CSC 348 – Intro to Compilers
Lecture 5

Dr. David Zaretsky
david.zaretsky@depaul.edu

Agenda

- Static semantics
- Attribute grammars
- Types & type checking
- Programming Assignment 4

What do we need to know to compile and 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.

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 declaration 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.)
  - Later: assign storage locations (stack frame offsets) for variables, add other annotations
- This is the final part of the analysis phase (front end) of the compiler

Examples of Semantic Errors

assigned type doesn't match declared type

```
int a; a = true;
```

relational operator applied to non-int type

```
1 < true
```

```
void foo(int x) {
    int x;
    foo(5,7);
}
```

argument list doesn't match formal parameters

```
class A {…}
class B extends A {
    void foo() {
        A a;
        B b;
        b = a;
    }
}
```

a is not a subtype of **b**

Semantic Checks

- For each language construct we want to know:
  - What semantic rules should be checked
    - Specified by language definition (type compatibility, required initialization, etc.)
  - For an expression, what is its type
    - Used to check whether expression is legal in the current context
  - For declarations, what information needs to be captured to use elsewhere

A Sampling of Semantic Checks (1)

- 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 (2)

- Binary operator: exp_1 op exp_2
  - Check: exp_1 and exp_2 have compatible types
    - Either 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 (3)

- 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 (4)

- 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 (in general this 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 (5)

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

A Sampling of Semantic Checks (6)

- Method call: exp.m(e_1, e_2, …, e_n)
  - Check: exp is a reference type (not primitive type)
  - Check: The type of exp has a method named m
    - (inherited or declared as part of the type)
  - Check: The method m has n parameters
    - Or, if overloading is 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., full Java, C++, others)
  - Compute: Inferred type is given by method declaration (or could be void)

A Sampling of Semantic Checks (7)

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

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 $X ::= Y_1\ Y_2\ \ldots\ Y_n$

- 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)

Attribute Equations

- For each kind of node we give a set of equations (*not* assignments) relating attribute values of the node and its children
  - Example: plus.val = exp_1.val + exp_2.val
- Attribution (evaluation) means 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 (lvalue vs rvalue)?

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 ⟶ (int, var)}
  - In the symbol table, we map **id** to a symbol type that is an integer variable.

Attributes for Program

- program ::= decl stmt
  - stmt.env = decl.env
  - The stmt env is inherited from decl

Attributes for Constants

- exp ::= I
  - exp.kind = val
  - exp.type = int
  - A constant is a value of type integer

Attributes for Identifier Exprs.

- exp ::= id
  - (type, kind) = exp.env.lookup(id)
  - exp.type = type (i.e., id type)
  - exp.kind = kind (i.e., id kind)
  - When an identifier is encountered, we lookup the symbol in the table (environment) and apply its type/kind to the expression.

Attributes for Addition

- exp ::= exp1 + exp2
  - exp1.env = exp.env (inherited env)
  - exp2.env = exp.env (inherited env)
  - error if exp1.type != exp2.type
    - (or error if not compatible, depending on language rules)
  - exp.type = exp1.type (or exp2.type)
    - (or whatever type that language rules specify)
  - exp.kind = val

Attribute Rules for Assignment

- stmt ::= exp1 = exp2;
  - exp1.env = stmt.env (inherited env)
  - exp2.env = stmt.env (inherited env)
  - Error if exp2.type is not assignment compatible with exp1.type
  - Error if exp1.kind is not var (can't be val)

Example

int x;

x = x + 1;

- decl.env = {x ⟶ (int, var)}
- (x.type, x.kind) = exp.env.lookup(x)
- r.value = 1; r.type = int; r.kind = val
- Check: (x.type == r.type)
- rhs.type = x.type
- (lhs.type, lhs.kind) = stmt.env.lookup(x)
- Check:
  - lhs.type (x) == rhs.type (x + 1)
  - lhs.kind == var

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.

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?

- One solution: create a shallow class hierarchy
- Example:
- abstract class Type { … }   // or interface
- class ClassType extends Type { … }
- class BaseType extends Type { … }
- Should not need too many of these

```
abstract class Type {...}

class IntType extends Type {...}

class BoolType extends Type {...}

class ArrayType extends Type {
  Type elemType;
}

class MethodType extends Type {
  Type[] paramTypes;
  Type returnType;
  ...
}

class ClassType extends Type {
  ICClass classAST;
  ...
}
...
```

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. (i.e., an ADT)
  - 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 a single object to represent it (singleton!)
  - 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 for the result "type" of 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 "compound type" or "type constructor" object that references the component types
  - Limited number of these – correspond directly to type constructors in the language (pointer, array, record/struct/class, function,…)
  - A compound type is 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:
  - # of dimensions
  - 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
```

- 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 & 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 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 Extended 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 Extended (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)
  - 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 in the AST
  - typecheck, codegen as usual
- In full Java, can implicitly coerce a 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 one of them does require runtime checks
- In Java: downcasts from superclass to subclass need runtime check to preserve type safety
  - static typechecker allows the cast
  - typechecker/codegen introduces runtime check (same code needed to handle "instanceof")
  - Java's main need for dynamic type checking

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: isolate these decisions in one place and hide 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

Type-checking in MiniJava

```
// Type t;
// Identifier i;
public void visit(VarDecl n) {
  Type t = n.t.accept(this);
  String id = n.i.toString();
  if (currMethod == null) {
    if (!currClass.addVar(id,t))
       error.complain(id + "is already defined in " + currClass.getId());
    } else if (!currMethod.addVar(id,t))
        error.complain(id + "is already defined in ″ + currClass.getId() + "." + currMethod.getId());
}

// Exp e1,e2;
public Type visit(Plus n) {
  if (! (n.e1.accept(this) instanceof IntegerType) )
    error.complain("Left side of LessThan must be of type integer");
  if (! (n.e2.accept(this) instanceof IntegerType) )
    error.complain("Right side of LessThan must be of type integer");
  return new IntegerType();
}
```

Type-checking in MiniJava

```
// Type t;
// Identifier i;
public void visit(VarDecl n) {
  Type t = n.t.accept(this);
  String id = n.i.toString();
  if (currMethod == null) {
    if (!currClass.addVar(id,t))
       error.complain(id + "is already defined in " + currClass.getId());
    } else if (!currMethod.addVar(id,t))
        error.complain(id + "is already defined in ” + currClass.getId() + "." + currMethod.getId());
}

// Exp e1,e2;
public Type visit(Plus n) {
  if (! (n.e1.accept(this) instanceof IntegerType) )
    error.complain("Left side of LessThan must be of type integer");
  if (! (n.e2.accept(this) instanceof IntegerType) )
    error.complain("Right side of LessThan must be of type integer");
  return new IntegerType();
}
```

Disclaimer

- This overview of semantics, type representation, etc. should give you a decent idea of what needs to be done in your project, but you'll need to adapt the ideas to the project specifics.
- And remember that these slides cover more than is needed for our specific project

Next…

- Need to start thinking about translating to target code (x86-64 assembly language for our project)
- Next lectures
  - X86-64 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, method calls, dynamic dispatch, …
- Programming Assignment 4: Semantic Analysis