CS1622

Lecture 10
Parsing (5)

LL(1) Parsing Table Example

- Left-factored grammar
  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε
- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production E → T X
  - This production can generate an int in the first place
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation in which Y → ε

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals - instead of procedure stack!
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
   case stack of
      <X, rest>  : if T[X,*next] = Y1…Yn
                        then stack ← <Y1… Yn
   rest>;
                        else  error ();
      <t, rest>   : if t == *next ++
                        then  stack ← <rest>;
                        else error ();
until stack == < >
```

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- We want to generate parsing tables from CFG

Constructing Parsing Tables (Cont.)

- If $A \rightarrow \alpha$, where in the line of A we place α ?
- In the column of t where t can start a string derived from α
  - $\alpha \rightarrow^{*} t\,\beta$
  - We say that $t \in First(\alpha)$
- In the column of t if α is ε and t can follow an A
  - $S \rightarrow^{*} \beta\, A\, t\, \delta$
  - We say $t \in Follow(A)$

Computing First Sets

Definition

$\text{First}(X) = \{ t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$

Algorithm:

- $\text{First}(t) = \{ t \}$
- $\varepsilon \in \text{First}(X)$ if $X \to \varepsilon$ is a production
- $\varepsilon \in \text{First}(X)$ if $X \to A_1 \dots A_n$
  - and $\varepsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$
- $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \to A_1 \dots A_n\ \alpha$
  - and $\varepsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$

First Sets. Example

- Recall the grammar

  $E \to T\ X$ $\qquad X \to +\ E \mid \varepsilon$

  $T \to (\ E\ ) \mid \text{int}\ Y$ $\qquad Y \to *\ T \mid \varepsilon$

- First sets

  First(( ) = { (} $\qquad$ First(T) = {int, (}

  First( )) = {) } $\qquad$ First(E) = {int, (}

  First(int) = { int } $\qquad$ First(X) = {+, ε }

  First(+) = { + } $\qquad$ First(Y) = {*, ε }

  First(*) = { * }

Computing Follow Sets for Nonterminals

- Follow(S) needed for productions that generate the empty string ε
- Definition:

  $\text{Follow}(X) = \{ t \mid S \to^* \beta\ X\ t\ \delta \}$

  - Note that ε CAN NEVER BE IN FOLLOW(X)!!
- Intuition
  - If $X \to A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \to^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
  - If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

- $\$ \in \text{Follow}(S)$
- For each production $A \to \alpha X \beta$
  - $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
- For each production $A \to \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$
  $\text{Follow}(A) \subseteq \text{Follow}(X)$

Follow Sets. Example

- Recall the grammar
  $E \to T X$ $\quad\quad X \to + E \mid \varepsilon$
  $T \to (E) \mid \text{int } Y$ $\quad\quad Y \to * T \mid \varepsilon$
- Follow sets

  Follow(E) = {), $}
  Follow(X) = {$,) }
  Follow(T) = {+,) , $}
  Follow(Y) = {+,) , $}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1). Possible reasons:
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - Grammar is not LL(1)
- Most programming language grammars are not LL(1)
  - Some can be made LL(1) though but others can't
- There are tools that build LL(1) tables
  - E.g. LLGen

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method in practice
- Concepts first, algorithms next time

An Introductory Example

- Bottom-up parsers don't need left-factored grammars
- Hence we can revert to the "natural" grammar for our example:

  E → T + E | T
  T → int * T | int | (E)

- Consider the string: int * int + int

Bottom-Up Parsing

The Idea

Bottom-up parsing *reduces* a string to the start symbol by inverting productions:

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

Observation

- Read the productions in parse in reverse (i.e., from bottom to top)
- This is a rightmost derivation!

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

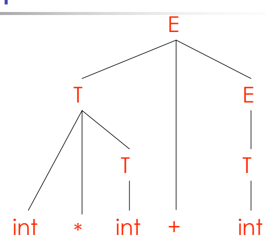
Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

int * int + int
int * T + int
T + int
T + T
T + E
E



A Bottom-up Parse in Detail (1)

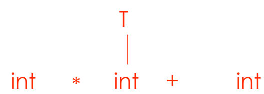
int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

