WysiScript: Programming via direct syntax highlighting

William Gunther
Epic
wgunther@epic.com

Brian Kell
Google, Inc.
bkell@google.com

SIGBOVIK '17
Carnegie Mellon University
March 31, 2017

Abstract

The efficiency of programming is often hampered by the need to type the right text in order to obtain the syntax highlighting that will produce the desired program behavior. The programmer can control the colors and formatting of the code only indirectly, through arcane textual incantations. In this paper we introduce WysiScript, a new language which frees the programmer from this antiquated dependence on text by allowing program semantics to be expressed through direct application of colors and formatting. We give a description of the main ideas of the language and demonstrate its power and ease of use with some example programs. We end by proposing a novel technique for understanding the structure of a program, which is made possible only by the fresh approach taken by WysiScript.

Introduction

An important question in the study of the foundations of computer science is the following: What makes a programming language a programming language? In other words, what distinguishes a programming language from plain text? Consider, for example, the samples of plain text and a programming language shown in Fig. 1 below.

(a) Plain text.

A major physiographic province of North Americ
Plains lie between the Rio Grande on the south a
where the Mackenzie River empties into the Arct
the north, between the Central Lowland of the U
and the Canadian Shield on the east and the Rock
on the west. Their length is some 3,000 miles,
from 300 to 700 miles, and their area ap
1,125,000 square miles (2,900,000 square kilomet
ly equivalent to one-third of the United States.
states of the United States (Montana, North Da
Dakota, Wyoming, Nebraska, Kansas, Colorado,
Texas, and New Mexico) and the three Prairie I
Canada (Manitoba, Saskatchewan, and Alberta), a
of the Northwest Territories are within the Great
er. Some writers have used the 100th meridian as
boundary, but a more precise one is an eastwar
carpment that runs from Texas to North Dakot
somewhat east of the 100th meridian. In the Cana
the line dividing the Great Plains from the Cana
runs east of the Red River of the North; cuts th

(b) A programming language.

```c
#include <stdio.h>
/* Returns the number of nodes in the sub
int count_nodes(const Node *t) {
  if (!t) return 0;
  return 1 + count_nodes(t->left) + count
}
/* Compares two nodes and returns the one
Node *max_node(const Node *a, const Node
  return a ? b ? a->key > b->key ? a : b
}
/* Returns the node with the largest key
Node *max_subtree_node(const Node *t) {
  if (!t) return 0;
  return max_node(max_node(t, max_subtree
                  max_subtree_node(t->rig
}
/* Emits a textual representation of a no
void emit_subtree(const Node *t) {
  putchar('[');
  if (t) {
    printf("%d,", t->key);
    emit_subtree(t->left);
    putchar(',');
    emit_subtree(t->right);
```

Fig. 1. Plain text versus a programming language.

A careful comparison of these two samples makes the difference clear: the primary distinction is that programming languages have colors and formatting. Obviously it is this special formatting that gives programming languages their power. This is why plain text is not executable—it lacks formatting. (This also explains the lack of functionality in Microsoft Excel.)

Traditionally programming has been done by typing complicated text in order to produce the proper colors and formatting to make the program work correctly. The colors and formatting that determine the program's behavior can be controlled only indirectly by adjusting the text. This is a tedious, roundabout method (Fig. 2).

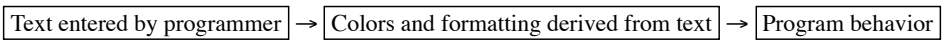

Fig. 2. The traditional process of programming.

In this paper, to improve programming efficiency, we introduce WysiScript, which allows the programmer to bypass the step of typing text and simply apply the desired formatting directly (Fig. 3). And WysiScript has a lot of formatting, which makes it more powerful than typical programming languages.

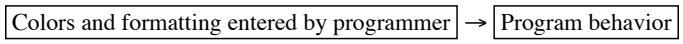

Fig. 3. The simplified process of programming in WysiScript.

This paper is intended to provide a general introduction to WysiScript. Further information about the language, as well as the standard reference implementation, is freely available at http://www.zifyoip.com/wysiscript/.

Previous work

Programming languages have not always been text-based. Early work explored many diverse paradigms. For example, the first widely used programming language, Build-A-New-Machine-For-Each-Task, did not employ text at all, nor did its popular successor, Plug-Wires-Into-Different-Places.

