CSC 425 - Principles of Compiler Design I

Abstract Syntax Trees

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 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 (ASTs) are like parse trees, but ignore some details

Abstract Syntax Trees

- Consider the grammar

$$E \rightarrow int|(E)|E + E$$

- and the string

$$5 + (2 + 3)$$

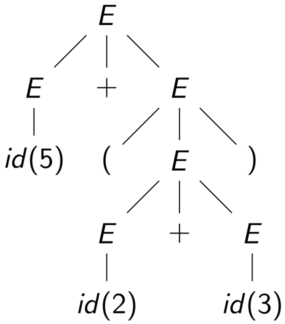
- After lexical analysis (a list of tokens)

$$int(5), plus, lparen, int(2), plus, int(3)$$

- During parsing, we build a parse tree ...

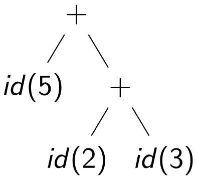
Example of Parse Tree

- Traces the operation of the parser
- Captures the nesting structure
- But has too much info, for example parentheses



Example of AST

- Also captures the nesting structure
- But *abstracts* from the concrete syntax making it more compact and easier to use
- An important data structure in a compiler



Semantic Actions

- Each grammar symbol may have *attributes*
  - An attribute is a property of a programming language construct
  - For terminal symbols attributes can be calculated by the lexer
- Each production may have an *action*
  - Written as: $X \rightarrow Y_1 \ldots Y_2\{action\}$
  - That can refer to or compute symbol attributes
- This is what we will use to construct ASTs

Semantic Actions: Example

- Consider the grammar

$$E \rightarrow int | (E) | E + E$$

- 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
- We annotate the grammar with actions:

$$\begin{array}{llll} E & \rightarrow & int & \{E.val = int.val\} \\ & | & (E_1) & \{E.val = E_1.val\} \\ & | & E_1 + E_2 & \{E.val = E_1.val + E_2.val\} \end{array}$$

Semantic Actions: Example Continued

- String: $5 + (2 + 3)$
- Tokens: int(5), plus, lparen, int(2), plus, int(3)

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int(5)$	$E_1.val = int(5).val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int(2)$	$E_4.val = int(2).val = 2$
$E_5 \rightarrow int(3)$	$E_5.val = int(3).val = 3$

Semantic Actions: Dependencies

- Semantic actions specify a system of equations, but the 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

Evaluating Attributes

- An attribute must be computed after all its successors in the dependency graph have been computed
- Such an order exists when there are no cycles
- In the previous example, attributes can be computed bottom-up

Types of Attributes

- Synthesized attributes
  - Calculated from attributes of descendants 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
- Inherited attributes
  - Calculated from attributes of the parent node(s) and/or siblings in the parse tree

Example: Line Calculator

- Each line contains an expression

$$E \to int \mid E + E$$

- Each line is terminated with the $=$ sign

$$L \to E = \mid + E =$$

- In the second form, the value of evaluating the previous line is used as a starting value
- A program is a sequence of lines

$$P \to \epsilon \mid PL$$

Attributes for the Line Calculator

- Each E has a synthesized attribute *val*
- Each L has a synthesized attribute *val*

$$\begin{aligned} L \to E = & \qquad \{L.val = E.val\} \\ \mid + E = & \qquad \{L.val = E.val + L.prev\} \end{aligned}$$

- We need the value of the previous line
- We use an inherited attribute *L.prev*

Attributes for the Line Calculator

- Each P has a synthesized attribute *val*

$$
\begin{array}{ll}
P \to \epsilon & \{P.val = 0\} \\
\quad \mid P_1 L & \{P.val = L.val; \\
 & \; L.prev = P_1.val\}
\end{array}
$$

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

Semantic Actions: Notes

- Semantic actions can be used to build ASTs
- And many other things, such as, type checking and code generation
- This process is called syntax-directed translation – a substantial generalization over context-free grammars

Constructing an AST

- We first define the AST data type
- Consider an abstract tree type with two constructors:
  - mkleaf(n)
  - mkplus(left_tree, right_tree)

Constructing a Parse Tree

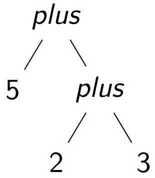
- We define a synthesized attribute *ast*
  - Values of *ast* values are ASTs
  - We assume that $int.lexval$ is the value of the integer lexeme
  - Computed using semantic actions

$$
\begin{aligned}
E \rightarrow\ & int && \{E.ast = makeleaf(int.val)\} \\
\mid\ & (E_1) && \{E.ast = E_1.ast\} \\
\mid\ & E_1 + E_2 && \{E.ast = mkplus(E_1.ast, E_2.ast)\}
\end{aligned}
$$

Parse Tree Example

- Consider the string: $5 + (2 + 3)$
- A bottom-up evaluation of the *ast* attribute:

$$
\begin{aligned}
E.ast = {} & mkplus(mkleaf(5) \\
& \qquad mkplus(mkleaf(2), mkleaf(3)))
\end{aligned}
$$



Review of Abstract Syntax Trees

- We can specify language syntax using a context-free grammar
- 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