NYACC C99 Munge Module

Matt Wette
March 2017

Introduction

The sxml parse tree can be used to provide autocoding via the `(nyacc lang c99 munge)` module. For example, start with the following C code

```
typedef const char *string_t;
extern string_t cmds[10];
```

The nyacc output (call it `the-tree`) for this will be

```
(trans-unit
  (decl (decl-spec-list
          (stor-spec (typedef))
          (type-qual "const")
          (type-spec (fixed-type "char")))
        (init-declr-list
          (init-declr
            (ptr-declr (pointer) (ident "string_t")))))
  (decl (decl-spec-list
          (stor-spec (extern))
          (type-spec (typename "string_t")))
        (init-declr-list
          (init-declr
           (array-of (ident "cmds") (p-expr (fixed "10")))))))
```

If we feed `the-tree` into `tree->udict` and use `assoc-ref` to lookup `"cmds"` we get

```
(udecl (decl-spec-list
         (stor-spec (extern))
         (type-spec (typename "string_t")))
       (init-declr
         (array-of (ident "cmds") (p-expr (fixed "10")))))
```

Now take this and feed into `expand-decl-typerefs` to get

```
(udecl (decl-spec-list
         (stor-spec (extern))
         (type-qual "const")
         (type-spec (fixed-type "char")))
       (init-declr
         (ptr-declr
           (pointer)
           (array-of (ident "cmds") (p-expr (fixed "10"))))))
```

which, when fed through the C99 pretty-printer, generates

```
extern const char *cmds[10];
```

Since the NYACC C99 parser captures some comments, these can be preserved in the above procedure.

The Util2 (or Munge) Module

Declarations must have one of

- declarators

```
int foo;
```

- struct or union reference

```
struct foo;
```

- enum value

```
enum  FOO = 1 ;
```

From Util2

(decl (decl-spec-list ...) (init-declr-list (init-declr ...) ...)) has been replaced by (decl (decl-spec-list ...) (init-declr ...)) ...

`declr->ident` *declr* => (*ident* "*name*") [Procedure]

Given a declarator, aka `init-declr`, return the identifier. This is used by `trans-unit->udict`. See also: declr->id-name in body.scm.

`c99-trans-unit->udict` *tree* [*seed*] [*#:filter f*] => *udict* [Procedure]
`c99-trans-unit->udict/deep` *tree* [*seed*]=> *udict* [Procedure]

Turn a C parse tree into a assoc-list of global names and definitions. This will unwrap `init-declr-list` into list of decls w/ `init-declr`.

```
BUG: need to add struct and union defn's: struct foo  int x; ;
how to deal with this
lookup '(struct . "foo"), "struct foo", ???
wanted "struct" -> dict but that is not great
solution: munge-decl => '(struct . "foo") then filter to generate
("struct" ("foo" . decl) ("bar" . decl) ...)
("union" ("bar" . decl) ("bar" . decl) ...)
("enum" ("" . decl) ("foo" . decl) ("bar" . decl) ...)
```

So globals could be in udict, udefs or anon-enum.

```
What about anonymous enums?  And enums in general?
Anonmous enum should be expaneded into
```

If *tree* is not a pair then *seed* – or `'()` – is returned. The filter *f* is either `#t`, `#f` or predicate procedure of one argument, the include path, to indicate whether it should be included in the dictionary.

`munge-decl` *decl seed* [*#:expand-enums #f*] => *seed* [Procedure]

This is a fold iterator to used by `tree->udict`. It converts the multiple `init-declr` items in an `init-declr-list` of a `decl` into an a-list of multiple pairs of name and `udecl` trees with a single `init-declr` and no `init-declr-list`. That is, a `decl` of the form

```
(decl (decl-spec-list ...)
      (init-declr-list (init-declr (... "a")) (init-declr (... "b")) ...))
```

is munged into list with elements

```
("a" . (udecl (decl-spec-list ...) (init-declr (... "a"))))
("b" . (udecl (decl-spec-list ...) (init-declr (... "b"))))
```

The `/deep` version will plunge into cpp-includes. Here we generate a dictionary of all declared items in a file:

```
(let* ((sx0 (with-input-from-file src-file parse-c))
```

TODO: add enums because they are global!!, but this should be user opt

```
enum  ABC = 123 ; => ???
```

Unexpanded, unnamed enums have keys `"enum"`. Enum, struct and union def's have keys `(enum . "name")`, `(struct . "name")` and `(union . "name)`, respectively.

