# 1 The Module Landscape of the Quick C-- Compiler

**Last major revision: June 2002**
Last CVS checkin: Date: 2006-04-10 21:28:53

The Quick C-- compiler currently is a collection of over one hundred modules. This document tries to sketch the landscape and to identify some landmarks for orientation.

## 1.1 Driver

The Quick C-- compiler is held together by a few driver modules that interact with the user.

- `../src/main2.nw`. The first function executed in the compiler is `Main.main`. It reads command line arguments, initializes the built-in Lua interpreter, catches exceptions, and defines the return code.

  All important phases in the compiler have an interface to the Lua interpreter. The Lua code that controls the flow of data between phases is defined in module `Main`. After the interpreter has been booted with this code, function `Main.main` passes the command line arguments into the interpreter for evaluation. From there on the Lua code controls the compiler.

- `../src/driver2.nw`. This module exports front end data structures and functions to the Lua interpreter. For example, `Driver2` implements a new Lua data type `ast` that represents the abstract syntax of a C-- source file.

## 1.2 Front End

The *front end* reads a C-- source file and translates it into a target-independent intermediate representation which the *back end* will later translate to machine code. Almost all aspects of the front end are independent independent of the actual target. All syntactic and most semantic errors in a C-- program are detected in the front end. It is the main source of error messages for the user.

- `../src/parser.nw`. This module defines the scanner and parser for the C-- language. The scanner is derived from an OCamlLex specification, the parser from an OCamlYacc specification. The main result from scanning and parsing is the abstract syntax tree of the source file.

- `../gen/ast.nw`. The abstract syntax data type is derived from an ASDL specification.

- `../cllib/srcmap.nw`. An OCamlYacc-generated parser represents a source code position in a file as a character offset from the beginning of the file.

This representation makes it difficult to annotate error messages with comprehensible source code locations. The `Srcmap` module provides a translation for offset-based source code positions to file- and line-oriented positions. Such a map is maintained by the scanner and returned by the parser together with the abstract syntax tree of a source file.

- `../src/fenv.nw`. The fat environment provides the symbol table of the compiler. This data structure records the denotation of names in the C-- source file that is currently translated.

- `../src/elab.nw`. This module checks the static semantics of a source file, reports errors, and builds up the fat environment. The module does not stop after the first error but uses error propagation combinators (`../src/error.nw`) to find as many errors as possible, even in the presence of errors.

  C-- names can denote compile-time constants and type aliases. The corresponding compile-time expressions and type aliases are resolved by module `Elab` and the results are recorded in the fat environment.

- `../src/types.nw`. The static semantics of C-- require type checks: types of arguments to primitive functions must match the types of their formal parameters,and `if`-statements are governed by an expression of type `bool`. This module lays out the type language for this purpose.

- `../src/expcheck.nw`. The QC-- compiler type checks AST expressions during several phases: evaluation of constant expressions, static semantics, translation to intermediate representation. To avoid code duplication, type checks for expressions are factored out into this module.

- `../src/error.nw`. The idea of error-handling in QC-- is not to stop in all cases after an error but to mark results as either good or bad. Bad values can be used for further computation although the overall result will be bad. The hope is, to detect even more errors that are *not* caused by any previous error. The error-propagating combinators in this module allow to make ordinary functions error-aware, allowing them to deal with inputs that might be bad.

New front end modules as of January 2004:

- `../src/nast.nw`. "Normalized AST." The Quick C-- AST follows the concrete syntax directly. As such, it is very flexible for the front end and correspondingly complicated to process. The goal of this module is to put the AST into a sort of a normal form. The normal form collects and segregates declarations for simpler processing. Every AST is normalizable, so this module never causes an error.

- `../src/nelab.nw`, `../src/elabexp.nw`, `../src/elabstmt.nw`. Elaboration of an AST. These modules split what was done by `../src/elab.nw`

in the old front end. Unlike the old front end, they not only detect errors but also put the program into an elaborated normal form. Thus, some work that was deferred to `../src/ast2ir.nw` in the old front end is done here in the new front end.

To ease understanding, elaboration is split across three modules: `../src/elabexp.nw` for expressions, `../src/elabstmt.nw` for statements, and `../src/nelab.nw` for programs.

These modules are the *only* modules that detect and return error values. Upstream, no errors are detected, and downstream, all errors are assumed to have been detected here.

## 1.3 Translation to Intermediate Representation

Front end and back end are connected by the *intermediate representation*. During the translation, that last step of the front end is to create the intermediate representation of the program under translation.

