Appendix L

Supporting discriminated unions in C

Contents

This appendix presents an ML program that reads the data descriptions from Chapters 2 to 4 and produces C declarations of types that represent the data and C functions that operate on the data. The format of the descriptions, which is inspired the Zephyr Abstract Syntax Description Language (Wang et al. 1997), is like this:

723 ⟨*example input* 723⟩≡
```
Lambda  = (Namelist formals, Exp body)
Def*    = VAL     (Name name, Exp exp)
        | EXP     (Exp)
        | DEFINE  (Name name, Lambda lambda)
        | USE     (Name)
```

For a name like Lambda, which defines a product (record), the program produces declarations like these:

```
typedef struct Lambda Lambda;
struct Lambda { Namelist formals; Exp body; };
Lambda mkLambda(Namelist formals, Exp body);
```

For a name like Def, which defines a sum, C code needs to identify which alternative of the sum is meant. This program creates a type Defalt, which identifies an alternative, as well as other declarations related to Def:

```
typedef struct Def *Def;
typedef enum { VAL, EXP, DEFINE, USE } Defalt;

Def mkVal(Name name, Exp exp);
Def mkExp(Exp exp);
Def mkDefine(Name name, Lambda lambda);
Def mkUse(Name use);

struct Def {
    Defalt alt;
    union {
        struct { Name name; Exp exp; } val;
        Exp exp;
        struct { Name name; Lambda lambda; } define;
        Name use;
    } u;
};
```

L.1 Lexical analysis

There are a few reserved symbols, a token in all upper case is a constructor, and anything else is a name. Constructors, like ML constructors, identify the alternatives in a sum type.

724a ⟨*lexical analysis* 724a⟩≡ (740) 724b▷

```
type name = string
datatype token
  = RESERVED of char
  | CONSTR   of name (* constructor *)
  | NAME     of name
```

Conversion to strings is typical.

724b ⟨*lexical analysis* 724a⟩+≡ (740) ◁724a 725a▷

```
fun tokenString (RESERVED c) = str c
  | tokenString (NAME n)     = n
  | tokenString (CONSTR c)   = c
fun isLiteral s t = tokenString t = s
```

```
tokenString : token -> string
isLiteral   : string -> token -> bool
```

The lexer converts a string to a sequence of tokens. Unlike the other languages in this book, this input language uses a C-like definition of identifiers. It also uses the C++ comment convention: a comment starts with two slashes and goes to the end of the line.

725a ⟨*lexical analysis* 724a⟩+≡ (740) ◁724b

⟨*support for streams, lexical analysis, and parsing* 644⟩

`asdlToken : token lexer`

```
val asdlToken =
  let fun validate NONE = NONE
        | validate (SOME (c, cs)) =
            case (c, streamGet cs)
              of (#"/", SOME (#"/", _)) => NONE (* comment to end of line *)
               | _ =>
                   let val msg = "invalid initial character in `" ^
                                 implode (c::listOfStream cs) ^ "'"
                   in  SOME (ERROR msg, EOS)
                   end

      fun or_ p c = c = #"_" orelse p c
      val alpha    = sat (or_ Char.isAlpha)    one
      val alphanum = sat (or_ Char.isAlphaNum) one
      fun constrOrName cs =
        (if List.all (or_ Char.isUpper) cs then CONSTR else NAME) (implode cs)
      val token =
            RESERVED <$> sat (Char.contains "(),*=|") one
        <|> constrOrName <$> (curry op :: <$> alpha <*> many alphanum)
        <|> (validate o streamGet)
  in  whitespace *> token
  end
```

L.2 Abstract syntax and parsing

There are two kinds of definitions: sums and products. The left-hand side of a definition gives a name and lets us know if the thing being defined is a pointer.

725b ⟨*abstract syntax* 725b⟩≡ (740)

`defName : def -> name`

```
type name = string
type ty   = string

type     lhs = name * {ptr:bool}
datatype rhs = SUM     of alt list
             | PRODUCT of arg list
     and alt = ALT     of name * arg list option
withtype def = lhs * rhs
     and arg = name * ty
fun defName ((n, _), _) = n
```

Our problem domain (generating C) is full of separators: for example, a function's arguments are separated by commas; assignments are separated by line breaks; and declarations are also separated by line breaks. To insert separators, we use a utility function we call `foldr1`. Function `foldr1` is a bit like the standard `foldr`, except that it inserts a binary operator *between* elements of a list. If a list contains a single element, `foldr1` returns that element unchanged. If a list is empty, and only then, `foldr1` uses its second argument.

