Abstract Syntax Trees & Top-Down Parsing
Review of Parsing

• Given a language $L(G)$, a parser consumes a sequence of tokens $s$ and produces a parse tree

• Issues:
  - How do we recognize that $s \in L(G)$?
  - A parse tree of $s$ describes how $s \in L(G)$
  - Ambiguity: more than one parse tree (possible interpretation) for some string $s$
  - Error: no parse tree for some string $s$
  - How do we construct the parse tree?
Abstract Syntax Trees

• So far, a parser traces the derivation of a sequence of tokens
• The rest of the compiler needs a structural representation of the program
• **Abstract syntax trees**
  - Like parse trees but ignore some details
  - Abbreviated as AST
Abstract Syntax Trees (Cont.)

• Consider the grammar
  \[ E \rightarrow \text{int} \mid (E) \mid E + E \]
• And the string
  \[ 5 + (2 + 3) \]
• After lexical analysis (a list of tokens)
  \[ \text{int}_5 \ ' + ' \ (\ ' \text{int}_2 \ ' + ' \ \text{int}_3 \ ' ) \]
• During parsing we build a parse tree …
Example of Parse Tree

- Traces the operation of the parser
- Captures the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes
Example of Abstract Syntax Tree

- Also captures the nesting structure
- But abstracts from the concrete syntax
  \[ \leftrightarrow \text{more compact and easier to use} \]
- An important data structure in a compiler
Semantic Actions

• This is what we’ll use to construct ASTs

• Each grammar symbol may have attributes
  - An attribute is a property of a programming language construct
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer

• Each production may have an action
  - Written as: \[ X \rightarrow Y_1 \ldots Y_n \{ \text{action} \} \]
  - That can refer to or compute symbol attributes
Semantic Actions: An Example

• Consider the grammar
  \[ E \rightarrow \text{int} \mid E + E \mid ( E ) \]

• For each symbol \( X \) define an attribute \( X.\text{val} \)
  - For terminals, \( \text{val} \) is the associated lexeme
  - For non-terminals, \( \text{val} \) is the expression’s value
    (which is computed from values of subexpressions)

• We annotate the grammar with actions:
  \[
  E \rightarrow \text{int} \quad \{ \text{E.val = int.val} \} \\
  | \quad E_1 + E_2 \quad \{ \text{E.val = E}_1.\text{val} + E_2.\text{val} \} \\
  | \quad ( E_1 ) \quad \{ \text{E.val = E}_1.\text{val} \}
  \]
Semantic Actions: An Example (Cont.)

- **String:** \(5 + (2 + 3)\)
- **Tokens:** `int5 '+' '(' int2 '+' int3 ')'`

<table>
<thead>
<tr>
<th>Productions</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E \rightarrow E_1 + E_2)</td>
<td>(E).val = (E_1).val + (E_2).val</td>
</tr>
<tr>
<td>(E_1 \rightarrow \text{int}_5)</td>
<td>(E_1).val = \text{int}_5).val = 5</td>
</tr>
<tr>
<td>(E_2 \rightarrow (E_3))</td>
<td>(E_2).val = (E_3).val</td>
</tr>
<tr>
<td>(E_3 \rightarrow E_4 + E_5)</td>
<td>(E_3).val = (E_4).val + (E_5).val</td>
</tr>
<tr>
<td>(E_4 \rightarrow \text{int}_2)</td>
<td>(E_4).val = \text{int}_2).val = 2</td>
</tr>
<tr>
<td>(E_5 \rightarrow \text{int}_3)</td>
<td>(E_5).val = \text{int}_3).val = 3</td>
</tr>
</tbody>
</table>
Semantic Actions: Dependencies

Semantic actions specify a system of equations
- Order of executing the actions is not specified

• Example:
  \[ E_3.\text{val} = E_4.\text{val} + E_5.\text{val} \]
  - Must compute \( E_4.\text{val} \) and \( E_5.\text{val} \) before \( E_3.\text{val} \)
  - We say that \( E_3.\text{val} \) depends on \( E_4.\text{val} \) and \( E_5.\text{val} \)

• The parser must find the order of evaluation
Dependency Graph

- Each node labeled with a non-terminal E has one slot for its `val` attribute
- Note the dependencies
Evaluating Attributes

• An attribute must be computed after all its successors in the dependency graph have been computed
  - In the previous example attributes can be computed bottom-up

• Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal
Semantic Actions: Notes (Cont.)