- `../src/ast3ir.nw`. The `translate` function takes an environment, the actual target and a program and translates it into an abstract assembler program. The assembler is functorized over the data type of instructions. This module puts a procedure body into a single instruction by basically treating a control-flow graph as an instruction.
- `../src/copyinout.nw`. The details of how parameters are passed to a procedure and results are returned are covered by a calling convention. The implementation of a calling conventions is a sequence of load and store instructions that move parameters into the registers and memory slots. The `Copyinout` module creates such sequences of assignments nodes in the CFG for parameter passing. It is used during the creation of the intermediate representation (by `../src/ast3ir.nw`).

## 1.4 Intermediate Representation

A program has two principal parts: initialized and uninitialized data like strings, buffers, and constants, and code. In the intermediate representation only code has is held in a data structure. Data is directly emitted to an assembler during the translation to the intermediate representation by module `../src/ast3ir.nw`.

- `../src/cfg4.nw`. The executable part of a procedure is represented as a control-flow graph (CFG). The graph is polymorphic over instructions that are embedded into its nodes. The most important specialization is when RTLs are used as instruction representation.
- `../src/asm3.nw`. The `../src/astr3ir.nw` module uses this interface to access text-based and binary assemblers. Assembler aspects like symbols and relocatable addresses (from module `../rtl/symbol.nw`) escape the

assembler and become part of symbol tables and expressions. To avoid that all aspects of the compiler must be functorized over the different assemblers we have chosen to use a very polymorphic interface that uses Objective Caml's object system.

- `../src/cfgprop.nw`. We may attach arbitrary properties to each node of a control-flow graph. Because the set of properties is expected to evolve while the flow-graph interface remains stable, we put the properties in a separate module.

- `../src/cfgutil.nw`. The `../src/cfg4.nw` module provides the essential functionality for control-flow graphs (CFGs). In order to separate essential functionality from nice-to-have functionality this module provides additional functions on CFGs whose implementation do not require knowledge of the internal details of `../src/cfg4.nw`.

## 1.5 Assemblers

An assembler is an implementation of the `../src/asm3.nw` interface.

- `../src/asdlasm.nw.nw`. This module implements an assembler that emits the RTLs of a procedure in ASDL format. Other assembler instruction do not create output.

- `../src/astasm.nw`. This is an assembler that emits C--. It implements the `Asm3.assembler` interface for assemblers in the QC-- compiler.

- `../src/dotasm.nw`. This module implements an assembler that emits every procedure as a graph in DOT format. The assember constructor receives the output channel for the assembler.

- `../src/msparcasm.nw`. This module provides an assembler that emits directives and instructions in SPARC assembly syntax. The interface conforms to the `Asm3` assembler interface.

- `../src/sparcasm.nw`. This is an assembler that emits SPARC assembly language. It implements the `Asm3.assembler` interface for assemblers in the QC-- compiler. It should be functorized to deal with differences between SunOS and Solaris, but to hell with SunOS. Unused.

## 1.6 Stack Frame and Calling Conventions

A procedure keeps local data at run-time in registers and in its stack frame. It is the compiler's responsibility to organize a the stack frame. The modules below all deal with memory blocks that are ultimately composed into a stack frame.

- `../src/block.nw`. A `Block.t` value represents a memory block. Memory blocks can be composed to larger blocks. This abstraction is used to model slots in a stack frame, parameter passing areas in a frame, entire stack frames, and memory for data.

- `../src/memalloc.nw`. This module provides an applicate abstraction to allocate a memory block incrementally by a sequence of `allocate` and `align` directives. The abstraction is similar to a pointer that is advanced and aligned to reserver memory. At any point the value of the pointer can be used as an address. After allocation is complete, the `t` value is `frozen` and the allocated block is returned.

- `../src/automaton2.nw`. Allocation of registers for parameter passing and slot allocation can be hidden behind an automaton. An automaton knows a set of locations and hands them out in response to queries. A location can be a hardware register, a memory location, or a combination of both. The details are hidden behind a common abstraction. Automata are target specific and implement allocation policies.

- `../src/contn.nw` A continuation is a pinter-sized C-- *value*. As such, it can be passed around and stored in registers. However, not all data necessary to represent a continuation fit into a pointer and therefore the pointer points to a pair of two values in memory: first, a pointer to the code belonging to a continuation, and second, a stack pointer value. We call this pair the continuation *representation*. The representation is stored in the activation record of a procedure and is initialized when the procedure becomes active. To deal with the different aspects of a continuation, this module provides an abstraction.