`munge-comp-decl` *decl seed* [*#:expand-enums #f*] [Procedure]

This will turn

```
(comp-decl (decl-spec-list (type-spec "int"))
           (comp-decl-list
            (comp-declr (ident "a")) (comp-declr (ident "b"))))
```

into

```
("a" . (comp-decl (decl-spec-list ...) (comp-declr (ident "a"))))
("b" . (comp-decl (decl-spec-list ...) (comp-declr (ident "b"))))
```

This is coded to be used with fold-right in order to preserve order in `struct` and `union` field lists.

`match-param-decl` *param-decl seed* [*#:expand-enums #f*] [Procedure]

This will turn

```
(param-decl (decl-spec-list (type-spec "int"))
            (param-declr (ident "a")))
```

into

```
("a" . (comp-decl (decl-spec-list ...) (comp-declr (ident "a"))))
```

This is coded to be used with fold-right in order to preserve order in `struct` and `union` field lists.

`gen-enum-udecl` *nstr vstr* => (*udecl ...*) [Procedure]

```
(gen-enum-udecl "ABC" "123")
=>
(udecl (decl-spec-list
        (type-spec
         (enum-def
          (enum-def-list
           (enum-defn (ident "ABC") (p-expr (fixed "123"))))))))
```

`udict-ref` *name* [Procedure]

`udict-ref-struct` *name* [Procedure]

`udict-ref-union` *name* [Procedure]

`find-special` *udecl-alist seed* => .. [Procedure]

NOT DONE

```
'((struct . ("foo" ...) ...)
  (union . ("bar" ...) ...)
  (enum . ("bar" ...) ...)
  seed)
```

`fixed-width-int-names` [Variable]

This is a list of standard integer names (e.g., `"uint8_t"`).

`typedef-decl?` *decl*) [Procedure]

`splice-declarators` *orig-declr tdef-declr* `=>` [Procedure]

Splice the original declarator into the typedef declarator. This is a helper for `expand-*-typename-ref` procecures.

`repl-typespec` *decl-spec-list replacement* [Procedure]

This is a helper for expand-decl-typerefs

`expand-typerefs` *udecl udecl-dict* [*#:keep* '()] [Procedure]

Given a declaration or component-declaration, return a udecl with all typenames (not in *keep*), struct, union and enum refs expanded. (but enums to int?)

```
typedef const int  (*foo_t)(int a, double b);
extern    foo_t    fctns[2];
=>
extern const int  (*fctns[2])(int a, double b);
```

Cool. Eh? (but is it done?) What about those w/ no init-declr? Like

```
struct foo;
struct foo  ... ;
```

`canize-enum-def-list` [Procedure]

Fill in constants for all entries of an enum list.

```
typedef int *x_t;
x_t a[10];
(spec (typename x_t)) (init-declr (array-of 10 (ident a)))
(spec (typedef) (fixed-type "int")) (init-declr (pointer) (ident "x_t"))
=>
(udecl (decl-spec-list (type-spec ...) ... (type-qual "const"))
       (init-declr (ptr-declr (pointer ...)
```

`stripdown` *udecl decl-dict* [*options*]`=>` *decl* [Procedure]

1) remove stor-spec

```
=>
```

`udecl->mspec` *udecl* [Procedure]

`udecl->mspec/comm` *udecl* [*#:def-comm* `""`] [Procedure]

Turn a stripped-down unit-declaration into an m-spec. The second version include a comment. This assumes decls have been run through `stripdown`.

```
(decl (decl-spec-list (type-spec "double"))
```

```
        (init-declr-list (
        (comment "state vector")
    =>
    ("x" "state vector" (array-of 10) (float "double")
```

`clean-field-list` *field-list* `=>` *flds* [Procedure]

Process the tagged field-list element of a struct and remove lone comments. If a field following a lone comment has no code-comment, the lone comment will be used. For example,

```
/* foo */
int x;
```

will be treated as if it was denereed

```
int x; /* foo */
```

```
(decl (decl-spec-list ...) (init-declr-list (init-declr ...) ...))
=>
((decl (decl-spec-list ...) (init-declr ...))
 (decl (decl-spec-list ...) (init-declr ...))
 ...)
```