Frances: A Tool For Understanding Code Generation

Tyler Sondag
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50014
sondag@cs.iastate.edu

Kian L. Pokorny
Division of Computing
McKendree University
701 College Road
Lebanon, IL 62254
klpokorny@mckendree.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50014
hridesh@cs.iastate.edu

ABSTRACT

Compiler and programming language implementation courses are integral parts of many computer science curricula. However, the range of topics necessary to teach in such a course are difficult for students to understand and time consuming to cover. In particular, code generation is a confusing topic for students unfamiliar with low level target languages. We present Frances, a tool for helping students understand code generation and low level languages. The key idea is to graphically illustrate the relationships between high level language constructs and low level (assembly) language code. By illustrating these relationships, we take advantage of the students existing understanding of some high level language. We have used Frances in a compiler design course and received highly positive feedback. Students conveyed to us that Frances significantly helped them to understand the concepts necessary to implement code generation in a compiler project.

Categories and Subject Descriptors

K.3.0 [**Computers and Education**]: General; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education, curriculum*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Education, Languages

Keywords

Frances, Code Generation, Compilers, Visualization

1. INTRODUCTION

A compiler is a software system designed to translate programs written in a source language (usually a high level language), to another target language (frequently machine code) [12]. Developing a compiler is a difficult but rewarding experience for students since it requires a wide range of knowledge and techniques [8]. Compiler and language implementation has always been listed in the ACM Computing Curricula (CC) as a main curriculum topic [16]. The latest revision of CC '01 also states the importance of these topics: "...good compiler writers are often seen as desirable; they tend to be good software engineers [2, pp.11]."

Unfortunately, the complexity of compilers makes it difficult to sufficiently cover the necessary concepts and construct a compiler in a single semester [8]. One problem is that students must have a thorough understanding of the high level (source) language and the target language [8]. Most students at this point in their education have learned at least one, if not several, high level languages (Java, C++, etc). However, many students have not learned assembly like languages commonly used as compiler target languages. In recent years assembly language courses have been supplanted by other topics in many undergraduate curricula [11]. This significantly complicates the task of writing the compiler, specifically, the portions dealing with code generation and optimizations.

Our contribution is a tool called *Frances*[1] which has the primary use of assisting in the learning of code generation [3]. The main intuition of our approach is to take advantage of the students existing knowledge of some high level language. We allow students to enter source code in a language of their choice, then show a graphical representation of the corresponding target code. This graphical representation is shown in a way that allows users to quickly identify how different types of high level language features are represented in a lower level language. Our representation makes use of several techniques to improve understanding, most importantly we,

- maintain actual target code **ordering**,
- show different types of possible run-time **paths**, and
- **color code** types of instruction blocks.

Frances provides a simple graphical interface that helps to reduce the burden of learning to use the tool.

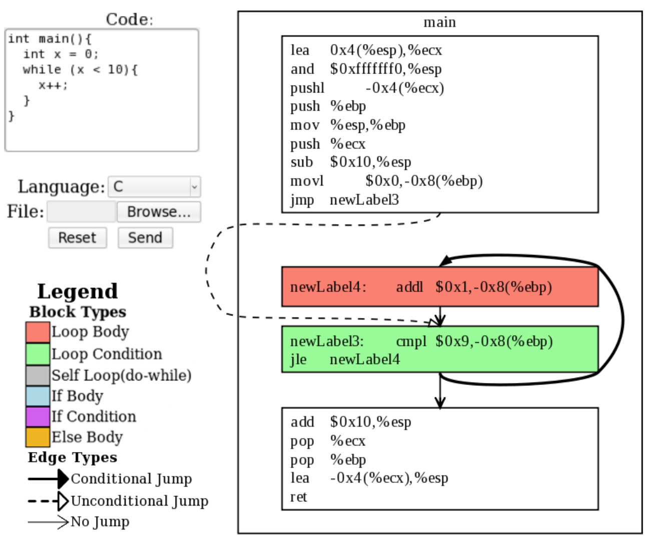
The key benefits of Frances is that it helps students

- understand **code generation**, and
- gain familiarity with **assembly language** .

