UNPL 1427

Bell Laboratories **Cover Sheet for Technical Memorandum**

Title- **Semantics of the C programming language, part 2: declarations**

Date- **March 2, 1979**

TM- **79-1271-06**

Other Keywords-

Author **Ravi Sethi** | Location **MH 2C-519** | Extension **4006**

Charging Case- **39199**
Filing Case- **39199-11**

ABSTRACT

Declarations allow meaning to be associated with identifiers. The syntax and informal semantics of declarations are introduced through examples. Types are treated as abstract entities. One purpose of a data declaration is to associate a type with an identifier. Depending on this type, storage is then reserved for the identifier. Since a structure may contain a pointer to itself, circularly defined types must be dealt with. An understanding of type determination, storage management, and the dynamics of statement execution is required to give the meaning of function declarations.

Pages Text	22	Other	3	Total	25
No. Figures	0	No. Tables	0	No. Refs.	14

BELL TELEPHONE LABCRATORIES, INC.

DISTRIBUTION
(REFER GEI 13.9-3)

TM-79-1271-6

COMPLETE MEMORANDUM TO

CORRESPCNDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FCR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE PILE COPY
(FORM E-1328)

10 REFERENCE CCPIES

<AHO,ALFRED V
<BALENSON,CHRISTINE M
<BARON,ROBERT V
<BECFER,RICHARD A
BROWN,W STANLEY
<CHEN,STEPHEN
<CHERRY,LORINDA L
<FELDMAN,STUART I
<FRASER,A G
<GOLDSTEIN,A JAY
<GRAHAM,R I
+HANNAY,N B
<JOHNSON,STEPHEN C
<KEESE,W M
<KERNIGHAN,BRIAN W
<LUDERER,GOTTFRIED W E
<MARANZANO,J F
<MARKY,GERALDINE A
+MC DONALD,H S
<MC GILI,R
MCILRCY,M DOUGLAS
<MENNINGER,R E
<MORGAN,SAMUEL P
+PRIM,R C
<RALEIGH,T M
<RIDDLE,GUY G
<SCHLEGEL,C T
<SETHI,RAVI
<STORER,JAMES A
<SZYMANSKI,THCMAS G
TERRY,MILTCN E
<WEINBERGER,PETER J
<YANNAKAKIS,MIHALIS
33 NAMES

COVER SHEET CNLY TC

CORRESPONDENCE FILES

4 COPIES PLUS CNE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ACKERMAN,A FRANK
ACKERMAN,J T
AHRENS,RAINER B
ALCALAY,D

COVER SHEET CNLY TC

AMRON,IRVING
ANDERSON,FREDERICK L
ANDERSON,KATHRYN J
ANDERSON,MILTCN M
APPELBAUM,MATTHEW A
ARAZY,URI
ARMSTRONG,D B
ARNOLD,GEORGE W
ARNOLD,PHYLLIS A
ARNCLD,THCMAS P
ASTHANA,ABHAYA
ATAL,BISHNU S
BAILY,DAVID E
BALLANCE,ROBERT A
BARCLAY,DAVID K
BAROFSKY,ALLEN
BASEIL,RICHARD J
BAUER,ANDREW E
BAUER,BARBARA T
BAUER,HELEN A
<BAUGH,C R
BEDNAR,JOSEPH A,JR
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BERGH,A A
BERGLAND,G D
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BEYER,JEAN-DAVID
BIANCHI,M H
BICKFORD,NEIL B
BILLINGTON,MARJCRIE J
BILOWOS,R M
<BIREN,IRMA B
BISHOP,VERONICA L
BLAZIER,S D
BLINN,J C
BLOSSER,PATRICK A
BLUE,JAMES L
BLUMER,THOMAS P
BLUM,MARION
BOCKUS,ROBERT J
BCDEN,F J
BOEHM,KIM R
BOGART,THOMAS G
BOIVIE,RICHARD H
BCNANNI,L E
BORDELON,EUGENE F
BORISON,ELLEN A
BCURNE,STEPHEN R
BOWERMAN,REBECCA ELAINE
BOWYER,L RAY
<BOYCE,W M
BOYER,PHYLLIS J
BRADLEY,M HELEN
BRADLEY,R H
BRAM,ALAN
<BRANDT,RICHARD B
BRAUNE,DAVID P
BRIGGS,GLCRIA A
BROSS,JEFFREY D

