CS164 Section Notes - 9/17/2003

Topics to be covered

- Announcements.
- Review of recursive descent parsing.
- `LL(1)` parsing.
- Constructing the `LL(1)` parsing tables with `FIRST` and `FOLLOW` sets.

Review of recursive descent parsing

Highlights of recursive descent parsing:

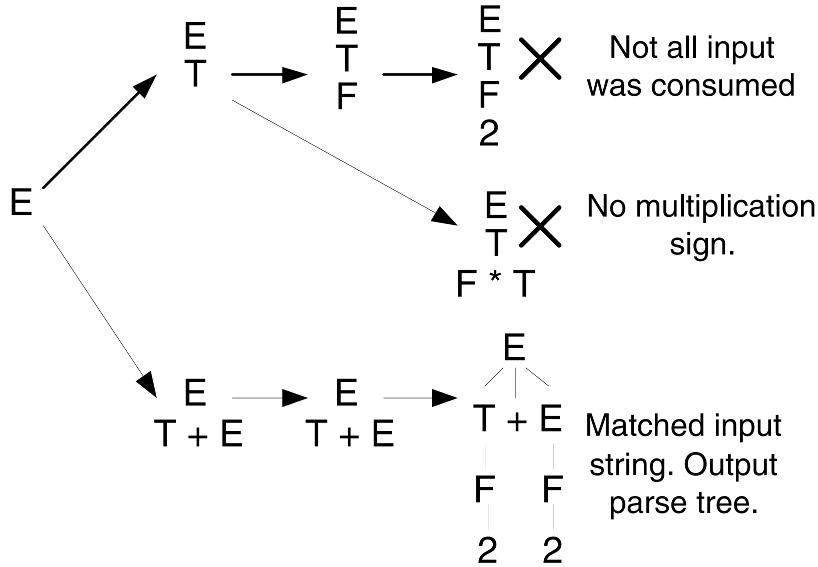
- Begin at the starting nonterminal.
- Try to match each production of a nonterminal with what appears in the input tokens.
- If you get stuck with one particular expansion, backtrack to the last place where you had a choice about which non-terminal you expanded. Keep backtracking if you get stuck.
- If you are still stuck, then the input doesn't match the grammar. This is a *syntax error*.
- Otherwise, you will have parsed the input tokens into parse tree.

Example done in section:

```
E  →  T | T + E
T  →  F | F * T
F  →  (E) | integer
```

Problems with the grammar: Describes arithmetic expressions, but the associativity is wrong! (`2+3+4` would be parsed as `2+(3+4)` rather than `(2+3)+4`. However, the grammar has no left-recursion, which makes it an attractive to illustrate recursive descent.

Recursive descent parsing of `2+3`, a lot of the recursive steps are missing, but two are shown as an example:



`LL(1)` parsing

`LL(1)` grammar:

```
E  →  T X
X  →  + E | ϵ
T  →  ( E ) | int Y
Y  →  * T | ϵ
```

- We say the previous grammar is `LL(1)` because at each point in the token stream, we can look at the next token and definitively decide which production we need to use.
- This is in contrast to the recursive descent parsing, where we may have needed to backtrack when we discovered that we used the wrong production rule.
- Because we know definitively which production rule to use given the nonterminal we are looking at and the next token, we can create a table for parsing, with non-terminals on one axis, and tokens that we are looking at on the other:

`LL(1)` **parsing table** for the grammar shown below:

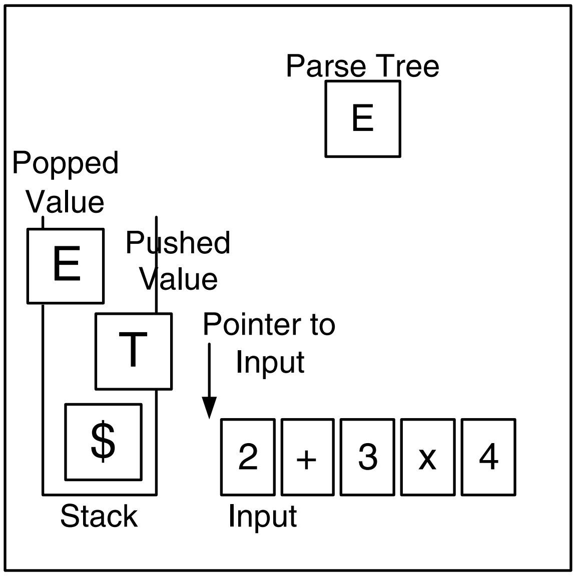
	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