• **Synthesized** attributes
  - Calculated from attributes of descendents in the parse tree
  - \texttt{E.val} is a synthesized attribute
  - Can always be calculated in a bottom-up order

• **Grammars with only synthesized attributes** are called \textit{S-attributed} grammars
  - Most frequent kinds of grammars
Inherited Attributes

• Another kind of attributes
• Calculated from attributes of the parent node(s) and/or siblings in the parse tree

• Example: a line calculator
A Line Calculator

• Each line contains an expression
  \[ E \rightarrow \text{int} \mid E + E \]

• Each line is terminated with the = sign
  \[ L \rightarrow E = \mid + E = \]

• In the second form, the value of evaluation of the previous line is used as starting value

• A program is a sequence of lines
  \[ P \rightarrow \varepsilon \mid P \ L \]
Attributes for the Line Calculator

- Each $E$ has a synthesized attribute $val$
  - Calculated as before
- Each $L$ has a synthesized attribute $val$
  
  $L \rightarrow E =$ \{ $L.val = E.val$ \}
  
  $| + E =$ \{ $L.val = E.val + L.prev$ \}

- We need the value of the previous line
- We use an inherited attribute $L.prev$
Attributes for the Line Calculator (Cont.)

• Each $P$ has a synthesized attribute $\text{val}$
  - The value of its last line
    $$ \begin{align*}
    P \rightarrow & \epsilon \quad \{ P.\text{val} = 0 \} \\
    | \quad P_1 L & \quad \{ P.\text{val} = L.\text{val} ; \\
                     & \quad L.\text{prev} = P_1.\text{val} \}
    \end{align*} $$

• Each $L$ has an inherited attribute $\text{prev}$
  - $L.\text{prev}$ is inherited from sibling $P_1.\text{val}$

• Example ...
Example of Inherited Attributes

- **val** synthesized

- **prev** inherited

- All can be computed in depth-first order

```
P + L = E4
```

```
P + E3 + =
```

```
P + E4 + =
```

```
P + E5 + =
```

```
P + 0 + =
```

```
P + int2 + =
```

```
P + int3 + =
```

```
P + 2 + =
```

```
P + 3 + =
```

```
P + int2 + =
```

```
P + int3 + =
```

```
P + 2 + =
```

```
P + 3 + =
```

```
P + int2 + =
```

```
P + int3 + =
```

```
P + 2 + =
```

```
P + 3 + =
```
Semantic Actions: Notes (Cont.)

• Semantic actions can be used to build ASTs

• And many other things as well
  – Also used for type checking, code generation, ...

• Process is called syntax-directed translation
  – Substantial generalization over CFGs
Constructing an AST

- We first define the AST data type
- Consider an abstract tree type with two constructors:

\[
\text{mkleaf}(n) = \begin{cases} 
\text{n} 
\end{cases}
\]

\[
\text{mkplus}(\begin{cases} 
\end{cases} , \end{cases}) = \begin{cases} 
\begin{array}{c}
\text{PLUS} \\
\downarrow \\
T_1 \\
\end{array} & \begin{array}{c}
\downarrow \\
T_2 \\
\end{array}
\end{cases}
\]
Constructing a Parse Tree

• We define a synthesized attribute \( \text{ast} \)
  - Values of \( \text{ast} \) values are ASTs
  - We assume that \( \text{int.lexval} \) is the value of the integer lexeme
  - Computed using semantic actions

\[
E \rightarrow \text{int} \quad \{ \text{E.ast} = \text{mkleaf(\text{int.lexval})} \}
\]
\[
| \quad E_1 + E_2 \quad \{ \text{E.ast} = \text{mkplus(E}_1\text{.ast, E}_2\text{.ast)} \}
\]
\[
| \quad (E_1) \quad \{ \text{E.ast} = E_1\text{.ast} \}
\]
Parse Tree Example

- Consider the string `int_5 + ( int_2 + int_3 )`
- A bottom-up evaluation of the ast attribute:
  \[ E.\text{ast} = \text{mkplus}(\text{mkleaf}(5), \text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3))) \]
Review of Abstract Syntax Trees

• 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

• Next two & a half lectures:
  - How do we answer $s \in L(G)$ and build a parse tree?
• After that: from AST to assembly language
Second-Half of Lecture 5: Outline

• Implementation of parsers
• Two approaches
  - Top-down
  - Bottom-up
• Today: Top-Down
  - Easier to understand and program manually
• Then: 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 the top
  - From left to right
Recursive Descent Parsing

- Consider the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]
- Token stream is: \( \text{int}_5 \ast \text{int}_2 \)
- 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 \( \text{int}_5 \)
- Try \( T_1 \rightarrow \text{int} \). Token matches.
  - But \( + \) after \( T_1 \) does not match input token \( * \)
- Try \( T_1 \rightarrow \text{int} * T_2 \)
  - This will match but \( + \) after \( T_1 \) will be unmatched
- Has exhausted the choices for \( T_1 \)
  - Backtrack to choice for \( E_0 \)

Token stream: \( \text{int}_5 * \text{int}_2 \)

\[
E \rightarrow T + E \mid T \\
T \rightarrow (E) \mid \text{int} \mid \text{int} * T
\]
Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for $T_1$
  - And succeed with $T_1 \rightarrow \text{int}_5 \ast T_2$ and $T_2 \rightarrow \text{int}_2$
  - With the following parse tree

```
E_0
  |
  T_1
  |
int_5 * T_2
```

