Lecture 5
Top-Down Parsing

Review

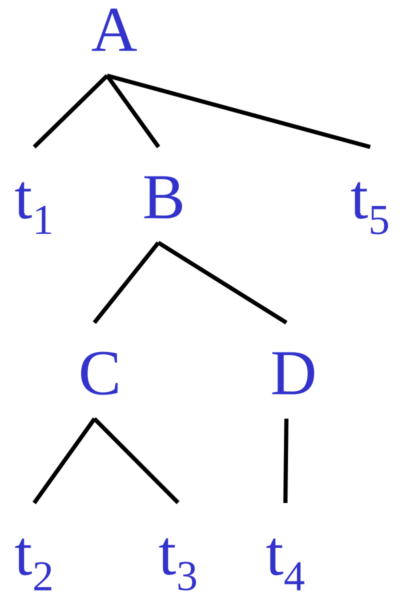
- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
  - … and will build a parse tree
  - … which we convert to an AST
  - … and pass on to the rest of the compiler

Outline

- Recursive descent parsing
- Top-down Parsing

How It's Done I: 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:

$$t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5$$

- … As for leftmost derivation

Recursive Descent Parsing

- Consider the grammar

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

- Token stream is: int * int

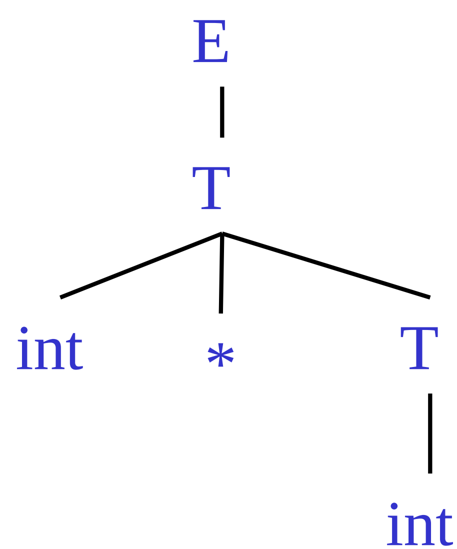
- Start with top-level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing. Example int * int

- Start with start symbol E
- Try $E \rightarrow T + E$ T + E
- Then try a rule for $T \rightarrow (E)$ (E) + E
  - But (≠ input int; backtrack to T + E
- Try $T \rightarrow$ int . Token matches. int + E
  - But + ≠ input *; back to T + E
- Try $T \rightarrow$ int * T int*T+E
  - But (skipping some steps) + can't be matched
- Must backtrack to E

Recursive Descent Parsing. Example int * int

- Try $E \to T$
- Follow same steps as before for T
  - And succeed with $T \to$ int * T and $T \to$ 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 (Limited) 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; }`
  - The nth production of S:

    bool S_n() { … }
  - Try all productions of S:

    `bool S() { … }`

A (Limited) Recursive Descent Parser (3)

- For production E → T

  bool E_1() { return T(); }

- For production E → T + E

  bool E2() { return T() && term(PLUS) && E(); }

- For all productions of E (with backtracking)

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

A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

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 the example parse
- Easy to implement by hand
  - But not completely general
  - Cannot backtrack once a production is successful
  - Works for grammars where at most one production can succeed for a non-terminal

Recursive Descent Parsing of $t_1\ t_2\ \ldots\ t_n$

- At a given moment, have sentential form that looks like this: $t_1\ t_2\ \ldots\ t_k\ A\ \ldots,\ 0 \le k \le n$
- Initially, $k=0$ and $A\ldots$ is just start symbol
- Try a production for A: if $A \to BC$ is a production, the new form is $t_1\ t_2\ \ldots\ t_k\ B\ C\ \ldots$
- Backtrack when leading terminals aren't prefix of $t_1\ t_2\ \ldots\ t_n$ and try another production
- Stop when no more non-terminals and terminals all matched (accept) or no more productions left (reject)

