Semantic Analysis

Read: Scott, Chapter 4.1-4.3
Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

- Attribute Grammars
  - Attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars
Static Semantics

- Earlier we considered **syntax analysis**
  - Informally, syntax deals with the **form** of programming language constructs
- We now look at **static semantic analysis**
  - Semantics deals with the **meaning** of programming language constructs
- The distinction between the two is fuzzy
  - In practice, anything that is not expressed in terms of certain CFG (LALR(1), in particular) is considered semantics
Static Semantics vs. Dynamic Semantics

- Static semantic analysis (compile-time)
  - Informally, reasons about program properties statically, before program execution
  - E.g., determine static types of expressions, detect certain errors

- Dynamic semantic analysis (run-time)
  - Reasons about program properties dynamically, during program execution
  - E.g., could expression $a[i]$ index out of array bounds, etc.?
The Role of Semantic Analysis

- Detect errors in programs!
- Static semantic analysis
  - Detect as many errors as possible early, before execution
    - Type inference and type checking
- Dynamic semantic analysis
  - Detect errors by performing checks during execution
    - Again, detect errors as early as possible. E.g., flagging an array-out-of-bounds at assignment $a[i] = \ldots$ is useful
    - Tradeoff: dynamic checks slow program execution
- Languages differ greatly in the amount of static semantic analysis and dynamic semantic analysis they perform
Examples of Static Semantic Errors

- **Type mismatch:**
  - \( x = y+z+w \): type of left-hand-side does not “match” type of right-hand-side
  - \( A \ a; \ldots \ ; a.m() \): \( m() \) cannot be invoked on a variable of type \( A \)

- Definite assignment check in Java: a local variable must be assigned before it is used
Examples of Dynamic Semantic Errors

- Null pointer dereference:
  - `a.m()` in Java, and `a` is null (i.e., uninitialized reference)
  - What happens?

- Array-index-out-of-bounds:
  - `a[i]`, `i` goes beyond the bounds of `a`
  - What happens in C++? What happens in Java?

- Casting an object to a type of which it is not an instance
  - C++? Java?

- And more...
Static Semantics vs. Dynamic Semantics

- Again, distinction between the two is fuzzy
- For some programs, the compiler can predict run-time behavior by using static analysis
  - E.g., there is no need for a nullness check:
    ```java
    x = new X();
    x.m(); // x is non-null
    ```
- In general, the compiler cannot predict run-time behavior
  - Static analysis is limited by the halting problem
Semantic Analyzer

Semantic analyzer performs static semantic analysis on parse trees and ASTs. Optimizer performs static semantic analysis on intermediate 3-address code.
Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

- Attribute Grammars
  - Attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars
Attribute Grammars: generalization of Context-Free Grammars

- **Attributes**
  - Each grammar symbol has one or more values called attributes associated with it. Each parse tree node has its own instances of those attributes; attribute value carries the “meaning” of the parse tree rooted at node

- **Semantic rules**
  - Each grammar production has associated rule, which may refer to and compute the values of attributes
Example: Attribute Grammar to Compute Value of Expression (denote grammar by AG1)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow E$</td>
<td>print($E.val$)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val := E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.val := T_1.val * F.val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.val := F.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{num}$</td>
<td>$F.val := \text{num}.val$</td>
</tr>
</tbody>
</table>

$\text{val}$: Attributes
Example: Decorated parse tree for input
3*5  +  2*4

S → E
E → E₁+T
E → T
T → T₁*F
T → F
F → num

print(E.val)
E.val := E₁.val+T.val
E.val := T.val
T.val := T₁.val*F.val
T.val := F.val
F.val := num.val
Example

- **val**: Attributes associated to symbols
  - Intuitively, $A.val$ holds the value of the expression, represented by the subtree rooted at $A$
  - Separate attributes are associated with separate nodes in the parse tree
- Indices are used to distinguish between symbols with same name within same production
  - E.g., $E \rightarrow E_1 + T$ \hspace{1cm} $E.val := E_1.val + T.val$
- Attributes of terminals supplied by scanner
  - In example, attributes of + and * are never used
Building an Abstract Syntax Tree (AST)

- An AST is an abbreviated parse tree
  - Operators and keywords do not appear as leaves, but at the interior node that would have been their parent
  - Chains of single productions are collapsed

- Compilers typically work with ASTs
Building ASTs for Expressions

Parse tree for $3*5+2*4$

Abstract syntax tree (AST)

How do we construct syntax trees for expressions?
Attribute Grammar to build AST for Expression (denote by AG2)

An attribute grammar:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.nptr := \text{mknode}(+, E_1.nptr, T.nptr)$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.nptr := T.nptr$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.nptr := \text{mknode}(*, T_1.nptr, F.nptr)$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.nptr := F.nptr$</td>
</tr>
<tr>
<td>$F \rightarrow \text{num}$</td>
<td>$F.nptr := \text{mkleaf}(\text{num}, \text{num.val})$</td>
</tr>
</tbody>
</table>

$\text{mknode}(\text{op}, \text{left}, \text{right})$ creates an operator node with label $\text{op}$, and two fields containing pointers $\text{left}$, to left operand and $\text{right}$, to right operand.

