Agenda

- Static semantics
- Types
- Attribute grammars
- Representing types
- Symbol tables
- Note: this covers a superset of what we need for MiniJava
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 context-free 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 instance \( p \)?
  - Is variable \( x \) guaranteed to be initialized before it is used?
  - Could \( p \) be null when \( p.q \) is executed?
  - Etc. etc. etc.
What else do we need to know to generate code?

- Where are fields allocated in an object?
- How big are objects? (i.e., how much storage needs to be allocated by new)
- Where are local variables stored when a method is called?
- Which methods are associated with an object/class?

£f(-) £

- In particular, how do we figure out which method to call based on the run-time type of an object?
Types

- Classical roles of types in programming languages
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (method or operator overloading, for example)
  - Provide information to optimizer
Semantics Analysis

- **Main tasks:**
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar

- **Key data structures: symbol tables**
  - For each identifier in the program, record its attributes (kind, type, etc.)
  - Later: assign storage locations (stack frame offsets) for variables; other annotations
## 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>

10/27/2009  
© 2002-09 Hal Perkins & UW CSE
Semantic Checks

- For each language construct we want to know:
  - What semantic rules should be checked: specified by language definition (type compatibility, etc.)
  - For an expression, what is its type (used to check whether the expression is legal in the current context)
  - For declarations in particular, what information needs to be captured to be used elsewhere
A Sampling of Semantic Checks (0)

- Name use: id
  - id has been declared and is in scope
  - Inferred type of id is its declared type
  - Memory location assigned by compiler

- Constant: v
  - Inferred type and value are explicit
A Sampling of Semantic Checks (1)

- Binary operator: \( \text{exp}_1 \text{ op } \text{exp}_2 \)
  - \( \text{exp}_1 \) and \( \text{exp}_2 \) have compatible types
    - Identical, or
    - Well-defined conversion to appropriate types
  - Inferred type is a function of the operator and operands
A Sampling of Semantic Checks (2)

- Assignment: $\text{exp}_1 = \text{exp}_2$
  - $\text{exp}_1$ is assignable (not a constant or expression)
  - $\text{exp}_1$ and $\text{exp}_2$ have compatible types
    - Identical, or
    - $\text{exp}_2$ can be converted to $\text{exp}_1$ (e.g., char to int), or
    - Type of $\text{exp}_2$ is a subclass of type of $\text{exp}_1$ (can be decided at compile time)
  - Inferred type is type of $\text{exp}_1$
  - Location where value is stored is assigned by the compiler
A Sampling of Semantic Checks (3)

- Cast: \((\text{exp}_1) \text{exp}_2\)
  - \(\text{exp}_1\) is a type
  - \(\text{exp}_2\) either
    - Has same type as \(\text{exp}_1\)
    - Can be converted to type \(\text{exp}_1\) (e.g., double to int)
    - Is a superclass of \(\text{exp}_1\) (in general requires a runtime check to verify that \(\text{exp}_2\) has type \(\text{exp}_1\))
  - Inferred type is \(\text{exp}_1\)
A Sampling of Semantic Checks (4)

- Field reference exp.f
  - exp is a reference type (class instance)
  - The class of exp has a field named f
  - Inferred type is declared type of f
A Sampling of Semantic Checks (5)

- Method call `exp.m(e_1, e_2, ..., e_n)`
  - `exp` is a reference type (class instance)
  - The class of `exp` has a method named `m`
  - The method has `n` parameters
  - Each argument has a type that can be assigned to the associated parameter
  - Inferred type is given by method declaration (or is void)
A Sampling of Semantic Checks (6)

- Return statement `return exp; return`;
  - The expression can be assigned to a variable with the declared type of the method (if the method is not void)
  - There’s no expression (if the method 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 separate phases
  - 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 there and later
Attribute Grammars

- A systematic way to think about semantic analysis
- Sometimes used directly, but even if not, AGs are a useful way to think about the analysis
Attribute Grammars

- Idea: associate attributes with each node in the (abstract) syntax tree
- Examples of attributes
  - Type information
  - Storage location
  - Assignable (e.g., expression vs variable – lvalue vs rvalue for C/C++ programmers)
  - Value (for constant expressions)
  - etc. ...
- Notation: \( X.a \) if \( a \) is an attribute of node \( X \)
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 | exp + exp | 1
Informal Example of Attribute Rules (2)

- Attributes
  - `env` (environment, e.g., symbol table); synthesized by `decl`, inherited by `stmt`
  - `type` (expression type); synthesized
  - `kind` (variable `[var, lvalue]` vs value `[val, rvalue]`); synthesized
Attributes for Declarations

- decl ::= int id;
- decl.env = \{identifier, int, var\}
Attributes for Program

- $\text{program ::= decl stmt}$
- $\text{stmt.env = decl.env}$
Attributes for Constants

