1 Lexical Grammar

```
// Whitespace -- ignored
" " | "\t" | "\n" | "\r" | "\f"

// Single line comment -- ignored
"//" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )

// Reserved words
BOOLEAN         ::= "boolean"
ELSE            ::= "else"
FALSE           ::= "false"
IF              ::= "if"
INT             ::= "int"
RETURN          ::= "return"
TRUE            ::= "true"
VOID            ::= "void"
WHILE           ::= "while"

// Separators
COMMA           ::= ","
LCURLY          ::= "{"
LPAREN          ::= "("
RCURLY          ::= "}"
RPAREN          ::= ")"
SEMI            ::= ";"

// Operators
ASSIGN          ::= "="
DIV             ::= "/"
EQUAL           ::= "=="
GE              ::= ">="
GT              ::= ">"
LAND            ::= "&&"
LE              ::= "<="
LT              ::= "<"
LNOT            ::= "!"
LOR             ::= "||"
MINUS           ::= "-"
NOT_EQUAL       ::= "!="
PLUS            ::= "+"
REM             ::= "%"
STAR            ::= "*"

// Identifiers
IDENTIFIER      ::= ( "a"..."z" | "A"..."Z" | "_" | "$" )
                      { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }

// Literals
BOOLEAN_LITERAL ::= "false" | "true"
INT_LITERAL     ::= ( "0"..."9" ) { "0"..."9" }

// End of file
EOF             ::= "<end of file>"
```

2 Syntactic Grammar

```
compilationUnit ::= { methodDeclaration } EOF

methodDeclaration ::= ( VOID | type ) IDENTIFIER formalParameters block

formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN

formalParameter ::= type IDENTIFIER

block ::= LCURLY { statement } RCURLY

statement ::= block
            | type IDENTIFIER [ ASSIGN expression ] SEMI
            | IF parExpression statement [ ELSE statement ]
            | RETURN [ expression ] SEMI
            | WHILE parExpression statement
            | statementExpression SEMI

parExpression ::= LPAREN expression RPAREN

type ::= BOOLEAN | INT

statementExpression ::= expression

expression ::= assignmentExpression

assignmentExpression ::= conditionalOrExpression [ ASSIGN assignmentExpression ]

conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }

conditionalAndExpression ::= equalityExpression { LAND equalityExpression }

equalityExpression ::= relationalExpression { ( EQUAL | NOT_EQUAL ) relationalExpression }

relationalExpression ::= additiveExpression [ ( GE | GT | LE | LT ) additiveExpression ]

additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }

multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }

unaryExpression ::= ( LNOT | MINUS ) unaryExpression
                  | parExpression
                  | IDENTIFIER [ arguments ]
                  | literal

arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN

literal ::= BOOLEAN_LITERAL | INT_LITERAL
```

The method `void main() { ... }` serves as the entry-point method, ie, is invoked when the program is run. If this method is not explicitly defined in the compilation unit, an implicit entry-point method with an empty body is provided.

3 Semantics

```
IBinaryExpression:
- IDivideOp, IMultiplyOf, IPlusOp, IRemainderOp, ISubtractOp
  - lhs and rhs must be integers
- IAssignOp:
  - lhs must be a variable
  - lhs and rhs must have the same type

IBooleanBinaryExpression:
- IEqualOp, INotEqualOp:
  - lhs and rhs must have the same type
- ILogicalAndOp, ILogicalOrOp:
  - lhs and rhs must be booleans

IComparisonExpression:
- lhs and rhs must be integers

IMessageExpression:
- The message must correspond to a valid method

IMethodDeclaration:
- Must not be defined already
- Non-void method must have a return statement

IIfStatement:
- The condition must be a boolean

IUnaryExpression:
- ILogicalNotOp
  - The operand must be a boolean
- INegateOp
  - The operand must be an integer

IVariable:
- The variable ust be declared
- The variable must be initialized

IVariableDeclaration:
- The variable must not shadow another local variable

IReturnStatement:
- Must not return a value from a void method
- The type of return value in a non-void method must match return type of the method
- A non-void method must return a value

IWhileStatement:
- The condition must be a boolean
```

4 Input and Output

The *iota* language supports the following builtin methods for reading from standard input and writing to standard output:

```
// Reads and returns an int from standard input.
int read()

// Writes x (1 for true and 0 for false) to standard output.
void write(boolean x)

// Writes x to standard output.
void write(int x)
```