Assignment 2, Week 1: Command Interpreter

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

Assigned: Saturday, 29 January 2022 12:00 CT
Due: Friday, 04 February 2022 23:59 CT
Last possible hand in: Sunday, 06 February 2022 23:59 CT

1 Introduction

Over the three weeks of this assignment, you will be writing an *interpreter* for EEL [1], a "little language" of expressions. A calculator program is an interpreter. A Linux shell [2] is an example of a *command interpreter* that you have used to execute Linux commands. You may also have used interpreters for programming languages such as Python or Matlab.

The focus of this lab is to give you practice in the style of C programming you will need to be able to do proficiently, especially for the later assignments in the class. Some of the skills developed are:

- Explicit memory management using the `malloc` and `free` calls, which is very different from what you have used in Java.
- Creating and manipulating pointer-based data structures: trees, linked lists, and hash tables. While pointers in C are similar to references in Java, there are key differences between them that you need to internalize.
- Working with strings. There is no `String` class in C, as you will find.
- Implementing robust code that operates correctly with invalid arguments, such as `NULL` pointers.

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`.

[1] EEL stands for "Expression Evaluation Language".

[2] There is nothing particular special about a shell. There are multiple shells available in Linux, and you can choose which to use. You can even write your own shell.

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

First, go to our GitHub Classroom page and create a copy of the assignment: `https://classroom.github.com/a/MFCb03MB`. This will automatically create an assignment repository specific to you on GitHub. Now clone this created repository to `~/cs429/projects/proj2` on a UTCS lab machine using the command `git clone` *`your-repo-url`*. Ask a TA for help if you are having trouble with these steps.

4 Details of the assignment

At their core, all interpreters go through a common three-phase cycle of activities, called the Read-Eval-Print Loop (REPL for short). They repeatedly *read* some input (typically interactive, but possibly from a file or from input redirection), *evaluate* that input (do something with it), and *print* the result of that evaluation (either an actual value or a message produced as a side-effect of evaluation). In the Linux shell example, the input is the command that you type in at the prompt (let's say it's `ls`). The shell program parses this input, validates it as a known command, and proceeds to execute it. [3] The results of executing the command (in this case, the desired listing of files in the current directory) is printed to the console, and the shell resumes waiting at its prompt for the next command.

Your task over the course of this assignment will be to write a similar (but much simpler) interpreter for EEL in stages. We will provide most of the code for reading and printing, so your focus will be on the evaluation part of REPL.

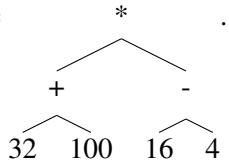
- **Week 1:** Your first interpreter will handle a core language called EEL-0 with only two data types (integers and Booleans), literals (i.e., constants, not variables) and a handful of unary, binary, and ternary operations.
- **Week 2:** You will expand the language to EEL-1 by adding a third type (strings) to the language, along with a handful of (overloaded) unary and binary operations.
- **Week 3:** You will add named variables to create the language EEL-2.

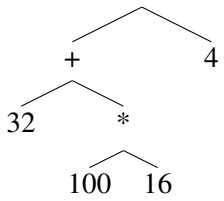
From here on, the assignment will primarily discuss the tasks you need to complete in week 1.

4.1 Background

Inside the EEL interpreter, you will represent the expression that the user types in at the terminal as a tree. For example, if you typed in the character string I_1 = `((32+100)*(16-4))` to the prompt, the internal

[3] The actual method of execution is hairy. You will learn about it in C S 439.

representation would look like this: T_1 = 

You probably realize by now that what we are doing is basically the PEMDAS (or BODMAS) rule that you learned in grade school. However, we have made things easier for you by engineering the EEL grammar to require that all expressions be fully parenthesized, which removes any issues of operator precedence. Thus, the input string I_1' = `32+100*16-4` would not be parsed as being a valid EEL expression, while the string I_1'' = `((32+(100*16))-4)` would have the internal representation T_1'' = 

We transform the input character string representation (the "concrete syntax") into the internal tree representation (the "abstract syntax tree" or AST) in two steps. First, we take the input line and break it into atomic chunks of the language ("tokens"). Then, we process this stream of tokens and convert it into the tree. In the example above, the intermediate token stream generated from input string I_1 would be

S_1 = [`(` `(` `32` `+` `100` `)` `*` `(` `16` `-` `4` `)` `)`],

with each token enclosed in a box and the entire stream delimited by square brackets.

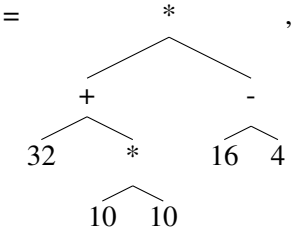
As another example, consider the input line

I_2 = `((32+(10*10))*(16-4))`,

which turns into the token stream

S_2 = [`(` `(` `32` `+` `(` `10` `*` `10` `)` `)` `*` `(` `16` `-` `4` `)` `)`]

and finally into the AST

T_2 = 

whose *structure* is recognizably different from that of T_1. [4] The difference between the AST structures is in the right child of the + node: a single leaf node for T_1, a subtree for T_2.

4.1.1 Converting a line of text into tokens

The process of going from the input character string to the sequence of tokens is called *lexical analysis* (aka "lexing" or "tokenization"). This process involves some details that would be a distraction right now, so we are providing the code for this module in the file `lex.c`.

