CS536 Lecture 15

Tuesday 17 March 2015

Last class:

- SDT Actions for Parsing

Today:

- Static Semantic Analysis
    - Name Analysis

(Static) Semantic Analysis

Syntax vs. Semantics

What can the parser guarantee and what can't it guarantee?

Static:

Two phases:

- Name analysis (name resolution)
    - Build the symbol table. For each scope:
        - Process declarations, add to symbol table (or report multiple declarations).
        - Process statements, update ID Nodes to point to symbol table entry (or report undeclared variable).
- Type analysis
    - Process statements: Use symbol table info to determine type of each expression.
    - Find type errors.

Why do we need this phase?

Code generation:

Want to make sure the right instructions are used for operands

Optimization:

Remove dead code

Weaken the type if necessary (int to bool, for example).

(using information in the symbol table)

Error checking

Some problems are undecidable (halting, crashes)

Some are hard (threaded programs, interprocedural dataflow)

Some simpler ones we can catch:

- Undeclared or multiply declared identifiers
- Ill-typedness

Name Analysis

Name Analysis: To associate identifiers with their uses.

Before typing uses though, bind names to their types.

What information do we need? i.e. What do we store in the symbol table?

How do we connect definitions to uses? Scope.

Symbol Table

A table binding a name to information we need:

- Kind (struct, variable, function, class)
- Type (int, int x string → bool, struct)
- Nesting level
- Runtime location (where it's stored in memory or offset into activation record)

Operations:

Insert entry
Lookup
Add new table (scope)
Remove/forget a table (scope)

Considerations: Efficiency of lookup, graceful expansion.

Scope

Scope: Block of code in which the use of a name is visible/valid.

Some languages have a “flat” scope (assembly language, FORTRAN).

Others have a block structure (nested visibility). C/C++ and Java.

Kinds of scope:

Static: Correspondence between definition and uses can be determined at compile time.

Dynamic: Correspondence is determined at runtime.

```
void print_a() { print a; }

void foo() { a = 1; print_a(); }

void bar() { a = 2; print_a(); }

int main() {
     ...

     foo();
     bar();

}
```

Scoping Issues

Shadowing: Do we allow names to be re-used in nesting relations?

```
void foo(int a) {
    int a;
    if (a) {
        int a;
        ...
    }
}
```

What about when the types are different?

```
void bar(int x) {
    bool x;
    int bar;
    float y;
    bool y;
}
```

Overloading:

```
int myfun(int x) { };
bool myfun(int x) { };
bool myfun(bool x) { };
bool myfun(int x, bool y) { };
```

Where must a declaration be relative to use? (forward-referencing)

What are the boundaries of a scope?

Scoping rules in Java/C++

Name Analysis for C--

What scoping rules will we allow?

- Static scoping
- Global scope + nested scopes. Most deeply nested scope determines binding.
    - Variable shadowing is allowed (in different scopes only).
    - Struct definition shadowing?
- All declarations at the top of a scope
- The end of a block is the end of its scope (symbol table entries removed).
- Parameters and local variables (at beginning of function) have the same scope.

Symbol Table considerations knowing the above.

Symbols in C--

Identifier types:

- Variables: Carries a name, primitive type.
- Function Declarations: Carries a name, return type, list of parameter types.
- Struct Definitions: Carries a name, list of fields (types with names), size.

Implementing the `Sym` Class (some suggestions):

