Overview of Semantic Analysis

Lecture 12

Outline

- The role of semantic analysis in a compiler
  - A laundry list of tasks
  - Errors difficult or impossible to detect earlier
  - Tied more to "meaning" of program rather than appearance
- Scope, OO inheritance
  - Implementation: symbol tables

- Types (of data, but also functions, methods)
  - Type of a function or method includes number and types of parameters and return value
  - Type of an array: base type, index set [how many subscripts?]
  - Type of other objects: collection of names and types

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal sequences (non-tokens).
  - Collects names, keywords, operators, numbers, etc.

- Parsing
  - Allows inputs with well-formed parse trees, rejects others.
  - Builds an abstract syntax tree.

- Semantic analysis
  - Last “front end” phase
  - Catches all static type errors in a strictly typed language
  - May generate more kinds of errors and warnings
  - Any program errors detected here can save time in debugging
  - Some errors cannot be found even yet: (duplicate library entry points?)

Why a Separate Semantic Analysis?

- Parsing cannot catch some errors

- Some language constructs are not context-free
  - Examples:
    - Identifier declaration and use.
    - Parameter / argument count agreement.
    - [declare   foo:int   ……   foo=3 …..] //good, foo is defined
    - [declare   bar:int   ……   foo=4 …..] //assume foo not defined, bad
    - [foo(a,b):= …   x=foo(1,2,3)….]   // bad, disagreement

  - An "abstract version" of the variable declaration and use problem is to return "valid sentence" for items in this language:

$$\{ wcw \mid w \in (a + b)^* \}$$

- The 1st w represents the declaration; the 2nd w represents a use, and c is the intervening program text. This is not context free.

Why not CF?

- An abstract version of the problem is to return "valid sentence" for items in this language:

$$\{ wcw \mid w \in (a + b)^* \}$$

Also note [declare foo:int; declare foo:bool …] is not valid…

Another famous non-context free language is

$$\{ a^n b^n c^n \mid n>0 \}$$

That is, CFG cannot count 3 things; recall that DFA cannot count 2…

Proof is based on "pumping lemma for CF languages" similar to proof that $a^n b^n$ is a problem for DFA. The texts first prove "Ogden's Lemma" that, if the language of a CFG is infinite, then for strings u v w x y,

$S \Rightarrow^* uvwxy$, but also $S \Rightarrow^* u\ v^i\ w\ x^i\ y$. So if $S \Rightarrow^* u\ a^i\ b^i\ c^i\ y$ then it also means $S \Rightarrow^* u\ a^{i+1}\ b^i\ c^{i+1}\ y$ [beyond scope of CS164, but not hard]

What Does Semantic Analysis Do?

- Checks of many kinds . . .
- MJ **typechecking program should check**:
  1. All identifiers are declared
  2. Types and numbers of arguments and return values agree with use (declarations, method bodies)
  3. Reserved built-in functions (e.g. Print) are not misused
  4. Arithmetic is performed only on numeric values
  5. Boolean expressions are used for conditionals
  6. Arrays are instantiated properly
  7. Declared identifiers are used (otherwise suspicious unused names!)

BIG WARNING "Same type" is subtle. Subclass object is an instance of its superclass too.

Other items can also be checked, e.g. no two method parameters have the same name! no two local variables/methods/ have the same name. No built-in functions are redefined. . .

What Does Semantic Analysis Do?

- Each programming language processor will check some items but maybe not others on that list.
- Which items can, in principle, be checked depends on the language definition
- Can Lisp do semantic analysis?
  - It must do *some* if "good" compiling is done
  - It can do some analysis even when not necessary… when processing "defun" even if not compiling
  - (Analyzing MCE from CS61a)
  - If interpreting, it sometimes (must?) wait.
    - (defun foo(x)(foo x x x x x)) ;wrong. When is it noticed?
    - (defun foo(x)(let ((y 1)(y 2)) y); what is returned?

Scope

- Matching identifier declarations with uses
  - Important static analysis step in most languages
  - Including MJ, (even Lisp)
  - In principle, the same name can be used repeatedly, though it might not be great "software engineering" to reuse names often.
  - (name X scope) → (declaration) //compiletime
  - (name X scope) → (location [or value]) //runtime

What's Wrong?

- Example 1

```
        class Fac {

    public int ComputeFac(int num){
    int num_aux ;  int foo;
    if (num < 1)
        num_aux = 1 ;
    else
        num_aux = num * (this.ComputeFac(num-1)) ;
    return num_aux ;
    }
```

Scope (Cont.)

