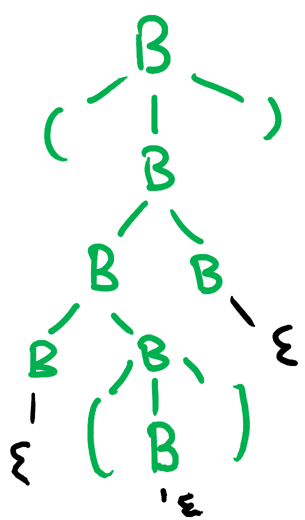
Exercise 1: Balanced Parentheses

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence) and find unambiguous grammar for the same language.

B ::= ε | (B) | B B

$\Sigma = \{ (,) \}$



same string,
2 different trees
→ ambiguous

$B \to (B) \to (BB)$

$\to (BBB)$

$\to (B(B)B) \to (B(B))$

$\to ((B)B)$

$\vdots$

$\to (())$

Remark

- The same parse tree can be derived using two different derivations, e.g.

  B -> (B) -> (BB) -> ((B)B) -> ((B)) -> (())

  B -> (B) -> (BB) -> ((B)B) -> (()B) -> (())

this correspond to different orders in which nodes in the tree are expanded

- Ambiguity refers to the fact that there are actually multiple *parse trees*, not just multiple derivations.

Towards Solution

- (Note that we must preserve precisely the set of strings that can be derived)
- This grammar:

B ::= ε | A
A ::= **( )** | A A | (A)

solves the problem with multiple ε symbols generating different trees, but it is still ambiguous: string **( ) ( ) ( )** has two different parse trees

Solution

- Proposed solution:

  B ::= ε | B **(**B**)**

- this is very smart! How to come up with it?
- Clearly, rule B::= B B generates any sequence of B's. We can also encode it like this:

  B ::= C*

  C ::= (B)

- Now we express sequence using recursive rule that does not create ambiguity:

  B ::= ε | C B

  C ::= (B)

- but now, look, we "inline" C back into the rules for so we get exactly the rule

  B ::= ε | B **(**B**)**

This grammar is not ambiguous and is the solution. We did not prove this fact (we only tried to find ambiguous trees but did not find any).

Exercise 2: Dangling Else

The dangling-else problem happens when the conditional statements are parsed using the following grammar.

S ::= S **;** S

S ::= id **:=** E

S ::= **if** E **then** S

S ::= **if** E **then** S else S

Find an unambiguous grammar that accepts the same conditional statements and matches the else statement with the nearest unmatched if.

Discussion of Dangling Else

```
if (x > 0) then
   if (y > 0) then
     z  = x + y
else x = - x
```

- This is a real problem languages like C, Java
  - resolved by saying **else** binds to innermost **if**
- Can we design grammar that allows all programs as before, but only allows parse trees where else binds to innermost if?

Sources of Ambiguity in this Example

- Ambiguity arises in this grammar here due to:
  - dangling **else**
  - binary rule for sequence (**;**) as for parentheses
  - priority between if-then-else and semicolon (**;**)

```
if (x > 0)
   if (y > 0)
     z  = x + y;
     u = z + 1       // last assignment is not inside if
```

Wrong parse tree -> wrong generated code

How we Solved It

We identified a wrong tree and tried to refine the grammar to prevent it, by making a copy of the rules. Also, we changed some rules to disallow sequences inside if-then-else and make sequence rule non-ambiguous. The end result is something like this:

S::= ε |A S // a way to write S::=A*

A ::= id **:=** E

A ::= **if** E **then** A

A ::= **if** E **then** A' **else** A

A' ::= id **:=** E

A' ::= **if** E **then** A' **else** A'

At some point we had a useless rule, so we deleted it.

We also looked at what a practical grammar would have to allow sequences inside if-then-else. It would add a case for blocks, like this:

A ::= **{** S **}**

A' ::= **{** S **}**

We could factor out some common definitions (e.g. define A in terms of A'), but that is not important for this problem.

