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
  \[ E \rightarrow \text{int} | (E) | E + E \]
- 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 will 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:  \( X \rightarrow Y_1 \ldots Y_n \)  \{ action \}
  - That can refer to or compute symbol attributes

Semantic Actions: An Example

- Consider the grammar
  \[ E \rightarrow \text{int} | E + E | ( E ) \]
- 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:
  \[
  \begin{align*}
  E &\rightarrow \text{int} & \{ E\.val = \text{int}.val \} \\
  | & E_1 + E_2 & \{ E\.val = E_1\.val + E_2\.val \} \\
  | & ( E_1 ) & \{ E\.val = E_1\.val \}
  \end{align*}
  \]
Semantic Actions: An Example (Cont.)

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

**Productions**

- $E \rightarrow E_1 + E_2$
- $E_1 \rightarrow \text{int5}$
- $E_2 \rightarrow (E_3)$
- $E_3 \rightarrow E_4 + E_5$
- $E_4 \rightarrow \text{int2}$
- $E_5 \rightarrow \text{int3}$

**Equations**

- $E.val = E_1.val + E_2.val$
- $E_1.val = \text{int5}.val = 5$
- $E_2.val = E_3.val$
- $E_3.val = E_4.val + E_5.val$
- $E_4.val = \text{int2}.val = 2$
- $E_5.val = \text{int3}.val = 3$

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 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
  - \( E\.val \) 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
  \[ P \rightarrow \varepsilon \mid P L \]

**Attributes for the Line Calculator**

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

- Each P has a synthesized attribute val
  - The value of its last line
    \[ P \rightarrow \varepsilon \quad \{ P.val = 0 \} \]
    \[ | \quad \varepsilon P_1 L \quad \{ P.val = L.val; \]
    \[ \quad L.prev = P_1.val \} \]

- Each L has an inherited attribute prev
  - L.prev is inherited from sibling P_1.val

- Example ...

Example of Inherited Attributes

- val synthesized
- 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:

  \[
 \begin{align*}
  mkleaf(n) & = n \\
  mkplus( & , ) & = PLUS
  \end{align*}
  \]
Constructing a Parse Tree

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

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

Parse Tree Example

• Consider the string \texttt{int}_5 \texttt{ '+ ' (' int}_2 \texttt{ '+ ' int}_3 \texttt{ ') '}
• A bottom-up evaluation of the \textit{ast} attribute:
  \[
  E.ast = \text{mkplus}(\text{mkleaf}(5), \\
  \text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3))
  \]

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: Outline

• Implementation of parsers
• Two approaches
  - Top-down
  - Bottom-up
• These slides: 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: Example

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

- 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 $T_1 \rightarrow \text{int}$.
  - Token matches.
    - But + after $T_1$ does not match input token *
- Try $T_1 \rightarrow \text{int * T}_2$
  - This will match and will consume the two tokens.
    - Try $T_2 \rightarrow \text{int}$ (matches) but + after $T_1$ will be unmatched
    - Try $T_2 \rightarrow \text{int * T}_3$ but * does not match with end-of-input
- Has exhausted the choices for $T_1$
  - Backtrack to choice for $E_0$

Recursive Descent Parsing: Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for $T_1$
  - And succeed with $T_1 \rightarrow \text{int}_5 * T_2$ and $T_2 \rightarrow \text{int}_2$
  - With the following parse tree
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 $S \to S a$
  ```
  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 $S$
  $S \to^* S\alpha$ for some $\alpha$
• Recursive descent does not work in such cases
  - It goes into an infinite loop

Elimination of Left Recursion

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

More Elimination of Left-Recursion

• In general
  $S \to S \alpha_1 | ... | S \alpha_n | \beta_1 | ... | \beta_m$
• All strings derived from $S$ start with one of
  $\beta_1, ..., \beta_m$ and continue with several instances of
  $\alpha_1, ..., \alpha_n$
• Rewrite as
  $$
  S \to \beta_1 \ S' | ... | \beta_m \ S' \\
  S' \to \alpha_1 \ S' | ... | \alpha_n \ S' | \varepsilon
  $$
**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
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow (E) \mid \text{int} \mid \text{int} \ast T \]

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

- This grammar is equivalent to the original one

---

LL(1) Parsing Table Example

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

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

<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></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*T</td>
<td></td>
<td></td>
<td></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 \( E \rightarrow T \ X \)”
  - 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 \([E,*]\) 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 choose 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**

```plaintext
initialize stack ← \(<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>(E $)</td>
<td>int * int $</td>
<td>(T X)</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>(Y X 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>ε</td>
</tr>
<tr>
<td>(X $)</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>($)</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>int</th>
<th>*</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td>(T)</td>
<td>(X)</td>
<td>(X)</td>
<td>(E)</td>
<td>(E)</td>
</tr>
<tr>
<td>(X)</td>
<td>(+)</td>
<td>(_)</td>
<td>(_)</td>
<td>(_)</td>
<td>(_)</td>
</tr>
<tr>
<td>(T)</td>
<td>(_)</td>
<td>(_)</td>
<td>((_)</td>
<td>() (_)</td>
<td></td>
</tr>
<tr>
<td>(Y)</td>
<td>(_)</td>
<td>(_)</td>
<td>(_)</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
- 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

### 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 \( X \rightarrow A_1 \ldots A_n \)
   \[ \text{and } \varepsilon \in \text{First}(A_i) \text{ for each } 1 \leq i \leq n \]
4. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \)
   \[ \text{and } \varepsilon \in \text{First}(A_i) \text{ for each } 1 \leq i \leq n \]

**More constructive algorithm**

1. \( \text{First}(t) = \{ t \} \)
2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) \setminus \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_1) \).
   - Add \( \text{First}(A_2) \setminus \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_2) \).
   - ... 
   - Add \( \text{First}(A_n) \setminus \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_n) \).
   - Add \( \{ \varepsilon \} \) to \( \text{First}(X) \).
First Sets: Example

• Recall the grammar

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

• First sets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>First Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>( { ( } )</td>
</tr>
<tr>
<td>)</td>
<td>( { ) }</td>
</tr>
<tr>
<td>int</td>
<td>( { \text{int} } )</td>
</tr>
<tr>
<td>+</td>
<td>( { +, \varepsilon } )</td>
</tr>
<tr>
<td>*</td>
<td>( { *, \varepsilon } )</td>
</tr>
</tbody>
</table>

Computing Follow Sets

• Definition

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

• 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 \( \varepsilon \in \text{First}(\beta) \)

More constructive algorithm

1. First compute the \( \text{First} \) sets for all non-terminals
2. If \( S \) is the start symbol, add \( \$ \) to \( \text{Follow}(S) \)
3. For all productions \( Y \rightarrow \ldots X A_1 \ldots A_n \)
   a. Add \( \text{First}(A_1) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_1) \).
   b. Add \( \text{First}(A_2) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_2) \).
   c. ... 
   d. Add \( \text{First}(A_n) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_n) \).
   e. Add \( \text{Follow}(Y) \) to \( \text{Follow}(X) \).
**Follow Sets: Example**

- Recall the grammar

\[
E \rightarrow TX \\
X \rightarrow +E | \varepsilon \\
T \rightarrow (E) | \text{int}Y \\
Y \rightarrow *T | \varepsilon
\]

- Follow sets

Follow(+) = \{ \text{int}, (\} \\
Follow(*) = \{ \text{int}, (\} \\
Follow(()) = \{ \text{int}, (\} \\
Follow(+) = \{ \text{int}, (\} \\
Follow((()) = \{ \text{int}, (\} \\
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 (LL(1))

- Next time: a more powerful parsing strategy