# CS412/413

Introduction to
Compilers and Translators
Spring '00

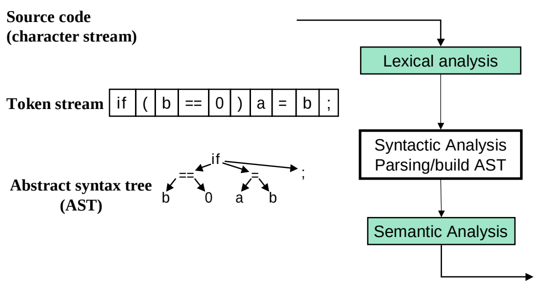
Lecture 4: Top-down parsing

# Outline

- Eliminating ambiguity in CFGs
- Top-down parsing
- LL(1) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing - parsing made simple



# Where we are






# Review of CFGs

- Context-free grammars can describe programming-language syntax
- Power of CFG needed to handle common PL constructs (e.g., parens)
- String is in language of a grammar if derivation from start symbol to string
- Ambiguous grammars a problem



# Limits of CFGs

- Syntactic analysis can't catch all "syntactic" errors
- Example: C++
  `HashTable<Key,Value> x;`
- Need to know whether `HashTable` is the name of a type to understand syntax! Problem: "`<`", "`,`" are overloaded
- Iota:
  `f(4)[1][2] = 0;`
- Difficult to write grammar for LHS of assign – may be easier to allow all exprs, check later



# if-then-else

- How to write a grammar for if stmts?

$S \rightarrow \text{if } (E)\ S$

$S \rightarrow \text{if } (E)\ S \text{ else } S$

$S \rightarrow other$

Is this grammar ok?

## No—Ambiguous!

- How to parse?

  if ($E$) if ($E$) $S$ else $S$

$S \to$ if ($E$) $S$
$S \to$ if ($E$) $S$ else $S$
$S \to$ *other*

$S \to$ if ($E$) $S$
$\to$ if ($E$) if ($E$) $S$ else $S$

$S \to$ if ($E$) $S$ else $S$
$\to$ if ($E$) if ($E$) $S$ else $S$

*Which "if" is the "else" attached to?*



## Grammar for Closest-if Rule

- Want to rule out if ($E$) if ($E$) $S$ else $S$
- Problem: unmatched if may not occur as the "then" (consequent) clause of a containing "if"

*statement* $\to$ *matched* | *unmatched*
*matched* $\to$ if ($E$) *matched* else *matched*
| *other*
*unmatched* $\to$ if ($E$) *statement*
| if ($E$) *matched* else *unmatched*



## Top-down Parsing

- Grammars for top-down parsing
- Implementing a top-down parser (recursive descent parser)
- Generating an abstract syntax tree



## Parsing a String Top-down

**Goal:** construct a leftmost derivation of string while reading in token stream

| Partly-derived String | Lookahead | parsed part unparsed part |
|---|---|---|
| $S$ | ( | (1+2+(3+4))+5 |
| $\to E+S$ | ( | (1+2+(3+4))+5 |
| $\to (S)+S$ | 1 | (1+2+(3+4))+5 |
| $\to (E+S)+S$ | 1 | (1+2+(3+4))+5 |
| $\to (1+S)+S$ | 2 | (1+2+(3+4))+5 |
| $\to (1+E+S)+S$ | 2 | (1+2+(3+4))+5 |
| $\to (1+2+S)+S$ | 2 | (1+2+(3+4))+5 |
| $\to (1+2+E)+S$ | ( | (1+2+(3+4))+5 |
| $\to (1+2+(S))+S$ | 3 | (1+2+(3+4))+5 |
| $\to (1+2+(E+S))+S$ | 3 | (1+2+(3+4))+5 |



## Problem

- Want to decide which production to apply based on next symbol

(1) $\quad S \to E \to (S) \to (E) \to (1)$

(1)+2 $\quad S \to E + S \to (S) + S \to (E) + S$
$\to (1)+E \to (1)+2$

- *Why is this hard?*



## Grammar is Problem

- This grammar cannot be parsed top-down with only a single look-ahead symbol
- Not **LL(1)**
- **L**eft-to-right-scanning, **L**eft-most derivation, **1** look-ahead symbol
- Is it LL(k) for some k?
- Can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language

