8-4-2010

A Language Designed for Programming I

Ben A. Blake
Cleveland State University, benblake@csuohio.edu

Publisher's Statement

The final publication is available at Springer via http://dx.doi.org/10.1007/s10639-010-9139-3.

Original Published Citation

Blake, B. (2010). A Language Designed for Programming I. Education and Information Technologies, 15, pp. 277-291.

BLAKE: A language designed for Programming I

Ben Blake

Abstract The process of comprehending a problem, strategically developing a solution and translating the solution into an algorithm is arguably the single most important series of skills acquired during the education of an undergraduate computer science or information technology major. With this in mind, much care should be taken when choosing a programming language to deploy in the first University programming course. BLAKE, Beginners Language for Acquiring Key programming Essentials, is designed specifically for use in a Programming I class. BLAKE aids in enforcing fundamental object-oriented practices while simultaneously facilitating the transition to subsequent programming languages. BLAKE's major features include; consistent parameter passing, single inheritance, non-redundant control structures, a simple development environment, and hardware independent data types. The syntax remains relatively small while still facilitating a straightforward transition to industry standard programming languages.

Keywords Programming language · Object oriented · Curriculum · Syntax · Grammar

1 Introduction

In the 1960's and 1970's many computing curricula used languages such as Fortran, Cobol, PL/I, Algol or BASIC in their initial programming course. Each of these languages was developed during the emergence of software engineering. Clearly the designers of these languages had little chance to incorporate language specific syntactical features that encourage good programming practices precisely because software engineering was in its infancy and these methodologies were either being fashioned or were yet to be discovered. Likewise, consideration of the language's effect on a beginning programmer had an insignificant influence on the pioneer language designers. Instead, the designers focused mainly on architecture specific issues, run-time efficiency and the compilation process with little to no consideration

B. Blake (✉)
Cleveland State University, 1860 E. 18th, Cleveland, OH 44114, USA
e-mail: benblake@csuohio.edu

on the pedagogical effects of the syntax. Consequentially, curricula using these languages in their first programming courses have vanished (Raadt et al. 2002). Throughout the remainder of the paper, the first University programming course will be termed *Programming I* as it is commonly referred to in the United States.

An eloquent editorial by Dykstra regarding the GOTO statement appeared that radically challenged programming practices. (Dijkstra 1968) This short critique coupled with the emergence of structured programming techniques highlighted the inherent gap between grammars of early programming languages and good programming practices. When Nicholas Wirth released the Pascal Programming Language (Wirth 1971) a great number of curricula switched to using Pascal. By the early 1980's, a majority of universities employed Pascal in Programming I courses (Raadt et al. 2002). This should be little surprise since one of Wirth's design goals was to create a language that encouraged structured programming. Pascal includes a fairly simple grammar with logical structures to aid in the conversion of structured designs into structured programs. Additionally, Pascal's small syntax allowed a subset of it to become an effective language for compiler design courses. With these competitive advantages, Pascal became the choice language used by the educational community in the 1980's.

Since then, the number of schools using Pascal in Programming I has steadily declined to near zero (Raadt et al. 2002). The reason for Pascal's fall in popularity relates to three major factors. First, a big selling point of Pascal was that it allowed and actually helped to reinforce structured design practices. This was mitigated when the C programming language, with a similar grammar, gained industrial popularity. The C programming language could easily replace Pascal in the curriculum. Having a viable substitute language would not by itself cause the mass exodus from Pascal. The second factor which led to Pascal's demise in the classroom use was that both students and employers exerted pressure on educational institutions to teach a language widely deployed in industry. If these two factors weren't great enough, the third strike appeared in the 1990's when the Object-Oriented paradigm became the rage. The Pascal programming language lacks syntactic support for Object-Oriented methodologies which other new languages include. These factors contribute to the fall of Pascal and the rise of other languages in Programming I classes.

Recent trends show the emergence of C++, C#, Visual Basic, Java, or Python in the introduction to Programming classes. A majority of schools surveyed use one of these languages (Raadt et al. 2002). Each language has both control structures and subroutine call/return mechanisms strikingly similar to Pascal and C. The main difference being that neither Pascal nor C provides language constructs that specifically encourage the use of object-oriented methodologies.