The field of nontextual programming languages was quite popular in the Eastern Bloc in the 1950s and early 1960s. In 1953 the Bulgarian computer scientist Dimitar Radjakov developed a programming language based on the timing and intensity of a sequence of thwacks delivered to the side of the computer cabinet [5]; Bulgarian computers relied on this system for a number of years. Late in 1957 the Yugoslav mathematician Dragana Šimunović proposed a language in which a program is a collection of rough and smooth objects thrown into a leather bag [6], but this was never satisfactorily implemented. And in 1962 the Soviet biochemist Andrey Mikhailovich Turapovsky was the first to describe a full-featured odor-based language [7].

However, because of the political climate of the time, most of this work was completely unknown in the West. Some promising results were demonstrated by Emily W. Hagenfried, including one programming language in 1955 encoded using the calls of the West Peruvian screech owl *Megascops roboratus* [3] and, with Pål Svendt, another in 1958 based on heating and cooling different parts of the hardware [4]. Nevertheless, by the mid to late 1960s the textual programming paradigm had become fully entrenched.

Consequently much work was devoted to the study of so-called "formal languages" (i.e., text-based languages, with thinly veiled disdain for nontextual systems) and their parsing and analysis. Eventually editors were developed that could perform the necessary translation of complex textual programs into colors and formatting. For programmers, this workflow is tedious and error-prone, requiring not only an understanding of how the colors affect program behavior but also the arcane knowledge of exactly what text is necessary to make the editor produce those colors. The sole focus on text-based programming languages has also had other undesirable effects, such as the proliferation of code written by colorblind people, the endless tabs-versus-spaces debate, and PHP.

Language description

In this section we give a brief description of the major ideas in WysiScript. At the time of this writing, the language is under active development (i.e., we threw most of this together after the first SIGBOVIK submission deadline had al-

ready passed, and we haven't started on the implementation yet), so details are subject to change. Please see http://www.zifyoip.com/wysiscript/ for the latest documentation.

Naturally, color is fundamentally important to WysiScript. If you are reading this document in a black-and-white format, such as, say, printed conference proceedings, some things may be difficult to understand. We recommend reading this document in the PDF version of the proceedings or from the URL above. Throughout this document, colors are expressed using CSS syntax.

Numeric literals

Most programming languages in use today use only a single color for all numeric literals, which is unnecessarily confusing and makes it impossible to distinguish different numbers using only a spectrometer. To improve clarity, WysiScript uses a different foreground color for each number. There are many possible ways to map RGB colors to numbers; WysiScript uses the simple scheme (256 · red + green) / blue, with the standard mathematical convention that division by zero really means division by 256. Additionally, to emphasize their immutability, numeric literals are underlined. So, for example, the number 12345.67 can be represented as #90AD03. Of course, this is only an approximation (that color actually represents 12345⅔), but it's probably what you meant anyway.

Not only does this system associate different colors to different numbers, it also often associates different colors to the *same* number. This provides a nice set of "pet names" for numbers to which a programmer feels a particular emotional attachment. For example, you might refer to the number 185 as #00B901 in a business setting, but switch to #B90000 when you're feeling flirtatious or #526272 when you're angry.

Variables and functions

Likewise, for clarity, WysiScript uses a different color for each variable and function. For example, A denotes the variable or function #F00BA2. The value of this expression is the value of the variable or the return value of the function.

Since each variable has a corresponding color, assignment is easily represented with background colors. For instance, A denotes the assignment of the number 12345⅔ to the variable #F00BA2. Note that this could also be written as mov, because we are removing the arbitrary indirect association between text and meaning to focus only on the clear meaning conveyed by the formatting.

This straightforward scheme allows certain expressions to be written quite concisely without sacrificing readability. For example, A means, "Assign the value of the variable #F00BA2 (or the return value of the function #F00BA2) to the variable #DABADA." In most traditional programming languages, such an expression would wastefully require at least three symbols: two variable names and an assignment operator.

Blocks and expressions

Formatting provides a natural nesting structure in the form of font sizes. For instance, in nearly all books the main title is in a larger font than the chapter titles, which in turn are larger than section headings, which are larger than subsection headings, which are larger than the main text, which is larger than footnotes. This system is the result of centuries of refinement by printers and graphic designers and allows the reader to understand the structure at a glance. In a similar way, WysiScript uses large fonts for top-level program elements, with smaller fonts representing nested structures (i.e., child nodes in the syntax tree).