This tool helps students understand code generation and assembly language by showing how high level language source code translates to assembly language. Additionally, Frances generates graphs which clearly illustrate the purpose of code segments. This greatly improves the understanding of the target code. Naps *et al.* claim

[1]We named the tool Frances in honor of Frances E. Allen . She received the Turning award for pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.

that in computer science education, visualization techniques are more beneficial when they engage students [15]. Our approach is not only visual, but allows students to experiment with program code in a variety of high level languages. A small example is shown in Figure 1 where the assembly language equivalent of a small loop in the high level language C is graphically illustrated in Frances.



```
int main(){
  int x = 0;
  while (x < 10){
    x++;
  }
}
```

```
lea   0x4(%esp),%ecx
and   $0xfffffff0,%esp
pushl      -0x4(%ecx)
push  %ebp
mov   %esp,%ebp
push  %ecx
sub   $0x10,%esp
movl       $0x0,-0x8(%ebp)
jmp   newLabel3
```

```
newLabel4:     addl  $0x1,-0x8(%ebp)
```

```
newLabel3:     cmpl  $0x9,-0x8(%ebp)
jle    newLabel4
```

```
add   $0x10,%esp
pop   %ecx
pop   %ebp
lea   -0x4(%ecx),%esp
ret
```

Figure 1: A simple C while loop

We have used this tool in an undergraduate compiler design course with encouraging results. The students in this course were given the opportunity to use Frances when learning the topics of code generation. These students communicated that the tool was very useful for understanding concepts necessary for code generation. Based on this experience, Frances is becoming a more integral part of future offerings of this course.

The rest of this paper is organized as follows. Section 2 gives a brief overview of related work. Then, Section 3 outlines the goals of Frances. Next, in Section 4, we describe the Frances tool. Section 5 discusses our experiences and observations from using the Frances tool as part of a compiler design course. Finally, Section 6 concludes and discusses future work.

2. RELATED WORK

Since developing a compiler is difficult, especially within a single semester course, a large body of work has been done to improve this process. Aiken presented Cool, a language and compiler designed for course projects to reduce the overhead for the instructor and keep assignments modular [5]. Similarly, Corliss *et al.* developed Bantam which is a Java compiler project for courses [8]. Modularity is also achieved in Bantam since components of the compiler can use the provided modules, or be swapped out with custom versions. Rather than developing a new infrastructure, our technique is complementary to these existing techniques in order to help understand specific portions of compiler implementation.

Resler *et al.* propose a visualization tool, VCOCO, for understanding compilers [13]. VCOCO provides several view panes which show source code, language grammar, compiler, parser, and scanner. Each pane is updated throughout the compilation process. We also propose a visual approach, however, we are interested specifically with code generation and present a graphical approach.

Bredlau *et al.* suggest using the Java Virtual Machine (JVM) for teaching assembly [7]. The idea is to let the java compiler create JVM code which is compared to the source code. We take a similar approach but with assembly language and we provide a simple graphical comparison to aid in this process.

3. GOALS FOR FRANCES

In this section, we discuss the goals we had in mind when developing the Frances tool as well as details for accomplishing these goals. Briefly, these goals include making code generation easy to understand by clearly and quickly showing how familiar high level language constructs translate to low level language code. Additionally, the tool is made as easy as possible, making the learning curve for the tool minimal.

3.1 Understanding Code Generation

The first and most important goal, was to help students understand the concepts behind code generation.

We have observed that for many students, code generation can be the most difficult challenge when writing a compiler for the first time. This is largely because of the differences in already familiar high level languages and unfamiliar low level languages. This includes differences in syntax as well as the ordering of statements related to the various programming constructs. For example, in Figure 1 the order of the loop condition and loop body are opposite in the two representations. With this tool, we intend to ease this challenge by clearly showing how familiar high-level language code and constructs map to target, low-level assembly code. The idea is that students already understand at least one high level language. By understanding how the high level and low level languages relate to each other, students can quickly understand how to translate from one language to the other. Then when students write their own compiler they can use Frances as a guide for dealing with various high level constructs including memory management and code order.

