Topic 5: semantic analysis

5.3 Ordering Attribute Evaluation
   5.3.2 Single Pass Approaches

Contents
   5.3.2.1 Evaluation during Bottom-Up Parsing
   5.3.2.2 Evaluation during Top-Down Parsing

Topic 5: semantic analysis

5.4 Special Forms of Attribute Grammars I:
   S-Attributed Grammars
Special Forms of Attribute Grammars

- We have seen that the evaluations of attributes is complex, potentially requiring passing of attribute values from one part of a tree to another.
- A one-pass compiler, however, would require the attributes of the current node of the parse tree to be resolved now.
- This is not possible with attribute grammars as defined earlier.
- We thus introduce 2 restricted kinds of attribute grammars, for use in single-pass compilers:

1. S-attributed grammars (today)
2. L-attributed grammars (tomorrow)

S-Attributed Grammars

- For bottom-up parsing, when we create a parse-tree node (through a reduce operation), its children are known, but the parent is not.
- If all attributes were of the synthesised kind, then we could evaluate all attributes of the node as it is created.
- We can thus restrict the attribute grammar to use only synthesised attributes.

**S-attributed grammar:** An attribute grammar that uses synthesized attributes exclusively
Semantic analysis
S-Attributed Grammars

Propagating synthesised attributes while parsing

• Reducing by: \[ \text{Expr} ::= \text{id} \{ \text{Expr.t} = \text{id.t} \} \]

\[ \text{id} \quad \text{t:int} \quad + \quad \text{id} \quad \text{t:int} \]

\[ \text{Expr} \quad \text{t:int} \]
\[ \text{id} \quad \text{t:int} \quad + \quad \text{id} \quad \text{t:int} \]
Semantic analysis
S-Attributed Grammars

Propagating synthesised attributes while parsing
• Reducing by: $\text{Expr} \::= \text{id} \{ \text{Expr}.t = \text{id}.t \}$

```
Expr $t$:int
  id $t$:int + id $t$:int
```

Semantic analysis
S-Attributed Grammars

Propagating synthesised attributes while parsing
• Reducing with: $\text{Expr} \::= \text{Expr}_1 + \text{Expr}_2 \{ \text{Expr}.t = \text{Expr}_1.t \}$

```
Expr $t$:int
  Expr $t$:int + Expr $t$:int
```

id $t$:int + id $t$:int
S-attributed grammars – overly restrictive

- It is possible to write grammars for real programming languages which use only synthesised attributes.
- Note however that for most languages, this will require substantial rewriting of the grammar, into a fairly unnatural form.
- For example, take the grammar for declaration statements:

```
Decl → Mode IDList ;
Mode → int
Mode → real
IDList → IDList, id
IDList → id
```

S-attributed grammars

Most natural way of writing the semantic rules

```
Decl → Mode IDList ; {IDList.type = Mode.dtype}
Mode → int {Mode.dtype = int}
Mode → real {Mode.dtype = real}
IDList → IDList_1, id {addSymbTable(id.token, IDList.type)}
IDList → id {addSymbTable(id.token, IDList.type)}
```
Semantic analysis

S-Attributed Grammars

S-attributed grammars – overly restrictive

\[\text{Decl} \rightarrow \text{Mode} \, \text{IDL} \text{st} ; \quad \{\text{IDL}.\text{type} = \text{Mode}.\text{dtype}\}\]

\[\text{Mode} \rightarrow \text{int} \quad \{\text{Mode}.\text{dtype} = \text{int}\}\]

\[\text{Mode} \rightarrow \text{real} \quad \{\text{Mode}.\text{dtype} = \text{real}\}\]

\[\text{IDL} \rightarrow \text{IDL}, \text{id} \quad \{\text{addSymbTable}(\text{id}.\text{token}, \text{IDL}.\text{type})\}
\]

\[\text{IDL} \rightarrow \text{id} \quad \{\text{addSymbTable}(\text{id}.\text{token}, \text{IDL}.\text{type})\}\]

\[\text{id}, \text{id} ;\]
Rewriting unrestricted AG to be S-attribute Grammar

- We can rewrite the semantic rules to work using only synthesised attributes.

