Abstract Syntax: What’s the Point?

CIS 461/561, Winter 2005

Direct Interpretation

• Many “little languages” embed an interpreter directly in parser actions
  – example: simplecalc example in Cup distribution

expr := expr1 PLUS expr2
  {: RESULT = new Integer(expr1.intValue() + expr2.intValue()); :}
|  expr1 MINUS expr2
  {: RESULT = new Integer(expr1.intValue() - expr2.intValue()); :}
|  expr1 TIMES expr2
  {: RESULT = new Integer(expr1.intValue() * expr2.intValue()); :}
| ...

Direct Translation

• We could just as well embed code generation directly in parser actions
  – “Single pass compilation”

  – Fine for very simple transformations in which information is available in the right order.

  – “Right order”? Looking at attribute grammars will help us talk about order of producing and using information; we’ll do that later.

Two (or more) passes: building an intermediate representation

We want:

• Simplicity (remove extraneous detail)
• Regularity (simplify processing)
• Suitability for next processing steps
  – Binding, type checking, maybe code generation
Some Options ...

• Build the parse tree (concrete syntax)
• Build a more “abstract” tree
• Build something else
  – “Target tree” (closer to machine)
  – Abstract machine code (linear)
  – Control flow graph
  – Single assignment form
  – ...

Why not the parse tree?

• Messy
  – Unnecessary stuff from the concrete syntax, like parentheses
  – Unnecessary stuff added to concrete syntax to help parsing
    • (e.g., extra productions to resolve dangling else)
• Irregular
  – More than one way to say something; complicates further processing

Abstract Syntax

• Regularize:
  - int i, j; $\rightarrow$ int i; int j;
• Simplify:
  - 3*(i+k) $\rightarrow$ Times(Plus(i,k))
• Organize:
  - Use node types to prepare for binding, type checking

Attribute Grammars

A way of thinking about what can be done during parsing, and during each pass over the tree

• Associate attributes with each terminal and non-terminal symbol in the grammar
  • Attribute = any kind of information that may be produced during one production and used in another: Types, object code sequences, etc.
• Associate attribute equations with productions
  • Attribute equation = use of attributes of some symbols to compute attributes for other symbols
A bit of history ...

- Language syntax systematized
  - with context-free languages, Backus-Naur form*, and parsing algorithms
- Syntax-directed compilation framework
  - attempts to describe semantics and translation in the same framework (through grammar)
  - Attribute grammars [Knuth 1968] describe types, values, other attribute through equations associated with grammar productions

Attributes and Denotation

- Meaning through structure, e.g.,
  - \([[ A \text{ "+" } B ]] = [[ A ]] \text{ + } [[ B ]]\)
    - where "+" is in syntactic domain, + is in semantic domain
- Generalizes from values to other attributes
  \(\text{exp}_1 ::= \text{exp}_2 \text{ "+" } \text{exp}_3\)
  \{:
    \text{exp}_1.\text{type} = \text{addtype}(\text{exp}_2.\text{type}, \text{exp}_3.\text{type});
    \}

Why Attribute Grammars?

- Rules are local to productions
  - A way to organize our thinking and our code
- We can reason about classes of grammars
  - Systematic creation of (hand-coded) evaluators
- Creation of automatic attribute evaluators
  - More popular in mid-80s than now; due for a resurgence?

Inherited and Synthesized

\(\text{Exp}_0 ::= \text{Exp}_1 \text{ "+" } \text{Exp}_2\)
\{:
  \text{Exp}_0.\text{type} = \text{type_add}(\text{Exp}_1.\text{type}, \text{Exp}_2.\text{type});
  \text{Exp}_1.\text{limit} = \text{Exp}_0.\text{limit};
  \}

- “Type” is synthesized attribute (bottom-up)
- “Limit” is inherited attribute (top-down)
  - More realistic examples require a mix in which inherited values depend on synthesized values
Inherited and Synthesized

\[
\text{Exp: Add}
\]

\[
\text{type} \quad \text{(synthesize)}
\]

\[
\text{limit} \quad \text{(inherit)}
\]

\[
\text{Exp}_0 ::= \text{Exp}_1 + \text{Exp}_2
\]

\[
\{:
\text{Exp}_0.\text{type} = \text{type}\_\text{add}(\text{Exp}_1.\text{type}, \text{Exp}_2.\text{type});
\text{Exp}_1.\text{limit} = \text{Exp}_0.\text{limit};
:}
\]

Beyond Inherited & Synthesized

- \(\text{Exp}_0 ::= \text{“let” Exp}_1 \text{ “in” Exp}_2 \text{ “end”}\)
- \(\{: \text{Exp}_0.\text{value} = ?? :\}\)
- Meaning of \(\text{Exp}_2\) depends on \(\text{Exp}_1\)
  - Value is neither inherited nor synthesized, but it can be evaluated left-to-right
- This grammar is (or could be) \textit{L-attributed}
  - Allow synthesized attributes in lhs, left-to-right flow of attributes in rhs of production

L-Attribution and Simple Assignment Form

\[
L ::= M \ N \ <\text{act}> \\
\]

- Non-trivial actions are associated with an action symbol \(<\text{act}>\) (the rest are “copy rules”)
- For L-attributed grammar
  - Synthesized attr. of \(L\) depend \textit{only} on its inher. attr. of \(L\) and on synth. attr. of \(M, N, <\text{act}>\)
  - Synthesized attributes of \(M\) may depend only on inherited attributes of \(L\)
  - Synthesized attributes of \(N\) may depend on any attributes of \(M\); similarly for \(<\text{act}>\)
  - No cycles (outputs of \(<\text{act}>\) depend only on inputs)

L-attributed Grammar Properties

- An L-attributed grammar can be evaluated on-the-fly during LL(1) parsing
  - but not during shift-reduce parsing; why?
- An L-attributed grammar can be evaluated during one depth-first traversal of an abstract syntax tree
  - even if AST was built with an S-attribution, e.g., by CUP action routines
Evaluating L-Attribution

Designing Attribute Equations

- Use synthesized attributes where possible
  - Simplest, most uniform to evaluate, either on-the-fly or during AST traversal
- Use inherited or left-to-right evaluation where necessary
- If L-attribution is not possible:
  - Analyze attribute dependence for multi-pass evaluation

Attribute Grammar as Design Step

- Even without AG-based tools ...
- First sketch evaluation as attribute grammar
  - What can be done with S-attribution? What requires L-attribution? Any inherited attributes? Any cycles?
  - Break cycles with multiple attributes
- Translate attribute evaluation to Java code
  - In depth-first traversals of AST

Example: Name Resolution

- The environment (symbol table) flows left to right; a reference is (often, not always) “resolved” to a link to the declaration sub-tree
  - This can be non-trivial when there is overloading; note binding may be partly or fully dynamic (e.g., OO dispatch)
Why No Tools? (1)

• Historically: Theory => Tools
  • CFGs and parsing theory gave us Yacc, etc.
  • Attribute grammars gave us ???
• Where are the tools?
  • Yacc/Bison/CUP/... action routines are bare-bones mechanism for on-the-fly evaluation
  • Synthesizer Generator (mid-80s) ... died out with syntax-directed editors
• Time for a resurgence?
  • E.g., type checkers from type inference rules? We know how to do it, if we agreed on a representation.

Aside: What you should know

• Distinguish synthesized from inherited attributes
• Recognize S-attribution, L-Attribution
  – pure bottom-up vs. left-to-right, depth-first
• Recognize and break circularity

Why No Tools? (2)

• Whatever happened to attribute grammars?
  • Introduced in 68, efficient tools produced in 80s; disappearing from compiler texts
• Main obstacles
  – Memory (for multi-pass evaluation)
    • Until recently, only academic toy compilers built a complete AST
    • No longer an issue?
  – Ad hoc link between concrete and abstract syntax

Organizing an AST

• Problem: Two dimensions of modularity
  – OO AST can capture one; hard to cut both ways
• Node classification dimension
  – e.g., “multiplication” is subclass of “binop”
  – Nicely captured in OO structure
• Compiler phase dimension
  – Type checking, improvement, code generation
  – Orthogonal to OO structure; even with “visitor” pattern, OO AST does not effectively localize change (e.g., adding an optimization pass).
What To Do?

- Simple languages / compilers:
  - One AST structure, multiple walks over it
    - Maybe with visitor pattern, maybe not
- Bigger languages / compilers:
  - A sequence of representations:
    - AST (for binding and type-checking), then
    - Flow graphs OR register transfer lists OR ...
      - Target-oriented representation of operational semantics, plus memory layout