$\text{mkleaf}(\text{num}, \text{num.val})$ creates a leaf node with label $\text{num}$, and a field containing the value of the number.
Constructing ASTs for Expressions

Input:
3 * 5 + 2 * 4

E → E₁ + T  \( E.nptr := \text{mknode}( \cdot, E₁.nptr, T.nptr) \)
E → T  \( E.nptr := T.nptr \)
T → T₁ * F  \( T.nptr := \text{mknode}( \cdot, T₁.nptr, F.nptr) \)
T → F  \( T.nptr := F.nptr \)
F → num  \( F.nptr := \text{mkleaf}(\cdot, \text{num}.\text{val}) \)
Exercise

We know that the language $L = a^n b^n c^n$ is not context free. It can be captured however with an attribute grammar. Give an underlying CFG and a set of attribute rules that associate an attribute $\text{ok}$ with the root $S$ of each parse tree, such that $S.\text{ok}$ is true if and only if the string corresponding to the fringe of the tree is in $L$. 
Exercise
Exercise

Consider the expression grammar

\[
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow \text{num} | (E)
\]

Give attribute rules to accumulate into the root a count of the maximum depth to which parentheses are nested in the expression. E.g., 

\[((1 + 2)*3 + 4)*5 + 6\] has a count of 2.
Exercise
Another Grammar

Now, the right-recursive LL(1) grammar:

\[
\begin{align*}
E & \rightarrow T TT \\
TT & \rightarrow - T TT \\
TT & \rightarrow \varepsilon \\
T & \rightarrow \text{num}
\end{align*}
\]

Goal: construct an attribute grammar that computes the value of an expression

- Values must be computed “normally”, i.e.,
  - 5–3–2 must be evaluated as \((5–3)–2\), not as \(5–(3–2)\)
What happens if we wrote a “bottom-up attribute flow” grammar?

*Example grammar:*

- \[ E \rightarrow T \ TT \]
  - \[ E.val = T.val - TT.val \]

- \[ TT \rightarrow - T TT_1 \]
  - \[ TT.val = T.val - TT_1.val \]

- \[ TT \rightarrow \epsilon \]
  - \[ TT.val = 0 \]

- \[ T \rightarrow \text{num} \]
  - \[ T.val = \text{num.val} \]

*Hack:*

- \[ E \rightarrow T \ TT \]
  - \[ E.val = T.val - TT.val \]

- \[ TT \rightarrow - T TT_1 \]
  - \[ TT.val = T.val + TT_1.val \]

- \[ TT \rightarrow \epsilon \]
  - \[ TT.val = 0 \]

- \[ T \rightarrow \text{num} \]
  - \[ T.val = \text{num.val} \]

Unfortunately, this won’t work if we add \[ TT \rightarrow + T TT_1 \]
Attribute Grammar to Compute Value of Expressions (denote by AG3)

\[
\begin{align*}
E & \rightarrow T \ TT \\
TT & \rightarrow -T \ TT \mid +T \ TT \mid \epsilon \\
T & \rightarrow \text{num}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E \rightarrow T \ TT)</td>
<td>(1) (TT.\text{sub} := T.\text{val})  (\text{or}) (2) (E.\text{val} := TT.\text{val})</td>
</tr>
<tr>
<td>(TT \rightarrow -T \ TT_1)</td>
<td>(1) (TT_1.\text{sub} := TT.\text{sub} - T.\text{val})  (\text{or}) (2) (TT.\text{val} := TT_1.\text{val})</td>
</tr>
<tr>
<td>(TT \rightarrow +T \ TT_1)</td>
<td>(1) (TT_1.\text{sub} := TT.\text{sub} + T.\text{val})  (\text{or}) (2) (TT.\text{val} := TT_1.\text{val})</td>
</tr>
<tr>
<td>(TT \rightarrow \epsilon)</td>
<td>(1) (TT.\text{val} := TT.\text{sub})</td>
</tr>
</tbody>
</table>

\(T \rightarrow \text{num}\)  \(\text{or}\) (1) \(T\.\text{val} := \text{num}.\text{val}\)  \(\text{(provided by scanner)}\)

Attributes flow from parent to node, and from “siblings” to node!
Attribute Flow

Attribute $TT_1.sub$: computed based on parent $TT$ and sibling $T$: $TT.sub - T.val$

E.g., $25 - 1 - 3 - 6$

$TT$ holds subtotal $24$ (for $25 - 1$, computed so far)

$T$ holds value $3$ (i.e., the value of next term)

$TT_1$ gets subtotal $21$ (for $25 - 1 - 3$)

