Language-Agnostic Program Rendering for Presentation, Debugging and Visualization

Christian S. Collberg Sean Davey
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{collberg,sdavey}@cs.arizona.edu

Todd A. Proebsting
Microsoft Research
One Microsoft Way
Redmond, WA
toddpro@microsoft.com

Abstract

We describe a language-independent and specification-driven program rendering tool that is able to produce high-quality code renderings of arbitrary complexity. The tool can incorporate arbitrary types of information together with the program code, allowing it to be used for debugging and profiling as well as for producing beautiful renderings of programs for publication.

We also present a model for the rendering of programs and apply it to the design of a rendering of Java control flow.

1. Introduction

Computer programs are read much more often than they are written. They are read by their author as they are being written and re-read as they are being debugged, they are read by code review teams to check for logical and stylistic errors, and they are read again and again by maintenance programmers as the programs are extended with new functionality. Programs, or at least fragments of programs, are also published in textbooks and manuals and read by students and professionals in educational situations.

In other words, while millions of lines of code are *written* every day, millions more are being *read.*

In this paper we will concern ourselves with the readability of programs. More precisely, we will discuss *program rendering*, how the text of a program is presented and laid out on the printed page. We will be particularly interested in

a) how information pertaining to the program (other than the actual code itself) can be incorporated into the rendering,

b) how different rendering views of a program can be produced to aid different programming tasks, and

c) how we can model the rendering of computer code.

We will also describe a program rendering tool, ART, that we have developed. ART[1] is language-independent, specification-driven, fully configurable and able to produce high-quality code renderings of arbitrary complexity, incorporating arbitrary types of information. We will explore how this flexibility allows ART to be used as a debugging tool as well as a tool for producing beautiful renderings of programs for publication.

2. Related Work

Obviously, we are not the first to investigate the layout and beautification of program code. Early LISP programmers discovered that the readability of S-expressions could be improved by *pretty-printing*, the careful layout of the program on the screen or printed page. Many similar tools have been developed for other languages, including Unix's `vgrind` which "formats the program sources ··· in a nice style using troff(1). Comments are placed in italics, keywords in bold face, and as each function is encountered its name is listed on the page margin."

Knuth [2] introduced *literate programming* [13, 16], where documentation prose is interspersed with nicely typeset program fragments. Fragments can be named, cross-referenced, and presented in any order. The resulting document is designed for reading by integrating code and documentation and presenting them in a "natural" order. Literate programming systems exist for several programming languages [10, 12].

Baecker and Marcus [1] give an in-depth discussion of the proper presentation of programs, and present a detailed

[1] *A*gnostic *R*endering *T*ool.

0-7695-0840-5/00 $10.00 © 2000 IEEE

typographic design for the C language. While their goals are similar to those of Knuth's literate programming, the visual result is very different and more similar to traditionally pretty-printed C. Programs are divided into chapters, keywords are highlighted, the size of punctuation characters is carefully chosen, and comments are typeset either as marginal notes or in special shaded sections. The result is a program presentation that looks more like a book or a manual than a computer printout.

Tripp [15] illustrates several graphical notations for describing control flow, some of which we have reproduced in Table 1. He classifies them into three groups: Box and line notations, Box notations, and Line notations. In spite of the vast number of proposals for graphical notations (Tripp presents eighteen), none of them ever gained acceptance with programmers. It is interesting to speculate why:

1. Programmers may find graphical notations to be unfamiliar and may prefer to view source code the traditional way, as text.
2. Programmers may find that graphical notations do not provide any information that is not already present in the source code, and hence lack the motivation to learn a new notation.
3. While box-based notations look very neat on academic examples, they tend not to scale well. Real programs have long identifiers, complex expressions, etc., that are difficult to render prettily within the confines of a box.

Erwig and Meyer [3] make a strong case for HVPL (heterogeneous visual languages) that use graphics when this is natural, and text otherwise. They present a programming environment that allows programmers to integrate visual programming concepts into their otherwise textual code. The system translates the pictorial part of the code into source code which can then be compiled and executed.

