Part V : Name Analysis

- Programming Languages are not Context Free
- Context Rules for JO
- Representation of Context in a Compiler
- Skeleton Specification of Visibility Rules
- Memory Management
- Optimisation
- Assignment

Programming Languages are not Context Free

- Counter-example: Every identifier needs to be declared
- « Being declared » is a property that depends on *context*.
- In theory, the syntax of a programming language could be specified completely in a context-dependent grammar.
- But in practice, we define a context-free *superset* of the language in EBNF, and then weed out illegal programs with further rules.
- Those rules typically need access to an identifier's declaration.

Context Rules for JO

- JO has the standard *block-structured* visibility rules for identifiers.
- For the purpose of this discussion a *block* is
  - anything enclosed in {} braces, or
  - the area consisting of a functions parameter list up to the end of its body.
- Then we have:
  - Every identifier has a scope, i.e. an area of the program text in which it can be referred to.
  - The scope of an identifier extends from the point of its declaration to the end of the enclosing block.
  - It is illegal to refer to an identifier outside its scope.
  - It is illegal to declare two identifiers with the same name in the same block.
  - However, it is legal to declare an identifier in a nested block which is also declared in an enclosing block.
  - In this case, the inner declaration hides the outer.

Representation of Context in a Compiler

- We represent context by a global data structure, which stores for every visible identifier data about its declaration.
- The data structure is called a *symbol table*, and the information associated with an identifier is called a *symbol table entry* (or entry for short).
- Since J0 has nested blocks, the symbol table should be structured in the same way.
- The symbol table can be represented as a stack of blocks, with the currently innermost block on top:

```
Symbol Table = Stack(Block)
  Block        = List(Entry)
  Entry        = ?
```

Symbols

- A symbol is a data structure which contains all information about a declared identifier which the compiler needs to know.
- Symbols have a name and a type.
- Symbols are grouped together in scopes.
- It is sometimes necessary to step through all symbols of a scope in the sequence they were declared.

  ⇒ Link symbols linearly with a next field.
- This leads to the following class for symbols.

```
class Symbol {
   Symbol next;
   String name;
   Type type;
   // constructor goes here
}
```

Types

- A type is a data structure which contains all information about a value of an expression or a symbol (except its name) which the compiler needs to now.
- Types come in a variety of forms: **`int`, `string`, `void`**, array types **`T[]`**.
- To record information about a function, we also introduce function types. Example:

  `void swap(int[] elems, int i, int j)` has type

  `void(int[] elems, int i, int j)`
- The parameter names **`elems, i, j`** are redundant. They are kept here since this leads to a simpler implementation: Parameters are simply represented by the scope which contains them.

Types (2)

- This leads to the following abstract syntax for types.

```
Type = IntType
     | StringType
     | VoidType
     | ArrayType Type
     | FunType Type Scope
```

A Class for Types (1)

- Applying our transformation from abstract syntax to tree classes systematically yields:

```
class Type {
   static class IntType {}
   static class StringType {}
   static class VoidType {}
   static class ArrayType {
      Type elemType;
      ArrayType(Type elemType) {
        this.elemType = elemType;
      }
   }
   static class FunType {
      Type resType;
      Scope params;
      FunType(Type resType, Scope params) {
          this.resType = resType; this.params = params;
      }
   }
   static Type intType = new IntType;
   static Type stringType = new StringType;
   static Type voidType = new VoidType;
}
```

A Class for Types (2)

- Some classes can be omitted and access canbe optimized by adding a *tag* which tells us the kind of a type.

```
Class Type {
  static final int
    INT = 1, STRING = 2, VOID = 3,
    ARRAY = 4, FUN = 4;
  int tag; // one of the above

  Type(int Tag) { this.tag = tag }

  static class ArrayType {
    Type elemType;
    ArrayType(Type elemType) {
      super(ARRAY);
      this.elemType = elemType;
    }
  }
```

```
  static class FunType {
    Type resType;
    Scope params;
    FunType(Type resType, Scope params) {
      super(FUN);
      this.resType = resType;
      this.params = params;
    }
  }

  static Type intType = new Type(INT);
  static Type stringType =
                        new Type(STRING);
  static Type voidType = new Type(VOID);
}
```

Scopes

- Scopes represent areas of visibility.
- A **`scope`** is a data structure which refers to all identifiers declared in it.
- Scopes are nested; therefore it is convenient to keep an **`outer`** field in a scope which refers to the next enclosing scope.
- This leads to the following class fragment.

