Semantic Analysis

Part V

Semantic Analysis

- We have used a context-free grammar for parsing.
- Programming languages are not really context-free, we can define new names.

Programming Languages are not really context-free

- In many languages the user can declare variables, and types.
- Whether a variable is declared then depends on the *context*.
- Example:

```
{
 int x;
 x = i + 1;
}
```


Programming languages are not really context-free

- We solve this problem in two steps:
  - We define a context-free superset of the language which can be parsed easily.
  - We write a special purpose context-sensitive analyzer that weeds out illegal programs.

Type- and Name Analysis

- The purpose of *type analysis* is to find out the type of expressions and variables, by finding out which variable declaration a certain variable name refers to.
- The *name analysis* finds out if there are uses of identifiers which are not defined, or if there are illegal double definitions.
- Some languages allows *type-inference* where the type of an identifier can be inferred from its use, without any declaration.

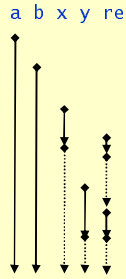
Scope

- Every identifier in a program has a *scope*, i.e., an area of program text, in which it can be referred to.
- Each language has its own *scoping-rules* that defines the scope of identifiers.

Scope in Expressive

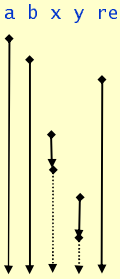
- Expressive has only one type of identifiers – variables (no functions, modules, or user defined types).
- The scoping rules that you need to implement are quite simple: the scope of a variable extends (lexically) from its first definition to the end of the program.
- The real scoping rules are that each "{" starts a new scope which is ended by a matching "}". (5 extra lines of code in the analyzer.)

Example of Scopes




```
readint a.
readint b.
if a < b {
  x:= a.
  res:= x * x.
} else {
  y:= b.
  res:= y * y.
}.
```


Example of Scopes

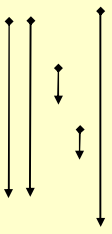



```
readint a.
readint b.
res := 0.
if a < b {
  x:= a.
  res:= x * x.
} else {
  y:= b.
  res:= y * y.
}.
print res.
```


Block Structured Languages

- Most programming languages, such as Java, have *block structured* visibility rules for identifiers.
- In these languages the scope of an identifier ends when the enclosing scope ends.
- The same identifier can mean different things in different blocks.
- Such languages often allow nested scopes and definitions. That is, a block can contain another block with new definitions.

Example of Scopes




```
class T{
 int res;
 void s(int a, int b){
  if (a < b) {
   int x= a;
   res= x * x;
  } else {
   double x= b * 1.0;
   res= b * b;
  };
 };
}
```


Different Types of Scopes.

- Many languages have different scopes for different types of identifiers.
- For example functions may be visible in the whole program, while variables are only visible within the defining block.
- Some languages, like Java, allows the programmer to modify the scope of a declaration with keywords like **private**, **public**, etc.

Representation of Context

- In a compiler we can represent context by a data structure which stores information about each identifier.
- We call this data structure a *symbol table*, and the stored information we call a *symbol table entry*.
- If a language allows nested scopes, the symbol table should be structured in the same way.

Symbol Table Entries

- A symbol table entry is a data structure, which contains all the information about a defined identifier that the compiler needs.
- In Java this can be implemented by a class Symbol (which can be sub-classed to add additional information).
- For a simple project like expressive we can get away with just the Symbol class, and we can link symbols together to create a symbol table

Symbols in the Project

- We can link symbols together to create a symbol table as a linked list. (This is not very efficient but easy to implement and sufficient for reasonably sized programs.)
- Nested scoping is easy to implement with a linked list.

Symbol.java

```
public class Symbol {
  public final String name;
  public final Type type;
  /** Next symbol in the symbol table. */
  public final Symbol next;

  public Symbol(String name, Type type, Symbol next) {
    this.name = name; this.type = type;
    this.next = next;
  }

  public String toString() {
    return name + ": " + type; }
}
```


Type.java

```
public final class Type {
  public static final Type Int = new Type();
  public static final Type Bool = new Type();
  public static final Type String = new Type();
  /** Represents the absence of type. */
  public static final Type NoType = new Type();
  /** Represents the impossibility to assign a type. */
  public static final Type Bad = new Type();

  private Type() {}

  public boolean isExprType() {
   return (this == Int) || (this == Bool);
  }

  /** to String returns a string representation of a type. */
...
}
```


Variables in Expressive

- Defined by first assignment.
- All subsequent assignments must be of the same type.
- All references to a variable will have the type given by the first assignment.
- A variable may not be used without a preceding definition (i.e., assignment).
- Project note:
  - You do not need to implement "the correct behavior" for if- and while-statements.