This feature of WysiScript yields a great improvement in readability. As any programmer knows, in traditional programming languages it easy to get lost in nested braces and parentheses. WysiScript does away with these clumsy textual representations of structure entirely and makes the full program structure immediately apparent.

Consider the example in Fig. 4 below, which defines a function to compute the greatest common divisor of two numbers using the Euclidean algorithm, calls the function with the arguments 45 and 105, and outputs the result. (We have not yet discussed function definitions or built-in operations, so the meaning of a few parts of this program may not be immediately clear.)

Fig. 4. A sample WysiScript program.

For ease of discussion, the nodes in this example have been given distinct text labels. Node B is a child of node A, because it has a smaller font size. Likewise, node C is a child of node B, and node D is a child of node C. Node E is also a child of node C (it is a sibling of node D), because it has the same font size as D but different colors. Node F is a child of node B and a sibling of node C, as is node J, and nodes G, H, and I are the three children of node F. Node K is interesting: it represents a node in the syntax tree that is a sibling of node B, but all of its non-size formatting is the same as that of B. In order to indicate that it is a sibling of node B and not just a continuation, it has been given a font size that is larger than that of node B but smaller than that of their parent, node A. The function definition in this example continues through node W. Node X follows the function definition; it is another top-level node, a sibling of node A. Note that node Y has two children, nodes Z and AA. The two characters 'A' at the end of the program are both part of the same node of the syntax tree because they have the same formatting, including font size.

This example also demonstrates the syntax of a function call. The function defined in this example is named #6CD, and it is called at node Y. The two arguments to this function, #002C01 and #006901 (representing the numeric literals 45 and 105), are provided to the function call as child nodes in the syntax tree, nodes Z and AA.

Recall that assignment is represented by background color. Naturally, if the value of an expression is assigned to some variable, the corresponding background color extends over the entire expression. Of course, within that expression there may be subexpressions whose values are assigned to other variables, so subexpressions may have their own background colors (as illustrated in Fig. 4). The font sizes make the nested structure clear, so no confusion arises. Note two obvious and common-sense corollaries: an expression may not be assigned to the same variable as an ancestor expression unless an intermediate expression is assigned to a different variable, and if no background color is explicitly set on a top-level expression then its value is assigned to the variable white (or whatever the background color of the environment happens to be).

Function definitions

Function definitions look the same as variable assignments except that they are italicized. The expression to be used as the function body is italicized and its background color is set to the color of the function. The return value is the result of that expression.

Be careful not to accidentally turn variable assignments inside a function body into local function definitions by italicizing too much. Even though a function definition is italicized, the variable assignments it contains should be unitalicized so that they are interpreted correctly. Unless, of course, you *want* local function definitions—then by all means italicize them. (Local function definitions will probably work, but who knows. Good luck if you decide to use them.)

Note that recursive function calls are invisible, because the foreground color of the function call matches the background color of the function definition. This is not a serious problem, though—if a programmer is really concerned about being able to see her code, she can always rewrite a recursive function as two mutually recursive functions with contrasting colors. In fact, the invisibility of recursive function calls can be a benefit for complexity analysis, because that's always easier if you don't have to worry about recursion.

Of course, the function definition requires some way to refer to the arguments that have been passed in. We make use of the well-known Roy G. Biv calling convention, which is also used, for example, by the Randy Pausch Bridge and the Allegheny County Belt System. Under this convention, the arguments of a function are named, in order, **red**, **orange**, **yellow**, **green**, **blue**, **indigo**, and **violet**. Note that these are formatted in boldface, which distinguishes them from user-defined variables. In addition to improved readability when compared to other programming languages that use the same color for all parameters, this convention has the advantage of encouraging good programming practice by keeping the number of function parameters small. (If a function really needs to take more than seven arguments, they can be redshifted with the built-in **deepskyblue** operation; see below.) This convention also provides a rigorous definition for **#F00**, a symbol that often appears in programming examples but whose meaning is usually ambiguous.

Function arguments are the only way that outside values can be used inside a function. To support good programming practices, there are no global variables in WysiScript.