When Depth-First Doesn't Work Well

- Consider productions $S \to S\ a$

  bool S_1() { return S() && term(a); }

  bool S() { return S_1(); }
- S() goes into an infinite loop

- A *left-recursive grammar* has a non-terminal S

  $S \to^{+} S\alpha$ 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's

- Can rewrite using right-recursion

$$S \to \beta\ S'$$

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

Elimination of left Recursion. Example

- Consider the grammar

$$S \to 1 \mid S\,0 \qquad (\ \beta = 1 \text{ and } \alpha = 0\ )$$

can be rewritten as

$$S \to 1\,S'$$

$$S' \to 0\,S' \mid \varepsilon$$

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

  $$S \to A\,\alpha \mid \delta \qquad (1)$$
  $$A \to S\,\beta \qquad (2)$$

  is also left-recursive because

  $$S \to^{+} S\,\beta\,\alpha$$

- This left recursion can also be eliminated by first substituting (2) into (1)
- See Dragon Book for general algorithm (section 4.3)
- But personally, I'd just do this by hand.

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) vs. Recursive Descent

- In recursive descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1),
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - The next input symbol is t
    - There is a unique production $A \to \alpha$ to use
      - Or no production to use (an error state) •
- LL(1) is a recursive descent variant without backtracking

But First: Left Factoring

- Recall the grammar

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

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to *left-factor* the grammar

Left-Factoring Example

- Starting with the grammar

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

- Factor out common prefixes of productions

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

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 a special end marker):

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

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - Means "When current non-terminal is E and next input is int, use production $E \rightarrow T X$"
  - This can generate an int in the first position
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if $Y \rightarrow \varepsilon$
  - We'll see later why this is right

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 the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack

LL(1) Parsing Algorithm

initialize stack = <S,$>
repeat
   case stack of
      <X, rest> : if T[X,next()] == $Y_1 \ldots Y_n$:
                              stack ← <$Y_1 \ldots Y_n$ rest>;
                           else: error ();
      <t, rest> : scan (t); stack ← <rest>;
until stack == < >

LL(1) Parsing Example

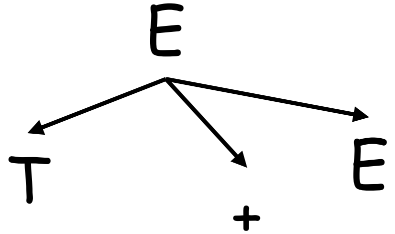
Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables

- LL(1) languages are those definable by a parsing table for the LL(1) algorithm such that no table entry is multiply defined
- Once we have the table
  - Can create table-driver or recursive-descent parser
  - The parsing algorithms are simple and fast
  - No backtracking is necessary
- We want to generate parsing tables from CFG

Predicting Productions I

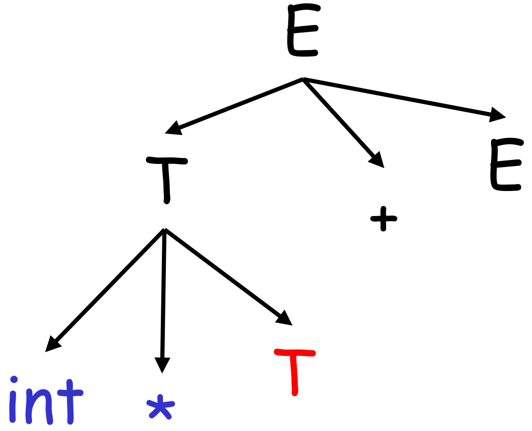
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Predicting Productions II