COVER SHEET ONLY TO

BROVMAN,INNA
BRCWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BRCWN,W R
BULLEY,R M
BURG,F M
BURKE,MICHAEL E
BURNETTE,W A
<BURNETT,DAVID S
BUROFF,STEVEN J
BURROWS,THOMAS A
BUTLETT,DARRELL L
BYRNE,EDWARD R
CAMPBELL,JERRY H
<CANADAY,RUDD H
CANNCN,LAYNE W
CARTER,DONALD H
CASPERS,BARBARA E
CAVINESS,JOHN D
CERMAK,I A
CHAI,D T
CHAMBERS,B C
<CHAMBERS,J M
CHAYUT,IRA G
CHENG,Y
CHEN,E
CHESSON,GREGORY L
CHEUNG,ROGER C
CHRISTENSEN,S W
CHRIST,C W,JR
CHUNG,MICHAEL
CLARK,DAVID L
CLAYTON,D P
CLINE,LAUREL M I
COBEN,ROBERT M
COCHRAN,ANITA J
<COLE,LOUIS M
COLE,MARILYN C
CCLLICCTT,R B
CONDON,J H
CCNKLIN,DANIEL L
CCNNERS,RONALD R
CCCPER,ARTHUR E
COOPER,MICHAEL H
CCSTELLC,PETER E
COTTRELL,JENNIE L
CRAGUN,DONALD W
CRISTOFOR,EUGENE
<CROME,L L
CRUPI,JOSEPH A
DAVIS,R DREW
DE FAZIO,M J
DE GRAAF,D A
DE TREVILLE,JOHN D
DEAN,JEFFREY S
DENNY,MICHAEL S
DENSMORE,SUSAN
DIB,GILBERT
DICKMAN,BERNARD N
<DIMMICK,JAMES O
DINEEN,THOMAS J
DITZEL,DAVID R

COVER SHEET ONLY TC

DOLOTTA,T A
DONNELLY,MARGARET M
DOWDEN,DOUGLAS C
DRAKE,LILLIAN
D'ANDREA,LOUISE A
DUCHARME,ROBERT LAWRENCE
DUFFY,F P
DUGGER,DONALD D
DUMAIS,VALERIE
DUNCANSON,ROBERT L
DWYER,T J
DYER,MARY E
EIGEN,D J
EISEN,STEVEN R
<EITELBACH,DAVID L
ELDREDGE,BARBARA D
ELLIOTT,RUBY J
ELLIS,DAVID J
ELY,T C
EPLEY,ROBERT V
ESCOLAR,CARLOS
ESSERMAN,ALAN R
FABISCH,M P
FABRICIUS,WAYNE N
FAGAN,EDWARD C
FAIRCHILD,DAVID L
FEDERICO,FRANK W
FEDER,J
FELLIN,JEFFREY K
FERIDUN,K K
FINUCANE,J J
FISCHER,HERBERT B
FISCHER,MICHAEL T
FISHER,EDWARD R
FISHMAN,DANIEL H
FLANDRENA,R
FLEISLEBER,RAYMOND C
FONG,K T
FORTNEY,V J
FCUNTOUKIDIS,A
FOWLER,GLENN D
FOWLER,H EUGENE
FOWLKES,EDWARD B
FOX,PHYLLIS A
FOY,J C
FRANK,AMALIE J
FRANK,RUDOLPH J
FREEMAN,K G
FREEMAN,MARTIN
FROST,H BONNELL
FRUCHTMAN,BARRY
GABBE,JOHN D
GADENZ,RENATO N
GALLANT,R J
GANA,JORGE L
GARST,BLAINE,JR
<GATES,G W
GAY,FRANCIS A
GEARY,M J
GEERS,T J,JR
GEORGEN,MICHAEL R
GEPNER,JAMES R

