Escola Politécnica da Universidade de São Paulo
Departamento de Computação e Sistemas Digitais - PCS

PCS2056 - Linguagens e Compiladores
Atividade 1: Analisador Léxico

Diego Henrique dos Reis Marques - nusp 7157321
Felipe Hamamoto Toyoda - nusp 7630228

São Paulo, Setembro de 2015.

Exercícios

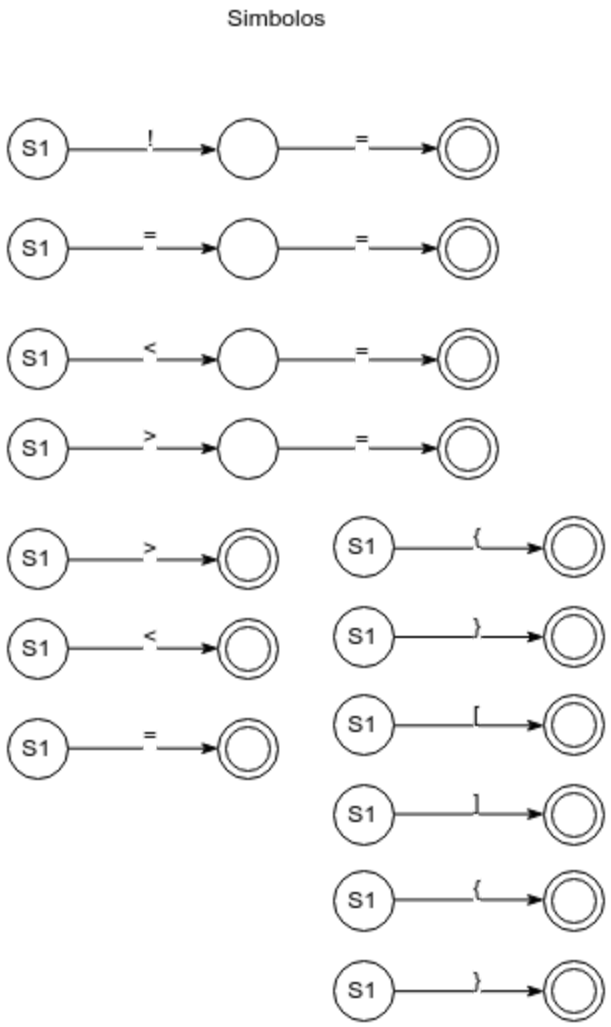
1. O analisador léxico nos compiladores/interpretadores tem como função principal a leitura de caracteres do código-fonte de entrada do compilador/interpretador e, dada a leitura de cada caracter, interpretá-lo como um *token*, por meio de autômatos capazes de reconhecer os caracteres de entrada e retornar os átomos correspondentes a eles, que servirão de entrada para o analisador sintático, responsável por interpretá-los como funções/ estruturas de código de fato.
2. As vantagens de se ter o analisador léxico como um módulo isolado do compilador são o fato de poder trabalhar com a entrada independentemente dos módulos subsequentes do compilador, de modo que seu trabalho não está atrelado à uma sub-rotina anterior que pode "engasgar" lendo a entrada (o código-fonte). Sem falar que, sendo modular, pode ser reaproveitado por outras estruturas similares, alterando-se apenas a forma como retorna os *tokens*, caso necessário.
   As desvantagens são o fato de, por não ser uma sub-rotina, caso haja um erro de leitura em tempo de execução, localizar a causa do erro pode ser mais difícil, pois há uma demanda maior de arquivos-cabeçalho e/ou arquivos pré-processados para serem trabalhados com, e no caso, os módulos subsequentes terão que esperar a obtenção de todos os *tokens* de uma vez, o que não ocorre com o léxico na forma de sub-rotina, que envia cada *token* diretamente para a próxima etapa do compilador/interpretador.
3. As expressões regulares definidas para as entradas do compilador são as seguintes:
   a. letra = a + b + ... + z + A + B + ... + Z
   b. digito = 0 + 1 + 2 + 3 ... + 9
   c. Numeros:
      i. Inteiros = <digito><digito>*
      ii. Ponto Flutuante = <digito><digito>*.<digito>.<digito>*
   d. Identificadores = <letra><letra>*
   e. Palavras Reservadas:
      i. Tipos = (int)+(float)+(bool)
      ii. Constantes Booleanas = ((T+t)rue)+((F+f)alse)
      iii. Return = return
      iv. If = if
      v. Else = else
      vi. While = while
   f. Simbolos Especiais:
      i. (= (
      ii.) =)
      iii. ; = ;
      iv. { = {

