CS 536 Announcements for Wednesday, March 13, 2024

**Programming Assignment 3** – due Friday, March 15

**Midterm 2** – Thursday, March 21

**Last Time**
- building a predictive parser
- predictive parsing and syntax-directed translation

**Today**
- static semantic analysis
- name analysis

**Next Time**
- continue name analysis
- exam review

---

Static Semantic Analysis

Two phases

- **name analysis** (aka **name resolution**)
  - for each scope
    - process declarations
      - add entries to symbol table
      - report multiply-declared names (error)
    - process statements
      - update IdNodes to point to appropriate symbol table entry
      - find uses of undeclared variables (error)

- **type checking**
  - process statements
    - use symbol table to find types of each expression & sub-expression
    - find type errors (error)
Why do we need this phase?

**Code generation**
- different operations use different instructions
  - consistent variable access
  - integer addition vs floating-point addition
  - operator overloading

**Optimization**
- symbol table entry serves to identify which variable is used
  - can help in removing dead code (with some further analysis)
  - *note*: pointers can make these tasks hard

**Error checking**

---

**Semantic error analysis**

For non-trivial programming languages, we run into fundamental undecidability problems:
- does the program halt?
- does the program crash?

Even with simplifying assumptions (sometimes infeasible in practice) as well
- combinations of thread interleavings
- inter-procedural data analysis

*In general - can’t guarantee the absence of errors*

**Goal of static semantic analysis:** catch some obvious errors
- **undeclared identifiers**
- **multiply-declared identifiers**
- **ill-typed terms**
Name analysis

Associating IDs with their uses

Need to bind names before we can do type analysis

Questions to consider:

- What definitions do we need about identifiers?
- How do we bind definitions and uses together?

Symbol Table

= (structured) dictionary that binds a name to information we need

Each entry in the symbol table stores a set of attributes:

- kind — tuple, variable, function, class
- type — integer, integer x string — logical, tuple
- nesting level
- runtime location — where in memory is it stored

Symbol table operations

- insert entry
- lookup name
- add new sub-table
- remove/forget a sub-table

Implementation considerations

- efficiency of access is important
- size unknown ahead of time — need expansion to be graceful and efficient
- don't need to delete entries
Scoping

**scope** = block of code in which a name is visible/valid = lifetime of a name

No scope (flat name scope)

- assembly, FORTRAN name is visible throughout program

Static/most-nested scope - starting with ALGOL 60
  - block structure
  - nested visibility
  - easy to tell which def of a name applies
  - new def's apply to local scope

- name scopes - limit region of definition

Kinds of scoping

static - can tell at compile time the correspondence between use & declaration

dynamic - correspondence is determined at run time

**Dynamic scoping example**

What does this print, assuming dynamic scoping?

```java
void main() {
    int x = 10;
    f1();
    g();
    f2();
}
void f1() {
    String x = "hello";
    g();
}
void f2() {
    double x = 2.5;
    f1();
    g();
}
void g() {
    print(x);
}
```

Output

```
hello
10
hello
2.5
```
Scoping issues to consider

Can the same name be used in multiple scopes?

variable shadowing

Do we allow names to be reused in nesting relations?

void verse(int a) {
    int a;
    if (a) {
        int a;
        if (a)
            int a;
    }
}

void chorus(int a) {
    int chorus;
}

What about when the kinds are different?

overloading

Same name; different type

int bridge(int a) { ... }
bool bridge(int a) { ... }
bool bridge(bool a) { ... }
int bridge(bool a, bool b) { ... }

How do we match up uses to declarations?

Determine which uses correspond to which declarations

int k = 10, x = 20;
void foo(int k) {
    int a = x 2 ;
    int x = k 4 ;
    int b = x 6 ;
    while (...) {
        int x;
        if (x 8 == k 4 ) {
            int k, y;
            k 9 = y 10 = x 8 ;
        }
        if (x 8 == k 4 ) {
            int x = y
        }
    }
}

}
Scoping issues to consider (cont.)

Where does declaration have to appear relative to use?

forward references

How do we implement it?

```java
void music(){
    lyrics();
}
void lyrics(){
    music();
}
```

Scope example

What uses and declarations are OK in this Java code?

```java
class animal {

    // methods
    void attack(int animal) {
        for (int animal = 0; animal < 10; animal++) {
            int attack;
        }
    }
    int attack(int x) {
        for (int attack = 0; attack < 10; attack++) {
            int animal;
        }
    }
    void animal() { }

    //fields
    double attack;
    int attack;
    int animal;  //ok
}
```
Name analysis for base

base is designed for ease of symbol table use
  - ** statically scoped**
  - global scope plus nested scopes
  - all declarations are made at the top of a scope
  - declarations can always be removed from table at end of scope

base scoping rules
  - use most deeply nested scope to determine binding
  - variable shadowing allowed
  - formal parameters of function are in same scope as function body

Walk the AST
  - put new entries into the symbol table when a declaration is encountered
  - augment AST nodes where names appear (both declarations & uses) with a link to the relevant object in the symbol table

Symbol-table implementation
  - use a list of hashmaps

---

**Example**

```plaintext
void f(integer a, integer b) {
    logical x.
    while ... [
        integer x, y.
        ...
    ]
}

void g() {
    f().
}
```

---

Each hashtable corresponds to one scope (i.e., contains all decls for that scope)
Symbol kinds
Symbol kinds (= types of identifiers)
• variable have a name, a type
• function declaration has a name, return type, list of parameter types
• tuple declaration has a name, list of fields (types w/ names), size

Implementation of Sym class
Many options, here's one suggestion
• Sym class for variable definitions
• FnSym subclass for function declarations
• TupleDefSym subclass for tuple type definitions
• TupleSym subclass for when you want an instance of a tuple

Symbol tables and tuples
• Compiler needs to
  • for each field: determine type, size, and offset with the tuple
  • determine overall size of tuple
  • verify declarations and uses of something of a tuple type are valid
• Idea: each tuple type definition contains its own symbol table for its field declarations
  • associated with the main symbol table entry for that tuple's name
  ie, global