Lecture 11: LL(1)

From Recursive Descent to Table Driven

- Our recursive descent parsers had a very regular structure.

Definition of nonterminal A:

```
A : α1
  | α2
  | ...
  | α3
```

Program for A:

```
def A():
    if next() in S1:
        translation of α1
    elif next() in S2:
        translation of α2
    ...
```

- Here,

$$S_i = \left\{ \begin{array}{ll} \text{FIRST}(\alpha_i), & \text{if } \epsilon \notin \text{FIRST}(\alpha_i) \\ \text{FIRST}(\alpha_i) \cup \text{FOLLOW}(A), & \text{otherwise.} \end{array} \right\}$$

- and the translation of α_i simply converts each nonterminal into a call and each terminal into a `scan`.
- If the S_i do not overlap, we say the grammar is LL(1): input can be processed from $\boxed{\text{L}}$eft to right, producing a $\boxed{\text{L}}$eftmost derivation, and checking $\boxed{1}$ symbol of input ahead to see which branch to take.

Table-Driven LL(1)

- Because of this regular structure, we can represent the program as a table, and can write a general LL(1) parser that interprets any such table.
- Consider a previous example:

Grammar

```
1. prog  : sexp '⊣'
2. sexp  : atom
3.       | '(' elist ')'
4.       | '\'' sexp
5. elist : ϵ
6.       | sexp elist
7. atom  : SYM
8.       | NUM
9.       | STRING
```

	Lookahead symbol						
Nonterminal	(	)	'	SYM	NUM	STRING	⊣
prog	(1)		(1)	(1)	(1)	(1)	
sexp	(3)		(4)	(2)	(2)	(2)	
elist	(6)	(5)	(6)	(6)	(6)	(6)	(5)
atom				(7)	(8)	(9)	

- The table shows nonterminal symbols in the left column and the other columns show which production to use for each possible lookahead symbol.
- Grammar is LL(1) when this table has at most one production per entry.

A General LL(1) Algorithm

Given a fixed table T and grammar G, the function LLparse(X), where parameter X is a grammar symbol, may be defined

```
def LLparse(X):
    if X is a terminal symbol:
        scan(X)
    else:
        prod = T[X][next()]
        Let p1p2···pn be the right-hand side of production prod
        for i in range(n):
            LLparse(pi)
```