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

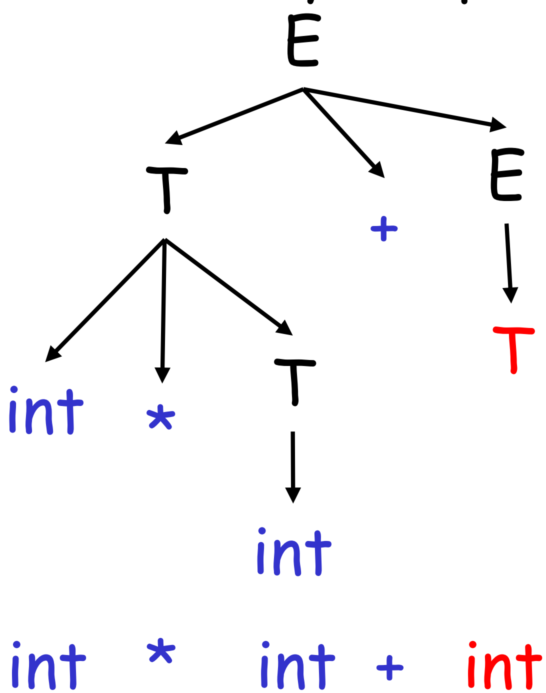


int * int + int

- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Predicting Productions III

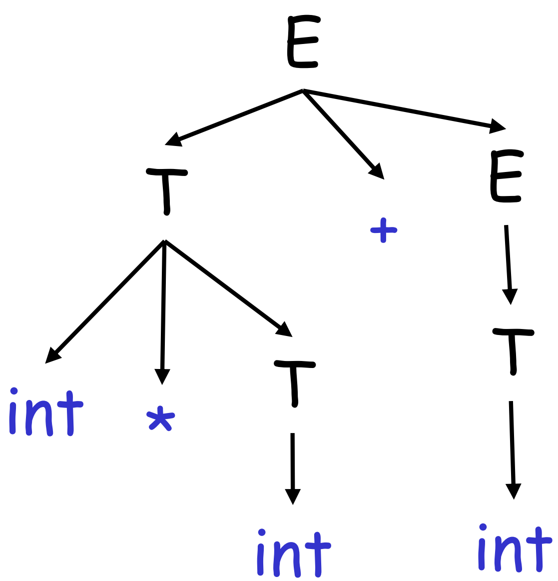
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$ (A=T, $\gamma=\varepsilon$)
  - β contains only terminals
  - γ contains any symbols
  - The input string is $\beta b \delta$ (b=int)
  - So $A\gamma$ must derive $b\delta$

Predicting Productions IV

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

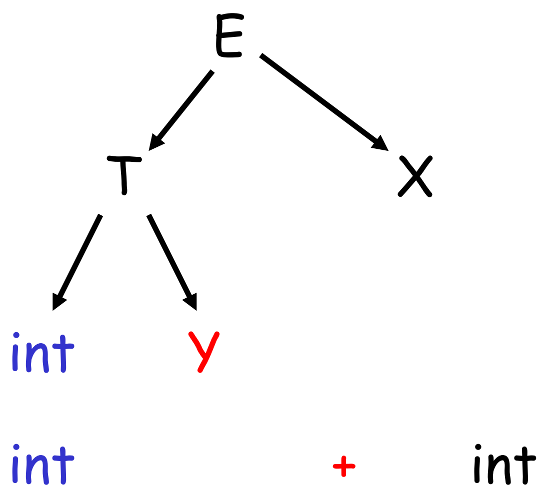
- So choose production for T that can eventually derive something that starts with int

Predicting Productions V

- Go back to previous grammar, with

$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$

$T \to (\,E\,) \mid \text{int}\;Y \qquad\qquad Y \to *\;T \mid \varepsilon$



Here, YX must match + int

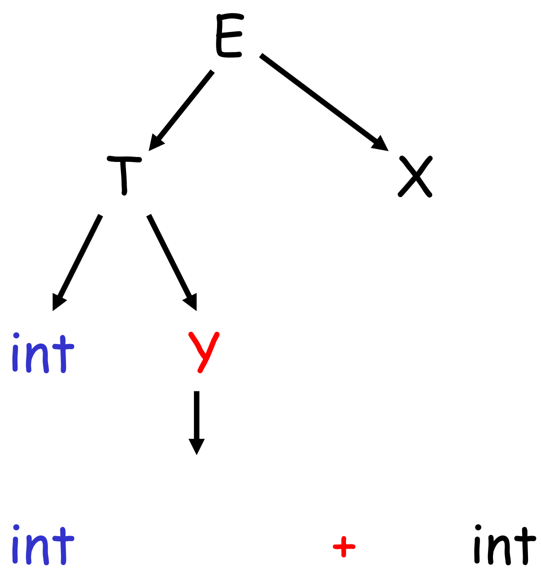
Since + int doesn't start with *, can't use $Y \to *\;T$

Predicting Productions V

- Go back to previous grammar, with

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

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



But + can *follow* Y, so we choose $Y \rightarrow \varepsilon$

FIRST and FOLLOW

- To summarize, if we are trying to predict how to expand A with b the next token, then either:
  - b belongs to an expansion of A
    - Any $A \rightarrow \alpha$ can be used if b can start a string derived from α
    - In this case we say that $b \in First(\alpha)$
  - or b does not belong to an expansion of A, A has an expansion that derives ε, and b belongs to something that can follow A (so $S \rightarrow^* \beta A b \omega$)
    - We say that $b \in Follow(A)$ in this case.