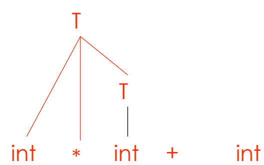


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



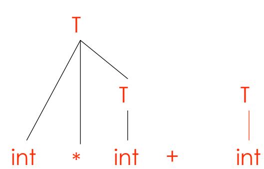
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

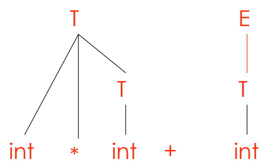
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

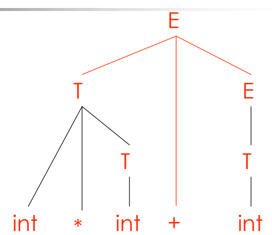
int * T + int

T + int

T + T

T + E

E



A Trivial Bottom-Up Parsing Algorithm

Let I = input string
  repeat
    pick a non-empty substring β of I
      where X→ β is a production
    if no such β, backtrack
    replace one β by X in I
  until I = "S" (the start symbol) or all possibilities are exhausted

Questions

- Does this algorithm terminate?
- How fast is the algorithm?
- Does the algorithm handle all cases?
- How do we choose the substring to reduce at each step?

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why? Because $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a |
  - The | is not part of the string
- Initially, all input is unexamined $|x_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- *Shift:* Move | one place to the right
  - Shifts a terminal to the left string

ABC|xyz $\Rightarrow$ ABCx|yz

Reduce

- Apply an *inverse production* at the right end of the left string
  - If A → xy is a production, then

Cbxy|ijk $\Rightarrow$ CbA|ijk

The Example with Reductions Only

int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

A Shift-Reduce Parse in Detail (1)

|int * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

|int * int + int
int | * int + int

int * int + int

A Shift-Reduce Parse in Detail (3)

|int * int + int
int | * int + int
int * | int + int

int * int + int

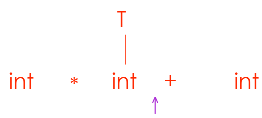
A Shift-Reduce Parse in Detail (4)

|int * int + int
int | * int + int
int * | int + int
int * int | + int

int * int + int

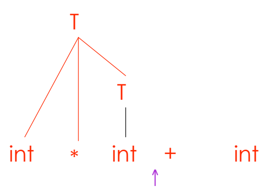
A Shift-Reduce Parse in Detail (5)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int



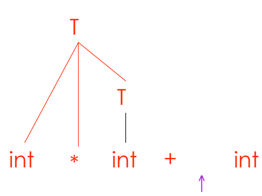
A Shift-Reduce Parse in Detail (6)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int



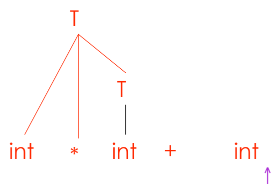
A Shift-Reduce Parse in Detail (7)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int



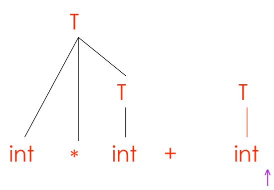
A Shift-Reduce Parse in Detail (8)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |



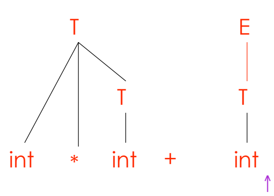
A Shift-Reduce Parse in Detail (9)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |



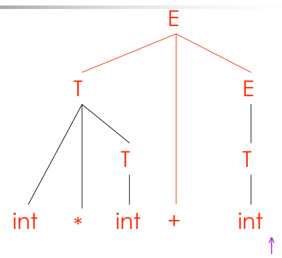
A Shift-Reduce Parse in Detail (10)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |



A Shift-Reduce Parse in Detail (11)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Key Issue (will be resolved by algorithms)

- How do we decide when to shift or reduce?
  - Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake: No way to reduce to the start symbol E

Conflicts

- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift
- But what if there is a choice?
  - If it is legal to shift or reduce, there is a *shift-reduce* conflict
  - If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict

Source of Conflicts

- Ambiguous grammars always cause conflicts
- But beware, so do many non-ambiguous grammars

Conflict Example

Consider our favorite ambiguous grammar:

$$
\begin{aligned}
E &\rightarrow E + E \\
&\mid E * E \\
&\mid (E) \\
&\mid \text{int}
\end{aligned}
$$

One Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	reduce E → E * E
E | + int	shift
E + | int	shift
E + int|	reduce E → int
E + E |	reduce E → E + E
E |	

Another Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	shift
E * E + | int	shift
E * E + int |	reduce E → int
E * E + E|	reduce E → E + E
E * E |	reduce E → E * E
E |	

Example Notes

- In the second step E * E | + int we can either shift or reduce by E → E * E
- Choice determines associativity of + and *
- As noted previously, grammar can be rewritten to enforce precedence
- Precedence declarations are an alternative

Precedence Declarations Revisited

- Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways
- Declaring "* has greater precedence than +" causes parser to reduce at E * E | + int
- More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +

Precedence Declarations Revisited (Cont.)

- The term "precedence declaration" is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!