

Web Turtle Compiler and Interpreter using Coco/R

IKT408 Project

June 6, 2008



Christian Auby
Thomas Jager
Torbjørn Skagestad

Abstract

In this report we describe our efforts to create a Webturtle Compiler and interpreter using the compiler generator Coco/R.

Contents

1	Background	3
1.1	LOGO	3
1.2	Turtle Graphics / Turtle Geometry	3
1.3	Web Turtle	3
1.4	Compiler Generator	4
1.5	Coco/R	4
1.5.1	Scanner	4
1.5.2	Parser	4
1.5.3	Attributed grammar	5
1.6	devkitPro	5
1.7	SDL	5
2	Project description	6
3	Implementation	6
3.1	Early Experiments	6
3.2	Compiler	8
3.3	Designing a stack machine	9
3.4	General purpose instructions	10
3.5	Special purpose instructions	10
3.6	Interpreter	11
3.7	Test suite	12
4	Discussion and further work	13
4.1	Difficulties	13
4.2	Future work	14

List of Figures

1	Extended Backus-Naur Form grammar	5
2	Successfull run of square test	7
3	Recursive Web Turtle programs	7
4	Compiler generation process.	8
5	Compiling Web Turtle source code into byte code	9
6	Interpreting byte code and generating the result	11
7	Interpreter running on Nintendo DS	12
8	Test output	13

1 Background

1.1 LOGO

Logo is a computer programming language used for functional programming. Today, it is known mainly for its turtle graphics, but it also has significant facilities for handling lists, files, I/O, and recursion.

Logo was created for educational use, more so for constructivist teaching, by Daniel G. Bobrow, Wally Feurzeig and Seymour Papert. It can be used to teach most computer science concepts, as UC Berkeley Lecturer Brian Harvey does in his Computer Science Logo Style trilogy.[2]

1.2 Turtle Graphics / Turtle Geometry

Turtle graphics is a method of programming vector graphics using a relative cursor (the "turtle") upon a plane. This geometry describes paths "from within" rather than "from outside" or "from above." For example, "turn right" means turn right relative to whatever direction you were heading before; by contrast, "turn east" specifies an apparently absolute direction.

The turtle has three attributes:

1. Position
2. Orientation
3. Pen (With attributes like color, width)

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it, setting its color, or setting its width. From these building blocks one can build more complex shapes like squares, triangles, circles and other composite figures. Combined with flow control, procedures, and recursion, Turtle Graphics can also generate fractals.[2]

1.3 Web Turtle

Web Turtle takes the drawing system from LOGO / Turtle Graphics and implements it in a more traditional programming language. It is presented as a web based tool where the user writes code to move the turtle, and when submitted the code is executed on the server side of Web Turtle, displaying the results as an image. The language specifications for Web Turtle were designed by Bill Kendrick of Tux Paint fame.[1]

1.4 Compiler Generator

Compiler generators, also known as compiler compilers, are programs that generate the source code of a scanner, parser and possibly other parts of the compiler from a programming language description. In the most general case, it takes a full machine-independent syntactic and semantic description of a programming language, along with a full language-independent description of a target instruction set architecture, and generates a compiler.

1.5 Coco/R

Coco/R is a compiler generator written by a group of people from the Johannes Kepler University of Linz, Austria. It provides a grammar language to define the syntax of the source language. By parsing this language definition it generates a scanner and a parser. We have chosen Coco/R as it was recommended to us by an individual in an on-line discussion, and as we looked at it we felt it would suit our needs without being too complicated. Coco/R supports several target languages, among such as C# and Java. The generated output will be in this language, as well as the language/machine specific code we supply to the generator. The Parts that must be supplied by us are the symbol table and the code generator. [5]

1.5.1 Scanner

The scanner provided by Coco/R is a deterministic finite automaton.[3] The scanner is generated in the language the user specifies, and is built from the grammar definition stored in a ATG file. As the scanner reads one symbol at the time, semantic errors are detected exactly where they occur, giving the user of your compiler very helpful error messages. Features such as these would be difficult to provide with a traditional built from the ground up approach. Although it is possible to separate the scanner and the tokenizer, the scanner generated by Coco/R implements both of these.

