CSE 401 - Compilers

Lecture 15: Semantic Analysis, Part III

Michael Ringenburg

Winter 2013

Reminders/ Announcements

- Project Part 2 due Wednesday
- Midterm Friday
  - Sections this week will be devoted to midterm review
- Lined up a guest lecture on register allocation of the last day of class (3/15) from Preston Briggs
  - Affiliate faculty here, who previously did some of the foundational work in register allocation – he's mentioned in your textbook (Chapter 13 notes).
- Also looking at a guest lecture that week about real-world, non-compiler applications of parsing.

Today's Agenda

- Symbol Tables
  - And symbol tables for MiniJava
- Typechecking

Symbol Tables

- Primary place where information collected during Semantic analysis is stored
- Maps identifiers to properties such as type, size, location, etc.
- 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 old 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 …
- Java:
  - Map (HashMap) will handle most cases
  - List (ArrayList) for ordered lists (parameters, etc.)

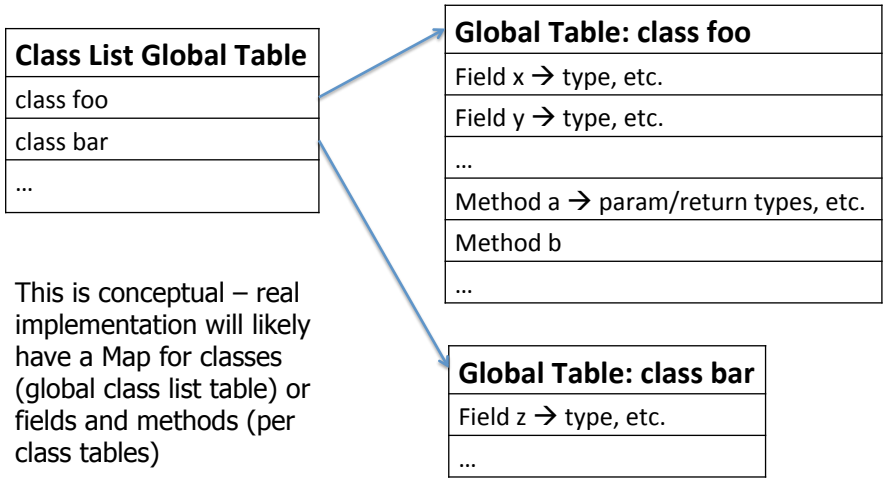
Symbol Tables for MiniJava

- Consider this a general outline, based on recommendations courtesy of 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 Table – 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 extract information about them (e.g., fields and methods)

Symbol Tables for MiniJava

- Other Global Tables – Per Class Information
  - 1 Symbol table for each class
    - 1 entry per method/field declared in the class
      - Contents: type information, public/private/protected (if implementing – not required in basic MiniJava), parameter types (for methods), storage locations (offset of fields in class - will be discussed later), etc.
      - Note: Storage info probably not needed for project part 3, but will be in part 4. Make sure it's easy to extend your implementation.
  - 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

Conceptual Diagram of Global Tables



This is conceptual – real implementation will likely have a Map for classes (global class list table) or fields and methods (per class tables)

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 (offset from stack - filled in/discussed 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 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 (*conceptually – but can't really …)*

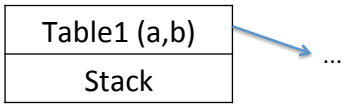
Scopes (Conceptual)

```
void foo() {
  int a, b;
  …
  while (a != b) {
    int x, y;
    …
  }
}
```

Stack

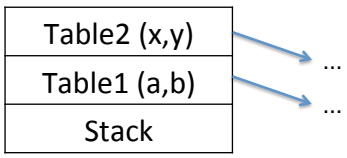
Scopes (Conceptual)

```
void foo() {
  int a, b;
  …
  while (a != b) {
    int x, y;
    …
  }
}
```



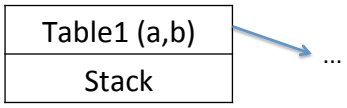
Scopes (Conceptual)

```
void foo() {
  int a, b;
  …
  while (a != b) {
    int x, y;
    …
  }
}
```



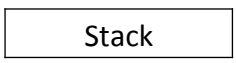
Scopes (Conceptual)

```
void foo() {
  int a, b;
  …
  while (a != b) {
    int x, y;
    …
  }
}
```



Scopes (Conceptual)

```
void foo() {
  int a, b;
  …
  while (a != b) {
    int x, y;
    …
  }
}
```



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-down 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?
  - Prefer to 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?

Today's Agenda

- Symbol Tables
  - And symbol tables for MiniJava
- Typechecking

Types

- Types play a key role in most programming languages. E.g.,
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (method or operator overloading, for example)
  - Provide information to optimizer
    - Strongly typed languages – what data might be used where
    - Type qualifiers (e.g., const and restrict in C)

Type Checking Terminology

Static vs. dynamic typing
- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

Strong vs. weak typing
- strong: guarantees no illegal operations performed
- weak: can't make guarantees

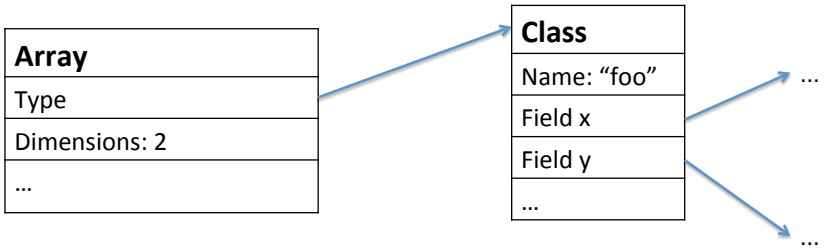
Caveats:
- Hybrids common
- Inconsistent usage common
- "untyped," "typeless" could mean dynamic or weak

	static	dynamic
strong	Java, ML	Scheme, Ruby
weak	C	PERL

Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char, bool
- Compound/Constructed Types
  - Built up from other types (recursively) via *type constructors*
  - Constructors include arrays, records/structs/ classes, pointers, enumerations, functions, modules, …

