COS 360 Grammar Writing Techniques

Team Assignment

Due: 11/28/20 @ 8:00 A.M.

It's probably much easier to DRAW the parse trees freehand and take a picture of them to send me. You can embed them in an MS Word doc (and probably any comparable Open Source document prep tool's document) I prefer that you NOT submit a pdf file because I cannot edit it.

Here we will use the BNF notation because it is easier on a standard

keyboard.

1. grammar variables are enclosed in angle brackets

2. the "can be replaced by" symbol is "::=" not the shorter "->"

3. alternative rhs's are separated by vertical bars(|)

YOU MAY NOT USE EXTENDED BNF, BECAUSE IT IS NOT CLEAR HOW YOU CAN SHOW A DERIVATION OR PARSE TREE FOR SUCH A GRAMMAR.

Although not a BNF convention, I will use upper case identfiers

for some terminals, and where I can w/o ambiguity, single symbols

for terminals like the comma(,), semicolon(;) and others. Blanks

in the right hand sides are not signficant but are there to

improve readability. To avoid ambiguity with two double quotes,

uses LAMBDA to stand for the empty string.

1. Define grammar rules for the interior of a string literal enclosed

in double quotes, <string interior>

<string literal ::= " <string interior> "

<string interior> should derive a possibly empty sequence of tokens

CH or ECH for char instances and escaped char instances with no

separating token.

You may use additional grammar variables, but use <A>, <B>, etc.

Answer:

<string literal ::= " <string interior> "

<string interior> ::= “<string interior><ch>|<string interior> <Ech> | <ch> | <Ech>”

<Ech> ::= “<A>|<B>”

2. Define grammar rules for the interior of a string array literal

in the rule

<string array literal> ::= { <string array literal interior> }

where <string array literal interior> should derive a possible

empty sequence of <string literal> where adjacent <string literal>

instances are separated by the comma(,).

You may use additional grammar variables, but use <A>, <B>, etc.

Answer:

<string array literal> ::= { <string array literal interior> }

<string array literal interior>::= { <string array literal interior>,<string literal> | <string literal>}

<string literal ::= " <string interior> "

<string interior> ::= “<string interior><ch>|<string interior> <Ech> | <ch> | <Ech>”

<Ech> ::= “<A>|<B>”

3. Modify the following grammar to elminate all common prefixes.

<A> ::= <B> <C> ; | <B> <C> , | <B> .

<B> ::= NEG <B> | NEG <C> | NEG

You will need additional grammar variables. Draw them from

<D>, <E>, <F>, etc. as needed.

4. Eliminate the left recursion from the following grammar

for arithmetic expressions involving \*, /, + and -. <A>

<A> ::= <A> + <B> | <A> - <B> | <B>

<B> ::= <B> / <C> | <B> \* <C> | <C>

<C> ::= VALUE | ( <A> )

You will need additional grammar variables. Draw them from

<D>, <E>, <F>, etc. as needed.

5. Define a grammar for type expressions to enforce the following

precedence and associativities. From high to low, the operators

are

LIST prefix, unary; for a type T, LIST T is the space of all lists

of values in type T. It must be repeatable as in

LIST LIST int would be lists of lists of ints.

PRODUCT infix, binary, left associative for the Cartesian product

of two types

DUNION infix, binary, left associative for the disjoint union of

two types

ARROW infix, binary, right associative for the function space

constructor; for types A and B, A ARROW B is the type of

partial functions from values in type A to values in type

B

Those are the terminals, along with the two parentheses, and PRIM

for all primitive types, and UNIT for the type w a single value.

PRIM and UNIT would be leaves in any type expression. That gives

eight terminals all told.

You should use grammar variables <A>, <B>, <C>, etc. as needed.

Your grammar should be unambiguous and enforce the given associativities

and precedences.

6. Do a leftmost derivation and corresponding parse tree using your grammar from the previous exercise for the string

PRIM ARROW UNIT PRODUCT PRIM DUNION LIST UNIT ARROW PRIM

7. We have indicated that certain linear recursions can be eliminated

by Arden's lemma that says when we have

X = B + AX

then X = (A\*)B

and when

X = B + XA

then X = B(A\*)

For each of the following, use Arden's lemma to eliminate the recursion.

For each first first identify WHICH VERSION of the lemma you are using,

X = B + XA

or

X = B + AX

and show the match of X, B, and A for the specific instance before

applying the rule to get the answer.

For example, if I have

<a> ::= ID | C | ! <a> | - <a> | \* <b> <a>

then the complete answer is

X = B + AX

where

X is <a>

B is (ID + C)

A is (! + - + \* <b>)

so the solution, X = A\*B is

<a> = (! + - + \* <b>)\*(ID + C)

You can use the | instead of + if you wish, in the matches and final solution, but you must employ parens to force the correct precedence.

A. <a> ::= CD | <b> | <a>AB | <b> | <a> F

B. <b> ::= G | <b> | A B C D <b> | <a>

C. <c> ::= <a> <b> <c> | <a> D | LAMBDA | A <c>

8. Suppose the nullable, First, and Follow table for grammar variables <A>, <B>, <C>, <D>, and <E> is given as the following the following table. The terminals here are a, b, c, d, e, f, g, h, i, and j.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | nullable | First | Follow |
| <A> | true | a | b |
| <B> | true | c | d |
| <C> | false | e | f |
| <D> | true | g | h |
| <E> | true | i | j |

Calculate the lookahead sets for each of the following productions.

1. <A> ::= <B><D><C>a
2. <C> ::= <A><E><B>
3. <B> ::= <E>a
4. <D> ::= LAMBDA
5. <E> ::= <C>a