Built-in functions

WysiScript provides a wide array of useful built-in functions. In order to distinguish these functions from user-defined functions, they are formatted in boldface. This is modeled after many other programming languages, which format their keywords in boldface; WysiScript is the same, except that it has keycolors.

To give a taste, a few selected functions are described below.

Control structures

honeydew	Compound expression, similar to a **do** block in some other programming languages (but with honey). Takes arbitrarily many arguments, evaluates them in order, and returns the value of the last one.
#1FE15E	Conditional. Takes an odd number of arguments, which are interpreted as condition–expression pairs with one unpaired expression at the end. The conditions are evaluated in order until one of them evaluates to a nonzero value, at which point the corresponding expression is evaluated and returned. If all conditions evaluate to zero, the value of the final unpaired expression is returned.
teal	Loop till a condition is nonzero. Takes two arguments: an expression representing the body of the loop, and a condition. Evaluates the body followed by the condition. If the value of the condition is nonzero, returns the value of the body; otherwise reevaluates the body and the condition again, continuing till the condition becomes nonzero. (Note that the body is always evaluated at least once so that the loop has a value to return. To get the effect of a **while** loop in some other programming languages, put the **teal** loop inside an **#1FE15E** expression, and negate the loop condition, of course.)
deepskyblue	Redshifts function arguments. In other words, the current value of **orange** is assigned to **red**, the current value of **yellow** is assigned to **orange**, and so on, and the first function argument that was not previously assigned to any of the colors **red** through **violet** is assigned to **violet**. This allows a function to accept arbitrarily many arguments. Returns the previous value of **red**.
ghostwhite	Takes one argument. Returns 1 if the argument is a ghost (i.e., a variable or function to which no value has been assigned); returns 0 otherwise.
#5CA1A2	Takes one argument. Returns 1 if the argument is a scalar (i.e., a single numeric or char value, as opposed to a chart—see below); returns 0 otherwise.
fuchsia	Takes one argument. Returns 1 if the argument is a ~~fuchsian~~ function, or 0 otherwise.

Comparisons and Boolean operations

The following operators that take arbitrarily many arguments all evaluate their arguments left to right and short-circuit.

plum	Equality. Takes arbitrarily many arguments. Returns 1 if they are all plumb equal, or 0 otherwise.
#1E55E2	Lesser than. Takes arbitrarily many arguments. Returns 1 if each argument is lesser than the next, or 0 otherwise.
#B166E2	Bigger than. Takes arbitrarily many arguments. Returns 1 if each argument is bigger than the next, or 0 otherwise.
#70661E	Toggles a Boolean value. Takes one argument. Returns 1 if the argument is zero, or 0 otherwise.
#A11	Conjunction (“and”). Takes arbitrarily many arguments. Returns 1 if all arguments are nonzero, or 0 otherwise.

gold	Disjunction (“or”). Takes arbitrarily many arguments. Evaluates the arguments in order until a nonzero value is found, at which point that value is returned. If none of the arguments is nonzero, returns 0. Note that if all arguments are 0 or 1, this has the effect of returning 1 if any argument is nonzero or 0 otherwise.

Math

#ADD	Addition. Takes arbitrarily many arguments and returns their sum.
#D1FFE2	Subtraction. Takes arbitrarily many arguments and returns the first minus the sum of the rest (i.e., left-to-right subtraction).
#D07	Multiplication. Takes arbitrarily many arguments and returns their product.
#D171DE	Division. Takes arbitrarily many arguments and returns the first divided by the product of the rest (i.e., left-to-right division). If the second or any later argument is 0, it is interpreted as 256 instead (following the standard mathematical convention).
#2E51D0	Residue. Takes arbitrarily many arguments and returns the result of a left-to-right remainder operation. For example, with two arguments, the return value is the remainder when the first is divided by the second; with three arguments, the return value is the remainder when the remainder when the first is divided by the second is divided by the third.
powderblue	Exponentiation. Takes arbitrarily many arguments and returns the result of a right-to-left exponentiation operation (but the arguments themselves are evaluated left to right). For example, with two arguments, the return value is the first raised to the power of the second; with three arguments, the return value is the first raised to the power of (the second raised to the power of the third).
#106	Natural logarithm (i.e., logarithm to the base #02ADFC). Takes one argument.
#AB5	Absolute value. Takes one argument.
#F10002	Floor. Takes one argument.
sienna	Sine. Takes one argument, expressed in radians.
#C05	Cosine. Takes one argument, expressed in radians.
tan	Tangent. Takes one argument, expressed in radians.
moccasin	Arcsine. Takes one argument and returns its arcsine, expressed in radians.
#A2CC05	Arccosine. Takes one argument and returns its arccosine, expressed in radians.
#A26	Argument (in the complex-analytic sense). Takes two arguments, specifying the ordinate and abscissa of a point in the complex plane, and returns the argument of that point. Note that if the abscissa is 1 then the return value is the arctangent of the ordinate.
#314159	Returns the constant #016371, the ratio of the circumference of a circle to its diameter.
#271828	Returns the constant #02ADFC, the base of the natural logarithm.

