CS 536 Announcements for Tuesday, April 5, 2022

Last Time
- static semantic analysis
- name analysis
  - symbol tables
  - scoping
- exam review

Today
- name analysis

Next Time
- type checking

Static Semantic Analysis

Two phases
- name analysis
- type checking

Name analysis
- for each scope
  - process declarations – add entries to symbol table
  - process statements – update IdNodes to point to appropriate symbol table entry
- each entry in symbol table keeps track of: kind, type, nesting level, runtime location
- identify errors
  - multiply-declared names
  - uses of undeclared variables
  - bad struct accesses
  - bad declarations

Scoping
- kinds of scoping
  - static
  - dynamic

Output: annotated AST

Scope: block of code in which a name is visible/valid
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
```

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;
}
```
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?

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

What about when the kinds are different?

```java
void chorus(int a) {
    int chorus;
}
```

**overloading**

Same name; different type

```java
int bridge(int a) { ... }
bool bridge(int a) { ... } // not allowed in Java
bool bridge(bool a) { ... }
int bridge(bool a, bool b) { ... }
```

Where does declaration have to appear relatative to use?

**forward references**

How do we implement it?

```java
void music(){
    lyrics();
}
void lyrics() {
    music();
}
```
Scoping issues to consider (cont.)

How do we match up uses to declarations?
Determine which uses correspond to which declarations

```c
int k = 10, x = 20;
void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
        int x;
        if (x == k) {
            int k, y;
            k = y = x;
        }
        if (x == k) {
            int x = y;
        }
    }
}
```

Name analysis for minim

minim 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

minim 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
  - efficient add & lookup
  - remove a lot of names from a scope at once
void f(int a, int b) {
    double x;
    while (...) {
        int x, y;
        ...
    }
}
void g() {
    f();
}

Symbol kinds

Symbol kinds (= types of identifiers)

- variable
  - name, primitive type

- function declaration
  - name, return type, list of parameter types

- struct declaration
  - 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
- StructDefSym subclass for struct type definitions
- StructSym subclass for when you want an instance of a struct
Name analysis and structs

Symbol tables and structs

- Compiler needs to
  - for each field: determine type, size, and offset with the structure
  - determine overall size of structure
  - verify declarations and uses of something of a struct type are valid

- Idea: each struct type definition contains its own symbol table for its field declarations
  - associated with the main symbol table entry for that struct's name

Relevant minim grammar rules

```
decl ::= varDecl
     | fnDecl
     | structDecl // struct defs only at top level
     ;
varDeclList ::= varDeclList varDecl
              | /* epsilon */
              ;
varDecl ::= type id SEMICOLON
          | STRUCT id id SEMICOLON
          ;
...
structDecl ::= STRUCT id LCURLY structBody RCURLY SEMICOLON
              ;
structBody ::= structBody varDecl
            | varDecl
            ;
...
type ::= INT
     | BOOL
     | VOID
     ;
loc ::= id
     | loc DOT id
id ::= ID
    ;
```
Definition of a struct type

```c
struct Point {  
  int x;
  int y;
};

struct Color {  
  int r;
  int g;
  int b;
};

struct ColorPoint {  
  struct Color color;
  struct Point point;
};
```

Declaring a variable of type struct

```c
struct Point pt;
struct Color red;
struct ColorPoint cpt;
```
Accessing fields of a struct

pt.x = 7;
pt.y = 8;
pt.z = 10;

red.r = 255;
red.g = 0;
red.b = 0;

(cpt.point.x) = pt.x;
cpt.color.r = red.r;
cpt.color.g = 34;

Recursively process LHS first
1) if L child is an id
   - check id has been declared of struct type
   - get sym-tab for that struct type
   - lookup R child in sym-tab
2) if L child is a * access
   - recursively process L child
   - if sym-tab in * access is null
     then error
     else look up R child in sym-tab