We agree with Erwig and Meyer that text and graphics are both important in programming, but our focus is on the presentation of programs rather than on the design of programming environments. In Section 3 we attempt to explore the limits of how programs can be *rendered*, i.e. presented (using combinations of text and graphics) on the printed page. In Section 4 we describe the design of a language-independent program renderer, and in Sections 5, 6, and 7 we discuss how the engine can be used for presentation, debugging, and profiling of programs. Section 8, finally, summarizes our results.

3. A Model of Program Rendering

In the following we will assume that programs are rendered in two dimensions, and that the rendering elements consist of text and arbitrary polygonal objects. Given these restrictions, we will try to explore and build a simple model for the types of renderings that are possible.

We will start by considering the simplest of control structures, the `if`-statement:

$$\texttt{if } E \texttt{ then } S$$

where E is an expression and S a statement. How can this structure be rendered on the printed page?

The first thing we must consider for our model is the relative positioning of the *abstract elements*[2] E and S of the rendering. There are eight possibilities:

$$\texttt{if} ::= \begin{array}{l} \mathsf{ES} \mid \begin{array}{l}\mathsf{E}\\ \mathsf{S}\end{array} \mid \begin{array}{l}\mathsf{E}\\ \mathsf{S}\end{array} \mid \begin{array}{l}\ \\ \mathsf{S}\end{array}\begin{array}{l}\mathsf{E}\\ \ \end{array} \mid \\ \mathsf{SE} \mid \begin{array}{l}\mathsf{S}\\ \mathsf{E}\end{array} \mid \begin{array}{l}\mathsf{S}\\ \mathsf{E}\end{array} \mid \begin{array}{l}\ \\ \mathsf{E}\end{array}\begin{array}{l}\mathsf{S}\\ \ \end{array} \mid \end{array} \quad (1)$$

ES represents the rendering where the statement is immediately to the right of the expression, $\begin{array}{l}\mathsf{E}\\ \ \end{array}\mathsf{S}$ represents the case when the statement is lower and somewhat to the right of the expression, etc. $\begin{array}{l}\mathsf{E}\\ \mathsf{S}\end{array}$, for example, models the conventional way a C `if` statement

```
if (E)
    S
```

is rendered.

In addition to the abstract elements of rule 1, most renderings will also contain *concrete elements*, or *decorations*. A decoration can be any textual or graphic component attached to an abstract element. To make the model practical, we will restrict decorations to be horizontal and vertical line-segments, keeping in mind that in any actual rendering these line-segments could be realized by arbitrary text or graphics. Rule 1 becomes:

$$\texttt{if} ::= \begin{array}{l} \boxed{\mathsf{E}}\boxed{\mathsf{S}} \mid \begin{array}{l}\boxed{\mathsf{E}}\\ \boxed{\mathsf{S}}\end{array} \mid \begin{array}{l}\boxed{\mathsf{E}}\\ \boxed{\mathsf{S}}\end{array} \mid \begin{array}{l}\ \\ \boxed{\mathsf{S}}\end{array}\begin{array}{l}\boxed{\mathsf{E}}\\ \ \end{array} \mid \\ \boxed{\mathsf{S}}\boxed{\mathsf{E}} \mid \begin{array}{l}\boxed{\mathsf{S}}\\ \boxed{\mathsf{E}}\end{array} \mid \begin{array}{l}\boxed{\mathsf{S}}\\ \boxed{\mathsf{E}}\end{array} \mid \begin{array}{l}\ \\ \boxed{\mathsf{E}}\end{array}\begin{array}{l}\boxed{\mathsf{S}}\\ \ \end{array} \end{array} \quad (2)$$

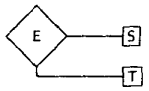
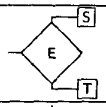
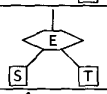
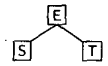
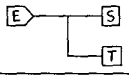
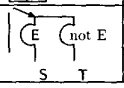
$\boxed{\mathsf{E}}\boxed{\mathsf{S}}$, for example, expresses that a rendering of an `if`-statement could consist of the abstract elements E and S (with E immediately to the left of S), a decoration between E and S, and decorations north, south, and west of E and north, south, and east of S.

$\boxed{\mathsf{E}}\boxed{\mathsf{S}}$ has 2^7 different renderings since any of its decorations could be either present or absent. In total, therefore, there are $4 \cdot 2^7 + 4 \cdot 2^8 = 1536$ different ways to render an `if`-statement.