Alternatively, a few schools use languages such as Haskell, Eiffel, Common Lisp, and Scheme in their initial series of programming courses (Hudak and Fasel 1992; Springer and Friedman 1989). These languages fall under the broad category of functional programming languages. Most proponents of functional languages site the extremely small syntax as an important characteristic for a language used in Programming I. The small syntax requires little effort to mastering, so more time can be assigned to problem solving efforts. While students benefit in Programming I using these languages, the transition from these languages to a widely accepted

industrial language such as C, C++ or Java often becomes problematic (McIver and Conway 1996; Brilliant and Wiseman 1996).

Attempts have been made to measure the fit between Programming I course objectives and the programming language being used (Kolling 1999; Gupta 2004). This research involves generating a list of criteria then applying the criteria to various programming languages (Kolling 1995; Parker et al. 2006; Mannila and de Raadt 2006). These systems provide useful information, but the criteria used to gauge the language remains a contentious issue. Even after reaching consensus of the proper criteria, the relative weight of each would require further research. Given these concerns, BLAKE was not developed for optimal measurement from any of these systems, but evaluates relatively well with all of the proposed systems.

BLAKE is created as an attempt to define a language specifically for use in a Programming I course. Appendix A highlights BLAKE's compact grammar. The syntax is designed in such a manner to allow a fairly trivial transition to mainstream languages such as C, Objective C, C++, C#, Java, and Visual Basic. BLAKE allows the Programming I course to remain focused on the problem solving process by removing nonessential topics currently covered in many Programming I courses.

The fundamental driving force behind BLAKE involves the creation of a language that allows both the instructor and student to acutely focus on problem solving and translating the solution into code. The language would allow the Programming I student to fixate on these processes while avoiding the introduction of many nonessential topics. Programming languages, such as C, C++, Java, and Visual Basic, currently used in Programming I courses expose students to many extraneous items causing an unnecessary distraction from the fundamental concepts. While astute instructors often minimize discussion of these topics, they still represent needless distractions. The grammatical design of BLAKE intentionally allows the complete omission of these extra topics in a Programming I course. BLAKE allows an increase in time devoted to designing solutions and translating the solutions into code.

The next section introduces many of BLAKE's syntactical design decisions and describes the impacts these decisions have on topics covered in many Programming I classes. The discussion includes subjects that may be safely removed from Programming I and a suggestion indicating a more appropriate placement of the concept within a computer science curriculum. Some of the decisions guiding the design of BLAKE stem directly from prior research, while anecdotal beliefs guide the balance of the design decisions. Following the syntax discussion of BLAKE is a section on other generic pitfalls that Programming I can avoid. Conclusions are followed by appendices consisting of the BLAKE grammar, a sample program, a sample Object, and a list of methods for each of BLAKE's base types.

2 General language syntax

2.1 Control structures syntax

Often a programming language redundantly specifies multiple control structures to accomplish the same task. For instance, C, C++, and Java include the *while*, *do*

while, and *for* statements to specify repetitive logic. These languages also offer both the *if* and the *switch* structures for performing conditional logic. BLAKE offers only one looping and one conditional control structure. The removal of redundant structures simplifies students' task in two ways. First, having only one construct for each type of logic unencumbers students from memorizing additional syntax. Second, it frees the student from agonizing over the question of, "Which construct should I use?" Clearly, the topic of multiple redundant control structure will arise later in the curriculum with the introduction of another programming language. Students at this upper level generally embrace the redundancy and are no longer needlessly distracted by it.

2.2 Optional syntax

Many languages offer the programmer the ability to skip the *begin* and *end* constructs associated with programming structures when only one statement exists within the logic. This allows the programmer to hit fewer keystrokes, but does not add anything to program readability nor enhance maintainability. BLAKE requires a begin and end statement with each control structure. Likewise an else clause must accompany every if statement. Finally, conditions appear with all clauses of conditional and looping logic. The optional else clause and other terse grammatical options can be discussed with the introduction of the second programming language.

