CS 536 Announcements for Monday, April 3, 2023

Last Monday

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

Today

- name analysis

Next Time

- type checking

Static Semantic Analysis

Two phases

- name analysis → P4 output: annotated AST
- type checking → P5

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 `record` accesses
  - bad declarations

Scoping

- **scope** = block of code in which a name is visible/valid
- kinds of scoping
  - **static** – correspondence between use & declaration made at compile time
  - **dynamic** – correspondence between use & declaration made at run time

Dynamic scoping example

What does this print, assuming dynamic scoping?

```
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?

```
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;
}
```

100 allowed!

overloaded, can't only differ in return type

cannot do

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;
        }
    }
}
```

← variable shadowing

What about when the kinds are different?

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

overloading

Same name; different type

```
int bridge(int a) { … }
bool bridge(int a) { … }
bool bridge(bool a) { … }
int bridge(bool a, bool b) { … }
```

← not allowed in Java

Where does declaration have to appear relatative to use?

forward references

How do we implement it?

this would require 2 passes
- 1 to fill symTab
- 1 pass to use symtab

```
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

```
int 1 k = 10, 2 x = 20;
void 3 foo(int 4 k) {
    int 5 a = x 2 ;
    int 6 x = k 4 ;
    int 7 b = x 6 ;
    while (...) {
        int 8 x;
        if (x 8 == k 2 ) {
            int 9 k, 10 y;
            k 9 = y 10 = x 6 ;
        }
        if (x 8 == k 4 ) { error use of undeclared variable
            int 11 x = y X ;
        }
    }
}
```

Name analysis for brevis

brevis 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

brevis 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

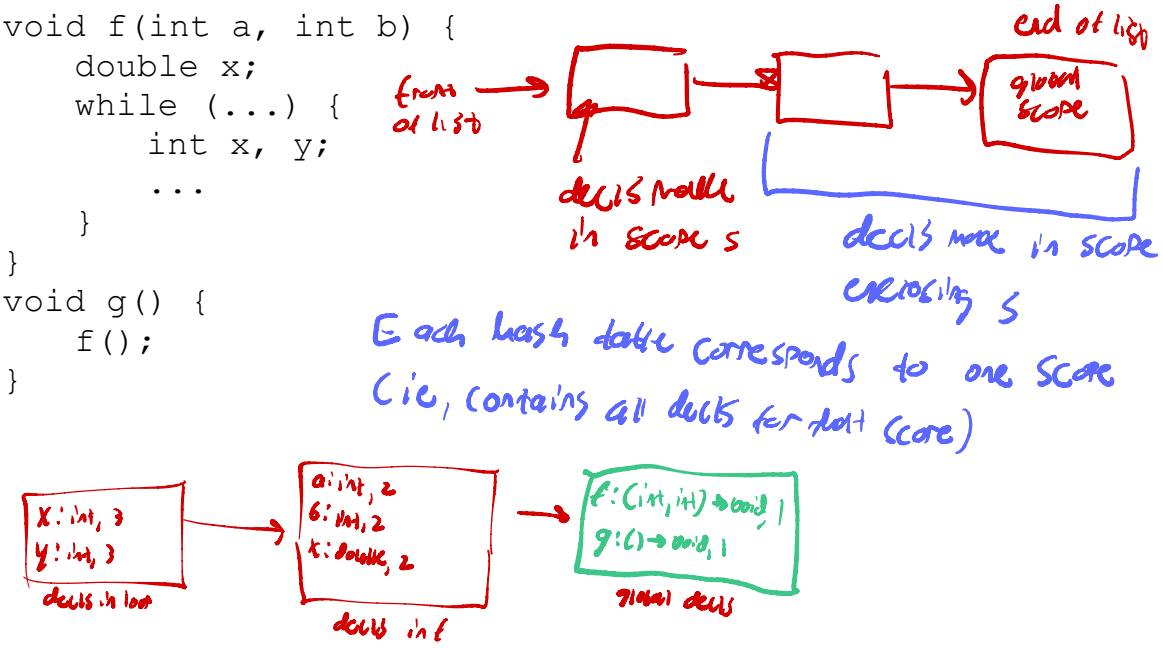
- put new entries into the symbol table when a declaration is encountered
- augment AST nodes where names appear (both declarations & uses) with a link to the relevant object in the symbol table

Symbol-table implementation

- use a list of hashmaps

Example