COVER SHEET ONLY TO

GERSHMAN,ANATCLE V
GEYLING,F T
GIBB,KENNETH R
GIBSON,J C
<GILLETTE,DEAN
GIMPEL,J F
<GITHENS,J A
GITHENS,JAY L
<GLASSER,ALAN L
GLUCK,F G
<GNANADESIKAN,R
GOGUEN,N H
GOLABEK,RUTH T
GOLDBERG,HAROLD JEFFREY
GCRDON,MOSHE B
<GORMAN,J E
GORTON,D R
GRAYSON,C F,JR
GREENBAUM,HOWARD J
GREENLAW,R L
GROSS,ARTHUR G
GRUENWALD,JCHN
GRZELAKOWSKI,MAUREEN E
GUIDI,PIER V
GUSTAVSON,J H
GUTTMAN,NEWMAN
HAFER,E H
HAIGHT,R C
HAISCH,H F,JR
HALEY,CHARLES B
HALE,A L
HALLIN,THOMAS G
HALL,ANDREW D,JR
HALL,MILTCN S,JR
HALL,WILLIAM G
HALPIN,T
HAMILTON,LINDA L
<HAMILTON,PATRICIA A
<HANNAH,JUDY R
<HARKNESS,CARCL J
HARTMAN,DON W
HARUTA,K
HAUSE,A DICKSON
<HAWKINS,DONALD T
<HAYDEN,DONALD F,JR
HERGENHAN,C B
<HESSELGRAVE,MARY R
HOCHBERG,GLENN A
HOLTMAN,JAMES F
HCRWEDEL,J H
HCUGHTCN,THOMAS F
HOWARD,PHYLLIS A
HCYT,WILLIAM F
HC,DON T
HO,JENNY
HO,TIEN-LIN
HSU,TAU
HUBER,RICHARD V
HUNNICUTT,C F
HUNT,JAMES W
IPPOLITI,O D
IRVINE,M M
...
592 TOTAL

+ NAMED BY AUTHOR   > CITED AS REFERENCE   < REQUESTED BY READER   (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATICNAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION...

COMPLETE MEMO TO:
127-SUP

COVER SHEET TO:
12-DIR   13-DIR   127

COPLGP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE

HO CORRESPONDENCE FILES
HO 5C101

TM-79-1271-6
TOTAL PAGES 25

TO GET A CCMPLETE CCPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE CUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELCPE.
4. INDICATE WHETHER MICROFICHE CR PAPER IS DESIRED.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY   ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: Semantics of the C programming language, part 2: declarations
Case- 39199 -- File- 39199-11

date: March 2, 1979

from: Ravi Sethi

TM: 79-1271-06

MEMORANDUM FOR FILE

1. Introduction

"It has been remarked to me (to my great regret I cannot remember by whom ...) that once a person has understood the way in which variables are used in programming, he has understood the quintessence of programming [dij72, pp.11]." This remark has particular relevance to understanding declarations in a programming language.

By "declarations" we mean the part of a programming language that allows meaning to be associated with identifiers. Identifiers are used not only to refer to basic values like characters, and data structures like arrays, but also to executable "functions", which take parameters and return values. A discussion of the meaning of declarations must therefore address issues suggested by the following phrases: basic and derived types; data declarations; type determination; block structure; storage allocation; function declarations.

Rather than assume familiarity with C [ker78], we will introduce declarations in the language through a sequence of examples, covering: program structure and where declarations can occur, §1.1; data declarations and the association of types with identifiers, §1.2; identifiers used as synonyms for types, §1.3.

In this introductory section, the terms "type" and "location" will be used informally. See §2 and §5.1, respectively, for precise definitions. Think of there being a set Ty, whose elements are called "types". Included in Ty are basic types like "integer", and derived types like "array of 8 integers", which is distinct from the type "array of 7 integers". One of the purposes of a data declaration will be to associate a type with an identifier. The term "location" corresponds to a storage cell in a machine, except that a location can hold any basic value. A location will be associated with each identifier representing a basic value. This basic value will be determined from the identifier in two stages: first the location for the identifier will be determined, and then the value held in the location will be looked up.

1.1. *Program structure.* Here we will suggest the syntax of declarations by discussing the following program fragment. (See appendix A for details of the syntax.)

```
int n = 3;
char select(x,c,d)
int x; char c; char d;
{
    char e;
    if (x>n) e = c; else e = d;
    return(e);
}
```

We will take a program in C to consist of a sequence of data declarations like

```
int n=3;
```

followed by a sequence of one or more function declarations. C-functions like `select` are similar to functions and subroutines in Fortran, or to procedures in Pascal, except that C-function declarations cannot be nested. As in Algol 60, the `char` in

```
char select(x,c,d)
```

specifies that `select` is a C-function that *returns* a character. Declarations of the formal parameters `x`, `c`, and `d` precede the body of the C-function. In the formal semantics it will be convenient to have exactly one identifier in each data declaration, but lists of identifiers will be allowed in examples of declarations.

