Assignment 2, Week 3: Command Interpreter

C S 429, Spring 2022
Unique IDs: 51215, 51220, 51225, 51230, 51235, 51240
Lead TA: Zach Leeper

Assigned: Saturday, 12 February 2022 12:00 CT
Due: Friday, 18 February 2022 23:59 CT
Last possible hand in: Sunday, 20 February 2022 23:59 CT

1 Introduction

In this third and final week of the assignment, you will be extending your *interpreter* for EEL to the language level EEL-2. This level adds the notion of state to the language in the form of named variables, which will introduce some extra complexities in both type inference and evaluation. You will keep track of such variables using a symbol table, which you will implement using a hash table data structure. Finally, you will need to make sure that your interpreter is not leaking memory. Needless to say, your extensions to the interpreter must not change its behavior on EEL-0 and EEL-1 input.

2 Logistics

You may use up to two late (slip) days for this assignment.

Start early enough to complete your work before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped internet connections, corrupted files, traffic delays, minor health problems, etc.

Unless otherwise stated, every project in C S 429 is an individual project. Before you begin, please take the time to review and abide by the course policy on academic integrity at: `https://www.cs.utexas.edu/academics/conduct`.

All hand-ins are electronic. You can do this assignment on any machine you choose. The testing for your code is automated, using OS and compiler configurations similar to those on the UTCS Linux machines. *Test your code thoroughly on a UTCS Linux machine before submitting it.*

Finally, this is a relatively new assignment, so please be patient as we uncover and fix bugs. Keep an eye on Piazza for updates.

3 Download and setup

There are no download and setup steps for this week's assignment. Continue working from where you left off last week.

If you were not able to finish last week's assignment, a reference solution will be released Monday morning, after the last late day has passed. You are in no way required to use it. You should be able to pull this

from GitHub using the provided update script by using `./update.sh` in the terminal. This should create the file `eval-ref-week2.c` in your repository, and if you want to use it, copy it to your `eval.c` file with the command `cp eval-ref-week2.c eval.c`. Please note that **this will replace all your code from last week**, but also note that using this reference does **not** count against you. Finally, if you get a permission denied error when trying to use the `update.sh` script, you can just use `chmod +x update.sh` to fix it.

4 Details of the assignment

Review the full details of the assignment from last week's handout if needed. From here on, the assignment will primarily discuss the tasks you need to complete this week. Once again, the lexical and syntactic analyses are done for you, so your starting point is the abstract syntax tree (AST).

4.1 Handling variables

You will maintain the names, types, and values of variables (in other words, attributes of objects—similar to what we have discussed in class) in a *symbol table*, which you will implement as a hash table. Keep in mind that we don't indicate the type of a variable explicitly in EEL, so you need to infer this type based on the last value it has been assigned. Given the dynamic nature of the language, it is perfectly acceptable for the same variable to be assigned an integer value initially and a string value afterwards.

4.2 Inferring the types of AST nodes

You will extend the type inference phase to handle variables. The overall structure of this will be a recursive postorder traversal of the AST, and you still need to enforce a static typing discipline.

You will know the type of a leaf node based on its value (for constants) or through the symbol table (for named variables). The type inference logic for internal nodes remains unchanged. Figure out what type inference should do for assignment nodes.

4.3 Computing the value of AST nodes

There should be no changes needed to your `eval_node()` method. Remember that the value of a named variable must be retrieved from the symbol table. "Evaluating" an assignment node will require you to update the value field of the symbol table entry for the left child (which must be a named variable) with the value of the right child; this operation is actually handled for you in the provided code, so everything should just work as long as you have implemented the hash table methods correctly.

4.4 Memory management

Once you have finished evaluating the AST, you will need to clean up any memory allocated for it. Similar to type inference and evaluation, you will need to (recursively) perform a post-order traversal of your AST, freeing each node as you go. Additionally, if a node contains a string, you should have allocated space for it during evaluation, so you will need to free it as well. Don't worry about freeing entries in the variable table, since they are supposed to be maintained between evaluation of multiple expressions. As long as your program runs without error, you can locally test for proper memory management with the Linux utility `valgrind`: `valgrind --leak-check=full ./ci  < tests/test_week3.txt`.

4.5 Error handling

Not all user inputs are valid, and you will need to handle errors. Follow the same error handling discipline that you did in weeks 1 and 2 of the assignment.

4.6 What you will code

The only files that you will be modifying this week are `eval.c` and `variable.c`. Search for the string `(STUDENT TODO)` in these files. These are the routines that you need to write, and the comments above the routines tell you what you have to do. For this week, complete the necessary work for the EEL-2 language *without changing the correct behavior for the EEL-0 and EEL-1 language levels*. You should start by testing the examples given in `tests/test_week3.txt`, which you can do automatically with the command `make test_week3`.

One more helper routine that is useful for debugging is `print_tree()`. It allows you to see a simple visualization of the AST. TAs will cover the details of this routine in the Friday sections.

