

# BitCC Release Notes<sup>†</sup>

Version 0.9.1

Jonathan Shapiro, Ph.D. Swaroop Sridhar  
*Systems Research Laboratory*  
Dept. of Computer Science  
Johns Hopkins University

February 17, 2006

## Abstract

Describes how to obtain BitCC version 0.9.1 and known issues in the implementation.

We are very pleased to announce the release of BitCC version 0.9.1. This is a compiler conforming (mostly) to the version 0.9 specification for the BitC programming language. This document describes how to obtain and build the compiler, known issues or incompletenesses in the implementation.

## 1 Obtaining The Release

This release of BitCC is only available as part of the Coyotos operating system release. Instructions for obtaining Coyotos can be found in *The Coyotos Build System*. Future versions of BitCC will certainly be available in a more standalone form.

This release is an *early alpha release*! It is undoubtedly broken in very basic ways. The purpose of this release is to discover *how* the compiler is broken so that we can stabilize it. What we are hoping for in this release is examples of short programs that break the compiler so that we can get the bugs identified and resolved. Discounting helper libraries, the compiler is just under 19,000 lines of C++ code — debugging it is going to take a little bit of time!

The BitCC build does not rely on most of the Coyotos cross tools, but it does rely on having the host portion of the Coyotos cross environment installed. Most notably, you will need to install the package providing the Boehm-Weiser garbage collector and its dependencies. At this stage, we are recommending that you simply install the Coyotos cross environment as recommended, and then use it to build BitCC. You can then *skip* the top-level build of Coyotos, and execute:

```
cd ../coyotos/src/ccs/  
make install
```

Following this command, you will find that the binary program `bitcc` and the supporting libraries and header files have been installed in:

```
$(BITC_ROOT)/host/bin/  
$(BITC_ROOT)/host/lib/  
$(BITC_ROOT)/host/include/
```

Once installed, you can run `bitcc` from the installed location. The compiler currently uses the environment variable **BITC\_ROOT** to locate the installed headers and libraries.

<sup>†</sup> Copyright © 2006, Jonathan S. Shapiro and Swaroop Sridhar.

## 2 Compiler Usage

This version of BitCC is a *whole program* compiler. The expected command line is:

```
bitcc [-I if-root] [-L libdir] -o target src1.bitc ... srcN.bitc
```

Source files specified on the command line should include *only* source units of compilation. Imports will be resolved and loaded automatically by the compiler.

The `-I if-root` option can be repeated. Each such option names the *top* of a directory tree containing interface modules. An attempt to import the module `foo.bar.baz` will be resolved by attempting to import `foo/bar/baz.bitc` in turn from each of the specified `if-root/` directory trees.

### 2.1 Seeing the C Code

If you wish to examine the C code generated by the compiler — which is interesting primarily as an illustration of just how ugly C code can get — you can use:

```
bitcc [-I if-root] [-L libdir] -l c -o target.c src1.bitc ... srcN.bitc
```

### 2.2 Wrapping An Existing Library

If you wish to implement a wrapper for an existing C library, you may wish to examine `ccs/bitcc-bootstrap/libbitc/` and its use of the `-h` option to BitCC. Where the `-l c` option says “generate C code”, the `-l h` command says “generate a header file that can be used to implement a supporting wrapper library.” This header file will contain declarations for:

- Every exception declared in the imported modules.
- Every `proclaim` that is marked `external`, provided that the proclaimed procedure does not have polymorphic type.
- Every type that is (recursively) reachable from these.

This mechanism should be considered *extremely* fragile. It has been used to partially wrap the C `stdio` library, which has convinced us that the mechanism for external binding declarations needs to be extended. A mechanism for specifying the externally bound identifier will be provided in the next compiler release.

## 3 Known Issues

This is very much a “draft zero” compiler, and several things are completely or partially unimplemented in this compiler.

**Type Classes** Type classes are not implemented. Procedures such as arithmetic, comparison, and so forth are declared in this compiler using polymorphic type:

```
(proclaim == : (fn ('a 'a) bool))
```

As we implement type classes, these declarations will be changed to use them. We hope that this will be implemented shortly.

**Union Representation** The union type tag declaration mechanism described in section 3.6.2.2 of the specification is not implemented. The declaration will be accepted, but is silently ignored in this implementation. The storage layout optimization described in the last paragraph of that section (the so-called “Cardelli Optimization” is not implemented).

**int type** This compiler does not implement an arbitrary precision integer type. It is looking like the implementation of the `int` type can be done equally efficiently in a library, and we are considering dropping it from the core language.

**case patterns** This compiler implements only non-nested patterns. This is sufficient for many programs, and it relieved us of the need to implement the full completeness check (which is mildly tricky to do). A non-nested pattern is either a sequence of literals with an optional final variable, or a list of constructor patterns all of whose arguments are variables.

**Restriction on Inner Lambdas** An inner lambda that closes over a non-global variable and escapes either inward or outward will generate an error. This is straightforward to fix; we have implemented the closure marking pass, but not the closure construction pass. We hope that this will be implemented shortly.

**Mutual Tail Recursion** Section 10.1 of the specification requires proper tail recursion for statically resolvable calls within a single `let rec` form. This is not implemented by the current compiler. We hope that this will be implemented shortly.

**Stateless Interfaces** Section 10.1 of the specification describes stateful and stateless interfaces. This is not implemented by the current compiler.

**Value Polyinstantiation** Immutable values of underspecialized polymorphic type will be multiply instantiated by the compiler. No BitC program can detect this, but it is potentially visible to runtime library extensions.

**Import Cross Reference** There is a *major bug* in the compiler whose effect is that two interfaces may exploit forward declarations to create mutually recursive top-level declarations. The intended resolution for this is to require a conservatively consistent ordering on imports similar to the one used by Standard ML. The BitC compiler will accept this, but subsequent compilation by the C compiler will fail.

**Interface Forward Reference** Within a single interface, forward declarations can similarly be exploited to express circular reference in initialization. The BitC compiler will accept this, but subsequent compilation by the C compiler will fail.

**Quad Precision Floats** Quad precision floating point values are unimplemented.