The declaration of identifier `n` is *external* to all function declarations. External identifiers can be referenced inside any function without being explicitly redeclared. Inside a C-function we assume that any data declarations like that of `e`, precede all statements within the function.[1]

An entire program will be assumed to appear in one place, and issues related to separate compilation of functions, or distributing a program across source files will not be dealt with.

1.2. *Data declarations.* The syntax in C for indicating the type of an identifier is a generalization of the mechanism used to declare arrays in many languages. For example, the following declaration specifies that `ab` is an array of 7 integer elements.

```
int ab[7];
```

The construction `ab[7]` is an instance of a "declarator".

Declarators. The syntax of an identifier declaration mimics the syntax of expressions in which the identifier might appear. For example, suppose that identifier `x` has type integer, and that `px` is a pointer, created in some as yet unspecified way. The unary operator `&` is such that the statement

```
px = &x;
```

assigns the location of `x` to the identifier `px`; `px` is now said to "point" to `x`. The unary operator `*` applied to a location, gives the value in the location. Thus if `y` also has type integer, then

```
y = *px;
```

assigns to `y` the value in the location that `px` points to. So the sequence

```
px = &x; y = *px;
```

assigns the same value to `y` as does

```
y = x;
```

Declaration of the identifiers `x`, `y`, and `px` can be done as follows:

```
int x,y; int *px;
```

The declaration of `x` and `y` is reasonably obvious, but that of `px` invites comment. The

[1] C allows a `goto` to jump anywhere within the current function, even if the jump is into the middle of a compound statement containing declarations. The compilers for C [joh78,rit78a] resolve the issue of jumps into blocks by preprocessing blocks away: declarations of local identifiers are moved to the head of a function declaration after suitably renaming identifiers and converting initializations into explicit assignments. In the next version of this sequence of reports on the semantics of C we will allow declarations within compound statements, by restricting `goto` statements. At the moment, blocks are assumed to have been preprocessed away.

declaration

```
        int *px;
```

says that the construction `*px` is an integer: that is, when `px` occurs in the context `*px`, it is equivalent to an identifier of type integer. This reasoning is useful in all cases involving complex declarations. For example,

```
        float xyz[3][5];
```

says that, in an expression, `xyz[m][n]` represents a value of type float. Then, `xyz[m]` must represent an array of 5 elements of type float. Similarly, `xyz` must represent an array of 3 subarrays; each subarray being an array of 5 elements of type float.

The use of the symbols `()` should be clarified by

```
        int f();
```

which declares a function `f` returning an integer. Note that the number, or type, of the operands of `f` is not specified by the declaration.[2]

The semantic rules for declarators appear in §3.

Structure declarations. The only derived types that are not mentioned in the above discussion of declarators are structures and unions. A structure is an object consisting of a sequence of named members. Each member may have any type. Unions are similar to structures, with the exception that at any given time a union may hold just one of its members. For the moment we will talk only of structures.

The following declaration associates a type with the identifier `complex`.

```
        struct complex {double re; double im;};
```

Identifiers like `complex` will be referred to as *structure tags*. The structure tag `complex` is subsequently used to declare other identifiers. The declaration

```
        struct complex z, *zp;
```

declares `z` to be a structure of the given sort and `zp` to be a pointer to a structure of the given sort.

As another example, the structure tag `tnode` is declared by

```
        struct tnode {
            char tword[20];
            int count;
            struct tnode *left, *right;
            };
```

to have type: structure consisting of an array of 20 characters, an integer, and two pointers to similar structures. The identifiers `tword`, `count`, `left`, and `right` are the names of the four *fields* or *members* of such structures.

The declaration of a structure tag and the subsequent use of that structure tag as a type specifier can be combined, as in

```
        struct complex {double re; double im;} x,y,z;
```

The structure tag must always be included and becomes part of the type corresponding to the structure.[3] For example, the type of `x`, `y`, and `z`, in

[2] Changes to C that are under consideration would permit the specification of the types of the arguments as well as the type of the result of a function.

[3] The C reference manual [ker78] allows a structure tag to be dropped entirely, raising the question whether `root` and `pole` below have the same type.

```
struct complex {double re; double im;} x,y,z;
```

will be: structure with tag `complex` containing fields `re` and `im`, both of type double.