A simple way to do this is to have students compare source code with equivalent assembly code (for example, gcc can generate assembly from source). The benefit of this is that it allows students to see what types of instructions their code is mapped to as well as ordering of instructions. This is a helpful process, however, we felt that more could be done to improve this process especially for larger programs. To improve this process, we show the source to target language mapping and improve upon this mapping in two ways. First, we represent target code as a graph that shows execution paths that may be taken at run-time. Second, we color code this graph to quickly show how control structures are represented in the target language. Most students are familiar with high level languages constructs. Thus, being able to quickly identify how language constructs in familiar high level languages map to assembly code significantly eases this comparison process. The details of these features and how they work are described in Section 4.

3.2 Ease of Use

We felt it was important for the tool to be as easy to use as possible. If not, the cost of learning how to use the tool could easily overshadow the benefits it provides. Therefore, we take several steps to make the tool easy to use: make it easy to run on a wide variety of platforms, provide a simple interface, and support a variety of high level or source languages.

First, to avoid issues with different operating systems, hardware platforms, software versions etc., we make the tool available via the web. Thus, we eliminate the need for users to build Frances on their machine. This removes any problems that may arise when building a software package from source and allows any user to make use of Frances as long as they have a web browser.

Second, to make the tool easy to use, we provide a simple graphical interface. Many compiler tools operate from the command line and require the users to learn complicated syntax. While powerful and flexible, learning how to use such a tool if you are only planning on using it for a short period of time is undesirable. We believe our interface is simple enough for users to immediately begin using it and understanding the output.

Third, it is required that users are familiar with some high level language. Restricting the tool to a single language would clearly not be useful for users not familiar with this language. Therefore, we provide support for several common high level languages (C, C++, and Fortran).

4. FRANCES

We now discuss the details of the Frances tool. This includes what facilities the tool provides to students as well as how it provides these facilities. Additionally, we give detailed explanations of why certain approaches are taken. A general discussion of why these facilities are provided is included in Section 3.

The core of Frances is a framework built for another research project. This framework makes use of the GNU Binutils [9] for converting an executable to an object oriented representation that we can analyze, modify, and output as a new executable. This framework is not yet publicly available, but will be released as an open source project in the future. Until then, the tool is made available as a web service.

Frances generates a simple graphical representation of the target code corresponding to the source code. For example, in Figure 1, this simple while loop is shown graphically as four blocks of code. Furthermore, the edges or *paths* between these blocks that can be taken at run-time are shown. To generate the graphical program representation, we make use of `dot` which is part of the GraphViz [10] graph visualization software. We now describe the major components of Frances including how blocks and edges are drawn as well as a brief discussion about the interface.

4.1 Blocks

Basic blocks of instructions are generated by Frances. A basic block is a sequence of instructions with a single entry point and single exit point with no jumps between [6]. For simple control structures (non-nested structures) basic blocks capture the main components of the structures. For example, in Figure 1, the sample while loop can be divided into two parts which have a different purpose: the *loop body*, and the *loop condition*. Therefore, the graphical version of the target code illustrates these two components of the loop in two separate blocks. Additionally, the blocks before and after the loop are also shown separately.

A major difference between previous tools and our tool is the way in which we lay out blocks. Similar tools [1, 4, 14] represent blocks as a flow chart. Since our major goal was to help students understand code generation, we make this graphical representation as close as possible to real generated code. We do this by maintaining the instruction ordering of the actual target code. This includes the ordering of the blocks. To make this ordering clear, we represent blocks in a linear fashion in the same way that programs are represented in target code.