2.3 Multiple loop exit points

Some programming languages allow advanced programmers the opportunity to exit and restart loops with conditional logic inside the loop. This corresponds directly to the *break* and *continue* statements in the C, C++ and Java. With no empirical evidence available, anecdotal opinion suggests it is preferable to avoid multiple exit points. Since Programming I assignments remain fairly simple, BLAKE allows only a single loop termination point. Statements such as these and the GOTO statement should be broached when transferring to a programming language that includes such statements. A class discussing assembly language would be a very appropriate place to introduce the concept of the GOTO statement.

2.4 Additional debugging syntax

BLAKE's philosophy banishes exception handling and assertion syntax. BLAKE acknowledges that there can be a case made for the usefulness of these constructs when debugging large programs, for the relatively small amount of code typically produced in Programming I, they become excessive. The fragile Programming I student can safely be spared of the mechanical syntax details while still being exposed to the underlying concepts. Debugging tools like the try/catch and assert statements can be introduced in a software quality assurance course.

The BLAKE viewpoint encourages programmers to explicitly verify all data before calling a routine. If followed strictly, this style of programming ensures that exceptions will not be raised and all assertions will remain valid. While teaching the

concept behind assertions and exceptions remains important, syntactically adding it to the programming language becomes extraneous. The inclusion of exceptions and/ or assertions in a programming language grammar would better be discussed in either a software engineering course or a comparative programming languages course.

2.5 Encoding systems and primitive types

Many languages include hardware dependent data types. Introducing these data types normally involves a modest discussion of computer architecture and the binary number system. While mastery of these concepts is essential for all CS and IS undergraduates, it adds little to a student's ability to solve problems and translate the solution into code. The discussion of encoding and the decision to make primitive data types specific sizes more appropriately belongs in a computer organization and architecture course.

BLAKE provides just five types, namely, IntegerNumber, RationalNumber, CharacterString, BooleanValue, and BasicObject. Two of these, IntegerNumber and RationalNumber represent numeric types. An IntegerNumber holds integer value, whereas a RationalNumber consists of two IntegerNumbers. These two values in a RationalNumber are the number's numerator and denominator. BLAKE provides no details regarding storage of these types which allows for total omission of discussing binary number system.

Along with avoiding the binary number system, the two numeric types largely avoid issues of underflow and overflow. The discussion of these topics can be limited to stating that each machine has a limited storage capacity and an improperly written program can exceed the capacity. This eliminates the need to discuss bits and bytes. The implementation of RationalNumber also avoids round-off errors; in fact, round-off errors do not exist in BLAKE. The programmers introduce and control any round-off error.

There is a single string type in BLAKE. CharacterString stores any number of characters solely limited by the computer system's memory size and memory allocation scheme. This avoids the confusion of the difference between a character and a string with a single character in it. Just like the discussion of the encoding of numbers can be avoided, so can the discussion of character encoding.

The final two types are BooleanValue and BasicObject. BooleanValue contains the value true or false. BasicObject represents the base object from which every other object inherits. It consists of two methods, CONSTANTCOPY and CONSTANTtoCharacterString. When creating a new object definition, BLAKE expects the programmer to override these methods. CONSTANTCOPY makes a copy of the object and CONSTANTtoCharacterString returns a string representation of the object.

The physical description of the number of bits and bytes used to store the different data types is a better fit for a computer architecture class. The number of bits/bytes in the various types was not decided by programmers, but instead by the machine designers. While this topic fits perfectly into an architecture class, it may require a brief introduction in a Programming II or Programming III course whenever a more architecturally restricted language is initially covered.

2.6 No assignment statement or operators

The same argument used against primitive types also holds for operators. This includes the assignment operator. The discussion of operators, including the assignment operator, need not be discussed in Programming I. Likewise, the reason why operators exist in most widely used languages would optimally be introduced in an architecture class. While this is theoretically the correct curricular placement of operators, in practice they would be introduced with the second programming language.

2.7 No arrays

BLAKE provides no subscripting mechanism and a list is not included in the base types. While this appears extremely restrictive, an instructor may provide the students with either a list or an array like object. Others may wish to use this as an introduction to the data structures class and have the students build their own list object(s). Once again, the concept of an array is more aligned to a data structure or computer architecture class.