The order in which the fields appear is significant.

The semantic rules for determining the type of a structure appear in §4.

1.3. *Typedef.* `typedef` declarations do not reserve storage, but instead declare synonyms for types which could be specified another way. The syntax of `typedef` declarations is just like the syntax of declarations that do reserve storage: the difference is that rather than reserving storage of type, say, *t* for the identifier within a declarator, the identifier becomes a synonym for type *t*. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is integer, that of `metricp` is pointer to integer, and that of `z` is the specified structure. `zp` is a pointer to the specified structure.

`typedef` does not introduce brand new types: in the example above `distance` is considered to have exactly the same type as any other `int` identifier.

The semantic rules for `typedef` declarations appear in §4.

2. Types

2.1. *Machine based types.* C supports several basic types of objects, including characters, various sizes of integers, floating point numbers, and enumerations of constants, §2.2. The exact list of basic types is machine and compiler dependent, but the following type-specifiers are accepted by all implementations.

machine_based_specifier:

```
    char
    short
    int
    long
    unsigned
    short int
    long int
    unsigned int
    float
    double
    long float /* double and long float are synonyms */
```

```
struct {double re; double im;} root;
struct {double re; double im;} pole, bode;
```

The view taken by the compilers for C [joh78,rit78a] is that two identifiers representing structures have the same type if and only if they are declared using the same structure tag, or they appear in the same declaration. Thus, `pole` and `bode` above have the same type, but `root` and `pole` do not. A human engineering justification is given for this view: even if two identifiers start out having similar declarations, modifications to the program may change one, but not the other declaration. If we want the identifiers to have the same type, it is better to either declare them together, or use the same structure tag while declaring them.

If the type-specifier is missing from a declaration, it is taken to be `int`. Some implementations will accept one or more of the following type-specifiers:

```
unsigned short
unsigned long
unsigned char
```

2.2. *Enumerations.* Enumerations of constants are analogous to the scalar types of Pascal. Consider for example

```
enum grade {kabinett,spatlese,auslese} w;
```

The identifiers `kabinett`, `spatlese`, and `auslese` are declared as constants. These constants are the only values that the identifier w may have. `grade` is called the *enumeration-tag* of the type of w. The enumeration-tag may subsequently be used to declare other identifiers:

```
enum grade v;
```

The syntax of an enumeration specifier is as follows:

enumeration_specifier:
  `enum` *identifier* { *identifier-list* }
  `enum` *identifier*

Enumeration tags and identifiers must all be distinct. Identifiers of a given enumeration type have a type distinct from objects of all other types.

2.3. *Basic type specifiers.*

basic_specifier:
  machine_based_specifier
  enumeration_specifier

2.4. *Derived types.* Derived types are constructed from the basic types in the following ways:

arrays of members of a given type;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing a sequence of members of various types;

unions capable of containing any one of several members of various types.

In general these methods of constructing objects can be applied recursively. Not all the possibilities suggested above are actually permitted. The restrictions are as follows:

there are no arrays of functions, although there may be arrays of pointers to functions;

functions may not return arrays, unions, or functions, although they may return pointers to such things;

a structure or union may not contain a function, but it may contain a pointer to a function.

The above restrictions will not be checked in the semantic rules that will be given. Such checking can easily be added to the rules, or can be specified separately.

2.5. *Domain of types.* The domain **Ty** of types will be the sum of the domain **Tb** of basic types and summand domains corresponding to each way of constructing derived types.

Before giving the exact structure of **Ty**, let us consider the functions that will "construct new types from old". *arr* maps an integer *n* and a type *t* to a new type *t'* corresponding to array of *n* members of type *t*; *fn_ret* maps a type *t* to *t'* corresponding to function returning type *t*; *point* maps a type *t* to *t'* corresponding to pointer to type *t*. *str*, applied to a structure tag and a list of pairs of member identifiers and associated types, yields a type *t'* corresponding to a structure with the appropriate tag and members. *uni* yields a derived union type and is similar to *str*.

