Compiler Design

Lecture 9: Semantic Analysis, part I: Name Analysis

Christophe Dubach

Winter 2025

Timestamp: 2025/02/06 17:20:00

Correctness Beyond Syntax

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

Example: broken C program

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

int 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;
  4 = i;
}
```

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
- cannot assign to an integer litteral
- no return statement for bar

Table of contents

To generate code, a 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

Name Analysis

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

- In theory, possible to specify this with a context-sensitive grammar
- In practice we use a Context-Free Grammar (CFG) for syntax and identify semantically invalid programs using other mechanisms

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

Name Analysis

Scopes

Scopes

Definition

The region where an identifier is visible is it'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 many languages, 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:

- Global scope (*e.g.* file)
- Local scope (*e.g.* block of code)

Can you think of other scopes?

Global scope

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

i has global scope

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

Global scope

```
GlobalScope({i})
```

Local scope

Any identifier declared within a block **{...}** of code is visible only within that block. Function parameter identifiers have local scope, as if they had been declared inside the block forming the body of the Function.

i,j have the same local scope

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

Local scope

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

Nested scopes

Scopes can be nested within each other.

C code example

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

Corresponding nested scopes

```
GlobalScope(
  {i}
  LocalScope(
    {j,k}
    LocalScope(
      {l}
    )
    LocalScope(
      {l,m}
    )
  )
)
```

Shadowing

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

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

Illegal shadowing

In some languages (*e.g.* 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 helps prevent potential bugs.
- However, Java does allow for shadowing of fields by local variables. Why?
  - if this were not 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, refers to a function or a variable name.

Illegal example 1 in C

```c
int i;
int i; // actually legal in C!!
void main(int j) {
  int j; // illegal
  int k;
  int k; // illegal
}
```

Illegal example 2 in C

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

Name Analysis

Data Structures

Name Analysis

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

Symbol

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

Symbol Table

A symbol table is a data structure that stores a mapping from symbol name (`String`) to the symbol.

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;
}
class FunSymbol extends Symbol {
  FunDecl fd;
  FunSymbol(FunDecl fd) {
    this.fd = fd;
    this.name = fd.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

```
class Scope {
  Optional<Scope> outer; // empty if top-level
  Map<String, Symbol> symbolTable = new HashMap();

  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. Implement the lookup methods.

Name Analysis

Implementation

Implementation

Let's write a pass to analyse names using pattern-matching.

The pass should:

- ensure variables and functions are declared before used
- ensure variable and function declarations 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 function declaration for each function call
  - this information is necessary for the later passes (*e.g.* type checking, code generation)

Variable Declaration:

```
int i;
```

NameAnalysis

```
class NameAnalysis {

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

  void visit(ASTnode node) {
    switch(node) {

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

Variable Use:

```
int i; // variable declaration
...
i+3;   // variable use
```

VarExpr class

```
class VarExpr {
  ...
  VarDecl vd;
}
```

NameAnalysis : variable use

```
...
case VarExpr ve → {
  Symbol sym = scope.lookup(ve.name);
  switch(sym) {
    case VarSymbol vs → ve.vd = vs.vd;
    case null,default → error();
  }
...
```

Not just analysis!

This does more than analysing the AST: it also remembers the result of the analysis directly in the AST node.

This information is necessary to identify which variable/function is used/called.

Code block:

```
...
{
   ...
}
...
```

NameAnalysis: block

```
...
case Block b → {
  // save current scope and create new one
  Scope oldScope = scope;
  scope = new Scope(oldScope);

  // visit the children
  ...

  //restore previous scope
  scope = oldScope;
  }
...
```

Next lecture

- Type analysis