Mock Examination Do Not Use!

001956 / 102691	Computer Systems - UG / PG COMP SCI 2000 / 7081

Official Reading Time:	10 mins
Writing Time:	120 mins
Total Duration:	130 mins

Questions	Time	Marks
Answer all 12 questions	120 mins	120 marks
		120 Total

Instructions for Candidates

- This is a Closed-book examination.
- Begin each answer on a new page.
- Examination material must not be removed from the examination room.

Materials

- Foreign Language Dictionaries are Permitted for Translation Only

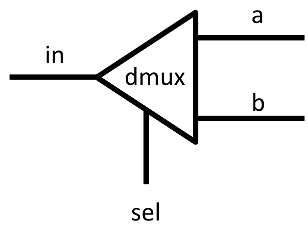
DO NOT COMMENCE WRITING UNTIL INSTRUCTED TO DO SO

Mock Examination Do Not Use!

Basic Gates and Boolean Logic

Question 1

(a) **Published in Mock Exam**
The following diagram shows a one bit de-multiplexor (dmux) chip.



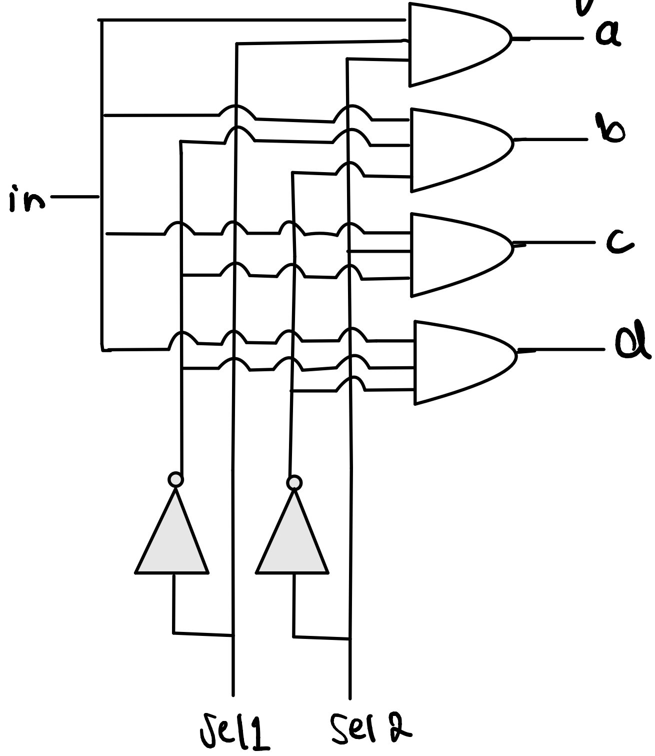
This chip directs the signal from `in` to either `a` or `b` depending on the value of `sel`. The non selected ouput is zero.

Now, given the 1-bit dmux above, draw an implementation for a dmux with four outputs and a two-bit selector. In your diagram assume that `in` remains as one bit.

[4 marks]

[Total for Question 1: 4 marks]

Dmux - 4 way



Mock Examination Do Not Use!

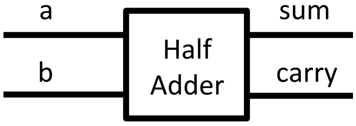
Boolean Arithmetic and ALU design

Question 2

For the following questions you may find the information in Figures 1 and 2 in the appendix of this paper useful.

(a) **Published in Mock Exam**

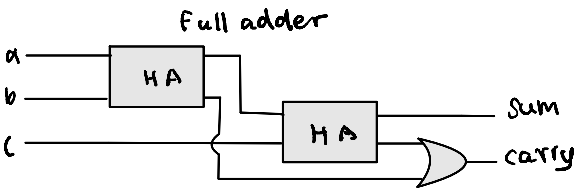
The following is a diagram the interface of a 1 bit half-adder:

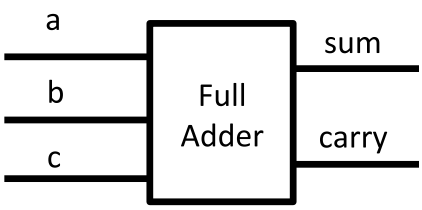


a half-adder sums its two input bits to produce a sum bit and a carry bit.