Token stream: int5 * int2

```
E \rightarrow T + E \mid T
T \rightarrow (E) \mid \text{int} \mid \text{int} \ast 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 left-recursive grammar has a non-terminal $S$
  
  $$S \rightarrow^* S\alpha \text{ for some } \alpha$$

• 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 \( \beta \) and followed by any number of \( \alpha \)'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

\[ S \rightarrow \beta_1 S' \mid \ldots \mid \beta_m S' \]
\[ S' \rightarrow \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \epsilon \]
General Left Recursion

- The grammar

\[
S \rightarrow A \alpha | \delta \\
A \rightarrow S \beta
\]

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 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 for arithmetic expressions
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow (E) \mid \text{int} \mid \text{int} \ast T \]

- Hard to predict because
  - For \( T \) two productions start with \text{int} \n  - 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
  \[ E \to T + E \mid T \]
  \[ T \to (E) \mid \text{int} \mid \text{int} \ast T \]

• Factor out common prefixes of productions
  \[ E \to T X \]
  \[ X \to + E \mid \varepsilon \]
  \[ T \to (E) \mid \text{int} Y \]
  \[ Y \to \ast T \mid \varepsilon \]
**LL(1) Parsing Table Example**

- **Left-factored grammar**

  \[
  E \rightarrow T \cdot X \\
  T \rightarrow (E) \mid \text{int } Y \\
  X \rightarrow + E \mid \varepsilon \\
  Y \rightarrow \ast T \mid \varepsilon
  \]

- **The LL(1) parsing table:**

  \[
  \begin{array}{|c|c|c|c|c|c|}
  \hline
  & \text{int} & \ast & + & ( & ) & $ \\
  \hline
  E & T \cdot X & T \cdot X & & & & \\
  \hline
  X & & + E & \varepsilon & \varepsilon & & \\
  \hline
  T & \text{int } Y & & (E) & & & \\
  \hline
  Y & & \ast T & \varepsilon & \varepsilon & \varepsilon & \\
  \hline
  \end{array}
  \]
LL(1) Parsing Table Example (Cont.)

- **Consider the [E, int] entry**
  - “When current non-terminal is $E$ and next input is int, use production $E \rightarrow T \times$
  - 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 $S$
  - 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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>TX</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

Input Productions:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int</td>
</tr>
<tr>
<td>X</td>
<td>int</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>int</td>
</tr>
<tr>
<td>*</td>
<td>int</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

Production Table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+E</td>
<td>ε</td>
<td>ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $A \rightarrow \alpha$, where in the line of $A$ we place $\alpha$?
• In the column of $t$ where $t$ can start a string derived from $\alpha$
  - $\alpha \rightarrow^* t \beta$
  - We say that $t \in \text{First}(\alpha)$
• In the column of $t$ if $\alpha$ is $\epsilon$ and $t$ can follow an $A$
  - $S \rightarrow^* \beta A t \delta$
  - We say $t \in \text{Follow}(A)$
Computing First Sets

**Definition**

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

**Algorithm sketch**

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

• Recall the grammar

\[
\begin{align*}
E & \rightarrow T X \\
T & \rightarrow ( E ) | \text{int } Y \\
X & \rightarrow + E | \varepsilon \\
Y & \rightarrow * T | \varepsilon
\end{align*}
\]

• First sets

\[
\begin{align*}
\text{First( ( ) )} & = \{ ( ) \} \\
\text{First( + ) } & = \{ + \} \\
\text{First( int) } & = \{ \text{int} \} \\
\text{First( T ) } & = \{ \text{int}, ( ) \} \\
\text{First( E ) } & = \{ \text{int}, ( ) \} \\
\text{First( X ) } & = \{ +, \varepsilon \} \\
\text{First( Y ) } & = \{ *, \varepsilon \}
\end{align*}
\]
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 $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$
Follow Sets: Example

• Recall the grammar

\[ E \rightarrow T \ X \]
\[ T \rightarrow (\ E\ ) \mid \text{int} \ Y \]

\[ X \rightarrow + \ E \mid \varepsilon \]
\[ Y \rightarrow * \ T \mid \varepsilon \]

• Follow sets

\[ \text{Follow}(\ +\ ) = \{\ \text{int}, (\ )\} \]
\[ \text{Follow}(\ *\ ) = \{\ \text{int}, (\ )\} \]
\[ \text{Follow}(\ (\ ) = \{\ \text{int}, (\ )\} \]
\[ \text{Follow}(\ (\ X\ )) = \{\ , \}$\} \]
\[ \text{Follow}(\ (\ )) = \{\ , \}$\} \]
\[ \text{Follow}(\ (\ int) = \{\ , \}$\} \]
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** $t \in \text{First}(\alpha)$ **do**
    • $T[A, t] = \alpha$
  - **If** $\varepsilon \in \text{First}(\alpha)$, **for each** $t \in \text{Follow}(A)$ **do**
    • $T[A, t] = \alpha$
  - **If** $\varepsilon \in \text{First}(\alpha)$ **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)

• There are tools that build LL(1) tables
Review

• For some grammars there is a simple parsing strategy

    Predictive parsing

• Next time: a more powerful parsing strategy