Transforming Grammars into Chomsky Normal Form

Steps:

1. remove unproductive symbols
2. remove unreachable symbols
3. remove epsilons (no non-start nullable symbols)
4. remove single non-terminal productions X::=Y
5. transform productions w/ more than 3 on RHS
6. make terminals occur alone on right-hand side

1) Unproductive non-terminals

How to compute them?

What is funny about this grammar:

stmt ::= identifier := identifier
| while (expr) stmt
| if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= (expr)

There is no derivation of a sequence of tokens from expr

Why? In every step will have at least one expr, term, or factor

If it cannot derive sequence of tokens we call it *unproductive*

1) Unproductive non-terminals

- Productive symbols are obtained using these two rules (what remains is unproductive)
  - Terminals (tokens) are productive
  - If $X ::= s_1 \ s_2 \ \ldots \ s_n$ is rule and each s_i is productive then X is productive

```
stmt ::=  identifier := identifier
        | while (expr) stmt
        | if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= ( expr )
program ::= stmt | stmt program
```

Delete unproductive symbols.

Will the meaning of top-level symbol (program) change?

2) Unreachable non-terminals

What is funny about this grammar with starting terminal ‘program’

program ::= stmt | stmt program
stmt ::= assignment | whileStmt
assignment ::= expr = expr
ifStmt ::= if (expr) stmt else stmt
whileStmt ::= while (expr) stmt
expr ::= identifier

No way to reach symbol ‘ifStmt’ from ‘program’

2) Computing unreachable non-terminals

What is funny about this grammar with starting terminal ‘program’

program ::= stmt | stmt program
stmt ::= assignment | whileStmt

assignment ::= expr = expr

ifStmt ::= if (expr) stmt else stmt
whileStmt ::= while (expr) stmt
expr ::= identifier

What is the general algorithm?

2) Unreachable non-terminals

- Reachable terminals are obtained using the following rules (the rest are unreachable)
  - starting non-terminal is reachable (program)
  - If $X ::= s_1\ s_2\ \ldots\ s_n$ is rule and X is reachable then each non-terminal among $s_1\ s_2\ \ldots\ s_n$ is reachable

Delete unreachable symbols.

Will the meaning of top-level symbol (program) change?

3) Removing Empty Strings

Ensure only top-level symbol can be nullable

program ::= stmtSeq | "" |stmt ;

stmtSeq ::= stmt | stmt ; stmtSeq | "" | ; stmtSeq | ;

stmt ::= "" | assignment | whileStmt | blockStmt

blockStmt ::= { stmtSeq }

assignment ::= expr = expr

whileStmt ::= while (expr) stmt

expr ::= identifier

$S \to \varepsilon \mid S'$

How to do it in this example?

3) Removing Empty Strings - Result

program ::= ε | stmtSeq

stmtSeq ::= stmt| stmt ; stmtSeq |

| ; stmtSeq | stmt ; | ;

stmt ::= assignment | whileStmt | blockStmt

blockStmt ::= { stmtSeq } | { }

assignment ::= expr = expr

whileStmt ::= while (expr) stmt

whileStmt ::= while (expr)

expr ::= identifier

3) Removing Empty Strings - Algorithm

- Compute the set of nullable non-terminals
- Add extra rules
  - If $X ::= s_1\ s_2 \ldots s_n$ is rule then add new rules of form

    $X ::= r_1\, r_2 \ldots r_n$ 2^n

  where r_i is either s_i or, if s_i is nullable then

  r_i can also be the empty string (so it disappears)
- Remove all empty right-hand sides
- If starting symbol S was nullable, then introduce a new start symbol S' instead, and add rule $S' ::= S \mid \varepsilon$

3) Removing Empty Strings

- Since stmtSeq is nullable, the rule

  blockStmt ::= { stmtSeq }

  gives

  blockStmt ::= { stmtSeq } | { }

- Since stmtSeq and stmt are nullable, the rule

  stmtSeq ::= stmt | stmt ; stmtSeq

  gives

  stmtSeq ::= stmt | stmt ; stmtSeq
  | ; stmtSeq | stmt ; | ;