1. Consider the grammar:

S -> aS | aSbS | epsilon

where S is the only non-terminal, and epsilon is the null string.

a) Show that the grammar is ambiguous, by giving two parse trees for the

string aab.

b) Show (by induction, e.g.) that this grammar generates strings such that in

any prefix of the string there are at least as many a's as b's. (Optional:

show that all such strings are generated by this grammar).

c) Find an unambiguous grammar that generates these strings.

Solution.

---------

a) The ambiguity is easy to show: you can derive the string aab as follows:

(at every step we expand the leftmost non-terminal);

S -> aSbS -> aaSbS -> aabS -> aab

S -> aS -> aaSbS -> aabS -> aab

These two parses correspond to associating the b with the first or the

second a. This is somewhat analogous to the problem of the dangling else,

where the last else may be associated with the previous then or with an

earlier one.

b) The proof that all prefixes have at least as many a's as b's follows from

the given productions. It is clearly true if the length of the string is

1 (it can only be a single a) or 2 (aa or ab). Assume that it is true for

all strings of length up to n. Then, using the given productions, we can

construct strings of length n+1. and longer, for which the property clearly

holds because we have added an a to any previous prefix. Therefore the

property holds for all n.

c) We can disambiguate by using a similar approach to the dangling else, and

decide that each b should be associated with the nearest a. This means that

the expansion within an ab pair should always be balanced. This leads to

the following grammar:

S -> a S | S1 S | epsilon

S1 -> a S1 S1 b | epsilon

It is easy to verify that this generates the same strings as the original

grammar, and the parse tree is always unique, because one b is always associated

with the most recent a.

Note that the answer is not necessarily unique. If the grammar is ambiguous,

it means that we get to choose between possible parses, and each choice is

in a sense a different language. For example, given the ambiguous grammar

for expressions:

E -> E + E | E \* E | id

We say that the unambiguous grammar we want is:

E -> E + T | T, T -> T \* T | id

because it gives us the proper precedence between the two operators. But that

choice is in no way mandated by the grammar. We could just as well choose:

E -> E \* T | T , T -> T + T | id

which generates the same strings, but gives the opposite precedence to

operators.

2. Consider a simple language with the following non-terminals:

stat-seq : a sequence of statements

if-stat : if-statement, both branches have a sequence of statements

while-stat : a loop construct, whose body has a sequence of statements

func-def : a function definition, whose body is a sequence of statements.

ret-stat : a return statement

any-stat : any of the above, assignment, goto, etc.,

Most languages have the semantic requirement that a function body have at least

one return statement somewhere, possibly nested in a loop or a conditional

statement.

For the non-terminals above, write the productions that enforce this rule,

namely that a function definition contain at least one return statement.

You will want to introduce additional non-terminals.

Comment: this kind of semantic restriction can be expressed grammatically, but

as you will see, it makes the grammar larger and harder to understand, In

practice, this kind of restriction is better stated in prose, and such a rule

is not enforced by the parser of a compiler, but by a separate semantic check.

Solution.

---------

We start with the following grammar:

stat-seq -> {statement}\*

if-stat -> IF expr THEN stat-seq ELSE stat-seq END

while-stat -> WHILE expr DO stat-seq END

ret-stat -> RETURN expr

any-stat -> any statement, including assignments, etc.

We now introduce a new non-terminal, which is a sequence

of statements that must have at least one return statement:

ret-stat-seq -> stat-with-return stat-seq |

any-stat ret-stat-seq

Now a statement with a return is one of the following

stat-with-return -> ret-stat | ret-if-stat | ret-while-stat

A ret-if-stat is an if statement that has at least one return

ret-if-stat -> IF expr THEN ret-stat-seq ELSE stat-seq END |

IF expr THEN stat-seq ELSE ret-stat-seq END

And similarly, a ret-while-stat is a while with at least one return

ret-while-stat -> WHILE expr DO ret-stat-seq END

Now a function is defined as having a ret-stat-seq instead

of a stat-seq, and we know there is at least one return somewhere in

its body.

1. Consider the following grammar:

R => R '|' R

R => R R

R => R \*

R => ( R )

R => a | b

where the terminals are { |, \*, (, ), a, b }. This grammar generates regular

expressions over a,b involving alternation, concatenation, and repetition.

a) Compute FIRST and FOLLOW for the only non-terminal in this grammar.

Answer:

FIRST(R) = {a, b, (}

FOLLOW(R) = {|, \*, ), a, b, (, $}

b) Show the grammar is ambiguous.

Answer:

There are two left-most derivations for the string "ab | a":

R => RR => aR => aR | R => ab | R => a b | a, that is to say a (b | a)

R => R | R => RR | R => aR | R => ab | R => ab | a. that is to say (ab) | a

(similar to the arithmetic ambiguity in a + b \* c when we give all arithmetic

operators the same precedence).

c) construct an equivalent unambiguous grammar that gives the operators

star (repetition), concatenation, and alternation the usual precedences :

repetition > concatenation > alternation.

Answer:

A new grammar is (e is the empty symbol):

R => B R'

R' => '|' R | e

B => A B'

B' => B | e

A => F A''

A' => \* A''

A'' => A' | e

F => a | b | ( R )

This corresponds to the standard strategy of creating different non-terminal

symbols for each level of binding we want to impose.

d) Build the FIRST and FOLLOW set for this new grammar. Is it LL (1)?

SYMBOL FIRST FOLLOW

F a, b, ( \*, a, b, (, |, ), $

A' \* a, b, (, |, ), $

A'' \*, e a, b, (, |, ), $

A a, b, ( a, b, (, |, ), $

B a, b, ( |, ), $

B' a, b, (, e |, ), $

R' |, e $, )

R a, b, ( $, )

It can be verified that the new grammar is LL(1) by constructing a

parsing table that has no conflicts (omitted here).