JASON SPECIFICATION DOCUMENT

The JASON language is a very simple language initially constructed using Python.

Created by **Jason Hess, Omar Madjitov, Pratham Merchant, and Mary Verzi**.

https://github.com/JHess5-GSU/PLC-ExtraCredit

Contents

Keywords:

<JASON> <NOSAJ> IF ELSE WHILE FOR RETURN CLASS int real bool string char

Data Types:

int	32 bits	stores signed integer ranging from -21474836487 to 2147483647
real	32 bits	stores signed decimal ranging from ~ $\pm1.18\times10^{-38}$ to $\pm3.4\times10^{38}$
bool	1 bit	stores either 0 (False) or 1 (True)
string	Varies	Essentially an array of characters, undetermined length.
char	8 bits	Any single letter or underscore, can be empty.

Operators:

*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulo
( )	Prioritized

The Jason Language requires that every lexeme be separated by at least one whitespace character, and all statements must end with a semicolon.

Ex:

```
varname A ;

A = 1 + -2 ;
```

Only single-line comments are currently supported, and they must start with ##

Ex: ## This is a comment.

JASON requires all program files to start with <JASON> and end with <NOSAJ> tags.

Anything before and after the tags is treated like a comment and ultimately ignored by the analyzers.

GRAMMAR

NOTES:

Curly brackets '{' or '}' mean ONE OR MORE REPETITIONS

Square brackets '[' or ']' mean ZERO OR MORE REPETITIONS

Parentheses '(' or ') ' signify a group to pick one option from.

'\' is an escape character, so take the next symbol literally, but not \.

'|' serves to mean 'OR', allowing alternative options.

Content enclosed with '#' means to match the token, but use the lexeme.

Ex: #VARNAME# means the lexeme must be a valid VARNAME token.

PROG	<JASON> **STMTS** <NOSAJ>
STMTS	\{ { **STMT** ; } \}
STMT	**DEC | IF_STMT | WHILE_STMT | ASS | CLASS | METHOD**
DEC	**ID** #VARNAME#
ID	int | real | bool | char | string
IF_STMT	if \(**BOOL_EXPR** \) **STMTS** [else **STMTS**]
WHILE_STMT	while \(**BOOL_EXPR** \) **STMTS**
ASS	#VARNAME# = **EXPR**
EXPR	**TERM** [(+ | -) **TERM**] | **STRING_LIT | BOOL_LIT | ~~CHAR_LIT~~**
TERM	**FACTOR** [(* | / | %) **FACTOR**]
FACTOR	#INT# | #REAL# | \(**EXPR** \)
BOOL_EXPR	**BAND** [or **BAND**]
BAND	**BCOMP** [and **BCOMP**]
BCOMP	**EXPR** { (!= | == | >= | <= | < | >) **EXPR** } | **BOOL_LIT**

CLASS	class #VARNAME# **PROG**
METHOD	method #VARNAME# \(#VARNAME# [, #VARNAME#] \) **STMTS**
INT_LIT	**^(\d+)$**
REAL_LIT	**^(\d+.\d+)$**
~~**CHAR_LIT**~~	~~**^‘(a-zA-Z_)?’$**~~
STRING_LIT	**^“(a-zA-Z_)*((a-zA-Z_)*)*”$**
BOOL_LIT	True | False

PROG

Every program file must begin with “<JASON>” , end with “<NOSAJ>”, and include at least one statement. At least one space is required between lexemes.

Ex:

```
<JASON> { int test ; test = 0 ; } <NOSAJ>
```

ASS

Variable names must consist of 1-64 letters or underscores, no numbers can be included. Reserve word or keyword use will result in errors.

String

Strings in JASON start and end with a quote ("). Quotes are not allowed inside the string, they will lead to an early ending. Multiple consecutive spaces are also not allowed. Leading and ending spaces are not allowed. If a string is not completed by the end of the line, it is not valid and will result in an error.

```
Ex: string a ;
    a = " this is a string. "
```

LEXEMES AND TOKENS

As lexemes are required to be separated by at least one whitespace character, the lexer looks at each lexeme as a whole instead of character by character when attempting to create a token. This greatly simplifies parsing for me at a slight detriment to writability.