\[
\text{Decl} \rightarrow \text{Mode IDList ;}
\]

\[
\{ \text{addAllToSymbTable}(\text{IDList.tokens, Mode.dtype} \}
\]

\[
\text{Mode} \rightarrow \text{int} \quad \{ \text{Mode.dtype} = \text{int} \}
\]

\[
\text{Mode} \rightarrow \text{real} \quad \{ \text{Mode.dtype} = \text{real} \}
\]

\[
\text{IDList} \rightarrow \text{IDList}, \text{id}
\]

\[
\{ \text{IDList.tokens} = \text{append} (\text{IDList\_1.tokens}, \text{id.token}) \}
\]

\[
\text{IDList} \rightarrow \text{id}
\]

\[
\{ \text{IDList.tokens} = \text{list} (\text{id.token}) \}
\]
S-Attributed Grammar: Summary

- S-Attributed grammars allow simpler processing in bottom-up parsing, since, at any point in parsing, we can derive all the attributes of the parent node from those of the already recognised children.

- However, they require more careful writing of the semantic rules to avoid use of inheritance.

- Next: a less restricted AG: L-Attributed Grammars
L-attributed grammars

• As well as synthesised attributes, L-attributed grammars allow limited inheritance:
  • Given a parse tree node generated by production:
    \[ P \rightarrow C_1 \cdot C_2 \ldots C_n \]
  • Any \( C_k \) can have inherited attributes if and only if that attribute depends on:
    1. attributes of the other child nodes to its left (e.g. it can use attributes from nodes \( C_1 \sim C_{k-1} \))
    2. inherited attributes of \( P \) itself.

• The important point here is that information flows only left-to-right, bottom-up, or top-down: never right-to-left!
Semantic analysis

L-Attributed Grammars

L-attributed grammars: restatement

• Basically, a node can only derive attribute information:

  1. From below (as with S-attributed grammars)
  2. From its left siblings (and only if synthesised)
  3. From the parent

  NEVER from siblings on the right!


Semantic analysis

L-Attributed Grammars

L-attributed grammars & Recursive Descent evaluation

• Lets return to the recursive descent process of attribute evaluation:
  1. Start at the top node of the tree
  2. On entering a node, execute all evaluation rules for all inherited attributes (whether from parent or from a left sibling)
  3. Call this procedure recursively on each child of this node
  4. On return, execute the evaluation rules for synthesised attributes (passing info from children to parent)

• Previously, this routine might need to be repeated multiple times until all attributes set.
• With L-Attributed grammars, one pass guaranteed to set all attributes.
• Basically, once we start evaluating a node, we can be sure that all sisters to the left are fully calculated.
L-attributed Grammars & S-attributed Grammars

- NOTE: an S-attributed grammar is a kind of L-attributed grammar, as all attributes meet condition (i):

L-attributed grammar: An attribute grammar that uses either:

i) synthesized attributes

ii) inherited attributes where the inherited attribute depends only on:

a) attributes of the other child nodes to its left

b) inherited attributes of P itself.

L-Attributed Grammars

L-attributed Grammars & Bottom-Up Analysis

- In a one-pass bottom-up compiler, L-attributed grammars are problematic:
  - Synthesised attributes are no problem because the children are always recognised before the parent
  - Attributes inherited from synthesised attributes of siblings are no problem because those children have also been constructed.
  - However, parent nodes are created only AFTER all children in a production have been processed, so attributes inherited from the parent cannot be calculated when needed.
  - However, it can be done, by careful rewriting.
  - YACC generates LALR(1) parsers which do this!
Topic 5: semantic analysis

5.6 Embedded Actions

Actions intermixed with Symbols

• Some recursive descent (top-down) systems depend on placing semantic actions between each symbol on the RHS, e.g.,

\[ A :- M_1 \{ \text{action1} \} M_2 \{ \text{action2} \} M_3 \{ \text{action3} \} \]

• For example:

\[ E :- T \{ R.i := T.val \} \quad R \{ E.val := R.s \} \]

• Often Presented as:

\[ E :- T \quad \{ R.i := T.val \} \]
\[ R \quad \{ E.val := R.s \} \]
**Semantic analysis**

