CSE 401 - Compilers

Lecture 14: Semantic Analysis, Part II

Michael Ringenburg

Winter 2013

Agenda

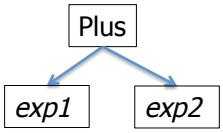
- Attribute Grammars
  - Review what we discussed at the end of class Wednesday
  - Go over a couple examples
- Symbol tables
  - For MiniJava
  - For real languages
- Next lecture: type checking

Review: Attributes

- An attribute grammar associates "attributes" (data) with nodes in the syntax tree.
- Given a production X ::= Y_1 Y_2 … Y_n
  - A *synthesized* attribute X.a is a function of some combination of attributes of Y_i's (bottom up)
    - E.g., the val attribute from last time
  - An *inherited* attribute Y_i.b is a function of some combination of attributes X.a and other Y_j.c (top down)
    - Often restricted a bit: only Y's to the left can be used.
    - E.g., a "type environment"

Review: Attribute Equations

- For each production we give a set of equations relating attribute values of the syntax tree node and its children
  - Example:



plus.val = exp1.val + exp2.val

- *Attribution* (aka, evaluation) means finding a solution that satisfies all of the equations in the tree

Attribute Example: Execution Cost

- Consider the simple grammar:

  Block :: = Block Assign | Assign
  Assign ::= id = Expr;
  Expr ::= Expr + Term | Expr – Term | Term
  Term :: = Term * Factor | Term / Factor | Factor
  Factor ::= (Expr) | INTEGER_LITERAL | id

- Goal: design an attribute grammar that approximates the execution cost of programs written in this language.
- The attribute X.cost will estimate the cost of computing the subtree rooted at node X.

Block Attribute Equations

- Intuition: The cost of a block is the sum of the cost of the statements that comprise it.
- $Block_1$ ::= $Block_2$ Assign
  - $Block_1$.cost = $Block_2$.cost + Assign.cost
  - Recursive case
- Block ::= Assign
  - Block.cost = Assign.cost
  - Base case

Assign Attribute Equation

- Intuition: The cost of an assignment is the cost of computing the right-hand side, plus the cost of a store (to write the result to the variable in memory).
- Assign ::= id = Expr;
  - Assign.cost = Cost(store) + Expr.cost

Expression Attribute Equations

- Intuition: Cost of an operation is the cost of computing its operands, plus the cost of the actual operation (add or subtract instruction).
- $Expr_1$ ::= $Expr_2$ + Term
  - $Expr_1$.cost = Cost(add) + $Expr_2$.cost + Term.cost
- $Expr_1$ ::= $Expr_2$ - Term
  - $Expr_1$.cost = Cost(subtract) + $Expr_2$.cost + Term.cost
- Expr ::= Term
  - Expr.cost = Term.cost

Term Attribute Equations

- Intuition: Cost of an operation is the cost of computing its operands, plus the cost of the actual operation (multiply or divide).
- $Term_1$ ::= $Term_2$ * Factor
  - $Term_1$.cost = Cost(mult) + $Term_2$.cost + Factor.cost
- $Term_1$ ::= $Term_2$ / Factor
  - $Term_1$.cost = Cost(divide) + $Term_2$.cost + Factor.cost
- Term ::= Factor
  - Term.cost = Factor.cost

Factor Attribute Equations

- Factor ::= (Expr)
  - Factor.cost ::= Expr.cost
  - The cost of the parenthesized expression.
- Factor ::= id
  - Factor.cost ::= Cost(load)
  - Reading a variable has the cost of a load
- Factor ::= INTEGER_LITERAL
  - Factor.cost ::= Cost(loadImm)
  - The cost of a load-immediate style instruction (place a constant into a register).

Example: Cost Attribution

```
x = y + 1; y = x * 3;
```

Observation

- If we had y = x + x + 1; z = x; a real compiler wouldn't load x three times. It would load it once and store it in a register.
- How can we track this, since attribute equations are defined locally (based on a single production)?
- Solution is to add "copy" rules to the grammar, but this can blow up grammar, and is one the reasons attribute grammars aren't used much in practice.
  - See "Improving the Execution-Cost Estimator" in 4.3.3 of Cooper & Torczon.

More Realistic Example of Attribute Rules

- Suppose we have the following grammar for a trivial language

```
program ::= declList stmt
declList ::= declList decl | decl
decl ::= int id;
stmt ::= exp = exp ;
exp ::= id | exp + exp | INTEGER_LITERAL
```

- Programs are a list of declarations, followed by a single assignment statement
- Let's give suitable attributes for basic type and lvalue/ rvalue checking, and constant folding

More Realistic Example of Attribute Rules

- Attributes of nodes
  - env (type environment) stores the types of all declared variables; synthesized by declarations, inherited by the statement
    - Each entry maps a name to its type
  - envPre (for declarations) – Used to build up the environment
    - Represents the environment prior to the declaration.
    - E.g., "int x; int y;". The envPre of "int y" will map x to an int. The env of "int y" will map x to int and y to int.
  - type (for expressions); synthesized from children (and possible env lookup)
  - kind: var (assignable) or val (not assignable); synthesized
  - value (for expressions): UNK (unknown) or an Integer, represents computed constant value; synthesized

Attributes for Declarations

- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}
  - Intuition: add (id, int) mapping to an environment containing mappings for previous declarations
- Example: Attribution for `int y`, given that we previously saw `int x`
  - Saw `int x` earlier, so assume decl.preEnv = {(x, int)}
  - decl :: = `int y;`
  - decl.env = decl.preEnv U {(y, int)} =
    {(x, int)} U {(y, int)} =
    {(x, int), (y, int)}

Attributes for Declarations

- $declList_1$ ::= $declList_2$ decl
  - decl.preEnv = $declList_2$.env
  - $declList_1$.env = decl.env
  - Intuition: $declList_2$.env contains all of the previously seen mappings, so use it as the pre-environment for our new declaration. The environment for the combined list (list 1) will be the result of adding the mapping for decl to the mappings of the prefix list (list 2).

Attributes for Declarations

- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
  - Intuition: For the first element in our declaration list, we can start with an empty environment, because we won't have seen any declarations yet. (True here, but probably not in a real language.)

Example Declaration List

```
int x; int y; int z;
```

- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- $declList_1$ ::= $declList_2$ decl
  - decl.preEnv = $declList_2$.env
  - $declList_1$.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Attributes for Program

- program ::= declList stmt
  - stmt.env = declList.env
  - Intuition: We want to typecheck our statement given the type environment synthesized by our declaration list.
- Example: If program was

```
int a; int b; b = a + 1;
```

We would typecheck the assignment statement with the environment {(a, int), (b, int)}

Attributes for Constants

- exp ::= INTEGER_LITERAL
  - exp.kind = val
  - exp.type = int
  - exp.value = INTEGER_LITERAL
  - Intuition: An integer constant (literal) clearly has type int, and explicit value. You can't assign to it (5 = x is not legal), so it is a value (val) not a variable (var).

Attributes for Identifier Expressions

- exp ::= id
  - id.type = exp.env.lookup(id)
    - If this lookup fails, issue an undeclared variable error.
  - exp.type = id.type
  - exp.kind = var
  - exp.value = UNK
  - Intuition: We look up the identifier's type in the environment, and use that as the expression's type. If it doesn't exist in the environment, it must not have been declared, so it's an error. Since it is a variable, it is assignable and has unknown value.
  - Example: Typechecking a with environment {(a,int)} gives type int. Typechecking b with the same environment gives an error.

Attributes for Addition

- exp ::= exp_1 + exp_2
  - exp_1.env = exp_2.env = exp.env
  - error if exp_1.type != exp_2.type (or if not compatible if using more complex rules)
  - exp.type = exp_1.type (or converted type if more complex rules)
  - exp.kind = val
  - exp.value = (exp_1.value == UNK || exp_2.value == UNK) ?
    UNK : exp_1.value + exp_2.value
  - Intuition: Typecheck operands with same environment as operation. Verify that types are compatible, and set result type appropriately. Not assignable, so set kind to val. Compute value if both operands have constant value.

Attribute Rules for Assignment

- stmt ::= exp_1 = exp_2;
  - exp_1.env = stmt.env
  - exp_2.env = stmt.env
  - Error if exp2.type is not assignment compatible with exp1.type
  - Error if exp_1.kind is not var (can't be val)
  - Intuition: Verify that left hand side is assignable, and that types of left and right hand sides are compatible.

Example

int x; int y; int z; x+1 = a + (1 + 2);

Example

```
int x; int y; int z; x = a + (1 + 2);
```

Example

```
int x; int y; int z; x = y + (1 + 2);
```

Extensions

- This can be extended to handle sequences of statements, and multiple declaration lists
  - Full environment is passed down to all statements and expressions
  - Declaration lists extended to have a pre-environment, which they pass to the first declaration via
    - declList ::= decl
      - decl.preEnv = declList.preEnv
    - $declList_1$ :: = $declList_2$ decl
      - $declList_2$.preEnv = $declList_1$.preEnv

Observations

- These are equational computations
- This can be automated (if equations are non-circular)
- But implementation problems
  - Non-local computation: Attribute equations can only refer to values associated with symbols that appear in a single production rule.
    - If you need non-local values, need to add special rules to the grammar to copy them. Can make grammar very large.
  - Can't afford to literally pass around copies of large, aggregate structures like environments.
  - Use of production rules binds attributes to the parse tree rather than the (typically smaller, and more useful) AST. Can work around this (use "AST grammar"), but results in more complex attribute rules.