1.5.2 Parser

The parser uses recursive descent, built with a set of mutually-recursive functions.[3] As the parser only uses one look-ahead token by default, there might be conflicts as a result of this. These can be resolved by additional multi-symbol look-ups, or semantic checks. In the case of our attributed grammar there was no need for us to do conflict resolution.

```
digit = "0123456789" .  
number = digit | { digit } .
```

Figure 1: Extended Backus-Naur Form grammar

1.5.3 Attributed grammar

The attributed grammar, a manually written ATG file, is written in Extended Backus-Naur Form (EBNF) . EBNF is used to describe a set of rules, used by the tokenizer. A simple example of this would be, in the style of Coco/R:

The first line describes a set of characters that make a digit. This alone would match only the characters listed; no repeats and no other characters. The second uses the first, requiring a digit, and allowing none or more digits following it. Thus matching 1, 22, 323, 1123 and so on. Using the simple rules of EBNF complex grammar can be described. Coco/R splits these in three categories; Characters, as seen by digit, Tokens, as seen by number, and Productions. Productions are more complex rules that can have arguments and generate code. From our experience the main task of writing the attributed grammar will be spent on the productions.

1.6 devkitPro

devkitPro is a set of compilers and libraries for embedded consoles, such as Nintendo DS, Nintendo Wii and Sony PlayStation Portable. [6]It provides a feature rich stable environment for building applications targeting these platforms. devkitPPC was used for Wii and devkitARM was used for the DS port of our interpreter, and selected on the basis of being the only actively developed tool chain to use with the consoles we have chosen. Previous experiences with devkitPro also contributed to the choice.

The tool chains are free of charge and do not contain any copyrighted code. The compiler is based on GCC and maintained by Dave Murphy, with additional contributors. The libraries are written in C and assembly, and are maintained by several enthusiast developers.

1.7 SDL

Simple DirectMedia Layer is a library for graphics, sound and input for a wide range of systems. [7] The C++ interpreter uses SDL for graphics, through a hardware abstraction in the actual interpreter. This allowed us to develop this interpreter efficiently, debugging on PC which has extensive debugging features, and once working, porting the C++ code to the other targets. Note that the console ports do not use SDL but the graphics hardware of the target system directly.

2 Project description

Our project is to create grammar suitable for the Coco/R compiler-compiler. With this grammar in place Coco/R can create a compiler that can convert Web Turtle code into a binary code that's easier to interpret. We have decided to implement the following Web Turtle commands in the compiler:

- Draw
- Right, Left
- Repeat
- Let
- Color
- Go, Return
- Remember, Goback
- Home
- Push, Pop

Some of these will require us to implement stack allocation in the compiler, as well as expression evaluation. The output of the compiler will be a binary format for a self-designed CPU instruction set. The output from the compiler is a binary intermediate code that we design ourselves. This code will be parsed by the interpreter to perform the actual drawing. Two of the lesser used functions in Turtle Graphics are if/else and user input. Due to the limited amount of Web Turtle scripts that use these we have decided to not include them in our requirements. If we feel we have time to implement them near the end we will reevaluate this decision. The original requirements can be found in Appendix B.

3 Implementation

3.1 Early Experiments

The first version of our compiler was written from scratch without the help from Coco/R, and was mostly feature complete. Due to the request from our mentor we decided to follow his advice and use a Compiler-Compiler instead, and generate the scanner and parser as opposed to writing all of it. The manual compiler was hard to maintain, especially when adding new instructions and debugging errors.

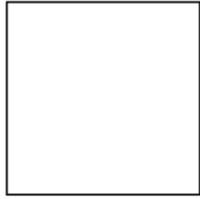


Figure 2: Successful run of square test

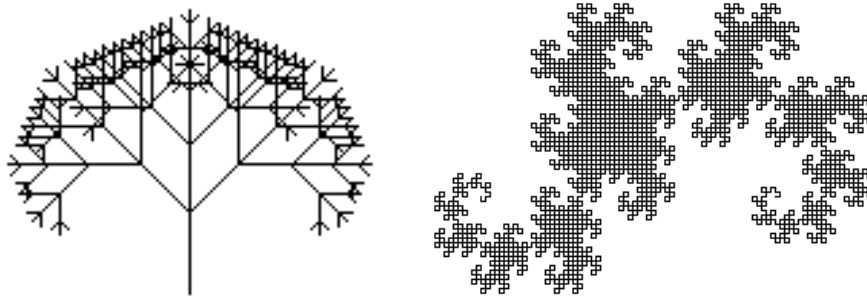


Figure 3: Recursive Web Turtle programs

From a user perspective the old compiler was very hard to use. If the source code being compiled had any errors the compiler hardly ever knew what was wrong, or was able to tell the user where the error occurred. Adding these features would mean a ton of extra work, compared to *Coco/R* where you get them automatically.