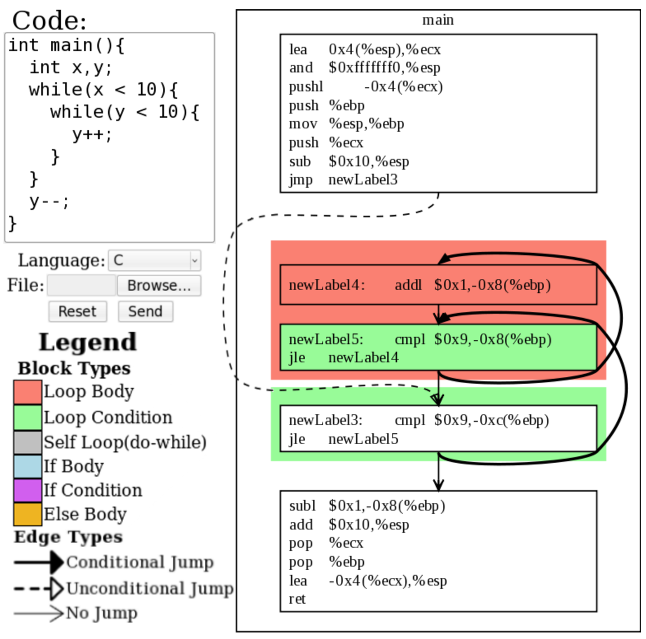
For example, in Figure 1, the layout of blocks is not done in a way that is immediately obvious from the source code. Consider the loop condition. In the source code, this is before the loop body whereas in the target code, it is after the loop body. This is not immediately clear; however, this is how target code is generated by the compiler. Thus, understanding this ordering is necessary for understanding code generation. Therefore, our tool exposes students to such orderings. Given that this ordering is confusing, we take steps to help clarify this ordering.

4.2 Color Coding

Following that students are already familiar with a high level language, we aim to quickly and clearly illustrate how control structures in a high level language are represented in the target language. We show this by coloring the graph to highlight the different parts of the various control structures. Our tool performs simple control flow analyses [12] on the code to determine the different parts of the control structures. Then, the tool colors blocks based on which part of the control structure they make up (loop condition, loop body, etc). For example, in Figure 1 a while loop is shown in both forms. Both the loop condition and the loop body are colored differently to make it easy to distinguish between the two. As mentioned previously, the ordering of these two blocks is confusing at first since it differs from the source code ordering. This coloring quickly points out this ordering by showing that for this high level language while loop, the loop condition goes after the loop body.

For simple control structures, that is, non-nested control structures, we shade the background of the blocks to corresponding colors for each part of control structures. This includes structures such as loops (loop body and loop condition are colored differently), *if/else* blocks, etc. As discussed previously, Figure 1 shows this for a simple while loop.

For nested control structures, the coloring must be done differently. We start by shading the blocks in the innermost structures as described previously. Then, all other structures are surrounded with boxes. These boxes are then shaded to show what kind of structure the member blocks are a part. Furthermore, this helps to show how the different structures interact. For example, consider the nested loop in Figure 2. The inner loop is composed of two blocks, the loop condition and the loop body. These two blocks are shaded in the figure. We can see that both of these blocks (the entire inner loop) are contained within another structure since they are contained in a larger shaded box. This structure is the loop body of the outer loop which is shown clearly by the shading.



Code:
```
int main(){
  int x,y;
  while(x < 10){
    while(y < 10){
      y++;
    }
  }
  y--;
}
```

Language: C
File:

Legend

Block Types

🟥	Loop Body
🟩	Loop Condition
⬜	Self Loop(do-while)
🟦	If Body
🟪	If Condition
🟧	Else Body

Edge Types

➔	Conditional Jump
- - ▷	Unconditional Jump
→	No Jump

main
```
lea     0x4(%esp),%ecx
and     $0xfffffff0,%esp
pushl           -0x4(%ecx)
push    %ebp
mov     %esp,%ebp
push    %ecx
sub     $0x10,%esp
jmp     newLabel3

newLabel4:      addl  $0x1,-0x8(%ebp)

newLabel5:      cmpl  $0x9,-0x8(%ebp)
jle     newLabel4

newLabel3:      cmpl  $0x9,-0xc(%ebp)
jle     newLabel5

subl    $0x1,-0x8(%ebp)
add     $0x10,%esp
pop     %ecx
pop     %ebp
lea     -0x4(%ecx),%esp
ret
```

Figure 2: A simple nested while loop

This drawing of blocks shows how target code is laid out. Then the coloring helps to quickly show how components in familiar high level code are represented in the target code. Furthermore, by breaking this representation down, we can focus on a smaller subset of the code. Next, we describe how edges are illustrated.