Answer the following:

i. Draw an implementation of a full-adder chip composed from half-adder chips and/or other gates. Recall that the interface for a full adder is:





[7 marks]

ii. Write the code in the PARTS section of a HDL file describing the full-adder you defined in your answer to part (i) above. In your code you must assume that the inputs to the full adder are as labelled in the diagram above.

[6 marks]

[Total for Question 2: 13 marks]

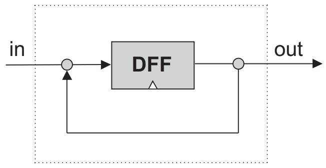
HalfAdder (a=a, b=b, sum=suma, carry=carrya)
HalfAdder (a=suma, b=c, sum=sum, carry=carryb)
Or(a=carrya, b=carryb, out=carry)

Mock Examination Do Not Use!

Sequential Logic

Question 3

(a) **Published in Mock Exam**
Look at the following diagram for an invalid design for a 1-bit register from figure 3.1 of the textbook.



Answer the following.

i. Briefly explain what is wrong with the design of the register above.
memory is overwritten every cycle [2 marks]

ii. Draw a correct design for the 1-bit register above and write down the equality that explains the relationship between the `in` and `out` wires.
[4 marks]

[Total for Question 3: 6 marks]

Hack Assembler and Machine Code

Question 4

For the following questions you may find the information in Figures 3, 4, 5, 6, 7 and 8 in the appendix of this paper useful.

(a) **Published in Mock Exam**
Look at the following Hack machine code:

```
0000000000010000  A   @16
1111110010001000  C   M = M-1
1111110000010000  C   D = M
0000000000000000  A   @0
1110001100000001  C   D;JGT
0000000000000101  A   @5
1110101010000111  C   0;JMP
```

Answer the following:

i. Using the instruction formats in Figures 3, 4, 5, 6, and 7 as a guide, write down the Hack assember instructions that are equivalent to this code.
[7 marks]

ii. Describe what the machine code does.
Changes data in RAM[16] to be itself - 1 [3 marks]
[Total for Question 4: 10 marks]
Stores result in D
if result > 0, jump to start
if < 0, jump to @5

Mock Examination Do Not Use!

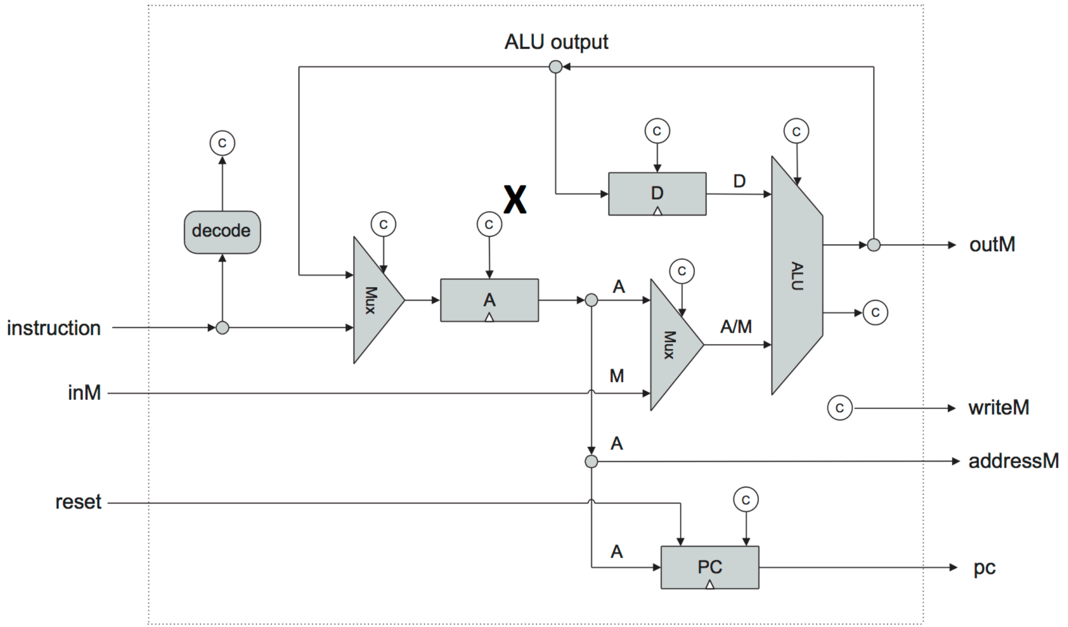
Computer Architecture

Question 5

For the following questions you may find the information in Figures 1, 2, 3, 4, 5, 6, 7 and 8 in the appendix of this paper useful.

