Assignment 2, Week 2: Command Interpreter

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

Assigned: Saturday, 05 February 2022 12:00 CT
Due: Friday, 11 February 2022 23:59 CT
Last possible hand in: Sunday, 13 February 2022 23:59 CT

1 Introduction

In this second week of the assignment, you will be extending your *interpreter* for EEL to the language level EEL-1. This level adds a new string type to language level EEL-0, along with unary and binary operations on this new type. You will need to handle the operator overloading that this change introduces, resolve the interactions arising between this type and the existing integer and Boolean types, and implement a helper function to implement one of the string operations. Needless to say, your extensions to the interpreter must not change its behavior on EEL-0 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 Tuesday 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` in the terminal. This should create the file `eval-ref-week1.c` in your repository, and if you want to use it, copy it to your `eval.c` file with the command `cp eval-ref-week1.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.

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 Inferring the types of AST nodes

You will extend the type inference phase to handle the string type and the operators it introduces. 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 (an integer constant, the Boolean values `true` and `false`, or a string literal). For an internal node, you first (recursively) infer the types of its children; then you figure out whether the operation represented by that node is compatible with the types of its arguments; if it is, you infer the type of that node. For example, the $\boxed{+}$ operation is defined only on pairs of integers and on pairs of strings, so both of its children must be of one of these types, and the output is also the corresponding type. Any other combination of input types signifies an error. Keep in mind operations like $\boxed{<}$, where the output type is different from the input types; and the ternary operation $\boxed{\texttt{?:}}$, for which not all inputs have the same type.

4.2 Computing the value of AST nodes

At this point of the process, the input characters have been converted into an AST and you have validated the wellformedness of the AST with respect to type. You are now ready to (recursively) associate values with each subtree of the AST. This is again done with a (mostly) postorder traversal of the AST[1], except that you will be computing values at each internal node rather than inferring types.

4.3 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 week 1 of the assignment.

4.4 What you will code

The only file that you will be modifying this week is `eval.c`. Search for the string `(STUDENT TODO)` in the file `eval.c`. 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-1 language *without changing the correct behavior for the EEL-0 language*. You should start by testing the examples given in `tests/test_week2.txt`, which you can do automatically with the command `make test_week2`.

[1]The one exception is the ternary choice operator.

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 and EEL-1 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.

A Strings in C

Strings in C are very different from the objects of the `java.lang.String` class that you are familiar with. In C, a string is nothing more than an array of `char`s terminated by the `char` literal `'\0'` (aka “the NULL character”; hence the technical term “null-terminated byte strings”). Very importantly, you find the length of a C string variable `s` by calling `strlen(s)`; there is nothing corresponding to the `s.len` syntax of Java. Also, think through the difference between `strlen(s)` and `sizeof(s)`.

The C standard library does provide a number of useful operations on strings. The system header file `<string.h>` contains their interfaces. See Section B3 of K&R2e or this online reference for the details. Get familiar with these functions, especially the ones whose names begin with `str`. Read their `man` pages carefully. Their behavior with respect to the null terminator is not always consistent or intuitive.

You will realize that there is no standard function to reverse a string. Fill in your implementation of this functionality in the skeletal code marked `strrev` in `eval.c`.

You will also need to allocate and deallocate memory for strings. The two functions you will need to do this are `malloc` (for allocation) and `free` (for deallocation). The system header file `<stdlib.h>` contains their interfaces. Again, read their `man` pages carefully. Also, be aware that while `strlen("hello")` returns 5, the string actually occupies six bytes in memory (five for the characters in the string, and one more for the `'\0'` that terminates it).

