Semantic Analysis

CS 440: Programming Languages and Translators, Fall 2020

A. Semantic Analysis

- Next compiler phase after parsing
- May come before or as part of constructing internal representation of program
  - AST abstract syntax tree: + of 2 and 2, not Expr \(\Rightarrow\) Term TermList where the Term \(\Rightarrow\) Factor
  \(\Rightarrow\) 2, the TermList \(\Rightarrow\) + Term TermList
- Constructed separately or interleaved with parse
- Internal representation makes it easier to check and process program
  - Enforce static semantic rules (e.g. typechecking)
  - Intermediate code generation (later)

B. Attributes

- **Attributes** are properties associated with grammar symbols.
- **Attribute grammars** are grammars that include attribute calculation instructions
  - Creation / calculation of attribute values specified by semantic rules
  - Some attribute values are built up from a node's children.
    - E.g., the values of subexpressions of calculations involving only constants.
  - Some values come from a node's parent or siblings.
    - E.g., what variables have been declared and can be used here?
  - Attribute values can be easy to calculate, but some require can making repeated passes through the parse tree.
- A **Syntax-Directed Definition (SDD)** is a context-free grammar with attributes attached to grammar symbols and semantic rules (program code) attached to productions.
- A **Syntax-Directed Translation Scheme (SDTS)** is a context-free grammar with semantic actions embedded within production rules (not just attached to rules). Yacc is an example.

Attribute References

- To refer to property of grammar symbol X, use X.property
  - If multiple X's in rule, distinguish using X₀, X₁, ...
  - Examples: Constant.value, Variable.type
- Attribute rules attached to each grammar rule build and use attributes.
Synthesized Attribute

- A synthesized attribute is an attribute that doesn’t depend only attributes of child nodes. In terms of an attribute grammar, for a rule $A \rightarrow \alpha$, a synthetic property of $A$ is calculated using only the attributes of symbols in $\alpha$.
- In terms of the parse tree, for a synthetic attribute, values flow up from leaves toward parents. A node's attribute value is calculated using the values of the attributes of its children.

Example: Value of an expression

- Figure 1 below shows the general parse tree and attribute calculations for a couple of cases of the expression grammar. **Notation:** attribute $\uparrow$ is short for attribute$_{syn}$ emphasizes that the attribute is synthesized for sending up to a node's parent. (All the attributes in Figure 1 are synthesized, so using the $\uparrow$ is a bit of overkill.)

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Attribute Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E + T$</td>
<td>$E_0.val = E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val = T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T * F$</td>
<td>$T_0.val = T_1.val * F.val$</td>
</tr>
<tr>
<td>$F \rightarrow Const$</td>
<td>$F.val = Const.val$</td>
</tr>
</tbody>
</table>

Figure 1: Partial Parse Tree and Attribute Calculation for $E$
C. S-Attributed Grammar/SDD

- An S-Attributed grammar or SDD (Syntax-Directed Definition) has only synthetic attributes.
- With a bottom-up (LR) parse, when you reduce \( A \rightarrow \alpha \), you've already parsed \( \alpha \), so you have the attributes of the symbols of \( \alpha \).
- With a top-down (LL) parse, we prepare for calculating the attribute when we decide which \( A \rightarrow \alpha \) rule we're going to use for \( A \). When we've finished parsing \( \alpha \), we can calculate a synthetic attribute value for ourself then.
  - With a recursive descent parser, you save the attribute results for each recursive call and combine them before returning.

