CS412/CS413

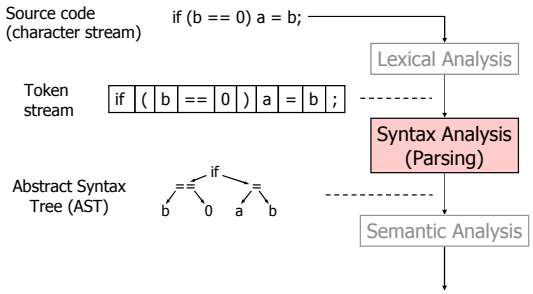
Introduction to Compilers
Tim Teitelbaum

Lecture 6: Top Down Parsing
February 4, 2005

Outline

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

Where We Are



Parsing Top-down

S → E + S | E
E → **num** | (S)

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

parsed part unparsed part

Partly-derived String	Lookahead	
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, 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 →

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	+	(	)	ε
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,token]
  for (i=0; i<=n; i++) {
      if (Xi in Σ)
          if (token == Xi) token = input.read();
          else throw new ParseError();
      else parse(Xi);
  }
  return;
}
```


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

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

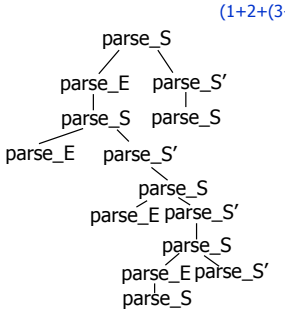
Recursive-Descent Parser

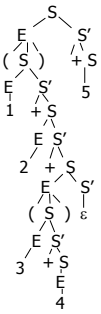
```
void parse_E() {
  switch (token) {
      case number: token = input.read(); return;
      case '(': token = input.read(); parse_S();
              if (token != ')') throw new ParseError();
              token = input.read(); return;
      default: throw new ParseError(); }
}
```

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

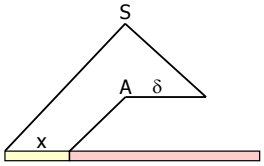
S → ES′
S′ → ε | +S
E → **num** | (S)

?

	N	+	(	)	ε
S	ES′		ES′		
S′		+S		ε	ε
E	N		(S)		

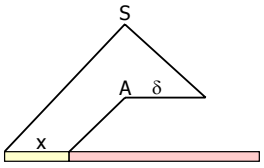
Intuition for LL(k)

- S ⇒* xAδ



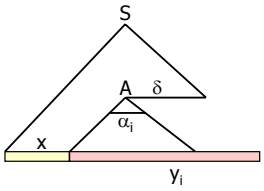
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$



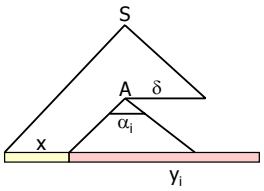
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$



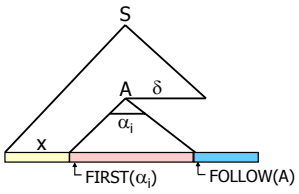
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$
- If $\alpha_1 \neq \alpha_2$ then y_1 and y_2 must differ in first k symbols



Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$
- If $\alpha_1 \neq \alpha_2$ then y_1 and y_2 must differ in first k symbols



Prerequisite Definitions

- Length of string w, denoted |w|, is the number of symbols in w
- k-limited prefix of string $w=a_1 \ldots a_n$, denoted k:w, is w (if $|w| \leq k$) and $a_1 \ldots a_k$ (if $|w| > k$)
- $FIRST_k(\alpha)$ = { k:w | $\alpha \Rightarrow^* w$ }
- $FOLLOW_k(A)$ = { $FIRST_k(\beta)$ | $S \Rightarrow^* \alpha A \beta$ }

LL(k) Grammars

- A grammar is LL(k) if for every nonterminal A,

  if $S \Rightarrow^* xA\delta$, and $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ are distinct productions, then $FIRST_k(\alpha_1\delta) \cap FIRST_k(\alpha_2\delta) = \varnothing$
- A grammar is SLL(k), known as strong LL(k), if for every pair of distinct productions $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$, $FIRST_k(\alpha_1 FOLLOW(A)) \cap FIRST_k(\alpha_2 FOLLOW(A)) = \varnothing$

LL vs SLL

- Grammars
  - LL(1) = SLL(1)
    - Proof?
  - LL(k) ≠ SLL(k), for k≥2
    - Example?
- Languages
  - Does every language with an LL(k) grammar have an SLL(k) grammar?

LL vs SLL

- LL(2) ≠ SLL(2)
  S → aAab | bAbb
  A → ε | a
- Language = {aab, aaab, bbb, babb}
- $FOLLOW_2(A)$ = {ab,bb}
- But is LL(2)
  - $FIRST_2$(ε ab) = {ab}; $FIRST_2$(a ab) = {aa} OK
  - $FIRST_2$(ε bb) = {bb}; $FIRST_2$(a bb) = {ab} OK
- But is not SLL(2)
  - $FIRST_2$(ε $FOLLOW_2$(A)) = {ab,bb};
  - $FIRST_2$(a $FOLLOW_2$(A)) = {aa,ab} BAD

LL(1) Parse Table Entries

- Consider each production A→α
  - Add →α to the A row for each symbol in FIRST(α FOLLOW(X))

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

  - If α ⇒* ε and S ⇒* wA then add →α to the A row for ε
- Grammar is LL(1) if no conflicting entries arise

Computing nullable

- B is nullable if it can derive the empty string
- Algorithm

nullable = { A | A → ε }

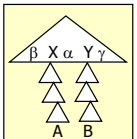
while (nullable changed)

if there exists A → $B_1 \ldots B_n$ with all B_i nullable

then nullable := nullable U {A}

Computing FIRST and FOLLOW

- Let G = ⟨V,Σ,S,→⟩ be a CFG
- Define three binary relations λ, μ, and ρ
  - AλX if A → αXβ and α is nullable
  - XμY if A → βXαYγ and α is nullable
  - AρX if A → βXα and α is nullable
- FIRST = $\lambda^+ \cap (V \times \Sigma)$
- FOLLOW = $((\rho^{-1})^* \bullet \mu \bullet \lambda^*) \cap (V \times \Sigma)$



if $X\rho^*A$, $X\mu Y$, $Y\lambda^*B$ then A FOLLOW B

Example

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

S → ES′
S′ → ε | **+**S
E → **num** | **(**S**)**

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