## Making an LL(1) grammar

$S \to E + S$
$S \to E$
$E \to$ number
$E \to ( S )$

⬇

$S \to ES'$
$S' \to \varepsilon$
$S' \to + S$
$E \to$ number
$E \to ( S )$

- **Problem**: can't decide which $S$ production to apply until we see symbol after first expression
- **Left-factoring**: Factor common $S$ prefix, add new non-terminal $S'$ at decision point. $S'$ derives $(+E)^*$
- **Also:** convert left-recursion to right-recursion



## Parsing with new grammar

| | | |
|---|---|---|
| $S$ | ( | (1+2+(3+4))+5 |
| $\to E\ S'$ | ( | (1+2+(3+4))+5 |
| $\to (S)\ S'$ | 1 | (1+2+(3+4))+5 |
| $\to (E\ S')\ S'$ | 1 | (1+2+(3+4))+5 |
| $\to (1\ S')\ S'$ | + | (1+2+(3+4))+5 |
| $\to (1+E\ S')\ S'$ | 2 | (1+2+(3+4))+5 |
| $\to (1+2\ S')\ S'$ | + | (1+2+(3+4))+5 |
| $\to (1+2+S)\ S'$ | ( | (1+2+(3+4))+5 |
| $\to (1+2+E\ S')\ S'$ | ( | (1+2+(3+4))+5 |
| $\to (1+2+(S)\ S')\ S'$ | 3 | (1+2+(3+4))+5 |
| $\to (1+2+(E\ S')\ S')\ S'$ | 3 | (1+2+(3+4))+5 |
| $\to (1+2+(3\ S')\ S')\ S'$ | + | (1+2+(3+4))+5 |
| $\to (1+2+(3+E)\ S')\ S'$ | 4 | (1+2+(3+4))+5 |



## Predictive Parsing

- **LL(1)** grammar:
  - for a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - top-down parsing = predictive parsing
  - Driven by ***predictive parsing table*** of
    non-terminals × input symbols → productions



## Using Table

$S \to ES'$
$S' \to \varepsilon \mid + S$
$E \to$ number $\mid ( S )$

| | | |
|---|---|---|
| $S$ | ( | (1+2+(3+4))+5 |
| $\to E\ S'$ | ( | (1+2+(3+4))+5 |
| $\to (S)\ S'$ | 1 | (1+2+(3+4))+5 |
| $\to (E\ S')\ S'$ | 1 | (1+2+(3+4))+5 |
| $\to (1\ S')\ S'$ | + | (1+2+(3+4))+5 |
| $\to (1+S)\ S'$ | 2 | (1+2+(3+4))+5 |
| $\to (1+E\ S')\ S'$ | 2 | (1+2+(3+4))+5 |
| $\to (1+2\ S')\ S'$ | + | (1+2+(3+4))+5 |

| | number | + | ( | ) | \$ (EOF) |
|---|---|---|---|---|---|
| $S$ | $\to E\ S'$ | | $\to E\ S'$ | | |
| $S'$ | | $\to +S$ | | $\to \varepsilon$ | $\to \varepsilon$ |
| $E$ | $\to$ number | | $\to ( S )$ | | |



## How to Implement?

- Table can be converted easily into a ***recursive-descent parser***

| | number | + | ( | ) | \$ |
|---|---|---|---|---|---|
| $S$ | $\to E\ S'$ | | $\to E\ S'$ | | |
| $S'$ | | $\to +S$ | | $\to \varepsilon$ | $\to \varepsilon$ |
| $E$ | $\to$ number | | $\to ( S )$ | | |

- Three procedures: parse_S, parse_S', parse_E



## Recursive-Descent Parser

*lookahead token*

```
void parse_S () {
  switch (token) {
      case number: parse_E(); parse_S'(); return;
      case '(': parse_E(); parse_S'(); return;
      default: throw new ParseError();
  }
}
```

| | number | + | ( | ) | \$ |
|---|---|---|---|---|---|
| $S$ | $\to ES'$ | | $\to ES'$ | | |
| $S'$ | | $\to +S$ | | $\to \varepsilon$ | $\to \varepsilon$ |
| $E$ | $\to$ number | | $\to ( S )$ | | |

