Problem Set 1: Course Introduction, LiveOak, SaM
Solution Key

C S 395T

01 September 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. Any solutions marked with an asterisk (*) may still be attempted and turned in after the release date of the solution key.

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`

Solution:

Identifier	LO-0	LO-1	LO-2	LO-3	Why illegal
`this_variable`	Legal	Legal	Legal	Legal	
`Break`	Legal	Legal	Legal	Legal	
`Bool1`	Legal	Legal	Legal	Legal	
`while`	Legal?	Illegal	Illegal	Illegal	Conflicts with keyword
`check_123`	Legal	Legal	Legal	Legal	
`123_check`	Illegal	Illegal	Illegal	Illegal	Starts with numeric character
`_check_123`	Illegal	Illegal	Illegal	Illegal	Starts with character `'_'`
`string`	Legal	Legal	Legal	Legal	No conflict with type name `String`
`IsThisValid?IDK`	Illegal	Illegal	Illegal	Illegal	Contains illegal character `'?'`
`x`	Legal	Legal	Legal	Legal	

Whether `while` should be allowed as a valid identifier at level 0 is a language design decision. The better choice is to make it illegal, so that there is not an abrupt change in user expectation moving up to level 1.

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.

Solution:

(a) To simplify this part, I am limiting the operators to `+-*/~` on `int` literals and variables and leaving out the ternary operator. The key insight is that we need to group the operators in different levels of precedence and layer the productions to match these levels.

⟨*Expr*⟩ → ⟨*AddExpr*⟩
⟨*AddExpr*⟩ → ⟨*MulExpr*⟩ | ⟨*AddExpr*⟩ `[+-]` ⟨*MulExpr*⟩
⟨*MulExpr*⟩ → ⟨*UnaryExpr*⟩ | ⟨*MulExpr*⟩ `[*/]` ⟨*UnaryExpr*⟩
⟨*UnaryExpr*⟩ → ⟨*PrimaryExpr*⟩ | `~` ⟨*PrimaryExpr*⟩
⟨*PrimaryExpr*⟩ → ⟨*Identifier*⟩ | ⟨*Literal*⟩ | `(` ⟨*Expr*⟩ `)`

Notice the very last production, which allows explicit parentheses to modify the normal order of operations. Parse the three expressions `x+y*z`, `x+(y*z)`, and `(x+y)*z` with this grammar to understand how it works.

(b) (*) Look at the solution to the previous part. How does it encode left-associativity? (Hint: It is manifest as a certain asymmetry in the productions.) Use this insight to modify the grammar to include a right-associative exponentiation operator at the highest level of precedence among the binary operators.

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.

Solution:

(a) Negative integers are not representable as numeric literals. Negative values can be created either using the unary ~ operator or by subtracting from zero.

(b) All representable integers have infinitely many representations, since leading zeros are allowed. This is not a problem because all these representations evaluate to the same value.

(c) ⟨*NUM*⟩ → 0 | [1-9]([0-9])*.

(d) ⟨*NUM*⟩ → ⟨*DecimalNum*⟩ | ⟨*OctalNum*⟩ | ⟨*HexNum*⟩
⟨*DecimalNum*⟩ → [1-9]([0-9])*
⟨*OctalNum*⟩ → 0[0-7]+
⟨*HexNum*⟩ → 0[xX]([0-9a-fA-F])+

(e) (*) Left as an exercise for the student. It is best to proceed by writing out all legal combinations of integer and fraction parts.

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.

Solution:

0-address	1-address	2-address	3-address
PUSH A	LOAD B	MOV R0, B	MOV R1, B
PUSH B	MUL C	MUL R0, C	MUL R1, R1, C
PUSH C	ADD A	ADD R0, A	ADD R1, A, R1
MUL	STORE T1	MOV R1, E	MOV R2, E
ADD	LOAD E	MUL R1, F	MUL R2, R2, F
PUSH D	MUL F	SUB D, R1	SUB R2, D, R2
PUSH E	STORE T2	MOV R2, G	MOV R3, G
PUSH F	LOAD D	MUL R2, H	MUL R3, R3, H
MUL	SUB T2	SUB D, R2	SUB R2, R2, R3
SUB	STORE T2	DIV R0, D	DIV X, R1, R2
PUSH G	LOAD G	MOV X, R0	
PUSH H	MUL H		
MUL	STORE T3		
SUB	LOAD T2		
DIV	SUB T3		
POP X	STORE T4		
	LOAD T1		
	DIV T4		
	STORE X		

(*) Other variations are possible depending on architectural assumption, e.g., whether the 3-address architecture is a load-store or a mixed-mode machine.

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 $x + y$ and $x - y$.

   (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.

Solution: (*) The important points for each example are outlined below. In most of these situations, it helps a lot to draw pictures of the stack organization and how it changes as instructions execute. Detailed solutions are left as an exercise for the student.

(a) Make copies of x and y to evaluate the two expressions. Then use `STOREOFF` instructions to overwrite the original values with these two results.

(b) The key is that multiplication by powers of 2 are equivalent to left-shifts. So $5x + 3y = 4x + x + 2y + y = (x << 2) + (y << 1) + (x + y)$.

(c) This is called the bit count or population count algorithm, and is a classic divide-and-conquer bit-parallel algorithm that can be done in $\lceil \lg n \rceil$ steps for an n-bit argument. Google the algorithm. Since you know that $n = 32$, you simply unroll the recursion to produce straight-line code.

(d) Once you know Euclid's algorithm (google it if needed), this is straightforward to implement in its iterative version. The recursive version is simply an exercise in using the SaM linkage convention properly.

(e) This is an example where you need to access the heap. The algorithmic idea is to swap characters from the two ends, moving inwards.

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.*

Solution: (*) Given the open-ended nature of this problem, there isn't a definitive answer. I will be happy to review your solution for your chosen language if you would like me to.

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.

Solution: (*) Again, given the open-ended nature of this problem, there isn't a definitive answer. Here is a sketch of my solution. I will be happy to review your solution if you would like me to.

Architectural state will clearly need to include where you are in the game (the PC), the state of each chessboard square (the memory), and possibly the position of the robot arm in terms of chessboard coordinates (although you can keep this component implicit).

For instructions, I would choose a small set of primitives: maybe moving to a square, picking up a piece, and putting down a piece. You can build up moving a piece, capturing a piece, castling, *en passant*, and other situations from this simple set. Specifying the effects of these instructions is pretty straightforward.

For condition codes, the most obvious ones are check and checkmate, for both players. You might also consider having an indicator for whether castling is allowed for each player at the current point in the 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`.