(a) **Published in Mock Exam**

Look at the following partial diagram of a Hack CPU taken from Figure 5.9 of the textbook:



Some of the control logic is missing from this diagram. These missing gates and wires are marked with a ⓒ symbol. In the diagram one such section of missing control logic is marked with a large **X**. Given what you know about Hack instruction formats and ALU design, describe in detail what this missing control logic is.

Hint: feel free to use the figures in the appendix for some of the information you need.

[6 marks]

[Total for Question 5: 6 marks]

d1 OR (Not i15)

- A instruction (MSB, i15 is 0) outputs not 0 (1), ensuring A is always loaded
- c instruction (MSB, i15 is 1) outputs not 1 (0), which means only allow writing to A register if d1 is true (1)

Mock Examination Do Not Use!

Assembler

Question 6

(a) **Published in Mock Exam**
Look at the following Hack assembler code:

`@X`	0	0000 0000 0001 0000
`D=M`	1	1111 1100 0001 0000
`@END`	2	0000 0000 0000 0110
`D;JGE`	3	1110 0011 0000 0011
`@X`	4	0000 0000 0001 0000
`M=-M`	5	1111 1100 1100 1000
`(END)`		0000 0000 0000 0110
`@END`	6	
`0;JMP`	7	1110 1010 1000 0111

Hand-assemble this code by writing out the binary machine code the assembler would produce. For this question you may find the information in Figures 3, 4, 5, 6, and 7 useful.

X 16

END 6

[9 marks]

[Total for Question 6: 9 marks]

Virtual Machine - Expressions

Question 7

(a) **Published in Mock Exam**
Translate the following Jack let statement into Hack Virtual Machine language:

`let d = ((2 - x) * y) + 5`

The variables d, x and y are in memory segment *local* at indexes 2,5 and 7 respectively. Assume there is a function named *multiply* that will take two arguments and return the result of multiplying the two numbers together.

[8 marks]

[Total for Question 7: 8 marks]

push constant 2
push local 5
sub
push local 7
call multiply 2
push constant 5
add
pop local 2

Mock Examination Do Not Use!

Virtual Machine - Subroutines

Question 8

(a) **Published in Mock Exam**
The Hack Virtual Machine language provides three function related commands:

- call f m
- function f n
- return

i. Briefly describe what the function command does during program execution.

[2 marks]

ii. Briefly describe what the call command does during program execution.

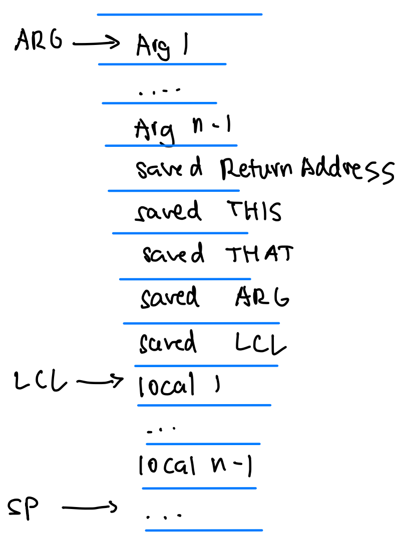
[7 marks]

iii. Briefly describe what the return command does during program execution.

