Announcements

- HW4 is due Monday after the break
  - We have office hours Mon morning
    - Lingxun: 9-11am
    - Ana: 11-1am
  - If you have questions, email us!

- Check your grades in HW Server

Last Class: Semantic Analysis

- Syntax analysis vs. static semantic analysis
- Static semantic analysis vs. dynamic semantic analysis
  - Languages differ in analysis they perform
    - C++: static, none (or very few) dynamic checks
    - Python: dynamic, none (or little) static checks
    - Java: a mixture of both
- Role of semantic analysis: prevent erroneous run-time behavior!

Today's Lecture Outline

- Static semantic analysis
  - Attribute grammars
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars

Semantic Analyzer

Attribute Grammars: Foundation for Static Semantic Analysis

- Attribute Grammars: generalization of Context-Free Grammars
  - Associate meaning with parse trees
  - Attributes
    - Each grammar symbol has one or more values called attributes associated with it. Each parse tree node has its own attributes; the 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 → E</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E + T</td>
<td>E.val := E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val := T.val</td>
</tr>
<tr>
<td>T → T * F</td>
<td>T.val := T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val := F.val</td>
</tr>
<tr>
<td>F → num</td>
<td>F.val := num.val</td>
</tr>
</tbody>
</table>

Example: Decorated parse tree for input 3 * 5 + 2 * 4

Building an Abstract Syntax Tree (AST)
- So far, we talked about parse trees
- In fact, compilers use **abstract syntax trees**
  - Suitable intermediate representation
    - Define semantic analyses over ASTs
    - AST is basis for translation into intermediate code
- 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

Building ASTs for Expressions

Exercise
- Show the parse tree and the AST for 2 * 3 * 4

So, 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>$T \rightarrow T_1 \ast F$</td>
<td>$T.nptr := \text{mknode}(\ast, T_1.nptr, F.nptr)$</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>

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

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

Constructing ASTs for Expressions

Input: $3 \ast 5 + 2 \ast 4$

| $E \rightarrow E_1 + T$ | $E.nptr := \text{mknode}(+, E_1.nptr, T.nptr)$ |
| $E \rightarrow T$ | $E.nptr := T.nptr$ |
| $T \rightarrow T_1 \ast F$ | $T.nptr := \text{mknode}(\ast, T_1.nptr, F.nptr)$ |
| $T \rightarrow F$ | $T.nptr := F.nptr$ |
| $F \rightarrow \text{num}$ | $F.nptr := \text{mkleaf}(\text{num}, \text{num}.val)$ |

Group Exercise

We know that the language $L = \text{a}^n\text{b}^n\text{c}^n$ 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 $R$ of each parse tree, such that $R.\text{ok}$ is true if and only if the string corresponding to the fringe of the tree is in $L$.

Question

Consider CFG and attribute grammar:

- $S \rightarrow E$ $\text{print}(E.val)$
- $E \rightarrow E_1 - T$ $E.val := E_1.val - T.val$
- $E \rightarrow T$ $E.val := T.val$
- $T \rightarrow \text{num}$ $T.val := \text{num}.val$

What is the result for $5-3-2$?
- Answer: 0

Decorated Parse Tree

Another LR grammar:

Input: $5 - 3 - 2$

Another Grammar

$T$ stands for term
$TT$ stands for term_tail

Now, the LL(1) version of same grammar:

- $E \rightarrow T TT$
- $TT \rightarrow T TT$
- $TT \rightarrow t$
- $T \rightarrow num$

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 write an attribute grammar in the “bottom-up” style we used so far?

**Attribute Grammar to Compute Value of Expressions (denote by AG3)**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow T ; TT$</td>
<td>(1) $TT_{\text{sub}} := T ; \text{val}$</td>
</tr>
<tr>
<td>$TT \rightarrow T ; TT$</td>
<td>(2) $E ; \text{val} := TT ; \text{val}$</td>
</tr>
<tr>
<td>$TT \rightarrow \varepsilon$</td>
<td>(1) $TT ; \text{val} := TT_{\text{sub}}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{num}$</td>
<td>(1) $T ; \text{val} := \text{num} ; \text{val}$ (provided by scanner)</td>
</tr>
</tbody>
</table>

**Attribute Flow**

- Attribute $TT_{\text{sub}}$ is computed based on parent $TT$ and sibling $T$: $TT_{\text{sub}} := T \; \text{val}$

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

**Synthesized and Inherited Attributes**

- **Synthesized attributes**
  - Attribute value computed from attributes of descendants in parse tree, and/or attributes of self
  - E.g., attributes $\text{val}$ in AG1, $\text{val}$ in AG3
  - E.g., attributes $\text{nptr}$ in AG2
- **Inherited attributes**
  - Attribute value computed from attributes of parent in tree and/or attributes of siblings in tree
  - E.g., attributes $\text{sub}$ 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 the 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?
  - In other words, in what order do we visit the nodes of the parse tree?
  - 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 \(A \rightarrow X_1 X_2 \ldots X_{j-1} X_j X_{j+1} \ldots X_n\) 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

Question

- An attribute grammar is **L-attributed** if each inherited attribute of \(X_j\) on the right-hand-side of \(A \rightarrow X_1 X_2 \ldots X_{j-1} X_j X_{j+1} \ldots X_n\) 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? Why not allow dependence on siblings to the right of \(X_j\), e.g., \(X_{j+1}\), etc.?
Evaluating Attributes and Attribute Flow

- S-attributed grammars
  - A very special case of attribute grammars
  - The 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

Semantic Analysis in Perspective

- Based on the theory of attribute grammars
- In compilers
  - E.g., translation into an AST, type checking, translation into intermediate code
  - Semantic analysis interleaved with LR parsing
  - Sometimes analysis is done separately on the AST

Semantic Analysis in Perspective

- Based on the theory of attribute grammars
- In software tools (e.g., Eclipse plug-ins)
  - Language extensions
    - Pluggable types: e.g., non-Null types
      - Catch null-pointer bugs or verify the absence of such bugs
    - Implement domain specific languages
  - Other static checkers and debuggers
  - Reverse engineering tools
  - Analysis is done on the AST