CSE P 501 – Compilers

Static Semantics

Hal Perkins

Winter 2016

Administrivia

- Parser + AST + print visitors due Mon. 11 pm
  - How's it going?
- "scanner-final" and "parser-final" tags must be *exactly* that
- HW1 grading: we allowed a lot of (too much?) slack for \escape sequences in regexps – made it hard to grade. For mathematical regexps, keep the notation simple. Maybe underline literal chars, <u>+</u>, <u>(</u>, etc., and use plain symbols as operators, only when possible confusion, or something equally simple.
- Next part of project: semantics + type check out next Tue.

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

Disclaimer: There's (lots) more here than the minimum needed for the MiniJava project

What do we need to know to check if this is legal and compile it?

```
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 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?
  - How do we figure out which method to call based on the run-time type of an object?

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

Semantic Analysis

- Main tasks:
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar
  - Resolve names – connect declarations and uses
  - "Understand" the program – last phase of front end …
  - … so program is "correct" for hand-off to back end
- Key data structure: Symbol tables
  - For each identifier in the program, record its attributes (kind, type, etc.)
  - Later: assign storage locations (stack frame offsets) for variables, add other annotations

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

- Grammar = BNF
  - Short: e.g., Java in a handful of pages
- Semantics = Language Reference Manual
  - Long: Java SE 8 = 788 pages
- For each language construct we want to know:
  - What semantic rules should be checked
  - For an expression, what is its type (is expression legal)
  - For declarations, what to capture for use elsewhere

A Sampling of Semantic Checks (0)

- Appearance of a name: id
  - Check: id has been declared and is in scope
  - Compute: Inferred type of id is its declared type
- Constant: v
  - Compute: Inferred type and value are explicit

A Sampling of Semantic Checks (1)

- Binary operator: exp_1 op exp_2
  - Check: exp_1 and exp_2 have compatible types
    - Identical, or
    - Well-defined conversion to appropriate types
  - Compute: Inferred type is a function of the operator and operand types

A Sampling of Semantic Checks (2)

- Assignment: $exp_1 = exp_2$
  - Check: exp_1 is assignable (not a constant or expression)
  - Check: exp_1 and exp_2 have (assignment-)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)
  - Compute: Inferred type is type of exp_1

A Sampling of Semantic Checks (3)

- Cast: (exp_1) exp_2
  - Check: exp_1 is a type
  - Check: exp_2 either
    - Has same type as exp_1
    - Can be converted to type exp_1 (e.g., double to int)
    - Downcast: is a superclass of exp_1 (usually requires a runtime check to verify; at compile time we can at least decide if it could be true)
    - Upcast (Trivial): is the same or a subclass of exp_1
  - Compute: Inferred type is exp_1

A Sampling of Semantic Checks (4)

- Field reference: exp.f
  - Check: exp is a reference type (not value type)
  - Check: The class of exp has a field named f
  - Compute: Inferred type is declared type of f

A Sampling of Semantic Checks (5)

- Method call: exp.m(e_1, e_2, …, e_n)
  - Check: exp is a reference type (class instance)
  - Check: The class of exp has a method named m
  - Check: The method exp.m has n parameters
    - Or, if overloading allowed, at least one version of m exists with n parameters
  - Check: Each argument has a type that can be assigned to the associated parameter
    - Same "assignment compatible" check for assignment
    - Overloading: need to find a "best match" among available methods if more than one is compatible – or reject if result is ambiguous (e.g., C++, others)
  - Compute: Inferred type is given by method declaration (or could be void)

A Sampling of Semantic Checks (6)

- Return statement: return exp; or: return;
- Check:
  - If the method is not void: The expression can be assigned to a variable with the declared return type of the method – exactly the same test as for assignment statement
  - If the method is void: There is no expression

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

Attribute Grammars

- A systematic way to think about semantic analysis
- Formalize properties checked and computed during semantic analysis and relate them to grammar productions in the CFG (or AST)
- Sometimes used directly, but even when not, AGs are a useful way to organize the analysis and think about it

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 in C/C++ terms)
  - 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 giving the computed value of that node
- AST and attribution for (1+2) * (6 / 2)

Inherited and Synthesized Attributes

Given a production $X ::= Y_1\ Y_2\ \dots\ Y_n$

- A *synthesized* attribute X.a is a function of some combination of the attributes of the 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)
  - Often restricted a bit: only Y's to the left can be used (has implications for evaluation)

Attribute Equations

- For each kind of node we give a set of equations relating attribute values of the node and its neighbors (usually children)
  - Example: $plus.val = exp_1.val + exp_2.val$
- Attribution (evaluation) means implicitly finding a solution that satisfies all of the equations in the tree
  - This is an example of a constraint language

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

- What attributes would we create to check types and assignability?

Informal Example of Attribute Rules (2)

- Attributes of nodes
  - 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 $\longrightarrow$ (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 Expressions

exp ::= id

(type, kind) = exp.env.lookup(id)

exp.type = type   (i.e., id type)

exp.kind = kind   (i.e., id kind)

Attributes for Addition

$exp ::= exp_1 + exp_2$

