Assignment 3 jack_parser()

jack_parser()

Description

You must complete the implementation of the **jack_parser()** function in the file **parser.cpp**. The function **jack_parser()** is used to implement a program **jparser** that reads a Jack class from standard input and writes an XML representation of its abstract syntax tree to standard output. The **jack_parser()** function uses the tokeniser functions described in **tokeniser.h** to parse a Jack class read from standard input and construct an abstract syntax tree using the functions as described in **csdocument.h**. The precompiled main program is responsible for calling the **jack_parser()** function and passing the result to the function **cs_print()**. The **cs_print()** function is responsible for writing an XML representation of the abstract syntax tree to standard output.

Compiling and Running jparser

When the **Makefile** attempts to compile the program **jparser**, it will use the file **parser.cpp**, any other **.cpp** files it can find whose names start **parser-** and any **.cpp** files it can find whose names start with **shared-**. For example, if we have our own class **abc** that we want to use when implementing **jack_parser()** and our own class **xyz** that we want to use with all of our functions, we would name the extra files, **parser-abc.cpp** and **shared-xyz.cpp** respectively with matching **parser-abc.h** and **shared-xyz.h** include files.
The program can be compiled using the command:
% make parser
The suite of provided tests can be run using the command:
% make test-parser
To just run the provided tests that should not compile, use the command:
% make test-parser-dnc
The test scripts do not show the program outputs, just passed or failed, but they do show you the commands being used to run each test. You can cut-paste these commands if you want to run a particular test yourself and see all of the output.
Note: Do **not** modify the provided **Makefile** or the sub-directory **includes** or the sub-directory **lib**. These will be replaced during testing by the web submission system.

Tokeniser

The tokeniser functions described in **tokeniser.h** return the following Jack tokens. This table is based on Figure 10.5 from the textbook and shows the tokens that the **jacktokens** tokeniser recognises.

Token Returned	Token Class		Definition / Value

nothing	comment	::=	'//' any characters until end of line | '/*' any characters up to and including the first '*/' | '/**' any characters up to and including the first '*/'
nothing	whitespace	::=	space, tab, carriage return or newline
the keyword, eg "method"	"keyword"	::=	'class' | 'constructor' | 'function' | 'method' | 'field' | 'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' | 'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' | 'while' | 'return'
the symbol, eg "{"	"symbol"	::=	'{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'
"integerConstant"	"integerConstant"	::=	('0'-'9')('0'-'9')*
"stringConstant"	"stringConstant"	::=	'"' A sequence of printable ASCII characters and whitespace characters not including double quote or newline '"'
"identifier"	"identifier"	::=	('a'-'z' | 'A'-'Z' | '_')('a'-'z' | 'A'-'Z' | '0'-'9' | '_')*
"?"	?	::=	end of file, or a non whitespace character, or integer outside the range 0 to 32767, or any other error

Notes:

- All input is read using **cin**.
- Whitespace characters are one of space, tab, carriage return or newline.
- An integerConstant is in the range 0 to 32767.
- The value of a stringConstant does not include the enclosing double quotes.
- A stringConstant can only contain whitespace or printable characters but not double quote or newline.
- No error messages are output.
- All errors are reported by returning the extra token "**?**". This token is not part of a legal Jack program.
- Once a "**?**" token is returned, all future attempts to read a token will return "**?**".
- In a definition, round brackets **( )** are used to group components together.
- In a definition, ***** denotes 0 or more occurrences of the preceding component.
- In a definition, **|** denotes an alternative definition for a component.

jack_parser()

A parser goes over the tokenised text and emits output indicating that it "understood" the text's grammatical structure. In order to do so, the parser must include functions that look for canonical structures in a certain language - in our case Jack - and then emit these structures

in some agreed upon formalism. The structure of your **jack_parser()** function should follow the one developed in workshops 10 and 11 rather than the structure in the textbook.
The following three tables are based on Figure 10.5 of the textbook and describe the grammar of the Jack language that must be recognised:

Classes		Definition
program	::=	One or more classes, each class in a separate file named <className>'.Jack'
class	::=	'class' className '{' classVarDecs subroutineDecs '}'
classVarDecs	::=	(staticVarDec | fieldVarDec)*
staticVarDec	::=	'static' type varName (',' varName)* ';'
fieldVarDec	::=	'field' type varName (',' varName)* ';'
type	::=	'int' | 'char' | 'boolean' | className
vtype	::=	'void' | 'int' | 'char' | 'boolean' | className
subroutineDecs	::=	(constructor | function | method)*
constructor	::=	'constructor' className subroutineName '(' parameterList ')' subroutineBody
function	::=	'function' vtype subroutineName '(' parameterList ')' subroutineBody
method	::=	'method' vtype subroutineName '(' parameterList ')' subroutineBody
parameterList	::=	((type varName) (',' type varName)*)?
subroutineBody	::=	'{' varDecs statements '}'
varDecs	::=	varDec*
varDec	::=	'var' type varName (',' varName)* ';'
className	::=	identifier
subroutineName	::=	identifier
varName	::=	identifier

