Top-Down Parsing

CPSC 310 - Programming Languages

Outline

- Implementation of parsers
- Two main approaches:
  - Top-down
  - Bottom-up
- This lecture: Top-Down
  - Easier to understand and program manually
- CSC 425: Bottom-Up
  - More powerful and used by most parser generators

Introduction to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream: t_2, t_5, t_6, t_8, t_9
- The parse tree is constructed: from top to bottom and from left to right

Recursive Descent Parsing

- Consider the grammar

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

- and token stream: $int(5) * int(2)$
- Start with the top-level non-terminal E
- Try the rules for E in order

Recursive Descent Parsing

1. Try $E_0 \to T_1 + E_2$
2. Try $T_1 \to (E_3)$
   - The left parenthesis does not match the token $int(5)$
3. Try $T_1 \to int$
   - Matches, but $+$ after T_1 does not match $*$
4. Try $T_1 \to int * T_2$
   - Matches and consumes two tokens
     - Try $T_2 \to int$ matches, but $+$ after T_1 does not
     - Try $T_2 \to int * T_3$ but $+$ does not match end-of-input
5. Has exhausted the choices for T_2
   - Backtrack to choice for E_0

Recursive Descent Parsing

1. Try $E_0 \rightarrow T_1$
2. Follow same steps as before for T_1
   - and succeed with $T_1 \rightarrow int(5) * T_2$ and $T_2 \rightarrow int(2)$
   - with the following parse tree

Recursive Descent Parsing: Notes

- Easy to implement by hand
- Somewhat inefficient due to backtracking
- Does not always work . . .

When Recursive Descent Does Not Work

- Consider a production $S \to Sa$
- And the following pseudo-code implementation

```
bool S1() { return S() && term(a); }
bool S()  { return S1(); }
```

- The function call $S()$ gets into an infinite loop
- A *left-recursive grammar* has a non-terminal S and production $S \xrightarrow{+} S\alpha$ for some α
- Recursive descent does not work in such cases

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 any number of αs
- This grammar can be rewritten using right-recursion

$$\begin{aligned} S &\to \beta S' \\ S' &\to \alpha S' \mid \epsilon \end{aligned}$$

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

$$\begin{aligned} S &\to \beta_1 S' \mid \ldots \mid \beta_m S' \\ S' &\to \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \epsilon \end{aligned}$$

General Left Recursion

- The grammar

$$\begin{aligned} S &\to A\alpha \mid \delta \\ A &\to S\beta \end{aligned}$$

  is also left-recursive because

$$S \xrightarrow{+} S\beta\alpha$$

- This left-recursion can also be eliminated (see a compilers text for a general algorithm)

Summary of Recursive Descent

- Simple and general 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 eliminated by restricting the grammar

Predictive Parsers

- Like recursive descent, but the parser can "predict" which production to use by looking at the next few tokens and does not need to backtrack
- 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, there may be multiple production choices for each non-terminal and input token
- $LL(1)$ means that there is only one production for each non-terminal and input token
- Can be specified via 2D tables
  - one dimension for the current non-terminal to expand
  - one dimension for the next token
  - a table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar for arithmetic expressions

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid int \mid int * T$$

- Difficult to predict because:
  - For T two productions start with *int*
  - For E it is not clear how to predict
- A grammar must be left-factored before it is used for predictive parsing

Left-Factoring Example

- Recall the grammar for arithmetic expressions

$$\begin{aligned} E &\to T + E \mid T \\ T &\to (E) \mid int \mid int * T \end{aligned}$$

- Factor out common prefixes of productions

$$\begin{aligned} E &\to T\ X \\ X &\to +E \mid \epsilon \\ T &\to (E) \mid int\ Y \\ Y &\to *T \mid \epsilon \end{aligned}$$

$LL(1)$ Parsing Table Example

- Left-factored grammar

$$\begin{aligned} E &\to T\ X \\ X &\to +E \mid \epsilon \\ T &\to (E) \mid int\ Y \\ Y &\to *T \mid \epsilon \end{aligned}$$

