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 multply-declared names
    - process statements
      – update IdNodes to point to appropriate symbol table entry
      – find uses of undeclared variables
- **type checking**
  - process statements
    – use symbol table to find types of each expression & sub-expression
    – find type errors

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

Goal of static semantic analysis: catch some obvious errors

-
-
-

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
- type
- nesting level
- runtime location

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
- don't need to delete entries

Scoping

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

No scope (flat name scope)

Static/most-nested scope

Kinds of scoping

static

dynamic

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);
}
```

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;
        }
    }
}
```

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) { … }
```

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   ;
    int   x = k   ;
    int   b = x   ;
    while (...) {
        int   x;
        if (x   == k   ) {
            int   k,   y;
            k   = y   = x   ;
        }
        if (x   == k   ) {
            int   x = y   ;
        }
    }
}
```

Scoping issues to consider (cont.)

Where does declaration have to appear relatative to use?

forward references

How do we implement it?

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

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;
}
```

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

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

Symbol kinds

Symbol kinds (= types of identifiers)

- variable
- function declaration
- `tuple` declaration

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 `tuple`s

- 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