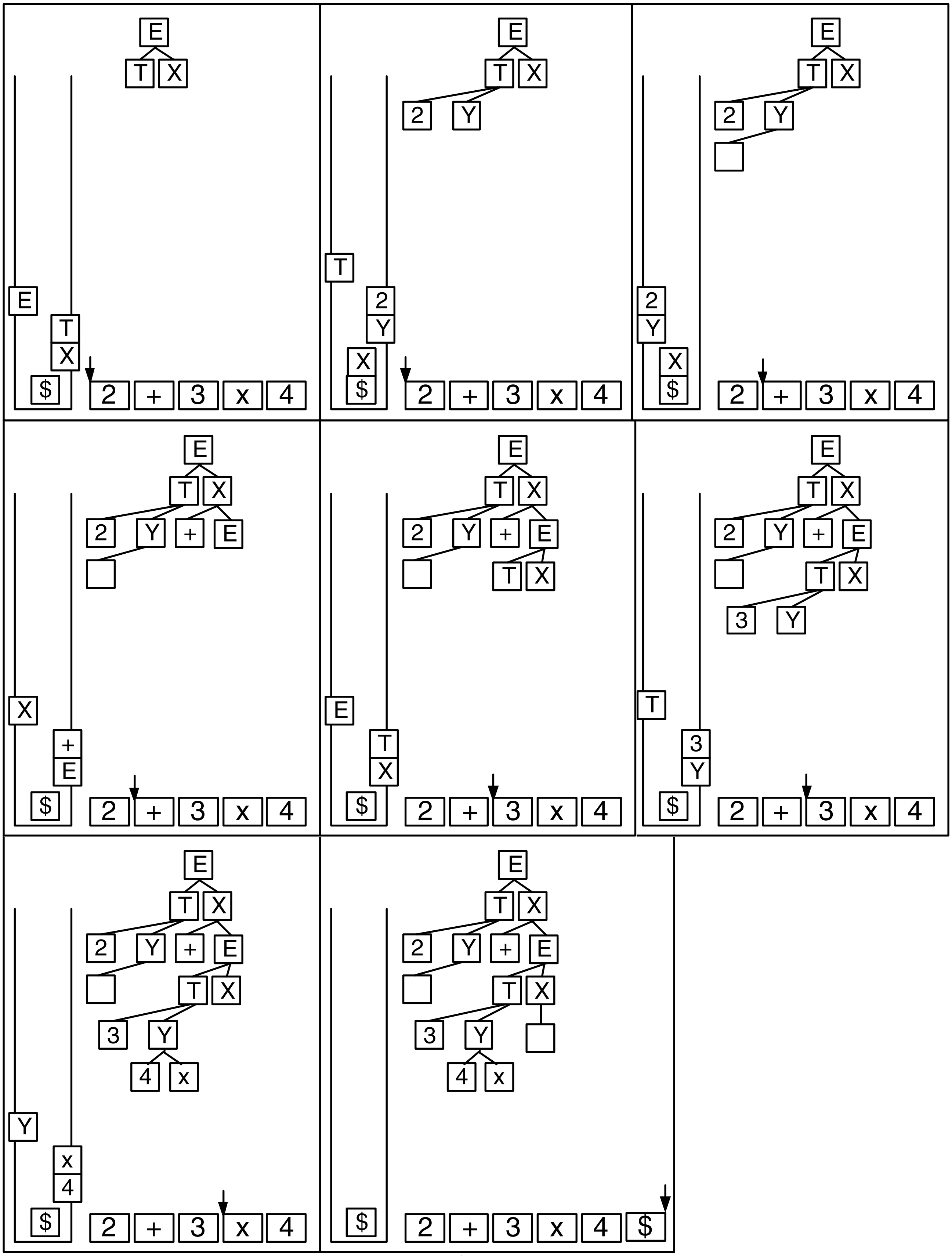
Parse State: The current state of the parser. In the case of an `LL(1)` parser, the parse state consists of:

- Contents of the current parse stack.

- Pointer into how much of the input has been consumed.
- Partial parse tree that's been generated.

Let's see how this works in a concrete example of parsing `2+3*4`:




2 + 3 x 4
2 + 3 x 4
2 + 3 x 4
2 + 3 x 4
2 + 3 x 4
2 + 3 x 4
2 + 3 x 4
2 + 3 x 4 $

Constructing LL(1) parsing tables with FIRST and FOLLOW sets

In the previous example, we filled up the parsing table by using the intuitive idea of which production rule to use when we encounter a particular token. Now we will make it more precise with idea of FIRST and FOLLOW sets:

Definition of FIRST: The FIRST set of a terminal just contains itself. The FIRST set of a non-terminal A is the set of all terminal tokens that could appear at the beginning of a string derived from A, including ϵ.

(See the formal definition of FIRST in the lecture notes for more details.)

For example, the `first` set of the non-terminal `T` is an `int` or a parenthesis, because from the `T` non-terminal, we can derive strings that start with `int` and strings that start with parenthesis.

We can also talk about the FIRST set of a string of grammar symbols, i.e. the FIRST set of `T X`. Note that if the non-terminal in front has ϵ in its FIRST set, then we have to add the FIRST set of the grammar symbols that follow it.

Definition of FOLLOW: FOLLOW(A) is the set of terminals that could follow non-terminal A in some derivation of the grammar.

What does this mean? Basically for any righthand side of the grammar that contains A, we look put any terminals that follow A into the FOLLOW(A) set.

Computing FOLLOW(A):

- Look for A on the righthand side of the grammar rules. So it would be some production rule $S \rightarrow \alpha A \beta$, where α and β are arbitrary strings of grammar symbols. (Remember, α and β can be ϵ as well).
- Add everything from the FIRST set of β to FOLLOW(A). This takes care of all characters that are produced by the grammar symbols immediately following A in the productions.
- If a nonterminal B ends a production, such as $S \rightarrow \alpha B$, then we must add the FOLLOW of S to the FOLLOW of B. This is because wherever an S, αB can be substituted for the symbol, therefore whatever follows S must also follow αB.
- Similarily, if there is a production $S \rightarrow \alpha B \beta$, and FIRST($\beta$) contains ϵ, then we must add the FOLLOW of S to the FOLLOW of B. (Because β can become ϵ, this similar to the case just described above.)
- FOLLOW sets of a grammar symbol are dependent on the FOLLOW set computed from other symbols. This means we have to keep running the algorithm until the FOLLOW sets stop changing.

So how does this all relate to the LL(1) table?

Consider a rule $A \rightarrow \alpha$:

- For each terminal t in FIRST(α), add $A \rightarrow \alpha$ to the cell indexed by A and t.
- If ϵ is in FIRST(α), then add $A \rightarrow \alpha$ for every terminal t in FOLLOW(A), add $A \rightarrow \alpha$ to the cell indexed by A and t.