- 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

- Consider the $[E, int]$ entry
    - If the current non-terminal is E and the next input is *int*, then use production $E \to T\ X$
    - This production can generate an *int* in the first place
- Consider the $[Y, +]$ entry
    - If the current non-terminal is Y and the current input is $+$, then eliminate Y
    - Y can be followed by $+$ only in a derivation in which $Y \to \epsilon$
- 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
- 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 context-free grammars

Constructing Parsing Tables

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

Computing First Sets

- Definition: $\mathit{First}(X) = \{t \mid \ \xrightarrow{*} t\alpha\} \cup \{\epsilon \mid X \xrightarrow{*} \epsilon\}$
- Algorithm sketch
  1. $\mathit{First}(t) = \{t\}$
  2. $\epsilon \in \mathit{First}(X)$ if $X \rightarrow \epsilon$ is a production
  3. $\epsilon \in \mathit{First}(X)$ if $X \rightarrow A_1 \ldots A_n$ and $\epsilon \in \mathit{First}(A_i)$ for each $1 \leq i \leq n$
  4. $\mathit{First}(\alpha) \subseteq \mathit{First}(X)$ if $X \rightarrow A_1 \ldots A_n\alpha$ and $\epsilon \in \mathit{First}(A_i)$ for each $1 \leq i \leq n$

First Sets Example

- Recall the grammar

$$\begin{aligned} E &\to T\ X \\ X &\to +E \mid \epsilon \\ T &\to (E) \mid \mathit{int}\ Y \\ Y &\to *T \mid \epsilon \end{aligned}$$

- First sets

$\mathit{First}(\ (\ ) = \{\ (\ \}$	$\mathit{First}(\ )\ ) = \{\ )\ \}$
$\mathit{First}(+) = \{+\}$	$\mathit{First}(*) = \{*\}$
$\mathit{First}(\mathit{int}) = \{\mathit{int}\}$	$\mathit{First}(T) = \{\mathit{int}, (\}$
$\mathit{First}(E) = \{\mathit{int}, (\}$	$\mathit{First}(X) = \{+, \epsilon\}$
$\mathit{First}(Y) = \{*, \epsilon\}$	

Computing Follow Sets

- Definition: $\mathit{Follow}(X) = \{t \mid \xrightarrow{*} \beta\ X\ t\ \delta\}$
- Intuition
  - If $X \to A\ B$, then $\mathit{First}(B) \subseteq \mathit{Follow}(A)$ and $\mathit{Follow}(X) \subseteq \mathit{Follow}(B)$
  - Also, if $B \xrightarrow{*} \epsilon$, then $\mathit{Follow}(X) \subseteq \mathit{Follow}(A)$
  - IF S is the start symbol, then $\$ \in \mathit{Follow}(S)$
- Algorithm sketch
  1. $\$ \in \mathit{Follow}(S)$
  2. $\mathit{First}(\beta) - \{\epsilon\} \subseteq \mathit{Follow}(X)$ for each production $A \to \alpha\ X\ \beta$
  3. $\mathit{Follow}(A) \subseteq \mathit{Follow}(X)$ for each production $A \to \alpha\ X\ \beta$ where $\epsilon \in \mathit{First}(\beta)$

Follow Sets Example

- Recall the grammar

$$E \to T\ X$$
$$X \to +E \mid \epsilon$$
$$T \to (E) \mid int\ Y$$
$$Y \to *T \mid \epsilon$$

- First sets

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

Constructing $LL(1)$ Parsing Tables

- Construct a parsing table T for context-free grammar G
- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do $T[A, t] = \alpha$
  - If $\epsilon \in First(\alpha)$, then for each $t \in Follow(A)$ do $T[A, t] = \alpha$
  - If $\epsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do $T[A, \$] = \alpha$

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 languages are not $LL(1)$
- There are tools that build $LL(1)$ tables
- For some grammars, predictive parsing is a simple parsing strategy