3.2 Compiler

We have chosen *C#* for our compiler language, and although it does not offer the best performance our priorities lie with ease of use and cross platform compatibility. *C#* will let us develop the compiler at a rapid pace using functionality from the vast *.NET* library.

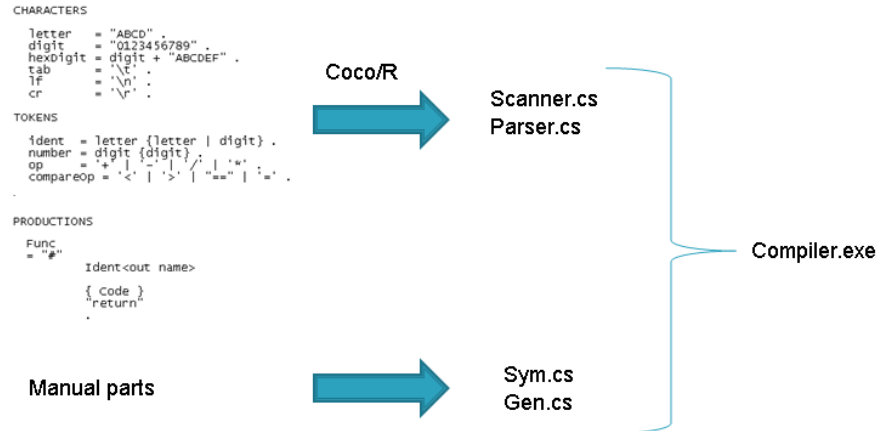


Figure 4: Compiler generation process.

Coco/R uses our attributed grammar and generates the scanner and parser for us. This however does not output any code that generates machine instructions by default, as the instruction set is machine specific. To compile concrete code the productions in the annotated grammar must have semantic actions using the code generator, as displayed in Figure 4. Memory allocation is handled by the external symbol table. These four files are compiled together using *csc.exe* (*C-Sharp Compiler*) from *Visual Studio*, to produce our final compiler executable.

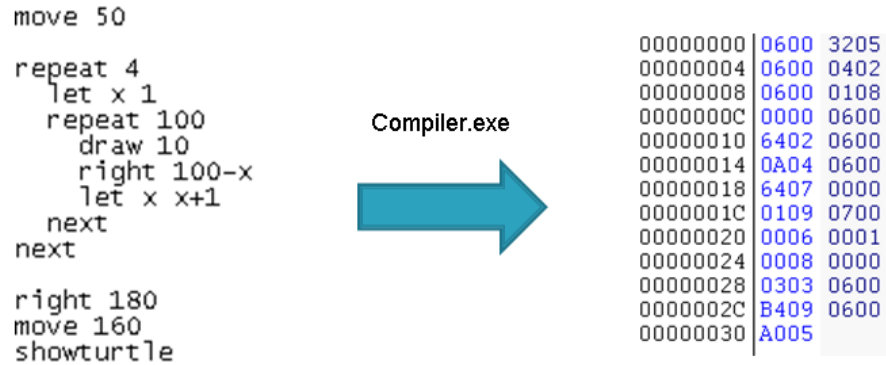


Figure 5: Compiling Web Turtle source code into byte code

The generated Coco/R scanner and parser reads the source code and using the symbol table allocates memory for variables. As our compiler is a one pass compiler it simultaneously generates the byte code using the code generator supplied by us, targeting our specific instruction set.

Originally we did not intend to implement if/else statements, as mentioned under project description. Once development got started we decided to implement them anyway. Because of this some of the more advanced examples also compile and run.

