". Este comando aceita vários parâmetros.

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

Este programa exemplifica:

- Comandos e expressões básicos.
- Comentários.

```
proc main()
{
   var n, nRebuilt - int;

   n := 51423;  ** numero a ser testado

   >> A divisao de inteiros arredonda para baixo (em caso
      de divisao inexata). Assim, numeros impares ficarao
      com uma unidade a menos do que seu valor inicial. <<

   nRebuilt := (n / 2) * 2;

   if (n = nRebuilt)
      prnt('P', 'A', 'R');
   else
      prnt('I', 'M', 'P', 'A', 'R');

}
```

Exemplo 2: Programa para somar os *n* primeiros ímpares positivos.

Este programa exemplifica:

- Variáveis globais.
- Chamada de função.
- Passagem de parâmetro. Veja que a sintaxe pode ser semelhante a de C, ou você pode especificar um nome composto intercalado com argumentos. Assim, Doxa permite nomes como "`adiciona(e)naPilha(p)`", "`elevar(2)a(10)`", etc.
- Retorno de valor.

```
var n, soma - int;

proc main()
{
    var result - int;

    result = soma(9)primeirosImpares();

    prnt(result);
}

proc soma(n - int)primeirosImpares() - int
{
   var i, proxImpar, resultado - int;

   resultado := 0;
   i := 0;

   while (i < n) {
      proxImpar := 2*i + 1;   ** o i-esimo impar positivo
      resultado := resultado + proxImpar;
      i := i + 1;
   }

   return resultado;
}
```

Exemplo 3: Programa para calcular o **mdc** (máximo divisor comum) de dois inteiros *x* e *y* (desde que não sejam ambos nulos, ou seja, $xy \neq 0$).

Este programa exemplifica:

- Passagem de vários parâmetros para uma mesma função.
- Funções usadas como expressões.
- Função recursiva.

```
proc main()
{
   var x, y - int;
   var m - int;

   x := 120;  >> dois valores a partir dos quais <<
   y := 640;  >> sera calculado o m.d.c.         <<

   m := mdcDe (x) e (y);

   prnt('m', 'd', 'c', '(');
   prnt( x );
   prnt(',');
   prnt( y );
   prnt(')', ':');
   prnt( m );

}

proc mdcDe(x - int)e(y - int) - int
{
   if (y = 0) {
      return x;
   } else {
      return mdcDe(y)e(x % y);
   }

}
```