CSCI 742 - Compiler Construction

Lecture 12

Recursive-Descent Parsers

Instructor: Hossein Hojjat

February 12, 2018

Recap: Predictive Parsers

- Predictive Parser:
  Top-down parser that looks at the next few tokens
  and predicts which production to use
- Efficient: no need for backtracking, linear parsing time
- 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

Implementations

Analogous to lexing:

Recursive descent parser (manual)

- Each non-terminal parsed by a procedure
- Call other procedures to parse sub-nonterminals recursively
- Typically implemented manually

Table-driven parser (automatic)

- Push-down automata: essentially a table driven FSA, plus stack to do recursive calls
- Typically generated by a tool from a grammar specification

Making Grammar LL

- Recall the left-recursive grammar:

$$S \to S + E \mid E$$
$$E \to \mathrm{num} \mid (E)$$

- Grammar is not LL(k) for any number of k
- **Left-recursion elimination:** rewrite left-recursive productions to right-recursive ones

$$S \to S + E \mid E \qquad\qquad S \to EE'$$
$$E \to \mathrm{num} \mid (E) \quad \Longrightarrow \quad E' \to +EE' \mid \epsilon$$
$$\qquad\qquad\qquad\qquad E \to \mathrm{num} \mid (E)$$

Making Grammar LL

- Recall the grammar:

$$S \to E + E \mid E$$
$$E \to \text{num} \mid (E)$$

- Grammar is not LL(k) for any number of k
- **Left-factoring:** Factor common prefix E, add new non-terminal E' for what follows that prefix

$$S \to E + E \mid E$$
$$E \to \text{num} \mid (E)$$

$\Longrightarrow$

$$S \to EE'$$
$$E' \to +E \mid \epsilon$$
$$E \to \text{num} \mid (E)$$

Parsing with new grammar

$$S \rightarrow EE' \qquad E' \rightarrow +E \mid \epsilon \qquad E \rightarrow \text{num} \mid (E)$$

Partly-derived String	Lookahead	parsed part unparsed part
$\mathbf{S}$	(	(num) +num
$\Rightarrow \mathbf{E}\ E'$	(	(num) +num
$\Rightarrow (\mathbf{E})\ E'$	num	(num) +num
$\Rightarrow (\text{num})\ \mathbf{E'}$	+	(num) +num
$\Rightarrow (\text{num}) + \mathbf{E}$	num	(num) +num
$\Rightarrow (\text{num}) + \text{num}$	\$	(num) +num

Predictive Parsing

- Predictive parser chooses the next production to use by
  - next few token symbols
  - current non-terminal being processed

In practice LL(1) is used:

- When a non-terminal A is leftmost in a derivation
- The next input symbol is a
- There is a unique production $A \rightarrow \alpha$ to use
  - Or no production to use (an error state)
- Predict decisions can be encoded into a table called a parse table

$$\text{non-terminal} \times \text{input symbol} \rightarrow \text{production}$$

Using Table

$S \to EE'$ $\quad E' \to +E \mid \epsilon$ $\quad E \to \text{num} \mid (E)$

Partly-derived String	Lookahead	parsed part unparsed part
S	(	(num)+num
$\Rightarrow \mathbf{E}\ E'$	(	(num)+num
$\Rightarrow (\mathbf{E})\ E'$	num	(num)+num
$\Rightarrow (\text{num})\ \mathbf{E'}$	+	(num)+num
$\Rightarrow (\text{num}) + \mathbf{E}$	num	(num)+num
$\Rightarrow (\text{num}) + \text{num}$	$	(num)+num

	num	+	(	)	$
S	$\to EE'$		$\to EE'$		
E'		$\to +E$			$\to \epsilon$
E	$\to \text{num}$		$\to (E)$		

- Dollar sign $ is end of file marker

Implementation

Two ways to implement a predictive parser based on parsing table:

1. Recursive descent parser
2. Non-recursive parser with explicit stack

Recursive-descent parser idea:

- Associate a procedure with each nonterminal in the grammar

	num	+	(	)	$
S	$\rightarrow EE'$		$\rightarrow EE'$		
E'		$\rightarrow +E$			$\rightarrow \epsilon$
E	$\rightarrow$ num		$\rightarrow (E)$		

- Three procedures: `parse_S`, `parse_E'`, `parse_E`
- Implement a recursive descent parser using mutually recursive procedures

Recursive-Descent Parser

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

lookahead token

	num	+	(	)	\$
S	$\to EE'$		$\to EE'$		
E'		$\to +E$			$\to \epsilon$
E	$\to \mathrm{num}$		$\to (E)$		

Recursive-Descent Parser

```
void parse_E' () {
  switch(lexer.current) {
    case '+':
      lexer.next(); parse_E(); return;
    case EOF: return;
    default: throw new ParseError();
  }
}
```

	num	+	(	)	$
S	$\to EE'$		$\to EE'$		
E'		$\to +E$			$\to \epsilon$
E	$\to \text{num}$		$\to (E)$		