In order to build an executable that you can run, type the command `make ci` at the Linux prompt. This will run the C compiler `gcc` on the necessary files and put everything together into a binary file called `ci`. We have also provided you a pre-built reference implementation of the interpreter, which is in the binary file `ci_reference`. To run either of these binaries, simply type its name at the Linux prompt, and you will be running in interactive mode. If you want the interpreter to read input from a text file called *`foo`*, you can do that by specifying the command-line parameter `-i` *`foo`*. If you want the interpreter to write output to a text file called *`bar`*, you can do that by specifying the command-line parameter `-o` *`bar`*. You can use the `driver.sh` shell script to check the output of your implementation against the reference one.

The `tests` subdirectory contains test files. If you want to write your own test files, look at the examples in here. The format is quite simple: one expression per line, with the last line being the string `@q`.

5 Handing in your assignment

You will hand in this assignment using Gradescope. For code, link your GitHub account to Gradescope, and select the `CI-Lab` repository when submitting the assignment. Make sure all files are up to date (i.e., committed) on GitHub before you hand in the assignment.

6 Evaluation

This assignment is worth 4% of the total course grade.

Your code must compile without error. The correctness of your solution will be determined by test cases covering the EEL-0, EEL-1, and EEL-2 language levels. These cases will include both good and bad inputs. Some tests will be provided to you, while others will be blind tests that you can run only on Gradescope. You will also need to ensure your code has no memory leaks to receive full credit.

A The EEL Grammar

This is the complete grammar for EEL-0, EEL-1, and EEL-2, repeated from last week. The pieces most relevant for this week are boldfaced.

Id	Production	EEL level(s)	AST Fragment	Comments
P1	*Root* → *Expr* `NL`	0, 1, 2	*Root* | *Expr*	`NL` is newline (`'\n'`).
P2	*Root* → *Expr* `#` *Format* `NL`	0, 1, 2	*Root* *Expr* *Format*	`NL` is newline (`'\n'`).
P3	*Root* → *Var* `=` *Expr* `NL`	2	*Root* | = *Var* *Expr*	An initialized variable.
P4	*Expr* → *Literal*	0, 1, 2	*Literal*	
P5	*Expr* → `(` *Expr* `?` *Expr* `:` *Expr* `)`	0, 1, 2	?: *Expr* *Expr* *Expr*	Ternary choice operator.
P6	*Expr* → `(` *Expr Binop Expr* `)`	0, 1, 2	*Binop* *Expr* *Expr*	Binary operations.
P7	*Expr* → `(` *Unop Expr* `)`	0, 1, 2	*Unop* | *Expr*	Unary operations.
P8	*Expr* → `(` *Expr* `)`	0, 1, 2	*Expr*	Extra parentheses.
P9	*Expr* → *Var*	2	*Var*	A named variable.
P10	*Binop* → `[+-*/%&|<>~]`	0, 1, 2	x, $x \in$ `[+-*/%&|<>~]`	Valid binary operations.
P11	*Unop* → `[_!]`	0, 1, 2	x, $x \in$ `[_!]`	Valid unary operations.
P12	*Literal* → *Num*	0, 1, 2	*Num*	Integral constant values.
P13	*Literal* → `true`	0, 1, 2	true	Boolean constant `true`.
P14	*Literal* → `false`	0, 1, 2	false	Boolean constant `false`.
P15	*Literal* → *String*	1, 2	*String*	A string contant.
P16	*Var* → *Identifier*	2	*Identifier*	
P17	*Num* → `[0-9]+`	0, 1, 2	x	x is the numerical value.
P18	*String* → `"` `[' _'-'~']*` `"`	1, 2	x	x is the string value.
P19	*Identifier* → `[a-zA-Z]([a-zA-Z][0-9])*`	2	x	x is the id name.
P20	*Format* → `[dxXbB]`	0, 1, 2	x	x is the format specifier.

- In production P5, evaluation proceeds as in C or Java: the condition is first evaluated, and only one choice is evaluated depending on the value of the condition.
- In productions P6 and P10, the interpretation of the binary operation depends on the types of the input operands.
  - For integers, `+-*/%` are sum, difference, product, quotient, and remainder; and `<>~` are the comparisons less-than, greater-than, and equal-to.
  - For Booleans, `&` is AND and `|` is inclusive-OR.

  - For strings, + is concatenation; * is repetition (with the second argument being an integer); and <>~ compare the lexicographic ordering of two strings.
  - No other combination is legal.

- In productions P7 and P11, the interpretation of the unary operation depends on the type of the input operand.
  - For integers, _ is negation.
  - For Booleans, ! is NOT.
  - For strings, _ is string reversal.
  - No other combination is legal.

- Productions P17–P20 use standard `grep` syntax. ␣ is the whitespace character.

- In production P18, the characters making up the string can be any printable characters. In the seven-bit ASCII character set, these are `0x20` (`'␣'`) through `0x7E` (`'~'`). The function `isprint()` in `<ctype.h>` tests for this property.

- The default format for printing integers and Booleans is `d`, which prints them as decimals. The format specifiers `x` and `X` cause integers to be printed in hexadecimal. while `b` and `B` causes Booleans to be printed as named values. Strings are always printed as character strings.