Projeto de Implementação de um Compilador para a Linguagem TPP: Análise Léxica (Trabalho – 1ª parte)

Autor: Alexandre Aparecido Scrocaro Junior
email: alescrocaro@gmail.com

Professor: Rogério Aparecido Gonçalves
Universidade Tecnológica Federal do Paraná (UTFPR)

Análise Léxica

Resumo

A primeira parte do trabalho consiste na implementação de um analisador léxico - também chamado de *scanner* ou sistema de varredura - para a linguagem TPP, para tanto, foi utilizada a documentação da linguagem disponibilizada pelo professor. A linguagem/ferramenta utilizada foi Python/PLY, além de fazer uso de expressões regulares para analisar os *tokens*.

Especificação da linguagem de programação TPP

- Tipos básicos de dados suportados: **inteiro** e **flutuante**
- Suporte a arranjos uni e bidimensionais **(*arrays*)**
  - Exemplos:
    - tipo: identificador[dim]
    - tipo: identificador[dim][dim]
- Variáveis locais e globais devem ter um dos tipos especificados
- Tipos de funções podem ser omitidos (quando omitidos viram um procedimento e um tipo *void* é devolvido explicitamente
- Linguagem quase fortemente tipificada: nem todos os erros são especificados mas sempre deve ocorrer avisos
- Operadores aritméticos: +, -, * e /
- Operadores lógicos: e (&&), ou (||) e não (!)
- Operador de atribuição: recebe (:=).
- Operadores de comparação: maior (>), maior igual (>=), menor (<), menor igual (<=), igual (=).

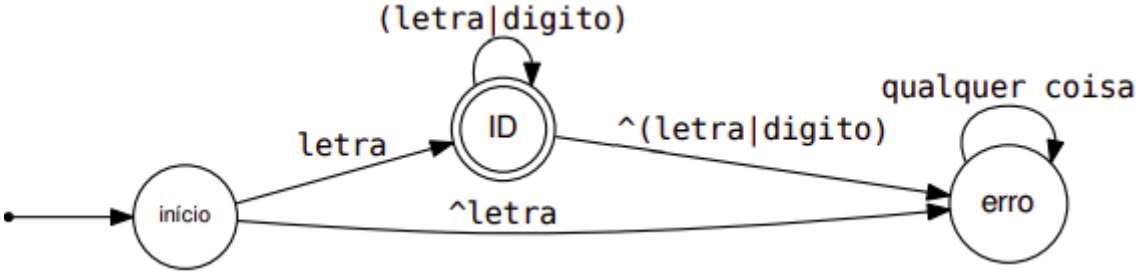
Especificação formal dos autômatos para a formação de cada classe de token da linguagem

Autômato de identificador:

Primeiramente, as classes mais simples:

- letra = [a-zA-Z]+
- digito = [0-9]+

Agora, podemos criar um autômato de um identificador da seguinte forma:

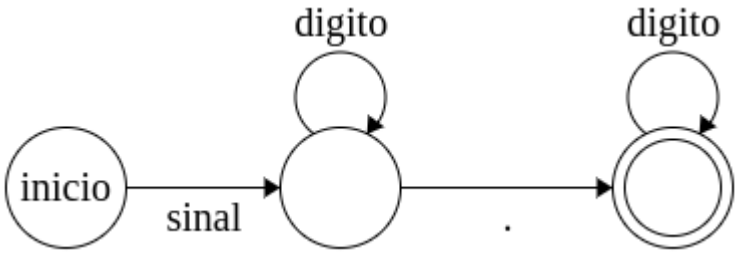


Autômato de número flutuante:

Primeiramente, as classes mais simples:

- sinal = [+-]* (o sinal pode ter nada, é repassado com épsilon)
- digito = [0-9]+

Agora, podemos criar um autômato de um número flutuante da seguinte forma:



Detalhes da implementação da varredura na LP e ferramenta (e/ou bibliotecas) escolhidas pelo projetista

O sistema de varredura, ou analisador léxico, é a fase do compilador na qual se lê o código-fonte como um arquivo de caracteres e o separa em um conjunto de *tokens*, os quais são reconhecidos através das expressões regulares. A implementação do analisador será explicada a seguir.
Em primeiro lugar, define-se os *tokens*:

```
tokens = [
    "ID",  # identificador
    # numerais
    "NUM_NOTACAO_CIENTIFICA",  # ponto flutuante em notaçao científica
    "NUM_PONTO_FLUTUANTE",  # ponto flutuate
    "NUM_INTEIRO",  # inteiro
    # operadores binarios
    "MAIS",  # +
    "MENOS",  # -
    "MULTIPLICACAO",  # *
    "DIVISAO",  # /
    "E_LOGICO",  # &&
    "OU_LOGICO",  # ||
    "DIFERENCA",  # <>
    "MENOR_IGUAL",  # <=
    "MAIOR_IGUAL",  # >=
    "MENOR",  # <
    "MAIOR",  # >
    "IGUAL",  # =
    # operadores unarios
    "NEGACAO",  # !
    # simbolos
    "ABRE_PARENTESE",  # (
    "FECHA_PARENTESE",  # )
    "ABRE_COLCHETE",  # [
    "FECHA_COLCHETE",  # ]
    "VIRGULA",  # ,
    "DOIS_PONTOS",  # :
    "ATRIBUICAO",  # :=
    # 'COMENTARIO',  # {***}
]
```

