Short-Circuit Evaluation

• A short-circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators.
  – Consider \((a < b) \&\& (b < c)\):
    • If \(a \geq b\), there is no point evaluating \(b < c\) because \((a < b) \&\& (b < c)\) is automatically false.
• \((x \&\& y) \equiv \text{if } x \text{ then } y \text{ else false}\)
• \((x \|\| y) \equiv \text{if } x \text{ then true else } y\)

Short-Circuit Evaluation

• Short-circuit evaluation may lead to unexpected side effects and cause error.
  – E.g., \((a > b) \|\| ((b++) / 3)\)
• C, C++, and Java:
  – Use short-circuit evaluation for Boolean operations (\&\& and \|\|)
  – Also provide bitwise operators that are not short circuit (\& and \|)
Short-Circuit Evaluation

• Ada: programmers can specify either

<table>
<thead>
<tr>
<th>Non-SC eval</th>
<th>SC eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x or y)</td>
<td>(x or else y)</td>
</tr>
<tr>
<td>(x and y)</td>
<td>(x and then y)</td>
</tr>
</tbody>
</table>

Control Structures

• Selection
• Iteration
  – Iterators
• Recursion
• Concurrency & non-determinism
  – Guarded commands
Iteration Based on Data Structures

- A data-based iteration statement uses a user-defined data structure and a user-defined function to go through the structure's elements
  - The function is called an iterator
  - The iterator is invoked at the beginning of each iteration
  - Each time it is invoked, an element from the data structure is returned
  - Elements are returned in a particular order

A Java Implementation for Iterator

```java
class BinTree<T> implements Iterable<T> {
    BinTree<T> left;
    BinTree<T> right;
    T val;
    ...
    // other methods: insert, delete, lookup, ...

    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }

    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>();
        TreeIterator(BinTree<T> n) {
            if (n.val != null) s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            BinTree<T> n = s.pop();
            if (n.right != null) s.push(n.right);
            if (n.left != null) s.push(n.left);
            return n.val;
        }
        public void remove() {
            throw newUnsupportedOperationException();
        }
    }
}
```
Guarded Commands

• New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
• We cover guarded commands because they are the basis for two linguistic mechanisms developed later for concurrent programming in two languages: CSP and Ada

Motivations of Guarded Commands

• To support a program design methodology that ensures correctness during development rather than relying on verification or testing of completed programs afterwards
• Also useful for concurrency
• Increased clarity in reasoning
Guarded Commands

- Two guarded forms
  - Selection (guarded if)
  - Iteration (guarded do)

Guarded Selection

if <boolean> -> <statement>
[] <boolean> -> <statement>
...
[] <boolean> -> <statement>
fi

- Semantics
  - When this construct is reached
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically
    - If none is true, it is a runtime error
- Idea: Forces one to consider all possibilities
An Example

\[
\begin{align*}
\text{if } i = 0 & \rightarrow \text{sum} := \text{sum} + i \\
[i] & i > j \rightarrow \text{sum} := \text{sum} + j \\
[i] & j > i \rightarrow \text{sum} := \text{sum} + i \\
\text{fi}
\end{align*}
\]

• If \( i = 0 \) and \( j > i \), the construct chooses nondeterministically between the first and the third assignment statements
• If \( i = j \) and \( i \neq 0 \), none of the conditions is true and a runtime error occurs

Guarded Selection

• The construction can be an elegant way to state that the order of execution, in some cases, is irrelevant

\[
\begin{align*}
\text{if } x \geq y & \rightarrow \text{max} := x \\
[i] & y \geq x \rightarrow \text{max} := y \\
\text{fi}
\end{align*}
\]

– E.g., if \( x = y \), it does not matter which we assign to \( \text{max} \)
– This is a form of abstraction provided by the nondeterministic semantics
Guarded Iteration

- **Semantics:**
  - For each iteration:
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically, and then start loop again
    - If none is true, exit the loop
- **Idea:** if the order of evaluation is not important, the program should not specify one

An Example

- **Given four integer variables:** q1, q2, q3, and q4, rearrange the values so that q1 ≤ q2 ≤ q3 ≤ q4
- **Without guarded iteration, one solution is to put the values into an array, sort the array, and then assigns the value back to the four variables**
An Example

• While the solution with guarded iteration is not difficult, it requires a good deal of code
• There is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts

Reference

Semantic Analysis

In Text: Chapter 4

Outline [1]