When looking for ways to render code that are readable and pleasing to the eye, it is enlightening to enumerate all

[2]The term "abstract" is taken from "abstract syntax."

Table 1. Some box and line notations from Tripp [15].

Rothon Diagrams 	Ferstl Chart 	PAD 
Compact Chart 	Doran Chart 	Schematic Logic 
Flowblocks 	Lindsey Chart 	SPDM Diagram 
UFC Diagram 	Dimensional Flowchart 	

possibilities. Space restrictions prevent us from presenting a complete enumeration here, so we will limit ourselves to an `if`-statement's *natural orientations*. These are the positions with which programmers are most familiar: control is indicated by a south- or southeast-ward flow, and the predicate (E) acts like a "guard" restricting flow from entering the body (S). We get:

$$\texttt{if}_{\texttt{nat}} \quad ::= \quad \boxed{\texttt{E}}\boxed{\texttt{S}} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \quad\boxed{\texttt{S}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \boxed{\texttt{S}}\end{array} \tag{3}$$

Table 2 enumerates the $2 \cdot 2^7 + 2^8 = 512$ different renderings generated by rule 3. Rendering $\mathcal{R}_{4,4}$ (`|E|S`) represents a traditional syntactic layout such as

```
if E then S.
```

Since the lines in the rendering prototypes can represent more than one decoration, rendering $\mathcal{R}_{4,4}$ also models C's `if`-statement `if (E) S`. In this case, the two tokens `if` and `(` are represented by the leftmost vertical bar of `|E|S`.

More complex language constructs with more abstract elements will of course generate even more layout opportunities. For example, for the `if-then-else`-statement

```
if E then S else T
```

we get the rule

$$\begin{aligned}\texttt{ifelse}_{\texttt{nat}} \quad ::= \quad & \boxed{\texttt{E}}\boxed{\texttt{S}}\boxed{\texttt{T}} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\boxed{\texttt{S}}\\ \quad\boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\boxed{\texttt{S}}\\ \boxed{\texttt{T}}\end{array} \\ & \Big|\; \begin{array}{|l}\boxed{\texttt{E}}\boxed{\texttt{S}}\\ \boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \quad\boxed{\texttt{S}}\boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \quad\boxed{\texttt{S}}\\ \quad\boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \quad\boxed{\texttt{S}}\\ \boxed{\texttt{T}}\end{array} \\ & \Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \quad\boxed{\texttt{S}}\\ \boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \boxed{\texttt{S}}\boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \boxed{\texttt{S}}\\ \quad\boxed{\texttt{T}}\end{array} \;\Big|\; \begin{array}{|l}\boxed{\texttt{E}}\\ \boxed{\texttt{S}}\\ \boxed{\texttt{T}}\end{array}\end{aligned} \tag{4}$$

which has a total of $5 \cdot 2^{10} + 4 \cdot 2^{11} + 2 \cdot 2^{12} = 21504$ possible renderings. Again, we have limited ourselves to natural orientations: the predicate (E) should precede the `then`-case (S) which should precede the `else`-case (T) in a top-down, left-to-right rendering.

4. The Rendering Engine

Figure 1 gives an overview of the ART code rendering engine. Input to ART consists of the source code files ① that are to be rendered and a set of style files ④ (written in XML) that describe the layout of the desired rendering. The source code is processed by a compiler ② that has been modified to produce an intermediate representation (in METAPOST [6]) of the program. The XML style files are also processed by a translator ⑤ into METAPOST macros ⑥, and these macros are applied by the renderer ⑦ to the intermediate representation to produce the final rendering ⑧. In Section 7 we will see that ART is also able to present profiling data Ⓐ.

Our current implementation renders Java [4], but ART is completely source language agnostic: it can easily be modified to render programs in any language. To target ART to a new language $\mathcal{L}$ only requires access to a compiler $\mathcal{C}$ for $\mathcal{L}$. $\mathcal{C}$ is typically modified by adding an extra pass that traverses the abstract syntax tree (AST) that most compilers build internally. During this traversal all relevant syntactic and semantic information about the source program is collected and emitted in the form of METAPOST macro calls.

