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
  - 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 $X.val$
  - For terminals, $val$ is the associated lexeme
  - For non-terminals, $val$ is the expression’s value (which is computed from values of subexpressions)
- We annotate the grammar with actions:
  $$E \rightarrow \text{int} \{ E.val = \text{int}.val \}$$
  $$\mid E_1 + E_2 \{ E.val = E_1.val + E_2.val \}$$
  $$\mid (E_1) \{ E.val = E_1.val \}$$
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 → E1 + E2</td>
<td>E.val = E1.val + E2.val</td>
</tr>
<tr>
<td>E1 → int5</td>
<td>E1.val = int5.val = 5</td>
</tr>
<tr>
<td>E2 → (E3)</td>
<td>E2.val = E3.val</td>
</tr>
<tr>
<td>E3 → E4 + E5</td>
<td>E3.val = E4.val + E5.val</td>
</tr>
<tr>
<td>E4 → int2</td>
<td>E4.val = int2.val = 2</td>
</tr>
<tr>
<td>E5 → int3</td>
<td>E5.val = int3.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:
  \[ E_3.val = E_4.val + E_5.val \]
  - Must compute \( E_4.val \) and \( E_5.val \) before \( E_3.val \)
  - We say that \( E_3.val \) depends on \( E_4.val \) and \( E_5.val \)

• The parser must find the order of evaluation

Dependency Graph

• Each node labeled with a non-terminal \( E \) has one slot for its \( \text{val} \) 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
  - \( E \cdot val \) is a synthesized attribute
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called 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
  \[ P \rightarrow \epsilon \mid PL \]

Attributes for the Line Calculator

- Each \( E \) has a synthesized attribute \( \text{val} \)
  - Calculated as before
- Each \( L \) has a synthesized attribute \( \text{val} \)
  \[ L \rightarrow E = \{ L \cdot \text{val} = E \cdot \text{val} \} \]
  \[ + E = \{ L \cdot \text{val} = E \cdot \text{val} + L \cdot \text{prev} \} \]
- We need the value of the previous line
- We use an inherited attribute \( L \cdot \text{prev} \)
Attributes for the Line Calculator (Cont.)

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

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

• Example ...

Example of Inherited Attributes

• $\text{val}$ synthesized

• $\text{prev}$ inherited

• All can be computed in depth-first order

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:

$$\text{mkleaf}(n) = \boxed{n}$$

$$\text{mkplus}(, , ) = \boxed{\text{PLUS}}$$

Compiler Design 1 (2011)
Constructing a Parse Tree

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

\[
E \rightarrow \text{int} \quad \{ \text{E.ast} = \text{mkleaf}(\text{int.}\text{lexval}) \} \\
\mid E_1 + E_2 \quad \{ \text{E.ast} = \text{mkplus}(E_1.\text{ast}, E_2.\text{ast}) \} \\
\mid (E_1) \quad \{ \text{E.ast} = E_1.\text{ast} \}
\]

Parse Tree Example

- Consider the string \( \text{int}_5 \ ' + ' (\text{int}_2 \ ' + ' \text{int}_3 ') \)
- A bottom-up evaluation of the \( \text{ast} \) 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
  \[
  E \rightarrow T \cdot E \mid T \\
  T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
  \]

• 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 \( E_0 \rightarrow T_1 \ast E_2 \)

• 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 \( \ast \) after \( T_1 \) does not match input token \( \ast \)

• Try \( T_1 \rightarrow \text{int} \ast T_2 \)
  - This will match but \( \ast \) 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 \( E_0 \rightarrow T_1 \)
  - Token stream: \( \text{int}_5 \ast \text{int}_2 \)