Também define-se as palavras reservadas:

```
reserved_words = {
    "se": "SE",
    "então": "ENTAO",
    "senão": "SENAO",
    "fim": "FIM",
    "repita": "REPITA",
    "flutuante": "FLUTUANTE",
    "retorna": "RETORNA",
    "até": "ATE",
    "leia": "LEIA",
    "escreva": "ESCREVA",
    "inteiro": "INTEIRO",
}
```

Assim como os símbolos e operadores que serão utilizados para reconhecer as facilidades da linguagem TPP, e suas expressões regulares:

```
# Expressões Regulaes para tokens simples.
# Símbolos.
t_MAIS = r'\+'
t_MENOS = r'-'
t_MULTIPLICACAO = r'\*'
t_DIVISAO = r'/'
t_ABRE_PARENTESE = r'\('
t_FECHA_PARENTESE = r'\)'
t_ABRE_COLCHETE = r'\['
t_FECHA_COLCHETE = r'\]'
t_VIRGULA = r','
t_ATRIBUICAO = r':='
t_DOIS_PONTOS = r':'

# Operadores Lógicos.
t_E_LOGICO = r'&&'
t_OU_LOGICO = r'\|\|'
t_NEGACAO = r'!'

# Operadores Relacionais.
t_DIFERENCA = r'<>'
t_MENOR_IGUAL = r'<='
t_MAIOR_IGUAL = r'>='
t_MENOR = r'<'
t_MAIOR = r'>'
t_IGUAL = r'='
```

```
digito = r"([0-9])"
letra = r"([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])"
sinal = r"([\-\+]?)"
```

```
    """
        id deve começar com uma letra
    """
    id = (
        r"(" + letra + r"(" + digito + r"+|_|" + letra + r")*)"
    )  # o mesmo que '((letra)(letra|_|([0-9]))*)'
```

```
inteiro = r"\d+"
```

```
flutuante = (
    # r"(" + digito + r"+\." + digito + r"+?)"
    # (([-\+]?)([0-9]+)\.([0-9]+))'
    r'\d+[eE][-+]?\d+|(\.\d+|\d+\.\d*)([eE][-+]?\d+)?'
    # r'[-+]?[0-9]+(\.([0-9]+)?)'
    # r'[+-]?(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?'
    # r"(([-\+]?)([0-9]+)\.([0-9]+))"
)

notacao_cientifica = (
    r"(" + sinal + r"([1-9])\." + digito + r"+[eE]" + sinal + digito + r"+)"
)  # o mesmo que '(([-\+]?)([1-9])\.([0-9])+[eE]([-\+]?)([0-9]+))'
```

Após isso, foram definidas as funções para reconhecer as classes (ID - que requer atenção especial para não coincidir com nenhuma palavra reservada, notação científica, número de ponto flutuante, número inteiro, comentários, novas linhas e colunas):

```
@TOKEN(id)
def t_ID(token):
    token.type = reserved_words.get(
        token.value, "ID"
    )  # não é necessário fazer regras/regex para cada palavra reservada
    # se o token não for uma palavra reservada automaticamente é um id
    # As palavras reservadas têm precedências sobre os ids

    return token


@TOKEN(notacao_cientifica)
def t_NUM_NOTACAO_CIENTIFICA(token):
    return token


@TOKEN(flutuante)
def t_NUM_PONTO_FLUTUANTE(token):
    return token


@TOKEN(inteiro)
def t_NUM_INTEIRO(token):
    return token


t_ignore = " \t"

# para poder contar as quebras de linha dentro dos comentarios
# t_COMENTARIO = r'(\{((.|\n)*?)\})'


def t_COMENTARIO(token):
    r"(\{((.|\n)*?)\})"
    token.lexer.lineno += token.value.count("\n")
    # return token


def t_newline(token):
    r"\n+"
    token.lexer.lineno += len(token.value)


def define_column(input, lexpos):
    begin_line = input.rfind("\n", 0, lexpos) + 1
    return (lexpos - begin_line) + 1
```

