CS 301 Spring 2016
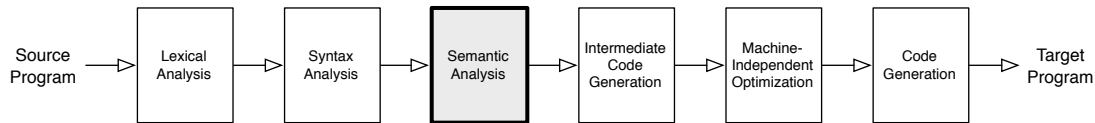**Meetings** February 29

Scopes and Type Systems

Plan

This week, we explore (1) symbol table management for the IC language and (2) type systems as a foundation for describing and reasoning about type checking. The reading focuses mainly on type systems. Review last week's reading as needed for symbol tables and scope.

Readings

- "Type Systems," Luca Cardelli, from *Handbook of Computer Science and Engineering*, 1997. Sections 1–3 (stop after Table 9), Section 9. `http://lucacardelli.name/Papers/TypeSystems.pdf`

  Even though you will not need to use it in any detail, this paper does assume familiarity with the λ-calculus as a basic syntax for a programming language. The Wikipedia entry for lambda calculus or the first few sections of `http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf` should suffice if you have not taken 251.

  Cardelli designed and implemented the first ML compiler and helped to develop much of the formal theory of object-oriented programming. These days, he is working on programming cells and chemicals.

- EC 4.2 if you skipped it last week.

- Dragon, Ch 6.3–6.3.2, 6.5. (Skim)

- IC Specification, Section 16 (IC Type System).

Exercises

1. This question applies the symbol table discussion from last week's problem set to write the design of the `symtab` package for your IC compiler. This package will contain two main classes: a `SymbolTable` class that encodes all symbol information for a scope, and a symbol table building function that will recursively traverse the AST, constructing the symbol tables and annotating each AST node with the symbol table corresponding to its scope, as described in the PA 2 handout.

   Your design need not be long or contain every detail, but it should address the following items:

   - Define the general interface to the `SymbolTable` class. Describe what each method does, and sketch the internal representation details. Include a description of what information is stored with each symbol in the tables (ie, your `SymbolTable` is really a map from identifier to what?)

   - Describe the operations performed by the scope-building function upon entry and exit to each form of scope (*e.g.*, global, class, method, block).

   - How will your building function create a new symbol entry when it encounters a declaration?

   - When, how, and where will the scoping rules be enforced?

   - Sketch the contents of your `SymbolTable` structure after processing the following program.

```
1:    class A {
2:      int x,u;
3:      void f(int y, int z) {
4:        int x;
5:        y = x + this.x;
6:        g(y-u);
7:      }
8:      void g(int y) { }
9:    }
10:
11:    class B extends A {
12:      int a,b,c;
13:      void g(int y) {
14:        if (y > 0) {
15:          int a = 2;
16:          f(y, x);
17:        }
18:      }
19:    }
```

- For each occurrence of CLASSID and ID in the IC grammar, identify whether or not it is a defining occurrence that introduces a new name into the current scope. For those that are not, indicate the scope in which the defining occurrence of that name would occur. (No need to hand this in — it is just for your reference.)

  Use your characterization to indicate how your compiler will resolve the following symbols:

  - the first x on line 5.
  - this on line 5.
  - the second x on line 5.
  - u on line 6.
  - x on line 16.

- In what ways, if any, will you change the AST package to support construction of the symbol tables?

2. Using the type system of Cardelli's paper, prove that

   (a) $\emptyset, x : Nat \vdash succ \ (succ \ x) : Nat$

   (b) $\emptyset, x : Nat \vdash if \ true \ then \ x \ else \ 0 : Nat$

3. For each of the following IC constructs, state whether it is well-typed in some well-formed typing context, according to the type system from the IC Specification. If so, give the most general typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why. (Note that, unlike in the Cardelli system, you do not need to prove that environments and types are well-formed. You should be able to convince yourself that any type or environment that you mention is well-formed, however.)

   (a) `(new int[x.length])[x[2]]`

   (b) `if (x == v[x] && y == "true") x = y;`

   (c) `((a == b) == c) && (a == (b + "c"))`

   (d) `f(x)[x.length] = y[2]`

   (e) `if (x == a[b[x]] && y) y = b[c[x]];`

4. Suppose we extend IC with tuples of the following form. A tuple type is written as a sequence of types in parentheses. For example, the type `(int,bool,string)` represents a 3-tuple. The individual elements of the tuple can be accessed (i.e., read or written) in a manner similar to array elements. For example, if `x` has type `(int,bool,string)`, the expression `x[0]` has type `int`, `x[1]` has type `bool`, and `x[2]` has type `string`. Tuples are unlike arrays in that the index expression must be a constant. For simplicity, we assume that tuples don't contain class types; this ensures that different tuples cannot be subtypes of each other.

   (a) Explain why it is necessary to require that the index of a type expression be a constant.

   (b) Write additional typing rules in the static semantics of IC for expressions and statements to support tuples.

   (c) Consider the types `T1 =(int,(int,int)[])` and `T2 =(int,(int,int))[]`. Consider a variable `x` having either type `T1` or type `T2`. Write an expression that type-checks and has the same type in both cases. Write an expression that type-checks if `x` has type `T1`, but doesn't if `x` has type `T2`. We require that `x`, 0, and 1 are the only variables and constants in your expressions.

   (d) Syntactically, the tuple element access expression looks like an array element access expression. Will this create problems for type checking? Explain briefly.

5. This problem concerns the typing and the translation of `for` loop constructs.

   (a) Suppose we extend IC to support for loops that operate over ranges of integer values:

   ```
   for (x from e1 to e2) s
   ```

   where `x` is an integer field or local variable (that has been declared before), and `e1` , `e2` are the loop bounds. The loop body is executed for all values of `x` in the range from `e1` to `e2`. Write a typing rule that ensures the safe execution of this loop.

   (b) It is difficult to reason about for loops when the execution of the loop body might change the iteration variable or the loop bounds. Describe a semantic check that would ensure this never happens. (You can either write down typing rules to capture your checking or just explain your checking in English.)

   (c) Suppose we further augment IC with extended for loops, in a fashion similar to those in Java 1.5. An extended for loop in IC will have the following form:

   ```
   for (T x : e) s
   ```

   Here, `x` is a newly declared local variable of type `T` that iterates over all of the elements of the array `e`, and `s` is the loop body. Write an appropriate typing rule for this construct.