Building a Parser III

CS164
3:30-5:00 TT
10 Evans

Administrativia

- WA1 due on Thu
- PA2 in a week

- Slides on the web site
  - I do my best to have slides ready and posted by the end of the preceding logical day
  - yesterday, my best was not good enough

Overview

- Finish recursive descent parser
  - when it breaks down and how to fix it
  - eliminating left recursion
  - reordering productions
- Predictive parsers (aka LL(1) parsers)
  - computing FIRST, FOLLOW
  - table-driven, stack-manipulating version of the parser

Review: grammar for arithmetic expressions

- Simple arithmetic expressions:
  $E \rightarrow n \mid id \mid (E) \mid E + E \mid E * E$

- Some elements of this language:
  - id
  - n
  - (n)
  - n + id
  - id * (id + id)

Review: derivation

Grammar: $E \rightarrow n \mid id \mid (E) \mid E + E \mid E * E$

- a derivation:

E	rewrite E with (E)
(E)	rewrite E with n
(n)	this is the final string of terminals

- another derivation (written more concisely):
  $E \rightarrow (E) \rightarrow (E * E) \rightarrow (E + E * E) \rightarrow (n + E * E) \rightarrow (n + id * E) \rightarrow (n + id * id)$
- this is left-most derivation (remember it)
  - always expand the left-most non-terminal
  - can you guess what's right-most derivation?

Recursive Descent Parsing

- Consider the grammar
  $E \rightarrow T + E \mid T$
  $T \rightarrow int \mid int * T \mid (E)$
- Token stream is: $int_5 * int_2$
- Start with top-level non-terminal E

- Try the rules for E in order

Recursive-Descent Parsing

- Parsing: given a string of tokens $t_1\ t_2\ \ldots\ t_n$, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1\ t_2\ \ldots\ t_k\ A\ \ldots$
  - Try all the productions for A: if A → BC is a production, the new fringe is $t_1\ t_2\ \ldots\ t_k\ B\ C\ \ldots$
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production S → S a:
  - In the process of parsing S we try the above rule
  - What goes wrong?
- A fix?
  - S must have a non-recursive production, say S → b
  - expand this production before you expand S → S a
- Problems remain
  - performance (steps needed to parse "baaaaa")
  - termination (parse the error input "c")

Solutions

- First, restrict backtracking
  - backtrack just enough to produce a sufficiently powerful r.d. parser
- Second, eliminate left recursion
  - transformation that produces a different grammar
  - the new grammar generates same strings
  - but does it give us same parse tree as old grammar?

- Let's see the restricted r.d. parser first

A Recursive Descent Parser (1)

- Define boolean functions that check the token string for a match of
  - A given token terminal

    `bool term(TOKEN tok) { return in[next++] == tok; }`
  - A given production of S (the nth)

    bool S_n() { … }
  - Any production of S:

    `bool S() { … }`

- These functions advance next

A Recursive Descent Parser (2)

- For production E → T + E

  bool E_1() { return T() && term(PLUS) && E(); }
- For production E → T

  bool E_2() { return T(); }
- For all productions of E (with backtracking)

```
bool E() {
  int save = next;
  return   (next = save, E1())
        || (next = save,  E2());  }
```

A Recursive Descent Parser (3)

- Functions for non-terminal T

```
bool T1() { return term(OPEN) && E() && term(CLOSE); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(INT); }

   bool T() {
     int save = next;
     return   (next = save, T1())
           || (next = save,  T2())
           || (next = save,  T3()); }
```

Recursive Descent Parsing. Notes.

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- Notice how this simulates our backtracking example from lecture
- Easy to implement by hand
- Predictive parsing is more efficient

Now back to left-recursive grammars

- Does this style of r.d. parser work for our left-recursive grammar?
  - the grammar: $S \to S\ a \mid b$
  - what happens when $S \to S\ a$ is expanded first?
  - what happens when $S \to b$ is expanded first?

Left-recursive grammars

- A <u>left-recursive grammar</u> has a non-terminal S

  $S \to^{+} S\alpha$ for some α
- Recursive descent does not work in such cases
  - It goes into an ∞ loop
- Notes:
  - α: a shorthand for any string of terminals, non-terminals
  - symbol $\to^{+}$ is a shorthand for "can be derived in one or more steps":
    - $S \to^{+} S\alpha$ is same as $S \to \ldots \to S\alpha$

Elimination of Left Recursion

- Consider the left-recursive grammar

  $S \to S\ \alpha \mid \beta$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

  $S \to \beta\ S'$

  $S' \to \alpha\ S' \mid \varepsilon$

Elimination of Left-Recursion. Example

- Consider the grammar

  $S \to 1 \mid S\ 0$ ($\beta = 1$ and $\alpha = 0$)

can be rewritten as

$S \to 1\ S'$

$S' \to 0\ S' \mid \varepsilon$

More Elimination of Left-Recursion

- In general

  $S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

  $S \to \beta_1\ S' \mid \ldots \mid \beta_m\ S'$

  $S' \to \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar

  $S \to A\,\alpha \mid \delta$

  $A \to S\,\beta$

  is also left-recursive because

  $S \to^{+} S\,\beta\,\alpha$

- This left-recursion can also be eliminated
- See [ASU], Section 4.3 for general algorithm

Summary of Recursive Descent

- simple parsing strategy
  - left-recursion must be eliminated first
  - … but that can be done automatically
- unpopular because of backtracking
  - thought to be too inefficient
  - in practice, backtracking is (sufficiently) eliminated by restricting the grammar
- so, it's good enough for small languages
  - careful, though: order of productions important even after left-recursion eliminated
  - try to reverse the order of $E \to T + E \mid T$
  - what goes wrong?