Whitespace is largely insignificant in EEL.[5] So one function of the lexer is to strip out ornamental whitespace (spaces or tabs) as it produces the token stream. This means that the input string I_1 and the

[4] Both T_1 and T_2 evaluate to the same value. That is an orthogonal matter.

[5] Not always, however. Consider the difference between inputs `32` and `3 2`. The first input produces the token stream [`32`], while the second produces the (ungrammatical) token stream [`3` `2`].

input string `(␣(␣32␣+␣(␣10␣*␣10␣)␣)␣*␣(␣16␣-␣4␣)␣)` (with additional whitespace characters, indicated as ␣) would both result in the same token stream S_2.

4.1.2 Building an AST from a stream of tokens

There are two key differences between the representations S_2 and T_2. First, S_2 is linear, while T_2 is non-linear. Second, T_2 is missing a number of tokens—specifically, the tokens `(` and `)`. The purpose of these parentheses was to linearize the nonlinear tree structure while retaining the relationships between different parts of the expression. Once we build the tree representation, we can safely discard these tokens. This is why the tree is called an *abstract* syntax tree.

The structure of all possible ASTs that are syntactically valid EEL expressions is specified by a *grammar*, much like the grammar for a natural language. A grammar [6] is simply an inductive way of defining infinite sets that have a natural tree structure. The complete EEL grammar is in Appendix A.

Once again, this process involves some details that would be a distraction right now, so we are providing the code for this module in the file `parse.c`.

4.1.3 Inferring the types of AST nodes

Unlike Java or C, we don't explicitly specify the types of the variables or constants in EEL. And if you look at the definition of *Literal* in the EEL grammar, you will see that in EEL-0 it encompasses integer constants and Boolean constants. This means that it is possible to create an AST for something like `2+true`, which is meaningless because you can't add an integer to a Boolean. This is like an English sentence "The ball threw the boy", which is grammatical (syntactically correct) but nonsensical (semantically incorrect). So, after creating the AST but before evaluating it, you need to validate that it is well-formed. This process is called *type inference*, and you will do this by a postorder traversal of the AST.

You will know the type of a leaf node based on its value (an integer constant or the Boolean values `true` and `false`). 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 `+` operation is defined only on pairs of integers, so both of its children must be of type integer, and the output is also the corresponding type. Any other combination of input types signifies an error. Keep in mind operations like `<`, where the output type is different from the input types; and the ternary operation `?:`, for which not all inputs have the same type.

While EEL is an implicitly-typed language, we also want it to be statically typed. By this we mean that an expression like `((20 < 15) ? 3 : true)` must fail during type inference rather than during evaluation.

4.1.4 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 postorder traversal of the AST, except that you will be computing values at each internal node rather than inferring types.

[6] Technically, this is called a context-free grammar. There are other types of grammars.

4.1.5 Error handling

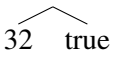
Not all user inputs are valid, and you will need to handle errors. Given the multi-step nature of the expression evaluation process, errors can happen in multiple places. The principle of error handling you will use is this: *Handle the error as early as you can in the process, but no earlier.* Typically, these failures means that none of the following stages can be executed meaningfully, so you will print an error message and return an appropriate indicator to your caller routine.

We classify errors by the earliest time that we can recognize them. Here are the classes.

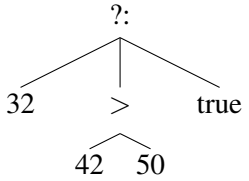
1. *Lexical errors* are those that cause lexical analysis to fail to create a valid token, e.g., an unrecognized character or an invalid syntax for an identifier. The supplied lexer module handles such errors for you.

2. *Syntactic errors* are those that cause the parser fail to create a valid AST. For instance, the token stream `1` `+` `2` is not a valid production (it needs to be fully parenthesized); neither is `)` `1` `+` `2` `(` (the parentheses occur in the wrong order) or `(` `1` `+` `2` (the parentheses aren't matched). The supplied parser module will handle these errors.

3. *Type inference errors* are those that fail to infer valid types for the AST. [7] Examples include



and



4. *Evaluation errors* are those that fail to produce a meaningful value for a subtree. The only cases where this happens this week are for the integer operations of division and remainder, where a divisor value of zero causes problems.

4.2 What you will code

The only two files that you will be modifying in this assignment are `interface.c` and `eval.c`.

Before you do anything else, update line 15 of the file `interface.c` with your name and UT EID as specified. Save the file; you are done with changes to this file.

Next, study the five files `err_handler.h`, `node.h`, `token.h`, `type.h`, and `value.h` carefully. These provide the documentation of the data structures you will be using. It may also be helpful to study the files `err_handler.c` and `lex.c` to understand the interface for handling errors.

Now you are ready to start writing your own code. 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-0 language. You should start by testing the examples given in `tests/test_week1.txt`, which you can do automatically with the command `make test_week1`.

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.

[7] Since there weren't any syntactic errors, you have a valid AST.

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 [8] 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 language level. 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 on Gradescope.

[8] A shell script is simply a program written in the "little language" that the shell interprets. It is a text file. Feel free to study it if you are curious.

A The EEL Grammar

This is the complete grammar for EEL-0, EEL-1, and EEL-2.

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.