[8 marks]

call

1. caller pushes callee's args onto stack, then function is called
2. caller pushes return address to stack (address of next instruction from caller)
3. caller pushes it's mem segments (THIS, THAT, ARG, LCL) to stack
4. caller sets ARG to be first argument of callee
5. caller sets LCL to be SP
6. Jump to callee and begin execution



function

1. push nVars onto stack and init to 0
2. begin execution

return

1. set a temporary variable = LCL
2. Extract return address relative to temp
3. pop the result and store in *ARG
4. Set SP to be ARG+1

this recycles the memory used by callee allowing it to be overwritten

5. Extract memory segments (THIS, THAT, ARG, LCL) relative to temp and restore for caller to continue execution
6. Jump to return address to continue caller's execution

[8 marks]

(b) **Published in Mock Exam**
The Hack Virtual Machine allocates an area of the stack for each active function call. Briefly describe the structure of one of these stack frames immediately after the execution of the function command in a **Jack** method that is declared with N parameters and M local variables.

[9 marks]

[Total for Question 8: 26 marks]

k

```
ARG -> Arg 1
       ....
       Arg n-1
       saved Return Address
       saved THIS
       saved THAT
       saved ARG
       saved LCL
LCL -> local 1      = 0
       ...          = 0
       local M-1    = 0
SP  -> ...
```

Question 9

(a) **Published in Mock Exam**
Write a **Jack** program that calls a recursive function to calculate the 7th fibonacci number. The result must be placed in an int variable x.

[8 marks]

[Total for Question 9: 8 marks]

```
Class Main {
    function int Fib(int n) {
        if (n=0) {
            return 0
        } else if (n=1) {
            return 1 } else {
        return Fib(n-1) + Fib(n-2); }
    }

    function int main() {
        var int x;
        let x = Fib(7)
    }
}
```

Mock Examination Do Not Use!

Parsing

Question 10

(a) **Published in Mock Exam**
Show the two symbol tables for the following code just ater the last variable declaration in the method has been parsed.

```
class BankAccount
{
        // Class variables
        static string key ;
        static int nAccounts ;

        // Instance variables ;
        field string owner ;
        field int balance ;

        method void transfer(int sum, BankAccount b2)
        {
                var Date due ;
                var int i,j ;

                let i = sum ;
        }
}
```

	name	type	segment	offset
class	key	String	static	0
	nAccounts	int	static	1
	owner	String	this	0
	balance	int	this	1
method	this	bankAccount	argument	0
	sum	int	argument	1
	b2	bankAccount	argument	2
	due	Date	local	0
	i	int	local	1
	j	int	local	2

[10 marks]

[Total for Question 10: 10 marks]

Code Generation

Question 11

(a) **Published in Mock Exam**
Consider the following **Jack** method:

```
method int useless(String x, String y)
{
        var Array local1 ;
        var int local0 ;
        var string local3 ;
}
```

What Hack Virtual Machine language code would implement the following Jack program fragments if they were in the body of the method `useless`?

i. `let local1[7] = x ;`

[6 marks]

ii. `return local0 + 1 ;`

[4 marks]

[Total for Question 11: 10 marks]

i). push arg 0
pop pointer 0
push this 0
push constant 7
add
pop pointer 1
push arg 1
pop that 0

ii). push arg 0
pop pointer 0
push this 1
push constant 1
add
return

Mock Examination Do Not Use!

Jack OS, Optimisation

Question 12

(a) **Published in Mock Exam**
How do caches take advantage of temporal and spatial locality to improve the performance of a computer?

[4 marks]

(b) **Published in Mock Exam**
What determines the minimum length of a clock cycle in a processor?

[2 marks]

(c) **Published in Mock Exam**
The Jack Operating System provides a small number of libraries that extend the functionality of the Jack programming language. Excluding support for graphical user interfaces, identify two operating system services that are not provided by the Jack OS but are provided by Linux. In each case explain why the service is important.

[4 marks]

[Total for Question 12: 10 marks]

a) temporal locality

- data accessed recently will likely be accessed again soon.
- looping through same instructions
- store recent data in cache

spatial locality

- data near recently accessed data will be accessed soon
- sequentially reading an array
- store blocks of data, not just specific addresses

b) minimum clock cycle length determined by:

time for signal in slowest logic path (longest path)

Mock Examination Do Not Use!

