Top-Down Parsing

Adapted from Lecture by
Profs. Alex Aiken & George Necula
(UCB)

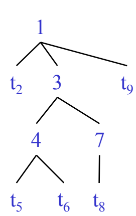
Lecture Outline

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

Intro to Top-Down Parsing

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:

  t_2 t_5 t_6 t_8 t_9



Recursive Descent Parsing

- Consider the grammar

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

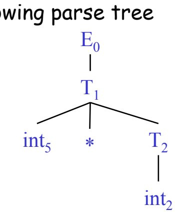
- Token stream is: int_5 * int_2
- Start with top-level non-terminal E
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int_5
- Try $T_1 \rightarrow int$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \rightarrow int * T_2$
  - This will match but + after T_1 will be unmatched
- Has exhausted the choices for T_1
  - Backtrack to choice for E_0

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \rightarrow int * T_2$ and $T_2 \rightarrow int$
  - With the following parse tree



A Recursive Descent Parser. Preliminaries

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global next point to the next token

A Recursive Descent Parser (2)

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

    `bool term(TOKEN tok) { return *next++ == tok; }`
  - A given production of S (the n^{th})

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

    `bool S() { … }`
- These functions advance next

A Recursive Descent Parser (3)

- For production $E \rightarrow T + E$
  bool E_1() { return T() && term(PLUS) && E(); }
- For production $E \rightarrow T$
  bool E_2() { return T(); }

For all productions of E (with backtracking)

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


A Recursive Descent Parser (4)

- Functions for non-terminal T

bool T_1() { return term(OPEN) && E() && term(CLOSE); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(INT); }

```
bool T() {
  TOKEN *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 previous example.
- Easy to implement by hand
- But does not always work …

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$
  bool S_1() { return S() && term(a); }
  bool S() { return S_1(); }
- S() will get into an infinite loop
- A <u>left-recursive grammar</u> has a non-terminal S
  $S \rightarrow^+ S\alpha$ 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 a number of α

$$\beta\alpha*$$

- Can rewrite using right-recursion

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

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.
- More examples on the following slides.

$$A \to Bb \mid a$$
$$B \to Aa \mid b$$

(*cf.* Gaussian Elimination)

$$A \to Bb \mid a$$
$$B \to (Bb \mid a)a \mid b$$

$$A \to Bb \mid a$$
$$B \to Bba \mid aa \mid b$$

$$A \to Bb \mid a$$
$$B \to (aa \mid b)Z \mid (aa \mid b)$$
$$Z \to baZ \mid ba$$

Example: Related to conversion to Griebach Normal Formal

$$A \to BC$$
$$B \to CA \mid b$$
$$C \to AB \mid a$$

$A \succ B \succ C$

$$A \to BC$$
$$B \to CA \mid b$$
$$C \to \underline{BCB} \mid a$$

$$C \to \underline{\underline{CACB}} \mid \underline{bCB} \mid a$$

Eliminating left recursion

$$C \to (\underline{bCB} \mid \underline{a})R$$
$$\mid \underline{bCB}\underline{a}$$
$$R \to \underline{ACBR} \mid \underline{ACB}$$

Introducing terminals as first element on RHS

$$C \to bCBR \mid aR \mid bCB \mid a$$
$$B \to bcBRA \mid aRA$$
$$\mid bCBA \mid aA \mid b$$
$$A \to bcBRAC \mid aRAC$$
$$\mid bCBAC \mid aAC \mid bC$$
$$R \to (bCBRAC \mid \ldots \mid bC)(CBR \mid CB)$$

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
  - Cf. Prolog execution strategy
- In practice, backtracking is eliminated by restricting the grammar
  - To enable "look-before-you-leap" strategy

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(k) grammars
- LR(k) grammars
  - L means "left-to-right" scan of input
  - R means "rightmost derivation"
  - k means "predict based on k tokens of lookahead"
- RL(1) grammars
  - R means "right-to-left" scan of input
- LR(0) , LR(1) grammars
- SLR(1) grammars, LALR(1) grammars

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.
- Can be specified via 2D tables.
  - One dimension for current non-terminal to expand.
  - One dimension for next token.
  - A table entry contains one production.

Predictive Parsing and Left Factoring

- Recall the grammar

  E → T + E | T

  T → int | int * T | (E)

- Hard 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 use for predictive parsing.

Left-Factoring Example

- Recall the grammar

  E → T + E | T

  T → int | int * T | (E)

- Factor out common prefixes of productions, possibly introducing ε-productions

  E → T X

  X → + E | ε

  T → (E) | int Y

  Y → * T | ε

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 $E \rightarrow T X$.
  - 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 $Y \rightarrow \varepsilon$.

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 X
  - We look at the next token t
  - And chose the production shown at [X,t]
- 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 CFG.

Constructing Parsing Tables (Cont.)

- If $A \to \alpha$,
  where in the line 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 $t \in First(\alpha)$.
- In the column of t
  if α is or derives ε and t can *follow* an A.
  - $S \to^* \beta\, A\, t\, \delta$
  - We say $t \in Follow(A)$.

Computing First Sets

Definition

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

Algorithm sketch:

1. $First(t) = \{\, t \,\}$
2. $\varepsilon \in First(X)$ if $X \to \varepsilon$ is a production
3. $\varepsilon \in First(X)$ if $X \to A_1 \dots A_n$
   - and $\varepsilon \in First(A_i)$ for $1 \le i \le n$
4. $First(\alpha) - \{\varepsilon\} \subseteq First(X)$ if $X \to A_1 \dots A_n\, \alpha$
   - and $\varepsilon \in First(A_i)$ for $1 \le i \le n$

First Sets. Example

- Recall the grammar

  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε

- First sets

  First(( ) = { (}   First(T) = {int, (}
  First( )) = {) }   First(E) = {int, (}
  First(int) = { int }   First(X) = {+, ε }
  First(+) = { + }   First(Y) = {*, ε }
  First(*) = { * }

Computing Follow Sets

- Definition:

  Follow(X) = { t | S →* β X t δ }

- Intuition
  - If X → A B then First(B) ⊆ Follow(A) and Follow(X) ⊆ Follow(B)
  - Also if B →* ε then Follow(X) ⊆ Follow(A)
  - If S is the start symbol then $ ∈ Follow(S)

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $ ∈ Follow(S)
2. First(β) - {ε} ⊆ Follow(X)
   - For each production A → α X β
3. Follow(A) ⊆ Follow(X)
   - For each production A → α X β where ε ∈ First(β)

Follow Sets. Example

- Recall the grammar

  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε

- Follow sets

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

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 $t \in First(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$ 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 language grammars are not LL(1). (Cf. Wirth's Pascal Compiler)
- There are tools that build LL(1) tables.