CS 4120
Introduction to Compilers

Ross Tate
Cornell University

Lecture 5: Top-down parsing

Parsing Top-down

$S \to E + S \mid E$
$E \to \textbf{num} \mid (S)$

- **Goal:** construct a leftmost derivation of string while reading in token stream
- Partly-derived String / Lookahead / parsed part unparsed part

Partly-derived String	Lookahead	parsed part unparsed part
S	(	(1+2+(3+4))+5
→ E+S	(	(1+2+(3+4))+5
→ (S) +S	1	(1+2+(3+4))+5
→ (E+S)+S	1	(1+2+(3+4))+5
→ (1+S)+S	2	(1+2+(3+4))+5
→ (1+E+S)+S	2	(1+2+(3+4))+5
→ (1+2+S)+S	(	(1+2+(3+4))+5
→ (1+2+E)+S	(	(1+2+(3+4))+5
→ (1+2+(S))+S	3	(1+2+(3+4))+5
→ (1+2+(E+S))+S	3	(1+2+(3+4))+5

Problem

$S \to E + S \mid E$
$E \to \textbf{num} \mid (S)$

- Want to decide which production to apply based on next symbol
- (1) $S \to E \to (S) \to (E) \to (1)$
- (1)+2 $S \to E + S \to (S) + S \to (E) + S \to (1)+E \to (1)+2$
- *Why is this hard?*

Grammar is Problematic

- 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 a grammar LL(1)

$S \to E + S$
$S \to E$
$E \to \textbf{num}$
$E \to (S)$

⇩

$S \to ES'$
$S' \to \varepsilon$
$S' \to + S$
$E \to \textbf{num}$
$E \to (S)$

- **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 $(+E)^*$

Parsing with new grammar

$S \to ES'$ $S' \to \varepsilon \mid + S$ $E \to \textbf{num} \mid (S)$

S	(	(1+2+(3+4))+5
→ E S'	(	(1+2+(3+4))+5
→ (S) S'	1	(1+2+(3+4))+5
→ (E S') S'	1	(1+2+(3+4))+5
→ (1 S') S'	+	(1+2+(3+4))+5
→ (1+E S') S'	2	(1+2+(3+4))+5
→ (1+2 S') S'	+	(1+2+(3+4))+5
→ (1+2 + S) S'	(	(1+2+(3+4))+5
→ (1+2 + E S') S'	(	(1+2+(3+4))+5
→ (1+2 + (S) S') S'	3	(1+2+(3+4))+5
→ (1+2 + (E S') S') S'	3	(1+2+(3+4))+5
→ (1+2 + (3 S') S') S'	+	(1+2+(3+4))+5
→ (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
  - uses predictive parsing
  - driven by *predictive parsing table* of
    non-terminals × input symbols → productions

Predictive Parse Table

$S \to E\ S'$
$S' \to \varepsilon \mid +S$
$E \to \mathbf{num} \mid (\ S\ )$

	num	+	(	)	EOF($)
S	ES'		ES'		
S'		+S			ε
E	num		(S)		

How to Implement?

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

	num	+	(	)	EOF($)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

- Three procedures: parse_S, parse_S', parse_E

Recursive-Descent Parser

lookahead token

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

	num	+	(	)	EOF($)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(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();
    }
}
```

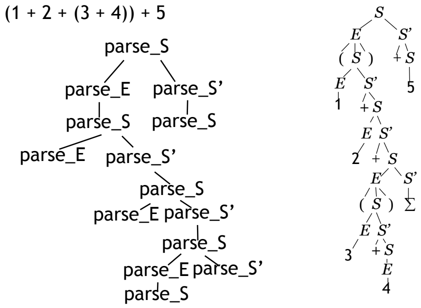
	num	+	(	)	EOF($)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

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(); }
}
```

	num	+	(	)	EOF($)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Call Tree = Parse Tree



How to Construct Parsing Tables

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

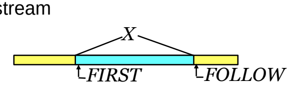
$S \rightarrow E\,S'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \textbf{num} \mid (\,S\,)$

?

	num	+	(	)	EOF
S	ES'		ES'		
S'		+S		ε	ε
E	num		(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(γ) for arbitrary string of terminals and non-terminals γ is:
  - set of symbols that might begin a fully expanded version of γ
- FOLLOW(X) for a non-terminal *X* is:
  - set of symbols that might follow the derivation of X in the input stream



Computing nullable and FIRST

- *X* is nullable if it can derive the empty string:
  - if it derives ε directly ($X \rightarrow \varepsilon$)
  - if it has a production $X \rightarrow YZ...$ 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(γ)
  - $FIRST(X) \supseteq FIRST(\gamma)$ if $X \rightarrow \gamma$
  - $FIRST(\textbf{a}\,\beta) = \{\,\textbf{a}\,\}$
  - $FIRST(X\beta) \supseteq FIRST(X)$
  - $FIRST(X\beta) \supseteq FIRST(\beta)$ if X is nullable
  - **Algorithm:** Assume $FIRST(\gamma) = \{\}$ for all γ, apply rules repeatedly to build *FIRST* sets.

Computing FOLLOW

- $FOLLOW(S) \supseteq \{\,\$\,\}$
- If $X \rightarrow \alpha Y \beta$,
  $FOLLOW(Y) \supseteq FIRST(\beta)$
- If $X \rightarrow \alpha Y \beta$ and β is nullable (or non-existent),
  $FOLLOW(Y) \supseteq FOLLOW(X)$
- **Algorithm:** Assume $FOLLOW(X) = \{\}$ for all X, apply rules repeatedly to build *FOLLOW* sets
- Common theme: iterative analysis. Start with initial assignment, apply rules until no change

Example

$S \rightarrow E\,S'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \textbf{num} \mid (\,S\,)$

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

Parse Table Entries

$S \to E\,S'$
$S' \to \varepsilon \mid +S$
$E \to \textbf{num} \mid (\,S\,)$

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

	num	+	(	)	EOF
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

- If γ can derive ε (γ is *nullable*), add γ for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicting entries

Ambiguous grammars

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

$S \to S + S \mid S * S \mid \textbf{num}$

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

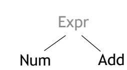
	num	+	*
S	**num**, *S* + *S*, *S* * *S*		

Completing the Parser

- Now we know how to construct a recursive-descent parser for an LL(1) grammar.
- LL(k) generalizes this to k lookahead tokens.
- LL(k) parser generators can be used to automate the process (e.g. ANTLR)
- Can we use recursive descent to build an abstract syntax tree too?

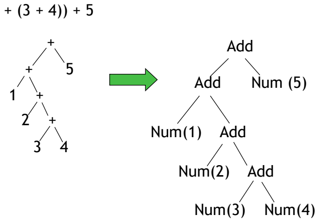
Creating the AST

```
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 to 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 num: // 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

$$S \rightarrow E\,S'$$
$$S' \rightarrow \varepsilon \mid +\,S$$
$$E \rightarrow \mathbf{num} \mid (\,S\,)$$

```
Expr parse_S() {
        switch (token) {
                case num:
                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* sets
  - Translate to recursive-descent code
  - Instrument with abstract syntax tree creation
- Systematic approach avoids errors, detects ambiguities
- Next time: converting a grammar to LL(1) form, bottom-up parsing