726a ⟨*parsing* 726a⟩≡ (740) 726b▷

```
fun foldr1 f z [] = z                 foldr1 : ('a * 'a -> 'a) -> 'a -> 'a list -> 'a
  | foldr1 f _ [x] = x
  | foldr1 f z (x::xs) = f (x, foldr1 f z xs)
```

Our first use of `foldr1` will be to take a sequence of tokens like `char *` or `char *name` and turn the sequence into a string where adjacent tokens are separated by spaces. This problem is part of our first parsing function, which takes a sequence of tokens and turns it into a field. Because we permit a lone field to be anonymous, we use a heuristic to turn the sequence into a "pre-argument," which is like an `arg` except that it may not be named.

726b ⟨*parsing* 726a⟩+≡ (740) ◁726a 726c▷

```
type pre_arg = name option * ty             preArg : string list -> pre_arg error
fun preArg [x] = OK (NONE, x)
  | preArg strings = case rev strings
                       of tys as "*" :: _       => OK (NONE,      space (rev tys))
                        | name :: tys           => OK (SOME name, space (rev tys))
                        | [] => ERROR "Empty argument"
and space tys = foldr1 (fn (s, s') => s ^ " " ^ s') "" tys
```

If a constructor carries multiple fields or arguments, every one must be named. The function is Curried so that we can partially apply it, then pass the result to `map`.

726c ⟨*parsing* 726a⟩+≡ (740) ◁726b 726d▷

```
                                      nameRequired : string -> pre_arg -> arg error
fun nameRequired thing (SOME x, tau) = OK (x, tau)
  | nameRequired thing (NONE, tau) =
      ERROR ("All arguments of " ^ thing ^ " must be named")
```

A constructor carries an optional list of arguments, and for each argument, a name is also optional. If there is only one argument, and if it has no name, the argument gets the same name as the constructor, except forced to all lower case. If there is more than one argument, *all* the arguments have to have names.

726d ⟨*parsing* 726a⟩+≡ (740) ◁726c 727a▷

```
                                   toAlt : name -> pre_arg list option -> alt error
fun toAlt c (NONE) = OK (ALT (c, NONE))
  | toAlt c (SOME args) =
      let fun nameArgs [(NONE, tau)] = OK [(lower c, tau)]
            | nameArgs args = errorList (map (nameRequired c) args)
      in  nameArgs args >>=+ (fn args => ALT (c, SOME args))
      end
```

726e ⟨*utility functions* 726e⟩≡ (740)

```
val lower = String.map Char.toLower
val upper = String.map Char.toUpper
```

Finally, our parser:

727a ⟨*parsing* 726a⟩+≡ (740) ◁726d

```
name : name     parser
alt  : alt      parser
arg  : pre_arg  parser
def  : def      parser
```

```
val name        = (fn (NAME   n) => SOME n  | _ => NONE) <$>? token
val constructor = (fn (CONSTR c) => SOME c  | _ => NONE) <$>? token

fun commas p = curry op :: <$> p <*> many ("," >-- p)
fun bars   p = curry op :: <$> p <*> many ("|" >-- p)

fun product (args : pre_arg list) =
  errorList (map (nameRequired "defined type") args) >>=+ PRODUCT

val arg   = preArg <$>! (many (name <|> "*" <$ literal "*"))
val type' = pair <$> name <*> ((fn t => {ptr = isSome t}) <$> optional (literal "*"))
val args  = "(" >-- commas (arg <?> "arg") --< ")"
val alt   = toAlt <$> constructor <*>! optional args
val def   = pair <$> type' <*> "=" >-- (product <$>! args <|> SUM <$> bars alt)
```

L.3 Interface to a general-purpose prettyprinter

We want to generate C code with reasonable indentation and line breaks. Laying out text with suitable indentation and line breaks is called *prettyprinting*. The problem has a long history (Oppen 1980; Hughes 1995; Wadler 1999). The code here is based on Christian Lindig's adaptation of Wadler's prettyprinter.

The prettyprinter's central abstraction is the *document*, of type `doc`. The most basic documents are formed from strings. Subdocuments may be concatenated (`^^`) to form larger documents, and subdocuments may also be indented. (Indentation is relative to surrounding documents.) Finally, the creator of the document controls *exactly* where a line break may be introduced: the `BREAK` indicates that a break is permissible, but if the break is not taken, the prettyprinter inserts the selected string instead.