Predictive parsers

Motivation

- Wouldn't it be nice if
  - the r.d. parser just knew which production to expand next?
  - Idea: replace

```
return   (next = save, E1())  || (next = save,  E2());  }
```

  - with

```
switch ( something ) {
case L1: return E1();
case L2: return E2();
otherwise: print "syntax error";
}
```

  - what's "something", L1, L2?
    - the parser will do lookahead (look at next token)

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
- In practice, LL(1) is used

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- Can be specified as a 2D table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Left factoring

Predictive Parsing and Left Factoring

- Recall the grammar
  - E → T + E | T
  - T → int | int * T | (E)
- Impossible to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- A grammar must be <u>left-factored</u> before use predictive parsing

---

Left-Factoring Example

- Recall the grammar
  - E → T + E | T
  - T → int | int * T | (E)
- Factor out common prefixes of productions
  - E → T X
  - X → + E | ε
  - T → (E) | int Y
  - Y → * T | ε

LL(1) parser (details)

LL(1) parser

- to simplify things, instead of

```
switch ( something ) {
case L1: return E1();
case L2: return E2();
otherwise: print “syntax error”;
}
```

- we’ll use a LL(1) <u>table</u> and a <u>parse stack</u>
  - the LL(1) table will replace the switch
  - the parse stack will replace the call stack

LL(1) Parsing Table Example

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

	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ε	ε
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"
  - We'll see later why this is so

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 choose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- 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 (pointer to tokens)
repeat
  case stack of
    <X, rest>  : if T[X,*next] = $Y_1…Y_n$
                   then stack ← <$Y_1…Y_n$ 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 Predictive Parsing Tables

- Consider the state $S \rightarrow^* \beta A \gamma$
  - With b the next token
  - Trying to match $\beta b \delta$

There are two possibilities:

1. b belongs to an expansion of A
   - Any $A \rightarrow \alpha$ can be used if b can start a string derived from α

     In this case we say that $b \in \text{First}(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. b does not belong to an expansion of A
   - The expansion of A is empty and b belongs to an expansion of γ
   - Means that b can appear after A in a derivation of the form $S \rightarrow^* \beta A b \omega$
   - We say that $b \in \text{Follow}(A)$ in this case

   - What productions can we use in this case?
     - Any $A \rightarrow \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in \text{First}(A)$ in this case

Computing First, Follow sets

First Sets. Example

- Recall the grammar

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

  $T \rightarrow (E) \mid \text{int } Y$ $\qquad$ $Y \rightarrow * T \mid \varepsilon$
- First sets

  First(( ) = { (}

  First( )) = {) }

  First(int) = { int }

  First(+) = { + }

  First(*) = { * }

  First(T) = {int, (}

  First(E) = {int, (}

  First(X) = {+, ε }

  First(Y) = {*, ε }

Computing First Sets

Definition $\quad \text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}$

1. First(b) = { b }

2. For all productions $X \rightarrow A_1 \dots A_n$
   - Add $\text{First}(A_1) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin \text{First}(A_1)$
   - Add $\text{First}(A_2) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin \text{First}(A_2)$
   - …
   - Add $\text{First}(A_n) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin \text{First}(A_n)$
   - Add ε to First(X)

3. Repeat step 2 until no First set grows

Follow Sets. Example

- Recall the grammar

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

  $T \rightarrow (E) \mid \text{int } Y$ $\qquad$ $Y \rightarrow * T \mid \varepsilon$
- Follow sets

  Follow(+) = { int, (}

  Follow(( ) = { int, (}

  Follow(X) = {$,) }

  Follow( )) = {+,) , $}

  Follow(int) = {*, +,) , $}

  Follow(*) = { int, (}

  Follow(E) = {), $}

  Follow(T) = {+,) , $}

  Follow(Y) = {+,) , $}

Computing Follow Sets

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

1. Compute the First sets for all non-terminals first
2. Add \$ to Follow(S) (if S is the start non-terminal)

3. For all productions $Y \to \ldots X A_1 \ldots A_n$
    - Add $\text{First}(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_1)$
    - Add $\text{First}(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_2)$
    - …
    - Add $\text{First}(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_n)$
    - Add Follow(Y) to Follow(X)

4. Repeat step 3 until no Follow set grows

Constructing the LL(1) parsing table

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 $b \in \text{First}(\alpha)$ do
        - $T[A, b] = \alpha$
    - If $\alpha \to^* \varepsilon$, for each $b \in \text{Follow}(A)$ do
        - $T[A, b] = \alpha$
    - If $\alpha \to^* \varepsilon$ and $\$ \in \text{Follow}(A)$ do
        - $T[A, \$] = \alpha$

Constructing LL(1) Tables. Example

- Recall the grammar

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

- Where in the line of Y we put $Y \to *\,T$?
    - In the lines of First(*T) = { * }

- Where in the line of Y we put $Y \to \varepsilon$?
    - In the lines of Follow(Y) = { \$, +,) }

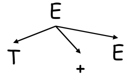
Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
    - If G is ambiguous
    - If G is left recursive
    - If G is not left-factored
    - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

Notes and Review

Top-Down Parsing. Review

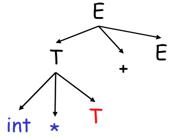
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

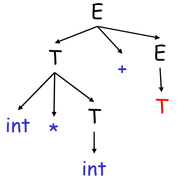


- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

int * int + int

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

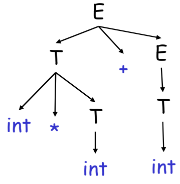


- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

int * int + int

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

int * int + int

Predictive Parsing. Review.

- A predictive parser is described by a table
  - For each non-terminal A and for each token b we specify a production $A \rightarrow \alpha$
  - When trying to expand A we use $A \rightarrow \alpha$ if b follows next

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary