CSE 582 – Compilers

Static Semantics
Hal Perkins
Autumn 2002

Agenda
- Static semantics
- Types
- Attribute grammars
- Representing types
- Symbol tables
  - Reminder about Java container classes
  - “Predefined” things

What do we need to know to compile this?

class C {
  int a;
  C(int initial) {
    a = initial;
  }
  void setA(int val) {
    a = val;
  }
}
class Main {
  public static void main() {
    C c = new C(17);
    c.setA(42);
  }
}

Beyond Syntax
- There is a level of correctness that is not captured by a grammar
  - Has a variable been declared?
  - Are types consistent in an expression?
  - In the assignment x=y, is y assignable to x?
  - Does a method call have the right number and types of parameters?
  - In a selector p.q, is q a method or field of class p?
  - Is variable x guaranteed to be initialized before it is used?
  - Etc. etc. etc.

What else do we need to know to generate code?
- Where are fields allocated in an object?
- How big are objects?
- Where are local variables stored when a method is called?
- Which methods are associated with an object/class?

Types
- Role of types in programming languages
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (operator overloading, for example)
Semantic Analysis

- Some key ideas
  - Extract types and other information from the program
  - Check language rules that go beyond the grammar
  - Assign storage locations (later)
  - Key data structures: symbol tables
    - For each identifier in the program, record its attributes (kind, type, etc.)

Some Kinds of Semantic Information

<table>
<thead>
<tr>
<th>Information</th>
<th>Generated From</th>
<th>Used to process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol tables</td>
<td>Declarations</td>
<td>Expressions, statements</td>
</tr>
<tr>
<td>Type information</td>
<td>Declarations, expressions</td>
<td>Operations</td>
</tr>
<tr>
<td>Constant/variable information</td>
<td>Declarations, expressions</td>
<td>Statements, expressions</td>
</tr>
<tr>
<td>Register &amp; memory locations</td>
<td>Assigned by compiler</td>
<td>Code generation</td>
</tr>
<tr>
<td>Values</td>
<td>Constants</td>
<td>expressions</td>
</tr>
</tbody>
</table>

A Sampling of Semantic Checks (0)

- Name: id
  - id has been declared and is in scope
  - Result type of id is its declared type
  - Memory location assigned by compiler
- Constant: v
  - Result type and value are explicit

A Sampling of Semantic Checks (1)

- Binary operator: exp₁ op exp₂
  - exp₁ and exp₂ have compatible types
    - Identical, or
    - Well-defined conversion to a common type
  - Result type is a function of the operator and operands

A Sampling of Semantic Checks (2)

- Assignment: exp₁ = exp₂
  - exp₁ is assignable (not a constant or expression)
  - exp₁ and exp₂ have compatible types
    - Identical, or
    - exp can be converted to exp₂ (e.g., char to int), or
    - Type of exp is a subclass of type of exp₂ (can be decided at compile time)
  - Result type is type of exp₁
  - Location where value is stored is assigned by the compiler

A Sampling of Semantic Checks (3)

- Cast: (exp₁) exp₂
  - exp₁ is a type
  - exp₂ either
    - Has same type as exp₁
    - Can be converted to type exp₁ (e.g., double to int)
    - Is a subclass of exp₁ (usually requires a runtime check)
  - Result type is exp₂
A Sampling of Semantic Checks (4)

- Field reference \( \text{exp}_1.\text{exp}_2 \)
  - The type of \( \text{exp}_1 \) is a class
  - \( \text{exp}_2 \) is an identifier
  - \( \text{exp}_1 \) has a field or method named \( \text{exp}_2 \)
  - Result type is declared type of \( \text{exp}_2 \)

A Sampling of Semantic Checks (5)

- Method call \( \text{m}(\text{e}_1, \text{e}_2, \ldots, \text{e}_n) \)
  - The method has \( n \) parameters
  - Each argument has a type that can be assigned to the associated parameter
  - Result type is given by method declaration (or is void)

Semantic Analysis

- Parser builds abstract syntax tree
- Now need to extract semantic information and check constraints
  - Can sometimes be done during the parse, but often easier to organize as a separate phase
  - And some things can't be done on the fly during the parse, e.g., information about identifiers that are used before they are declared (fields, classes)
- Information stored in *symbol tables*
  - Generated by semantic analysis, used later

Attribute Grammars

- A systematic way to think about semantic analysis
- Sometimes used directly, but even if ad hoc techniques are used, AGs are a useful guide to organizing the analysis

Attribute Example

- Assume that each node has an attribute .val
- AST and attribution for \((1+2) \times (6/2)\)
Inherited and Synthesized Attributes

- Given a production $X ::= Y_1 Y_2 \ldots Y_n$
- A **synthesized** attribute is $X.a$ is a function of some combination of attributes of $Y_i$’s (bottom up)
- An **inherited** attribute $Y_i.b$ is a function of some combination of attributes $X.a$ and other $Y_j.c$ (top down)

Informal Example of Attribute Rules (1)

- Attributes for simple arithmetic language
- Grammar
  - $program ::= decl \ stmt$
  - $decl ::= int \ id;$
  - $stmt ::= exp = exp ;$
  - $exp ::= id \mid exp + exp \mid 1$

Informal Example of Attribute Rules (2)

- Attributes
  - $env$ (environment, e.g., symbol table); inherited by $stmt$, synthesized by $decl$
  - $type$ (expression type); synthesized
  - $kind$ (variable [lvalue] vs value [rvalue]); synthesized
  - $expectedtype$ (type required); inherited

Attributes for Declarations

- $decl ::= int \ id;$
- $decl.env = \{\text{identifier, int, var}\}$

Attributes for Program

- $program ::= decl \ stmt$
- $stmt.env = decl.env$

Attributes for Constants

- $exp ::= 1$
  - $exp.kind = val$
  - $exp.type = int$
Attributes for Expressions

- \( \text{exp} ::= \text{id} \)
  - \( \text{id.type} = \text{exp.env.lookup(id)} \)
  - \( \text{exp.type} = \text{id.type} \)
  - error if \( \text{id.type} \neq \text{exp.expectedtype} \)
  - \( \text{exp.kind} = \text{id.kind} \)

Attributes for Addition

- \( \text{exp} ::= \text{exp}_1 + \text{exp}_2 \)
  - \( \text{exp}_1.env = \text{exp.env} \)
  - \( \text{exp}_2.env = \text{exp.env} \)
  - \( \text{exp}_1.expectedtype = \text{exp.expectedtype} \)
  - \( \text{exp}_2.expectedtype = \text{exp.expectedtype} \)
  - error if \( \text{exp}_1.type \neq \text{exp}_2.type \)
  - \( \text{exp.type} = \text{exp}_1.type \)
  - \( \text{exp.kind} = \text{val} \)

Attribute Rules for Assignment

- \( \text{stmt} ::= \text{exp}_1 = \text{exp}_2; \)
  - \( \text{exp}_1.env = \text{stmt.env} \)
  - \( \text{exp}_2.env = \text{stmt.env} \)
  - \( \text{exp}_2.expectedtype = \text{exp}_1.type \)
  - error if \( \text{exp}_1.kind \neq \text{var} \)

Example

- \( \text{int } x; \ x = x + 1; \)

Extensions

- This can be extended to handle sequences of declarations and statements
  - Sequence of declarations builds up combined environment with information about all declarations
  - Full environment is passed down to statements and expressions

Observations

- These are equational (functional) computations
  - In principle, this could be automated, provided the attribute equations are non-circular
- Problems
  - Non-local computation
  - Can’t afford to literally pass around copies of large, aggregate structures like environments (i.e., copy rules)
In Practice

- Attribute grammars give us a good way of thinking about how to structure semantic checks
- Use symbol tables to hold environment information
- Add fields to AST nodes for common attributes (expression types, symbol table entries for identifiers, etc.)
- Put in appropriate places in inheritance tree – statements don’t need types, for example

Symbol Tables

- Map identifiers to <type, location, other properties>
- Operations
  - Lookup(id) => information
  - Enter(id, information)
  - Open/close scopes

Aside: Implementing Symbol Tables in Java

- Classic topic in compiler course: implementing a hashed symbol table
- These days: use the Java collection classes (or equivalent in C#, C++, etc.)
  - Map (HashMap) will solve most of the problems
  - List (ArrayList) for ordered lists (parameters, etc.)

