Linguagens Formais e Autômatos (LFA)

Aula de 21/10/2013

Linguagens Livres de Contexto: Notações BNF e EBNF

Sobre os primeiros slides da aula

Os primeiros slides da aula são parte do jogo de slides do autor do livro texto, disponível na url: http://www.univasf.edu.br/~marcus.ramos/livro-lfa/cap4.pdf

Foram utilizados os slides com os seguintes números:

- 7 e 21
- 24 a 26
- 27 a 39

Exercício da aula passada

Considere que os termos "a", "b", "c", "d", "e" são variáveis booleanas de uma linguagem de programação.

Considere também que, com os conectivos "AND", "OR" e "NOT" podem-se montar expressões booleanas, as quais podem conter sub-expressões delimitadas por parênteses balanceados. Exemplo: "(a AND ((b OR c OR d) OR (d AND e)))"

Escreva uma Gramática Livre de Contexto (GLC) para caracterizar tais expressões booleanas (ie. que podem conter sub-expressões delimitadas por parênteses balanceados).

Uma Gramática de Solução para o Exercício

Em linguagem Prolog

```
/**
S --> T C T | ( T C T ) | N ( T )
T --> a | b | c | d | e | ( S )
C --> and | or
N --> not
**/

n --> ['not'].
c --> ['and']; ['or'].
t --> ['a']; ['b']; ['c']; ['d']; ['e']; ['('],s,[')'].
s --> t,c,t; ['('],t,c,t,[')']; n,['('],t,[')'].
```

Exemplo de Conversão para BNF

```
S --> T C T | ( T C T ) | N ( T )
T --> a | b | c | d | e | ( S )
C --> and | or
N --> not
```

```
<S>::= <T><C><T>|(<T><C><T>)|<N>(<T>)
<T>::= a|b|c|d|e|(<S>)
<C>::= and|or
<N>::= not
```

Na notação BNF, os não-terminais são representados por textos delimitados pelos metassímbolos “<” e “>”; para distingui-los dos símbolos terminais, o metassímbolo “→” é substituído por “::=” e, finalmente, todas as alternativas de substituição para um mesmo não-terminal são agrupadas, separando-se umas das outras com o metassímbolo “|”. Os terminais são denotados sem delimitadores.

Marcus Ramos (UNIVASF) LFA 2010-1 4 de junho de 2012 28 / 295

Exemplo de Especificação em BNF já adaptada

Fonte: http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/r0003509.htm

The BNF specification for DATALINKs:

URL

```
url          =    httpurl | fileurl | uncurl | emptyurl
```

HTTP

```
httpurl      =    "http://" hostport [ "/" hpath ]
hpath        =    hsegment *[ "/" hsegment ]
hsegment     =    *[ uchar | ";" | ":" | "@" | "&" | "=" ]
```

Note that the search element from the original BNF in RFC1738 has been removed, because it is not an essential part of the file reference and does not make sense in DATALINKs context.

FILE

```
fileurl      =    "file://" host "/" fpath
fpath        =    fsegment *[ "/" fsegment ]
fsegment     =    *[ uchar | "?" | ":" | "@" | "&" | "=" ]
```

Note that `host` is not optional and the "`localhost`" string does not have any special meaning, in contrast with RFC1738. This avoids confusing interpretations of "`localhost`" in client/server and partitioned database configurations.

UNC

```
uncurl       =    "unc:\\" hostname "\" sharename "\" uncpath
sharename    =    *uchar
uncpath      =    fsegment *[ "\" fsegment ]
```