- `../src/eqn.nw`, `../src/const2.nw`. The memory block abstraction represents constraints by equations over RTL expressions.

  The `Const2` module allows to build an equation system by equating RTL expressions. The equation system determines the value of late compile-time constants that are part of the participating expressions. For solving the equation system, equations are transformed into linear equations over integers.

  `Eqn` provides a solver for linear equations: the solver takes a set of linear equations as input and finds the values of the embedded variables. A linear equation has the form $\sum_i c_i t_i = 0$ where each coefficient $c_i$ is an integer and each term $t_i$ is either a variable ($v$) or a non-zero constant ($k$): $t ::= v|k$.

## 1.7 Register Transfer Lists

The register transfer list (RTL) is a central data structure in the compiler. A register transfer list is a general data structure that can represent any machine instruction. Register-transfer lists capture the meaning of effects, like storing

values into memory or register location. They are used as part of the abstract representation of C--. The meaning of a register-transfer list is well defined and especially independent from any properties of a target architecture. RTLs are carried in nodes of the control-flow graph and created by the `ast3ir.nw` module.

- `../rtl/rtl.nw`. The definition of a RTL. The interface is divided into three parts: *public*, *private*, and *common*. The public part provides constructor functions to build RTLs but hides their actual implementation. The private part reveals these and is intended only for parts of the back end that does re-writing of RTLs. Both private and public parts share *common* types. The public part includes function that converts public values into private values and thus makes their details accessible.

- `../src/space.nw`. The RTL framework groups memory cells (main memory, registers, temporaries) into named spaces. The cells forming a space share properties but the RTL framework does not announce them. This module provides a type to describe an RTL space.

- `../src/rtlutil.nw`. Functions to observe and transform RTLs. For example, function to observe the width of a location or an expression, substitution routines, and functions to create a string representation of an RTL.

- `../src/rtlop.nw`. This module provides the translation of a C-- operator to an RTL operator. C-- and RTL operators are closely related: a C-- operator is in general an instance of a polymorphic RTL operator.

- `../src/rtleval2.nw`. An RTL expression can denote a compile-time, link-time, or run-time value. The goal is to evaluate any RTL expression as early as possible to minimize computation at run-time. This module provides routines to evaluate RTL expressions and to simplify RTLs by evaluating expressions inside of them. The promise is, that the returned RTLs and expressions are simpler as the ones provided.

- `../src/rtldebug.nw`. Functions to type check an RTL, i.e. to check its internal consistency. Used for debugging.

## 1.8 Values (Bit Vectors)

C-- treats all values as bit vectors. Any interpretation of a value as an integer or floating point value is left to operators working on values.

- `../rtl/bits.nw`. This module provides an abstract bit vector type `bits`. It is used for holding C-- values. Bit values can be converted to and from numbers and strings. Conversion implies an *interpretation* of a value. Most commonly, values can be interpreted as signed or unsigned integers. Therefore, constructors and observers of `bits` values are organized along

these lines. The conversions try hard not to assume a certain representation for OCAML integers.

- `../rtl/bitops.nw`. This module encapsulates operations on bit vectors. Its primary use is to support constant evaluation, but it is also used to evaluate 'early' expressions during RTL creation. Another mayor client is the encoding and decoding of instructions in RTL representation.

- `../gen/rtlasdl.nw`, `../src/rtlx.nw`. To allow external software to read RTLs we have generated, we export them using ASDL. Since the RTL data type is defined as a OCAML data type and not as a ASDL data type, we create an additional definition of an RTL using the ASDL specification language. We then convert an existing RTL into an ASDL-based RTL which we can export.

## 1.9 Registers, Liveness, and Temporaries

A register-like location is an RTL location whose address is always an compile-time integer constant. Because registers are central to the compiler, they have their own data structure and functions.

- `../src/register.nw`. A `Register.t` describes a target-specific register-like location. Such a location includes hardware-registers or temporaries, but not spill locations. The distinguishing feature of a register is a fixed address within its space: the address is an integer rather than a general expression.

- `../src/live.nw`. Liveness analysis determines for every node in a flow graph (see module `../src/cfg4.nw`) the set of registers that will be used in that node's successors.

- `../src/lifetime.nw`. A lifetime is a set of disjoint intervals. In the context of the QC-- compiler a lifetime represents the parts in a linear program representation a variable is live. The fact that a lifetime is a set of disjoint intervals means, that a variable can be live in different parts of a (linearized) program that are not adjacent to each other.

- `../src/talloc.nw`. Multiple parts of the compiler may need to allocate temporaries. Here we provide simple allocators for temporaries.

## 1.10 Variable Placement

Initially, RTLs as part of CFG nodes include variables. They must be replaced by temporaries.

- `../src/placevar.nw`. A toy variable placer.

## 1.11 Register Allocation

Register allocation replaces temporaries in CFG nodes by hardware registers. When necessary, registers are spilled to stack slots.

- `../src/ocolorgraph.nw`. A graph-coloring register allocator with whose individual components can be controlled from Lua.
- `../src/linscan.nw`. A linear-scan register allocator with a minimal Lua interface.

## 1.12 Lua Interpreter

The compiler is controlled by a built-in Lua 2.5 interpreter. All important phases of the compiler have interfaces to Lua which allows them to be controlled by Lua code. The implementation of the Lua interpreter is in its own sub directory `lua/`.

- `../lua/lua.nw`. Central module that provides access to the Lua interpreter.
- `../src/luautil.nw`. This module provides a new Lua primitive that allows to load a Lua file that is searched along a search path.

## 1.13 Back Plane

The backplane is a set of new Lua primitives that allows to control compiler phases with Lua interface from Lua.

- `../src/backplane.nw`. The backplane allows the user to customize the backend of the compiler. The building blocks of the backplane are the stages between which the user can define a control flow. Stages represent compiler components, such as the optimizer or the register allocator. The backplane can also be used with a finer level of granularity of compiler components, such as individual optimization stages.

## 1.14 Procedure Representation in the Back End

The compiler's back end is mostly concerned with the translation of procedures. A procedure in the back end is represented by a record that groups all associated informations.

- `../src/proc.nw`. Data type definition for procedure representation.

## 1.15 Targets

The compiler has to know the properties of the target it is compiling code for. Here are the modules that provide the data structures and values for this information.

- `../src/target2.nw`. Data structure `Target2.t` that describes a target,
- `../src/targets.nw`. All targets the compiler knows about. Defined a number of `Target2.t` values. Targets are exported to Lua.

## 1.16 Code Expansion

A code expander expands nodes in the CFG such that each node's RTL represents a single machine instruction. Code expansion is target specific and handled by different modules. Each module has a Lua interface and thus can be selected at run-time.

- `../src/expander.nw`. A toy code expander from the early days of the compiler.
- `../src/dummyexpander.nw`. The RTLs generated by the `../src/ast3ir.nw` module violate the machine invariant that every RTL must be representable as a machine instruction. A code-expander establishes this invariant by simplifying RTLs. This module provides a toy code expander for the `Dummy` target. Although it would be possible to write a generic code expander that works for all targets, we keep things simple and write a specialized one. The expander is specified as a set of BURG rules.
- `../src/sparcexpander.nw`. This module provides an `expand` function that re-writes an RTL into a sequence of Sparc machine instructions. The module is parameterized over a module `I` that provides constructor functions for abstract Sparc instructions `I.t` The code expander is implemented as a BURG rule set.

## 1.17 Machine Instructions and Recognizer

The code expander creates RTLs that represent machine instructions, which are later matched and emitted by a recognizer. Constructor functions that build machine instructions and the recognizer must agree on the representation of individual instructions.

- `../src/minisparc.nw`. Constructor functions and recognizer for the SPARC.
- `../gen/sparcmkasm.nw`. Emitter for SPARC instructions.
- Directory `../gen` contains machine-generated constructor functions and recognizer for the SPARC. However, they are currently not used.

## 1.18 Utilities

The compiler includes a number of self-contained utility modules.

- Directory `../cllib` contains modules for pretty printing and parser combinators.
- `../src/topsort.nw`. Topological sort function.
- `../src/idgen.nw`. Generator for unique names.
- `../src/newer.nw`. Tool to compare file dates; used in `mkfile`.
- `../src/strutil.nw`. Sets and maps over strings.
- `../src/impossible.nw`. Exception for internal errors.
- `../rtl/uint64.nw`. Unsigned operations on 64 bit intergers.
- `../src/auxfuns.nw`. Various small utilities.
- `../src/base.nw`. Type aliases for OCAML base types. Once used by Daniel Damien but unused as of today.
- `../src/mangle.nw`. A name mangler; used by assemblers.