1 identifiers in LiveOak

Identifier: start with alphabets, then followed by 0 or more a-zA-Z0-9_

(a) this_variable

(b) Break

(c) Bool1

(d) while : reserved?

(e) check_123

(f) 123_check : start with number

(g) _check_123: starts with _

(h) string: reserved?

(i) IsThisValid?IDK : has ?

(j) x

2 Tradeoff between ease of use and ease of implementation

(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.

```
EXPR ? : EXPR : EXPR |
EXPR BINOP EXPR |
UNOP EXPR |
VAR |
LITERAL
```

(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.

```
EXPR ^ EXPR |
EXPR ^ EXPR
```

[Q: missing wikipedia page]

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?

```
negative integers, use BINOP "-" and NUM
```

(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?

```
0, 00, 000 ... 0, 01, ... no
```

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

```
[1-9][0-9]* | 0
```

(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.

```
Octal: 0[0-8]+
Hex: 0x[A-F0-9]
```

(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.

```
[0-9]+.[eE] | .[0-9]+[eE] | [0-9]+.[0-9]+[eE]
```

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.

0- address machine

```
No registers, **only manipulate TOS**
(postfix) A B C * + D E F * - G H * - /
PUSH A
PUSH B
PUSH C
MUL
ADD
PUSH D
PUSH E
PUSH F
MUL
SUB
PUSH G
PUSH H
MUL
SUB
DIV
```

1-address machine

```
1 source/ destination address
Manipulate TOS + **accumulator** (1 other place to store result)
Calculate denominator, store in temp, then calculate the rest
LOAD G    AC <- M[G]
MUL H   AC <- AC * M[H]
STORE T1  M[T1] <- AC (G*H)
LOAD E    AC (accumulator) <- M[E]
```

```
MUL F   AC <- AC * M[F]
STORE T2  M[T2] <- E * F
LOAD D    AC <- M[D]
SUB T2    AC <- D - E*F
SUB T1    AC <- AC - M[T1]
STORE T3  M[T3] <- D - E * F - G * H
LOAD B
MUL C
ADD A
DIV T3
STORE T4
```

2-address machine

```
1 source address + 1 destination address
Has **Rn** to store temp results
MOV R1,B  R1 <- M[B]
MUL R1,C  R1 <- R1*M[C]
ADD R1,A  R1 <- R1*M[A]
MOV R2,D
MOV R3,E
MUL R3,F
SUB R2,R3
MOV R3,G
MUL R3,H
SUB R2,R3
DIV R1,R2
MOV R1,X  Move back to stack
```

3-address machine

```
1 destination addresses + 2 source address
MUL R1,B,C    R1 <- M[B]*M[C]
ADD R1,R1,A   R1 <- R1 + M[A]
MUL R2,E,F
SUB R2,D,R2
MUL R3,G,H
SUB R2,R2,R3
DIV X,R1,R3
```

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 $x - y$.

x + y:

```
ADD (replace top two items by result)
```

x - y:

```
SUB
```

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

```
DUP
DUP   [x,y,y,y]
ADD
ADD   [x,3y]
POP a [x], a=3y
DUP * 4
ADD * 4
ADD a
```

(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.