v. } = }
vi. [= [
vii.] =]

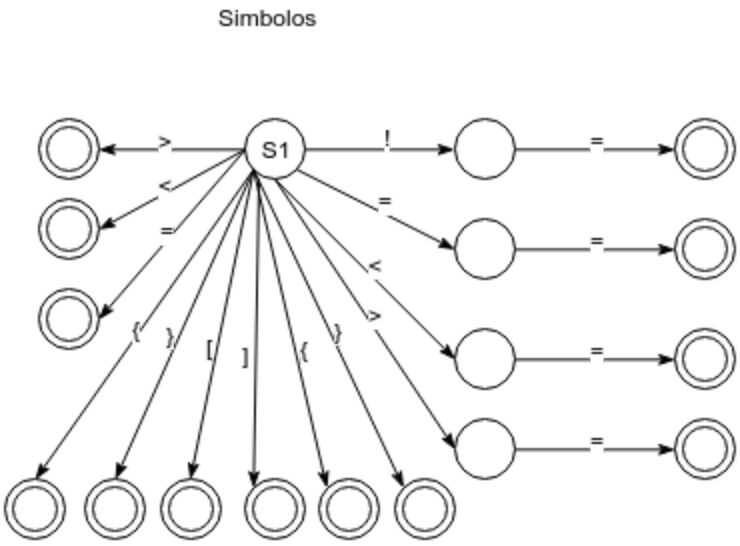
g. Operadores:
   i. ‘=’ = ‘=’
   ii. + = +
   iii. - = -
   iv. * = *
   v. / = /
   vi. ‘>’ = ‘>’
   vii. ‘<’ = ‘<’
   viii. ‘>=’ = ‘>=’
   ix. ‘<=’ = ‘<=’
   x. ‘==’ = ‘==’
   xi. ‘!=’ = ‘!=’
   xii. not = not
   xiii. and = and
   xiv. or = or

h. Comentários = #<qualquer_caracter_sem_ser_new_line>*

4. Os autômatos são bem simples, mas muito numerosos. Uma parte dos autômatos serão colocados (os dos símbolos), mas os outros são igualmente fáceis, mas com um número maior de estados intermediários entre o inicial (S1) e o final.



5. Mesma observação feita no exercício 4 é válida para esse exercício.



6. É possível diminuir o número de estados agrupando palavras reservadas e operadores lógicos “and”, “or” e “not” com identificadores e agrupando os outros operadores e símbolos especiais (menos o “# de comentário).

Transições com “<texto>” são saídas do transdutor, e a transição não consome nenhum símbolo de entrada.

7. O autômato obtido anteriormente foi codificado na linguagem C e foi implementado através de uma tabela (matriz) de transição, que “recebe” um estado e o tipo (letra, dígito, ‘>’, ‘<’, ‘\n’, entre outros) do caractere de entrada (codificados como números) e devolve o próximo estado do autômato.
   A sub-rotina lê caracteres do arquivo de entrada (entrada.txt), e guardando em um buffer, até acontecer uma transição no autômato/transdutor que gere alguma saída (tipo do token) e retorna o valor do buffer e o tipo em uma struct TOKEN (que possui uma string com os caracteres e um tipo).

