CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

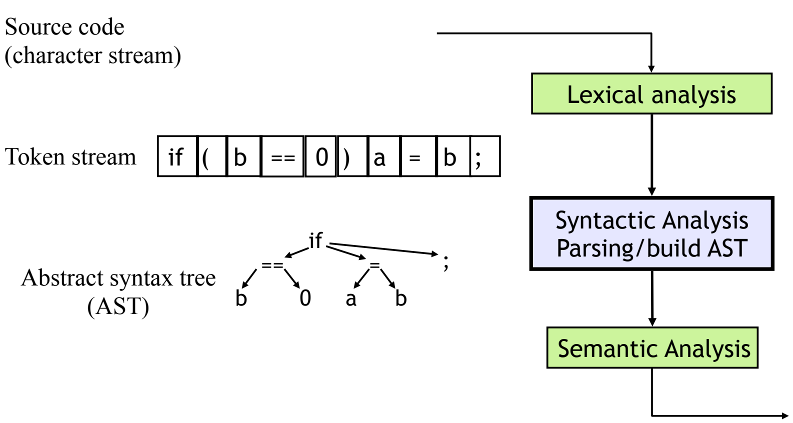
Lecture 5: Top-down parsing

7 Sep 2009

Outline

- More on writing 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

Top-down Parsing

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

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
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	2	(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 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 a grammar LL(1)

$S \to E + S$
$S \to E$
$E \to$ **num**
$E \to (S)$

⇓

$S \to ES'$
$S' \to \varepsilon$
$S' \to + S$
$E \to$ **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 $(+E)^*$
- **Also:** convert left-recursion to right-recursion

Parsing with new grammar

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

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
$\to (1{+}E\,S')\,S'$	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

$S \to E\,S'$
$S' \to \varepsilon \mid +S$
$E \to$ **num** $\mid (S)$

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

EOF

	num	+	(	)	$
S	$\to E\,S'$		$\to E\,S'$		
S'		$\to +S$		$\to \varepsilon$	$\to \varepsilon$
E	$\to$ **num**		$\to (S)$		

How to Implement?

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

	num	+	(	)	$
S	→ *E S'*		→ *E S'*		
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();
  }
}
```

	number	+	(	)	$
→ *S*	→ *ES'*		→ *ES'*		
S'		→ +*S*		→ ε	→ ε
E	→ number		→ (*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	→ *ES'*		→ *ES'*		
→ *S'*		→ +*S*		→ ε	→ ε
E	→ number		→ (*S*)		

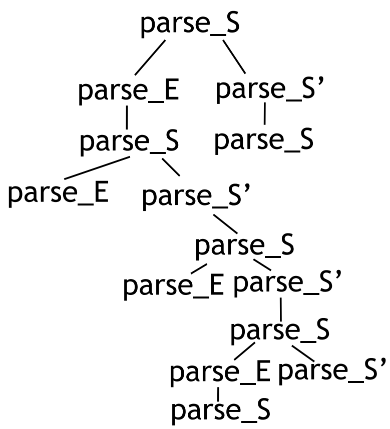
Recursive-Descent Parser

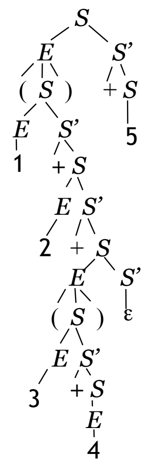
```
void parse_E() {
  switch (token) {
      case number: token = input.read(); return;
      case '(': token = input.read(); parse_S();
              if (token != ')') throw new ParseError();
              token = input.read(); return;
      default: throw new ParseError(); }
}
```

	number	+	(	)	$
S	→ *ES'*		→ *ES'*		
S'		→ +*S*		→ ε	→ ε
→ *E*	→ number		→ (*S*)		

Call Tree = Parse Tree

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





How to Construct Parsing Tables

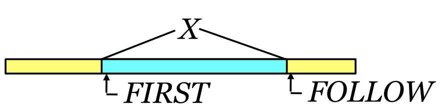
- Needed: algorithm for automatically generating a predictive parse table from a grammar

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

	N	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
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(γ) for arbitrary string of terminals and non-terminals γ is:
  - set of symbols that might begin the 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



Parse Table Entries

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

	num	**+**	**(**	**)**	**$**
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

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

Computing FOLLOW

- $FOLLOW(S) \supseteq \{\ \$\ \}$
- If $X \to$
  $\alpha Y\beta$,
  $FOLLOW(Y) \supseteq FIRST(\beta)$
- If $X \to \alpha Y\beta$ 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

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

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

	num	**+**	**(**	**)**	**$**
S	$\to E\,S'$		$\to E\,S'$		
S'		$\to +S$		$\to \varepsilon$	$\to \varepsilon$
E	$\to$ num		$\to (\ S\ )$		

Ambiguous grammars

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

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

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

	num	+	*
S	$\to \mathbf{num}, \to S + S, \to S * S$		

Completing the parser

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

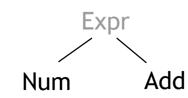
LL(k) generalizes this to k lookahead tokens.

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?

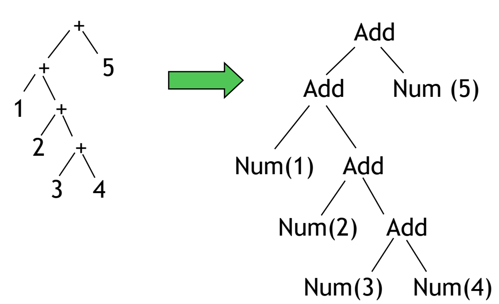
Creating the AST

```
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

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

```
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!

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

```
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