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

- \( \text{exp} ::= \text{id} \)
  - \( \text{id.type} = \text{exp.env.lookup(id)} \)
  - \( \text{exp.type} = \text{id.type} \)
  - \( \text{exp.kind} = \text{id.kind} \)
Attributes for Addition

- $\text{exp} ::= \text{exp}_1 + \text{exp}_2$
  - $\text{exp}_1.\text{env} = \text{exp}.\text{env}$
  - $\text{exp}_2.\text{env} = \text{exp}.\text{env}$
  - error if $\text{exp}_1.\text{type} \neq \text{exp}_2.\text{type}$
    - (or error if not combatable if rules are move complex)
  - $\text{exp}.\text{type} = \text{exp}_1.\text{type}$
  - $\text{exp}.\text{kind} = \text{val}$
Attribute Rules for Assignment

- stmt ::= exp₁ = exp₂;
  - exp₁.env = stmt.env
  - exp₂.env = stmt.env
  - Error if exp2.type is not assignment compatible with exp1.type
  - error if exp₁.kind == val (must be var)
Example

- `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
- This can 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
In Practice

- Attribute grammars give us a good way of thinking about how to structure semantic checks
- 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 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
Aside: Implementing Symbol Tables

- Topic in classical compiler course: 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.)
- For Java:
  - Map (HashMap) will solve most of the problems
  - List (ArrayList) for ordered lists (parameters, etc.)
Symbol Tables for MiniJava (1)

- 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 field/method names and extract information about them
Symbol Tables for MiniJava (2)

- Global – Per Class Information
  - 1 Symbol table for each class
    - 1 entry for each method/field declared in the class
      - Contents: type information, public/private, parameter types (for methods), storage locations (later), etc.
    - In full Java, multiple symbol tables (or more complex symbol table) per class since methods and fields can have the same names in a class
Symbol Tables for MiniJava (3)

- Global (cont)
  - All global tables persist throughout the compilation
    - And beyond in a real Java or C# compiler...
      - (e.g., symbolic information in Java .class files)
Symbol Tables for MiniJava (4)

- 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
Beyond MiniJava

- 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
    - Scope entry/exit operations
- In multipass compilers, symbol table info needs to persist after analysis of inner scopes for use on later passes
  - See a compiler textbook for details
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

- Many languages have some “predefined” items
- Include code 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
Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char

- Compound/Constructed 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 {
    - ...
  } // or interface
  - class ClassType extends Type {
    - ...
  }
  - class BaseType 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 it
  - Symbol table entries and AST nodes for expressions refer to these to represent type info
  - Usually create at compiler startup
- Useful to create a “void” type object to tag functions that do not return a value (if you implement these)
- Also useful to create an “unknown” type object for errors
  - (Having “void” and “unknown” type objects reduces the need for special case code for these in various places.)
Compound Types

- Basic idea: represent with a "type constructor" object that refers to component types
  - Limited number of these – correspond directly to type constructors in the language (record/struct, array, function,...)
  - A compound type is a graph
Class Types

- class Id { fields and methods }

  class ClassType extends Type {
  
  ✓ Type baseClassType; // ref to base class  
  ✓ Map fields;        // type info for fields  
  ✓ Map methods;       // type info for methods  
  }  

  (Note: may not want to do this literally, depending on how  
  class symbol tables are represented; i.e., class symbol tables  
  might be useful as the representation of the class type.)
Array Types

- For regular Java this is simple: only possibility is # of dimensions and element type

```java
class ArrayType extends Type {
    int nDims;
    Type elementType;
}
```
Array Types for Pascal &c.

- 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 GeneralArrayType extends Type {
    Type indexType;
    Type elementType;
  }
  ```
Methods/Functions

Type of a method is its result type plus an ordered list of parameter types

```java
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 component types 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

- Create a handful of methods to decide different kinds of type compatibility:
  - Types are identical
  - Type t1 is assignment compatible with t2
  - Parameter list is compatible with types of expressions in the call

- Normal modularity reasons: isolates these decisions in one place and hides the actual type representation from the rest of the compiler
Implementing Type Checking for MiniJava

- Create multiple visitors for the AST
- First passe(s): gather information
  - Collect global type information for classes
  - Could do this in one pass, or might want to do one pass to collect class information, then a second one to collect per-class information about fields, methods
- Next set of passes: go through method bodies to check types, other semantic constraints
Coming Attractions

- Need to start thinking about translating to object code (actually x86 assembly language, the default for this project)

- Next:
  - x86 overview (as a target for simple compilers)
  - Runtime representation of classes, objects, data, and method stack frames
  - Assembly language code for higher-level language statements