APPENDICES

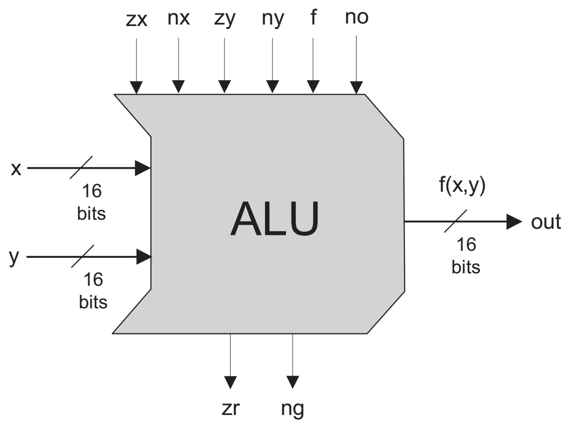


Figure 1: An interface diagram for the ALU. From figure 2.5 of the textbook.

These bits instruct how to preset the `x` input		These bits instruct how to preset the `y` input		This bit selects between `+ / And`	This bit inst. how to postset `out`	Resulting ALU output
`zx`	`nx`	`zy`	`ny`	`f`	`no`	`out=`
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`f(x,y)=`
1	0	1	0	1	0	`0`
1	1	1	1	1	1	`1`
1	1	1	0	1	0	`-1`
0	0	1	1	0	0	`x`
1	1	0	0	0	0	`y`
0	0	1	1	0	1	`!x`
1	1	0	0	0	1	`!y`
0	0	1	1	1	1	`-x`
1	1	0	0	1	1	`-y`
0	1	1	1	1	1	`x+1`
1	1	0	1	1	1	`y+1`
0	0	1	1	1	0	`x-1`
1	1	0	0	1	0	`y-1`
0	0	0	0	1	0	`x+y`
0	1	0	0	1	1	`x-y`
0	0	0	1	1	1	`y-x`
0	0	0	0	0	0	`x&y`
0	1	0	1	0	1	`x|y`

Figure 2: The Hack ALU truth table. From figure 2.6 of the textbook.

Mock Examination Do Not Use!

A-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

value (v = 0 or 1)

Binary:																
	0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v

Figure 3: The format of an A-instruction. From page 64 of the text book.

C-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.

Binary:																
	1	**1**	**1**	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3
					comp						*dest*			*jump*		

Figure 4: The format of an C-instruction. From page 66 of the text book.

(when a=0) ***comp** mnemonic*	c1	c2	c3	c4	c5	c6	(when a=1) ***comp** mnemonic*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

Figure 5: The meaning of C-instruction Fields. From figure 4.3 of the textbook.

Mock Examination Do Not Use!

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	`null`	The value is not stored anywhere
0	0	1	`M`	Memory[A] (memory register addressed by A)
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D register
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A], and D register

Figure 6: The meaning of the destination bits of the C-instruction From figure 4.4 of the textbook.

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	**Mnemonic**	**Effect**
0	0	0	`null`	No jump
0	0	1	`JGT`	If $out > 0$ jump
0	1	0	`JEQ`	If $out = 0$ jump
0	1	1	`JGE`	If $out \geq 0$ jump
1	0	0	`JLT`	If $out < 0$ jump
1	0	1	`JNE`	If $out \neq 0$ jump
1	1	0	`JLE`	If $out \leq 0$ jump
1	1	1	`JMP`	Jump

Figure 4.5 The *jump* field of the *C*-instruction. *Out* refers to the ALU output (resulting from the instruction's *comp* part), and *jump* implies "continue execution with the instruction addressed by the A register."

Figure 7: The meaning of the jump bits of the C-instruction From figure 4.5 of the textbook.

Label	*RAM address*
`SP`	0
`LCL`	1
`ARG`	2
`THIS`	3
`THAT`	4
`R0-R15`	0-15
`SCREEN`	16384
`KBD`	24576

Figure 8: The predefined symbols in Hack Assembly language. From page 110 of the text book.

Mock Examination Do Not Use!

