Tutorial 3: Syntax Analysis

Model Solutions

Spring 2023

Contents

Question 1

The grammar

$S \rightarrow S\ (\ S\ )\ \mid\ \epsilon$

is unambiguous.

To show this, we have to prove that for all strings *str* in $\mathcal{L}(S)$, *str* has a unique parse tree. Here, $\mathcal{L}(S)$ is the set of balanced parenthesis strings (BPS). Examples of BPS are: "`(()(()))`", "`()(())`" and ϵ, and an example of a parenthesis string that is not balanced is: "`(()(`"

Observe that BPS can have one of the following two structures:

(i) (BPS) - has an outer enclosing parenthesis pair.

(ii) $BPS_1\ BPS_2\ \ldots BPS_n$ i.e. n parallel BPS, each of type of type (i)

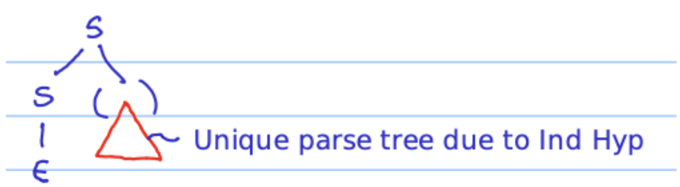
• Proof of unambiguity

The proof is by induction on the length of BPS.

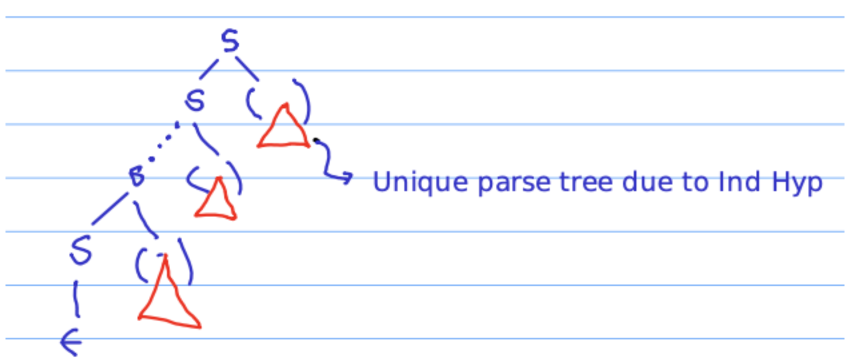
Base Case : The smallest BPS ϵ has a unique parse.

Induction hypothesis : All BPS of length n or less have unique parses. Now consider a BPS of length $n + 1$

(a) If BPS is of type (i) then the unique parse tree for it is



(b) If BPS is of type (ii) then the unique parse tree for it is



Question 2

The grammar is not ambiguous. The strings of *assignment_expression* are, in general, of the form

$$unary\_exp = unary\_exp = unary\_exp = \ldots = unary\_exp$$

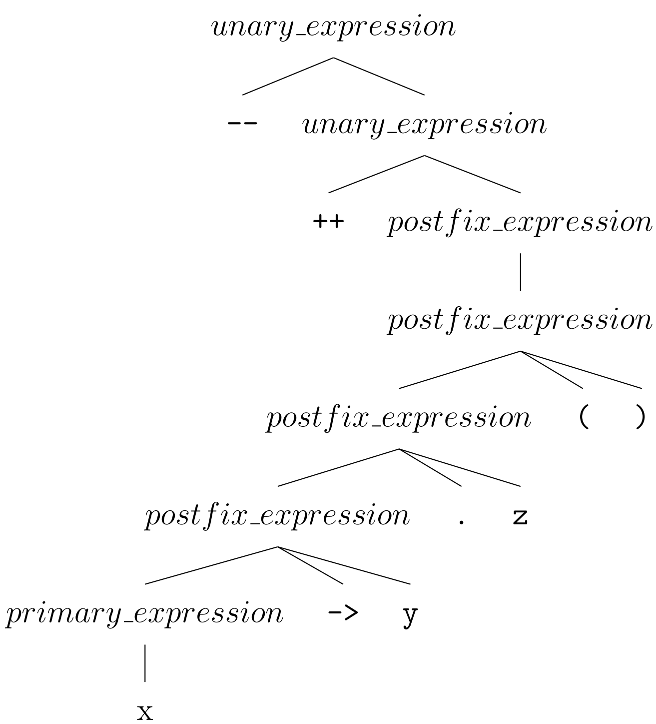
Also, according to the grammar, = is right-associative. So if *unary_exp* is unambiguous, so is *assignment_expression*.

Strings of *unary_exp* have the form: `PRE_OP`* *primary_exp* `POST_OP`*. This has a unique parse, because

1. *primary_exp* has a unique parse.
2. `POST_OP`s have a higher precedence than `PRE_OP`s.
3. `POST_OP`s are left associative and `PRE_OP`s are right associative.

