CS412/413

Introduction to Compilers

Radu Rugina

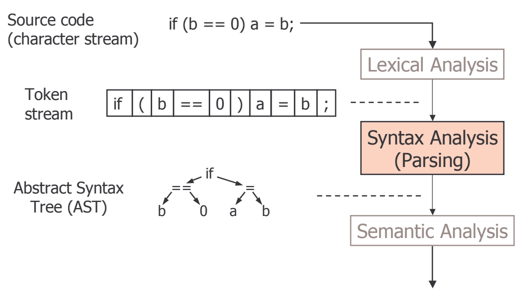
Lecture 6: Top-Down Parsing

06 Feb 04

Outline

- More on writing CFGs
- Top-down parsing
- LL(1) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing

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

if-then-else

- How to write a grammar for if stmts?

S → if (E) S

S → if (E) S else S

S → other

Is this grammar ok?

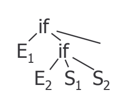
No—Ambiguous!

- How to parse?

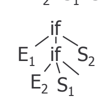
  if (E_1) if (E_2) S_1 else S_2

S → if (E) S S → if (E) S else S S → other

S → if (E) **S**
→ if (E) if (E) S else S



S → if (E) **S** else S
→ 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
- Impose that unmatched "if" statements occur only on the "else" clauses

statement	→	matched | unmatched
matched	→	if (E) matched else matched
	|	other
unmatched	→	if (E) statement
	|	if (E) matched else unmatched

Top-down Parsing

- Grammars for top-down parsing
- Implementing a top-down parser (recursive descent parser)

Parsing Top-down

S → E + S | E
E → **num** | (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 → 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, Left-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 → 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:
  - 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 × terminals → productions

Using Table

S → E S′
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	→ E S′		→ E S′		
S′		→ +S		→ ε	→ ε
E	→ **num**		→ (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

```
void parse_S () {        lookahead token
   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();
   }
}
```

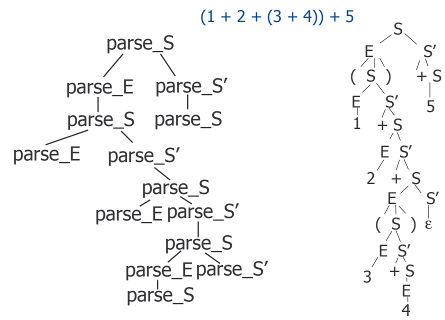
	number	+	(	)	$
S	→ ES′		→ ES′		
→ S′		→ +S		→ ε	→ ε
E	→ number		→ (S)		

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



How to Construct Parsing Tables

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

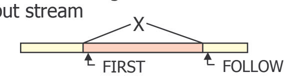
S → ES′
S′ → ε | + S
E → number | (S)

?

	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 → γ
- Add → γ 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 → γ 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→ ε)
  - if it has a production X→ YZ... where all RHS symbols (Y, Z) are nullable
  - Algorithm: assume all non-terminals non-nullable, apply rules repeatedly until no change
- Determining FIRST(γ)
  - FIRST(X) ⊇ FIRST(γ)   if X→ γ
  - FIRST(**a** β) = { **a** }
  - FIRST(X β) ⊇ FIRST(X)
  - FIRST(X β) ⊇ FIRST(β) if X is nullable
  - Algorithm: Assume FIRST(γ) = {} for all γ, apply rules repeatedly to build FIRST sets.

Computing FOLLOW

- Compute FOLLOW(X):
  - FOLLOW(S) ⊇ { $ }
  - If X → αYβ, FOLLOW(Y) ⊇ FIRST(β)
  - If X → αYβ and β is nullable (or non-existent), FOLLOW(Y) ⊇ 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 → E S′
S′ → ε | **+** S
E → **num** | **(** S **)**

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

	num	**+**	**(**	**)**	**$**
S	→ E S′		→ E S′		
S′		→ **+**S		→ ε	→ ε
E	→ num		→ (S)		

Ambiguous grammars

- Construction of predictive parse table for ambiguous grammar results in conflicts

S → S + S | S * S | **num**

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

	num	+	*
S	→**num**, →S + S, →S * S		

Summary

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