Intuitively, two types *t* and *t'* are equivalent if and only if they are constructed in the same way from the same basic types. Thus the domain **Ty** will correspond (loosely) to the set of expressions over the above operators and the elements of **Tb**.

$$\begin{aligned}\mathbf{Ty} = \ & \mathbf{Tb} \\ & + \{\mathbf{array}\} \times \mathbf{N} \times \mathbf{Ty} \\ & + \{\mathbf{fn\_returning}\} \times \mathbf{Ty} \\ & + \{\mathbf{pointer}\} \times \mathbf{Ty} \\ & + \{\mathbf{struct}\} \times \mathbf{Ide} \times [\mathbf{Ide} \times \mathbf{Ty}]^{+} \\ & + \{\mathbf{union}\} \times \mathbf{Ide} \times [\mathbf{Ide} \times \mathbf{Ty}]^{+}\end{aligned}$$

For clarity, single point domains like {**array**} have been included in the above specification of **Ty**.

3. Declarators

Declarators play a central role in the process of associating a type with an identifier. Simple examples of declarators and their use were given in §1.2. The meaning of declarators can be explained by considering the declaration

```
float xyz[3][5];
```

The syntax in §3.1 will parse the declarator `xyz[3][5]` as follows:

```
( ( xyz ) [3] ) [5]
```

In order for `xyz` to be an array of 3 subarrays of 5 elements each, we must read the fully parsed declarator inside out.

As further examples, consider the declarations

```
char *( fcp() ); int ( *pfi )();
```

From the discussion in §1.2, a construction like `*( fcp() )` can appear in any context where a character is expected, and a construction like `( *pfi )()` can appear in any context where an integer is expected. Reading the declarators inside-out, `fcp` is a function returning a pointer to — from the type specifier `char` — a character. Similarly, `pfi` is a pointer to a function returning an integer.

The meaning of declarators is given in §3.2.

3.1. *Syntax.*

Data declarations have the form

declaration:
    type_specifier init_declarator ;

init_declarator:
    declarator
    declarator = *initializer*

The declarator contains the identifier being declared. Initializers will be discussed in §5.4.

Declarators have the syntax:

```
declarator:
    identifier
    ( declarator )
    declarator [ constant ]
    declarator ()
    * declarator
```

The * operator on a declarator has lower precedence than all other operators, so `*fcp()` will be parsed as `*(fcp())`.

3.2. *Meaning of declarators.* After a declarator has been examined, in addition to uncovering the embedded identifier, the type of this identifier will also be known. The meaning of a declarator will therefore be a function from a type to an identifier and its type.[4]

See §2.5 for the operators *arr*, *fn_ret*, and *point*.

```
⟦declarator⟧(t):  (Ide,Ty)
  | identifier
      — (identifier, t)
  | ( declarator )
      — ⟦declarator⟧(t)
  | declarator [ constant ]
      — let   n = ⟦constant⟧;
              t' = arr(n,t);
        in ⟦declarator⟧(t')
  | declarator ()
      — let   t' = fn_ret(t)
        in ⟦declarator⟧(t')
  | * declarator
      — let   t' = point(t)
        in ⟦declarator⟧(t')
```

3.3. *Abstract declarators.* In two contexts (to specify type conversions explicitly within expressions, and as an argument of the built-in operator `sizeof`) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declarator without an embedded identifier.

```
type_name:
    type_specifier abstract_declarator
```

[4] The meaning of a declarator should really be a function from a type and an environment to an identifier and its type. The environment is needed to evaluate constant expressions, which can be used to specify the number of elements in an array. Within a constant expression, the `sizeof` operator can be applied to "an object", and the environment is needed to determine the size of this "object". By insisting that constants rather than constant expressions be used in array declarations, we eliminate the need for knowing the type of each identifier during the process of determining the types of structure tags in §4.

abstract_declarator:
    /* empty */
    (*abstract_declarator*)
    abstract_declarator [*constant*]
    abstract_declarator ( )
    * *abstract_declarator*

To avoid ambiguity, in the construction

(*abstract_declarator*)

the abstract declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the place in the abstract_declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the missing identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types integer, pointer to integer, array of 3 pointers to integers, pointer to an array of 3 integers, function returning pointer to integer, and pointer to function returning an integer.

