Problem Set 1: Course Introduction, LiveOak, SaM

C S 395T

25 August 2021

This is the problem set for week 1 of C S 395T SIMPL. It is intended to help you learn the material by working out more examples and exercises than is possible to cover in the videos. Feel free to work individually or in groups. Ask questions on Piazza. You are not required to submit anything, and the problem set doesn't count directly towards your course grade.

The solution key for the problem set will be made available one week after its release.

1. *Identifiers in LiveOak.*
   Which of the following identifiers are valid at each language level (0, 1, 2, 3) of LiveOak? For the identifiers that are invalid, explain why.

   (a) `this_variable`
   (b) `Break`
   (c) `Bool1`
   (d) `while`
   (e) `check_123`
   (f) `123_check`
   (g) `_check_123`
   (h) `string`
   (i) `IsThisValid?IDK`
   (j) `x`

2. *Tradeoff between ease of use and ease of implementation.*
   As explained in the video "The Source Language: LiveOak", the expression sublanguage is fully parenthesized to remove any issues of precedence and associativity.

   (a) Suppose we wanted to make the language more user-friendly so that binary operators follow the conventional rules for order of operations (which you may remember as the mnemonic PEMDAS, BEDMAS, BODMAS, or BIDMAS, depending on where you are from) and unary operators have higher precedence than binary operators. Rewrite the grammar for the expression sublanguage with this change.
   (b) All of the binary operators in LiveOak are left-associative. Add a right-associative exponentiation operator `^` and rewrite the resulting expression grammar to honor both precedence and associativity. To simplify this part, restrict your operators to `+*^~` on `int` literals and variables.

Hint: Look at the definitions of the expression sublanguages of C or Java. They are, of course, much more elaborate, but you will get the general idea. The Wikipedia page referenced in the problem statement is also a good source of information.

3. *Numeric literals.*
   This problem deals with the numeric literals representable in LiveOak. These are described by the production for the non-terminal `NUM`.

   (a) Which integers are not representable as numeric literals? How would you create such an integer value in a LiveOak program?

   (b) Which representable integers do not have a unique representation (i.e., there are multiple character strings representing the same integer value)? Is this a problem?

   (c) Modify the grammar for the sublanguage of numeric literals so that every representable integer has a unique representation.

   (d) Suppose we wanted to add octal (base-8) and hexadecimal (base-16) integer literals to the existing language. A sequence of digits is taken to be octal if it begins with `0` (digit zero), decimal otherwise. Octal literals do not contain the digits `8` or `9`. A sequence of digits preceded by `0x` or `0X` is taken to be a hexadecimal literal. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. Augment the grammar for the sublanguage of numeric literals to incorporate these features.

   (e) This is the word definition of a floating-point literal: "A floating-point literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing." Convert this word definition to a formal definition.

4. *ISA taxonomy.*
   Compare 0-, 1-, 2-, and 3-address machines by writing assembly-language program fragments to compute `X = (A + B * C) / (D - E * F - G * H)` for each of the four types of instruction sets.

   - All of the architectural variants have instruction for addition, subtraction, multiplication, and division.
   - The 0-address machine has push and pop instructions to move operands between memory and stack. The 1-address machine has load and store instructions to move operands between memory and accumulator. The 2- and 3-address machines have a move instruction to move operands between memory and registers or between two registers.
   - All variables are initially in memory, and the result must be placed in memory.
   - There are a sufficient number of registers for the 2- and 3-address machines.

5. *Writing SaM code by hand.*
   Write SaM code by hand to perform the following tasks. In all cases, the TOS element is the last one in the list. Use the SaM architecture manual as your reference for SaM bytecodes.

   (a) Given an initial stack containing two integers x and y, return a final stack containing $x + y$ snd $x - y$.

(b) Given an initial stack containing two integers x and y, return a final stack containing $5x + 3y$ *without using integer multiplication*.

(c) Given an initial stack containing a 32-bit integer x, return a final stack containing the number of 1-bits in the binary representation of x. The code should not have any branches in it. Try to minimize the number of instructions.

(d) Given an initial stack containing two positive integers x and y, return a final stack containing the greatest common divisor of x and y. Use Euclid's algorithm in either a recursive or iterative form.

(e) Given an initial stack containing a pointer to a heap-allocated character string, reverse that character string in-place.

If you are feeling particularly adventurous, repeat the exercise for JVM bytecodes.

6. *Binding time.*
Choose a computer language other than the ones already covered in the video "Introduction and Scope". It could be a traditional programming language, a scripting language, or a markup language. Do some digging around (Wikipedia is often a good starting point) to learn how the language is implemented, and create a version of the binding time chart for your chosen language. *I strongly recommend doing this problem in a group.*

7. *Designing an architecture.*
The formal notion of an architecture can be used in various domains. In this problem, your task is to specify the architecture of a chess-playing robot. This problem does require familiarity with the rules of chess[1] and algebraic chess notation[2].

Think of a chess game written in algebraic notation[3] as a "high-level program" that is fed to a robot that moves the pieces on a physical chessboard as specified by this program. Design an architecture for this robot and describe it formally as discussed in the video "Machine Architectures". Your design must:

- define what is considered to be "architectural state";
- provide a small set of robot-level "instructions" into which you can "compile" the chess game; and
- formally specify the effects of each instruction on architectural state.

Remember to include some "condition codes" in your state to highlight some interesting situations in the chess game.

[1]See `https://en.wikipedia.org/wiki/Rules_of_chess`.
[2]See `https://en.wikipedia.org/wiki/Algebraic_notation_(chess)`.
[3]Here's a classic example known as the "Opera Game" `https://en.wikipedia.org/wiki/Opera_Game`.