B 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	$\mathit{Root} \rightarrow \mathit{Expr}\ \boxed{\texttt{NL}}$	0, 1, 2	$\begin{tikzcd}[row sep=small] \mathit{Root} \arrow[d, dash] \\ \mathit{Expr} \end{tikzcd}$	$\boxed{\texttt{NL}}$ is newline (`'\n'`).
P2	$\mathit{Root} \rightarrow \mathit{Expr}\ \boxed{\texttt{\#}}\ \mathit{Format}\ \boxed{\texttt{NL}}$	0, 1, 2	$\begin{tikzcd}[row sep=small, column sep=tiny] & \mathit{Root} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathit{Expr} & & \mathit{Format} \end{tikzcd}$	$\boxed{\texttt{NL}}$ is newline (`'\n'`).
P3	$\mathit{Root} \rightarrow \mathit{Var}\ \boxed{\texttt{=}}\ \mathit{Expr}\ \boxed{\texttt{NL}}$	2	$\begin{tikzcd}[row sep=small, column sep=tiny] & \mathit{Root} \arrow[d, dash] & \\ & = \arrow[dl, dash] \arrow[dr, dash] & \\ \mathit{Var} & & \mathit{Expr} \end{tikzcd}$	An initialized variable.
P4	$\mathit{Expr} \rightarrow \mathit{Literal}$	0, 1, 2	$\mathit{Literal}$	
P5	$\mathit{Expr} \rightarrow \boxed{\texttt{(}}\ \mathit{Expr}\ \boxed{\texttt{?}}\ \mathit{Expr}\ \boxed{\texttt{:}}\ \mathit{Expr}\ \boxed{\texttt{)}}$	0, 1, 2	$\begin{tikzcd}[row sep=small, column sep=tiny] & ?: \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \mathit{Expr} & \mathit{Expr} & \mathit{Expr} \end{tikzcd}$	Ternary choice operator.
P6	$\mathit{Expr} \rightarrow \boxed{\texttt{(}}\ \mathit{Expr}\ \mathit{Binop}\ \mathit{Expr}\ \boxed{\texttt{)}}$	0, 1, 2	$\begin{tikzcd}[row sep=small, column sep=tiny] & \mathit{Binop} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathit{Expr} & & \mathit{Expr} \end{tikzcd}$	Binary operations.
P7	$\mathit{Expr} \rightarrow \boxed{\texttt{(}}\ \mathit{Unop}\ \mathit{Expr}\ \boxed{\texttt{)}}$	0, 1, 2	$\begin{tikzcd}[row sep=small] \mathit{Unop} \arrow[d, dash] \\ \mathit{Expr} \end{tikzcd}$	Unary operations.
P8	$\mathit{Expr} \rightarrow \boxed{\texttt{(}}\ \mathit{Expr}\ \boxed{\texttt{)}}$	0, 1, 2	Expr	Extra parentheses.
P9	$\mathit{Expr} \rightarrow \mathit{Var}$	2	Var	A named variable.
P10	$\mathit{Binop} \rightarrow$ `[+-*/%&|<>~]`	0, 1, 2	$\boxed{x}$, $x \in$ `[+-*/%&|<>~]`	Valid binary operations.
P11	$\mathit{Unop} \rightarrow$ `[_!]`	0, 1, 2	$\boxed{x}$, $x \in$ `[_!]`	Valid unary operations.
P12	$\mathit{Literal} \rightarrow \mathit{Num}$	0, 1, 2	Num	Integral constant values.
P13	$\mathit{Literal} \rightarrow \boxed{\texttt{true}}$	0, 1, 2	true	Boolean constant `true`.
P14	$\mathit{Literal} \rightarrow \boxed{\texttt{false}}$	0, 1, 2	false	Boolean constant `false`.
P15	$\mathit{Literal} \rightarrow \mathit{String}$	1, 2	String	A string contant.
P16	$\mathit{Var} \rightarrow \mathit{Identifier}$	2	$\mathit{Identifier}$	
P17	$\mathit{Num} \rightarrow$ `[0-9]+`	0, 1, 2	$\boxed{x}$	x is the numerical value.
P18	$\mathit{String} \rightarrow \boxed{\texttt{"}}$ `['␣'-'~']*` $\boxed{\texttt{"}}$	1, 2	$\boxed{x}$	x is the string value.
P19	$\mathit{Identifier} \rightarrow$ `[a-zA-Z]([a-zA-Z][0-9])*`	2	$\boxed{x}$	x is the id name.
P20	$\mathit{Format} \rightarrow$ `[dxXbB]`	0, 1, 2	$\boxed{x}$	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.**