Symbol Tables for JFlat (1)

- Global
  - 1 Symbol table per class
    - 1 entry for each method/field
      - Contents: type information, public/private, storage locations (later), etc.
      - In full Java, multiple symbol tables per class since methods and fields can have the same names in a class

Symbol Tables for JFlat (2)

- Global (cont)
  - Single global table to map class names to class symbol tables
    - Created in pass over class definitions
    - Used in remaining parts of compiler to check field/method names and extract information
    - All global tables persist throughout the compilation
    - And beyond in a real Java or C# compiler...

Symbol Tables for JFlat (3)

- Local symbol table for each method
  - 1 entry for each local variable or parameter
    - Contents: type information, storage locations (later), etc.
  - Needed only while compiling the method; can discard when done
Symbol Tables Beyond JFlat

- What we aren’t dealing with: nested scopes
- Inner classes
- Nested scopes in methods – reuse of identifiers in parallel or (if allowed) inner scopes
- Basic idea: new symbol table for inner scopes, linked to surrounding scope’s table
  - Look for identifier in inner scope; if not found look in surrounding scope (recursively)
  - Pop back up on scope exit

Engineering Issues

- In practice, want to retain O(1) lookup
  - Use hash tables with additional information to get the scope nesting right
- In multipass compilers, symbol table information needs to persist after analysis of inner scopes

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 to avoid redundant error messages

“Predefined” Things

- JFlat, like all other languages has some “predefined” items
  - Class JFSystem, in our case
  - Include startup code in the compiler to create symbol table entries for these
  - Rest of compiler generally doesn’t need to know the difference between “predeclared” items and ones found in the program

Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, float, char
- CompoundConstructed Types
  - Built up from other types (recursively)
  - Constructors include arrays, records/structs/classes, pointers, enumerations, functions, modules, ...

Type Representation

- Create a shallow class hierarchy
  - abstract class Type { ... }
  - class ClassType extends Type { ... }
  - Should not need too many of these
Base Types
- For each base type (int, boolean, others in other languages), create a single object to represent
  - Use references to these objects to to represent these types
  - Useful to create a “void” type to tag functions that do not return a value

Compound Types
- Basic idea: represent with an object that refers to component types
  - Limited number of these – correspond directly to type constructors in the language (record/struct, array, function, ...)

Class Types
- class Id { fields and methods }
- class ClassType extends Type {
  Type parentClassType; // ref to base class
  Set fields; // type info for fields
  Set methods; // type info for methods
}
  - (Note: may not want to do this literally, depending on how you chose to represent symbol tables for classes.)

Array Types
- For Java this is simple: only possibility is # of dimensions and element type
- class ArrayType extends Type {
  int nDims;
  Type elementType;
}

Array Types for Pascal
- Pascal allows arrays to be indexed by any discrete type
  - array[indexType] of elementType
- Element type can be any other type, including an array
  - class PascalArrayType extends Type {
    Type indexType;
    Type elementType;
  }

Functions/Methods
- Type of a function is its result type plus an ordered list of parameter types
  - class MethodType extends Type {
    Type resultType; // type or “void”
    List parameterTypes;
  }
Type Equivalence

- For base types this is simple
  - Types are the same if they are identical
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically or when requested by programmer (casts)

Type Equivalence for Compound Types

- Two basic strategies
  - Structural equivalence: two types are the same if they are the same kind of type and their subtypes are equivalent, recursively
  - Name equivalence: two types are the same only if they have the same name, even if their structures match

Different language design philosophies

Type Equivalence and Inheritance

- Suppose we have
  
  ```java
class Base { ... }  
class Extended extends Base { ... }
```

- A variable declared with type Base has a compile-time type of Base
- During execution, that variable may refer to an object of class Base or any of its subclasses like Extended (or can be null, which is compatible with all class types)
  - Sometimes called the runtime type

Useful Compiler Functions

- You will want methods like this in the objects representing types
  ```java
  boolean assignableTo(Type other) { ... }  
  ```

- Other useful methods might be ones that report whether one type is the same as another

Coming Attractions

- Need to start thinking about translating to object code (actually x86 assembly language for this project)
- Next time: x86 overview/review
- Then
  - Runtime representation of classes, objects, and data
  - Assembly language code for higher-level language statements