Charts

Charts are roughly similar to what other programming languages call “arrays” or “lists,” but more nautical. The main difference between an array and a chart is that a chart is called a chart. Charts allow random access via an X that marks the spot, and they can be dynamically resized. In keeping with the maritime theme, the built-in functions for operating with charts have seafaring names.

coral	Takes arbitrarily many arguments and corrals them into a chart. The X's of the returned chart are increasing consecutive integers starting at #000101.
seashell	Takes one argument, a chart. Returns 1 if the chart is an empty shell (i.e., contains no values), or 0 otherwise.
navy	Takes two arguments: a chart and an X. Navigates to the indicated spot in the chart and returns the value there.
chartreuse	Takes three arguments: a chart, an X, and a value. Writes the value to the chart at the indicated spot. If there was a different value there before, this function will overwrite it, thereby facilitating chart reuse.
salmon	Takes one argument, a chart. Returns another chart whose values are the X's of the argument (and whose X's are increasing consecutive integers starting at #000101). Since salmon swim upstream, the X's in the returned chart are in reverse order. This means that if the X's of the argument chart are themselves increasing consecutive integers starting at #000101, then the value at X #000101 in the returned chart is the number of spots in the argument chart (as long as the argument chart is not an empty shell, for which salmon would return an empty shell). In any case, taking the salmon of the salmon of a nonempty chart will always give the number of spots in the chart as the value at X #000101. Therefore, if the variable #F00BA2 holds a chart, then the number of values it contains can be determined by the straightforward expression ABCDEFGHI. This simplicity of finding the number of values in a chart is a clear advantage of WysiScript over other programming languages. For example, to find the length of an array `a` in Standard ML it is necessary to type `Array.length a`, which is five characters longer than the equivalent WysiScript and much less colorful.
maroon	Takes two arguments: a chart and an X. Removes the value at that spot in the chart, leaving it marooned. Returns the marooned value.

Strings

Characters, or chars, are represented by their Unicode code points. A string is simply a chart of chars.

#DEC0DE	Takes a string, parses it as a number, and returns the parsed value.
#2EC0DE	Takes a number and converts it to a string.
ivory	Takes one argument. Returns 1 if it is 'i', 'v', or 'y', or 0 otherwise.

I/O

#6E7	Standard input. Gets one character from stdin and returns it. Returns #E0F on EOF. Of course, this value is also returned if the character read from stdin was '฀', but what are the chances of that?
#FACADE	Standard output. Takes arbitrarily many arguments and writes them to stdout. Arguments that are single values are interpreted as numbers; arguments that are charts are interpreted as strings.
#B00B00	Standard error. Like the above but writes to stderr.
#D1E	Aborts program execution with a specified error message.

Turing-completeness

It is self-evident that WysiScript is more powerful than most existing programming languages because it has more colors and formatting. However, some snooty theoretical computer scientists have shown a reluctance to accept this clear

truth because of the lack of color-based results in the literature, and a couple have even gone so far as to question whether WysiScript is a practical language at all. To answer their objections, in this section we formally prove the power of WysiScript.

Theorem. *WysiScript is Turing-complete.*

Proof. We make use of the language $\mathcal{P}''$, which was introduced by Böhm and Jacopini in 1964 and proven to be Turing-complete [1, 2]. Therefore it suffices to exhibit a $\mathcal{P}''$ interpreter in WysiScript. We assume the reader is familiar with $\mathcal{P}''$, so we will not belabor the details of that language.

