Assignment #4: Interpreter Phase I (Evaluation w/o Functions)

Due: February 17, 2010 11:59pm

Overview

For this assignment, you will add an additional phase to your interpreter (the first two phases being the lexer and the parser). Specifically, you must extend your interpreter to evaluate syntactically valid programs. For this assignment, however, your interpreter will *not* support functions.

Educational Goals

- Dynamic Semantics (Evaluation)
- Dynamic Type Checking

Semantics Overview

D♭'s semantics are, for the most part, similar to those of C or Pascal, and have been illustrated formally in lecture.

The primary difference from the material covered in lecture relates to the output statements. The `write` statement prints the result of the evaluation of the expression followed by a *single space*. The `writeline` statement prints the result of the evaluation of the expression followed by a newline (no trailing space). Integer values are printed as the corresponding integer (negative integers must be output with a '-' and not a '~'). Boolean values are printed as either `true` or `false`. The `unit` value is printed as `unit`.

Program Entry Point

For this assignment you are adding only a single phase to your interpreter. Later assignments will add additional functionality. As such, we will now standardize on a single entry point to the entire interpreter. You must provide a function named `interpret` that takes a file name as a string. This function then invokes the appropriate phases of the interpreter (parsing and then evaluation for this assignment).

Part 1: Evaluation (Dynamic Semantics) without Functions

Modify your interpreter to evaluate syntactically valid programs through meaning functions. Your solution to this part should be able to evaluate any valid D♭ program that does not contain functions, function invocations, or return statements.

Evaluation of D♭ programs will require a representation of the state of the program associating variables with their values. The state can be represented as a flat list, but a hash table will typically improve performance. You may modify the hash table presented in the text or use the HashTable module in the sml implementation.

For the purposes of this assignment, all variables are initialized to the `unit` value.

Part 2: Dynamic Type Checking

Though not strictly a separate phase this is listed as a separate part for testing purposes. You must modify your interpreter to support dynamic type checking. Specifically, your interpreter must report type errors as they occur during execution (exit the program with an error message). The error messages must indicate the type of error discovered; it is not sufficient to simply indicate that *an error* has been discovered, but rather you must indicate the type of error.

The constraints that must be enforced are

- A variable must have been declared before it can be used.
- 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.

Because the typechecking is being done dynamically, the type of a variable may change as different values are assigned to it.

Notes

- Strive for simplicity in your programming.
- Test data with "correct" output will be given. Your output can differ in minor ways (e.g., the type error messages) from this "correct" output yet still be correct; it is up to you to verify your code's correctness. The output from the evaluation of $D^\flat$ programs (generated by `print` statements) **must** match the corresponding "correct" output exactly.
- 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 OS.Process.failure".
- You must follow the steps described above in developing your program. Grading will be divided as follows.

Part	Percentage
1	80
2	20

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