4.3 Paths

The edges between blocks represent the *paths* that can be taken at run-time. A jump in the target code can have up to two possible next instructions. The paths show what these possible next instructions are. For example, in Figure 1, we see that the loop condition (shown in green) has two outgoing paths: one edge leading to the loop body if the condition is true and one edge leading to the next block after the loop (exiting the loop) if the condition is false.

In combination with blocks, edges help the user see how different structures are represented. For example, consider the first block in Figure 1. This figure illustrates how, in the target code, you first jump past the loop body to the loop condition for this type of loop. This illustrate a key difference between `while` and `do-while` loops since `do-while` loops are not organized this way.

As mentioned previously, the instruction (and block) ordering in the target code can be confusing to students because it is frequently different than the source code ordering. Our graphical representation of blocks helps by highlighting the components of the different control structures. Illustrating the ordering of control structures is helpful, however, we still need to show execution flows between these structures. Figure 2 shows an example of a nested loop where paths help to illustrate the initially confusing code layout. In this figure, we see that the edge corresponding to entering the inner loop actually goes to the second block in the inner loop. This is slightly confusing at first since the path does not go to the beginning of the inner loop code. This example shows that it is important to understand how execution enters and exits loops. Furthermore, understanding how execution flows through others structures such as `if` /`else` blocks is also important. Thus, we take steps to help contrast the differences between edges.

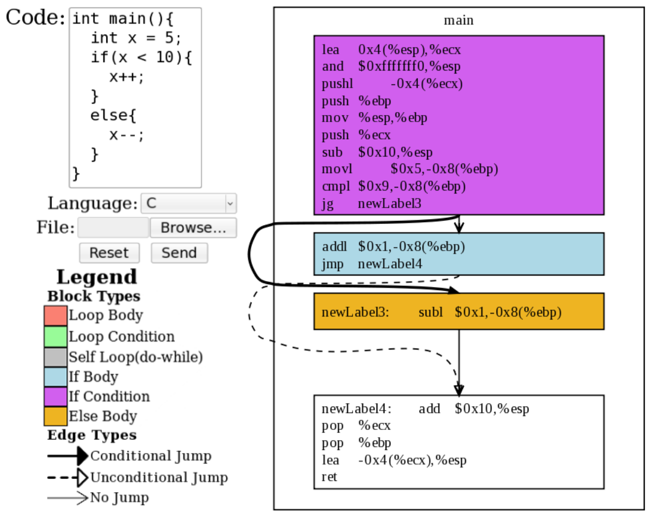
4.4 Edge Types

There are multiple types of edges. We illustrate the different types by using different styles of lines and arrowheads for drawing the edges. For example, in Figure 1, we see all three different types of edges. We now give a brief description of each edge type.

- First, we have *"unconditional jumps"*. In the figure, this jump is illustrated with a dashed line and an empty triangular arrowhead. In Figure 1, the first block ends with the instruction `jmp newLabel3`. With this type of jump, the path is taken no matter what when the instruction is executed.

- Next, we have *"fall through"* or *"branch not taken"* edges. This edge type is illustrated with a thin edge and a "wedge shaped" arrowhead. This edge type refers to when we simply go to the next sequential instruction when either the current instruction is not a jump or a condition is false. For example, the edge going from the loop body to the loop condition in the figure. In this case, since the block does not end with a jump, the next instruction is just the next sequential block. Another example of this type of edge is the edge from the loop condition to the last block in Figure 1. This edge is taken when the condition on the jump, in this case `$0x9 >= -0x8(%ebp)`, is false. This may seem trivial since the *"branch not taken"* edge is always the edge to the next sequential block, however, for students just learning this concept may not be immediately obvious.

