CSE 401 – Compilers

Static Semantics

Hal Perkins

Winter 2010

Agenda

- Static semantics
- Types
- Attribute grammars
- Representing types
- Symbol tables
- Note: this covers a superset of what we need for the MiniJava project

What do we need to know to compile & check 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?
  - In particular, how do we figure out which method to call based on the run-time type of an object?

Semantic Analysis

- Main tasks:
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar
  - Resolve names
    - Relate declarations and uses of each variable
  - “Understand” the program well enough for synthesis
- Key data structure: Symbol tables
  - Map each identifier in the program to information about it (kind, type, etc.)
- Final part of the analysis phase / front end of the compiler

Some Kinds of Semantic Information

Information	*Generated From*	*Used to process*
Symbol tables	Declarations	Expressions, statements
Type information	Declarations, expressions	Operations
Constant/variable information	Declarations, expressions	Statements, expressions
Register & memory locations	Assigned by compiler	Code generation
Values	Constants	Expressions

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 use elsewhere

A Sampling of Semantic Checks (0)

- Appearance of a name: 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: exp_1 op exp_2
  - exp_1 and exp_2 have compatible types
    - Either 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: $exp_1 = exp_2$
  - exp_1 is assignable (not a constant or expression)
  - exp_1 and exp_2 have compatible types
    - Identical, or
    - exp_2 can be converted to exp_1 (e.g., char to int), or
    - Type of exp_2 is a subclass of type of exp_1 (can be decided at compile time)
  - Inferred type is type of exp_1
  - 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 superclass of exp1 (in general requires a runtime check to verify that exp2 has type exp1)
  - Inferred type is exp1

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)

Attribute Grammars

- A systematic way to think about semantic analysis
- Sometimes used directly, but even when not, AGs are a useful way to organize 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 a .val attribute
- AST and attribution for (1+2) * (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)

- Suppose we have the following grammar for a trivial language

  program ::= decl stmt

  decl ::= int id;

  stmt ::= exp = exp ;

  exp ::= id | exp + exp | 1

- We want to give suitable attributes for basic type and lvalue/rvalue checking

Informal Example of Attribute Rules (2)

- Attributes
  - env (environment, e.g., symbol table); synthesized by decl, inherited by stmt
    - Each entry maps a name to its type and kind
  - type (expression type); synthesized
  - kind (variable [var or lvalue] vs value [val or rvalue]); synthesized

Attributes for Declarations

- decl ::= int id;
  - decl.env = {id, 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 Identifier Exprs.

- exp ::= id
  - id.type = exp.env.lookup(id)
  - exp.type = id.type
  - exp.kind = id.kind

Attributes for Addition

- $exp ::= exp_1 + exp_2$
  - $exp_1.env = exp.env$
  - $exp_2.env = exp.env$
  - error if $exp_1.type$!= $exp_2.type$
    - (or error if not combatable if rules are move complex)
  - $exp.type = exp_1.type$
  - $exp.kind = val$

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 compatibile with exp1.type
  - error if exp_1.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
- But implementation 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 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

Aside: Implementing Symbol Tables

- Big topic in classical 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.)
- 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 class types and their 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 per method/field declared in the class
      - Contents: type information, public/private, parameter types (for methods), storage locations (later), etc.
  - In full Java, need multiple symbol tables (or more complex symbol table) per class
    - Java allows the same identifier to name both a method and a field in a class – multiple namespaces

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
    - But if type checking and code gen, etc. are done in separate passes, this table needs to persist until we're done with it

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
    - Allows you to only complain once! (How?)

“Predefined” Things

- Many languages have some “predefined” items (functions, classes...)
- Include init 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

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

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	Lisp
weak	C	PERL (1-5)

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, for example:

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

  - Should not need too many of these
- Not the same as the AST representation for source program type or variable declarations
  - Difference is we want to capture the semantics of the type system here for inference, checking, etc.
  - An instance of this graph represents each compile-time type found in the program

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 create at compiler startup
- Useful to create a “void” type object to tag functions that do not return a value (if you have or add these)
- Also useful to create an “unknown” type object for errors
  - (“void” and “unknown” types reduce the need for special case code in various places in the type checker)

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, 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 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;
}
```

Array Types for Pascal &c

- Pascal allows arrays to be indexed by any discrete type like an enum, char, or subrange of int or other 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

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

Type Equivalance

- 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

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  - atomic types are tautologically 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 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) don't define new types
- Implement with pointer 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, which is compatible with all class types)
  - Sometimes called the *runtime type*

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)

Type Conversions and Coercions

- In Java, can explicitly convert an value of type double to one of type int
  - can represent as unary operator
  - typecheck, codegen 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

C and Java: type casts

- In C: safety/correctness of casts not checked
  - allows writing low-level code that's type-unsafe
  - more often used to work around limitations in C's static type system
- In Java: downcasts from superclass to subclass include run-time type check to preserve type safety
    - static typechecker allows the cast
    - codegen introduces run-time check
    - Java's main form of dynamic type checking

Various Notions of Equivalance

- So 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 compatibile 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
- Probably belongs in the same package with the type representation classes

Implementing Type Checking for MiniJava

- Create multiple visitors for the AST
- First pass/passes: 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 – 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 you'll project, but you'll need to adapt the ideas to the project specifics.
- You'll also find good ideas in your compiler book...

Coming Attractions

- Need to start thinking about translating to object code (actually x86 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 in there somewhere...