727b ⟨*algebraic laws for the prettyprinting combinators* 727b⟩≡ 727c▷

```
type doc
doc    : string -> doc
^^     : doc * doc -> doc
empty  : doc
brk    : doc
indent : int * doc -> doc
empty  : doc
```

```
doc (s ^ t) = doc s ^^ doc t
doc ""      = empty
empty ^^ d  = d
d ^^ empty  = d

indent (0, d)              = d
indent (i, indent (j, d))  = indent (i+j, d)
indent (i, doc s)          = doc s
indent (i, d ^^ d')        = indent (i, d) ^^ indent (i, d')
```

There are also laws relating to layout:

727c ⟨*algebraic laws for the prettyprinting combinators* 727b⟩+≡ ◁727b

```
layout : int -> doc -> string
```

```
layout (d ^^ d')          = layout d ^ layout d'
layout empty              = ""
layout (doc s)            = s
layout (indent (i, brk))  = "\n" ^ copyChar i " "
```

The last law, together with the laws for `indent`, are the keys to understanding the prettyprinter: `indent` affects *only* what happens to `brk`. In other words, strings aren't indented; instead, indentation is attached to line breaks.

And the last law for `layout` is a bit of a lie; the truth about `brk` is that it is not *always* converted to a newline (plus indentation):

- When `brk` is in a *vertical group*, it always converts to a newline followed by the number of spaces specified by its indentation.
- When `brk` is in a *horizontal group*, it never converts to a newline; instead it converts to a space.
- When `brk` is in an *automatic group*, it converts to a space only if the entire group will the width available; otherwise the `brk`, and *all* `brk`s in the group, convert to newline-indents.
- When `brk` is in a *fill group*, it *might* convert to a space. Each `brk` is free to convert to newline-indent or to space independently of all the other `brk`s; the layout engine uses only as many newlines as are needed to fit the text into the space available.

Groups are created by grouping functions, and for our convenience we add a line-breaking concatenate (`^/`) and some support for adding breaks and semicolons:

728 ⟨*prettyprinting combinators* 728⟩≡ (740)

```
⟨definition of doc and functions 737a⟩
infix 2 ^/
fun l ^/ r = l ^^ brk ^^ r
fun addBrk d = d ^^ brk
val semi = doc ";"
fun addSemi d = d ^^ semi
```

vgrp	:	doc -> doc
hgrp	:	doc -> doc
agrp	:	doc -> doc
fgrp	:	doc -> doc
^/	:	doc * doc -> doc
addBrk	:	doc -> doc
semi	:	doc
addSemi	:	doc -> doc

L.4 C types

The main C types we are interested in are

- Structs and unions, which represent products and sums
- Enumerations, which tag alternatives in a sum
- Pointer types
- Opaque named types (CTY)
- "Named" types, which behave just like unnamed types, except we emit typedefs for them.

A "field" of a struct or union has a type and a name. It also does double duty as an argument to a function.

729a ⟨*C types* 729a⟩≡ (740) 729b▷

fieldName : field -> name

```
type kind  = string (* struct or union *)
type tag   = string (* struct, union, or enum tag *)
datatype ctype
  = SU     of kind * tag option * field list  (* struct or union *)
  | ENUM   of tag option * name list
  | PTR    of ctype
  | CTY    of string
  | NAMED  of typedef
and        field = FIELD of ctype * name
withtype typedef = ctype * name
fun fieldName (FIELD (_, f)) = f
```

Named types can be extracted so we can emit typedefs:

729b ⟨*C types* 729a⟩+≡ (740) ◁729a 729c▷

namedTypes : ctype -> typedef list

```
fun namedTypes tau =
  let fun walk (NAMED (ty, name))  tail = walk ty ((ty, name)::tail)
        | walk (SU (_, _, fields)) tail = foldr addField tail fields
        | walk (PTR ty)            tail = walk ty tail
        | walk (CTY  _)            tail = tail
        | walk (ENUM _)            tail = tail
      and addField (FIELD (ty, _), tail) = walk ty tail
  in  walk tau []
  end
```

Tagged types, which must be defined exactly once, can also be extracted.

729c ⟨*C types* 729a⟩+≡ (740) ◁729b

taggedTypes : ctype -> ctype list

```
fun taggedTypes tau =
  let fun walk (NAMED (ty, _))                  tail = walk ty tail
        | walk (t as SU (_, SOME _, fields)) tail = foldr addField (t::tail) fields
        | walk (t as SU (_, NONE,   fields)) tail = foldr addField tail      fields
        | walk (PTR ty)                       tail = walk ty tail
        | walk (CTY  _)                       tail = tail
        | walk (t as ENUM (SOME _, _))        tail = t :: tail
        | walk (ENUM (NONE, _))               tail = tail
      and addField (FIELD (ty, _), tail) = walk ty tail
  in  walk tau []
  end
```

L.5 Prettyprinting C types

We have two ways of prettyprinting a C type:

- The *short* method refers to a `struct`, `union`, or `enum` by its tag, omitting the fields.
- The *long* method includes the fields of a `struct`, `union`, or `enum`.

The long method is used for definition, and the short method is used for everything else. The functions are mutually recursive, so they go into one big nest.