Figure 2 shows the ART XML specification of `if`-statements in the Flowblocks [11] notation, $\boxed{\begin{array}{|l}\boxed{\texttt{E}}\\ \boxed{\texttt{S}}\boxed{\texttt{T}}\end{array}}$. The renderings in Table 1 were all produced by similar specifications. At ① in Figure 2 we declare the three abstract elements of the rendering, `expr`, `then`, and `else`. At ② the `then` and `else` cases are joined together and at ③ the result is joined with the rendering of `expr`. At ④ the appropriate borders are drawn around `expr`, `then`, and `else`, and

Table 2. Enumeration of all decorated `if`-statements.

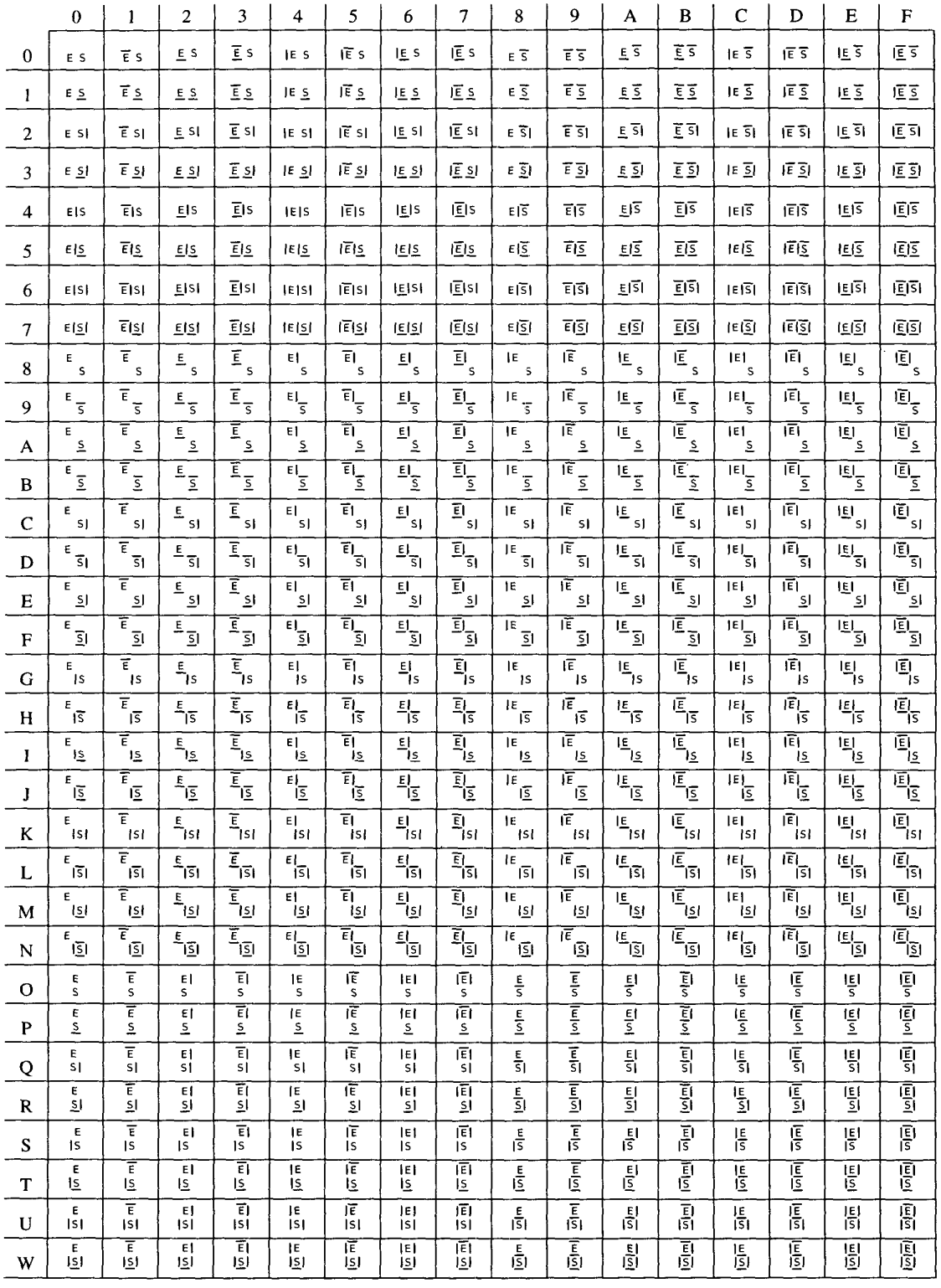


Figure 1. Overview of ART.