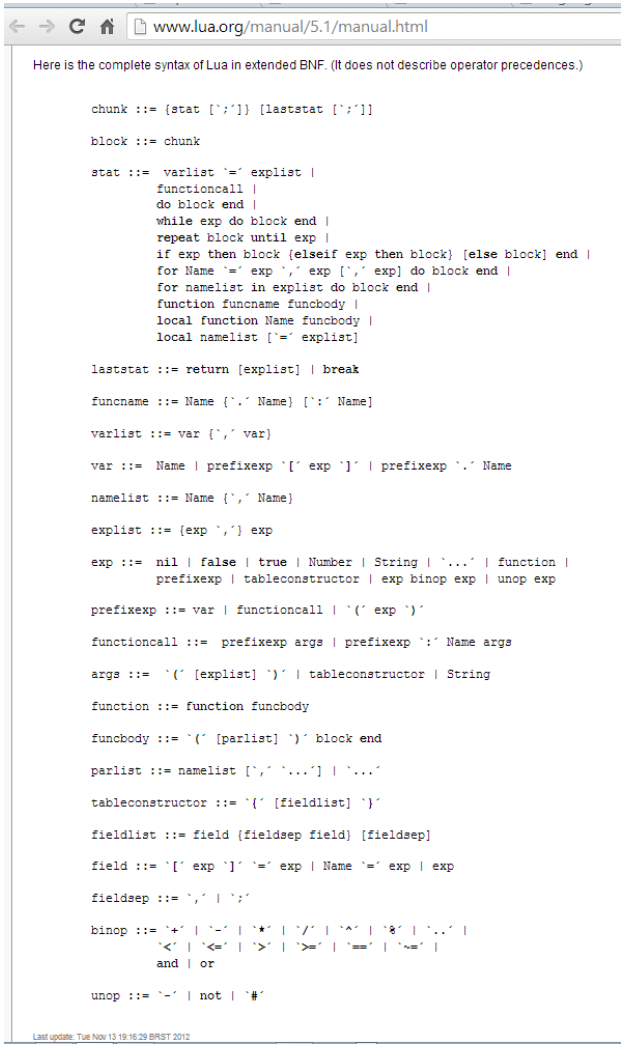
```
emptyurl     =    ""
hostport     =    host [ ":" port ]
host         =    hostname | hostnumber
hostname     =    *[ domainlabel "." ] toplabel
domainlabel  =    alphadigit | alphadigit *[ alphadigit | "-" ] alphadigit
toplabel     =    alpha | alpha *[ alphadigit | "-" ] alphadigit
alphadigit   =    alpha | digit
hostnumber   =    digits "." digits "." digits "." digits
port         =    digits
```

Empty (zero-length) URLs are also supported for DATALINK values. These are useful to update DATALINK columns when reconcile exceptions are reported and non-nullable DATALINK columns are involved. A zero-length URL is used to update the column and cause a file to be unlinked.

Miscellaneous Definitions

```
lowalpha     =    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
                  "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
                  "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
                  "y" | "z"
hialpha      =    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
                  "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                  "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
                  "Y" | "Z"
alpha        =    lowalpha | hialpha
digit        =    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                  "8" | "9"
safe         =    "$" | "-" | "_" | "." | "+"
extra        =    "!" | "*" | "'" | "(" | ")" | ","
hex          =    digit | "A" | "B" | "C" | "D" | "E" | "F" |
                  "a" | "b" | "c" | "d" | "e" | "f"
escape       =    "%" hex hex
unreserved   =    alpha | digit | safe | extra
uchar        =    unreserved | escape
digits       =    1*digit
```

Exemplo de Especificação em EBNF padrão



www.lua.org/manual/5.1/manual.html

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=  varlist `=´ explist |
         functioncall |
         do block end |
         while exp do block end |
         repeat block until exp |
         if exp then block {elseif exp then block} [else block] end |
         for Name `=´ exp `,´ exp [`,´ exp] do block end |
         for namelist in explist do block end |
         function funcname funcbody |
         local function Name funcbody |
         local namelist [`=´ explist]

laststat ::= return [explist] | break

funcname ::= Name {`.´ Name} [`:´ Name]

varlist ::= var {`,´ var}

var ::=  Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

explist ::= {exp `,´} exp

exp ::=  nil | false | true | Number | String | `...´ | function |
         prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `(´ exp `)´