730a ⟨*prettyprinting C types* 730a⟩≡ (740) 730b▷

```
shortTypeDoc : ctype -> doc
longTypeDoc  : ctype -> doc
fieldDoc     : field -> doc
```

```
fun shortTypeDoc (SU   (kind, SOME n, _)) = doc (kind ^ " " ^ n)
  | shortTypeDoc (ENUM (SOME n, _))       = doc ("enum" ^ " " ^ n)
  | shortTypeDoc (PTR ty)                 = shortTypeDoc ty ^^ doc " *"
  | shortTypeDoc (CTY ty)                 = doc ty
  | shortTypeDoc (NAMED (_, name))        = doc name
  | shortTypeDoc (t as (SU (_, NONE, _))) = longTypeDoc t
  | shortTypeDoc (t as ENUM (NONE, _))    = longTypeDoc t
```

When we're writing a field declaration, we want the code to look nice, so if the type ends in a star (i.e., it's a pointer type), we don't put a space between the type and the field name. That way we get declarations like "`Value v;`" and "`Exp *e;`", but never anything like "`Exp * e;`", which is ugly.

730b ⟨*prettyprinting C types* 730a⟩+≡ (740) ◁730a 730c▷

```
and fieldDoc (FIELD (ty, name)) =
  let fun nonptrSpace (PTR _)       = empty
        | nonptrSpace (CTY ty)      = (case rev (explode ty) of #"*" :: _ => empty
                                                                | _ => doc " ")
        | nonptrSpace _             = doc " "
  in  shortTypeDoc ty ^^ nonptrSpace ty ^^ doc name
  end
```

In a long type declaration, we give the literals of `enum`s and the fields of `struct`s and `union`s. Otherwise it's just like a short type declaration. Auxiliary function `embrace` arranges indentation and groups so that a newline after an opening brace has extra indentation, but a newline before a closing brace does not.

730c ⟨*prettyprinting C types* 730a⟩+≡ (740) ◁730b 731a▷

```
and longTypeDoc (ENUM (tag, n :: ns)) =
      let val lits = foldl (fn (n, p) => p ^^ doc "," ^/ doc n) (doc n) ns
      in  agrp (doc "enum" ^^ tagDoc tag ^^ doc " " ^^ embrace (fgrp lits))
      end
  | longTypeDoc (SU (kind, tag, fs)) =
      let val fields = foldr1 (op ^/) empty (map (addSemi o fieldDoc) fs)
      in  agrp (doc kind   ^^ tagDoc tag ^^ doc " " ^^ embrace (agrp fields))
      end
  | longTypeDoc (NAMED (ty, _)) = longTypeDoc ty
  | longTypeDoc ty = shortTypeDoc ty

and embrace d = indent(4, doc "{" ^/ d) ^/ doc "}"
and tagDoc (SOME n) = doc (" " ^ n)
  | tagDoc (NONE)   = empty
```

The prototype for a constructor is associated with a constructor name, and it contains a result type, a function name, and a list of arguments. Function `foldr1` easily implements the C convention that an empty list of arguments is given by a prototype like `f(void)`.

731a ⟨*prettyprinting C types* 730a⟩+≡ (740) ◁730c

```
type cons_proto = name * (ctype * name * field list)   | protodoc : cons_proto -> doc |

fun protodoc (_, (result, fname, args)) =
  let fun bracket d = doc "(" ^^ d ^^ doc ")"
  in  fieldDoc (FIELD (result, fname)) ^^
      agrp (indent (4, bracket (foldr1 (fn (x, y) => x ^^ doc "," ^/ y)
                                        (doc "void")
                                        (map fieldDoc args))))
  end
```

L.6 Creating C types from sums and products

Once we have a sum or product in the form of a `def`, we convert a sum to a tagged union, which means "`struct` containing `enum` and `union`," and we convert a product to a `struct`.

Because the `ctype` representation is set up to be easy to prettyprint, not to be easy to create, we proved convenience functions for creating `struct`, `union`, and pointer types.

731b ⟨*converting sums and products to C types* 731b⟩≡ (740) 731c▷

```
                                   | anonstruct : field list -> ctype          |
                                   | anonunion  : field list -> ctype          |
                                   | struct'    : name * field list -> ctype   |
                                   | union      : name * field list -> ctype   |
                                   | withPtr    : ptr:bool * ctype -> ctype    |
fun anonstruct fields = SU ("struct", NONE, fields)
fun anonunion  fields = SU ("union",  NONE, fields)
fun struct' (name, fields) = SU ("struct", SOME name, fields)
fun union   (name, fields) = SU ("union",  SOME name, fields)
fun withPtr ({ptr}, ty) = if ptr then PTR ty else ty
```

One function is called `struct'` because `struct` is a reserved word of ML.

An argument can be converted to a field. And if an alternative in a sum carries arguments, a field is reserved to hold those arguments—for a single argument, a single field, and for multiple arguments, a structure containing them all.