3 Object syntax

3.1 Parameter passing and aliasing

A little after Dykstra published an article on the GOTO statement, it seemed as if everyone jumped on the "anti-spaghetti code band wagon" (Dijkstra 1968). Interestingly, another just as important concept that surfaced around the same time remains largely ignored. The uncontrolled aliasing of a single variable exposes a weakness that has remained relatively anonymous. Just as imprudent use of the GOTO creates "spaghetti code" or logic that is difficult to comprehend, abundant aliasing jumbles the mapping of variable names to data. Many programming languages unintentionally allow incorporating such rampant aliasing.

When passing information to a method students become easily confused with the "by reference" and the "by value" passing mechanics and syntax. In Visual Basic, the manner in which an argument is passed is specified in the method parameter list. While the valid claim of Java is everything is passed by value, most experienced programmers argue that this claim is merely a technicality. C assumes the method and the method caller will agree on the passing method. C++ allows both Java like and C like passing. In the end, none are particularly helpful in the beginning student's understanding of the two types of parameter passing.

BLAKE allows both by value and by reference parameter passing. It differs from most common languages because the caller of the method dictates what is passed. In the method call, the calling code specifies whether to pass a copy or an alias for each argument. The overall design of BLAKE encourages passing copies and only in rare instances should a Programming I student be instructed to pass an alias. This philosophy assists the student in understanding the concept of side effects, including unexpected side effects. From the method's perspective, the parameters simply become initialized local variable and the method freely alters their values as needed.

3.2 No access modifiers and no abstract methods

BLAKE has two implied access modifiers, public and private. All data within an object receive the private access modifier and the methods are assigned public. The automatic designation postpones the need to discuss these distinctions and meshes perfectly with correct encapsulation. All other access modifiers such as protected, shared, static, or friend, are simply performance enhancements. These topics also are better suited for a compiler design or a comparative programming language course, but will likely be introduced as needed in later classes that incorporate other programming languages.

3.3 Constant and mutator methods

Most languages do not distinguish between a method that changes the state of an object and a method that does not change the state. BLAKE requires the programmer of an object to distinguish between the two types of methods. The method caller also explicitly states the distinction. This ensures that both the object designer and object creator understand if a method can or cannot change the state of the Object. Mutator methods have no return type and constant methods have a single return type.

3.4 Delineation between an object and a program

Many object-oriented languages blur the distinction between a program and an object. Java, in particular, forces a programmer to place a program inside of syntax precisely the same as defining an object. Many "Object Oriented" languages allow a programmer to define multiple objects in a single file. In a BLAKE file, there is precisely either a one program or one object definition with syntax clearly identifying the difference.

4 Other items

4.1 Development environment

Since the goal of BLAKE is to allow students to focus acutely on programming and problem solving, there are also some other issues commonly encountered in a Programming I class that we advocate avoiding. The most problematic of these might be something an advanced programmer couldn't imagine relinquishing, an integrated program development environment. While incredibly helpful for the well versed students, the complicated software development environments often represent a huge distraction for a beginning programmer (Deek and McHugh 1998). "Microsoft Visual Studio" represents a prime example of such a system. BLAKE offers no development environment. Programmers simply use a basic text editor and a command line compiler. There is no graphical package, no debugger, and no sophisticated development packages. This creates an extremely short learning curve for the Programming I student.

4.2 Early inheritance

The base classes IntegerNumber, RationalNumber, and CharacterString include little functionality. This intentional design decision facilitates the early introduction of inheritance with non-contrived examples. For instance, fairly common functionality such as absolute value can be added to a class such as IntegerNumber. This forces useful inheritance early to instill the necessity and beauty without creating contrived examples. Many methods, both simple and complex can be added to the base classes.

4.3 No macros or constants

A quick glance at the syntax shows BLAKE provides no macro expansions or constants. BLAKE opts away from exposing Programming I students to unnecessary syntactic details. Similarly, no default values are assigned to variables and no constructor methods appear in objects. BLAKE requires programmers to explicitly state all their intentions.

