Class 25: Semantic Actions

Held: Friday, 28 October 2011

Summary: Today we consider ways to extend parse trees so that they can be used to compute relevant values.

Related Pages:
- EBoard.
- Reading: Aho et al., 5.1-5.2.
- Due: Project, Phase 2: Parser.

Notes:
- We have a visitor today and Monday.
- An issue from the current homework: How do you represent Boolean expressions?
- A few of you asked for extensions until 6pm Saturday. I am happy to grant that extension to anyone who wants one.
- Next week’s Thursday extra is on Tuesday: Study abroad opportunities in Budapest.
- I will be out of town next Friday. I will provide a lab for your edification. One of my colleagues may supervise that lab or you may do it on your own.
- Project, Phase 3, assigned.

Overview:
- Adding attributes to parse trees.
- Example: Evaluating and typing expressions.
- Computing the values of attributes.
- Example: Sequences of assignments

Attributes
- As you know, the parse tree is only one step on the way to compilation of a source text.
- What do we do next? Typically, we add attributes to the nodes of the parse tree.
- An attribute is a computed value associated with a node in the parse tree.
- What can attributes be computed from? Typically values of “neighboring” nodes (children, parent, siblings).
- How are they computed? Using a rule that is associated with a production in the language.
- Grammars augmented with rules are called attribute grammars.
An Attribute Grammar for Evaluating expressions

- Let’s consider how we might compute a value and type type of arithmetic expressions as they are parsed.
- What attributes will we need?
  - A value attribute (associated with expressions, terms, factors, and numbers) is clearly necessary.
  - In order to compute the values of variables (identifiers), we’ll need to look them up in a symbol table. We’ll leave that as a subject for future research (perhaps Monday’s class).
- Some rules are fairly trivial, as they only involve copying attributes

```plaintext
exp : term
    exp.value = term.value
    exp.type = term.type
term : factor
    term.value = factor.value
    term.type = factor.type
factor : LPAREN exp RPAREN
    factor.value = exp.value
    factor.type = exp.type

For the base case of numbers, we simply get the value of the number (we assume that this is computed during the lexical analysis phase)

factor : NUMBER
    factor.value = NUMBER.value
    factor.type = NUMBER.type

Alternately

factor : INTEGER
    factor.value = INTEGER.value
    factor.type = integer
factor : REAL
    factor.value = REAL.value
    factor.type = real

If all that we have for tokens is the “string representation”, we will need to convert it

factor : NUMBER
    factor.value = stringToNumber(NUMBER.string)

Similarly, for the base case of identifiers, we look up the type and value of the identifier. (What should we do if it has not been initialized?)

factor : ID
    factor.type = lookupType(symbols, ID)
    factor.value = lookupValues(symbols, ID)

Addition and subtraction are also fairly easy
\[
\text{exp}_0 : \text{exp}_1 + \text{term} \\
\text{exp}_0.\text{value} = \text{exp}_1.\text{value} + \text{term}.\text{value} \\
\text{exp}_0.\text{type} = \text{mostGeneral}\left(\text{exp}_1.\text{type}, \text{term}.\text{type}\right)
\]

\[
\text{exp}_0 : \text{exp}_1 - \text{term} \\
\text{exp}_0.\text{value} = \text{exp}_1.\text{value} - \text{term}.\text{value} \\
\text{exp}_0.\text{type} = \text{mostGeneral}\left(\text{exp}_1.\text{type}, \text{term}.\text{type}\right)
\]

- If we use yacc or bison and pay attention to only values, the rules will look like

\[
\text{exp} : \text{exp} \ ' + ' \ \text{exp} \quad \{ \$$ = \$1 + \$3; \}
\]

\[
\text{exp} : \text{exp} \ ' - ' \ \text{exp} \quad \{ \$$ = \$1 - \$3; \}
\]

- If we treat MULTIPLY and DIVIDE as tokens, we might write something like

\[
\text{mulop} : \text{MULTIPLY} \\
\text{mulop}.\text{fun} = (\lambda (x \ y) \ (\times x \ y))
\]

\[
\text{mulop} : \text{DIVIDE} \\
\text{mulop}.\text{fun} = (\lambda (x \ y) \ (\div x \ y))
\]

- We can then add a rule for terms

\[
\text{term}_0 : \text{term}_1 \ \text{mulop} \ \text{factor} \\
\text{term}_0.\text{value} = \text{mulop}.\text{fun}(\text{term}_1.\text{value}, \text{factor}.\text{value}) \\
\text{term}_0.\text{type} = \text{mostGeneral}(\text{term}_1.\text{type}, \text{factor}.\text{type})
\]

**Attributes, Revisited**

- We’ve seen one mechanism for computing two kinds of attribute: values and types. Let’s now look more broadly at attributes.
- What kinds of attributes can be computed? A wide variety. We typically compute
  - symbol tables to use in type checking and correctness analysis;
  - values of expressions, as we’ve just seen;
  - abstract syntax trees that correspond to the structure underlying the parse tree. Such ASTs help eliminate problems of parse trees introduced by changes to the syntax required by parsing techniques;
  - some compilers use attributes to compute code for the parse tree.
- How are the rules applied? In the abstract, one can say that a topological sort of the dependencies is done and the rules are applied in order.
  - If you don’t know what a topological sort is, ask someone who took 301.
- In practice, a rule is often applied when the corresponding production is “executed” by the parser.
  - This limits the types of rules that are allowed.
  - In an LR parser, attributes must by synthesized: computed from the attributes of the children or earlier symbols.
  - In an LL parser, attributes can also be inherited: computed from attributes of the parents or left siblings.
- In practice, rules can also do things other than compute values. This means that the behavior of the parser depends on the order in which the rules are applied. This is a bad thing, and you should avoid rules that have such side effects.
Symbol Tables

- For the “compute a value from other values” aspect of the grammar, the value attribute is synthesized, computed only from children.
- But if we also need to make values depend on previously computed values, the values will be inherited.
- Consider a simple extension to the extension grammar in which we use expressions either in assignment statements or print statements.

```scheme
s : statement_list
statement_list : statement ';’ statement
| epsilon
statement : : assignment_statement
| print_statement
assignment_statement : ID ASSIGN exp
print_statement : PRINT ID
```

- Now let’s consider how to associate values with identifiers.
- We’ll start by adding a function, `valueOf`, that maps every assigned identifier to its value.
  - We generally implement that function as a table and call it a “symbol table”.
  - I’ve been teaching enough Scheme that I want to think of it as a function.
- Print statements can use that table

```scheme
print_statement : PRINT ID
  display Printstatement.valueOf(ID.name))
```

- Where does it get that function? Presumably, from its parent node.

```scheme
statement : print_statement
  print_statement.value_of = statement.value_of
```

- Where does the statement get the function? Presumably, from its enclosing statement list.

```scheme
statement_list_0 : statement ';’ statement_list_1
  statement.valueOf = statement_list_0.valueOf + statement.list_1.valueOf
```

- Where does the statement list get the function? From a combination of prior statements.

```scheme
S : statement_list
  statement_list.valueOf = (lambda (x) ’undefined)
  statement_list_0 : statement ';’ statement_list_1
  statement_list_0.valueOf =
    extend(statement.list_0.valueOf,statement.extension)
```

- You can fill in much of the rest.