[Q: need instruction set]

```

```

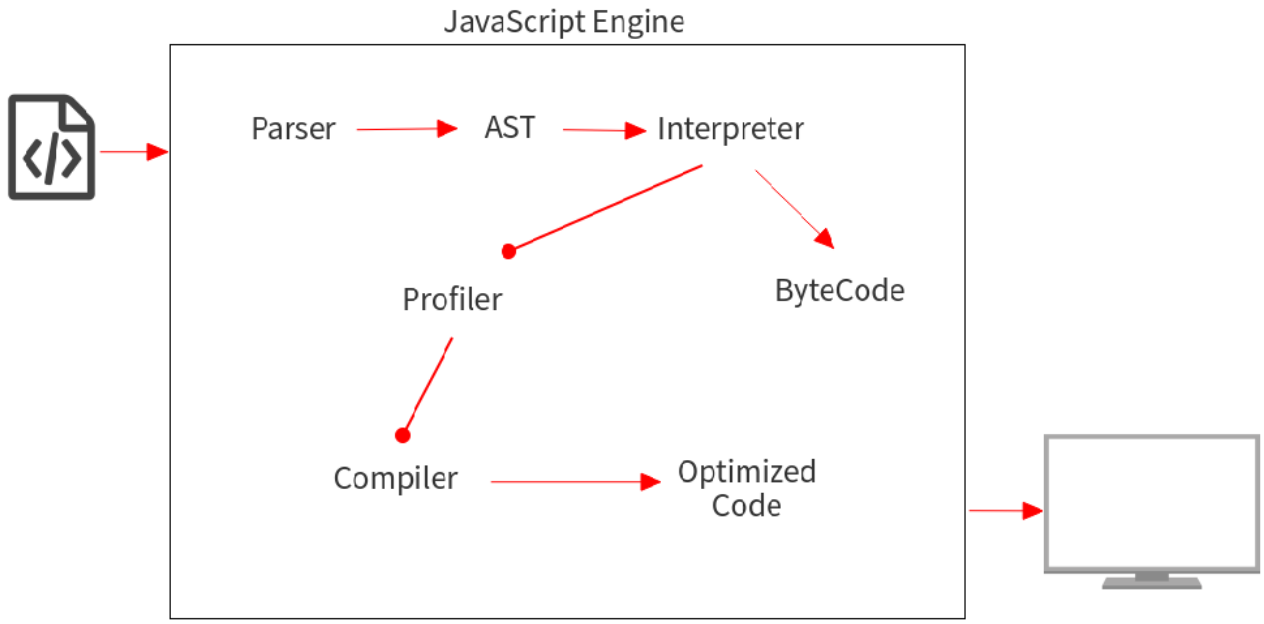
(d) Given an initial stack containing two positive integers x and y, return a final stack containing the greatest common divisor of x and y. Use Euclid's algorithm in either a recursive or iterative form.
(e) Given an initial stack containing a pointer to a heap-allocated character string, reverse that 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.

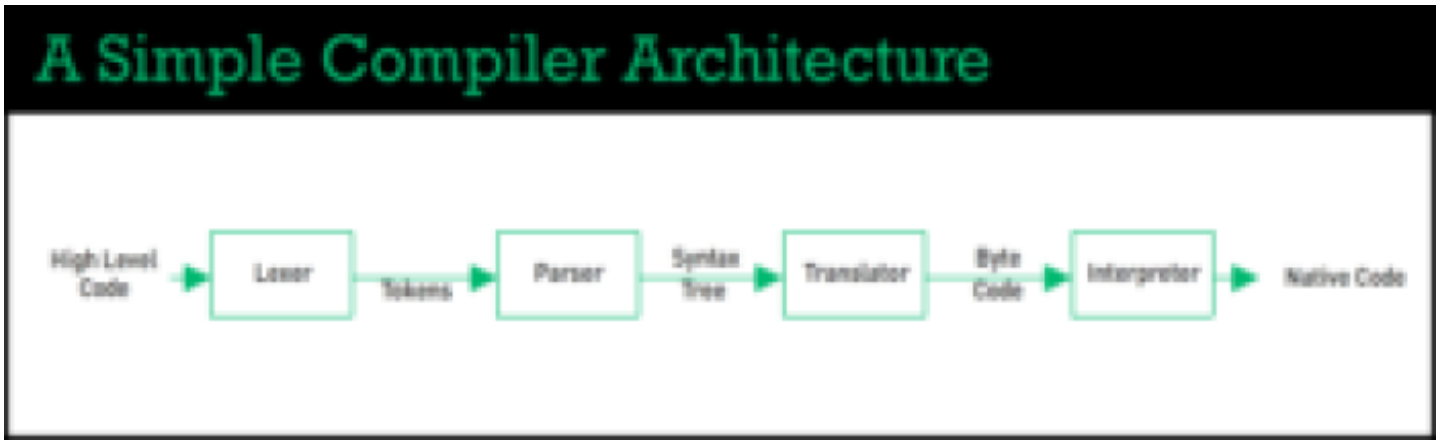
JavaScript

https://blog.bitsrc.io/javascript-under-the-hood-632ccae06b27



All happens during execution, (V8 engine) to ByteCode is faster than Profiler - compiler - path. Platform independent

https://softwareengineeringdaily.com/2018/10/03/javascript-and-the-inner-workings-of-your-browser/



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.

https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

```
Architectural state: the board and position of each piece

MOV srcLoc targetLoc
RMV loc
END
```