c2nim User's manual 0.8.15

Andreas Rumpf

May 9, 2012

Contents

1 Introduction

c2nim is a tool to translate Ansi C code to Nimrod. The output is human-readable Nimrod code that is meant to be tweaked by hand after the translation process. c2nim is no real compiler!

c2nim is preliminary meant to translate C header files. Because of this, the preprocessor is part of the parser. For example:

```
#define abc 123
#define xyz 789
```

Is translated into:

```
const
  abc* = 123
  xyz* = 789
```

c2nim is meant to translate fragments of C code and thus does not follow include files. c2nim cannot parse all of Ansi C and many constructs cannot be represented in Nimrod: for example duff's device cannot be translated to Nimrod.

2 Preprocessor support

Even though the translation process is not perfect, it is often the case that the translated Nimrod code does not need any tweaking by hand. In other cases it may be preferable to modify the input file instead of the generated Nimrod code so that c2nim can parse it properly. c2nim's preprocessor defines the symbol `C2NIM` that can be used to mark code sections:

```
#ifndef C2NIM
  // C2NIM should ignore this prototype:
  int fprintf(FILE* f, const char* frmt, ...);
#endif
```

The `C2NIM` symbol is only recognized in `#ifdef` and `#ifndef` constructs! `#if defined(C2NIM)` does **not** work.

c2nim *processes* `#ifdef C2NIM` and `#ifndef C2NIM` directives, but other `#if[def]` directives are *translated* into Nimrod's `when` construct:

```
#ifdef DEBUG
#  define OUT(x) printf("%s\n", x)
#else
#  define OUT(x)
#endif
```

Is translated into:

```
when defined(debug):
  template OUT*(x: expr): expr =
    printf("%s\x0A", x)
else:
  template OUT*(x: expr): stmt =
    nil
```

As can been seen from the example, C's macros with parameters are mapped to Nimrod's templates. This mapping is the best one can do, but it is of course not accurate: Nimrod's templates operate on syntax trees whereas C's macros work on the token level. c2nim cannot translate any macro that contains the `##` token concatenation operator.

c2nim's preprocessor supports special directives that affect how the output is generated. They should be put into a `#ifdef C2NIM` section so that ordinary C compilers ignore them.

2.1 `#skipinclude` directive

Note: There is also a `-skipinclude` command line option that can be used for the same purpose.

By default, c2nim translates an `#include` that is not followed by < (like in `#include <stdlib>`) to a Nimrod `import` statement. This directive tells c2nim to just skip any `#include`.

2.2 `#stdcall` and `#cdecl` directives

Note: There are also `-stdcall` and `-cdecl` command line options that can be used for the same purpose.

These directives tell c2nim that it should annotate every proc (or proc type) with the `stdcall` / `cdecl` calling convention.

2.3 `#dynlib` directive

Note: There is also a `-dynlib` command line option that can be used for the same purpose.

This directive tells c2nim that it should annotate every proc that resulted from a C function prototype with the `dynlib` pragma:

```
#ifdef C2NIM
#  dynlib iupdll
#  cdecl
#  if defined(windows)
#    define iupdll "iup.dll"
#  elif defined(macosx)
#    define iupdll "libiup.dylib"
#  else
#    define iupdll "libiup.so"
#  endif
#endif

int IupConvertXYToPos(PIhandle ih, int x, int y);
```

Is translated to:

```
when defined(windows):
  const iupdll* = "iup.dll"
elif defined(macosx):
  const iupdll* = "libiup.dylib"
else:
  const iupdll* = "libiup.so"

proc IupConvertXYToPos*(ih: PIhandle, x: cint, y: cint): cint {.
  importc: "IupConvertXYToPos", cdecl, dynlib: iupdll.}
```

Note how the example contains extra C code to declare the `iupdll` symbol in the generated Nimrod code.

2.4 `#header` directive

Note: There is also a `-header` command line option that can be used for the same purpose.

The `#header` directive tells c2nim that it should annotate every proc that resulted from a C function prototype and every exported variable and type with the `header` pragma:

```
#ifdef C2NIM
#  header "iup.h"
#endif

int IupConvertXYToPos(PIhandle ih, int x, int y);
```

Is translated to:

```
proc IupConvertXYToPos*(ih: PIhandle, x: cint, y: cint): cint {.
  importc: "IupConvertXYToPos", header: "iup.h".}
```

The `#header` and the `#dynlib` directives are mutually exclusive. A binding that uses `dynlib` is much more preferable over one that uses `header`! The Nimrod compiler might drop support for the `header` pragma in the future as it cannot work for backends that do not generate C code.

2.5 #prefix and #suffix directives

Note: There are also `-prefix` and `-suffix` command line options that can be used for the same purpose.

c2nim does not do any name mangling by default. However the `#prefix` and `#suffix` directives can be used to strip prefixes and suffixes from the identifiers in the C code:

```
#ifdef C2NIM
#  prefix Iup
#  dynlib dllname
#  cdecl
#endif

int IupConvertXYToPos(PIhandle ih, int x, int y);
```

Is translated to:

```
proc ConvertXYToPos*(ih: PIhandle, x: cint, y: cint): cint {.
  importc: "IupConvertXYToPos", cdecl, dynlib: dllname.}
```

2.6 #mangle directive

Even more sophisticated name mangling can be achieved by the `#mangle` directive: It takes a PEG pattern and format string that specify how the identifier should be converted:

```
#mangle "'GTK_'{.*}" "TGtk$1"
```

For convenience the PEG pattern and the replacement can be single identifiers too, there is no need to quote them:

```
#mangle ssize_t  int
// is short for:
#mangle "'ssize_t'" "int"
```

2.7 #private directive

By default c2nim marks every top level identifier (proc name, variable, etc.) as exported (the export marker is `*` in Nimrod). With the `#private` directive identifiers can be marked as private so that the resulting Nimrod module does not export them. The `#private` directive takes a PEG pattern:

```
#private "@('_'!.)" // all identifiers ending in '_' are private
```

Note: The pattern refers to the original C identifiers, not to the resulting identifiers after mangling!

2.8 #skipcomments directive

Note: There is also a `-skipcomments` command line option that can be used for the same purpose.

The `#skipcomments` directive can be put into the C code to make c2nim ignore comments and not copy them into the generated Nimrod file.

2.9 #typeprefixes directive

Note: There is also a `-typeprefixes` command line option that can be used for the same purpose.

The `#typeprefixes` directive can be put into the C code to make c2nim generate the `T` or `P` prefix for every defined type.

2.10 `#def` directive

Often C code contains special macros that affect the declaration of a function prototype but confuse c2nim's parser:

```
// does not parse!
EXTERN(int) f(void);
EXTERN(int) g(void);
```

Instead of removing `EXTERN()` from the input source file (which cannot be done reliably even with a regular expression!), one can tell c2nim that `EXPORT` is a macro that should be expanded by c2nim too:

```
#ifdef C2NIM
#  def EXTERN(x) static x
#endif
// parses now!
EXTERN(int) f(void);
EXTERN(int) g(void);
```

`#def` is very similar to C's `#define`, so in general the macro definition can be copied and pasted into a `#def` directive.

3 Limitations

- C's `,` operator (comma operator) is not supported.
- C's `union` are translated to Nimrod's objects and only the first field is included in the object type. This way there is a high chance that it is binary compatible to the union.
- The condition in a `do while(condition)` statement must be `0`.
- Lots of other small issues...