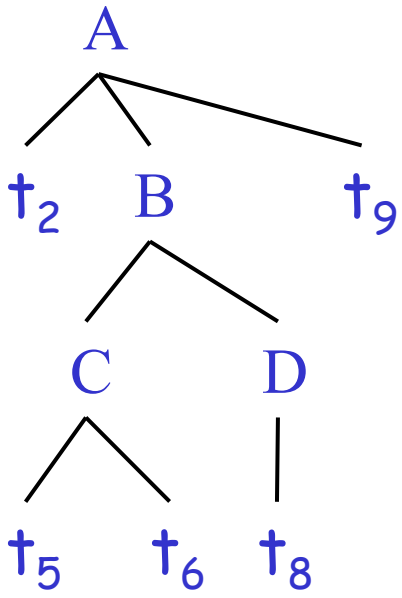
Introduction to Top-Down Parsing

- 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 the top
  - From left to right



Recursive Descent Parsing: Example

- Consider the grammar

  $E \rightarrow T + E \mid T$

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

- 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.)

Token stream: int_5 * int_2

- 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 and will consume the two tokens.
    - Try $T_2 \rightarrow int$ (matches) but + after T_1 will be unmatched
    - Try $T_2 \rightarrow int * T_3$ but * does not match with end-of-input
- Has exhausted the choices for T_1
  - Backtrack to choice for E_0

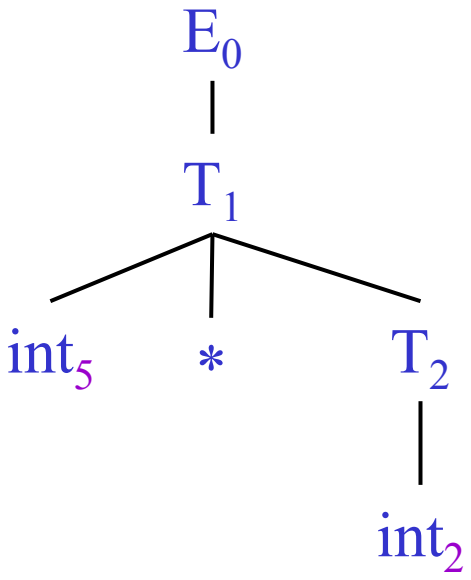
$E \rightarrow T + E \mid T$

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

Recursive Descent Parsing: Example (Cont.)

Token stream: int_5 * int_2

- Try $E_0 \rightarrow T_1$
- 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



$E \rightarrow T + E \mid T$

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

Recursive Descent Parsing: Notes

- Easy to implement by hand
- Somewhat inefficient (due to backtracking)
- 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 <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
  - It goes into an infinite loop

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 any number of α's

- The grammar can be rewritten using right-recursion

$$S \rightarrow \beta\ S'$$

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

More Elimination of Left-Recursion

- In general

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

- 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

$$S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$$
$$S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$$

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

[See a Compilers book 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 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(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of productions
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- 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 for arithmetic expressions

  $E \rightarrow T + E \mid T$

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

- 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 it is used for predictive parsing

Left-Factoring Example

- Recall the grammar

  $E \rightarrow T + E \mid T$

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

- Factor out common prefixes of productions

  $E \rightarrow T X$

  $X \rightarrow + E \mid \varepsilon$

  $T \rightarrow (E) \mid int Y$

  $Y \rightarrow * T \mid \varepsilon$

- This grammar is equivalent to the original one

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table ($ is the end marker):

	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 a
  - And chose the production shown at [X,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

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

Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- where no table entry is multiply defined

- Once we have the table
  - The parsing is simple and fast
  - No backtracking is necessary

- We want to generate parsing tables from CFG

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 $t \in First(\alpha)$
- In the column of t if α is ε and t can follow an A
  - $S \rightarrow^* \beta\ A\ t\ \delta$
  - We say $t \in Follow(A)$

Computing First Sets

Definition

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

Algorithm sketch

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

Computing First Sets

Definition

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

More constructive algorithm

1. First(t) = { t }
2. For all productions $X \rightarrow A_1 \ldots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_1)$.
   - Add $First(A_2) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_2)$.
   - ...
   - Add $First(A_n) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_n)$.
   - Add $\{\varepsilon\}$ to First(X).

First Sets: Example

- Recall the grammar

  $E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- First sets

  First(( ) = { (}

  First( )) = {) }

  First(int) = { int }

  First(+) = { + }

  First(*) = { * }

  First(T) = { int, (}

  First(E) = { int, (}

  First(X) = { +, ε }

  First(Y) = { *, ε }

Computing Follow Sets

- **Definition**

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

- **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

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$

   For each production $A \to \alpha\ X\ \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$

   For each production $A \to \alpha\ X\ \beta$ where $\varepsilon \in \text{First}(\beta)$

Computing Follow Sets (Cont.)

Definition

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

More constructive algorithm

1. First compute the First sets for all non-terminals
2. If S is the start symbol, add \$ to Follow(S)
3. For all productions $Y \rightarrow \ldots X\ A_1 \ldots A_n$
   - Add First(A_1) – {ε} to Follow(X). Stop if $\varepsilon \notin$ First(A_1).
   - Add First(A_2) – {ε} to Follow(X). Stop if $\varepsilon \notin$ First(A_2).
   - …
   - Add First(A_n) – {ε} to Follow(X). Stop if $\varepsilon \notin$ First(A_n).
   - Add Follow(Y) to Follow(X).

Follow Sets: Example

- Recall the grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- 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 $A \rightarrow \alpha$ 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)
- There are tools that build LL(1) tables

Review

- For some grammars there is a simple parsing strategy

  Predictive parsing (LL(1))

- Next time: a more powerful parsing strategy