A Visitor for Type Analysis

- Go through all nodes in the AST. (Remember that is what a visitor does.)
- For each node calculate the type of the node.
- To calculate the type of a node, first visit the children of the node to get their types.
- Use a symbol table to keep track of the types of variables.

A Visitor for Name Analysis

- For a variable assignment (x := Exp):
  - Check whether the name is already defined or not (in the symbol table).
  - If not, add the symbol and its type to the symbol table.
  - If it is defined check the type.
- For a use:
  - Check whether the symbol is in the symbol table or not.
  - If not report an error.
  - If it is defined, check the type.

Analyzer …

```
public class Analyzer implements Visitor,OpCodes{

  final Global global;  // Ref to global for IO.
  Symbol symtab = null; // The symbol table.

  // Constructor
  public Analyzer(Global global) {
    this.global = global;
  }
…
```


… Analyzer …

```
  /** Find a symbol in the symbol table. */
  Symbol lookup(String name) {
    for(Symbol s = symtab; s != null; s = s.next) {
      if (name.equals(s.name)) return s;
    }
    return null; // Did not find the symbol.
  }

  /** Add new symbol to the symbol table. */
  Symbol addSymbol(String name, Type type) {
    return symtab = new Symbol(name, type, symtab);
  }
…
```


… Analyzer …

```
  /** Signal an error if the given type does not
      match the expected type. */
  void expectType(int pos,
                  Type expected,
                  Type found) {

    if (expected != found) {
      global.error(pos, "Type error: expected = "
        + expected +
        "; found = " + found + ".");
    }
  }
…
```


… Analyzer …

```
  /** Analyze and attribute the tree node. */
  public Type analyze(Tree tree) {
    tree.apply(this); // Call the analyzer-visitor.
                      // The analyzer sets the type
    return tree.type; // of the node – return it.
  }

  /** Analyze and attribute an array of trees. */
  public Type analyze(Tree[] trees) {
    // In Expressive we can have a sequence of
    // statements, stored as an array of trees.
    for(int i = 0; i < trees.length; i++)
      analyze(trees[i]); // Analyze each statement.

    return Type.NoType; // A statement has no type.
  }
…
```

… Analyzer …

```
// Visitor methods

public void caseProg(Prog tree) {
  analyze(tree.body); // Analyze the body.

  // A program in itself has no type.
  tree.type = Type.NoType;

}
…
```


… Analyzer …

```
// while Expr:bool { Statements }
public void caseWhile(While tree) {
  // Check the type of Expr
  expectType(tree.cond.pos,
             Type.Bool,
             analyze(tree.cond));

  // Analyze all the Statements in the body.
  analyze(tree.body);

  tree.type = Type.NoType;
}
…
```


(… Analyzer …)

```
// while Expr:bool {Statements} CORRECT SCOPING
public void caseWhile(While tree) {
  Symbol old_symtab=symtab; // Save outer scope

  // Check the type of Expr
  expectType(tree.cond.pos, Type.Bool,
             analyze(tree.cond));

  // Analyze all the Statements in the body.
  analyze(tree.body);

  symtab = old_symtab; // Restore outer scope.

  tree.type = Type.NoType;
}
…
```


… Analyzer…

```
// Identifier
public void caseIdent(Ident tree) {
  // Try to find the symbol.
  Symbol sym = lookup(tree.name);
  if (sym == null) { // Not found.
    global.error(tree.pos,
      "Unknown identifier: "
      + tree.name);
    tree.type = Type.Bad;
  } else { tree.symbol = sym;
           tree.type = sym.type; }
}
…
```


… Analyzer

- All other node types in the AST has to be handled in a similar way:
  - caseIO, caseAssign, caseIf, caseOp, and caseLiteral.
- That is, check the type of any sub-trees and set the result type of the current node.

Summary

- Most programming languages aren't really context free, since names can be defined.
- We use a separate pass, *semantic analysis*, to weed out illegal programs.
- We use a *symbol table* to keep track of the meaning of names and their *scope*.
- In Java the semantic analysis can be implemented with a visitor over the AST.