Syntactic Directed Translation

Attribute Grammars & Syntax-Directed Definitions

What is Attribute Grammar?
- A Context-Free Grammar Augmented with a Set of Rules
- Each Symbol in the derivation has a set of values, or Attributes
- The Rules specify How to Compute a value for each Attribute

Examples

Attribute Grammars

Add rules to compute the decimal value of a signed binary number

<table>
<thead>
<tr>
<th>Productions</th>
<th>Number</th>
<th>Sign</th>
<th>List</th>
<th>Bit</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number →</td>
<td>Sign List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign →</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List →</td>
<td>List Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit →</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Val →</td>
<td>Bit.val</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit List Sign Number

First Possible Evaluation Order
1. List pos.
2. Sign pos.
4. List val.
5. Bit val.
6. Number val.
Other orders are possible

Possible data-flow model for Evaluation
- Independent Attributes First
- Others in Order as Input Values become Available

Copyright 2011, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.

What is Syntax-Directed Translation?
- Translation Process guided by Context-Free Grammars
- Each Symbol in the derivation has a set of values, or Attributes
- The Rules specify How to Compute a value for each Attribute

Examples

Attribute Grammars

Add rules to compute the decimal value of a signed binary number

<table>
<thead>
<tr>
<th>Productions</th>
<th>Number</th>
<th>Sign</th>
<th>List</th>
<th>Bit</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number →</td>
<td>Sign List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign →</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List →</td>
<td>List Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit →</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Val →</td>
<td>Bit.val</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit List Sign Number

First Possible Evaluation Order
1. List pos.
2. Sign pos.
4. List val.
5. Bit val.
6. Number val.
Other orders are possible

Possible data-flow model for Evaluation
- Independent Attributes First
- Others in Order as Input Values become Available

Copyright 2011, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
Attributes, Rules and Evaluation Order

- Attributes:
  - Synthesized: Depend on Values from Children or Itself
  - Inherited: Depend on Values from Parent, Siblings or Itself

- How to Determine the Evaluation Order of Rules?
  - Construct the Parse Tree with Attributes
  - Build an Attribute Dependence Graph
  - Topological Sort It and Evaluate the Rules

Evaluation Methods

- Dynamic, dependence-based methods
  - Build the Parse Tree
  - Build the Dependence Graph
  - Topological Sort the Dependence Graph
  - Evaluate Attributes in Topological Order

- Rule-based methods
  - Analyze Rules at Compiler-Generation Time
  - Determine a Fixed (static) Ordering
  - Evaluate Nodes in that Order

- Oblivious Methods
  - Ignore Rules & Parse Tree
  - Pick a Convenient Order (at design time) & Use It
Inherited Attributes

Synthesized Attributes

If we show the computation ...
& then peel away the parse tree …
**Example**

All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance:

The dynamic methods sort this graph to find independent values, then work along graph edges.

The rule-based methods try to discover "good" orders by analyzing the rules.

The oblivious methods ignore the structure of this graph.

The dependence graph must be acyclic.

---

**Circularity**

We can only evaluate Acyclic Instances

- We can prove that some grammars can only generate instances with acyclic dependence graphs
- Largest such class is "strongly non-circular" grammars (SNC)
- SNC grammars can be tested in polynomial time
- Failing the SNC test is not conclusive

Many evaluation methods discover circularity dynamically

⇒ Bad property for a compiler to have

SNC grammars were first defined by Kennedy & Warren

---

**A Circular Attribute Grammar**

<table>
<thead>
<tr>
<th>Production</th>
<th>Attribution Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>List</td>
</tr>
<tr>
<td>List</td>
<td>List, Bit</td>
</tr>
<tr>
<td>Bit</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

**Circular Grammar**

Number → List

<table>
<thead>
<tr>
<th>Bit</th>
<th>List, Bit, Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>List1, Bit, 0</td>
</tr>
<tr>
<td>1</td>
<td>List2, Bit, 1</td>
</tr>
</tbody>
</table>

For "=101"

