# FFIGEN User's Manual*
# (Preliminary)

Lars Thomas Hansen
lth@cs.uoregon.edu

February 6, 1996

## 1 Introduction

FFIGEN is a program system which facilitates the writing of translators from C header files to foreign function interfaces for particular programming language implementations. This document describes its structure and use. The discussion is aimed at translator writers; everyone else should confine themselves to section 3. A companion document, *FFIGEN Manifesto and Overview*, motivates the work, and other companion documents describe specific translator implementations. In particular, the document *FFIGEN Back-end for Chez Scheme Version 5* describes one translator in detail.

FFIGEN is based on the `lcc` C compiler, which is copyrighted software. See section 10 for a full copyright notice.

## 2 Writing Translators

To generate a translation of a header file you run the **`ffigen`** command to generate an intermediate form of the C header files you want to translate, and then run the back-end on the resulting files to generate the foreign function interface for the library.

Your task, should you choose to accept it, is to implement the target-specific parts of the back-end for your particular target (which is to say, combination of host language implementation, operating system, architecture, foreign language implementation, and translation policy). You should be able to use the FFIGEN front-end and the target-independent parts of the back-end pretty much as they are.

How to implement the target-specific parts of the back-end is discussed in section 6. Use of the front end is described in section 3. The intermediate format is described in section 4, and the target-independent parts of the back-end and their interface to the target-dependent part are described in section 5. Finally, section 7 covers some issues which need to be tackled in the future.

## 3 Running FFIGEN

The command **`ffigen`** is run on a set of header files with preprocessor option and include file options. Arguments are processed in order. For each header file (type `.h`) and all the files it includes, a single preprocessor file (type `.ffi`) is produced.

The options are:

**`-Dname[=value]`** Define preprocessor macro.

**`-Uname`** Undefine preprocessor macro

*This work has been supported by ARPA under U.S. Army grant No. DABT63-94-C-0029, "Programming Environments, Compiler Technology and Runtime Systems for Object Oriented Parallel Processing".

**-Idirectory** Add directory to the *beginning* of the list of include files. Standard directories include the `lcc` include directory, `/usr/include`, and the current directory (in that order). See the release notes for information about how to change the defaults.

`ffigen` performs full syntax and type checks on its input.

The back-end is run by starting your favorite Scheme system and then loading first the target-independent file `process.sch` and second the target-dependent part of the translator; in the case of the Chez Scheme back-end the file is called `chez.sch`. You then call the procedure `process` with the name of the `.ffi` file to process, as discussed in section 5.

# 4 Intermediate Format

The intermediate format consists of s-expressions following this grammar:

```
<file>      -> <record> ...
<record>    -> (function <filename> <name> <type> <attrs>)
             | (var <filename> <name> <type> <attrs>)
             | (type <filename> <name> <type>)
             | (struct <filename> <name> ((<name> <type>) ...))
             | (union <filename> <name> ((<name> <type>) ...))
             | (enum <filename> <name> ((<name> <value>) ...))
             | (enum-ident <filename> <name> <value>)
             | (macro <filename> <name+args> <body>)
<type>      -> (<primitive> <attrs>)
             | (struct-ref <tag>)
             | (union-ref <tag>)
             | (enum-ref <tag>)
             | (function (<type> ...) <type>)
             | (pointer <type>)
             | (array <value> <type>)
<attrs>     -> (<attr> ...)
<attr>      -> static | extern | const | volatile
<primitive> -> char | signed-char | unsigned-char | short
             | unsigned-short | int | unsigned | long
             | unsigned-long | float | double | void
<value>     -> <integer>
<filename>  -> <string>
<name>      -> <string>
<body>      -> <string>
<name+args> -> <string>
<tag>       -> <string>
```

Notes relating to the grammar:

- ... means "zero or more of" the preceding item.
- The grammar is a little more general than the actual output language. All structs, unions, and enums in parameter lists, return types, and variable declarations are encoded as `struct-ref`, `union-ref`, and `enum-ref`, respectively; structure, union, and enum type definitions occur only in `struct`, `union`, and `enum` records.
- The `<tag>` field in structs/unions/enums (and their `-ref` forms) is the tag. If one of these types has a user-defined tag, then that tag is used in the `struct-ref` item for the type; if the structure had no user-defined tag then a tag has been generated by `lcc`. Generated tags have the syntax of positive integers; in particular they start with a digit. There is one namespace each for structs, unions, and enums.

