Compiling Techniques

Lecture 8: Semantic Analysis

Christophe Dubach

6 October 2017

Beyond Syntax

There is a level of correctness deeper than syntax (grammar).

Example: broken C program

```
foo(int a, b, c, d) {...}

bar() {
  int f[3],g[0],h,i,j,k;
  char * p;
  foo(h,i,"ab",j,k);
  k = f*i+j;
  h = g[17];
  printf("%s,%s\n",p,q);
  p = 10;
}
```

What is wrong with this program?

- declared `g[0]`, used `g[17]`
- wrong number of arguments for `foo`
- `"ab"` is not an `int`
- used `f` as scalar but is array
- undeclared variable `q`
- `10` is not a character string

Table of contents

To generate code, the compiler needs to answer many questions about names:

- is x a scalar, an array or a function?
- is x declared? Are there names declared but not used?
- which declaration of x does each use reference?

about types:

- is the expression x*y+z type-consistent?
- in a[i,j,k], does a have three dimensions?
- how many arguments does foo take? What about printf?

about memory:

- where can z be stored? (register, local, global heap, static)
- does *p reference the result of a malloc()?
- do p and q refer to the same memory location?

...

Name Analysis

The property "each identifier needs to be declared before use" depends on context information.

- In theory it is possible to specify this with a context-sensitive grammar
- In practice we define a context-free grammar (CFG) and identify invalid programs using other mechanisms enforcing language properties that cannot be expressed with a CFG

In order to check such a property, we need to find the declaration of each identifier. Additional constraints might exist depending on the specific language.

Different languages, different constraints

Example

```
...

void main() {
  i=3;
}
int i;

...
```

- Invalid in C
- Valid in Java

Scopes

Definition

The region where an identifier is visible is referred to as the identifier's scope.

This means it is only legal to refer to the identifier within its scope. Here identifier refers to function or variable name.

In addition, in our language, it is illegal to declare two identifiers with the same name if the are in the same scope (ignoring nesting). In our language we have two types of scopes:

- File scope (a.k.a. global scope)
- Block scope (a.k.a. local scope)

File scope (global scope)

Any name declared outside any block has file scopes. It is visible anywhere in the file after its declaration.

i has file scope

```
int i;
void main() {
  i = 2;
}
```

File scope

```
FileScope({i})
```

Block scope (local scope)

Any identifier declared within a block is visible only within that block. Procedure parameter identifiers have block scope, as if they had been declared inside the block forming the body of the procedure.

i,j have the same block scope

```
void foo(int i) {
 int j;
 i = 2;
 j = 3;
}
```

Block scope

```
BlockScope({i,j})
```

Nested scopes

Scopes are nested within each other.

Code

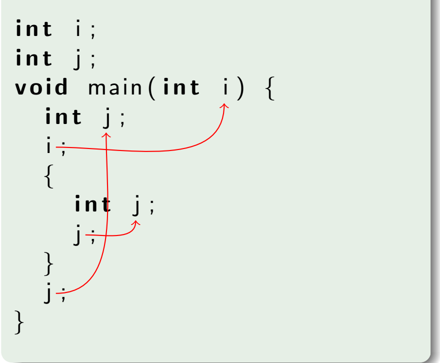
```
int i;
void main(int j) {
  int k;
  {
    int l;
  }
  {
    int l;
    int m;
  }
}
```

Nested scopes

```
FileScope(
  {i}
  BlockScope(
    {j,k}
    BlockScope(
      {l}
    )
    BlockScope(
      {l,m}
    )
  )
)
```

Shadowing occurs when an identifier declared within a given scope has the same name as an identifier declared in an outer scope. The outer identifier is said to be shadowed and any use of the identifier will refer to the one from the inner scope.

Legal example in C



```c
int i;
int j;
void main(int i) {
  int j;
  i;
  {
    int j;
    j;
  }
  j;
}
```

Illegal shadowing

Note that in some languages, such as Java, it is illegal to shadow local variables.

Illegal example in Java

```
public static void foo() {
  int i;
  for (int i = 0; i < 5; i++) // illegal to redeclare i
    System.out.println(i);
}
```

- Making this illegal help prevent potential bugs.
- However, Java does allow for shadowing of fields by local variables (if this was allowed, the introduction of a new field in a superclass might create problems in the sub-classes)

Illegal shadowing

In most languages, it is illegal to declare two identifiers with the same name if the are in the same scope (ignoring nesting). Here identifier refer to function or variable name.

Illegal example 1 in C

```
int i;
int i; // illegal
void main(int j) {
  int j; // illegal
  int k;
  int k; // illegal
}
```

Illegal example 2 in C

```
int i;
void i() { // illegal
}
```

Name Analysis

In order to perform name analysis, we need to define a few data structures:

Symbol Table

A symbol table is a data structure that stores for each identifier information about their declaration.

Symbol

A symbol is a data structure that stores all the necessary information related to a declared identifier that the compiler must know.

Scope

A scope is a data structure that stores information about declared identifiers. Scopes are usually nested.

Symbols

Symbol classes

```
abstract class Symbol {
  String name;
  boolean isVar() {...}
  boolean isProc() {...}
}
class ProcSymbol extends Symbol {
  Procedure p;
  ProcSymbol(Procedure p)
    { this.p = p; this.name = p.name}
}
class VarSymbol extends Symbol {
  VarDecl vd;
  VarSymbol(VarDecl vd)
    { this.vd = vd; this.name = vd.var.name;}
}
```

Scope and Symbol Tables

The symbols are stored in the symbol table within their scope.

Scope class

```
abstract class Scope {
  Scope outer;
  Map<String, Symbol> symbolTable;

  Scope(Scope outer) { ... };

  Symbol lookup(String name) { ... };
  Symbol lookupCurrent(String name) { ... };

  void put(Symbol symbol)
    { symbols.put(symbol.name,symbol); }
}
```

Exercise

1. Why are there two lookup methods?
2. Implements the lookup methods.

Vistor Implementation

We can now write our pass which will analyse names by creating a visitor which traverses the AST. The goals of the name analysis are to:

- ensure variables and functions are declared before used
- ensure variable and function declaration name are unique within the same scope
- save the results of the analysis back in the AST nodes:
  - a reference to the variable declaration for each variable use
  - a reference to the procedure declaration for each function call
  - this information is necessary for the later passes (*e.g.* type checking, code generation)

NameAnalysis visitor : variable declaration

```
class NameAnalysis implements ASTVisitor<Void> {

  Scope scope;
  NameAnalysis(Scopt scope) { this.scope = scope; };

  public Void visitVarDecl(VarDecl vd) {
    Symbol s = scope.lookupCurrent(vd.var.name);
    if (s != null)
      error();
    else
      scope.put(new VarSymbol(vd));
    return null;
  }
```

NameAnalysis visitor : block

```
public Void visitBlock(Block b) {
  Scope oldScope = scope;
  scope = new Scope(oldScope);
  // visit the children
  ...
  scope = oldScope;
  return null;
}
```

NameAnalysis visitor : variable use

```
public Void visitVar(Var v) {
  Symbol vs = scope.lookup(v.name);
  if (vs == null)
    error();
  else if (!vs.isVar())
    error();
  else // everything is fine, record var. decl.
    v.vd = ((VarSymbol) vs).vd;
  return null
}
```

Not just analysis!

The visitor does more than analysing the AST: it also remembers the result of the analysis directly in the AST node. Need to do this for variable uses and function calls.

Next lecture

- Type analysis