- The *scope* of an identifier is the portion of a program in which that identifier is "accessible" – its binding to a value can be used

- The same identifier may refer to different things in different parts of the program
  - That is, if different scopes for same name don't overlap

- An identifier may have restricted scope
  - Names and their values and types may be **shadowed** in different ways; binding by method/function call, local vars

Static vs. Dynamic Scope

- Most languages these days have *static* scope
  - Scope depends only on the program text, not run-time behavior, and can be checked at compile-time (or type-checking time)
  - Java, Common Lisp and Scheme have static scope

- A few languages are *dynamically* scoped
  - Lisp ("special" variables only), SNOBOL
  - Access to values in dynamic scope depends on execution history of the program

Static Scoping Example (in Common Lisp)

```
(let ((x 1)                          ;;x bound to 1
      (y 2))
    (+ y                             ;;y bound to 2
      (let((x 3))
         x                           ;;x bound to 3, returned from let
    ))
```

Static vs Dynamic Scoping Example in CL

```
(defun foo(x) (bar)) ;; regular, static scope
(defun bar() (print x))
(foo 3) → attempt to take value of unbound
   variable x

(defun foo(x)(declare (special x)) (bar))
(defun bar()  (declare (special x)) (print x))
(foo 3) → 3
3
```

Functional Scoping Example in CL

```
(setf r (let((x 5))
  #'(lambda()  (format t "x=~s,y=~s" x y))))
(setf x 3)
(setf y 'toplevel)
(funcall r) → x=5, y=toplevel

(let ((y 'middle)) (funcall r)) →  x=5, y=toplevel
```

Environments!

Scope in MiniJava

- MiniJava identifier bindings are introduced by
  - variable type declarations produce default bindings
  - Formal parameters id's
- Classes, methods and variable names are in the same "namespace."
- Can we redefine boolean or int as a Class or are they in a separate namespace?

“Scope” in CL is more complex

- Generally lexical scope for names.
- Some declarations (e.g. fixnum, special) can be file-wide
- “Defstructs” are global
- Some names cannot be rebound (e.g. car, cdr) without major effort
- Packages provide another scoping/ hiding mechanism– you can redefine car and cdr and use them in a package to shadow the regular programs.
- Details here are not important except to be aware that you haven’t seen all the possibilities!
- Contrary to Scheme, there are separate spaces for names of functions and variables.
- So (setf car ‘(3 4)) (car car) is ok.
- (setf list #’+) (apply list car) -> 7 (apply #’list car)->(3 4)

“Scope” in C is almost unused

- Variables declared in Blocks, rarely used
- Local scope for parameters and locals
- Static external global

Implementing the Most-Closely Nested Rule

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*

- When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined

Implementing . . . (Cont.)

- Example: the scope of variable bindings is one subtree

```
Class foo{int x;
     E ;}
```

- x can be used in subtree E. In a classic lexical scope situation, we can hide x in the subtree E by defining a sub-block like {block int x; ….}

In a class/inheritance setup, we hide any x in parent by

Class foo extends parent{ int x; E;}

Symbol Tables

- Consider again: {int x; E;}
- Idea:
  - Before processing (e.g. typechecking, evaluating) E, add (type, default value) definition of x to current definitions, overriding any other definition of x

  - After processing E, remove definition of x and restore old definition of x

- A *symbol table* is a data structure that tracks the current bindings of identifiers, e.g. the name $\rightarrow$ type correspondence

Runtime analogs of Symbol Tables can hold values.

;Because we know so much about them, we can
;use very simple structures: a stack to do
common lisp variable bindings

```
(let ((a 3))
   (let  ((a 1)
            (b a))
(+ a b))
```

```
(let ((a 3))
   (let* ((a 1)   ;sequential
            (b a))
(+ a b))
```

A *Simple* Symbol Table Implementation, C-style

- Structure is a stack

- Operations
  - add_symbol(x) push x and associated info, such as x's type, on the stack
  - find_symbol(x) search stack, starting from top, for x. Return first x found or NULL if none found
  - remove_symbol() pop the stack

A *Simple* Lisp Symbol Table Implementation

- Structure is the stack used by lisp to call and return functions.
- Storage is by (rebinding) symbol table ST to (cons newbinding ST)

A *Simple* Lisp Symbol Table Implementation

```
(setf globals '((glob1 . integer) (glob2 . string)))

(defun dotypechecking (expression symbol-table)
  ;; POINT A
  ;; look in expression.  If there is a reference to
  ;; x in expression, look up x in symbol-table to
  ;; find type, or value or other property. ...
  ;; by (assoc x symbol-table) . Check that it is being used correctly.

  ;; If the expression is a ClassDecl in MJ, and defines new variable x say
;; (VarDecls (VarDecl IntType (id x  lc)))      (Statements  ….)
  ;; then  REBIND this way.
  (dotypechecking STATEMENTS (cons(cons 'x stuff) symbol-table)) ;rebind symbol-
table.
  ;; after you return,
  ;; if you continue processing more expressions, symbol-table
  ;; is same as POINT A
  ;; etc.
  ;; If you are finished, return typechecked results.
  ;; in the process of returning, the argument symbol-table is
  ;; popped off lisp run-time stack.
```

A *Simple* Lisp Symbol Table Implementation

.