Passed down the tree of $TT_1$ to next $TT$ on chain

Eventually, we hit $TT \rightarrow \varepsilon$ and value gets subtotal $15$

Value $15$ is passed back up
Example
Attribute Flow

- Attribute `.val` carries the total value
- Attribute `.sub` is the subtotal carried from left

Rules for nonterminals $E$, $T$ do not perform computation
- No need for `.sub` attribute
- `.val` attribute is carried to the right
  - In $E \rightarrow T TT : \text{val}$ of $T$ is passed to sibling $TT$
  - In $TT \rightarrow -T TT_1 : \text{val}$ of $T$ is passed to sibling $TT_1$
Attribute Flow

- Rules for nonterminal $TT$ do perform computation
  - $TT$ needs to carry subtotal in .sub
    - E.g., in $TT \rightarrow - T TT_1$ the subtotal of $TT_1$ is computed by subtracting the value of $T$ from the subtotal of $TT$
Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

- Attribute Grammars
  - Attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars
Synthesized and Inherited Attributes

- **Synthesized attributes**
  - Attribute value computed from attributes of *descendants* in parse tree, and/or attributes of *self*
  - E.g., attributes *val* in AG1, *val* in AG3
  - E.g., attributes *nptr* in AG2

- **Inherited attributes**
  - Attribute value computed from attributes of *parent* in tree and/or attributes of *siblings* in tree
  - E.g., attributes *sub* in AG3
    - In order to compute value “normally” we needed to pass sub down the tree (sub is inherited attribute).
S-attributed Grammars

- An attribute grammar for which all attributes are synthesized is said to be S-attributed
  - “Arguments” of rules are attributes of symbols from the production right-hand-side
    - I.e., attributes of children in parse tree
  - “Result” is placed in attribute of the symbol on the left-hand-side of the production
    - I.e., computes attribute of parent in parse tree
  - I.e., attribute values depend only on descendants in tree. They do not depend on parents or siblings in tree!
Questions

- Can you give examples of S-attributed grammars?
  - Answer: AG1 and AG2

- How can we evaluate S-attributed grammars?
  - I.e., in what order do we visit nodes of the parse tree and compute attributes, bottom-up or top-down?
    - Answer: bottom-up
L-attributed Grammar

- An attribute grammar is **L-attributed** if each inherited attribute of $X_j$ on the right-hand-side of $A \rightarrow X_1 X_2 \ldots X_{j-1} X_j \ldots X_n$ depends only on
  - (1) the attributes of symbols to the left of $X_j$: $X_1, X_2, \ldots, X_{j-1}$
  - (2) the inherited attributes of $A$
Questions

- Can you give examples of L-attributed grammars?
  - Answer: AG3

- How can we evaluate L-attributed grammars?
  - I.e., in what order do we visit the nodes of the parse tree?
  - Answer: top-down
An attribute grammar is **L-attributed** if each inherited attribute of $X_j$ on the right-hand-side of $A \rightarrow X_1 X_2 \ldots X_{j-1} X_j \ldots X_n$ depends only on

1. the attributes of symbols to the left of $X_j$: $X_1, X_2, \ldots, X_{j-1}$
2. the inherited attributes of $A$

Why the restriction on siblings and kinds of attributes of parent? Why not allow dependence on siblings to the right of $X_j$, e.g., $X_{j+1}$, etc.?
Recursive Descent (sketch)

\[
\begin{align*}
S & \rightarrow E \, \$\$ \\
E & \rightarrow T \, TT \\
TT & \rightarrow - \, T \, TT \mid + \, T \, TT \mid \epsilon \\
T & \rightarrow \text{num}
\end{align*}
\]

num \text{ } S()
   \text{case lookahead() of}
   \quad \text{num: } \text{val} = E(); \text{match($\$$); return val}
   \quad \text{otherwise PARSE\_ERROR}

num \text{ } E()
   \text{case lookahead() of}
   \quad \text{num: } sub = T(); \text{val} = TT(sub); \text{return val}
   \quad \text{otherwise PARSE\_ERROR}

num \text{ } TT(num \text{ sub})
   \text{case lookahead() of}
   \quad - : \text{match( ‘−’); } Tval = T(); \text{val} = TT(sub − Tval); \text{return val}
   \quad + : \text{match( ‘+’); } Tval = T(); \text{val} = TT(sub − Tval); \text{return val}
   \quad \$\$ : \text{val} = sub; \text{return val}
   \quad \text{otherwise: PARSE\_ERROR}
Evaluating Attributes and Attribute Flow

- S-attributed grammars
  - A very special case of attribute grammars
  - Most important case in practice
  - Can be evaluated on-the-fly during a bottom-up (LR) parse

- L-attributed grammars
  - A proper superset of S-attributed grammars
    - Each S-attributed grammar is also L-attributed because restriction applies only to inherited attributes
  - Can be evaluated on-the-fly during a top-down (LL) parse
The End