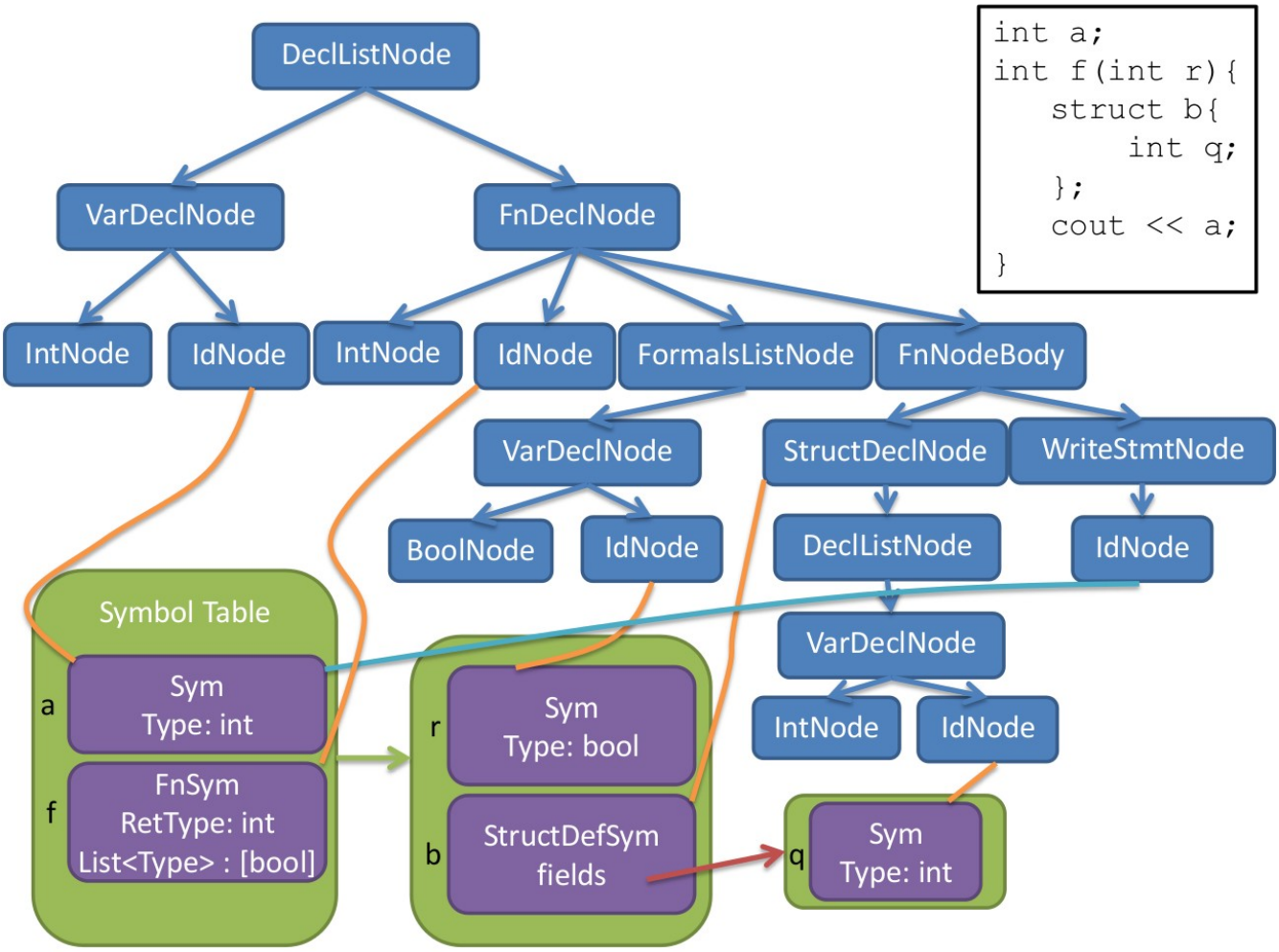
- `Sym` class for variables definitions
- `FnSym` subclass for function declarations
- `StructDefSym` for struct *type* definitions, containing it's **own** symbol table for its field definitions.
- `StructSym` for struct instances.

Name Analysis with ASTs

Walk the AST (similar to the `unparse()` method in P3):

- Augment AST nodes with a reference to the relevant `Sym`.
- Whenever you come across a new declaration, add an entry in the symbol table.

Example:



Type Analysis

What is a type?

- Classification identifying kinds of data
- A set of possible values with a variable can possess
- Operations that can be done on member values
- A representation (e.g. in memory)

Intuition: What might this mean? Can you do it?

```
int a = 0;
int* ptr_a = &a;
float frac = 1.2;
a = ptr_a + frac;
```

Type Checking

Components of a type system:

- Base/primitive types (built-in)
    - e.g. int, bool, void
- Rules for building types (type constructors)
    - e.g. struct (in our language)
- Means of determining whether types are compatible
    - e.g. is it permissible to use bool in place of int? (Not in C--)
- Rules for inferring the types of expressions

Type Rules:

For every operator:

What types can the operand(s) be?

```
int x;
x = 3.4
```

Rules for implicit conversion (coercion)

```
7 + 3.4
```

What type is the result?

```
if (x = 4) { … }
```

Type checking is to walk the AST to determine types of expressions using the type rules of the language and find type errors, if any.