A Class for Scopes

```
class Scope {
   Symbol first;
   Scope outer;
   Scope(Scope outer) { this.outer = outer; }
   /** find symbol with given name in this scope.
     * return null if non exists
     */
  Symbol lookup(String name) {...}
  /** enter given symbol in current scope
    */
  void enter(Symbol symbol) {...}
}
```

- Scopes refer to first symbol declared in scope; other symbols are accessed via **`next`** field in class **`Symbol`**.
- Exercise: Write implementations for **`lookup`** and **`enter`**.

How It Hangs Together

- Consider the J0 program

```
module Main {
   void makeArray(int len) { ... }
   void swap (int[] elems, int i, int j) {
      int t;
      t = elems[i]; // [[in red]] ****
      elems[i] = elems[j];
      elems[j] = i;
   }
   ...
}
```

- Then at the point marked ****, the symbol table would look as given on the blackboard.

Memory Management

- Symbol table entries for local variables in blocks that have already been parsed completely are no longer needed.
- How do we get rid of them?
- In Java, the garbage collector will take care of this.
- In C/C++ the most effective strategy is a custom memory allocator that uses mark/release instead of dealloc.
- On block-entry: mark the current heap top
- On block-exit: reset heap top to previous mark.

Optimisation

- The current scheme uses a linear search for identifiers
- In a production compiler this is far too slow.
- Better schemes:
  - Additionally link entries as a binary tree and use that for searching.
  - Use a hash table for each block
  - Use a global hash table (fastest)

Specification of Context Rules

- How are symbol tables used in a compiler?
- Need to ask first: How do we specifiy use of symbol tables in the context rules of a language?
- More generally: How do we specify context rules?
- Several methods are possible.
- We use just a semi-formal method, which adds *attributes* to symbols and connects attributes with *constraints*.

Skeleton specification of visibility rules :

```
P     = ModDecl ident {D}
```
"create a new outermost scope"

```
D     = VD | FD
VD    = VarDecl T(t) name
```
"create a new symbol in current scope with given **name** and type **T**.

```
FD    = FunDecl RT(t) name {VD} S
```
"process parameters **{VD}** in a nested scope; create a new symbol in current scope with given **name** and a function type which refers to parameters and resulttype **T**.

```
T(t)  = int                          t = Type.intType
      | string                       t = Type.stringType
      | T(t1)[]                      t = new Type.ArrayType(t1)
RT(t) = T(t1)                        t = t1
      | void                         t = Type.voidType
E     = Ident(name)                  e = findSymbol(name)
```

Attribute grammars

- Context-dependend syntax is sometimes specified using an *attribute grammar*.
- similar to what we have done, but completely formal.
- Attribute grammars are based on concrete context-free syntax.
- Symbols are given attributes, which can have arbitrary type.
- Attributes are evaluated by assignments similar to our constraints.
- Attributes are represented as instance variables in tree nodes.

Type Systems

- Express context-dependend syntax as a deduction system.
- Judgements are the of the form $\vdash$ <term> : <type>.
- A program P is well-typed iff a judgement $\vdash$ P: T is provable.
- Example: A typing rule for addition:

$$\frac{\vdash A: \text{int} \quad \vdash B: \text{int}}{\vdash A + B: \text{int}}$$

- We usually keep also en environment representing the current symbol table in a judgement.
- Type systems are often more concise and legible than attribute grammars.
- Attribute grammars are closer to an implementation.