- Finally, we have *"branch taken"* edges. These edges are drawn with a thick solid line and a solid triangular arrowhead. These edges are those which are taken when a jump condition is true. For example, in Figure 1, we have an edge from the loop condition block to the loop body block. This edge is taken whenever the loop condition on the loop is true. Another example is shown in Figure 3. In this example we can see a branch taken edge from the *if condition* block to the *else body* block. This edge is taken whenever the condition is true, however, in the source version, we have that this edge is taken whenever the if condition is false. This is another interesting difference between source and target code which is nicely illustrated in Frances.


Code:
```
int main(){
  int x = 5;
  if(x < 10){
    x++;
  }
  else{
    x--;
  }
}
```
Language: C

```
main

lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl        -0x4(%ecx)
push   %ebp
mov    %esp,%ebp
push   %ecx
sub    $0x10,%esp
movl         $0x5,-0x8(%ebp)
cmpl   $0x9,-0x8(%ebp)
jg     newLabel3

addl   $0x1,-0x8(%ebp)
jmp    newLabel4

newLabel3:      subl   $0x1,-0x8(%ebp)

newLabel4:      add    $0x10,%esp
pop    %ecx
pop    %ebp
lea    -0x4(%ecx),%esp
ret
```

Figure 3: if-else block

This edge drawing helps illustrate the finer details of the target code. This includes how individual instructions such as jumps are created and how complex control structure components interact such as nested loops. Together with our block drawing and coloring, Frances generates informative and easy to understand figures which illustrate how code generation is performed.

4.5 Interface

To make Frances as easy to use as possible, we make it available via a web interface. Since we make it available through a web interface, it does not require installation and thus avoids compatibility problems. This simple and easy to use interface is available at http://www.cs.iastate.edu/˜sapha/tools/frances/.

Since we can not be sure what high level languages each user is familiar with, we give users the option of writing code in a variety of high level languages. The language is selected in a simple drop down list. Figure 1 shows an example where the C language is selected.

To enter code, users can type into a text box as part of the web interface of Frances. Since a user may not want to enter all the code for each example in the web interface, users have the option to upload a file. The input code is shown side-by-side with the graphical representation generated by Frances. Because of space restrictions the figures in this paper show a trimmed version of this interface.

By not requiring installation, interfacing with a wide variety of high level languages, and giving users options for inputting code, we have a tool which is easily accessible to a wide range of users.

Summary of Representation: We believe that Frances' block layout and coloring in combination with the edge drawing greatly helps to teach the instruction layout of low level language code. Furthermore, we believe that when viewed alongside familiar source code, this representation makes the process of understanding translating between the two languages significantly easier. With its simple and easy to use interface we believe Frances is easy to use in a course, will help students understand these difficult concepts, and save valuable course time for other topics.

5. EXPERIENCES AND DISCUSSION

Our first uses of Frances in an undergraduate compilers course gave very positive results. Initially, Frances introduced students to fundamental assembly language concepts through demonstration. The demonstration consisted of a simple program with only variable declaration and a single if-statement. This simple program demonstrated the initialization of registers, memory allocation and a few simple assembly instructions. Prior to this most students had little or no exposure to actual assembly language code.

As stated previously, the order of instructions in high level languages contrasted with the generated low level language is foreign to most students. Often, students that only have exposure to high-level languages have difficulty understanding the relationship between the source and target code. The graphical features of Frances help to close this gap in understanding by providing a side by side comparison with color coded identification of control structure components. The visualization techniques used in Frances allow students to quickly assimilate how the assembly is accomplishing the implementation of the source code.

Several course exercises, available on the Frances website, have been developed that allow students to experiment with Frances. The exercises are developed to demonstrate each basic control structure. Additionally, the nesting of structures provides a deeper understanding of why it is important for the assembly code to be generated in the given order. Other course materials demonstrate more concepts. Memory allocation for variables, including arrays, provides the student insights into how space requirements of a program are utilized. Additionally, materials have been created to demonstrate details of transferring control to functions.

