CHAPTER 4

SEMANTIC ANALYSIS

Objectives

- Semantics
- Types
- Attribute grammars
- Representing types
- Symbol tables

An example

```
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…

What else do we need 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?

Types

- Role of types in programming languages
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (method or operator overloading, for example)

Semantic Analysis

- Some key ideas
  - 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

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 according to the language definition (type compatibility, etc.)
  - For an expression, what is its type (used to check whether the expression is used in the right context)
  - For declarations in particular, what information needs to be captured to be used elsewhere

A Sampling of Semantic Checks

- 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

- Binary operator: exp_1 op exp_2
  - exp_1 and 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

- 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

- Cast: (exp_1) exp_2
  - exp_1 is a type
  - exp_2 either
    - Has same type as exp_1
    - Can be converted to type exp_1 (e.g., double to int)
    - Is a superclass of exp_1 (in general requires a runtime check to verify that exp_2 has type exp_1)
  - Inferred type is exp_1

A Sampling of Semantic Checks

- 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

- 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

- 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 (AGs)

- A systematic way to think about semantic analysis
- Sometimes used directly, but even if ad hoc techniques are used, AGs are a useful guide to organizing the analysis

Attribute Grammars (AGs)

- 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 (abstract syntax tree) 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

- 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

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

Attributes for Declarations

- decl ::= int id;
    - decl.env = {identifier, 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 Expressions

- exp ::= id
  - id.type = exp.env.lookup(id)
  - exp.type = id.type
  - error if id.type != exp.expectedtype
  - exp.kind = id.kind

Attributes for Addition

- $exp ::= exp_1 + exp_2$
  - $exp_1.env = exp.env$
  - $exp_2.env = exp.env$
  - error if $exp_1.type \mathrel{!=} exp_2.type$
    - (or error if not combatable when 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)

Attribute 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 (abstract syntax tree) nodes to refer to appropriate attributes (symbol table entries for identifiers, expression types, etc.)
    - Put in appropriate places in inheritance tree – 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++, etc.)
- For Java:
  - Map (HashMap) will solve most of the problems
  - List (ArrayList) for ordered lists (parameters, etc.)

Symbol Tables for MiniJava

- 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

- 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

- 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

- 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
- In multipass compilers, symbol table information needs to persist after analysis of inner scopes for use on later passes
  - See a good 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
- 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 an object that refers to component types
  - Limited number of these – correspond directly to type constructors in the language (record/struct, array, function,…)

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 you chose to represent symbol tables for classes. i.e., class symbol tables might be useful as the representation of the class type.)

Array Types

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

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

Array Types for Pascal

- 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 PascalArrayType 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)

Equivalence of 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

  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

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/review
- Then
  - Runtime representation of classes, objects, data, and method stack frames
  - Assembly language code for higher-level language statements