## Recursive-Descent Parser

```
void parse_S'() {
  switch (token) {
      case '+': token = input.read(); parse_S();
  return;
      case ')': return;
      case EOF: return;
      default: throw new ParseError();
  }
}
```

| | number | + | ( | ) | $ |
|---|---|---|---|---|---|
| $S$ | $\rightarrow ES'$ | | $\rightarrow ES'$ | | |
| $S'$ | | $\rightarrow +S$ | | $\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |
| $E$ | $\rightarrow$ number | | $\rightarrow ( S )$ | | |



## Recursive-Descent Parser

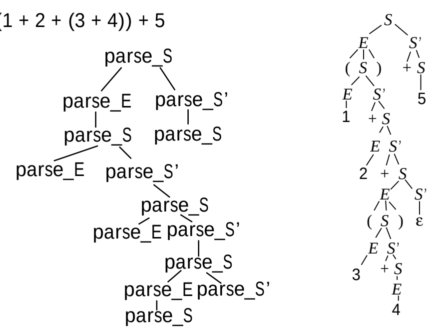
```
void parse_E() {
  switch (token) {
      case number: token = input.read(); return;
      case '(': token = input.read(); parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return;
      default: throw new ParseError(); }
}
```

| | number | + | ( | ) | $ |
|---|---|---|---|---|---|
| $S$ | $\rightarrow ES'$ | | $\rightarrow ES'$ | | |
| $S'$ | | $\rightarrow +S$ | | $\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |
| $E$ | $\rightarrow$ number | | $\rightarrow ( S )$ | | |



## Call Tree = Parse Tree

(1 + 2 + (3 + 4)) + 5






## How to Construct Parsing Tables

- Needed: algorithm for automatically generating a predictive parse table from a grammar

$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid + S$
$E \rightarrow$ number $\mid ( S )$

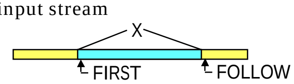
?

| | N | + | ( | ) | $ |
|---|---|---|---|---|---|
| $S$ | $ES'$ | | $ES'$ | | |
| $S'$ | | $+S$ | | $\varepsilon$ | $\varepsilon$ |
| $E$ | N | | $( S )$ | | |



## Constructing Parse Tables

- Can construct predictive parser if:
  - For every non-terminal, every look-ahead symbol can be handled by at most one production
- FIRST($\gamma$) for arbitrary string of terminals and non-terminals $\gamma$ is:
  - set of symbols that might begin the fully expanded version of $\gamma$
- FOLLOW(X) for a non-terminal X is:
  - set of symbols that might follow the derivation of X in the input stream






## Parse Table Entries

- Consider a production $X \rightarrow \gamma$
- Add $\rightarrow \gamma$ to the X row for each symbol in FIRST($\gamma$)

| | N | + | ( | ) | $ |
|---|---|---|---|---|---|
| $S$ | $\rightarrow ES'$ | | $\rightarrow ES'$ | | |
| $S'$ | | $\rightarrow +S$ | | $\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |
| $E$ | $\rightarrow$ N | | $\rightarrow ( S )$ | | |

- If $\gamma$ can derive $\varepsilon$ ($\gamma$ is *nullable*), add $\rightarrow \gamma$ for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicting entries

## Computing nullable, FIRST

- X is nullable if it can derive the empty string:
  - if it derives $\varepsilon$ directly
  - if it has a production $X \to YZ\ldots$ where all RHS symbols (Y, Z) are nullable
  - Algorithm: assume all non-terminals non-nullable, apply rules repeatedly until no change in status
- Determining FIRST($\gamma$)
  - FIRST(X) $\supseteq$ FIRST($\gamma$)    if $X \to \gamma$
  - FIRST(**a** $\beta$) = { **a** }
  - FIRST(X $\beta$) $\supseteq$ FIRST(X)
  - FIRST(X $\beta$) $\supseteq$ FIRST($\beta$) if X is nullable
  - **Algorithm:** Assume FIRST($\gamma$) = {} for all $\gamma$, apply rules repeatedly