As part of the curriculum of the compiler course, students are required to build a compiler for a language they create. The compiler is written directly in Java or C++. The use of Frances allows students to gain an understanding of how and why to generate code for given control structures and function calls. This has allowed the pace of the course to significantly increase. Students questioning how to program the code generation for a particular construct simply go to Frances and get a direct demonstration.

6. CONCLUSION AND FUTURE WORK

Compiler design courses are an integral component to most Computer Science curricula [2]. Prior to this course, in much of today's curriculum, students have limited exposure to low level languages. This makes the code generation components of these compiler design courses particularly challenging. In this paper, we presented Frances, a tool to aid in the understanding of code generation and compiler construction. The main goal of this tool is to help students understand the relation between familiar high level languages and not so familiar low level target languages. This is accomplished by giving a side by side comparison with graphical cues of the target language. The graphical representation includes color coded target code to highlight control structures and execution paths. This representation also maintains the actual order of instructions of the compiled version of the program. Frances has shown to be highly useful in practice by significantly easing the process of teaching code generation concepts.

As part of future work we plan to extend the tool in the direction of illustrating and understanding program analysis techniques and automatic compiler optimizations. We plan to have the option in Frances to illustrate a variety program analysis techniques (control flow, data flow, etc). This will help illustrate where optimizations are applicable and how optimizations are performed. For optimizations, the plan is to have the ability to illustrate common optimizations step by step using similar figures as those used to illustrate code generation techniques shown in this paper. We plan to implement a wide range of common optimizations as well as a hot-spot finder to illustrate which portions of the program are in most need of optimization. Along with making common techniques available, we also plan to allow users to write their own analysis and optimization routines. Frances is a powerful tool for a basic compilers course and we believe these enhancements will extend its use to advanced undergraduate and graduate level compiler and programming language courses.

Acknowledgments

The authors would like to thank the students who tested the tool as well as the anonymous reviewers for their comments and suggestions. Sondag and Rajan are supported in part by the US National Science Foundation under grants 06-27354 and 08-08913.

7. REFERENCES

[1] *aiSee - Graph Visualization*. http://www.absint.com/aisee/.
[2] Computing curricula 2008: An interim revision of cs 2001. http://www.acm.org/education/curricula/ComputerScience2008.pdf.
[3] *Frances: Control FLow Graph Generator*. http://www.cs.iastate.edu/~sapha/tools/frances/.
[4] *ICD-C Compiler Framework*. http://www.icd.de/es/icd-c/.
[5] A. Aiken. Cool: a portable project for teaching compiler construction. *SIGPLAN Not.*, 31(7):19–24, 1996.
[6] F. E. Allen. Control flow analysis. In *Symposium on Compiler optimization*, pages 1–19, 1970.
[7] C. Bredlau and D. Deremer. Assembly language through the java virtual machine. In *SIGCSE*, 2001.
[8] M. L. Corliss and E. C. Lewis. Bantam: a customizable, java-based, classroom compiler. In *SIGCSE*, 2008.
[9] Free Software Foundation. Gnu binutils: a collection of binary tools, 2009. http://www.gnu.org/software/binutils/.
[10] J. Ellson *et al*. Graphviz - open source graph drawing tools. *Graph Drawing*, 2001.
[11] M. C. Loui. The case for assembly language programming. *IEEE Transactions on Education*, E-31(3):160–164, 1988.
[12] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Academic Press, 1997.
[13] R. D. Resler and D. M. Deaver. Vcoco: a visualisation tool for teaching compilers. *SIGCSE Bull.*, 30(3):199–202, 1998.
[14] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks (IPSN)*, 2005.
[15] T.L. Naps *et al*. Exploring the role of visualization and engage- ment in computer science education. In *ITiCSE-WGR*, 2002.
[16] L. Xu. Language engineering in the context of a popular, inexpensive robot platform. In *SIGCSE*, 2008.