Comments are ignored and essentially tell the lexer to skip to the next line.

Text or *Regex*	Token	Notes
<JASON>	0	Start
<NOSAJ>	1	End
;	2	Semicolon
+	3	Plus
~~++~~	~~4~~	~~Increment~~
-	5	Minus
~~--~~	~~6~~	~~Decrement~~
*	7	Multiply
/	8	Divide
%	9	Modulo
<	10	Less Than
>	11	Greater Than
<=	12	Less Than/Equal To
>=	13	Greater Than/Equal To
=	14	Assignment
==	15	Equality
!=	16	Not Equal To
and	17	Boolean “and”
or	18	Boolean “or”
(	19	Left Parenthesis
)	20	Right Parenthesis
if	21	if Statement Keyword
for	22	for loop Keyword
while	23	while loop Keyword
else	24	else keyword
{	25	Left Curly Brace
}	26	Right Curly Brace
char	27	Character Data type identifier
int	28	Integer Data type identifier
real	29	Real number Data type identifier
bool	30	Bool data type identifier
string	31	String Data type identifier
^(\d+)$	32	Any number of digits (int)
^(\d+.\d+)$	33	Any number of digits, with a decimal point. (real)
^(([A-Za-z]*[_]*)+)$	34	One or more letters or underscores. (varname)
" #STRING# "	35	String token, see the STRING_LIT or String info.
,	36	Parameter separator
True | False	37	Boolean literal, either “True” or “False”

class	38	Start of class
method	39	Start of method

LANGUAGE DESIGN TRADEOFFS

The lexer requires that lexemes be separated by a space or any number of whitespace characters. This is to make it easier to iterate through lexemes. The consequence of this is that literals such as strings cannot contain multiple spaces, as that information is currently lost during lexing. However, it can be implied by separated lexemes that there was at least one whitespace character between them, so that could be recovered.

Variable names must consist of 1-64 letters or underscores. No numbers can be included. Reserve word or keyword use will result in errors. This is to ensure consistency in the variable names throughout the program and makes them easy for the lexer to identify. A drawback to this choice would be not being able to include numbers, which would increase readability and specificity.

Strings must begin and end with quotation marks. This makes strings in the program easy to identify, increasing writability and readability. Strings are not allowed to have multiple consecutive spaces, a leading or ending space, or quotation marks inside of the string. These requirements, unfortunately, limit what the coder can include in a string.

Single-line comments are allowed, but multi-line comments are not supported. Having just one type of comment makes them easy to identify and increases writability and readability. The lexer will always know to skip just one line when a comment is identified. However, not being able to write multi-line comments potentially takes away from the readability of long comments and comments describing methods.

Statements must end with a semicolon. This increases the readability of the program because a semicolon will always indicate a statement. However, requiring a semicolon after each statement can potentially decrease writability, as it is easy to forget to add one or add one where it is not allowed.

Methods are named using variable names and contain variables that are included in the method followed by statements. The escape character helps with readability of each method made. The variable that is taken in can have zero or more repetitions.

HOW TO RUN THE TOKENIZER ONLY

The tokenizer (toker.py or the first half of lexer.py) takes in a .txt (or .jason) file and, if it is a correct file, it will output a list of tokens. If there are errors, it will describe the errors by line number and potentially by lexeme.

Code to run example:

```
Tokenize('ifTest.txt').run('ifTestOutput.txt')
```

Sample good input file (ifTest.txt):

```
<JASON>
{
    if ( True ) {
        varA = 10 ;
    } else {
        varB = 10.10 ;
    }
}
<NOSAJ>
```

Sample good output file (ifTestOutput.txt):

```
['<JASON>', '0']
['{', '25']
['if', '21']
['(', '19']
['True', '34']
[')', '20']
['{', '25']
['varA', '34']
['=', '14']
['10', '32']
[';', '2']
['}', '26']
['else', '24']
['{', '25']
['varB', '34']
['=', '14']
['10.10', '33']
[';', '2']
['}', '26']
['}', '26']
['<NOSAJ>', '1']
```

HOW TO RUN THE SYNTAX ANALYZER ONLY

Unfortunately the syntax analyzer is not complete, but it may work in some limited cases.

It is run using `ParseTokens("testtokens1.txt").run()`