731c ⟨*converting sums and products to C types* 731b⟩+≡ (740) ◁731b 732a▷

```
fun argToField (f, ty) =                 | argToField        : arg -> field        |
      FIELD (CTY ty, f)                  | altToFieldOption : alt -> field option |

fun altToFieldOption (ALT (name, NONE))        = NONE
  | altToFieldOption (ALT (name, SOME []))     = NONE
  | altToFieldOption (ALT (_,    SOME [arg])) = SOME (argToField arg)
  | altToFieldOption (ALT (name, SOME args))  =
      SOME (FIELD (anonstruct (map argToField args), lower name))
```

A product and a sum with a single alternative are treated almost identically: each becomes a structure with fields for the arguments.

- For a product, we get the fields from the arguments.
- For a sum, we have two fields: a named enumeration `alt`, which identifies which element of the sum is represented, and an anonymous union `u`, which holds the arguments (if any) carried by each alternative.

Because the enumeration in a sum is named, it will be `typedef`'d.

732a ⟨*converting sums and products to C types* 731b⟩+≡ (740) ◁731c 733a▷

```
toCtype   : def -> ctype
mapOption : ('a -> 'b option) -> 'a list -> 'b list
```

```
⟨definitions of functions mapOption and camelCase 732b⟩
val altsuffix = "alt"

fun toCtype ((n, ptr), PRODUCT args) = withPtr (ptr, struct'(n, map argToField args))
  | toCtype ((n, ptr), SUM alts) =
      let val enumname = n ^ altsuffix
          val enum = NAMED (ENUM (NONE, map (fn (ALT (n, _)) => n) alts), enumname)
          val u    = anonunion (mapOption altToFieldOption alts)
      in  withPtr (ptr, struct' (n, [FIELD (enum, altsuffix), FIELD (u, "u")]))
      end
```

Function `mapOption f` applies `f` to a list of values and returns only the results that are not `NONE`.

732b ⟨*definitions of functions* `mapOption` *and* `camelCase` 732b⟩≡ (732a) 733b▷

```
mapOption : ('a -> 'b option) -> 'a list -> 'b list
```

```
fun mapOption f =
  let fun add (x, tail) = case f x of NONE => tail | SOME y => y :: tail
  in  foldr add []
  end
```

L.7 Creating constructor functions and prototypes

Because C provides no convenient way of creating values of `struct` types, it's not enough just to emit definitions of the types: we also emit *constructor functions* for creating values of the types. Given a `PRODUCT`, we create a single constructor function. Given a `SUM`, we create a constructor function for each alternative in the sum. In both cases, when we create a function, we also create a prototype.

733a ⟨*converting sums and products to C types* 731b⟩+≡ (740) ◁732a

`toConsProtos : def -> cons_proto list`

```
fun toConsProtos (lhs as (n, {ptr}), rhs) =
  let val struct_ty = CTY ("struct " ^ n)
      val result_ty = if ptr then NAMED (PTR struct_ty, n) else CTY n
      fun toConsProto suffix rty (ALT (altname, args)) =
            (altname, (rty, "mk" ^ camelCase altname ^ suffix,
                       map argToField (getOpt (args, []))))
      fun altProtos   alts   suffix ty = map (toConsProto suffix ty) alts
      fun fieldProtos fields suffix ty = [toConsProto suffix ty (ALT (n, SOME fields))]
      fun dualProtos protos =
            protos "" result_ty @ (if ptr then protos "Struct" struct_ty else [])
  in  case rhs
        of SUM alts       => dualProtos (altProtos alts)
         | PRODUCT fields => dualProtos (fieldProtos fields)
  end
```

To get the name of the constructor function, we start with `mk`, followed by the name of the constructor in "camel case:" the first letter is upper case, as is every letter that follows an underscore. Other letters are lower case, and underscores are dropped. For example, `BOOL` is built by `mkBool`, and `USER_METHOD` would be built by `mkUserMethod`.

733b ⟨*definitions of functions* `mapOption` *and* `camelCase` 732b⟩+≡ (732a) ◁732b

```
fun camelCase n =
  let fun cap (#"_" :: cs) = cap cs
        | cap (c :: cs) = Char.toUpper c :: lower cs
        | cap [] = []
      and lower (#"_" :: cs) = cap cs
        | lower (c :: cs) = Char.toLower c :: lower cs
        | lower [] = []
  in  (implode o cap o explode) n
  end
```

Code that emits code is always complex. We begin with some auxiliary functions. Functions `isPtr` tells if a C type is a pointer type, and `defSum` tells if it is a sum.