Além disso, também são reconhecidos erros de caracteres especiais que a linguagem não contém (como $ e ç) e é mostrada sua posição no código (linha e coluna):

```
def t_error(token):
    # file = token.lexer.filename
    line = token.lineno
    column = define_column(token.lexer.lexdata, token.lexpos)
    message = "Caracter inválido '%s'" % token.value[0]

    # print(f"[{file}]:[{line},{column}]: {message}.")

    # print(message)
    # print abaixo mostra linha e coluna do erro
    print(f"[{line},{column}]: {message}")

    token.lexer.skip(1)

    #token.lexer.has_error = Trueb
```

Por último, o código retorna a lista de tokens correspondentes ao arquivo escolhido por quem executa o comando para iniciá-lo, percorrendo todo o código TPP do arquivo selecionado.

```
def imprimir_tokens():
    while True:
        tok = lexer.token()
        if not tok:
            break      # No more input
        # print(tok)

        # print para mostrar o token
        print(tok.type)

        # print(tok.value)


def main():
    #argv[1] = 'teste.tpp'
    aux = argv[1].split('.')
    if aux[-1] != 'tpp':
        raise IOError("Not a .tpp file!")
    data = open(argv[1])

    source_file = data.read()
    lexer.input(source_file)

    # Tokenize
    imprimir_tokens()
```

A ferramenta utilizada foi o Python PLY, que é uma implementação do lex e yacc e utiliza LR-parsing - que é razoavelmente eficiente. O lex possui a facilidade na tokenização de uma *string* de entrada, por exemplo:

***String* de entrada:**
x = 3 + 42 * (s - t)\

***String* após tokenização:**

'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'

***Tokens* obtidos:**

ID, IGUAL, INTEIRO, MAIS, INTEIRO, VEZES, ABRE_PARENTESE, ID, MENOS, ID, FECHA_PARENTESE

Exemplos de saída do sistema de varredura (lista de tokens) para exemplos de entrada (código fonte)

Código para verificar valor 10:

ENTRADA:

```
inteiro principal()
  inteiro: a
  a:=1
  repita
    se a=10
      escreva("valor 10")
    fim
    a++
  ate a=10

  reutn(0)
fim
```

SAIDA:

INTEIRO
ID
ABRE_PARENTESE
FECHA_PARENTESE
INTEIRO
DOIS_PONTOS
ID
ID
ATRIBUICAO
NUM_INTEIRO
REPITA
SE
ID
IGUAL
NUM_INTEIRO
ESCREVA
ABRE_PARENTESE
[6,12]: Caracter inválido ' " '
ID
NUM_INTEIRO
[6,21]: Caracter inválido ' " '
FECHA_PARENTESE
FIM
ID
MAIS
MAIS
ID
ID
IGUAL
NUM_INTEIRO
ID
ABRE_PARENTESE
NUM_INTEIRO
FECHA_PARENTESE
FIM

Código de produto escalar:

ENTRADA:

```
inteiro escalar(inteiro : array1[], inteiro : array2[], inteiro : n)
  inteiro : produto
  produto := 0
  inteiro : i
  i := 0
  se n > 0 entao
    repita
      produto := produto + array1[i] * array2[i]
      i := i + 1
    ate i = n
  retorna (produto)
fim
```

SAIDA:

INTEIRO
ID
ABRE_PARENTESE
INTEIRO
DOIS_PONTOS
ID
ABRE_COLCHETE
FECHA_COLCHETE
VIRGULA
INTEIRO
DOIS_PONTOS
ID
ABRE_COLCHETE
FECHA_COLCHETE
VIRGULA
INTEIRO
DOIS_PONTOS
ID
FECHA_PARENTESE
INTEIRO
DOIS_PONTOS
ID
ID
ATRIBUICAO
NUM_INTEIRO
INTEIRO
DOIS_PONTOS
ID
ID
ATRIBUICAO
NUM_INTEIRO
SE
ID
MAIOR
NUM_INTEIRO
ID
REPITA
ID
ATRIBUICAO
ID
MAIS
ID

ABRE_COLCHETE
ID
FECHA_COLCHETE
MULTIPLICACAO
ID
ABRE_COLCHETE
ID
FECHA_COLCHETE
ID
ATRIBUICAO
ID
MAIS
NUM_INTEIRO
ID
ID
IGUAL
ID
RETORNA
ABRE_PARENTESE
ID
FECHA_PARENTESE
FIM

Implemente uma função que imprima a lista de tokens, não utilize a saída padrão da ferramenta de implementação de Analisadores Léxicos

Para tanto, basta tokenizar o arquivo de entrada e imprimir todos os *tokens* obtidos, que é feito na função a seguir.

```
def imprimir_tokens():
    while True:
        tok = lexer.token()
        if not tok:
            break      # No more input
        # print(tok)

        # print para mostrar o token
        print(tok.type)
```