4.4 Input and output

Finally, all interactions between BLAKE and users are through CharacterStrings. BLAKE disallows programmers to type numbers in their programs. Instead BLAKE requires the programmer to type the CharacterString "12" and assign that CharacterString to an IntegerNumber. Likewise all input is received as a Character-String one line at a time. This includes input from both the standard input stream and a file. At most, two files may be opened at any time—one for input and one for output. Input files are exactly like the standard input stream with the additional feature of and end of file method. The standard output stream and file output have two commands each. One method outputs a CharacterString without a trialing end of line and the other outputs the CharacterString with the trailing end of line.

5 Conclusions

BLAKE represents a language specifically designed for the initial programming class in a computing curriculum. It attempts to delay the introduction of topics that divert attention from the process of translating a problem solution into code. To achieve this goal BLAKE's grammar remains small through elimination of redundant control, optional clauses, multiple loop exit points and debugging syntax. BLAKE avoids architectural issues such as encoding schemes, operators, and arrays while still providing an easy transition to traditional languages commonly used in industry. The concept of objects and especially inheritance naturally flow from the limited functionality of the base types.

While it is specifically designed for use in Programming I, BLAKE could also be used in Programming II. BLAKE also serves as an ideal language for discussion in a Programming Languages course and as a target in a compiler/interpreter design course.

Appendix A

program->	**USES** *idlist ;* **BEGINPROGRAM** *singleid ;* *declarationlist* *statementlist* **ENDPROGRAM** *singleid ;*
objectdefinition->	**USES** *idlist ;* **BEGINOBJECT** *singleid* **INHERITSFROM** *idlist ;* *declarationlist* *mutatormethodlist* *methodlist* **ENDOBJECT** *singleid ;*
idlist->	*singleid* | *singleid , idlist* | *epsilon*
singleid->	**['A-Za-z']** *idrest*
idrest->	**['A-Za-z']** *idrest* | *epsilon*
declarationlist->	*singleid singleid*.**CREATE ( ) ;** *declarationlist* | *singleid singleid*.**EMPTY ( ) ;** *declarationlist* | *epsilon*
statementlist->	*statement ; statementlist* | *epsilon*
returnlist->	*rstatement ; returnlist* | *epsilon*
statement->	*mutatormethodcall* | *ifstatement* | *untilstatement* | *outputstatement* | *errorstatement*
rstatement->	*mutatormethodcall* | *rifstatement* | *runtilstatement* | *outputstatement* | *filestatement* | *returnstatement* | *errorstatement*
errorstatement->	**ERROR** (*qoute*)
outputstatement->	**OUTPUT** (*quoteidcall*) | **OUTPUTLINE** (*quoteidcall*)

filestatement-> `FILEOUTPUT` (*quoteidcall*) |
`FILEOUTPUTLINE` (*quoteidcall*) |
`FILEOPENOUTPUT` (*quoteidcall*) |
`FILEOPENINPUT` (*quoteidcall*) |
`FILECLOSEOUTPUT` ( ) |
`FILECLOSEINPUT` ( )

returnstatement-> `RETURN` (*quoteidcall*)

untilstatement-> `BEGINLOOPUNTIL` (*condition*) ;
statementlist
`ENDLOOPUNTIL` (*condition*)

ifstatement-> `BEGINIF` (*condition*) ;
statementlist
`ELSE` (*condition*) ;
statementlist
`ENDIF` (*condition*)

runtilstatement-> `BEGINLOOPUNTIL` (*condition*) ;
rstatementlist
`ENDLOOPUNTIL` (*condition*)

rifstatement-> `BEGINIF` (*condition*) ;
rstatementlist
`ELSE` (*condition*) ;
rstatementlist
`ENDIF` (*condition*)

quote-> "*anychars* "

anychars-> **['any character']** *anychars* |
epsilon

mutatormethodcall-> *singleid* . *mutatorcall* |
`PARENT` . *mutatorcall* |
`ME` . *mutatorcall* |
`ME` . *singleid* . *mutatorcall*

methodcall-> *singleid* . *call* |
`PARENT` . *call* |
`ME` . *call* |
`ME` . *singleid* . *call*