733c ⟨*auxiliary functions for emitting a constructor function* 733c⟩≡ (735a) 734a▷

```
isPtr  : ctype -> bool
defSum : def   -> bool
```

```
fun isPtr (NAMED (ty, _)) = isPtr ty
  | isPtr (PTR _)         = true
  | isPtr _               = false
fun defSum (_, SUM _     ) = true
  | defSum (_, PRODUCT _) = false
```

The value returned by a constructor function is called the *answer*. Normally the answer is called `n`, but if the name `n` conflicts with an argument, we keep adding more `n`'s until we get a name that doesn't conflict. Value `argfields` is in scope and contains the fields that represent the arguments to the constructor function.

734a ⟨*auxiliary functions for emitting a constructor function* 733c⟩+≡ (735a) ◁733c 734b▷

```
argfields : field list
answer    : string
```

```
val answer =
  let fun isArg x =
        List.exists (fn f => fieldName f = x) argfields
      fun answerName x = if isArg x then answerName "n" ^ x else x
  in  answerName "n"
  end
```

We'd like to write code that manipulates the answer, but we don't know what the answer is going to be called. Function `ans` enables us to refer to the answer as `%` within a string.

734b ⟨*auxiliary functions for emitting a constructor function* 733c⟩+≡ (735a) ◁734a 734c▷

```
ans : string -> doc
```

```
val ans =
  doc o String.translate (fn #"%" => answer | c => str c)
```

Function `outerfield` names a field of the answer, and `innerfield` names the subfield of the inner union `u` that is associated with an argument (for a sum type only).

734c ⟨*auxiliary functions for emitting a constructor function* 733c⟩+≡ (735a) ◁734b 734d▷

```
outerfield : name  -> string
innerfield : field -> string
```

```
fun outerfield f =
  answer ^ (if isPtr result then "->" else ".") ^ f
fun innerfield arg =
  let val single = case argfields of [_] => true | _ => false
      fun select s =
        outerfield (if defSum def then
                      if single then "u." ^ s else "u." ^ lower cname ^ "." ^ s
                    else s)
  in  select (fieldName arg)
  end
```

Finally, `fieldAssignments` assigns each argument to a field of the answer.

734d ⟨*auxiliary functions for emitting a constructor function* 733c⟩+≡ (735a) ◁734c

```
fieldAssignments : doc
```

```
val fieldAssignments =
  let fun assignTo arg = concat [innerfield arg, " = ", fieldName arg, ";"]
  in  foldr1 (op ^/) empty (map (doc o assignTo) argfields)
  end
```

With these auxiliary functions in place, here is the prettyprinting document that represents the definition of a constructor function:

735a ⟨*emit info* 735a⟩≡ (740) 735b▷

consFunDoc : def -> cons_proto -> doc

```
fun consFunDoc def (proto as (cname, (result, fname, argfields))) =
  let ⟨auxiliary functions for emitting a constructor function 733c⟩
  in  vgrp (protodoc proto ^^ doc " " ^^ embrace (
             fieldDoc (FIELD (result, answer)) ^^ semi ^/  (* declare answer *)
             (if isPtr result then
                ans "% = malloc(sizeof(*%));" ^/           (* allocate answer *)
                ans "assert(% != NULL);" ^^ brk
              else
                empty) ^^
             empty ^/
             (if defSum def then  (* if sum, set tag for this constructor *)
                doc (concat [outerfield altsuffix, " = ", upper cname, ";"]) ^^ brk
              else
                empty) ^^
             fieldAssignments ^/  (* initialize all the fields *)
             ans "return %;"))  (* and return the answer *)
  end
```

L.8 Writing the output

This program's output includes chunk definitions for `noweb`. The `root` may be something like `"type definitions"`, the `language` is the language into whose implementation the generated code will be incorporated, and the `name` identifies the exact source of the chunk. (In general a language will have many sets of type definitions; the `name` identifies the source of *these* definitions.)

735b ⟨*emit info* 735a⟩+≡ (740) ◁735a 735c▷

```
fun chunkdefn (root, language, name) =
  let fun defn s = concat ["<<", s, " ((", name, "))>>="]
      fun shared "par" = true
        | shared _     = false
  in  if shared name then defn ("shared " ^ root)
      else                defn (root ^ " for \\" ^ language)
  end
```

A C `typedef` uses the same concrete syntax as a field definition, so we reuse `fieldDoc`.

735c ⟨*emit info* 735a⟩+≡ (740) ◁735b 735d▷

typedefdoc : typedef -> doc

```
fun typedefdoc (ty, name) =
  agrp (doc "typedef " ^^ fieldDoc (FIELD (ty, name)) ^^ semi)
```

We emit a `typedef` for every definition, plus additional `typedefs` for internal, named types.