---

**Circular Grammar**

Number → List

<table>
<thead>
<tr>
<th>Bit</th>
<th>List, Bit, Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>List1, Bit, 0</td>
</tr>
<tr>
<td>1</td>
<td>List2, Bit, 1</td>
</tr>
</tbody>
</table>

For "=101"
### Circular Grammar for a Basic Block

**Grammar for a Basic Block**

**Assumptions:**
- Each operation has a COST
- Add them, bottom up
- Assume a load per value
- Assume no reuse via registers

**Simple problem for an Attributive Grammar:**

- \( \text{cost(div)} = 9 \)
- \( \text{cost(load)} = 2 \)
- \( \text{cost}() = 1 \)

### An Example: Estimating Cycle Counts

**Block**
- \( \text{Block} \rightarrow \text{Block Assign} \)
- \( \text{Assign} \rightarrow \text{Assign} \text{Expr} \)
- \( \text{Expr} \rightarrow \text{Expr} + \text{Term} \)
- \( \text{Term} \rightarrow \text{Term} * \text{Factor} \)
- \( \text{Factor} \rightarrow \text{Factor} \text{Exp} \)
- \( \text{Exp} \rightarrow \text{Exp} \text{Id} \)
- \( \text{Id} \rightarrow \text{Id} \text{Assign} \)
- \( \text{Assign} \rightarrow \text{Assign} \text{Expr} \)
- \( \text{Expr} \rightarrow \text{Expr} \text{Term} \)
- \( \text{Term} \rightarrow \text{Term} \text{Factor} \)
- \( \text{Factor} \rightarrow \text{Factor} \text{Exp} \)
- \( \text{Exp} \rightarrow \text{Exp} \text{Id} \)
- \( \text{Id} \rightarrow \text{Id} \text{Assign} \)

**Value flow from AHI to LASS is: prod**

\[
\begin{align*}
\text{Block} & \rightarrow \text{Block Assign} \\
\text{Assign} & \rightarrow \text{Assign} \text{Expr} \\
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
\text{Term} & \rightarrow \text{Term} * \text{Factor} \\
\text{Factor} & \rightarrow \text{Factor} \text{Exp} \\
\text{Exp} & \rightarrow \text{Exp} \text{Id} \\
\text{Id} & \rightarrow \text{Id} \text{Assign} \\
\end{align*}
\]

---

**An Example (continued)**

- \( \text{expr,Id} = 1 \)
- \( \text{cond} = 2 \)
- \( \text{cost} = 5 \)
- \( \text{cost} = 9 \)
Properties of the Example Grammar

- All Attributes are Synthesized
- Rules can be Evaluated Bottom-up in a Single Pass
  - Good fit to bottom-up, shift/reduce parser
- Easily Understood Solution
- Seems to fit the Problem Well

An Interesting Example - Tracking Loads

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded
- Assumes an Infinite Register Set to Hold Variables

A Better Execution Model

Adding load tracking

- Need Sets Before and After for each production
- Must be initialized, updated, and passed around the tree

These copy rules multiply rapidly
Each creates an instance of the set
Lots of work, lots of space, lots of rules to write
### An Even Better Model

What about accounting for Finite Register Sets?
- **Before & After** must be of Limited Size
- Adds complexity to **Factor** — **Identifier**
- Requires more complex initialization

- Jump from Tracking Loads to Tracking Registers is small
- Copy rules are already in place
- Some local code to perform the Allocation

### The Moral of the Story

- **Non-local** computation needed lots of supporting rules
- **Complex local** computation was relatively easy

### The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - Need copies of attributes
  - Can use pointers, for even more cognitive overhead
- Result is an attributed tree *(somewhat table points)*
  - Must build the parse tree
  - Either search tree for answers or copy them to the root

### Reworking the Example

#### (with load tracking)

#### Syntax-Directed Definitions

- Augment `CFG` with Attributes and Rules
- Inherited and Synthesized Attributes

#### Summary

- **Attribute Grammar**
  - Augment `CFG` with Attributes and Rules
  - Inherited and Synthesized Attributes

- **Syntax-Directed Definitions**
  - Find Dependence Graph and Evaluation Order
  - Useful for Semantic Analysis