quoteidcalllist-> *quoteidcall* |
qouteidcall , *quoteidcalllist* |
epsilon

condition-> *callid* |
methodcall

quoteidcall-> *quote* |
callid |
methodcall |
`INPUTLINE` |
`FILEINPUTLINE` |
`FILEEOF` |
`NOTHING`

callid-> *singleid* `.` `ALIAS` `(` `)` |
singleid `.` `COPY` `(` `)` |
`ME` `.` *singleid* `.` `ALIAS` `(` `)` |
`ME` `.` *singleid* `.` `COPY` `(` `)` |
`ME` `.` `ALIAS` `(` `)` |
`ME` `.` `COPY` `(` `)` |
`PARENT` `.` `COPY` `(` `)`

mutatorcall-> `MUTATOR`*singleid* `(` *quoteidcalllist* `)` |
`ALIAS` `(`*quoteidcall* `)` |
`CREATE` `(` `)` |
`EMPTY` `(` `)`

call-> `CONSTANT`*singleid* `(` *quoteidcalllist* `)` |
`ISEMPTY` `(` `)`

parameterlist-> *singleid singleid* `,` *parameterlist* |
singleid singleid |
epsilon

mutatormethodlist-> *mutatormethod* `;` *mutatormethodlist* |
epsilon

methodlist-> *method* `;` *methodlist* |
epsilon

method-> `BEGINCONSTANTMETHOD` *returnid singleid* `(` *parameterlist* `)` `;`
declarationlist
returnlist
`ENDCONSTANTMETHOD` *singleid* `;`

mutatormethod-> `BEGINMUTATORMETHOD` *singleid* `(` *parameterlist* `)` `;`
declarationlist
statementlist
`ENDMUTATORMETHOD` *singleid* `;`

Reserved words : USES, BEGINPROGRAM, ENDPROGRAM, BEGINOBJECT, ENDOBJECT, INHERITSFROM, BEGINLOOPUNTIL, ENDLOOPUNTIL, BEGINIF, ELSE, ENDIF, BEGINCONSTANTMETHOD, ENDCONSTANTMETHOD, BEGINMUTATORMETHOD, ENDMUTATORMETHOD, RETURN, NOTHING, TRUE, FALSE, INPUTLINE, OUTPUT, OUTPUTLINE, FILEOUTPUT, FILEOUTPUTLINE, FILENPUTLINE, FILEOPENINPUT, FILEOPENOUTPUT, FILECLOSEINPUT, FILECLOSEOUTPUT, FILEEOF, ME, PARENT, CREATE, EMPTY, ISEMPTY, ALIAS, COPY, ERROR (and all java reserved words)

Appendix B

```
USES CharacterString,
     IntegerNumber;

BEGINPROGRAM IfProgram;
  CharacterString inputString.CREATE();
  IntegerNumber firstValue.CREATE();
  IntegerNumber secondValue.CREATE();

  OUTPUT("Enter first value : ");
  inputString.MUTATORset(INPUTLINE);
  firstValue.MUTATORsetCharacterString(inputString.ALIAS());
  OUTPUT("Enter second value : ");
  inputString.MUTATORset(INPUTLINE);
  secondValue.MUTATORsetCharacterString(inputString.ALIAS());

  OUTPUTLINE("");
  OUTPUT(firstValue.ALIAS());
  BEGINIF (firstValue.CONSTANTlessThan(secondValue.CONSTANTCOPY()));
    OUTPUT(" < ");
  ELSE (firstValue.CONSTANTlessThan(secondValue.CONSTANTCOPY()));
    OUTPUT(" >= ");
  ENDIF (firstValue.CONSTANTlessThan(secondValue.CONSTANTCOPY()));
  OUTPUT(secondValue.ALIAS());
  OUTPUTLINE("");

  inputString.EMPTY();
  firstValue.EMPTY();
  secondValue.EMPTY();
ENDPROGRAM IfProgram;


USES RationalNumber;

BEGINOBJECT RationalNumberAbsolute INHERITSFROM RationalNumber;
  BEGINMUTATORMETHOD absoluteValue();
    RationalNumber zero.CREATE();
    zero.MUTATORsetCharacterString("0");
    BEGINIF (PARENT.CONSTANTlessThan(zero.CONSTANTCOPY()));
      PARENT.MUTATORset(zero.CONSTANTsubtract(ME.CONSTANTCOPY()));
    ELSE (PARENT.CONSTANTlessThan(zero.CONSTANTCOPY()));
    ENDIF (PARENT.CONSTANTlessThan(zero.CONSTANTCOPY()));
    zero.EMPTY();
  ENDMUTATORMETHOD absoluteValue;


  BEGINCONSTANTMETHOD RationalNumberAbsolute COPY();
    RationalNumberAbsolute ret.CREATE();
    RationalNumberAbsolute.MUTATORset(PARENT.COPY());
    return (RationalNumberAbsolute.ALIAS());
  ENDCONSTANTMETHOD RationalNumberAbsolute COPY();

ENDOBJECT RationalNumberAbsolute;
```