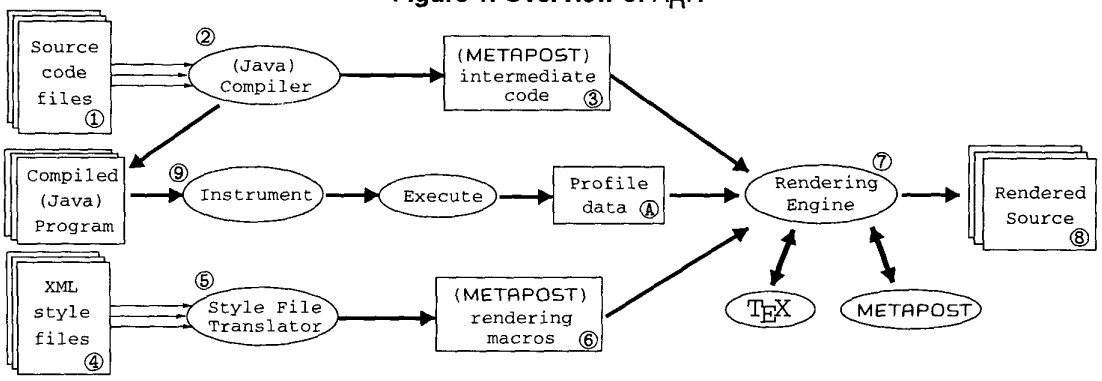


Figure 2. An ART specification of `if`-statements in the Flowblocks [11] notation.

```
<description name="ConditionalExpr">
   <input name="expr"/> <input name="then"/> <input name="else"/>  ①
   <group name="all">
      <group name="both">
         <join prev_pt="ne" next_pt="nw"> <object>then</object> <object>else</object> </join>  ②
         <border type="partial" start="ne" end="se"> <object>then</object> </border>  ④
         <border type="partial" start="sw" end="nw"> <object>else</object> </border>  ④
      </group>
      <border v_gap="0"> <object>both</object> </border>  ④
      <join prev_pt="sw" next_pt="nw"> <object>expr</object> <object>both</object> </join>  ③
   </group>
   <line>  ⑤
      <point>all.nw</point> <point h_offset="0">all.ne</point>
      <point h_offset="0" v_offset="0">all.se</point>
      <point v_offset="0">all.sw</point> <point>all.nw</point>
   </line>
</description>
```

at ⑤, finally, the outer border is drawn.

ART is based on TEX [8] and METAPOST for very good reasons. Both these systems are renowned for being highly flexible tools, but, more importantly, they are able to produce text and graphics of the highest typographic quality. TEX excels at formatting text and mathematics and is used by ART to render identifiers and expressions, while METAPOST (a derivative of METAFONT [7] that produces PostScript) is used to render graphic elements. As a result, ART is able to produce any conceivable rendering by combining text, graphics, and arbitrary mathematical structures.

5. Rendering for Presentation

As has been discussed by many authors, the physical layout of a program on the printed page can either improve or diminish the readability of the code. The flexibility of ART allows individuals to experiment with different types of renderings for different situations.

We have found, through our own experience, that the readability of traditional layouts of imperative and object-oriented languages is hampered by the similarity of keywords and identifiers. This is particularly evident when programs are prepared for presentation, for example on overhead transparencies where the code has to be read from a long distance away. To make keywords (such as `for`, `while`, and `if`) stand out they are sometimes rendered in boldface or with a different font than identifiers, but even that is not enough to reduce visual clutter and make a clean separation between control-flow and data-manipulation.

As a result, we advocate a rendering style where control-flow is presented graphically and manipulation of data (assignments, predicates, etc.) are rendered textually.

We find it particularly interesting and challenging to render *non-local control-flow*. Programs that make extensive use of exception handling (most Java programs fall in this

category) are notoriously difficult to read and understand, since most statements can potentially throw an exception which may transfer control to any number of exception handlers. Many programmers make use of exceptions not just to handle unusual error situations but as an additional control construct. For example, a common idiom is to read a file without explicitly testing for end-of-file. Instead, the read-loop is put inside a `try-catch`-block and the end-of-file-exception is caught and ignored.