**Example 1.** (From Aho.) We can use synthetic attributes to translate signed bitstrings to decimal values. The attributes, \( \text{BNum.val} \), \( \text{Sign.val} \), \( \text{List.val} \), and \( \text{Bit.val} \), are all synthesized attributes that represent integers. The result is an S-attributed SDD.

\[
\begin{align*}
\text{BNum} & \rightarrow \text{Sign List} & \{ \text{BNum.val} &= \text{Sign.val} \times \text{List.val} \} \\
\text{Sign} & \rightarrow + & \{ \text{Sign.val} &= +1 \} \\
\text{Sign} & \rightarrow - & \{ \text{Sign.val} &= -1 \} \\
\text{List} & \rightarrow \text{List1 Bit} & \{ \text{List.val} &= 2 \times \text{List1.val} + \text{Bit.val} \} \\
\text{List} & \rightarrow \text{Bit} & \{ \text{List.val} &= \text{Bit.val} \} \\
\text{Bit} & \rightarrow 0 & \{ \text{Bit.val} &= 0 \} \\
\text{Bit} & \rightarrow 1 & \{ \text{Bit.val} &= 1 \}
\end{align*}
\]

**Example 2:** (Still from Aho.) Here are Yacc translation rules implementing the SDD above for translating signed bit strings into decimal numbers. The identifiers \( $$ \), \( $1 \), \( $2 \) and so on in Yacc actions are synthesized attributes.

\[
\begin{align*}
\text{BNum} : & \text{Sign List} \{ \$$ &= $1 \times $2; \} \\
\text{Sign} : & \text{'}+\text{'} & \{ \$$ &= +1; \}
| \text{'}-\text{'} & \{ \$$ &= -1; \}
\}
\text{List} : & \text{List Bit} \{ \$$ &= 2 \times $1 + $2; \}
| \text{Bit}
\}
\text{Bit} : & \text{'}0\text{'} & \{ \$$ &= 0; \}
| \text{'}1\text{'} & \{ \$$ &= 1; \}
\}
\]

**Example 3.** (Adapted from Figure 5.10, p.319, PDB) Here is an S-attributed SDD based on an SLR(1) grammar that translates arithmetic expressions into abstract syntax trees. \( E \) has the synthesized
attribute \( E.\text{node} \) and \( T.\text{node} \) the synthesized attribute \( T.\text{node} \). \( E.\text{node} \) and \( T.\text{node} \) point to a node in the abstract syntax tree. The function \( \text{Node}(\text{op}, \text{left}, \text{right}) \) returns a pointer to a node with three fields: 1: \( \text{op} \); 2: pointer to the left subtree; 3: pointer to the right subtree. The function \( \text{Leaf}(\text{id}, \text{value}) \) returns a pointer to a node with two fields: 1: \( \text{id} \); 2: the value of the token. See Example 5.11 in ALSU.

\[
\begin{align*}
E & \rightarrow E_1 + T \quad \{ E.\text{node} = \text{Node}(\text{`+'}, E_1.\text{node}, T.\text{node}); \} \\
E & \rightarrow T \quad \{ E.\text{node} = T.\text{node}; \} \\
T & \rightarrow ( E ) \quad \{ T.\text{node} = E.\text{node}; \} \\
T & \rightarrow \text{id} \quad \{ T.\text{node} = \text{Leaf}(\text{id}, \text{id}.\text{entry}); \}
\end{align*}
\]

**D. Inherited and General Attributes**

**Inherited Attribute**

- An *inherited attribute* is an attribute whose value depends on attributes of a node's parents, siblings, and the node itself. They're typically used when one part of a parse tree creates a value and we want that value passed on to the rest of the tree.

- **Example:** With a symbol table; each node receives a symbol table and can augment it or look up information in it.

- E.g., take a grammar with \( \text{Block} \rightarrow \text{Decl} \text{ Stmt where Decl} \Rightarrow^{\star} \text{ int } x; \text{ and Stmt} \Rightarrow^{\star} x = x + 1; \) The node for \( \text{Block} \) receives a symbol table, passes it on to the declaration, which adds an entry for \( x \). \( \text{Block} \) then passes this new table to \( \text{Stmt} \) where eventually the assignment can make sure that the uses of \( x \) are type-correct.

**L-Attributed Grammar**

- In an *L-attributed grammar*, for a rule \( A \rightarrow \alpha \beta \), an inherited property of \( B \) can depend on inherited properties of \( A \) and inherited and synthetic properties of symbols in \( \alpha \).

- I.e., for a given parse tree node, the value of an inherited attribute can depend not only on the values of the subtrees of the node but also on values passed down from the parent node and also the node's left siblings.
  - (I.e., attribute values can be generated using a depth-first search of a parse tree.)

