Chapter 4 - Semantic Analysis

June 2, 2015

The role of the semantic analyzer

- Compilers use semantic analysis to enforce the static semantic rules of a language
- It is hard to generalize the exact boundaries between semantic analysis and the generation of intermediate representations (or even just straight to final represenations); this demarcation is the logical boundary between the front-end of a compiler (lexical analysis and parsing) and the back-end of the compiler (intermediate representations and final code.)

The role of the semantic analyzer

- For instance, a completely separated compiler could have a well-defined lexical analysis and parsing stage generating a parse tree, which is passed wholesale to a semantic analyzer, which could then create a syntax tree and populate a symbol table, and then pass it all on to a code generator;
- Or a completely interleaved compiler could intermix all of these stages, literally generating final code as part of the parsing engine.

The role of the semantic analyzer

- The text focuses on an organization where the parser creates a syntax tree (and no full parse tree), and semantic analysis is done over a separate traversal of the syntax tree.

Dynamic checks

- The Tony Hoare observation about disabling semantic checks being akin to a sailing enthusiast who wears a life jacket when training on dry land, but removes it when going to sea.
- Assertions, invariants, preconditions, and post-conditions: let the programmer express logical requirements for values. (If you continue along this route, you see run into thickets of various ideas about formalizing semantics, such as operational semantics, axiomatic semantics, and denotational semantics.)

Static analysis

- "In general, compile-time algorithms that predict run-time behavior are known as *static analysis*."
- In Ada, ML, and Haskell, type checking is static. (An amusing, related paper: Dynamic Typing in Haskell).

Static analysis

- Language design is where you make the decisions that drive what can be statically checked.

Attribute grammars

- Attribute grammars add two concepts to a CFG:
  - Semantic rules (also called "semantic actions" or "grammar rules")
  - Attributes; these are values associated with terminals and non-terminals. Synthesized attributes are derived from children; inherited attributes come a parent or a sibling; L-attributes ("L" from "Left-to-Right, all in a single pass") come either from a left sibling or from the parent.

Evaluating attributes

- The process of assigning values to a tree is called *annotation* or *decoration*.

Evaluating attributes in an S-attributed grammar

- If all of the attributes in an attribute grammar are synthesized (i.e., derived from children), then the attributed grammar is said to be "S-attributed".
- All of the rules assign attributes only to the left-hand side (LHS) symbol, and all are based on the set of attribute values of the right-hand side (RHS) symbols.

Evaluating attributes that are inherited

- If any of the attributes in an attribute grammar are inherited (i.e., a LHS symbol which has an attribute derived either from the RHS or from a left sibling symbol), then the attributed grammar is said to be “L-attributed”.

Evaluating attributes that are inherited

- Consider the following (Lemon syntax rather than the text's syntax):

```
expr(A) ::= CONST(B) exprtail(C).
  {C.st = B.val; A.val = C.val}
exprtail(A) ::= MINUS CONST(B) exprtail(C).
  {C.st = C.st - B.val; A.val = C.val}
exprtail(A) ::= .
  {A.val = A.st}
```

Evaluating attributes that are inherited

```
expr(A) ::= CONST(B) exprtail(C).
  {C.st = B.val; A.val = C.val}
exprtail(A) ::= MINUS CONST(B) exprtail(C).
  {C.st = C.st - B.val; A.val = C.val}
exprtail(A) ::= .
  {A.val = A.st}
```

- Consider the inherited actions: In the first line, exprtail (a LHS symbol) has its subtotal ("st") attribute set to the value of its left sibling, CONST. In the second line, the LHS exprtail's subtotal is reduced by CONST's value. In the third line, exprtail has its value derived from its subtotal.

Upshot

- Practical programming languages generally require some L-attributed flow.

Attribute flow

- As a notation, grammars are *declarative* and do not imply an ordering; a grammar is well-defined iff the rules determine a unique set for each and every possible parse tree. A grammar is noncircular if no attribute depends on itself.
- Your text defines a translation scheme is an algorithm that annotates a tree using the rules of an attribute grammar in an order consistent with the tree's attribute flow. (See, however, pp.37-40 of the Dragon Book for a slightly different take.)

Attribute flow

- Clearly, a S-attributed grammar can be decorated in the same order as a LR parser, allowing a single pass that interleaves parsing and attribute evaluation.
- Equally clearly, an L-attributed grammar can be decorated in the same order as LL parser, allowing a single pass that interleaves parsing and attribute evaluation.

Attribute flow is not just “LR <-> S-attribute”

- Page 188: “. . . it is often possible for a bottom-up parser to accommodate some non-S-attributed attribute flow. . . ”

One pass compilation

- Compilers that interleave both semantic analysis and target code generation are said to be one-pass compilers. (The text notes on page 189 that some authors also allow intermediate code generation to count as single-pass.)

Building a syntax tree

- A parser can use semantic actions to just build a syntax tree, and then use a separate semantic analyzer to decorate the tree.

Building a syntax tree

- Nice feature of S-attributed grammars is that the LR parser's semantic actions are very straightforward (see pseudo-code in figure 4.5 on page 190):

Building a syntax tree

Figure 4.7 from text

Building a syntax tree

- Using LL is not as simple since syntax tree information has to flow from left siblings to right siblings (see figure 4.6 on page 191). You end up with both an inherited "syntax tree" (st) attribute and a tree pointer:

Building a syntax tree

Figure 4.8 from text

Action Routines

- Page 191: "Most production compilers, however, use an ad hoc, handwritten translation scheme, interleaving parsing with at least the initial construction of a syntax tree, and possibly all of semantic analysis and intermediate code generation."
- In LL parsing, one can embed action routines anywhere on the RHS; if you are writing a recursive descent parser, the action routine at the beginning of a production rule is placed at the beginning of the implementing subroutine, and so on.

Action Routines

- In LR parsing, in general, one can always put action routines at the end of a production rule (indeed, Lemon only allows this placement), though various tools such Bison allow more "flexibility". (See here for more about mid-rule-actions in Bison.)

Space managment for attributes

- If you are building a tree, you can use those nodes to hold the attribute information.
- If you don't build a tree, then for bottom-up parsing with an S-attributed grammar, one can use an attribute stack mirroring the parse stack.
- For top-down parsing, while you can use a stack, it's not simple. It's probably just better to build a tree, though, or at least have an automated tool keep up with items. (See ANTLR or Coco/R)

Decorating a Syntax Tree

- As mentioned earlier, the focus in this text is having the parser create a syntax tree and then using a separate stage for semantic analysis and intermediate code generation.
- Tree grammars augmented with semantic rules are used to decorate syntax trees, analogous to the way that context-free grammars augmented with semantic rules can create decorated parse trees.
- Generally, these are implemented with mutually recursive subroutines.
- For instance, take a look at the compiler passes for GCC 4.1

Example of S-attributed grammar

Example here