Universidade Federal Rural de Pernambuco
Disciplina: Compiladores
Professor: Pablo Azevedo Sampaio
Semestre: 2015.2
Última alteração: 14/09/2015

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 **Mitte**. Ela é inspirada em várias linguagens, em especial as linguagens C e Python, porém com muitas simplificações.

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 Mitte. 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 Mitte.

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:
  `[0-9]*[a-zA-Z]([a-zA-Z]|[0-9])*`

  Observações: Lexemas deste token podem começar com dígitos, depois tem uma letra (obrigatoriamente), depois pode apresentar letras e números intercalados em qualquer quantidade.

- 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` | `def` | `print` | `int` | `and` | `or` | `not` | `string` | `call`

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

- Valores reais (de ponto flutuante) literais:
  `[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).
- O lexema: `'\n'` representa quebra de linha, enquanto `'\t'` representa tabulação, tal como em C.

- Valores strings literais:
  `"([0-9]|[a-zA-Z]|\n|\t| |,|(|)|:)*"`

  Observações:
  - Exemplos de lexemas válidos: `"Hello World"`, `"success"`, etc.

Outros Requisitos

Mitte **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 estritamente 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). Esses caracteres, quando aparecerem na entrada, 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 Mitte permitirá dois tipos de **comentários** idênticos aos de C. Um deles é o comentário de linha, que inicia por "**##**" e ignora tudo que tiver até o final da linha. O outro tipo é o comentário multi-linha, que inicia com "**(***" e ignora tudo que tiver até o primeiro "***)**".

2. Analisador Sintático (Parser)

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

Gramática Abstrata

A linguagem tem uma estrutura de blocos aninhados (como em C) e está descrita na notação **BNF** acrescida do operador regular * (para indicar zero ou mais ocorrências, como em expressões regulares). Parênteses são usados para agrupar o trecho no qual o operador * 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> ::= <lista_idents> ":" <tipo> ";"

<lista_idents> ::= IDENTIFICADOR ("," IDENTIFICADOR)*

<tipo>  ::= "int" | "char" | "float" | "string"

<decl_funcao> ::= <assinatura> <bloco>

<assinatura> ::= "def" IDENTIFICADOR "(" <param_formais> ")" ":" <tipo>
               | "def" 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> ::= <lista_idents> "=" <expressao> ";"

<iteracao> ::= "while" "(" <expressao> ")" <comando>

<decisao> ::= "if" "(" <expressao> ")" <comando> "else" <comando>
            | "if" "(" <expressao> ")" <comando>
```

```
<escrita> ::= "print" "(" <expressao> ")" ";"

<chamada_func_cmd> ::= "call" <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> "||" <expressao>
              | <expressao> "==" <expressao>
              | <expressao> "!=" <expressao>
              | <expressao> "<=" <expressao>
              | <expressao> "<" <expressao>
              | <expressao> ">=" <expressao>
              | <expressao> ">" <expressao>
              | <expr_basica>

<expr_basica> ::= "(" <expressao> ")"
                | "!" <expressao_basica>
                | "-" <expressao_basica>
                | INT_LITERAL
                | CHAR_LITERAL
                | FLOAT_LITERAL
                | STRING_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`**
2. **`*`**, **`/`**, **`%`** (resto da divisão)
3. **`+`**, **`-`** (menos binário)
4. **`==`**, **`!=`**, **`<`**, **`>`**, **`<=`**, **`>=`**
5. **`or`**, **`and`**

Apêndice: Exemplos de Código

Neste apêndice, apresentamos alguns exemplos de códigos-fonte válidos na linguagem **Mitte**, para ajudar a entender os códigos desta linguagem. Observe que, comparando com C, algumas diferenças são:

- Na declaração de variáveis e na definição dos argumentos de funções, o tipo vem depois da lista de identificadores (o inverso de C).
- A declaração de uma função (assinatura) começa com "def". O tipo vem no fim da assinatura, mas, se não retornar valor não coloca-se nada (em C, informa-se "void" no lugar do tipo).
- Um comando de chamada de função inicia com a palavra-chave "call". Porém, se a função for usada como uma expressão, ela é chamada como em C (sem "call").

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

Este programa exemplifica:

- Comandos e expressões básicos.
- Comentários.

```
def main()
{
   n, nRebuilt : int;
   msg : string;

   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 valor inicial. *)

   nRebuilt = (n / 2) * 2;

   if ( n == nRebuilt )
      msg = "par";
   else
      msg = "impar";

   print(msg);
}
```

Observação: Assim como em C, o nome "main" indica a função principal do programa.

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

Este programa exemplifica:

- Variáveis globais.
- Passagem de parâmetro e retorno de valor.
- Função usada como comando.

```
n, soma : int;

def main()
{
   n = 9;  ## quantidade de numeros impares positivos

   call somaImpares(n);

}

def somaImpares(n: int) : int
{
   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;
   }

   print(resultado);
   print('\n');

   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.

```
def main()
{
   x, y : int;
   m    : int;

   x = 120;  (* dois valores a partir dos quais *)
   y = 640;  (* sera calculado o m.d.c.         *)

   m = mdcEuclides(x, y);

   print("mdc(");
   print( x );
   print(',');
   print( y );
   print(")=");
   print( m );

}

def mdcEuclides(x: int, y: int) : int
{
   if (y == 0) {
      return x;
   } else {
      return mdcEuclides(y, x % y);
   }

}
```