Appendix C

INPUT and OUTPUT methods

CharacterString	INPUTLINE()
	OUTPUT(CharacterString)
	OUTPUTLINE(CharacterString)

BooleanValue methods

BooleanValue	CONSTANTand(BooleanValue)
BooleanValue	CONSTANTor(BooleanValue)
BooleanValue	CONSTANTnot()

RationalNumber methods

RationalNumber	CONSTANTadd(RationalNumber)
RationalNumber	CONSTANTaddCharacterString(CharacterString)
RationalNumber	CONSTANTaddIntegerNumber(IntegerNumber)
RationalNumber	CONSTANTCOPY()
RationalNumber	CONSTANTdivideBy(RationalNumber)
RationalNumber	CONSTANTdivideByCharacterString(CharacterString)
RationalNumber	CONSTANTdivideByIntegerNumber(IntegerNumber)
BooleanValue	CONSTANTequals(RationalNumber)
BooleanValue	CONSTANTequalsCharacterString(CharacterString)
BooleanValue	CONSTANTequalsIntegerNumber(IntegerNumber)
BooleanValue	CONSTANTgreaterThan(RationalNumber)
BooleanValue	CONSTANTgreaterThanCharacterString(CharacterString)
BooleanValue	CONSTANTgreaterThanIntegerNumber(IntegerNumber)
BooleanValue	CONSTANTlessThan(RationalNumber)
BooleanValue	CONSTANTlessThanCharacterString(CharacterString)
BooleanValue	CONSTANTlessThanIntegerNumber(IntegerNumber)
RationalNumber	CONSTANTmultiplyBy(RationalNumber)
RationalNumber	CONSTANTmultiplyByCharacterString(CharacterString)
RationalNumber	CONSTANTmultiplyByIntegerNumber(IntegerNumber)
RationalNumber	CONSTANTsubtract(RationalNumber)
RationalNumber	CONSTANTsubtractCharacterString(CharacterString)
RationalNumber	CONSTANTsubtractIntegerNumber(IntegerNumber)
CharacterString	CONSTANTtoCharacterString()
IntegerNumber	CONSTANTtoIntegerNumber()
	MUTATORset(RationalNumber)
	MUTATORsetCharacterString(CharacterString)
	MUTATORsetDenominator(IntegerNumber)
	MUTATORsetDenominatorCharacterString(CharacterString)
	MUTATORsetIntegerNumber(IntegerNumber)

	MUTATORsetNumerator(IntegerNumber)
	MUTATORsetNumeratorCharacterString(CharacterString)
	MUTATORsetPlaces(IntegerNumber)
	MUTATORsetPlacesCharacterString(CharacterString)

