Floyd Syntax Specification

The Floyd language is a block-structured object-oriented language. This document describes the Floyd grammar using extended BNF notation. Non-quoted square brackets [ ] indicate optional phrases and curly braces { } indicate zero, one, or more repetitions of a phrase.

Terminals appear in **`bold courier`** (for keywords and other leximes who do not need supplementary information) and *italic arial* (for leximes such as *id* which involve supplementary information); nonterminals appear in *<italic arial>* with brackets around them.

When <cr> appears, it indicates the presence of one or more newlines.

```
<start>               ::= [ <cr> ] <class> { <cr> <class> } [ <cr> ]
<class>               ::= class id [ inherits from id ] is <cr>
                          { <var_decl> }
                          { <method_decl> }
                          end id


<var_decl>            ::= id [ : <type> ] [ := <expression> ] <cr>
<method_decl>         ::= id ( [ <argument decl list> ] ) [ : <type> ] is <cr>
                          {  <var_decl> }
                          begin <cr>
                          <statement list>
                          end id <cr>
<argument decl list>  ::= { <argument decl> ; } <argument decl>
<argument decl>       ::= id : <type>
<type>                ::= int | string | boolean
                        | id
                        | <type> '[' [ <expression> ] ']'
<statement list>      ::= { <statement> <cr> }
<statement>           ::= <assignment stmt>
                        | <if stmt>
                        | <loop stmt>
                        | <call stmt>


<assignment stmt>     ::= id { '[' <expression> ']' } := <expression>
```

```
<if stmt>            ::= if <expression> then <cr>
                         <statement list>
                         [ else <cr> <statement list> ]
                         end if


<loop stmt>          ::= loop while <expression> <cr>
                         <statement list>
                         end loop


<call stmt>          ::= [<expression> . ] id ( [ <expression list> ] )


<expression list>    ::= { <expression> , } <expression>

<expression>         ::= id | string_literal | int_literal | true | false | null | me
                       | new <type>
                       | <expression> binary_op <expression>
                       | unary_op <expression>
                       | ( <expression> )
                       | [<expression> . ] id ( [ <expression list> ] )
                       | id '[' <expression> ']' { '[' <expression> ']' }
```

Note that the binary operations are all left associative, except for the relational operators which do not "associate," i.e., x = x >= x is syntactically illegal. Also note the following operator precedence chart (highest precedence listed first):

., new	method call, new
-, +, not	unary operators
*, /	multiplication, division
+, -	addition, subtraction
&	string concatenation
=, >, >=	relational operators
and	logical and
or	conditional or

- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two operators of equal precedence is bound to the one on its left (if the operator is left associative).