Thus add_symbol is essentially cons.
find_symbol is assoc
remove_symbol is unnecessary! .. Removed by return

if there are several x's on symbol-table. No sweat.
assoc get the right one.

Limitations

- The simple symbol table works for let
  - Symbols added one at a time
  - Declarations are perfectly nested
  - Could add a bunch of bindings all at once
    - (dotypechecking expr (append list_of_bindings symboltab)

- What if your language allows same name for type, variable, function… distinguished by usage. Need fancier lookup.
- All names are handled lexically

- Other problems?

A *Fancier* Symbol Table

- Find _typename(x) ➔ find_symbol(x,kind='type)
- Find_varname(x) ➔ find_symbol(x,kind='var)
- Find_funname(x) ➔ find_symbol(x,kind='fun)
- Rebinding ➔ make_ST_entry(x,kind=type) These turn out to be trivial Lisp programs; if assoc returns the wrong kind of entry, keep looking.

Function (method) Definitions

- Method names can be used in the MJ text of a program before being defined. (mutually recursive, even)
- So we can't check use-definition agreement using a sequential stack-based symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all function names, arg-types, return types
  - Pass 2: Do the checking of all the bodies
- Semantic analysis can require multiple passes
  - Usual design requires 1, maybe 2.
  - Type inference, badly designed, can use many.. (Ada)

Why extra passes?

- Consider a language in which a:=b+c means, if a has type double-float, b and c are single-float: convert b to double, c to double, add double and store in a.
- If we parse bottom up, we don't know what the type is of a until "too late"
- synthesized vs. inherited attributes.

Not everyone likes a stack

- Assoc takes time O(n) for n items on the stack. What if you have thousands of identifiers? (AWA page 110)?

Thousands of identifiers?

- Well, how likely is THAT? It happens in C code with piles of #include foo.h, .. But it doesn't happen in lisp; a cost of checking (pro/con)..
- A hash-table typically is used then: every time one enters/leaves the scope of bindings, they are inserted/deleted from the hash-table, and the OLD bindings if any, are restored.

Still maybe you are in a hurry

- What's your hurry?
  - If you are using this as a RUN TIME mechanism for finding the values of variables, a stack implemented as a list is not good; a stack as an array may be ok.
  - E.g. X:=X+1 becomes:
    - Find X on the stack O(n) or O(1) or memoized
    - Change its value
    - What if X is very far away on the stack??
    - Advantage: changing scope is trivial push/pop.

    - Sometimes O(n) used in dynamic scope languages, called deep binding: cheap binding, expensive access.

Not everyone likes a stack (continued)

- Another runtime stack tactic
  - You can always find the value for anything named X in a fixed location in a hash table or array, say location 1234
  - Changing its value is easy.
  - When a "new X" is needed in some scope, store the contents of 1234, or the "old X" on a stack. Initialize location 1234 to the value for the new X. Any reference to the new X goes to 1234. When the process exits the scope of new X, pop the value of the old X off the stack and put it in 1234.
  - This is called shallow binding, used in dynamic scope languages (old lisp, also "special" variables in CL). Cheap access, expensive binding.

Block structured symbol tables

- Here's another mechanism
  - Put all variables/bindings at a given lexical level L in an array, vector or hashtable
  - Make a collection of pointers to each lexical level L, L-1 …1, 0=global.

Now to look up the value/type/etc of X, look for X in lexical level L, then L-1, until you find X or give up… So you look in L hash-tables, rather than searching item by item through the stack.

When you leave a scope, you abandon the whole level L hashtable.

If you can actually pre-calculate that X is the 4th item in level 3 then you may be able to access its value in one indexed memory access, e.g. if you have level 1, 2, 3's… address in a register. This is the usual situation for lexically scoped languages. For historical reasons this mechanism is sometimes called a DISPLAY

Summary: basic static semantic analysis

- Information generated by a pass over the AST can be used
  - For type analysis
  - Checking other details of proper usage
- Must be coordinated with analysis of scope keep track of name – attribute correspondences