IntegerNumber methods

IntegerNumber	CONSTANTadd(IntegerNumber)
IntegerNumber	CONSTANTaddCharacterString(CharacterString)
RationalNumber	CONSTANTaddRationalNumber(RationalNumber)
IntegerNumber	CONSTANTCOPY()
RationalNumber	CONSTANTdivideBy(IntegerNumber)
RationalNumber	CONSTANTdivideByCharacterString(CharacterString)
RationalNumber	CONSTANTdivideByRationalNumber(RationalNumber)
BooleanValue	CONSTANTequals(IntegerNumber)
BooleanValue	CONSTANTequalsCharacterString(CharacterString)
BooleanValue	CONSTANTequalsRationalNumber(RationalNumber)
BooleanValue	CONSTANTgreaterThan(IntegerNumber)
BooleanValue	CONSTANTgreaterThanCharacterString(CharacterString)
BooleanValue	CONSTANTgreaterThanRationalNumber(RationalNumber)
BooleanValue	CONSTANTlessThan(IntegerNumber)
BooleanValue	CONSTANTlessThanCharacterString(CharacterString)
BooleanValue	CONSTANTlessThanRationalNumber(RationalNumber)
IntegerNumber	CONSTANTmultiplyBy(IntegerNumber)
IntegerNumber	CONSTANTmultiplyByCharacterString(CharacterString)
RationalNumber	CONSTANTmultiplyByRationalNumber(RationalNumber)
IntegerNumber	CONSTANTsubtract(IntegerNumber)
IntegerNumber	CONSTANTsubtractCharacterString(CharacterString)
RationalNumber	CONSTANTsubtractRationalNumber(RationalNumber)
CharacterString	CONSTANTtoCharacterString()
RationalNumber	CONSTANTtoRationalNumber()
	MUTATORset(IntegerNumber)
	MUTATORsetCharacterString(CharacterString)
	MUTATORsetRationalNumber(RationalNumber)

CharacterString methods

CharacterString	CONSTANTCOPY()
BooleanValue	CONSTANTequals(CharacterString)
CharacterString	CONSTANTgetLeft(IntegerNumber)
IntegerNumber	CONSTANTgetLength()
CharacterString	CONSTANTgetRight(IntegerNumber)
BooleanValue	CONSTANTgreaterThan(CharacterString)
BooleanValue	CONSTANTisAlpha()
BooleanValue	CONSTANTlessThan(CharacterString)

IntegerNumber	CONSTANTtoIntegerNumber()
RationalNumber	CONSTANTtoRationalNumber()
	MUTATORappend(CharacterString)
	MUTATORset(CharacterString)
	MUTATORsetIntegerNumber(IntegerNumber)
	MUTATORsetRationalNumber(RationalNumber)
	MUTATORtoLowerCase()
	MUTATORtoUpperCase()

References

Brilliant, S., & Wiseman, T. R. (1996). The first programming paradigm and language dilemma. *ACM SIGCSE Bulletin, 28*(1), 338–342.

Deek, F., & McHugh, J. A. (1998). A survey and critical analysis of tools for learning programming. *Computer Science Education, 8*(2), 130–178.

Dijkstra, E. (1968). Go to statement considered harmful. *Communtcations of the ACM, 11*(3), 147–148.

Gupta, D. (2004). What is a good first programming language? *Crossroads, 10*(4), 1–15.

Hudak, P., & Fasel, J. H. (1992). A Gentle Introduction to Haskell. *SIGPLAN Notices, 27*(5).

Kolling, M. K. (1995). Requirements for a first year object-oriented teaching language. *SISCSE Technical Symposium on Computer Science Education* (pp. 173–177). Nashville: ACM Press.

Kolling, M. (1999). The problem of teaching object-orineted programming, part 1: languages. *Journal of Object-Oriented Programming, 11*(8), 8–15.

Mannila, L., & de Raadt, M. (2006). An objective comparison of languages for teaching introductory programming. *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006* (pp. 32–37). Uppsala: ACM.

McIver, L., & Conway, D. (1996). Seven deadly sins of introductory programming language design. *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice* (pp. 309–316). Dunedin: IEEE COmputer Society.

Parker, K., Chao, J., Ottaway, T., & Chang, J. (2006). A formal language selection process for introductory programming courses. *Journal of Information Technology Education, 5*, 133–151.

Raadt, M., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses. *Proceedings of the Informing Science + IT Education Conference* (pp. 329–337). Cork: Informing Science Institute.

Springer, G., & Friedman, D. P. (1989). *Scheme and the art of programming.* The Massachusetts Institute of Technology.

Wirth, N. (1971). The programming language pascal. *Acta Informatica, 1*, 35–63.