Quiz - Lecture 18

Due 9 Oct at 23:59 **Points** 5 **Questions** 5 **Available** 8 Oct at 9:10 - 9 Oct at 23:59 1 day
Time limit None **Allowed attempts** Unlimited

Take the quiz again

Attempt history

	Attempt	Time	Score
KEPT	Attempt 2	2 minutes	5 out of 5
LATEST	Attempt 2	2 minutes	5 out of 5
	Attempt 1	less than 1 minute	0.5 out of 5

Score for this attempt: **5** out of 5
Submitted 9 Oct at 11:10
This attempt took 2 minutes.

Question 1 — 1 / 1 pts

Consider these rules for a tokeniser (* means 0 or more):

```
int ::= '0' | (('1'-'9')('0'-'9')*)
name ::= ('a'-'z') ( 'a'-'z'|'0'-'9' )*
```

What tokens would be produced from the following string assuming that whitespace is not returned as tokens?

"43 bobis50 99balloons"

- int "43", name "bobis50", name "99balloons"
- int "43", name "bobis", int "50", int "99", name "balloons"
- int "43", name "bobis", int "50", name "99balloons"

Correct!

- int "43", name "bobis50", int "99", name "balloons"

Question 2

1 / 1 pts

Consider these rules for a tokeniser, ***** means 0 or more, **?** means 0 or 1:

```
num ::= ('0' | (('1'-'9')('0'-'9')*)) ('.' ('0'-'9')*)?
dot ::= '.'
```

How many tokens would be produced from the following string?

"07.09.0.0.0"

- 7

- 1
- 3
- Correct! 6
- 5
- 11

The correct answer is 6:

```
"0"   "7.09"   "."   "0.0"   "."   "0"
```

Question 3

1 / 1 pts

Indicate the ordering of the following components of a Jack compiler that produces Hack Virtual Machine code. Do not include optional components?

Correct! **tokeniser** — this is first

Correct! **parser**

after the tokeniser

Correct!	**print an abstract syntax tree as XML**	this is optional
Correct!	**parse XML to construct an abstract syntax tree**	this is optional
Correct!	**code generator**	after the parser
Correct!	**optimiser**	this is optional
Correct!	**print an optimised abstract syntax tree as XML**	this is optional
Correct!	**parse XML to construct an optimised abstract syntax tree**	this is optional

Other Incorrect Match Options:

- after parsing XML to construct an abstract syntax tree
- after the optimiser
- after printing an abstract syntax tree as XML
- after printing an optimised abstract syntax tree as XML
- after the code generator
- after parsing XML to construct an optimised abstract syntax tree

The only essential components are the tokeniser, followed by the parser, followed by the code generator. Everything else is optional.

Question 4

1 / 1 pts

You are writing a parser in C++ that involves creating a parse tree for a while loop. The tree for a while loop contains a sub-tree for the loop condition and a sub-tree for the loop body. Your parser includes the following code to construct the tree.

```
...
while_tree = create_while_tree(parse_condition(), parse_body()) ;
...
```

You have passed all the tests on your own computer. Which of the following could happen when your parser is run on the web submission system?

- [ ] It compiles with warnings.

Correct!

- [x] It passes all the tests.

If you are using the clang++ compiler and the web submission's test scripts also use the clang++ compiler, this will probably happen.

- [ ] It fails all tests involving while loops.

Correct!

- [x] It reports finding a '(' when it is expecting to see '{'.

If the parse_body() function is called first, it will be looking for the '{' that starts the loop body but instead find the '(' that starts the loop condition.

Correct!

It successfully builds a parse tree for:

- [x] `while { } ( true )`

If the parse_body() function is called first, this would accidentally be parsed as a syntactically correct while loop!

Correct!

- [x] It does not compile.

Different C++ compilers may automatically include extra ".h" files when they compile programs. If your C++ compiler was being too helpful it may have included a file that is not included on the web submission system and that may cause your parser to not compile!

The code fragment in the question contains a serious error. It will only work as you intend if the C++ compiler generates code that calls the parse_condition() function before the parse_body() function. Unfortunately, the C++ compiler is free to generate code that calls parse_body() first! If you are a Java programmer it is very easy to make this mistake because Java will always execute the calls in right order.

Question 5

1 / 1 pts

Consider the grammar rule for a Jack local variable declaration:

```
varDec ::= 'var' type varname (',' varname)* ';'
```

Which of the following code fragments would be able to parse local variable declarations?

```
mustbe(tk_var) ;
parse_type() ;
parse_varname() ;
mustbe(tk_comma) ;
parse_varname() ;
mustbe(tk_semi) ;
```

Correct!

```
mustbe(tk_var) ;
parse_type() ;
parse_varname() ;
while (have(tk_comma))
{
    mustbe(tk_comma) ;
    parse_varname() ;
}
mustbe(tk_semi) ;
```

```
mustbe(tk_var) ;
parse_type() ;
parse_varname() ;
mustbe(tk_semi) ;
```

```
mustbe(tk_var) ;
mustbe(tk_identifier) ;
mustbe(tk_identifier) ;
while (have(tk_comma))
{
    next_token() ;
    mustbe(tk_identifier) ;
}
mustbe(tk_semi) ;
```

Quiz score: **5** out of 5