Constructed/Compound Types



Type Representation

- Typical compiler representations create a shallow class hierarchy, for example:

```
abstract class Type { … }   // or interface
class ClassType extends Type { … }
class BaseType extends Type { … }
```

  - Should not need too many of these

Types vs ASTs

- Types are not AST nodes!
  - AST nodes may have a type *field*, however
- AST = abstract representation of source program (including source program type info)
- Types = abstract representation of type semantics for type checking, inference, etc.
  - Can include information not explicitly represented in the source code, or may describe types in ways more convenient for processing
- Be sure you have a separate "type" class hierarchy in your compiler distinct from the AST

Base Types

- For each base type (int, boolean, others in other languages), create a single object to represent it
  - Base types in symbol table entries and AST nodes are direct references to these objects
  - Base type objects usually created at compiler startup
- Useful to create a type "void" object to tag functions that do not return a value
- Also useful to create a type "unknown" object for errors
  - ("void" and "unknown" types reduce the need for special case code in various places in the type checker – no null type or return type fields))

Compound Types

- Basic idea: use a appropriate "type constructor" object that refers to the component types
  - Limited number of these – correspond directly to type constructors in the language (record/struct/ class, array, function,…)

Class Types

- Type for: 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
}
```

  - Base class pointer, so we can check field references against base class if we don't find in this class.
  - (Note: may not want to do this literally, depending on how class symbol tables are represented; i.e., class symbol tables might be useful or sufficient as the representation of the class type.)

Array Types

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

```
class ArrayType extends Type {
  int nDims;
  Type elementType;
}
```

- More interesting in languages like Pascal (more complex array indexing)

Methods/Functions

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

```
class MethodType extends Type {
  Type resultType;       // type or "void"
  List parameterTypes;
}
```


Type Equivalance

- For base types this is simple
  - If you have just a single instance of each base type (as recommend), then types are the same if and only if they are identical
    - Pointer/reference comparison in the type checker
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically or when requested by programmer (casts) – often involves inserting cast/conversion nodes in AST
    - Basic MiniJava doesn't need these – only int's

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
    - E.g., two struct types, each with exactly two int fields
  - *Name equivalence*: two types are the same only if they have the same name. If their structures match, but have distinct names, they are not equal.
    - I.e., two variables only have the same type if they are declared from the same class (even if two classes are structurally identical).
- Different language design philosophies
  - Same languages use a mixture, as well

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  - Identical base types clearly have the same structure
  - if type constructors:
    - same constructor
    - recursively, equivalent arguments to constructor
- Ex: atomic types, array types, ML record types
- Implement with recursive implementation of equals, or by canonicalization of types when types created then use pointer/reference equality

Name Equivalence

- Name equivalence says that two types are equal iff they came from the same textual occurrence of a type constructor
  - Ex: class types, C struct types (struct tag name), datatypes in ML
  - special case: type synonyms (e.g. typedef in C) do not define new types
- Implement with pointer/reference equality assuming appropriate representation of type info

Type Equivalence and Inheritance

- Suppose we have

```
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)
  - Sometimes called the *runtime type*
  - Subclasses guaranteed to have all fields/methods of base class, so typechecking as base class suffices

Type Casts

- In most languages, one can explicitly cast an object of one type to another
  - sometimes cast means a conversion (e.g., casts between numeric types)
  - sometimes cast means a change of static type without doing any computation (casts between pointer types or pointer and numeric types)
  - With class types, may also mean upcast (free) or downcast (runtime check)
  - Note: Casts not present in basic MiniJava

Type Conversions and Coercions

- In Java, we can explicitly convert an value of type double to one of type int
  - Can represent as unary operator
  - Typecheck, generate code normally
- In Java, can implicitly coerce an value of type int to one of type double
  - Compiler must insert unary conversion operators, based on result of type checking
- Once again – only ints in basic MiniJava

C and Java: type casts

- In C: safety/correctness of casts not checked
  - Allows writing low-level code that's type-unsafe
  - Result is often implementation dependent/undefined. Not portable, but sometimes useful.
- In Java: downcasts from superclass to subclass need run-time check to preserve type safety
  - Otherwise, might use field (or call method) that is not present in superclass
  - Static typechecker allows the cast
  - Code generator introduces run-time check
    - (same code needed to handle "instanceof")
  - Java's main form of dynamic type checking

Various Notions of Equivalance

- There are usually several relations on types that we need to deal with:
  - "is the same as"
  - "is assignable to"
  - "is same or a subclass of"
  - "is convertible to"
- Be sure to check for the right one(s)

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
- Usual modularity reasons: isolates these decisions in one place and hides the actual type representation from the rest of the compiler
- Probably belongs in the same package with the type representation classes (package for dealing with types)

Implementing Type Checking for MiniJava

- Create multiple visitors for the AST
- First pass/passes: gather class 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 – you decide
- Next set of passes: go through method bodies to check types, other semantic constraints

Disclaimer

- This discussion of semantics, type representation, etc. should give you a good idea of what needs to be done in your project, but you'll need to adapt the ideas to the project specifics.
  - Project part 3 out later this week – targeting Thursday (day after part 2 is due).
- You'll also find good ideas in your compiler book(s).

Coming Attractions

- Need to start thinking about translating to object code (actually x86(-64) assembly language, the default for this project)
- Next lectures
  - 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
- And there's a midterm on Friday!