8. O programa principal abre o arquivo de entrada e cria o autômato no estado inicial. Em seguida ele chama a sub-rotina que pega tokens, passando o arquivo e o autômato criado, repetidas vezes até receber um token do tipo “TT_EOF”, que indica que o fim do arquivo foi encontrado e nenhum existe nenhum token a ser gerado. O programa imprime os tokens (valor e tipo) em um arquivo de saida (“saida.txt”).
9. Funcionamento do analisador léxico:
   a. Descrição teórica: Dadas as substituições descritas no item 3, o analisador léxico deve ler os dados de entrada e, assim sendo, gerar os *tokens* correspondentes para o analisador sintático. Dado que este ainda não existe, a obtenção dos *tokens* é representada pelo arquivo de saída gerado pelo código do analisador léxico.

b. Descrição da estrutura: O analisador léxico define 11 tipos distintos de entrada e 8 estados possíveis de transição do autômato que o define, estados e tipos definidos por diferentes *enums* da linguagem C. Além disto, é definido um tipo de estrutura específico para os tipos possíveis de *tokens* a serem gerados (6, no total), a estrutura do *token*, composta do tipo de *token* e por um identificador (denominado "string"), além da tabela de transição de estados e da estrutura correspondente ao autômato, que guarda o estado atual do próprio.
c. Descrição do funcionamento: Este analisador léxico funciona baseado na identificação do tipo de entrada, obtido pela função *get_input_type*; além da obtenção do próximo estado a partir do estado atual e da entrada obtida até então usando-se a função *get_next_state*; obtém-se o tipo de *token* a ser retornado usando-se a função *get_token_type*, dado o estado anterior; além destas funções, existe a função *get_token_type_name*, responsável pela impressão do tipo de *token* a ser obtido, bem como se obtém o próximo *token* usando-se a função *get_next_token*, que recebe como entradas o arquivo de entrada e o autômato.

   A execução do analisador léxico se dá a partir do ponto de entrada do código (a função *main*), onde se obtém primeiramente o arquivo e, em seguida, instancia-se o autômato com o estado inicial. Em seguida, enquanto o final de arquivo não é encontrado, lê-se o *token* correspondente ao estado atual e o imprime na tela do console, juntamente com o seu tipo (via *get_token_type_name*), sendo que a função de obter o *token* apenas obtém o *token* dado o estado atual (via tabela) e, se o estado atual não for equivalente ao inicial e não for encontrado um "final de arquivo", as transições de estado ocorrem normalmente e, antes disto, o *token* correspondente àquele estado é retornado, usando-se a função *get_token_type* para se obter o tipo de *token* a ser retornado.
d. Testes realizados
   i. Testes Simples:
      1.
         a. Entrada: 1 + 1
         b. Saída: 1 (Inteiro)
            + (Simbolo)
            1 (Inteiro)
            (EOF)
      2.
         a. Entrada: int chamadaFuncao(var);
         b. Saída: int (Identificador)
            chamadaFuncao (Identificador)
            ((Simbolo)

var (Identificador)
) (Simbolo)
; (Simbolo)
(EOF)

3.

a. Entrada: while(True==false){
   # faz nada
   }
b. Saída: while (Identificador)
   ((Simbolo)
   True (Identificador)
   == (Simbolo)
   false (Identificador)
) (Simbolo)
   { (Simbolo)
   } (Simbolo)
   (EOF)

10. Para o analisador léxico em questão, a expansão de macros poderia ser de forma que o analisador, ao se deparar com uma entrada indicando um macro (como no caso da linguagem C a expressão “#DEFINE”) identificaria se a entrada seguinte ao identificador de macros é um macro existente. Para tanto, os identificadores de macros deveriam ser armazenados em uma tabela a qual o autômato responsável pela análise léxica consiga acessar e identificar a semelhança entre os macros armazenados e o macro que está sendo lido. Em caso positivo, o macro lido então é expandido, o que pode ser feito em tempo de execução, expandindo-se o código identificado pelo macro ou usando-se código pré-compilado para ser carregado no lugar do macro em questão.