• Static semantics
  – Attribute grammars
• Dynamic semantics
  – Operational semantics
  – Denotational semantics
Syntax vs. Semantics

• Syntax concerns the form of a valid program
• Semantics concerns its meaning
• Meaning of a program is important
  – It allows us to enforce rules, such as type consistency, which go beyond the form
  – It provides the information needed to generate an equivalent output program

Two types of semantic rules

• Static semantics
• Dynamic semantics
Static Semantics

• There are some characteristics of the structure of programming languages that are difficult or impossible to describe with BNF
  – E.g., type compatibility: a floating-point value cannot be assigned to an integer type variable, but the opposite is legal

Static Semantics

• The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs
  – Syntax rather than semantics
• Many static semantic rules of a language state its type constraints
Dynamic semantics

- It describes the meaning of expressions, statements, and program units
- Programmers need dynamic semantics to know precisely what statements of a language do
- Compiler writers need to define the semantics of the languages for which they are writing compilers

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
Role of Semantic Analysis

• The principal job of the semantic analyzer is to enforce static semantics
  – Constructs a syntax tree (usually first)
  – Performs analysis of information that is gathered
  – Uses that information later during code generation

Conventional Semantic Analysis

• Compile-time analysis and run-time “actions” that enforce language-defined semantics
  – Static semantic rules are enforced at compile time by the compiler
    • Type checking
  – Dynamic semantic rules are enforced at runtime by the compiler-generated code
    • Bounds checking
STATIC SEMANTICS

Attribute Grammar

- A device used to describe more of the structure of a programming language than can be described with a context-free grammar
- It provides a formal framework for decorating parse trees
- An attribute grammar is an extension to a context-free grammar
Attribute Grammar

- The extension includes
  - Attributes
  - Attribute computation functions
  - Predicate functions

A Running Example

- Context-Free Grammar (CFG)

```plaintext
<assign> -> <var> = <expr>
<expr>   -> <var> + <var>
<expr>   -> <var>
<var>    -> A | B | C
```

- Note:
  - It only focuses on potential structured sequence of tokens
  - It says nothing about the meaning of any particular program
Attributes

- Associated with each grammar symbol $X$ is a set of attributes $A(X)$. The set $A(X)$ consists of two disjoint sets: $S(X)$ and $I(X)$.

Attributes

- $S(X)$: synthesized attributes, which are used to pass semantic information up a parse tree.
Attributes

- $I(X)$: inherited attributes, which pass semantic information down or across a tree. Similar to variables because they can also have values assigned to them.

Intrinsic Attributes

- Synthesized attributes of leaf nodes whose values are determined outside the parse tree
  - E.g., the type of a variable can come from the symbol table
  - Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values.
Example Synthesized Attribute

- **actual_type**
  - A synthesized attribute associated with nonterminals: `<var>` and `<expr>`
  - It is used to store the actual type, int or real, of a variable or expression
  - For each variable, the actual_type is intrinsic
  - For expressions and assignments, the attribute is determined by the actual types of children nodes

Evaluation Order of Synthesized Attribute *actual_type*

- Parser tree of \( A = A + B \)
- \( A \) and \( B \) have type “real” or “int” according to the symbol table
Example Inherited Attribute

• **expected_type**
  – An inherited attribute associated with the nonterminal `<expr>`
  – It is used to store the expected type, either int or real
  – It is determined by the type of the variable on the left side of the assignment statement

Evaluation Order of Inherited Attribute **expected_type**

• The **expected_type** of `<expr>` is decided by the **actual_type** of the assignment's left side
**Attribute Grammar**

- Defines the attributes, and attribute evaluation rules mentioned in the example

**Example Attribute Grammar**

<table>
<thead>
<tr>
<th>Syntax Rule</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;assign&gt; -&gt; &lt;var&gt; = &lt;expr&gt;</code></td>
<td>R1. <code>&lt;expr&gt;.expected_type &lt; - &lt;var&gt;.actual_type</code></td>
</tr>
<tr>
<td><code>&lt;expr&gt; -&gt; &lt;var&gt;[1] + var[2]</code></td>
<td>R2. <code>&lt;expr&gt;.actual_type &lt; - if (&lt;var&gt;[2].actual_type = int) and (&lt;var&gt;[3].actual_type = int) then int else real end if; predicate: </code>&lt;expr&gt;.actual_type == <code>&lt;expr&gt;.expected_type</code></td>
</tr>
<tr>
<td><code>&lt;expr&gt; -&gt; &lt;var&gt;</code></td>
<td>R3. <code>&lt;expr&gt;.actual_type &lt; - </code>&lt;var&gt;.actual_type; predicate: <code>&lt;expr&gt;.actual_type == </code>&lt;expr&gt;.expected_type`</td>
</tr>
<tr>
<td>`&lt;var&gt; -&gt; A</td>
<td>B</td>
</tr>
</tbody>
</table>
Semantic Functions

• Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule
• Specify how attribute values are computed for \( S(X) \) and \( I(X) \)

Semantic Functions

• For a rule \( X_0 \rightarrow X_1 \ldots X_n \), the synthesized attributes of \( X_0 \) are computed with semantic functions of the form \( S(X_0) = f(A(X_1), \ldots, A(X_n)) \)
• The value of a synthesized attribute on a parse tree node depends only on the attribute values of the children node
Semantic Functions

• Inherited attributes of symbols $X_j$, $1 \leq j \leq n$, are computed with a semantic function of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$
• The value of an inherited attribute on a parse tree node depends on the attribute values of the node’s parent and siblings
• To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$

Revisit the Semantic Functions

<table>
<thead>
<tr>
<th>Syntax Rule</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\text{assign}&gt; \rightarrow &lt;\text{var}&gt; = &lt;\text{expr}&gt;$</td>
<td>1. $&lt;\text{expr}&gt;.\text{expected_type} \leftarrow &lt;\text{var}&gt;.\text{actual_type}$</td>
</tr>
<tr>
<td>$&lt;\text{expr}&gt; \rightarrow &lt;\text{var}&gt;[1] + &lt;\text{var}&gt;[2]$</td>
<td>2. $&lt;\text{expr}&gt;.\text{actual_type} \leftarrow \text{if} (&lt;\text{var}&gt;[2].\text{actual_type} = \text{int}) \text{ and } (&lt;\text{var}&gt;[3].\text{actual_type} = \text{int})$ then int else real end if</td>
</tr>
<tr>
<td>$&lt;\text{expr}&gt; \rightarrow &lt;\text{var}&gt;$</td>
<td>3. $&lt;\text{expr}&gt;.\text{actual_type} \leftarrow &lt;\text{var}&gt;.\text{actual_type}$</td>
</tr>
<tr>
<td>$&lt;\text{var}&gt; \rightarrow A \mid B \mid C$</td>
<td>4. $&lt;\text{var}&gt;.\text{actual_type} \leftarrow \text{look-up(&lt;var&gt;.string)}$ The look-up function looks up a given variable name in the symbol table and returns the variable’s type</td>
</tr>
</tbody>
</table>
Predicate Function

- A predicate function has the form of a Boolean expression on the union of the attribute set \( \{A(X_0), \ldots, A(X_n)\} \), and a set of literal attribute values.
- The only derivations allowed with an attribute grammar are those in which every predicate associated with every nonterminal is true.
- A false predicate function value indicates a violation of the syntax or static semantic rules.

Example Semantic Rules & Predicates

<table>
<thead>
<tr>
<th>Syntax Rule</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;assign&gt; \rightarrow &lt;var&gt; = &lt;expr&gt;)</td>
<td>R1. (&lt;expr&gt;.expected_type \leftarrow &lt;var&gt;.actual_type)</td>
</tr>
<tr>
<td>(&lt;expr&gt; \rightarrow &lt;var&gt;[1] + &lt;var&gt;[2])</td>
<td>R2. (&lt;expr&gt;.actual_type \leftarrow \text{if} (&lt;var&gt;[1].actual_type = \text{int}) \text{ and} (&lt;var&gt;[2].actual_type = \text{int}) \text{ then} \text{ int} \text{ else} \text{ real} \text{ end if}) (\text{predicate:} &lt;expr&gt;.actual_type == &lt;expr&gt;.expected_type)</td>
</tr>
<tr>
<td>(&lt;expr&gt; \rightarrow &lt;var&gt;)</td>
<td>R3. (&lt;expr&gt;.actual_type \leftarrow &lt;var&gt;.actual_type) (\text{predicate:} &lt;expr&gt;.actual_type == &lt;expr&gt;.expected_type)</td>
</tr>
<tr>
<td>(&lt;var&gt; \rightarrow A \mid B \mid C)</td>
<td>R4. (&lt;var&gt;.actual_type \leftarrow \text{look-up(&lt;var&gt;.string)}) The look-up function looks up a given variable name in the symbol table and returns the variable’s type</td>
</tr>
</tbody>
</table>