- Such inherited attributes can be calculated during an LL parse: At any point in the parse, we've looked at the nodes from the current node up to the root and also to the left within the same rule.

- L-attributed grammars are a strict superset of the S-attributed grammars.
Example 4: Building a Parse Tree with an LR Grammar

- This is adapted from Example 5.12 (p.319), which uses the grammar shown in Figure 5.13 (p.321).
  The adaptations: Instead of $X.inh$, $X.syn$, and $X.node$, used below are $X.tree\downarrow$, $X.tree\uparrow$, and $X.tree\uparrow$.

- $E \rightarrow T E' \quad \{ E'.tree\downarrow = T.tree\uparrow; \ E.tree\uparrow = E'.tree\uparrow \}$
  (E' receives the tree built by T, and E returns the tree returned by E'.)

- $E' \rightarrow + T E'_{1} \quad \{ E'.tree\downarrow = \text{Node}('+', E'.tree\downarrow, T.tree\uparrow); \ E_{1}.tree\uparrow = E'.tree\uparrow \}$

- $E' \rightarrow - T E'_{1} \quad \{ E'.tree\downarrow = \text{Node}('-', E'.tree\downarrow, T.tree\uparrow); \ E_{1}.tree\uparrow = E'.tree\uparrow \}$

- $E' \rightarrow \epsilon \quad \{ A.tree\uparrow = A.tree\downarrow \}$

- $T \rightarrow ( E ) \quad \{ T.tree\uparrow = E.tree\uparrow \}$

- $T \rightarrow \text{id} \quad \{ T.tree\uparrow = \text{Leaf}(\text{id}, \text{id}.entry) \}$

- $T \rightarrow \text{num} \quad \{ T.tree\uparrow = \text{Leaf}(\text{num}, \text{num}.val) \}$

Example 5: Symbol Table

- The rule $\text{Block} \rightarrow \text{DeclList} \text{StmtList}$ is the key to how the symbol table is used.
  - The block inherits a symbol table, which it gets inherited by the declaration list.
  - The declaration list synthesizes an updated symbol table (one with the declarations added), which gets inherited by the statement list.
  - The statement list can check the table to verify that variables have already been declared, and it can update the table with information it finds about how variables get used.
    - E.g., if a statement assigns an integer to a variable then we can check that against what's already in the table or add it to the table, if we didn't already know the type.
  - So the statement list synthesizes an updated symbol table and passes it up to the block rule.
  - When exiting the block, we remove the local variables that were added by the declaration list and send the resulting symbol table up to whoever called Block.

Program $\rightarrow$ Block

  \{ Block.symtab\downarrow = \text{empty table} \}

Block $\rightarrow$ DeclList StmtList

  \{ DeclList.symtab\downarrow = Block.symtab\downarrow \}
  StmtList.symtab\downarrow = DeclList.symtab\uparrow
  Block.symtab\uparrow = StmtList.symtab\uparrow \text{ minus local variables declared by DeclList. } \}

(Question: Why don't we want Block.symtab\uparrow = Block.symtab\downarrow ?)

DeclList_{0} $\rightarrow$ Decl DeclList_{1}

  \{ Decl.symtab\downarrow = DeclList_{0}.symtab\downarrow \}
  DeclList_{1}.symtab\downarrow = Decl.symtab\uparrow
  DeclList_{0}.symtab\uparrow = DeclList_{1}.symtab\uparrow \}

Decl $\rightarrow$ Type Id

  \{ Decl.symtab\uparrow = Decl.symtab\downarrow + \text{new_symtab_entry}(\text{Type.type\uparrow}, \text{Id.name\uparrow}) \}
Figure 3 below shows the attribute flow attached to a parse tree.

\[ \text{Decl.symtab} \uparrow = \text{Decl.symtab} \downarrow + \text{new_symtab_entry}(\text{Type.type} \uparrow, \text{Id.name} \uparrow) \]

Figure 3: Path of symbol table attribute