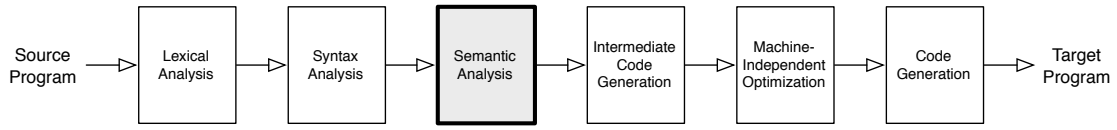
CS 301 Spring 2021 T4
Tutorial Assignment
29 April

Symbol Tables, Types

1 Plan



This tutorial explores symbol table management for name and scope analysis in the Roost compiler and type systems as a foundation for describing and reasoning about type checking.

2 Readings

- Skim EC 5.5. *Do not take the concrete symbol table pseudocode from EC too literally. Think abstractly for yourself about how to implement this.*
- Skim EC 4.2
- *Type Systems.* **Sections 1–3 (stop after Table 9), Section 9.**
  Luca Cardelli. In *Handbook of Computer Science and Engineering*, CRC Press, 1997.
  `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 CS 251.
- Roost *Language Specification*, *Type System* section
  `https://cs.wellesley.edu/~cs301/s21/project/roost-lang.pdf`
- Roost *Compiler Front End* project page, *Building Scopes and Resolving Names* section
  `https://cs.wellesley.edu/~cs301/s21/project/frontend/#scope`

3 Exercises

1. Design a `SymbolTable` data structure for your Roost compiler. The symbol table will be used to encode all symbol (name/identifier) information for a scope, as well as its relation to other scopes. A scope-building function will traverse the AST recursively, constructing symbol tables and annotating each AST node with the symbol table describing its scope, as described in the Front End project document.

   Do not take the concrete symbol table pseudocode from EC too literally. Think abstractly for yourself about how to implement this. You may find it useful to work the next exercise alongside this one and consult the Front End project description for general design hints.

   Your design, in whatever format is easiest to capture your thoughts, should address the following items:

   (a) Define the general interface to the `SymbolTable`. Describe the operations it supports and sketch the internal representation details. Include a description of what information is stored with each symbol in the tables (*i.e.*, your `SymbolTable` is really a map from identifier to...?).

   (b) Describe how the scope-building function will create or manipulate symbol tables as it enters and exits each form of scope (*e.g.*, top-level, function, structure, enumeration, let-block, module).

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

(d) When, where, and how will the scoping rules be enforced?

(e) When, where, and how will names be resolved to their declarations?

(f) What explicit support, if any, must your AST design provide for construction and storage of symbol tables, checking of scoping rules, and resolution of name references? How can Scala traits make this more straightforward?

(g) For each occurrence of Id in the Roost grammar, identify whether it is a definition that introduces a new name into the current scope or a reference that uses a name from the current scope. For references, indicate the scope in which the defining occurrence of the referenced name would occur.

2. Sketch the contents of your `SymbolTable` structure after processing the following Roost code, which uses generic types (similar to Java, Scala, Rust, Swift) from the Roost standard extensions.

```
1  extern fn string_length(s: String) -> i64;
2
3  struct A<T> {
4    x: i64,
5    y: T
6  }
7
8  enum B<U,V> {
9    C,
10   D(A<U>),
11   E(V)
12 }
13
14 fn f<W>(s: A<W>, mut y: i64, x: W) -> W {
15   {
16     y = y + s.x;
17     let x = 1 + y;
18     y = y + x;
19   }
20   if (y < 0) {
21     x
22   } else {
23     s.y
24   }
25 }
26
27 fn g(s: A<i64>, r: B<A<bool>, String>, q: i64) -> i64 {
28   if (q == 0) {
29     let q = A { x: s.y, y: "Roost" };
30     f(q, s.x, "Compiler")
31   } else {
32     match (r) {
33       C => g(s, E("Roost"), s.x),
34       D(x) => x.x + s.y,
35       E(s) => string_length(s),
36     }
37   }
38 }
```

3. Use your symbol table sketch to indicate how your compiler will resolve all function, parameter, variable, field, and type identifiers in the above code.

4. For each of the following ROOST constructs, state whether it is well-typed in some well-formed typing context, according to the type system from the ROOST Language Specification. If the construct is well-typed, give the most general typing context in which the construct is well-typed and write the corresponding typing derivation / proof tree. If the construct is not well-typed in any type context, explain why. (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) `[0; 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]]; }`

   (f) `{f(y, g(x))[1] = g(f(x.length, z)); ()}`

   (g) `if (x[a.length] == a.length) { x } else { a }`

5. Suppose we extend ROOST with tuples of the following form. A tuple type is written as a sequence of types in parentheses. For example, the type `(i64, 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 `(i64, bool, String)`, the expression `x[0]` has type `i64`, `x[1]` has type `bool`, and `x[2]` has type `String`. Tuples are unlike arrays in that the index must be an integer literal. (It cannot be an arbitrary expression.)

   (a) Explain why it is necessary to require that the index of a tuple indexing expression be a integer literal (a constant).

   (b) Write additional ROOST typing rules for support *immutable* tuples. (Parts may be accessed but not assigned.)

   (c) Consider the types `T1 = (i64, [(i64, i64)])` and `T2 = [(i64, (i64, i64))]`. Write an expression using a variable `x` that type-checks regardless of whether `x` has type `T1` or type `T2`. Write an expression that type-checks if `x` has type `T1`, but does not type-check 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.

6. This problem concerns the typing of `for` loop constructs.

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

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

   In this form, `x` is a variable in scope with type `i64`, `e1` and `e2` are arbitrary expressions providing the loop bounds, and `b` is an arbitrary block expression: the body of the loop. The loop body, `b`, is executed once for each value of `x` in the range from the result of evaluating `e1` up to but not including the result of evaluating `e2`. Write a typing rule that ensures safe execution such loops.

   (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 ROOST with extended `for` loops, in a fashion similar to those in Python, Java 1.5+, or Scala. An extended for loop in ROOST will have the following form:

   ```
   for (x: T in e) b
   ```

   In this form, `x` is a newly declared local variable of type `T` that is bound to each element of the array resulting from evaluating the expression `e`, and `b` is the loop body, a block expression. Write an appropriate typing rule for this construct.

7. Section 1 of Cardelli's *Type Systems* as well EC 4.2 discuss nominal (name) and structural type equivalence.

   - What is the difference?
   - Which does Java use? Which does ROOST use? Can you think of examples of both in other statically-typed languages you have used (*e.g.*, ML, Scala, Haskell, C#, Rust, Swift, TypeScript, ...)? In dynamically-typed languages (*e.g.*, Racket, Python, Ruby, JavaScript, ...?) Do any languages mix the two approaches?
   - Both authors describe tradeoffs between nominal and structural typing. Do you agree with them? Which is better? Which issues should you worry about?

8. **Extra Depth:** Is the large ROOST code above well-typed? Use the type system from the ROOST Language Specification to explain informally, but in detail, how type-checking should proceed on this code. There's no need to write it all down, but capture the most subtle or surprising parts and be ready to explain everything.