Rendering control-flow graphically rather than textually gives us the opportunity to present non-local control-flow explicitly, rather than implicitly, which can make the code easier to read and understand.

5.1 A Visual Design for Java

We have found all proposed box-and-line notations (summarized in Table 1) unsatisfactory. Many of these notations add to the visual clutter by an overabundance of graphical elements. Boxes, in particular, are very visually distracting, particularly when the program has a great deal of syntactic nesting. Furthermore, many notations fail to take advantage of the fact that programmers are conditioned to a top-down, left-to-right, flow-of-control.

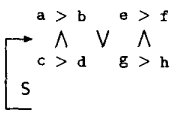
For these reasons, we favor renderings in the style of $\mathcal{R}_{T,5}$ ($\overline{|E}$ / $\underline{S}$) and $\mathcal{R}_{T,D}$ ($\overline{|E}$ / $\underline{S}$). These follow the natural flow of control and, when using simple lines as decorations, allow us to neatly and unobtrusively indicate syntactic nesting and non-local control-flow. As we shall see in Sections 6 and 7, these styles furthermore lend themselves well to renderings that incorporate semantic and run-time information.

Table 3 shows our proposed rendering grammar for the control structures of Java. Most of the designs are obvious, but some require explanation.

Complex boolean expressions are notoriously difficult to read and understand, particularly for languages with short-circuit evaluation. We have found that a "success-goes-down-failure-goes-right" layout enhances readability. For example, the predicate below is laid out such that when a sub-expression evaluates to `true` we continue reading downwards; when it evaluates to `false` we go right:

```
while (
   a>b && c>d ||
   e>f && g>h)
   S
```

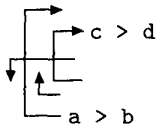
≡



In our design, "normal" control flow moves vertically on the page, while exceptional flow moves horizontally. This is consistent with how programs are read most of the time: we scan the code top-down, concentrating on the main-line and ignoring exceptional conditions. For example, `break` and `continue` (which transfer control to the beginning or end of a loop, respectively) are rendered as arrows going to the left edge of the page:

```
lab:  do {
   while (c>d)
      break lab;
   continue;
} while (a>b);
```

≡

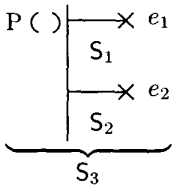


Non-local control-flow (exception throws and catches, method calls and returns) is rendered as arrows going to the right. At the right edge of the page these arrows connect to a set of "non-local bus-lines" that visually transfer control in and out of a method.

Because of its difficult semantics [14], `try-catch-finally`-statements are particularly difficult to render intuitively. Control can flow into the `finally`-clause from the `try`-body or from any of the `catch`-clauses:

```
try {
   P();
} catch (e1) {
   S1
} catch (e2) {
   S2
} finally {
   S3
}
```

≡



Naturally, the actual number of bus-lines will be language dependent. In a rendering of Icon [5], for example, special bus-lines will be needed for generator suspend and resume.

Figure 4 shows some different renderings of the same procedure, using rendering styles with or without keywords and with different kinds of graphic elements.

6. Rendering for Debugging

One of the novel and useful aspects of ARTˆ is its ability to combine the rendering of syntactic and semantic information. This turns out to be particularly useful when trying to read, understand, and debug object-oriented languages such as Java. For example, unlike in an imperative language where a procedure call ⌜`p()`⌝ always transfers control to the same procedure, a method invocation ⌜`r.m()`⌝ in an object-oriented language could go to any number of methods, depending on the run-time type of `r`. This makes reading and following the flow of control in an object-oriented language very challenging.

ARTˆ treats semantic information no differently from syntactic information. Any data that the modified compiler (② in Figure 1) can collect about the program can be passed down to the rendering engine and rendered according to

Table 3. Proposed rendering grammar.