Statements		Definition
statements	::=	statement*
statement	::=	letStatement | ifStatement | whileStatement | doStatement | returnStatement
letStatement	::=	'let' (varName | arrayIndex) '=' expression ';'
ifStatement	::=	'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
whileStatement	::=	'while' '(' expression ')' '{' statements '}'
doStatement	::=	'do' ((className | varName) '.')? subroutineName '(' expressionList ')' ';'
returnStatement	::=	'return' expression? ';'

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

Notes:

- All input must be read using the tokeniser functions described in **tokeniser.h**.
- You should use the symbol table functions described in **symbols.h**.
- There must be no output written to **cerr** or **cout** or using the **iobuffer** functions.

- During testing you may write error messages and other log messages to **cerr**. These must be removed before you submit your work. The tokeniser_context() function will show the tokeniser's current position in the input being parsed.
- If a parsing error occurs, exit(0) must be called immediately.
- In a definition, round brackets **( )** are used to group components together.
- In a definition, **?** denotes 0 or 1 occurrence of the preceding component.
- In a definition, ***** denotes 0 or more occurrences of the preceding component.
- In a definition, **|** denotes an alternative definition for a component.

The Abstract Syntax Tree

The abstract syntax tree returned by the **jack_parser()** function should contain one node for each rule given in the tables above with the following exceptions.
There is no **program** node. Jack source files only contain a single class so the root node must be a **class**.
Each variable declaration must have its own **varDec** node. There are no **staticVarDec**, **fieldVarDec** nodes, these are replaced by a list of **varDec** nodes, one for each static or field variable. The **parameterList** node has one child **varDec** node for each parameter in the list. Local variables have one **varDec** node for each variable. The order of these nodes must match the order in which the variables are declared. Each **varDec** node must have four children in this order, a **varName** node, a **type** node, a **segment** node and an **offset** node.
The first child node of a **constructor**, **function** or **method** node must be a **type** node, not a **className** node or a **vType** node.
The **expression** node must only have a single child node. When parsing an expression, each time a new infixOp is found, a new **infix** node is created. The first child of the **infix** node is the node representing the expression parsed so far, the second child is an **infixOp** node, and the third child is the node representing the next term to be parsed. The **infix** node becomes the single child that will be appended to the **expression** node.
When parsing an expression, each time a new a new unaryOp is found, a new **unary** node is created. The first child of the **unary** node is an **unaryOp** and the second child is the node representing the next term to be parsed. The **unary** node becomes the single child that will be appended to the **expression** node.
When creating nodes to represent a subroutine call in a do statement or expression where no varName or className has been provided, the subroutine is assumed to be a method of the class being parsed. Therefore, a **keywordConstant** node containing **this** should be created.
The following nodes must have a child text_node added to record their corresponding token values, **type**, **className**, **subroutineName**, **varName**, **infixOp, unaryOp**, **keywordConstant**, **integerConstant, stringConstant**, **segment** and **offset**. **Note**, in the case of **type** nodes, the **className** node is never created.
Please review the expected test outputs if you are unsure of any of these requirements.

Errors to Catch

There are lots of different kinds of errors that a compiler may be able to detect. However, for the purposes of this assignment we are only interested in detecting the following errors:

Syntax errors. If at any point in the parsing you cannot find the next symbol that must be present you have detected a syntax error.
Declarations of more than one variable with the same name in the same context. That is, no two static or field variables in a class can have the same name and no two parameters or local variables in a subroutine can have the same name.
Attempting to use an undeclared variable. Not all such errors can be detected because in a subroutine call we cannot tell the difference between an undeclared variable and the name of another class.
Attempting to return a value from a void function or void method or an attempt to not return a value from a non void function or method or an attempt to return something other than **this** from a constructor.
A constructor, function or method that might not execute a return statement.
A constructor declared with a return type that is not its own class.

Errors to Ignore

The following semantic errors will be ignored and the parsing allowed to complete:
Attempts by a function to access a field of its class. This is a significant error that we will ignore. The program will still run on the Hack machine but it will be reading or writing the wrong memory locations.
Attempts to declare more than one constructor, function or method with the same name or to call a constructor, function or method that does not exist. Detecting errors in naming subroutines will be deferred to the assembler when the final VM code version of a program is translated into assembly language.
Attempts to apply operators, infix or unary, to values of the wrong types. This is a potentially significant error that we will ignore.
Attempts to return a value of a different type from the declared return type of a function or method. This is a potentially significant error that we will ignore.