- `typedef` names are not used anywhere: they occur in `type` records only.
- The attributes on primitive types are `const` or `volatile`; the attributes `static` and `extern` are used only on functions and global variables.
- Functions which are known to take no parameters (i.e. `t f(void)`) have one parameter, of type `(void ())`. The void type appears in a parameter list only as the last element.
- Functions which take a variable number of arguments have at least one defined non-void parameter and a last parameter of type `(void ())`.
- Functions for which no parameters were defined (i.e. `t f()`) have no parameters.
- The ordering of records in the input has no relation to the relative ordering of declarations in the original source.
- The `<value>` field in the array is its size. If the size is not known, it is 0.
- Multidimensional arrays are represented as nested array types with the leftmost dimension outermost in the expected way; i.e., it looks like an array of arrays.
- Arrays are not valid return types.
- Array parameters lose some semantic information in the translation in the current system. An array parameter `t a[n]` is always converted to a pointer: `(pointer t)` regardless of whether `n` is known or not. As expected, then, something like `t a[n][m][o]` gets the parameter type `(pointer (array m (array o t)))`. Note that this only pertains to parameter types; variables of array type are not converted in this manner. (The semantic information claimed lost is the size of the leftmost dimension. This lossage may make it impossible to perform array conversion at call boundaries, for example.)
- The grammar describes the current format, which will change: line number and column information will be incorporated. You should always use the accessor functions defined in the target-independent part of the back-end; see section 5. The grammar does not allow for bit fields or qualifications on anything but primitive types, but these will be accomodated eventually.

# 5 The Target-Independent Back-End

The target-independent back-end is a Scheme program called `process`[1] which reads the intermediate form into memory and performs some initial processing. It exports some global variables and a number of procedures which are used to access the structures in the database of intermediate records, and imports two target-dependent functions from the target-dependent back-end. This section describes the interfaces.

The global variables which hold the database are:

```
(define functions '())      ; list of function records
(define vars '())           ; list of var records
(define types '())          ; list of type records
(define structs '())        ; list of struct records
(define unions '())         ; list of union records
(define macros '())         ; list of macro records
(define enums '())          ; list of enum records
(define enum-idents '())    ; list of enum-ident records
```

Each of these contains a list of all the records of the type indicated by their names. Note that records may look different internally than in the defined intermediate form, so accessor functions (see below) should always be used.

In addition, there are two globals which are set but not used by the target-independent back-end:

[1] For the lack of something better.

```
(define source-file #f)      ; name of the input file itself
(define filenames '())       ; names of all files in the input
```

The main entry point to the back end is the procedure `process`, which takes a single file name as an argument. `Process` initializes globals, reads the file, and processes the records.

```
(define (process filename) ...)
```

Record processing consists of some general analysis and target-specific code generation. First, the target-specific procedure `select-functions` is called; it must set or reset the "referenced" bit in each record depending on whether the function is interesting to the back-end or not. After computing reachability of structured types and setting the referenced bits of those types which are reachable, a translation is generated by a call to the back-end function `generate-translation`, which takes no arguments.

```
(define (select-functions) ...)
(define (generate-translation) ...)
```

A number of data structure accessors and mutators are also available. These are generic procedures which work on all of the record types.

```
(define (file r) ...)           ; file name of record
(define (name r) ...)           ; name in records which have one
(define (type r) ...)           ; type in records which have one
(define (attrs r) ...)          ; attrs in records which have one
(define (fields r) ...)         ; fields in struct/union record
(define (value r) ...)          ; value of enum-ident record
(define (tag r) ...)            ; tag in struct/union/union/-ref record

(define (referenced? r) ...)    ; is record referenced?
(define (referenced! r) ...)    ; set referenced bit
(define (unreferenced! r) ...)  ; reset referenced bit
```

Arguably the `tag` accessor should go away and `name` should simply be used in its place. As it is, `name` is not defined on `struct-ref`, `union-ref`, and `enum-ref` records.

The procedure `record-tag` returns the tag of the record currently being held. It can also be applied to types.

```
(define (record-tag r) ...)     ; get record tag
```

All records can have back-end specific values attached to them; usually these are cached names for operations on structured values, so for now the procedures which manipulate the back-end specific data are called `cache-name` to remember a value and `cached-names` to return the list of remembered values:

```
(define (cache-name r v) ...)   ; remember value in record
(define (cached-names r) ...)   ; retrieve remembered values
```

We should probably replace this with a more general property-list-like mechanism.