```
void f(int a, int b) {
    double x;
    while (...) {
        int x, y;
        ...
    }
}
void g() {
    f();
}
```



Symbol kinds

Symbol kinds (= types of identifiers)

- variable

  has a name, type
- function declaration

  has a name + return type + list of parameter types
- record declaration

  has 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
- `RecordDefSym` subclass for `record` type definitions
- `RecordSym` subclass for when you want an instance of a `record`

Name analysis and `records`

Symbol tables and `records`

- Compiler needs to
  - for each field: determine type, size, and offset with the record
  - determine overall size of record
  - verify declarations and uses of something of a `record` type are valid
- Idea: each `record` type definition contains its own symbol table for its field declarations
  - associated with the main symbol table entry for that `record`'s name

Relevant brevis grammar rules

```
decl          ::= varDecl
              | fnDecl
              | recordDecl    // record defs only at top level
              ;
varDeclList   ::= varDeclList varDecl
              | /* epsilon */
              ;
varDecl       ::= type id SEMICOLON
              | RECORD id id SEMICOLON
              ;
...
recordDecl    ::= RECORD id LPAREN recordBody RPAREN SEMICOLON
              ;
recordBody    ::= recordBody varDecl
              | varDecl
              ;
...
type          ::= BOOL
              | INT
              | VOID
              ;
loc           ::= id
              | loc DOT id
id            ::= ID
              ;
```

Definition of a `record` type

```
record Point (
    int x;
    int y;
);

record Color (
    int r;
    int g;
    int b;
);

record ColorPoint (
    record Color color;
    record Point point;
);
```

Make sure not already in sym tab
- Create a sym tab for this record + store in sym for record's name
- for each varDecl in body of record
  if type is record, make sure record type is in global (main) sym tab

make sure field is not in record's sym tab (before adding)

integer

Declaring a variable of type `record`

```
record Point pt;
record Color red;
record ColorPoint cpt;
```

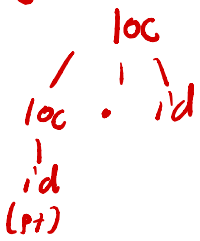
look up globally
- make sure it exists + it's a record

look up (locally)
- make sure it doesn't exist

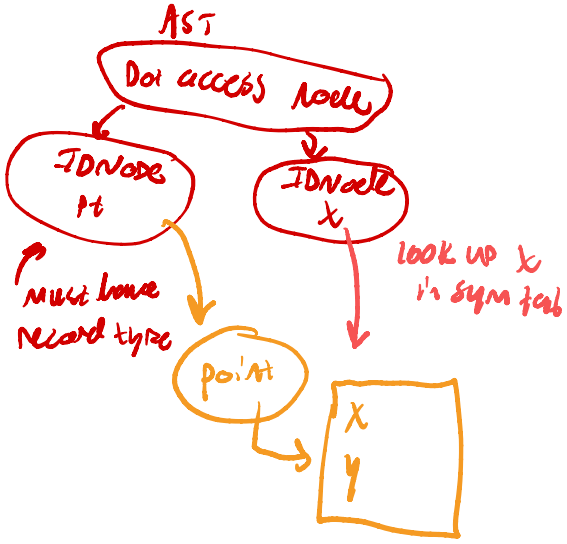
Accessing fields of a `record`

```
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;
```

Parse tree



AST

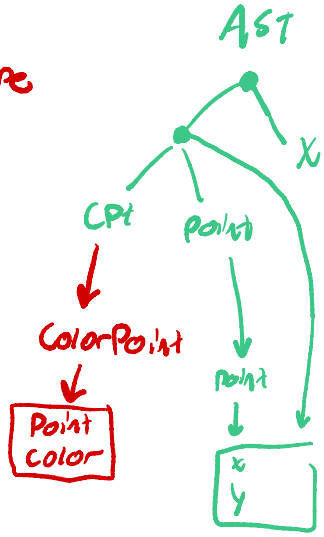


Recursively handle L child

If L child is an identifier

- check identifier has been declared of record type
- get symbol table for that record type
- lookup Right child in the symbol table

AST



If L child is a dot-access

- recursively process L child
- if symbol table in • access node is null

  then error

  else look up R child in sym tab

If R child is a record type

- then Set reference in • access to records sym tab

- else Set ref to null