$exp_1.env = exp.env$

$exp_2.env = exp.env$

error if $exp_1.type \mathrel{!=} exp_2.type$

(or error if not compatible, depending on language rules)

$exp.type = exp_1.type$ (or $exp_2.type$)

$exp.kind = val$

Attribute Rules for Assignment

stmt ::= exp_1 = exp_2;

exp_1.env = stmt.env

exp_2.env = stmt.env

Error if exp_2.type is not assignment compatible with exp_1.type

Error if exp_1.kind is not var (can't be val)

Example

int x; x = x + 1;

Extensions

- This can be extended to handle sequences of declarations and statements
  - Sequences of declarations builds up larger environments, each decl synthesizes a new env. from previous one plus the new binding
  - Full environment is passed down to statements and expressions

Observations

- These are equational computations
  - Think functional programming, no side effects
- Solver 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 and exploit inheritance. Most statements don't need types, for example, but all expressions do.

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

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 augment) in later compiler phases

Aside: Implementing Symbol Tables

- Big topic in classical (i.e., ancient) 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

- We'll outline a scheme that does what we need, but feel free to modify/adapt as needed
- Mix of global and local tables

Symbol Tables for MiniJava: Global

- 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: Class

- One symbol table for each class
  - One entry per method/field declared in the class
    - Contents: type information, public/private, parameter types (for methods), storage locations (later), etc.
- Reached from global table of class names
- In Java, we actually need multiple symbol tables (or more complex symbol table) per class
  - The same identifier can be used for both a method name and a field name in a single class

Symbol Tables for MiniJava: Global/Class

- All global tables persist throughout the compilation
  - And beyond in a real compiler…
    - Symbolic information in Java .class or MSIL files, link-time optimization information in gcc)
    - Debug information in .o and .exe files
    - Some or all information in library files (.a, .so)
    - Type information for garbage collector

Symbol Tables for MiniJava: Methods

- One local symbol table for each method
  - One entry for each local variable or parameter
    - Contents: type info, storage locations (later), etc.
  - Needed for project only while compiling the method; can discard when done in a single pass compiler
    - But if type checking and code gen, etc. are done in separate passes, this table needs to persist until we're done with it
      - And beyond: often need type info for runtime debugging, memory management/garbage collection, etc.
    - Even for our project, the MiniJava compiler will likely have multiple passes

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 (ML, …)
  - Lambdas and function closures
- Basic idea: new symbol table for inner scopes, linked to surrounding scope's table (i.e., stack of symbol tables, top = current innermost scope)
  - Look for identifier in inner scope; if not found look in surrounding scope (recursively)
  - Pop symbol table when we exit a scope
- Also ignoring static fields/methods, accessibility (public, protected, private), package scopes, …

Engineering Issues (1)

- In multipass compilers, inner scope symbol tables need to persist for use in later passes
  - So really can't delete symbol tables on scope exit
  - Retain and add a pointer to the parent scope (effectively a reverse tree of scope symbol tables with root = global table)
    - Keep a pointer to current innermost scope (leaf) and start looking for symbols there

Engineering Issues (2)

- In practice, want to retain O(1) lookup or something close to it
  - Would like to avoid O(depth of scope nesting), although some compilers assume this will be small enough not to matter
  - When it matters, use hash tables with additional information (linked lists of various sorts) to get the scope nesting right
    - Scope entry/exit operators
  - See a compiler textbook for ideas & details

Error Recovery

- What to do when an undeclared identifier is encountered?
  - Only complain once (Why?)
  - Can forge a symbol table entry for id 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 (constants, functions, classes, namespaces, standard libraries, …)
- Include initialization code or declarations 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
  - Can put “standard prelude” information in a file or data resource and use that to initialize
    - Tradeoffs?

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

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
    - In strongly typed languages, allows compiler to make assumptions about possible values
    - Qualifiers like const and restrict in C allow for other assumptions

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, SML	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)
  - Constructors include records/structs/classes, arrays, pointers, enumerations, functions, modules, …
    - Most language provide a small collection of these

How to Represent Types in a Compiler?

Create a shallow class hierarchy

- Example:

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

- Should not need too many of these

Types vs ASTs

- Types nodes are not AST nodes!
- 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, char, double, etc.) create exactly one object to represent it (singleton pattern!)
  - Symbol table entries and AST nodes reference these objects to represent entry/node types
  - 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; don't have to return "null" for "no type" or "not declared" cases)

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 (pointer, array, record/struct/class, function,…)
  - So a compound type is represented as a graph
- Some examples…

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
}
```