In addition, two procedures extract parts of function types:

```
(define (arglist r) ...)        ; function argument types
(define (rett r) ...)           ; function return type
```

Some utilities to deal with file names are also provided:

```
(define (strip-extension fn) ...)
(define (strip-path fn) ...)
(define (get-path fn) ...)
```

A string macro expander makes it easier to generate C code, for the back ends that need it. The macro expander is called `instantiate` and is called with a string template and a vector of arguments (which are also strings). The template contains patterns of the form `@n` where `n` is a single digit; when such a pattern is seen it is replaced with the corresponding value from the argument vector.

```
(define (instantiate template arguments) ...)
```

Two procedures, `struct-names` and `union-names`, take a structure (or union) and returns a list of all the typedef names which reference the structure directly.

```
(define (struct-names struct) ...)
(define (union-names union) ...)
```

An association function which searches one of the record lists for a given record by the `name` field is also available:

```
(define (lookup key items) ...)
```

The procedure `user-defined-tag?` determines whether a tag was defined by the user or generated by the system:

```
(define (user-defined-tag? x) ...)
```

The procedure `warn` takes some arbitrary arguments and generates a warning message on standard output:

```
(define (warn msg . rest) ...)
```

Some standard predicates take a type and test its kind: `primitive-type?` is true if the argument is of a primitive type as outlined in the grammar above; `basic-type?` is true if the argument is a primitive type or a pointer type; `array-type?` is true if the argument is an array type, and finally, `structured-type?` is true if the argument is a `structr-ef` or `union-ref` type:

```
(define (primitive-type? t) ...)
(define (basic-type? t) ...)
(define (array-type? t) ...)
(define (structured-type? t) ...)
```

## 6 Writing a Target-Dependent Back-End

To write the target-dependent back-end, you must decide on the policy for the translation and then implement the translation. The policy covers such issues as: which constructs in C are or are not handled; the translation for each handled construct; how non-handled constructs are dealt with (ignored, detected with warnings, detected with errors); how to deal with exceptional cases (consider the `fgets` example from the *Manifesto*).

For a concrete example, see the companion document *FFIGEN Backend for Chez Scheme Version 5*, which addresses many of the choices to be made and their possible solutions.

## 7 Future Work

A number of features *will* be supported in the future.

- There will be a line and a column field in each record, giving the source line on which the identifier was defined.
- Bitfields will be supported.
- Qualifiers (what's now called attributes, that is, const and volatile) will be supported on all types, not just on primitive non-pointer types like now.

- The intermediate representation will include the name of the orignal input file, and its path.
- The intermediate representation will include a representation of the include file hierarchy which was traversed to produce the intermediate representation.

A number of features will most likely be supported, but need to be investigated:

- It would be nice to retain comments.
- Various popular extensions to C are not currently supported by `lcc`, but would be extremely useful: `long long` is used extensively in Unix header files, and header files for compilers on PCs often use the common Microsoft extensions `__huge`, `__far`, and `__near` (and their non-underscore equivalents). Some C compilers support `__inline` declarations, and although we can't generate code for in-line procedures we can at least parse them if the compiler can cope with `__inline`. (`__inline` is the easier, since it can be ignored. The others must show up as type qualifiers or new types.)
- The current shell-program driver will probably be replaced by something based on the lcc driver.
- I'm going to experiment with partial macro application in the front end so that back-ends can have simple support for macro definitions. Currently, for example, even something as simple as the `EOF` macro will be ignored by the Chez Scheme back-end because its form is `"(-1)"` rather than simply `"-1"`.
- Information about the layout of fields within structured types should possibly be emitted; this information would be useful to low-level FFIs which need byte offset and size to access the field of a structure.

In addition, there are some issues to investigate in a larger perspective:

- General (target-independent) support for useful policy mechanisms.
- How well can the intermediate language support other front-ends? I don't want to fall into the UNCOL pit, but it would be interesting to see how languages which resemble C in their parameter passing mechanisms (Pascal, Modula, Oberon) could be mapped onto the intermediate language. This is not high priority with me, however. If I embark on supporting another front-end language it will probably be (sigh) C++.

# 8 Please Contribute!

My goal is to support as many target languages as is reasonable, but I can't write all the translators myself (I lack the time and, in many cases, the knowledge). Targets that I will take care of include STk, and, if no-one beats me to it, Scsh, both Scheme systems. Someone has already volunteered to write the ILU back-end. Others are interested in back-ends for Modula-3 and Mercury.

Volunteers for any translator back-end are welcome to e-mail me and volunteer their help. I will coach, coordinate, and help out as much as possible.

# 9 Credits

FFIGEN is based on the freely available `lcc` ANSI C compiler, implemented by Christopher Fraser (of AT&T Bell Labs) and David Hanson (of Princeton University).

I would like to thank Fraser and Hanson for producing such an excellent system; `lcc` has been a joy to work with, and their book, *A Retargetable C Compiler: Design and Implementation*, made the implementation of the FFIGEN front end in the matter of roughly a single work day possible. Would it be that all software was this clean!

The development of FFIGEN was supported by ARPA.

# 10 Copyrights

`lcc` is covered by the following Copyright notice: