1 C--

C-- is an assembler language used as the back-end for the Glasgow Haskell Compiler. It is designed to have a minimalistic C-like syntax that is capable of giving very specific computation and memory specifications to the runtime as well as constraints that allow for better optimization of the code.

The previous work (bachelor thesis "Type inference and polymorphism for C") showcased the benefits of type inference and polymorphism in the context of the C language and identified multiple problems this work solves, examples: multiple-parameter type-classes for overloaded structure field accessors (multiple structures having fields with the same name), sub-typing support in the type inference algorithm (in the previous work referenced as *const specifiers* problem). The choice of C-- was guided by its closeness to C, its minimalism and its low-level orientation. The minimalistic syntax allows us to easily demonstrate the benefits of type inference and the implications of introducing it into an imperative language, while the low-level orientation of the language, which we preserve in the final design, makes the language viable for systems programming. Furthermore, C-- allows for expressing stronger and more flexible contracts (possible use-case: guiding alias analysis and branch analysis).

The syntax of a C-- program mainly consists of data declarations, functions, (virtual) registers and statements. The syntax of statements is a derivative of the syntax of the C language, but it is significantly constrained. Most notably:

- C-- expressions, in contrast to expressions in C, cannot produce side effects - one of the most used expressions in the C language is the increment expression, which is a part of virtually every *for* loop. Omission of increments and similar expressions gives the C-- code a look very different from the C code.
- All variables are either numeric (of the kind "float" if it is a floating-point number or of the kind "" otherwise) or addresses (of the kind "address").
- All value assignments and function calls have to be the top-most subexpressions of the statements they appear in - thus, instead of assignment expressions and call expressions, we have assignment statements and call statements.
- Results of comparisons cannot be stored in variables.

- C-- does not have *for* loop statements nor *while* loop statements; and for each *if* statement (with optional parentheses around the condition), the nested statement has to be a brace-enclosed block statement.

Figure 1: Euclid's algorithm (greatest common divisor)

```
f(bits32 u, bits32 w) {
        bits32 r;

while:
        if w != 0 {
                r = u % w;
                u = w;
                w = r;

                goto while;
        }

        return (u);
}

foreign "C" gcd(bits32 u, bits32 v) {
        bits32 r;

        r = f(u, v);

        return (r);
}
```

Figure 1 shows an implementation of Euclid's algorithm as a C-- function. As C-- lacks an explicit syntax for loops, we implement a *while* loop via labels: we put a label before the condition and then branch to this label at the end of the nested block statement - this is how we would express any loop; and doing so could be abstracted via a language sugar that would introduce *for* loops and *while* loops into the language in a fashion similar to how they are introduced into the C language.

The foreign "C" specifier before the "gcd" function specifies the function shall be subject to C-linkage (in C++, we would write extern "C") and it will be callable from C code.

Then, apart from using the (to a C programmer foreign-sounding) bits32 type, which specifies, quite unsurprisingly, that the corresponding register contains 32 bits, and the omitted return types of the functions, the code is superficially almost indistinguishable from C.