3.3 Designing a stack machine

We wanted the output of our compiler to mean something, that is, to actually do something. This would require a target instead of a dummy implementation, and although targeting x86 or another known hardware architecture would have been interesting, the scope of such an implementation would go way beyond the time allotted to this course. The solution was simple, and something we had thought about even before starting: design our own machine architecture and target this in our compiler.

A stack machine is a machine that works on one or more stacks to perform most of it's operations. The input to a stack machine is the initial content of stack 1; all the other stacks start empty. Each state of a stack machine is either a read state or a write state; and each state specifies a stack number to read (pop) from, or write (push) onto. In addition, a write state specifies the symbol to write, and the next state to transition to. A read state specifies, for each symbol in the alphabet, what state it would transition to if that symbol were read; in addition, it also specifies what state to transition to if the stack were empty. A stack machine halts when it transitions into a special halting state. Due to this

the instruction set is compact, as most instructions do not have any operands but operate on the top most value(s) of the stack implicitly. This way the job of designing the instruction set is made easier, as most will only be a single byte. Some simple abstract stack machines can work with a single stack. In our case the machine will be running compiled Web Turtle applications, and as such a more specialized design makes more sense. Some features of Web Turtle is easiest to implement using additional stacks, and as a result the machine we have designed mimics this.

Our stack machine is reminiscent of the Java virtual machine, in that it allows a single compiled binary to run on many platforms.

As a side note we can mention that a stack machine with multiple stacks is equivalent to a Turing machine. For example, a 2-stack machine can emulate a Turing Machine by using one stack for the tape portion to the left of the Turing Machine's current head position and the other stack for the portion to the right.[5]

3.4 General purpose instructions

Even though the machine will be specialized, it will still need a set of instructions found in most modern CPUs. These are instructions unrelated to any Turtle Graphics feature, but required for computations, program flow and so on. These instructions include add (for adding numbers together), sub (subtracting numbers), mul (multiply numbers), div (divide numbers), neg (negate) and several others are in this group of general purpose instructions. See the instruction reference for details in Appendix A.

3.5 Special purpose instructions

Providing a code library written for our stack machine that implements line drawing, angle calculations and other complex library functions would be difficult, and in some cases, most likely impossible. As a specialized machine, we can decide to implement these in hardware. Instructions that perform complex tasks otherwise given to a code library include move and draw, which do line calculations. Repeat which uses the repeat stack to support recursive functions that use repeat, and “remember”, which puts the current turtle location and angle on a remember stack. The set of special purpose instructions represent most of the Turtle Graphics specific functionality and takes up most of the code in the interpreter. Note that this also makes the code generator easier to write; there is no extra code to generate for complex operations as they are still written by the code generator as a single instruction.

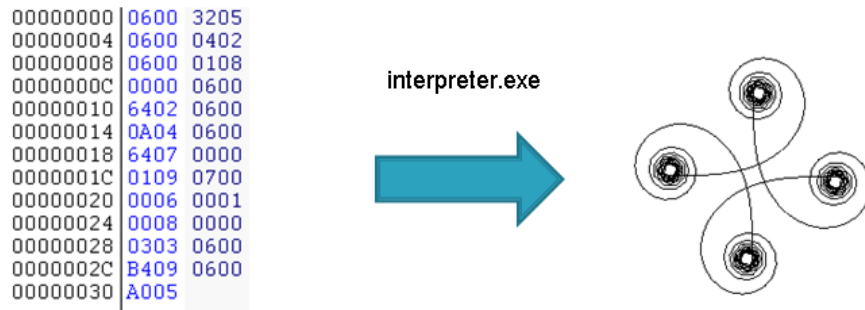


Figure 6: Interpreting byte code and generating the result

3.6 Interpreter

To actually use the generated binary we had to implement our specialized stack machine. A hardware approach would have been very interesting, but hardly time efficient, and also harder to debug. A software solution was the obvious approach; implement an interpreter in the same language as the target language of the compiler, and have this interpreter store the generated image in a file on disk. This also helps debugging the actual code generator, as following the code in an interpreter is easier than reading the binary to look for errors. The interpreter starts by opening the generated .bin file, and reads a byte. Using the same enumeration as the compiler it identifies the current instruction, and performs the function of this instruction. This continues until the last byte is read. If there is an error in the compiled binary the error will be executed, and the state of the interpreter will be erroneous; this is the default behavior of most execution environments, as doing code quality checks is not the responsibility of the host. Capturing the output makes it possible to compare the results with the Web Turtle official results, providing a nice and easy way to check for bugs as well as being very satisfying visually.

