Assignment 3 jack_parser()

jack_parser()

Description

You must complete the implementation of the **jparser** program in the file **parser.cpp**.

The program reads a Jack class from standard input and writes an XML representation of its abstract syntax tree to standard output. It uses the tokeniser functions described in **tokeniser.h** and **j-tok.h** to parse a Jack class read from standard input and construct an abstract syntax tree using the functions as described in **j-ast.h**. The main function is responsible for calling the **jack_parser()** function and passing the result to the function **jn_print_as_xml()**. The **jn_print_as_xml()** function is responsible for writing an XML representation of the abstract syntax tree to standard output.

Compiling and Running jack_parser

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 **jparser** and our own class **xyz** that we want to use with all of our programs, 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 jparser
```

The suite of provided tests can be run using the command:

```
% make test-jparser
```

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 copy-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-directories **bin**, **includes** or **lib**. These will be replaced during testing by the web submission system.

Tokeniser

The tokeniser described in **tokeniser.h** and **j-tok.h** returns Jack tokens. This table is based on Figure 10.5 from the textbook and shows the tokens that the tokeniser recognises.

Token	Token Kind		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 jk_method	jk_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 jk_lcb	jk_symbol	::=	'{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'
jk_integerConstant	jk_integerConstant	::=	('0'-'9')('0'-'9')*
jk_stringConstant	jk_stringConstant	::=	'"' A sequence of printable ASCII characters and whitespace characters not including double quote or newline '"'
jk_identifier	jk_identifier	::=	('a'-'z' | 'A'-'Z' | '_')('a'-'z' | 'A'-'Z' | '0'-'9' | '_')*
jk_eoi	jk_eoi	::=	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 integer constant is in the range 0 to 32767.
- The value of a string constant does not include the enclosing double quotes.
- A string constant 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 token jk_eoi. This token is not part of a legal Jack program.
- Once a jk_eoi token is returned, all future attempts to read a token will return jk_eoi.
- 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 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** and **j-tok.h**.
- You should use the symbol table functions described in **symbols.h**.
- There must be no output written to **cerr** or **cout**.
- All output must be written using the functions in **iobuffer.h**, remember to call print_output().
- If a parsing error occurs the program must immediately call ***exit(0)*** and have not produced any output.
- During testing you may output error messages and other log messages using the functions in **iobuffer.h**. The ***tokeniser_context()*** function will show the tokeniser's current position in the input being parsed.
- 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 a few exceptions. These exceptions must match those in the supplied test data.

There is no **program** node. Jack source files only contain a single class so the root node must be a **class**.

Static variables, field variables, parameters and local variables are all represented using **jn_var_dec** nodes. The order of these nodes must match the order in which their variables are declared.

There are no explicit **statement** nodes, everywhere a **statement** node may appear a node for the specific kind of statement is provided instead.

There are two **ifStatement** nodes, one with an else statement and one without.

There are two **return** nodes, one for a simple return and one for returning an expression.

There are no explicit **expression** nodes, everywhere an **expression** node may appear a node for the specific kind of expression is provided instead.

When parsing an expression, each time a new **infixOp** is found, a new **jn_infix** node is created. The left hand side of the new **jn_infix** node is the expression parsed so far and the right hand side is the next term to be parsed.

There are no **term** nodes implicit or explicit in the abstract syntax tree implementation.

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.

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, field, parameter or local variables can have the same name, no static variable can have the same name as a field variable and no parameter can have the same name as a local variable.

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.

Attempts by a function to access a field of its class.

Errors to Ignore

The following semantic errors will be ignored and the parsing allowed to complete:

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 except in the case of constructors.