Announcements

Talk by Yan Shoshitaishvili at 4pm today

Title: How To Train Your Dragon: The Quest Towards (first) Mastery And (then) Automation in Cybersecurity

Time and place: 4pm in LOW 3051

- No office hour 4-5pm today

Announcements

- Quiz 4
- Exam 1 is graded
  - Grades available in Submitty
  - Average is 101 (81%)
  - About 60% in the A-B range
- HW4 is due Tuesday, October 16
- There is no HW5

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
- Attribute grammars: attributes and rules
- Role of semantic analysis: prevent erroneous run-time behavior!

Today’s Lecture Outline

- Static semantic analysis
  - Attribute grammars: attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars
- Recap: syntax analysis, semantic analysis, logic programming and Prolog

Semantic Analysis

Reading: Scott, Chapter 4.1-4.3

Semantic Analyzer

Semantic analyzer performs static semantic analysis on parse trees and ASTs. Optimizer performs static semantic analysis on intermediate 3-address code.
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: Decorated parse tree for input 3 * 5 + 2 * 4

- **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

- **Exercise**
  - Homework 4, Problem 4 grammar:
    \[ B \rightarrow \text{and} \quad B \quad \text{or} \quad B \quad | \quad \text{not} \quad B \quad | \quad \text{id} \]
  - Show the parse tree and the AST for
    \[ \text{and} \quad \text{and} \quad b \quad c \quad d \quad a \]
  - So, how do we construct syntax trees?
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 → E₁ + T</td>
<td>E.nptr := mknode(+, E₁.nptr, T.nptr)</td>
</tr>
<tr>
<td>E → T</td>
<td>E.nptr := T.nptr</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.nptr := mknode(*, T₁.nptr, F.nptr)</td>
</tr>
<tr>
<td>T → F</td>
<td>T.nptr := F.nptr</td>
</tr>
<tr>
<td>F → num</td>
<td>F.nptr := mkleaf(num, num.val)</td>
</tr>
</tbody>
</table>

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

mkleaf(num, num.val) creates a leaf node with label num, and a field containing the value of the number.

Constructing ASTs for Expressions

Input:

3 * 5 + 2 * 4

![AST Diagram]

Question

- What happens if we wrote a “bottom-up attribute flow” grammar?

```
E → T T₄
T₄ → T T₄
T → ε
T → num
```

```
E.nptr := mkleaf("+", E₁.nptr, T.nptr)
T₁.nptr := mknode(*, E₁.nptr, T.nptr)
F₁.nptr := mkleaf(T₁.nptr)
```

Unfortunately, this won’t work if we add `T₄ + T T₄`.

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

```
E → T T₄
T₄ → T T₄ | + T T₄ | ε
T → num
```

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → T T₄</td>
<td>(1) T₁.sub := T₁.val</td>
</tr>
<tr>
<td>T₄ → T T₄</td>
<td>(2) T₁.val := T₁.val</td>
</tr>
<tr>
<td>T → ε</td>
<td>(3) T₁.val := T₁.val</td>
</tr>
<tr>
<td>T → num</td>
<td>(4) T₁.val := T₁.val</td>
</tr>
</tbody>
</table>

```
F₁.nptr := mkleaf(T₁.nptr)
```

Attributes flow from parent to node, and from “siblings” to node!
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 → T TT`: `.val` of `T` is passed to sibling `TT`
  - In `TT → T TT`: `.val` of `T` is passed to sibling `TT`

---

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?
  - 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 → X_1, X_2, ..., X_{j-1}, X_j` depends only on
  - (1) the attributes of symbols to the left of `X_j`: `X_1, X_2, ..., 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 $A \rightarrow X_1 X_2 \ldots X_j X_{j+1} \ldots X_n$ 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 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 (partial sketch)

```plaintext
S → E $$
E → T TT
TT → + T TT | - T TT | ε
T → num
```

```plaintext
num S()
  case lookahead() of
  $\$: val = E(); match($$$); return val
  otherwise PARSE_ERROR

um E()
  case lookahead() of
  $\$: val = T(); return val
  otherwise PARSE_ERROR

um TT(num sub)
  case lookahead() of
    $\$: val = sub; return val
    + : match('+'); Tval = T(); val = TT(sub - Tval); return val
    $\$: val = sub; return val
    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

Semantic Analysis in Perspective

- Based on the theory of attribute grammars
- Type checking/inference
  - In compilers
    - E.g., AST, type checking/inference, translation into intermediate code
    - Typically interleaved with LR parsing
  - In software tools (e.g., Eclipse plug-ins)
    - E.g., pluggable types: e.g., retrofit types onto a dynamic language such as PHP

So Far

- Programming language syntax
- Programming language semantics
- Prolog: it is useful and important!
  - Declarative programming paradigm allows for easy prototyping in many domains!
  - Datalog
  - Sql
Moving On

- Functional programming
- Scheme
- Haskell
- Download DrRacket and set the language to R5RS