Lexical Elements

```
keyword         ::= ’class’ | ’constructor’ | ’function’ | ’method’ | \
                    ’field’ | ’static’ | ’var’ | ’int’ | ’char’ | \
                    ’boolean’ | ’void’ | ’true’ | ’false’ | ’null’ | \
                    ’this’ | ’let’ | ’do’ | ’if’ | ’else’ | ’while’ | \
                    ’return’
symbol          ::= ’{’ | ’}’ | ’(’ | ’)’ | ’[’ | ’]’ | ’.’ | \
                    ’,’ | ’;’ | ’+’ | ’-’ | ’*’ | ’/’ | ’&’ | \
                    ’|’ | ’<’ | ’>’ | ’=’ | ’~’ | ’
integerConstant ::= A decimal number in the range 0 .. 32767
stringConstant  ::= ’"’ A sequence of Unicode characters not including
                        double quote or newline ’"’
identifier      ::= A sequence of letters, digits and underscore (’_’)
                    not starting with a digit.
```

Statements

```
statements      ::= statement*
statement       ::= letStatement | ifStatement | whileStatement | \
                    doStatement | returnStatement}
letStatement    ::= ’let’ varName (’[’ expression ’]’)? ’=’ expression ’;’
ifStatement     ::= ’if’ ’(’ expression ’)’ ’{’ statements ’}’ \
                   (’else’ ’{’ statements ’}’)?
whileStatement  ::= ’while’ ’(’ expression ’)’ ’{’ statements ’}’
doStatement     ::= ’do’ subroutineCall ’;’
returnStatement ::= ’return’ expression? ’;’
```

Expressions

```
expression      ::= term (op term)*
term            ::= integerConstant | stringConstant | \
                    keywordConstant | varName | \
                    varName ’[’ expression ’]’ | subroutineCall | \
                    ’(’ expression ’)’ |  unaryOp term
subroutineCall  ::= subroutineName ’(’ expressionList ’)’ |  \
                    (className | varName) ’.’ subroutineName ’(’ expressionList ’)’
expressionList  ::= (expression (’,’ expression)*)?
op              ::= ’+’ | ’-’ | ’*’ | ’/’ | ’&’ | ’|’ | ’<’ | ’>’ | ’=’
unaryOp         ::= ’-’ | ’~’
keywordConstant ::= ’true’ | ’false’ | ’null’ | ’this’
varName         ::= identifier
```

Figure 9: The Jack grammar. From figure 10.5 of the textbook.

END OF EXAMINATION PAPER

Mock Examination Do Not Use!

Lexical Elements

```
keyword         ::= 'class' | 'constructor' | 'function' | 'method' | \
                    'field' | 'static' | 'var' | 'int' | 'char' | \
                    'boolean' | 'void' | 'true' | 'false' | 'null' | \
                    'this' | 'let' | 'do' | 'if' | 'else' | 'while' | \
                    'return'
symbol          ::= '{' | '}' | '(' | ')' | '[' | ']' | '.' | \
                    ',' | ';' | '+' | '-' | '*' | '/' | '&' | \
                    '|' | '<' | '>' | '=' | '~' | '
integerConstant ::= A decimal number in the range 0 .. 32767
stringConstant  ::= '"' A sequence of Unicode characters not including
                        double quote or newline '"'
identifier      ::= A sequence of letters, digits and underscore ('_')
                    not starting with a digit.
```

Statements

```
statements      ::= statement*
statement       ::= letStatement | ifStatement | whileStatement | \
                    doStatement | returnStatement}
letStatement    ::= 'let' varName ('[' expression ']')? '=' expression ';'
ifStatement     ::= 'if' '(' expression ')' '{' statements '}' \
                   ('else' '{' statements '}')?
whileStatement  ::= 'while' '(' expression ')' '{' statements '}'
doStatement     ::= 'do' subroutineCall ';'
returnStatement ::= 'return' expression? ';'
```

Expressions

```
expression      ::= term (op term)*
term            ::= integerConstant | stringConstant | \
                    keywordConstant | varName | \
                    varName '[' expression ']' | subroutineCall | \
                    '(' expression ')' |  unaryOp term
subroutineCall  ::= subroutineName '(' expressionList ')' |  \
                    (className | varName) '.' subroutineName '(' expressionList ')'
expressionList  ::= (expression (',' expression)*)?
op              ::= '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
unaryOp         ::= '-' | '~'
keywordConstant ::= 'true' | 'false' | 'null' | 'this'
varName         ::= identifier
```

Figure 9: The Jack grammar. From figure 10.5 of the textbook.

END OF EXAMINATION PAPER