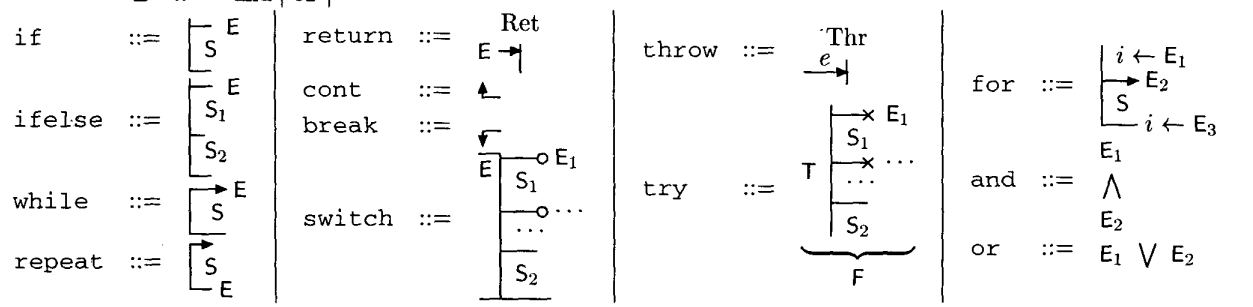
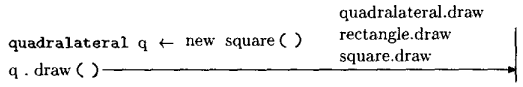


Figure 3. Rendering Java method call targets.



an ART specification. In Figure 3, for example, potential method call targets are rendered at the call site.[3]

In other languages, other types of static semantic information may be available to a compiler but not directly visible in the source. In Icon, a highly polymorphic language, there are no variable declarations that show a reader what type of value a particular variable may hold at run-time. An optimizing Icon compiler would have to collect such information (using aggressive inter-procedural data-flow type-analysis) and could easily pass it on to the rendering engine. Presenting this information to a programmer in a succinct rendering of the code would provide an invaluable debugging tool.

7. Rendering for Profiling

The final application of ART that we will discuss is as a tool for visualizing execution profiles. The current distribution of ART includes a simple statement-level profiler for Java, built on the BIT [9] Java class file instrumentation tool. The profiler (Ⓐ in Figure 1) instruments the executable (Java class-files in our current implementation) such that when the program is run, statement execution counts are produced. The counts are used to rank the execution frequency of the statements within each method. ART then renders this information according to the provided specification.

Figure 4(a) shows a rendering of ShellSort incorporating statement-level profiling information. The gray-level[4] at which each statement is rendered indicates its execution rank within the method. Other renderings are, of course, possible, including putting the actual execution counts in the margin, removing or high-lighting any statements that are never executed, etc. Such renderings can either aid a first code read-through by removing clutter, or aid debugging by alerting the programmer to statements which should have been, but never were, executed.

8. Summary and Discussion

This paper was inspired by a conversation where two of the authors discussed how they write pseudo-code by hand. Typically, such codes bear little resemblance to syntactically correct code in any particular language, but use arrows for loops, braces or brackets for block-structure, subscripts for array accesses, etc.

Thus arose the question of whether programmers would actually prefer to see their real code rendered similar to their hand-written code, integrating a limited amount of graphics with the textual source.