Fig. 5 gives the source for a simple $\mathcal{P}''$ interpreter. This interpreter assumes that the $\mathcal{P}''$ program, written with the characters 'R', 'λ', '(', and ')', will be entered on standard input, followed by the character '.', followed by the initial contents of the tape cells (one character per cell), followed by EOF. The tape cells in $\mathcal{P}''$ hold symbols from a specified finite alphabet, which this interpreter takes to be the set {0, 1, 2, …, 255}, with 0 as the blank symbol; the alphabet is easily changed if desired by modifying the constant #F0F in the code. The interpreter then executes the $\mathcal{P}''$ program on the given tape. The tape head begins at the rightmost cell of the left-infinite tape. Execution stops when the instruction pointer moves past the end of the program, at which point the interpreter prints the final tape contents to standard output.

For simplicity, no syntax checking is done on the input $\mathcal{P}''$ program. For example, it is assumed that the program contains only valid characters and all parentheses nest properly. Of course, such syntax checking should be added before this $\mathcal{P}''$ interpreter is used for mission-critical applications. This easy extension is left as an exercise for the reader.

Since WysiScript can emulate any $\mathcal{P}''$ program on any input tape, and $\mathcal{P}''$ has been proven to be Turing-complete, we can conclude that WysiScript itself is Turing-complete. ∎

Fig. 5. A simple $\mathcal{P}''$ interpreter implemented in WysiScript.

We thank the anonymous SIGBOVIK reviewers for their careful verification of the correctness of this program, which is, in their words, "100% certified to be absolutely free of any possible errors whatsoever."

Comments and syntax texting

The astute reader may have noticed that we have not yet mentioned anything about comments, which may be surprising because comments are typically covered early in the descriptions of most programming languages. WysiScript has an extraordinarily natural and flexible commenting mechanism, but it can seem somewhat puzzling until the rest of the language is understood. Since the artificial link between text and meaning has been discarded, the text of a WysiScript program is free to be used for any comments at all!

For example, the text of the sample WysiScript program in Fig. 4 can be rewritten slightly to help illustrate its structure, as shown in Fig. 6 below.

