Assignment #7: Interpreter Phase IV

Due: March 12, 2010 11:59pm

Overview

For this assignment you will extend your interpreter to support static type checking. This change requires modifications to the parser and new phase just before evaluation. Specifically, your interpreter will parse a program, build an abstract syntax tree, validate the correct use of types (this is the new phase), and then evaluate type-correct programs.

The D♭ Language

The following modifications are made to D♭'s syntax. These modifications require the user to specify the type of each set of declarations, the type of each parameter, and the return type for each function. Only the new and modified rules are shown.

declarations	→	{`var` **type** `id` {, `id`}* ; }*
function	→	`fn` **type** `id` (**parameters**) **declarations compound_statement**
parameter	→	**type** `id`
factor	→	(**expression**) | `id` {(**arguments**)}$_{opt}$ | `number` | `true` | `false` | `unit`
		| `fn` **type** (**parameters**) **declarations compound_statement**
type	→	**primitive_type** | **function_type**
function_type	→	({ **type** {∗ **type**}* }$_{opt}$ `->` **type**)
primitive_type	→	`int` | `bool` | `unit`

Part 1: Parsing

Modify the parser to support the extended syntax. This will also require modifications to the abstract syntax. In the interest of time, the required syntax differs a bit from what one might see in a real language.

The most obvious difference is the manner in which function types are specified. The `->` introduces a function type. The types of the arguments precede the `->` and are separated by `*`; the return type follows the `->`.

Part 2: Static Typechecking without Functions

Extend the interpreter to support a static typechecking phase. If this phase fails, due to a type error, then report the error and exit the program. Do not attempt to evaluate type invalid programs.

For this part, you may ignore functions (both named and unnamed). This, of course, implies that you ignore function invocations and the `return` statement as well.

Validate the base expressions and statements. Specifically, the static semantics that must be enforced follow.

- All variables must be declared before use.
- All variables must have unique names.
- Arithmetic operators require integer operands and evaluate to an integer value; relational operators require integer operands and evaluate to a boolean value; boolean operators require boolean operands and evaluate to a boolean value; and the only unary operator requires a boolean operand and evaluates to a boolean value.
- The expression guarding an `if` statement must be of boolean type.
- The expression guarding a `while` statement must be of boolean type.
- The source and target of an assignment statement must have the same type.

For an undeclared variable, a redeclared variable, or a typechecking error, give an appropriate error message, and then stop your interpreter.

Part 3: Static Typechecking with Functions

Extend your typechecker to support named functions and function invocation. Specifically, the following rules need to be checked.

A function declaration is valid if

- the name of the function does not match any global variable or function
- the formals are valid (i.e., no duplicates)
- the local declarations are valid (i.e., no duplicates)
- the statements in the function body are valid under an extended typemap containing the formal parameters and local declarations
- any `return` appearing in the body evaluates an expression of type matching the function `return` type[1]

A function invocation is valid if

- the identifier being invoked was declared as of function type (or as an actual function)
- each of the actual argument expressions is valid
- the number of actual arguments is equal to the number of formal parameters expected for the function
- the types of the actual arguments match the expected types of the formal parameters

Part 4: Static Typechecking with Anonymous Functions

This part actually differs little from the previous part. The reason for dividing the two is that this part will more apparently exercise the function datatype (though your implementation above will likely use the same type).

Extend your typechecker to validate anonymous functions according to the same requirements stated for named functions.

Part 5: Return

You may have noticed that the requirements in part 3 state that a `return` must evaluate an expression of the `return` type of the enclosing function, but it says nothing about requiring a `return` statement.

For this part, your typechecker (which is more generally a static semantics analyzer) must verify that a function without `unit` return type does, in fact, return a value. This means that no path through the function can result in a `return` not being executed.

For this part, and in the interest of time, it is enough to require that a non-unit function end with a valid return statement.

NOTE: Initializing variables to unit, as was required in previous assignments, is no longer valid for the current type system. You should modify the initial values to be zero values (define some such zero value for each function type) or a single invalid value sentinel.

[1] See the last part of the assignment for a stronger requirement.

Notes

- Test data with "correct" output will also be given. Your output can differ in minor ways (e.g., syntax error message format) from this "correct" output yet still be correct; it is up to you to verify your code's correctness.
- Your program will be exercised by some shell scripts, which will be provided. These scripts depend on the exit status of your program. If your program detects a "serious" error, your program should use "OS.Process.exit" to terminate.
- Grading will be divided as follows.

Part	Percentage
1	20
2	40
3	25
4	10
5	5

- Get started **now** to avoid the last minute rush.