CSCI 742 - Compiler Construction

Lecture 15

Recursive-Descent Parsers

Instructor: Hossein Hojjat

February 27, 2017

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

non-LL(1) Grammar

- Consider the grammar:

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

- and the two derivations

$$S \Rightarrow E \qquad \Rightarrow (E) \qquad \Rightarrow (\mathsf{num})$$
$$S \Rightarrow E + E \Rightarrow (E) + E \Rightarrow (\mathsf{num}) + E \Rightarrow (\mathsf{num}) + \mathsf{num}$$

- **Question.** Can we decide between

$$S \Rightarrow E$$
$$S \Rightarrow E + E$$

  as the first derivation step based on finite number of lookahead tokens?
- **Answer.** No. Grammar is not LL(k) for any number of k

Making a grammar LL(1)

- **Problem:** can't decide which S production to apply until we see symbol after first expression
- **Left-factoring:** Factor common prefix E, add new non-terminal E' for what follows that prefix

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

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

Making a grammar LL(1)

- An LL(1) grammar does not have left recursion
- Conversion to LL(1):

1) First step: remove left recursion from grammar

$$A \rightarrow A\alpha$$
$$\mid \ \beta$$

2) Second step: left factor the grammar

$$A \rightarrow \alpha \ \beta_1$$
$$\mid \alpha \ \beta_2$$

- This procedure does not convert any CFG to LL(1)

Parsing with new grammar

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

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
$\Rightarrow (\texttt{num})\ \mathbf{E'}$	+	(num) +num
$\Rightarrow (\texttt{num}) + \mathbf{E}$	num	(num) +num
$\Rightarrow (\texttt{num}) + \texttt{num}$	$	(num) +num

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

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

Using Table

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

Partly-derived String	Lookahead	parsed part unparsed part
S	(	(num) +num
$\Rightarrow$ **E** E'	(	(num) +num
$\Rightarrow$ (**E**) E'	num	(num) +num
$\Rightarrow$ (num) **E′**	+	(num) +num
$\Rightarrow$ (num) + **E**	num	(num) +num
$\Rightarrow$ (num) + num	$	(num) +num

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

- Dollar sign $ is end of file marker

Implementation

- Table can be converted easily into a **recursive-descent parser**
- Idea: associate a procedure with each nonterminal in the grammar

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

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

Recursive-Descent Parser

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

lookahead token

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

Recursive-Descent Parser

```
void parse_E' () {
  switch(token) {
    case '+':
      token = input.read(); parse_E(); return;
    case EOF: return;
    default: throw new ParseError();
  }
}
```

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

Recursive-Descent Parser

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

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

Call Tree = Parse Tree

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

input: `( num ) + num`