In Practice

- Attribute grammars give us a good way of thinking about how to structure semantic checks
  - What to verify or track at each node
- Symbol tables will hold environment information
- Add fields to AST nodes to refer to appropriate attributes (symbol table entries for identifiers, types for expressions, etc.)
  - Put in appropriate places in AST class inheritance tree (most statements don't need types, for example)

Symbol Tables

- Map identifiers to <type, kind, location, other properties>
- Operations
  - Lookup(id) => information
  - Enter(id, information)
  - Open/close scopes
- Build & use during semantics pass
  - Build first from declarations
  - Then use to check semantic rules
- Use (and add to) during later phases as well

Aside: Implementing Symbol Tables

- Big topic in classical compiler courses: implementing a hashed symbol table
- These days: use the collection classes that are provided with the standard language libraries (Java, C#, C++, ML, Haskell, etc.)
  - Then tune & optimize if it really matters
    - In production compilers, it really matters
      - Up to a point …
        - » (if you can read this, I'm impressed).
- Java:
  - Map (HashMap) will handle most cases
  - List (ArrayList) for ordered lists (parameters, etc.)

Symbol Tables for MiniJava

- Consider this a general outline, based on recommendations courtesy from Hal Perkins (whose given this project many times).
  - Feel free to modify to fit your needs
- A mix of global and local tables.
- First Global – Per Program Information
  - Single global table to map class names to per-class symbol tables
    - Created in a pass over class definitions in AST
    - Used in remaining parts of compiler to check class types and their field/method names and extract information about them

Symbol Tables for MiniJava

- Other Globals – Per Class Information
  - 1 Symbol table for each class
    - 1 entry per method/field declared in the class
      - Contents: type information, public/private/protected, parameter types (for methods), storage locations (filled in later), etc.
  - In full Java, need multiple symbol tables (or more complex symbol table) per class
    - Ex.: Java allows the same identifier to name both a method and a field in a class – multiple namespaces

Symbol Tables for MiniJava

- Global (cont)
  - All global tables persist throughout the compilation
    - And beyond in a real compiler…
      - (e.g., symbolic information in Java .class or MSIL files, link-time optimization information in gcc)
      - Cray compilers generate "program libraries", which contain full symbols tables and full post-front-end IR for every function in every module.
        - » Can use this for interprocedural optimization across source files (modules). Traditionally, each module compiled and optimized individually into a .o/.class file (containing object- or byte-code).

Symbol Tables for MiniJava

- 1 local symbol table for each method
  - 1 entry for each local variable or parameter
    - Contents: type information, storage locations (filled in later), etc.
  - Needed only while compiling the method; in a single pass compiler, you could discard when done with the method
    - But if type checking and code gen, etc. are done in separate passes, this table needs to persist until we're done with it
    - Your project implementation will likely be multipass

Beyond MiniJava

- What we aren't dealing with: nested scopes
  - Inner classes
  - Nested scopes in methods – reuse of identifiers in parallel or inner scopes; nested functions
- Conceptual idea: keep a stack of symbol tables (pointers to tables, really)
  - Push a new symbol table when we enter an inner scope
  - Look for identifier in inner scope; if not found look at the element above it in the stack, recursively.
  - Pop symbol table when we exit scope

Engineering Issues

- In multipass compilers, symbol table info needs to persist after analysis of inner scopes for use on later passes
  - So popping can't "really" delete the scope's table.
  - Keep around with pointer to parent scope. Effectively creates an upside tree of scopes (nodes have parent pointers rather than children pointers). Statements have pointers to their innermost scope.
- May want to retain O(1) lookup
  - Not O(depth of scope nesting) – although some compilers just assume this will be small enough to not matter.
  - Compilers that care may use hash tables with additional information to get the scope nesting right.

Error Recovery

- What to do when an undeclared identifier is encountered?
  - Only complain once (Why?)
  - Can forge a symbol table entry for it once you've complained so it will be found in the future
  - Assign the forged entry a type of "unknown"
  - "Unknown" is the type of all malformed expressions and is compatible with all other types
    - Allows you to only complain once! (How?)

"Predefined" Things

- Many languages have some "predefined" items (functions, classes, standard library, …)
- Include initialization code or declarations in the compiler to manually create symbol table entries for these when the compiler starts up
  - Rest of compiler generally doesn't need to know the difference between "predeclared" items and ones found in the program
  - Possible to put "standard prelude" information in a file or data resource and use that to initialize
    - Tradeoffs?

What's coming up?

- Type checking!
- x86 overview, from the perspective of a compiler targeting x86 assembly.
- And a midterm!