CS412/413

Introduction to Compilers
Radu Rugina

Lecture 6: Top-Down Parsing
3 Feb 2006

Outline

- Ambiguous grammars
- Top-down parsing
- LL(k) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing

An Ambiguous Grammar

- + associates to right because of right-recursive production $S \rightarrow E + S$
- Consider another grammar:

$$S \rightarrow S + S \mid S * S \mid num$$

- Ambiguous grammar : a string in the language has multiple parse trees
  - Note: this is different than multiple derivations

Different Parse Trees

$S \rightarrow S + S \mid S * S \mid num$

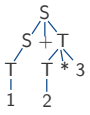
- Consider expression $1 + 2 * 3$
- Derivation 1: $\mathbf{S} \Rightarrow \mathbf{S} + S \Rightarrow 1 + \mathbf{S} \Rightarrow 1 + \mathbf{S} * S \Rightarrow 1 + 2 * \mathbf{S} \Rightarrow 1 + 2 * 3$
- Derivation 2: $\mathbf{S} \Rightarrow \mathbf{S} * S \Rightarrow \mathbf{S} + S * S \Rightarrow 1 + \mathbf{S} * S \Rightarrow 1 + 2 * \mathbf{S} \Rightarrow 1 + 2 * 3$

- These derivations correspond to different parse trees!
- Hence, the grammar is ambiguous

Eliminating Ambiguity

- Often can eliminate ambiguity by adding nonterminals and allowing recursion only on right or left

$S \rightarrow S + T \mid T$
$T \rightarrow T * num \mid num$



- T nonterminal enforces precedence
- Left-recursion : left-associativity

Parsing Top-down

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

Goal: construct a leftmost derivation of string while reading in token stream left to right

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 → E + S | E
E → **num** | (S)

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

(1) **S** ⇒ **E** ⇒ (**S**) ⇒ (**E**) ⇒ (1)

(1)+2 **S** ⇒ **E**+S ⇒ (**S**)+S ⇒ (**E**)+S
⇒ (1)+**E** ⇒ (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) = Left-to-right-scanning, producing Left-most derivation, using 1 symbol look-ahead
- 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 → E+S
S → E
E → **num**
E → (S)

⇓

S → ES'
S' → ε
S' → +S
E → **num**
E → (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 → ES' S' → ε | +S E → **num** | (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 G = ⟨V,Σ,S,→⟩
  - For a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - Top-down parsing a.k.a. predictive parsing
  - Driven by predictive parsing table that maps $V \times (\Sigma \cup \{\varepsilon\})$ to productions

Using Table

S → ES'
S' → ε | +S
E → **num** | (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+**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

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

How to Implement, Version 1

- A table-driven parser:

```
void parse(nonterminal A) {
  int i;
  let A → X0X1…Xn = TABLE[A,next]
  for (i=0; i<=n; i++) {
    if (Xi in Σ)
      if (next == Xi) next = input.read();
      else throw new ParseError();
    else parse(Xi);
  }
}
```


How to Implement, Version 2

- Convert table into a recursive-descent parser

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

- Three procedures: `parse_S`, `parse_S'`, `parse_E`

Recursive-Descent Parser

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

lookahead token

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

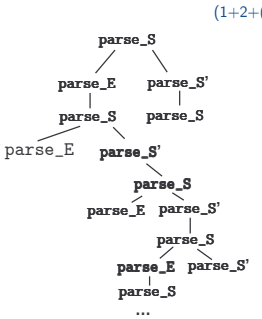
Recursive-Descent Parser

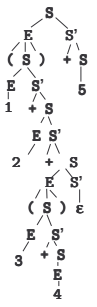
```
void parse_E() {
  switch (token) {
      case num: token = input.read(); return;
      case '(': token = input.read(); parse_S();
                if (token != ')') throw new ParseError();
                token = input.read(); return;
      default: throw new ParseError(); }
}
```

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

Call Tree = Parse Tree

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





How to Construct Parsing Tables

- Next time: automatically generating a predictive parse table from a grammar

S → ES'
S' → ε | +S
E → **num** | (S)

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

Summary

- LL(k) grammars
  - left-to-right scanning
  - leftmost derivation
  - can determine what production to apply from the next k symbols
- Predictive parsers
  - Can be easily built for LL(k) grammars from the parsing tables
  - Also called recursive-descent, or top-down parsers