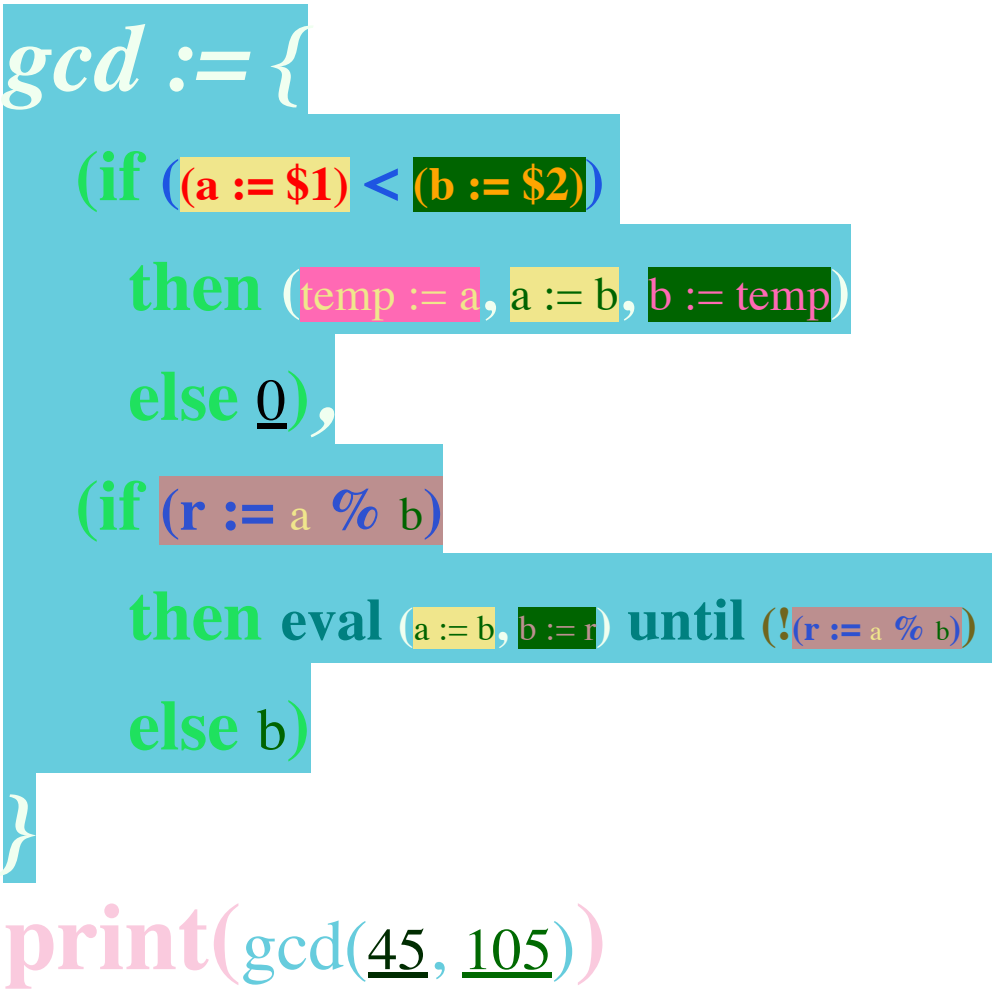

Fig. 6. A sample WysiScript program with comments in the text.

In fact, an intelligent WysiScript editor can automate the writing of comments in this way, modifying the text of the program on the fly in order to highlight its syntax. We call this technique "syntax texting," and as far as we know WysiScript is the first programming language to fully support such a system. Of course, a well-written WysiScript program is usually self-explanatory, but occasionally these textual comments can provide an additional boost to readability.

We understand that this new idea of syntax texting may take some time for traditional programmers to get used to, but we are confident that its benefits will soon become evident. With the availability of syntax texting, we hope that the text of a program will come to be seen as a useful aid in understanding the program's structure.

Conclusions and future work

In this paper we introduced WysiScript, a formatting-based programming language that streamlines the programming process by removing the awkward necessity of writing text. We gave an overview of its main ideas and a tour of its built-in features, demonstrated its simplicity and ease of use with several examples, and proved rigorously that it is equally as powerful as other languages and significantly more colorful. We also proposed syntax texting, a new technique that describes the structure of a program in its own text, which was not previously possible before the introduction of WysiScript.

WysiScript represents the first example of a new and powerful programming paradigm. We believe it is poised to become an important general-purpose language, and we urge its adoption in introductory computer programming courses. In particular, we believe it will appeal to many types of students who may not have an interest in traditional text-based programming, including painters, graphic designers, and the illiterate.

An obvious limitation of WysiScript in its current form is the restriction, imposed by the RGB color model, that a single program can have at most 16,777,216 different user-defined variables and functions. This hinders the use of WysiScript for census-taking in the Netherlands, for example, because the Dutch population is 16,979,729. Future versions of the language may relax this restriction by adding support for additional colors not representable in RGB, such as ultraviolet, infrared, and plaid.

Another promising direction for future research is the extension of WysiScript to object-oriented programming. CSS already provides classes, which can be used as a foundation. With the potential for an object-oriented language in mind, we have reserved the keycolor thistle to be used as a reference to the current object.

In order to support quantum computing, which makes use of value superposition and probabilistic algorithms, future versions of WysiScript may make use of the alpha channel in color specifications.

References

[1] Böhm, Corrado. On a family of Turing machines and the related programming language. *ICC Bulletin*, 3(3):187–194, July 1964.

[2] Böhm, Corrado, and Jacopini, Giuseppe. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[3] Hagenfried, E. W. An analysis of communication in *Megascops roboratus* with applications to computability. *Journal of South American Ornithology*, 26(1):65–74, January 1955.

[4] Hagenfried, E. W., and Svendt, P. Thermocomputing with diamagnetic flux cores. *Transactions on Circuit Design and Analysis*, 11(4):295–306, October 1958.

[5] Раджаков, Димитър. Спецификация на компютърни програми чрез навременна натупвам здравата. *Вестник на Българската Академия на Науките*, 43(2):217–242, April 1953.

[6] Šimunović, Dragana. Grube i glatki predmeti bačeni u kožnoj torbi. *Zbornik Radova Četvrtog Godišnjoj Konferenciji o Računanja s Vrećice i Kutije*, 71–84, December 1957.

[7] Тураповский, А. М. Новая система для ароматических вычислений с приложениями для производства сыра. *Российский журнал вычислительной химии*, 14(4):320–344, August 1962.