Our reference interpreter is written in C# as a part of the compiler project, but to illustrate the advantages of compiling to a binary format another interpreter was written in C++. There are several target platforms possible when the interpreter is written in C++.

The interpreter uses the bresenham line drawing algorithm, through an implementation written by Kenny Hoff. This implementation does not come with any licence at all, and is used by the SDL, Wii and DS ports.

Even a simple example shown in Figure 2 requires that the scanner, parser, code generator, symbol table and interpreter are in a working state.

After initial development was done on the C++ interpreter, which uses SDL and runs on Windows / Linux, we started work on the console ports. All the C++

based interpreters share the same code base, but require specific initialization and video code. The Nintendo Wii and Nintendo DS ports use GCC and also requires a makefile to target each platform. On both cases we use makefiles provided by devkitPro.



Figure 7: Interpreter running on Nintendo DS

3.7 Test suite

Once the compiler and interpreter started compiling and running simple demos it became quite apparent we needed a better way to check for bugs without manually trying the examples every time something was changed. A test application was written that takes all the source files it finds in a certain folder, compiles them, runs them in the interpreter and stores the interpreter output as a png graphics file with the same name as the source code it comes from. Once this is done it generates a HTML document with image links to the reference image as well as the result from our implementation. Opening this file displays the test results as side by side comparisons.

The test application can be easily launched after every new build of the compiler and/or interpreter, and speed up development of the more advanced parts rapidly. It also assures us that new changes does not break examples that used

Compiler and Interpreter test suite

By Team Turtle

Starting test of 19 test files...




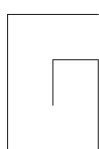
Input Name	Reference	Generic
tests\candle.tg		
tests\colors.tg		

Figure 8: Test output

to work earlier, as this would be spotted right away. A full test run from the test suite can be found in Appendix C.

4 Discussion and further work

4.1 Difficulties

Although we are quite satisfied with the result, there were some difficulties along the way. Early on writing the EBNF grammar was very difficult; being completely different from anything else we have done before. This made for a slow start, but fortunately once we got into it functionality started to come in place. The grammar and the resulting auto generated code also shows weakness; any small error will result in completely unpredictable errors in the resulting compiler. These were handled one at a time, but thankfully we did not have many of them.

In addition to this we decided to skip two of Web Turtle's features, namely "thick", which changes the thickness of the line, and "color", which changes

the color. The reason for this is inconsistent syntax in Web Turtle. This inconsistency makes it very difficult to implement in the grammar. For regular arithmetic operations “+1” means the positive integer one. For “thick” and “color” this has a different meaning; add one to the current value. We use the same arithmetic parser for all instructions and as such we could not change its behavior for these ops only. Instead we decided to drop these two, of which only one was in our original plan, as they are purely cosmetic and do not affect the actual path the turtle takes. Thick would also require a different line drawing algorithm from the one we currently use.

4.2 Future work

In its current state the compiler is very basic. It parses the source code and outputs the byte code in a single sweep, skipping optional compiler steps such as memory and speed optimizations. A future revision could include byte code optimization. Other targets for the interpreter could also be implemented.

The two instructions we skipped could be implemented by separate rules. There are also other Web Turtle instructions, such as “text”, that we decided not to implement that with additional work could be supported.

Web Turtle was made as a way to introduce Turtle Graphics. By extending Web Turtle and adding more features we could enhance its capabilities and create new and exciting graphics. User input during interpretation would be an interesting way to affect the graphic as it is being drawn.

References

- [1] <http://www.sonic.net/~nbs/webturtle>
- [2] <http://www.erzwiss.uni-hamburg.de/Sonstiges/Logo/logofaqx.htm>
- [3] <http://www.ssw.uni-linz.ac.at/coco/>
- [4] <http://catalog.compilertools.net/>
- [5] http://www.ece.cmu.edu/~koopman/stack_computers/
- [6] <http://www.devkitpro.org/about/>
- [7] <http://www.libsdl.org/index.php>