Static semantics of languages

K. Stirewalt
CSE 891 Fall 2004

Static semantics

Not all properties can be checked by a parser
Practical limitations
- Listing all possible combinations of C declaration modifiers would obfuscate the grammar
Theoretical limitations
- Checking whether a variable has been declared before it is used is a context-sensitive property that cannot be checked by a context-insensitive parser
Another mechanism must be provided
- Typically provided by a post-parsing pass of the AST

Attribute grammars (Knuth)

Extension of BNF to specify some context-sensitive properties of a language and some semantics, e.g., type checking
- Each non-terminal has an associated set of attributes (named, typed values)
- Productions have functions added to them to assign values to the attributes
- Walk the parse tree, evaluating the rules at each node

Attribute

Variable associated with a symbol in a grammar
- Can store data to facilitate semantic analysis
- Can store state information to control parsing
In principle, an attribute could just copy the input text associated with its node in the AST and pass it up the tree. The root node could then do all of the processing
- Hence, attribute grammars are powerful enough to do anything any other compilation technique can do
An algorithm exists to test for circularity of attribute grammars

Kinds of attributes

Attributes are either synthesized or inherited, but not both
- Synthesized attributes derive their values from the attributes of child nodes in the parse tree. That is, the attribute's non-terminal is on the left-hand side of a rule, and the function for that rule gives it a value
  \[ A_x \rightarrow B_y C_z \quad x \leftarrow f(y, z) \]
- Inherited attributes derive their values from the attributes of parent nodes in the AST. The attribute's non-terminal is on the right-hand-side of a rule, and the function for that rule gives it a value
  \[ A_x \rightarrow B_y C_z \quad y \leftarrow g(x); z \leftarrow h(x) \]

Well-formed rules

A given rule must be unambiguously expressed in such a way that all values are well defined
The values computed by a set of rules for a given production must be independent of the order in which the rules are evaluated
The set of rules for the grammar can be recursive but may not have any circular dependencies
Part of the job in designing an attribute grammar is specifying the tree walk that will evaluate the parse trees
Example: Parsing Fortran strings

In older versions of Fortran, string literals were expressed in the following format:
- `<stringLiteral>` ::= `<numeral> "H` `<charSeq>`
- The `"H` stands for Hollenth, one of the early contributors to the language
- Example:
  - ` IO FORMAT (I2, XH, I2)`

Problem: Grammar recognizes
- `E` =
  - `E` = `I`
  - `E` = `<charSeq>`
  - `E` = `<numeral>`

Solution using attributed grammar

```plaintext
stringLit -> numeral "H" charSeq.
  Size[charSeq] = Val[numeral]
numeral -> digit
  Val[numeral] = Val[digit]
  | numeral, digit
    Val[numeral] = Val[numeral] * 10 + Val[digit]
charSeq -> char
  Size[charSeq] = 1
  | charSeq, char.
    Size[charSeq] = Size[charSeq] + 1
```

Exercise

Using a parse tree, demonstrate that the solution correctly checks the example literal

Question

What would the grammar look like if we used an inherited attribute to enforce the length restriction on the `charSeq`?

Answer

```plaintext
stringLit -> numeral "H" charSeq.
  Size[charSeq] = Val[numeral]
numeral -> digit
  Val[numeral] = Val[digit]
  | numeral, digit
    Val[numeral] = Val[numeral] * 10 + Val[digit]
charSeq -> char
  Size[charSeq] = 1
  | charSeq, char.
    Size[charSeq] = Size[charSeq] + 1
```

Attribute grammars and semantics

Attribute grammars can specify syntax that is context-sensitive

Also useful for specifying *translational semantics*
- E.g., procedure for evaluating expressions
- E.g., Translation of statements in a language into some machine representation

Technique: Invent attributes to represent the target of translation
- E.g., expression values
- E.g., sequences of machine instructions
Grammar for Boolean numerals

B → '0'.
B → '1'.
W → B.
W → WB.
N → W.
N → W₁ '·' W₂.

Legend:
• B represents "bits"
• W represents "words", i.e., sequences of bits.
• N represents "numerals"

Try parsing the following input:
1101.01

Parse tree

Evaluation

Computing the value of a Boolean numeral:
– Input is a character string
– Output is a decimal number

Parse tree:
– Leaves denote lexemes (i.e., '0', '1', and '·')
– Roots denote non-terminal phrases

Computation:
– Assign value to each leaf
– Synthesize values for roots while walking up tree

Evaluation (continued)

Each boolean digit (B) will contribute a value that is determined by its position in its word
– This position is called the scale
– Knowing position requires knowing length of the word

Each word (W) will synthesize a value by adding the values of its constituent bits

The numeral (N) will synthesize a value by adding the values associated with its two constituent words (if the numeral contains a radix point)

Attributes

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>V</td>
<td>synthesized</td>
<td>value</td>
</tr>
<tr>
<td>B</td>
<td>S</td>
<td>inherited</td>
<td>scale</td>
</tr>
<tr>
<td>W</td>
<td>L</td>
<td>synthesized</td>
<td>length</td>
</tr>
<tr>
<td>W</td>
<td>S</td>
<td>inherited</td>
<td>scale</td>
</tr>
<tr>
<td>N</td>
<td>V</td>
<td>synthesized</td>
<td>value</td>
</tr>
</tbody>
</table>

Rules

| B → '0'     | V(B) ← 0  |
| B → '1'     | V(B) ← 2^SB |
| W → B       | V(W) ← V(B); S(B) ← S(W); L(W) ← 1 |
| W₁ → W₂ B   | V(W₁) ← V(W₂) + V(B); S(B) ← S(W₁); S(W₂) ← S(W₁) + 1; L(W₁) ← L(W₂) + 1 |
| N → W       | V(N) ← V(W); S(W) ← 0 |
| N → W₁ '·' W₂ | V(N) ← V(W₁) + V(W₂); S(W₁) ← 0; S(W₂) ← L(W₁) - L(W₂) |
General solution

Consider the general expression:
<left operand> <op> <right operand>

Assume code for operands leaves result in accumulator
Expression implemented using code template:
code for <left operand>
STO Tm+1>, e.g., if m=0, this is T1
code for <right operand>
STO Tm+2>
LOAD Tm+1>
OpCode Tm+2>; OpCode depends on <op>

Example: Code generation

Grammar for binary arithmetic expressions
integerExpr → term
| integerExpr weakOp term.
weakOp → '·' | '·'.

Machine language:
- Implicit accumulator register
- LOAD <label> ; load accumulator with word at location <label>
- STO <label> ; store accumulator into word at location <label>
- SUB <label> ; subtract from accumulator word at <label>
- ADD <label> ; add to accumulator word at <label>

Solution

integerExpr → term
Temp [term] ← Temp [integerExpr]
| integerExpr, weakOp term.
Code [integerExpr] ←
concat Code [integerExpr],
optimize (Code [term],
Temp [integerExpr],
Temp [integerExpr] ← Temp [integerExpr] + 1
weakOp → '·',
| '·',