functioncall ::=  prefixexp args | prefixexp `:´ Name args

args ::=  `(´ [explist] `)´ | tableconstructor | String

function ::= function funcbody

funcbody ::= `(´ [parlist] `)´ block end

parlist ::= namelist [`,´ `...´] | `...´

tableconstructor ::= `{´ [fieldlist] `}´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp

fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
         `<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
         and | or

unop ::= `-´ | not | `#´
```

Last update: Tue Nov 13 19:16:29 BRST 2012

8 – The Complete Syntax of Lua

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=  varlist `=´ explist |
         functioncall |
         do block end |
         while exp do block end |
         repeat block until exp |
         if exp then block {elseif exp then block} [else block] end |
         for Name `=´ exp `,´ exp [`,´ exp] do block end |
         for namelist in explist do block end |
         function funcname funcbody |
         local function Name funcbody |
         local namelist [`=´ explist]

laststat ::= return [explist] | break

funcname ::= Name {`.´ Name} [`:´ Name]

varlist ::= var {`,´ var}

var ::=  Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

explist ::= {exp `,´} exp

exp ::=  nil | false | true | Number | String | `...´ | function |
         prefixexp | tableconstructor | exp binop exp | unop exp
```

Continua

Exemplo de Especificação em EBNF padrão

www.lua.org/manual/5.1/manual.html

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=  varlist `=´ explist |
         functioncall |
         do block end |
         while exp do block end |
         repeat block until exp |
         if exp then block {elseif exp then block} [else block] end |
         for Name `=´ exp `,´ exp [`,´ exp] do block end |
         for namelist in explist do block end |
         function funcname funcbody |
         local function Name funcbody |
         local namelist [`=´ explist]

laststat ::= return [explist] | break

funcname ::= Name {`.´ Name} [`:´ Name]

varlist ::= var {`,´ var}

var ::=  Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

explist ::= {exp `,´} exp

exp ::=  nil | false | true | Number | String | `...´ | function |
         prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `(´ exp `)´

functioncall ::=  prefixexp args | prefixexp `:´ Name args

args ::=  `(´ [explist] `)´ | tableconstructor | String

function ::= function funcbody

funcbody ::= `(´ [parlist] `)´ block end

parlist ::= namelist [`,´ `...´] | `...´

tableconstructor ::= `{´ [fieldlist] `}´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp

fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
         `<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
         and | or

unop ::= `-´ | not | `#´
```

Last update: Tue Nov 13 19:16:29 BRST 2012

8 – The Complete Syntax of Lua

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

Continuação

```
prefixexp ::= var | functioncall | `(´ exp `)´

functioncall ::=  prefixexp args | prefixexp `:´ Name args

args ::=  `(´ [explist] `)´ | tableconstructor | String

function ::= function funcbody

funcbody ::= `(´ [parlist] `)´ block end

parlist ::= namelist [`,´ `...´] | `...´

tableconstructor ::= `{´ [fieldlist] `}´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp

fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
         `<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
         and | or

unop ::= `-´ | not | `#´
```

Last update: Tue Nov 13 19:16:29 BRST 2012

Exercício de Autoaprendizado 1

Faça a Análise Sintática (i.e. produza uma árvore de derivação) para o seguinte "statement" em Lua, utilizando a parte relevante da especificação dos slides anteriores:

```
function addfile (filename)
     local sum = 0
          for line in io.lines(filename) do
          sum = sum + tonumber(line)
          end
     return sum
end
```

Exercício de Autoaprendizado 2

Escolha e linguagens de programação que tenham "chamada de função" (sugestão: C, Pascal e Perl) e:

1. Escreva em BNF ou EBNF a sintaxe do comando nas 3 linguagens para o equivalente à seguinte função (em pseudocódigo):

```
funcao diferenca (x,y: inteiro): inteiro
inicio
retorne x-y
fim
```

2. Comente as diferenças sintáticas entre as linguagens que você escolheu.
3. Tente repetir o exercício 1, acima, com comandos equivalentes à função `diferença` em **linguagens de paradigma diferente das que você escolheu** (sugestão: Lisp e Prolog). Aprecie as diferenças sintáticas encontradas agora.