735d ⟨*emit info* 735a⟩+≡ (740) ◁735c 736a▷

```
fun typedefs d =
  let val ty = toCtype d
      val typedefs = map typedefdoc ((ty, defName d) :: namedTypes ty)
  in  vgrp (foldr1 (op ^/) empty typedefs) ^^ brk
  end
```

We emit definitions for every tagged type, which in practice includes only `struct` types.

736a ⟨*emit info* 735a⟩+≡ (740) ◁735d 736b▷
```
fun structDefs d =
  let val defs = map (agrp o addBrk o addSemi o longTypeDoc) (taggedTypes (toCtype d))
  in  vgrp (foldr1 (op ^/) empty defs)
  end
```

For a function declaration, every prototype is followed by a semicolon. For a function definition, we call `consFunDoc`. Function definitions are separated by blank lines.

736b ⟨*emit info* 735a⟩+≡ (740) ◁736a 736c▷
```
fun constructProto d =
  vgrp (foldr1 (op ^/) empty (map (addSemi o protodoc) (toConsProtos d)))

fun constructorFunction d =
  let val funs = map (consFunDoc d) (toConsProtos d)
  in  vgrp (foldr1 (fn (x, y) => x ^/ empty ^/ y) empty funs) ^^ brk
  end
```

We write constructor functions to a C file, and we write definitions of four `noweb` chunks to a `.xnw` file.

736c ⟨*emit info* 735a⟩+≡ (740) ◁736b
```
fun process cname webname name lang defstream =
  let val cfile  = TextIO.openOut cname
      val webout = TextIO.openOut webname
      fun printdoc file s = TextIO.output(file, layout 75 (vgrp (agrp s^^brk)))
      val (printc, printw) = (printdoc cfile, printdoc webout)
      val defs = listOfStream defstream
      fun chunk (c, mkDoc) =
          ( printw (doc (chunkdefn (c, lang, name)))
          ; app (printw o mkDoc) defs
          )
  in  ( printc (doc "#include \"all.h\"")
      ; app (printc o constructorFunction) defs

      ; chunk ("type definitions",      typedefs)
      ; chunk ("structure definitions", structDefs)
      ; chunk ("type and structure definitions",
                                        (fn d => typedefs d ^^ structDefs d ^^ brk))
      ; chunk ("function prototypes",   constructProto)
      ; app TextIO.closeOut [cfile, webout]
      )
  end
```

L.9 Implementation of the prettyprinter

The prettyprinter is derived from one written by Christian Lindig for the C-- project, which in turn is based on Wadler's (1999) prettyprinter. The definition of `doc` simply gives the alternatives.

737a ⟨*definition of* doc *and functions* 737a⟩≡ (728) 737b▷

```
datatype doc
  = ^^     of doc * doc
  | TEXT   of string
  | BREAK  of string
  | INDENT of int * doc
  | GROUP  of break_line or_auto * doc
```

The grouping mechanisms is defined two layers. The inner layer, `break_line`, includes the three basic ways of deciding whether `BREAK` should be turned into newline-plus-indentation. The outer layer adds `AUTO`, which is converted to either `YES` or `NO` inside the implementation:

737b ⟨*definition of* doc *and functions* 737a⟩+≡ (728) ◁737a 737c▷

```
and break_line
  = NO       (* hgrp -- every break is a space *)
  | YES      (* vgrp -- every break is a newline *)
  | MAYBE    (* fgrp -- paragraph fill (break is newline only when needed) *)
and 'a or_auto
  = AUTO     (* agrp -- NO if the whole group fits; otherwise YES *)
  | B of 'a
```

Because the ML constructors can be awkward to use, we provide convenience functions.

737c ⟨*definition of* doc *and functions* 737a⟩+≡ (728) ◁737b 738▷

```
val doc    = TEXT
val brk    = BREAK " "
val indent = INDENT
val empty  = TEXT ""
infix 2 ^^

fun hgrp d = GROUP (B NO,    d)
fun vgrp d = GROUP (B YES,   d)
fun agrp d = GROUP (  AUTO,  d)
fun fgrp d = GROUP (B MAYBE, d)
```

The layout function converts a document into a string. It turns out to be easier to understand the code if we solve a more general problem: convert a *list* of documents, each of which is tagged with a *current indentation* and a *break mode*.[1] Making the input a tagged list makes most of the operations easy:

- If we remove a `d ^^ d'` from the head of the list, we put back `d` and `d'` separately.
- If we remove a `TEXT s` from the head of the list, we add `s` to the result list.
- If we remove an `INDENT (i, d)` from the head of the list, we replace it with `d`, appropriately tagged with the additional indentation.
- If we remove a `BREAK` from the head of the list, we may or may not add a newline and indentation to the result, depending on the break mode and the space available.
- If we remove a `GROUP(AUTO, d)` from the head of the list, we tag `d` with either `Flat` or `Break`, depending on space available, and we put it back on the head of the list.
- If we remove any other kind of `GROUP(B mode, d)` from the head of the list, we tag `d` with `mode` and put it back on the head of the list.