Summary of Definitions

- For $b \in T$, the set of terminals; α a sequence of terminal & non-terminal symbols, S the start symbol, $A \in N$, the set of non-terminals:
- $FIRST(\alpha) \subseteq T \cup \{ \varepsilon \}$

  $b \in FIRST(\alpha)$ iff $\alpha \rightarrow^* b \ldots$

  $\varepsilon \in FIRST(\alpha)$ iff $\alpha \rightarrow^* \varepsilon$
- $FOLLOW(A) \subseteq T$

  $b \in FOLLOW(A)$ iff $S \rightarrow^* \ldots A\ b \ldots$

Computing First Sets

Definition $First(X) = \{ b \mid X \to^* b\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$, X any grammar symbol.

1. $First(b) = \{ b \}$ for b any terminal symbol
2. For all productions $X \to A_1 \ldots A_n$
   - 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 ε to $First(X)$
3. Repeat 2 until nothing changes ("compute fixed point")

Computing First Sets, Contd.

- That takes care of single-symbol case.
- In general:

$$\begin{aligned}
&\mathrm{FIRST}(X_1\ X_2 \ldots X_k) = \\
&\qquad \mathrm{FIRST}(X_1) \\
&\quad \cup\ \mathrm{FIRST}(X_2)\ \text{ if } \varepsilon \in \mathrm{FIRST}(X_1) \\
&\quad \cup\ \ldots \\
&\quad \cup\ \mathrm{FIRST}(X_k)\ \text{ if } \varepsilon \in \mathrm{FIRST}(X_1 X_2 \ldots X_{k-1}) \\
&\quad -\{\ \varepsilon\ \}\ \text{unless } \varepsilon \in \mathrm{FIRST}(X_i) \quad \forall\ i
\end{aligned}$$

First Sets. Example

- For the grammar

$E \to T\ X$

$T \to (\ E\ )\ |\ \text{int}\ Y$

$X \to +\ E\ |\ \varepsilon$

$Y \to *\ T\ |\ \varepsilon$

- 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 $\quad$ $Follow(X) = \{ b \mid S \to^* \beta X b \omega \}$

1. Compute the First sets for all non-terminals first
2. Add $\$$ to Follow(S) (if S is the start non-terminal)
3. For all productions $Y \to \ldots X A_1 \ldots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_n)$
   - Add Follow(Y) to Follow(X)
4. Repeat 3 until nothing changes.

Follow Sets. Example

- For the grammar

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

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

- Follow sets

Follow(E) = {), $}

Follow(X) = {$,) }

Follow(Y) = {+,) , $}

Follow(T) = {+,) , $}

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 $b \in First(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\alpha \to^* \varepsilon$, for each $b \in Follow(A)$ do
    - $T[A, b] = \alpha$

LL(1) Parsing Table Example

$E \to T\ X$

$T \to (\ E\ )\ |\ \text{int}\ Y$

$X \to +\ E\ |\ \varepsilon$

$Y \to *\ T\ |\ \varepsilon$

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

Follow(E) = {), $}

Follow(X) = {$,) }

Follow(Y) = {+,) , $}

Follow(T) = {+,) , $}

First(T) = {int, (}

First(E) = {int, (}

First(X) = {+, ε }

First(Y) = {*, ε }

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1). This happens
  - 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))
  - Once you build the LL(1) table (or know the FIRST and FOLLOW sets), you can write the parser by hand: recursive descent.

- Next: a more powerful parsing strategy for grammars that are not LL(1)

Assignment

Consider the following CFG, where the set of terminals is $\Sigma = \{a, b, \#, \%, !\}$:

$S \rightarrow \%aT \mid U!$

$T \rightarrow aS \mid baT \mid \varepsilon$

$U \rightarrow \#aTU \mid \varepsilon$

(a) Construct the FIRST sets for each of the nonterminals.

(b) Construct the FOLLOW sets for each of the nonterminals.

(c) Construct the LL(1) parsing table for the grammar.

(d) Show the sequence of stack, input and action configurations that occur during an LL(1) parse of the string "#abaa%aba!".