As an example, consider the string. $--\ ++$ `x` $\rightarrow$ `y.z()`

The unique parse for this is



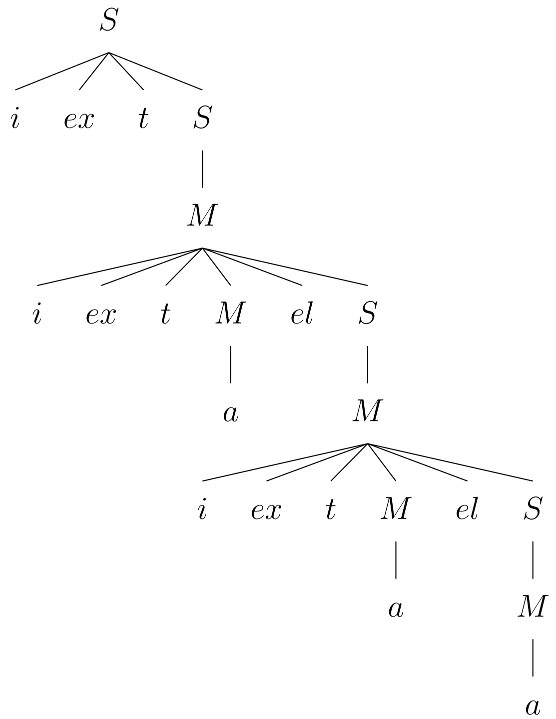
Question 3

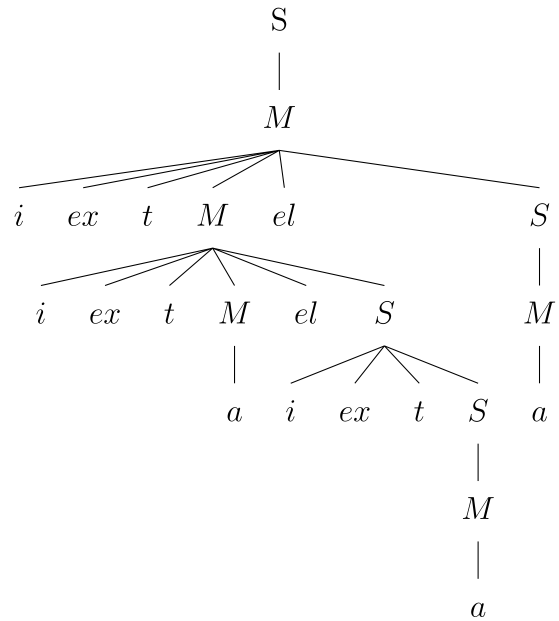
a) The grammar represents strings that give postfix expressions with $+$ and $*$ as operators and a representing digits.

b) This grammar is unambiguous. You can write a simple parser that keeps pushing the inputs until it finds a `+` or a `*`. At this point, it pops the top two elements from the stack, forms a parse tree with the operator symbol, and pushes the parse tree back into the stack.

Question 4

a) Consider the string: *i ex t i ex t a el i ex t a el a*
To save space we write i: `if`, ex: *expr*, t: `then`, S: *stmt*, M: *matched_statement*, el: `else`, a: `assignment`





b) The conflict shows up on the *else* symbol. The set of viable prefixes corresponding to the state where the conflict shows is $(i\ ex\ t)^{+}\ i\ ex\ t\ M\ el\ i\ ex\ t\ M$

Question 5

Discussed in class

Question 6

$$
\begin{array}{lcl}
S & \rightarrow & M\ \texttt{a} \\
S & \rightarrow & \texttt{b}\ M\ \texttt{c} \\
S & \rightarrow & \texttt{d}\ \texttt{c} \\
S & \rightarrow & \texttt{b}\ \texttt{d}\ \texttt{a} \\
M & \rightarrow & \texttt{d}
\end{array}
$$

a) There are 2 conflicts.

b) While one can construct the parsing table and answer these questions, it is fun to do it just through observation.

- Right at the beginning of the parse after shifting `d`, there is a conflict on the symbol `c`. The state of the conflicting entry is represented by the viable prefix `d`, and the conflicting items are $S \rightarrow \texttt{d} \bullet \texttt{c}$ and $M \rightarrow \texttt{d} \bullet$. The conflict is on symbol `c`

  This is a SLR parser problem: Note that if you reduced `d` to M (the stack now contains M only), `c` cannot follow the viable prefix M in any right-sentential form. However `c` can follow the viable prefix `b` M ($S \rightarrow \texttt{b}\ M\ \texttt{c}$).

- Another such conflict arising out of the weakness of the SLR parser is when the viable prefix is `b d`. The conflicting items are $M \rightarrow \texttt{d} \bullet$ and $S \rightarrow \texttt{b d} \bullet \texttt{a}$. The conflict is on symbol `a`.

Question 7

$$
\begin{aligned}
S &\rightarrow \texttt{ass} \\
S &\rightarrow \texttt{ifcondthen}\ S \\
S &\rightarrow \texttt{ifcondthen}\ S\ \texttt{else}\ S
\end{aligned}
$$

a) The conflict is in the state represented by $\texttt{ifcondthen}^{+}\ S$.

b) The self loop is on the state represented represented by the viable-prefix $\texttt{ifcondthen}^{+}$.

c) Suppose we resolve the conflict in the usual way (match a then with its closest `else`). Then we cannot reduce by $S \rightarrow \texttt{ifcondthen}\ S$ when the next symbol is `else`. Here is the list of productions and the terminal symbols on which they can reduce:

$S \rightarrow \texttt{ass}$	can reduce using this rule on \$, `else`
$S \rightarrow \texttt{ifcondthen}\ S$	can reduce only on \$
$S \rightarrow \texttt{ifcondthen}\ S\ \texttt{else}\ S$	can reduce on \$, `else`

Question 8

a) Actually there are two self loops. These are identified by viable prefixes:

   (a) `id[(`$^+$ that is `id[` followed by one or more `(`

   (b) `id[E]:=(`$^+$

b) Not covered yet

Question 9

Not covered yet