

Top-Down Parsing

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

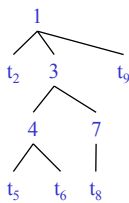
Lecture Outline

- 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
 - $E \rightarrow T + E \mid T$
 - $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$
- Token stream is: $\text{int}_5 * \text{int}_2$
- 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

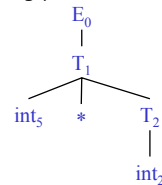
CS780(Prasad)

L101TDP

5

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



CS780(Prasad)

L101TDP

6

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

CS780(Prasad)

L101TDP

7

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 Sn() { ... }
```
 - Any production of S:

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

CS780(Prasad)

L101TDP

8

A Recursive Descent Parser (3)

- For production $E \rightarrow T + E$
`bool E1() { return T() && term(PLUS) && E(); }`
- For production $E \rightarrow T$
`bool E2() { 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 T1() { return term(OPEN) && E() && term(CLOSE); }`
`bool T2() { return term(INT) && term(TIMES) && T(); }`
`bool T3() { 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 S1() { return S() && term(a); }`
`bool S() { return S1(); }`
- `S()` will get into an infinite loop
- A left-recursive grammar 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

$$S \rightarrow S\alpha \mid \beta$$
- S generates all strings starting with a β and followed by a number of α

$\beta\alpha^*$

- Can rewrite using right-recursion

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \rightarrow S\alpha_1 \mid \dots \mid S\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from S start with one of β_1, \dots, β_m and continue with several instances of $\alpha_1, \dots, \alpha_n$

- Rewrite as

$$S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A\alpha \mid \delta$$

$$A \rightarrow S\beta$$

is also left-recursive because

$$S \rightarrow^* S\beta\alpha$$

- This left-recursion can also be eliminated.
- More examples on the following slides.

$$A \rightarrow Bb \mid a$$

$$B \rightarrow Aa \mid b$$

(cf. Gaussian Elimination)

$$A \rightarrow Bb \mid a$$

$$B \rightarrow (Bb \mid a)a \mid b$$



$$A \rightarrow Bb \mid a$$

$$B \rightarrow Bba \mid aa \mid b$$



$$A \rightarrow Bb \mid a$$

$$B \rightarrow (aa \mid b)Z \mid (aa \mid b)$$

$$Z \rightarrow baZ \mid ba$$

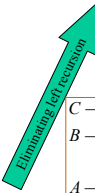
Example: Related to conversion to Griebach Normal Form

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

$A > B > C$

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

$C \rightarrow \underline{C} \underline{A} \underline{C} \underline{B} \mid \underline{b} \underline{C} \underline{B} \mid a$



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

Introducing terminals as first element on RHS

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

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

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- 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

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

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

$$E \rightarrow TX$$
$$X \rightarrow +E \mid \epsilon$$
$$T \rightarrow (E) \mid \text{int} Y$$
$$Y \rightarrow *T \mid \epsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

$$E \rightarrow TX \qquad X \rightarrow +E \mid \epsilon$$
$$T \rightarrow (E) \mid \text{int} Y \qquad Y \rightarrow *T \mid \epsilon$$

- The LL(1) parsing table:

	int	*	+	()	\$
E	TX			TX		
X			+E		ϵ	ϵ
T	int Y			(E)		
Y		*T	ϵ		ϵ	ϵ

LL(1) Parsing Table Example (Cont.)

- Consider the $[E, \text{int}]$ entry
 - "When current non-terminal is E and next input is int , use production $E \rightarrow TX$."
 - 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 $Y \rightarrow \epsilon$.

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 \dagger
 - And chose the production shown at $[X, \dagger]$
- 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

CS780(Prasad)

L101TDP

29

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.

CS780(Prasad)

L101TDP

30

Constructing Parsing Tables (Cont.)

- If $A \rightarrow \alpha$,
where in the line of A do we place α ?
- In the column of t
where t can *start* a string derived from α .
 - $\alpha \rightarrow^* t\beta$
 - We say that $t \in \text{First}(\alpha)$.
- In the column of t
if α is or derives ϵ and t can *follow* an A .
 - $S \rightarrow^* \beta A t \delta$
 - We say $t \in \text{Follow}(A)$.

CS780(Prasad)

L101TDP

31

Computing First Sets

Definition

$$\text{First}(X) = \{ t \mid X \rightarrow^* t\alpha \} \cup \{ \epsilon \mid X \rightarrow^* \epsilon \}$$

Algorithm sketch:

1. $\text{First}(t) = \{ t \}$
2. $\epsilon \in \text{First}(X)$ if $X \rightarrow \epsilon$ is a production
3. $\epsilon \in \text{First}(X)$ if $X \rightarrow A_1 \dots A_n$
- and $\epsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$
4. $\text{First}(\alpha) - \{ \epsilon \} \subseteq \text{First}(X)$ if $X \rightarrow A_1 \dots A_n \alpha$
- and $\epsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$

CS780(Prasad)

L101TDP

32

First Sets. Example

- Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \varepsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \varepsilon \end{array}$$

- First sets

$$\begin{array}{ll} \text{First}(() = \{ (\} & \text{First}(T) = \{ \text{int}, (\} \\ \text{First}()) = \{) \} & \text{First}(E) = \{ \text{int}, (\} \\ \text{First}(\text{int}) = \{ \text{int} \} & \text{First}(X) = \{ +, \varepsilon \} \\ \text{First}(+) = \{ + \} & \text{First}(Y) = \{ *, \varepsilon \} \\ \text{First}(*) = \{ * \} & \end{array}$$

Computing Follow Sets

- Definition:

$$\text{Follow}(X) = \{ \dagger \mid S \rightarrow^* \beta X \dagger \delta \}$$

- Intuition

- If $X \rightarrow A B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
- Also if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
- If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

- $\$ \in \text{Follow}(S)$
- $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$
- $\text{Follow}(A) \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets. Example

- Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \varepsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \varepsilon \end{array}$$

- Follow sets

$$\begin{array}{ll} \text{Follow}(+) = \{ \text{int}, (\} & \text{Follow}(*) = \{ \text{int}, (\} \\ \text{Follow}(() = \{ \text{int}, (\} & \text{Follow}(E) = \{ \}, \$ \} \\ \text{Follow}(X) = \{ \$,) \} & \text{Follow}(T) = \{ +,) , \$ \} \\ \text{Follow}()) = \{ +,) , \$ \} & \text{Follow}(Y) = \{ +,) , \$ \} \\ \text{Follow}(\text{int}) = \{ *, +,) , \$ \} & \end{array}$$

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $t \in \text{First}(\alpha)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ 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.