## Computing FOLLOW

- FOLLOW($S$) $\supseteq$ { \$ }
- If $X \to \alpha Y \beta$,
  FOLLOW(Y) $\supseteq$ FIRST($\beta$)
- If $X \to \alpha Y \beta$ and $\beta$ is nullable (or non-existent),
  FOLLOW(Y) $\supseteq$ FOLLOW(X)
- **Algorithm:** Assume FOLLOW(X) = { } for all other X, apply rules repeatedly

- Common theme: iterative analysis. Start with initial assignment, apply rules until no change



## Example

$S \to ES'$
$S' \to \varepsilon \mid +S$
$E \to$ number $\mid$ ( $S$ )

- nullable
  - only $S'$ is nullable
- FIRST
  - FIRST($E\,S'$) = { **number** , **(** }
  - FIRST($+S$) = { **+** }
  - FIRST(number) = { **number** }
  - FIRST( ($S$) ) = { **(** }, FIRST($S'$) = { + }
- FOLLOW
  - FOLLOW($S$) = { **\$**, **)**, **+** }
  - FOLLOW($S'$) = { ), \$}
  - FOLLOW($E$) = { **+**, **)** }

| | **num** | **+** | **(** | **)** | **\$** |
|---|---|---|---|---|---|
| ***S*** | $\to ES'$ | | $\to ES'$ | | |
| ***S'*** | | $\to +S$ | | $\to \varepsilon$ | $\to \varepsilon$ |
| ***E*** | $\to$ num | | $\to (S)$ | | |



## Detecting ambiguity

- Construction of predictive parse table results in *conflicts* (*but converse does not hold*)

$S \to S + S \mid S * S \mid$ **num**

FIRST($S + S$) = FIRST($S * S$) = FIRST(**num**) = { **num** }

| | **num** | + | * |
|---|---|---|---|
| S | $\to$**num**, $\to S + S$, $\to S * S$ | | |



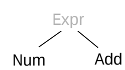
## Completing the parser

Now we know how to construct a recursive-descent parser for an LL(1) grammar.

Can we use recursive descent to build an abstract syntax tree too?



## Creating the AST



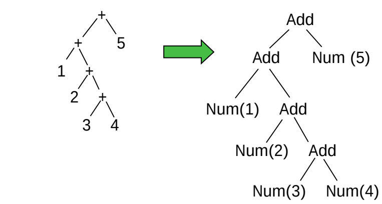

```
abstract class Expr { }

class Add extends Expr {
   Expr left, right;
   Add(Expr L, Expr R) { left = L; right = R; }
}

class Num extends Expr {
   int value;
   Num (int v) { value = v);
}
```

## AST Representation

(1 + 2 + (3 + 4)) + 5




How can we generate this structure during recursive-descent parsing?



## Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- parse_S, parse_S', parse_E all return an Expr:

    void parse_E() ⇒ Expr parse_E()
    void parse_S() ⇒ Expr parse_S()
    void parse_S'() ⇒ Expr parse_S'()



## AST creation code

```
Expr parse_E() {
    switch(token) {
      case number: // E → number
            Expr result = Num (token.value);
            token = input.read(); return result;
      case '(': // E → ( S )
            token = input.read();
            Expr result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
      default: throw new ParseError();
    }
}
```



## parse_S

$S \to ES'$
$S' \to \varepsilon \mid + S$
$E \to \text{number} \mid ( S )$

```
Expr parse_S() {
   switch (token) {
        case number:
        case '(':
            Expr left = parse_E();
            Expr right = parse_S'();
            if (right == null) return left;
            else return new Add(left, right);
        default: throw new ParseError();
   }
}
```



## Or...an Interpreter!

```
int parse_E() {
    switch(token) {
        case number:
            int result = token.value;
            token = input.read(); return result;
        case '(':
            token = input.read();
            int result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
            default: throw new ParseError(); }}

int parse_S() {
    switch (token) {
        case number:
        case '(':
          int left = parse_E();
          int right = parse_S'();
          if (right == 0) return left;
          else return left + right;
        default: throw new ParseError(); } }
```



## Summary

- We can build a recursive-descent parser for LL(1) grammars
  - Make parsing table from FIRST, FOLLOW
  - Translate to recursive-descent code
- Systematic approach avoids errors, detects ambiguities
- Next time: converting a grammar to LL(1) form, bottom-up parsing