Recursive-Descent Parser

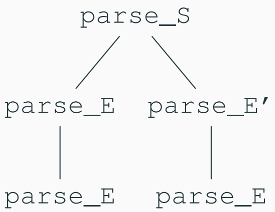
```
void parse_E () {
  switch(lexer.current) {
    case num:
          lexer.next(); return;
    case '(':
          lexer.next(); parse_E();
          if(lexer.current != ')')
              throw new ParseError();
          lexer.next(); return;
    default: throw new ParseError();
  }
}
```

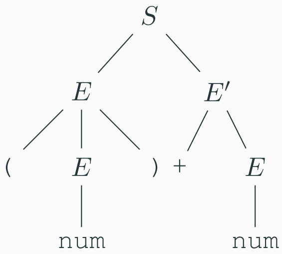
	num	+	(	)	$
S	$\to EE'$		$\to EE'$		
E'		$\to +E$			$\to \epsilon$
E	$\to$ num		$\to (E)$		

Call Tree = Parse Tree

$S \to EE'$ $\qquad$ $E' \to +E \mid \epsilon$ $\qquad$ $E \to \mathtt{num} \mid (E)$

input: `( num ) + num`



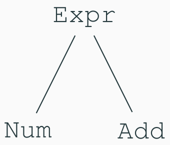


- Now we know:
  How to construct a recursive-descent parser from the parsing table
- Can we use recursive descent to build an abstract syntax tree too?

Creating the AST

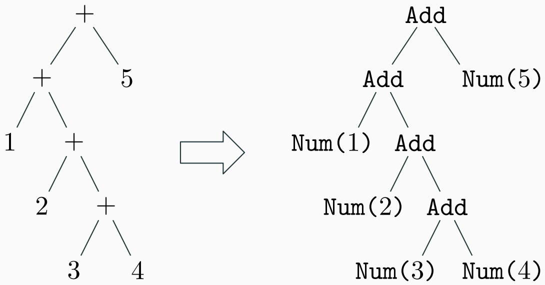
$$S \to EE'$$
$$E' \to \epsilon \mid +E$$
$$E \to \text{num} \mid (E)$$

```
abstract class Expr { }
class Add extends Expr {
  Expr left, right;
  Add(Expr L, Expr R) {
    left = L; right = R;
  }
}
class Num extends Expr {
  int value;
  Num (int v) { value = v);
}
```



AST Representation

input: `(1 + 2 + (3 + 4)) + 5`



How can we generate the AST during recursive descent parsing?

AST Generation Procedures

- Just add code to each parsing procedure to create the appropriate nodes
- Works because parse tree and call tree have same shape
- `parse_S`, `parse_S'`, `parse_E` all return an `Expr`:

```
void parse_E()      ⇒      Expr parse_E()
void parse_S()      ⇒      Expr parse_S()
void parse_S'()     ⇒      Expr parse_S'()
```

AST Creation: `parse_S`

```
Expr parse_S() {
  switch (lexer.current) {
    case num:
    case '(':
      Expr left = parse_E();
      Expr right = parse_E'();
      if (right == null) return left;
        else return new Add(left, right);
    default: throw new ParseError();
  }
}
```

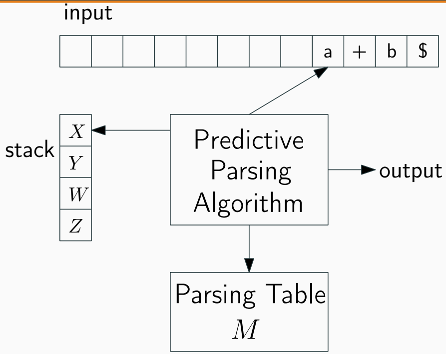
	num	+	(	)	$
S	$\rightarrow EE'$		$\rightarrow EE'$		
E'		$\rightarrow +E$			$\rightarrow \epsilon$
E	$\rightarrow$ num		$\rightarrow (E)$		

AST Creation: parse_E

```
Expr parse_E() {
  switch(lexer.current) {
    case num: // E → num
      Expr result = Num (lexer.current.value);
      lexer.next(); return result;
    case '(': // E → ( E )
      lexer.next();
      Expr result = parse_E();
      if (lexer.current != ')') throw new
          ParseError();
      lexer.next(); return result;
    default: throw new ParseError();
}}
```

	num	+	(	)	$
S	$\rightarrow EE'$		$\rightarrow EE'$		
E'		$\rightarrow +E$			$\rightarrow \epsilon$
E	$\rightarrow$ num		$\rightarrow (E)$		

Non-Recursive LL(1) Parsers




```
while S.top() is not $ do
 if S.top() = current then
   S.pop()
   advance()
 else ... // error
 else if M[S.top(),current]=X → Y1Y2 ... Yk
    then
   output production X → Y1Y2 ... Yk
   S.pop()
   Push Yk,...,Y2,Y1 onto S with Y1 on top
```