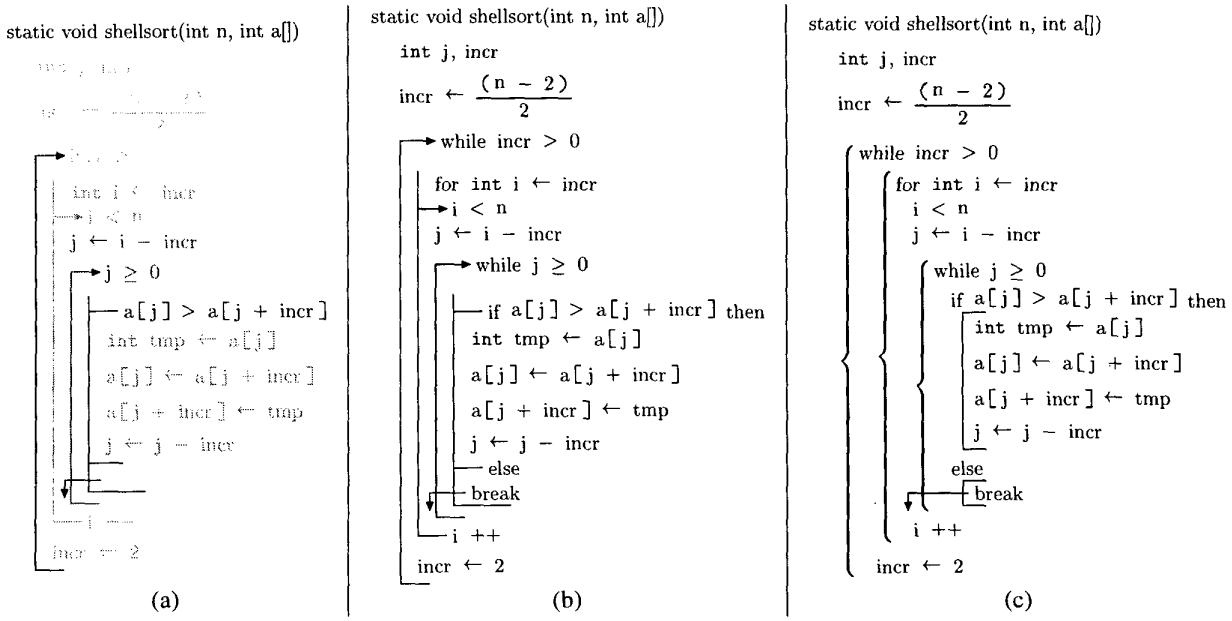
These thoughts evolved into the Java rendering design in Section 5, the more general model of code rendering in Section 3, and, in order to test the design on real code, the implementation of the specification-driven code rendering engine in Section 4.

Experiments with the new code layouts indicated that replacing keywords with unobtrusive graphical elements opened up new visual real-estate that could be used to extend the renderings with additional information. This lead

[3] This type of information must collected by looking at all the classes in a program.

[4] We prefer to use color to indicate execution rank, but, unfortunately, color is not available to us in this paper.

Figure 4. Three renderings of ShellSort.



to the extensions in Sections 6 and 7 that allowed the rendering engine to incorporate semantic and run-time information with the program itself.

The prototype rendering engine, example specifications, the modified Java compiler, and the Java statement-level profiler is available for download from `http://www.cs.arizona.edu/~collberg/Research/ART`. The prototype is under active development. In particular, we are working on improving ART's ability to combine individual specifications. The goal is to allow programmers at different stages of a project (debugging, performance-tuning, or documentation) to easily produce different views of the same program, employing different rendering styles and incorporating different types of compile-time and run-time information.

References

[1] R. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, 1990.

[2] J. L. Bentley, M. D. McIlroy, and D. E. Knuth. Programming pearls: A literate program. Reprinted in: *Literate Programming*, pages 151–177, 1992.

[3] M. Erwig and B. Meyer. Heterogenous visual languages—integrating visual and textual programming. In *IEEE VL'95*, pages 318–325, 1995.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[5] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Prentice Hall, 2nd edition, 1990.

[6] J. D. Hobby. Introduction to MetaPost. In *EuroTEX '92 Proceedings*, pages 21–36, Sept. 1992.

[7] D. E. Knuth. *The METAFONTbook*, volume C of *Computers and typesetting*. Addison-Wesley, 1993.

[8] D. E. Knuth. *The TEXbook*, volume A of *Computers and typesetting*. Addison-Wesley, 1994.

[9] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 73–82, Berkeley, Dec. 8–11 1997. USENIX Association.

[10] S. Levy. Literate programming in C. *TEXniques, Publications for the TEX community*, 5:125–130, 1987.

[11] C. H. Lindsey. Structure charts—a structured alternative to flowcharts. *ACM SIGPLAN Notices*, 12(11):36–49, Nov. 1977.

[12] J. D. Ramsdell. SchemeTEX — simple support for literate programming in Lisp. *TEXhax*, 88(39), Apr. 1988.

[13] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, Sept. 1994.

[14] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM TOPLAS*, 21(1):90–137, Jan. 1999.

[15] L. L. Tripp. A survey of graphical notations for program design—an update. *ACM SIGSOFT Software Engineering Notes*, 13(4):39–44, Oct. 1988.

[16] V. Wyk. Literate programming: An assessment. *CACM: Communications of the ACM*, 33, 1990.