Function `format` takes a total line width, the number of characters consumed on the current line, and a list of tagged `docs`. "Putting an item back on the head of the list" is accomplished with internal function `reformat`.

738 ⟨*definition of* doc *and functions* 737a⟩+≡ (728) ◁737c 739a▷

```
                 format : int -> int -> (int * break_line * doc) list -> string list
  fun format w k [] = []
    | format w k (tagged_doc :: z) =
        let fun copyChar 0 c = [] | copyChar n c = c :: copyChar (n-1) c
            fun addString      s = s :: format w (k + size s) z
            fun breakAndIndent i = implode (#"\n" :: copyChar i #" ") :: format w i z
            fun reformat item = format w k (item::z)
        in  case tagged_doc
              of (i,b, x ^^ y)         => format w k ((i,b,x)::(i,b,y)::z)
               | (i,b,TEXT s)          => addString s
               | (i,b,INDENT(j,x))     => reformat (i+j,b,x)
               | (i,NO, BREAK s)       => addString s
               | (i,YES,BREAK _)       => breakAndIndent i
               | (i,MAYBE, BREAK s)    => if fits (w - k - size s, z)
                                            then addString s
                                            else breakAndIndent i
               | (i,b,GROUP(AUTO, x))  => if fits (w - k, (i,NO,x) :: z)
                                            then reformat (i,NO,x)
                                            else reformat (i,YES,x)
               | (i,b,GROUP(B break,x)) => reformat (i,break,x)
        end
```

[1] And for efficiency, I make the result a list of strings, which are concatenated at the very end. This trick is important because repeated concatenation has costs that are quadratic in the size of the result; the cost of a single concatenation at the end is linear.

Decisions about whether space is available are made by the `fits` function. It looks ahead at a list of documents and says whether *everything* up to the next possible break will fit in `w` characters.

739a ⟨*definition of* `doc` *and functions* 737a⟩+≡ (728) ◁738 739b▷

```
and fits (w, []) = w >= 0          fits : int * (int * break_line * doc) list -> bool
  | fits (w, tagged_doc::z) =
      w >= 0 andalso
      case tagged_doc
       of (i, m,      x ^^ y)       => fits (w, (i,m,x)::(i,m,y)::z)
        | (i, m,      TEXT s)       => fits (w - size s, z)
        | (i, m,      INDENT(j,x)) => fits (w, (i+j,m,x)::z)
        | (i, NO,     BREAK s)      => fits (w - size s, z)
        | (i, YES,    BREAK _)      => true
        | (i, MAYBE, BREAK _)       => true
        | (i, m,      GROUP(_,x))  => fits (w, (i,NO,x)::z)
```

If we reach a mandatory or optional `BREAK` before running out of space, the input fits. The interesting policy decision is for `GROUP`: for purposes of deciding whether to break a line, all groups are considered without line breaks (mode `NO`). This policy ensures that we will break a line in an outer group in order to try to keep documents in an inner group together on a single line.

The layout function takes the problem of laying a single document and converts it to an instance of the more general problem: wrap the document in an `AUTO` group (so that lines are broken optionally); tag it in `NO`-break mode with no indentation; put it in a singleton list; and format it on a line of width `w` with no characters consumed.

739b ⟨*definition of* `doc` *and functions* 737a⟩+≡ (728) ◁739a

```
fun layout w doc = concat (format w 0 [(0, NO, GROUP (AUTO, doc))])
```

L.10 Putting everything together

740 ⟨*asdl.sml* 740⟩≡

```
⟨utility functions 726e⟩
⟨lexical analysis 724a⟩
⟨abstract syntax 725b⟩
⟨parsing 726a⟩
⟨prettyprinting combinators 728⟩
⟨C types 729a⟩
⟨prettyprinting C types 730a⟩
⟨converting sums and products to C types 731b⟩
⟨emit info 735a⟩
val asdlSyntax = (asdlToken, def <?> "definition")
val defs = reader asdlSyntax noPrompts ("standard input", streamOfLines TextIO.stdIn)

(*
val getDef = astReader(rdr, asdlSyntax)
fun loop defs' = loop (getDef() :: defs') handle EOF => rev defs'
*)
fun usage () = concat ["Usage: ", CommandLine.name(), " cfile nwfile name language\n"]

val _ = case CommandLine.arguments ()
          of [c, web, name, lang] => process c web name lang defs
           | [base, name, lang] => (* legacy usage *)
               process (base ^ "-code.c") (base ^ ".xnw") name lang defs
           | _ => TextIO.output(TextIO.stdErr, usage())
```