**Embedded Actions**

**Benefit of actions between symbols**

- The benefit of this approach in top down analysis is that:
  - If semantic actions only performed at the completion of a production, then semantic attributes for each element will only be available once all constituents recognised.
  - By placing actions between symbols, attributes can be calculated before we start analysing a constituent, and can thus be passed down to the constituent’s children.

```
E :- T  {R.i := T.val}
R  {E.val := R.s}
```

**Actions between symbols in Bottom-Up analysis**

- In an LR parser, a given parsing state may be processing several alternative productions, e.g.,

```
S
E :: (E.)
E :: (E.+T)
```

- If the current parse stack has symbols:

```
Stack: ( E
```

…then it is not clear if we are working on E :- (E+T) or E :- (E)

- Only when we ‘reduce’ is the decision made as to which rule is intended.
- THUS, we cannot perform actions placed between RHS symbols!
Rewriting grammars to allow actions between symbols

- We can rewrite grammars with actions between symbols to work with LR() parsers.
- Basically, for each action between symbols:
  - We create an artificial symbol
  - We provide a lambda rule for that symbol with the action at its end

Old:

```
E ::= T [R.i := T.val]
R [E.val := R.s]
```

New:

```
E ::= T X R [E.val := R.s]
X ::= λ [R.i := T.val]
```

- Now, the action is performed when the lambda rule is reduced!

Rewriting grammars to allow actions between symbols

- Note that the semantic rule of X specifies symbols from the production for E.
- The compiler must keep track of the symbol context for each lambda rule.
- YACC does this automatically.

```
E ::= T X R [E.val := R.s]
X ::= λ [R.i := T.val]
```

- In some cases, this is not needed:

```
Prog ::= begin CST Decls Stmts end
CST ::= λ {clearSymbTable()}
```
5.7 Summary

Summary of Semantic Analysis

- An Attribute Grammar is an extension of a CFG, with:
  1. Attributes associated with each Symbol
  2. Rules associated with each CFG production to move attributes about
- Attributes can be inherited or synthesised
- Semantic rules can access global data structures, such as a symbol table.
Semantic analysis

Summary

Summary of Semantic Analysis (ii)
• In a multiple-pass compiler, one can:
  1. Build the parse tree
  2. Calculate dependencies between attributes
  3. Find the topological sort of the attributes
  4. Execute the attribute evaluation rules in that order

OR

• Use a recursive descent approach to resolving attributes on parse tree nodes.

Semantic analysis

Summary

Summary of Semantic Analysis (iii)
• A single-pass compiler utilises restricted attribute grammars to enable attribute evaluation while constructing the parse tree:
  • S-attributed grammars suite bottom-up analysis
  • L-attributed grammars suite top-down parsing
  • L-attributed grammars can be used in bottom-up parsing with careful writing
Topic 5: semantic analysis

5.8 Examples

Semantic analysis

Examples

List Depth

List → ( IList ) { IList.depth = IList.depth + 1 ;
                 print “Depth:”, List.depth }

IList → IList₁ ( IList₂ ) { if (IList₁.depth > IList₂.depth):
                          IList.depth = IList₁.depth
                        else:
                          IList.depth = IList₂.depth + 1 }

IList → λ { IList.depth = 0 }

Draw the parse tree and calculate attributes for:

((() (()) ())))
Semantic analysis

L-Attributed Grammars

L-Attributed Grammars & Top-Down Analysis

• Assume a single-pass compiler, where semantic actions are performed as the parse tree is constructed,
  • In a top-down parser, there is no problem, as the growing of the tree exactly matches the recursive-descent ordering of the evaluation rules. E.g.,
  • When we expand the start symbol ‘E’ with production ‘D :: T L’, then we can apply the rules which pass attributes from D to T and L.
  • We next expand T using ‘T :: int’
  • As ‘int’ is a terminal, we then return to the T node, and can now pass any synthesised attributes from ‘int’ up to T.
  • As ‘T’ is now finished, we pass synthesised attributes up to D
  • Moving on to T’s sister, L, it can be given any synthesised attributes of T
  • We then construct the subtree for L.
  • Etc.