• 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

\[
\begin{array}{c}
E_0 \\
T_1 \\
\text{int}_5 \\
\ast \\
T_2 \\
E \rightarrow T \cdot E \mid T \\
T \rightarrow (E) \mid \text{int} \mid \text{int} \ast T \\
\end{array}
\]
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 $S_1()$ { return $S()$ && term(a); }
  bool $S()$ { return $S_1();$ }
  ```
- $S()$ will get into an infinite loop

- A left-recursive grammar has a non-terminal $S$
  
  $S \rightarrow S\alpha$ for some $\alpha$
- Recursive descent does not work in such cases

Elimination of Left Recursion

- Consider the left-recursive grammar
  
  $S \rightarrow S \alpha | \beta$
- $S$ generates all strings starting with a $\beta$ and followed by any number of $\alpha$’s
- The grammar can be rewritten using right-recursion
  
  $S \rightarrow \beta S'$
  $S' \rightarrow \alpha S' | \varepsilon$

More Elimination of Left-Recursion

- In general
  
  $S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m$
- 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
  
  $S \rightarrow \beta_1 S' | \ldots | \beta_m S'$
  $S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon$
**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}
  - 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
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow (E) \mid \text{int} \mid \text{int} \ast T \]

• Factor out common prefixes of productions
  \[ E \rightarrow TX \]
  \[ X \rightarrow + E \mid \varepsilon \]
  \[ T \rightarrow (E) \mid \text{int} Y \]
  \[ Y \rightarrow \ast T \mid \varepsilon \]

LL(1) Parsing Table Example

• Left-factored grammar
  \[ E \rightarrow TX \]
  \[ X \rightarrow + E \mid \varepsilon \]
  \[ T \rightarrow (E) \mid \text{int} Y \]
  \[ Y \rightarrow \ast T \mid \varepsilon \]

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>( )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+E</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>intY</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td>*T</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

LL(1) Parsing Table Example (Cont.)

• Consider the \([E, \text{int}]\) entry
  - “When current non-terminal is \( E \) and next input is \text{int}, use production \( E \rightarrow TX \)
  - This production can generate an \text{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 \([\text{E,*}]\) entry
  - "There is no way to derive a string starting with * from non-terminal \(\text{E}\)"

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 = \(<\text{S $}>\) and next
repeat
  case stack of
    \(<X, \text{rest}>\) : if \(T[X,*\text{next}] = Y_1 \ldots Y_n\)
    then stack ← \(<Y_1 \ldots Y_n \text{rest}>\);
    else error();
    \(<t, \text{rest}>\) : if \(t == *\text{next}++\)
    then stack ← \(<\text{rest}>\);
    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>(\text{E $})</td>
<td>(\text{int * int $})</td>
<td>(\text{T X})</td>
</tr>
<tr>
<td>(\text{T X $})</td>
<td>(\text{int * int $})</td>
<td>(\text{int Y})</td>
</tr>
<tr>
<td>(\text{int Y X $})</td>
<td>(\text{int * int $})</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{Y X $})</td>
<td>(* \text{int $})</td>
<td>(* \text{T})</td>
</tr>
<tr>
<td>(\text{* T X $})</td>
<td>(* \text{int $})</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{T X $})</td>
<td>(\text{int $})</td>
<td>(\text{int Y})</td>
</tr>
<tr>
<td>(\text{int Y X $})</td>
<td>(\text{int $})</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{Y X $})</td>
<td>($)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(\text{X $})</td>
<td>($)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>($)</td>
<td>($)</td>
<td>ACCEPT</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 \( A \)
  - \( S \rightarrow^* \beta A \mid \delta \)
  - We say \( t \in \text{Follow}(A) \)

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 \( X \rightarrow A_1 \ldots A_n \)
   and \( \epsilon \in \text{First}(A_i) \) 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 \( \epsilon \in \text{First}(A_i) \) for each \( 1 \leq i \leq n \)

First Sets: Example

- Recall the grammar
  \[
  \begin{align*}
  E &\rightarrow TX \\
  T &\rightarrow (E) \mid \text{int } Y \\
  X &\rightarrow +E \mid \epsilon \\
  Y &\rightarrow *T \mid \epsilon
  \end{align*}
  \]
- First sets
  \[
  \begin{align*}
  \text{First}( ( ) ) &= \{ ( ) \} \\
  \text{First}( + ) &= \{ + \} \\
  \text{First}( * ) &= \{ * \} \\
  \text{First}( \text{int} ) &= \{ \text{int} \} \\
  \text{First}( T ) &= \{ \text{int}, ( ) \} \\
  \text{First}( E ) &= \{ \text{int}, ( ) \} \\
  \text{First}( X ) &= \{ +, \epsilon \} \\
  \text{First}( Y ) &= \{ *, \epsilon \}
  \end{align*}
  \]
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 \( \$ \in \text{Follow}(S) \)

Follow Sets: Example

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

• Follow sets
  \[
  \begin{align*}
  \text{Follow}(+) & = \{ \text{int}, ( \} \\
  \text{Follow}(*) & = \{ \text{int}, ( \} \\
  \text{Follow}(() & = \{ \text{int}, ( \} \\
  \text{Follow}(E) & = \{ ), \$ \} \\
  \text{Follow}(X) & = \{ \$ , ) \} \\
  \text{Follow}(Y) & = \{ + , ), \$ \} \\
  \text{Follow}(\text{int}) & = \{ *, + , ) , \$ \}
  \end{align*}
  \]

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