(MiniJava note: May not want to represent class types exactly like this, depending on how class symbol tables are represented; e.g., the class symbol table(s) might be a sufficient representation of a class type.)

Array Types

- For regular Java this is simple: only possibility is # of dimensions and element type (which can be another array type or anything else)

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

Array Types for Other Languages

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

  ```
  array [indexType] of elementType
  ```

  (fantastic idea – would be nice if it became popular again)
- Element type can be any other type, including an array (e.g., 2-D array = 1-D array of 1-D 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;
}
```

- Sometimes called the method "signature"

Type Equivalance

- For base types this is simple: types are the same if they are identical
    - Can use pointer comparison in the type checker if you have a singleton object for each base type
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically where required by the language spec or when written explicitly by programmer (casts) – often involves inserting cast or conversion nodes in AST

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
  - e.g., are Complex and Point the same?
  - e.g., are Point (Cartesian) and Point (Polar) the same?

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  – Atomic types are tautologically the same structure and equal if they are the same type
  – For type constructors: equal if the same constructor and, recursively, type (constructor) components are equal
- 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/ref. 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 equality assuming appropriate representation of type info

Type Equivalence and Inheritance

- Suppose we have

  class Base { … }

  class Derived extends Base { … }
- A variable declared with type Base has a *compile-time type* or *static type* of Base
- During execution, that variable may refer to an object of class Base or any of its subclasses like Derived (or can be null), often called the the *runtime type* or *dynamic type*
  - Since subclass is guaranteed to have all fields/methods of base class, type checker only needs to deal with declared compile-time types of variables and, in fact, can't track all possible runtime types

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 in C)
  - for objects can be a upcast (free and always safe) or downcast (requires runtime check to be safe)

Type Conversions and Coercions

- In full Java, we can explicitly convert a value of type double to one of type int
  - can represent as unary operator
  - typecheck, codegen normally
- In full 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/C++: safety/correctness of casts not checked
  - allows writing low-level code that's type-unsafe
  - C++ has more elaborate casts, and at least one of them does imply runtime checks
- In Java: downcasts from superclass to subclass need runtime check to preserve type safety
    - static typechecker allows the cast
    - codegen introduces runtime check
      - (same code needed to handle "instanceof")
    - Java's main need for dynamic type checking

Various Notions of Type Compatibility

- There are usually several relations on types that we need to analyze in a compiler:
  - "is the same as"
  - "is assignable to"
  - "is same or a subclass of"
  - "is convertible to"
- Exact meanings and checks needed depend on the language spec.
- 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 t_1 is assignment compatible with t_2
  - Parameter list is compatible with types of expressions in the method 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

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

Agenda

- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Wrapup

Disclaimer

- This overview of semantics, type representation, etc. should give you a decent idea of what needs to be done in you're project, but you'll need to adapt the ideas to the project specifics.
- You'll also find good ideas in your compiler book…
- And remember that this lecture covers more than is needed for our specific project

Coming Attractions

- To get a running compiler we need:
  - Execution model for language constructs
  - x86-64 assembly language for compiler writers
  - Code generation and runtime bootstrap details
- We'll also spend considerable time on compiler optimization
  - Intermediate reps., graphs, SSA, dataflow
  - Optimization analysis and transformations
- Immediate problem is to keep lectures from getting too far ahead of the project - maybe hold off on runtime details?
  - Thoughts? Suggestions? Opinions?