The semantic rules for abstract declarators are very similar to those for declarators.

```
⟦abstract_declarator⟧(t):  Ty
  | /* empty */
      — t
  | ( abstract_declarator )
      — ⟦abstract_declarator⟧(t)
  | abstract_declarator [ constant ]
      — let   n = ⟦constant⟧;
              t' = arr(n,t);
        in ⟦abstract_declarator⟧(t')
  | abstract_declarator ( )
      — let   t' = fn_ret(t)
        in ⟦abstract_declarator⟧(t')
  | * abstract_declarator
      — let   t' = point(t)
        in ⟦abstract_declarator⟧(t')
```

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left, *right;
    };
```

Note that `tnode` appears as a type specifier in the declaration of the members `left` and `right`. Thus the type represented by `tnode` depends on itself.

As another example of circularly defined types, consider the structure tags **x** and **y** declared by:

```
struct x {
    int count;
    struct y *py;
    };
struct y {
    int count;
    struct x *px;
    };
```

`typedef` names, §1.2, can be used in declarations of structure members, and conversely, structure tags can be used as type specifiers in `typedef` declarations. Thus the types of structure tags and `typedef` names must be determined together. From the syntax, structure tags are part of *type specifiers*, which will be discussed in §4.1-2. Types are associated with `typedef` names by *declarations*, which will be discussed in §4.3.

An example in §4.4 clarifies the handling of circularly defined types. The example also suggests the need for the valuation **dz** in §4.5, which sets up the initial environment on function or "block" entry.

The scope of a tag **x** in a C-function is the current function: **x** may be used for other purposes in a declaration external to all functions. Thus, on entering a function body, we must distinguish between the type of **x** within the function from the type of **x** outside the function. The purpose of valuation **dz** in §4.5 is to "reset" the types of all structure tags and `typedef` names within the current function by entering $\perp$ for all such identifiers.

Finally, §4.6 suggests how the valuations of this section interact to determine the types of structure tags and `typedef` names. Complete details will be given in §6 on function declarations.

4.1. *Syntax of type specifiers.* The types of unions are determined just like the types of structures, so semantic rules for unions will not be given.

```
type_specifier:
    basic_specifier
    struct identifier { member_decl_p }
    struct identifier
    identifier          /* typedef name */

member_decl:
    type_specifier declarator ;

member_decl_p:
    member_decl
    member_decl_p member_decl .
```

4.2. *Meaning of type specifiers.* A type specifier must clearly yield a type. In order to determine this type, we may need to refer to the environment for the types associated with previously declared structure and union tags (or with identifiers in `typedef` declarations). Moreover, since types must be associated with structure tags within a type specifier, a type specifier may change the environment. Thus the meaning of a type specifier will be a function from an environment to a type and a new environment.

$\mathrm{te}[\![\mathit{type\_specifier}]\!](e)\colon\ (\mathbf{Ty},\mathbf{En})$

Member declarations. Before we can give the type of a structure, we need to determine the member identifiers and their associated types. The members of a structure are generated by the nonterminal *member_decl_p*. A sequence, *pairlist*, containing pairs of member identifiers and their associated types will be yielded by *member_decl_p*. At the same time, type specifiers embedded within member declarations might change the environment, so the meaning of *member_decl_p* will be a function from an environment to a pair list and a possibly new environment. The pair list will eventually become one of the arguments of the operator *str*, which constructs a type from a structure tag and such a pair list.

```
me⟦member_decl_p⟧(e):  ((Ide,Ty)⁺,En)
  | member_decl
       — me⟦member_decl⟧(e)
  | member_decl_p  member_decl
       — let   (pairlist,e′) = me⟦member_decl_p⟧(e);
               (pair,e″) = me⟦member_decl⟧(e);
         in  (pairlist · pair,e″)

me⟦member_decl⟧(e):  ((Ide,Ty),En)
  | type_specifier  declarator ;
       — let   (t,e′) = te⟦type_specifier⟧(e);
               pair = ⟦declarator⟧(t);
         in  (pair,e′)
```

The meaning of a type specifier can now be given. A basic specifier, §2.3, can either specify a machine based type or an enumeration. The meaning of a basic specifier will be assumed to be an appropriate element of the domain **Tb** of basic types, and will not be discussed any further. Finally, if a type specifier is just an identifier, then a type has already been associated with this identifier using a `typedef` declaration, §4.3, and can be looked up in the environment.

```
te⟦type_specifier⟧(e):  (Ty,En)
    | basic_specifier
        — (⟦basic_specifier⟧, e)
    | struct identifier { member_decl_p }
        — let    (pairlist, e′) = me⟦member_decl_p⟧(e);
                 t = str(identifier, pairlist);
                 e″ = e′[t/identifier];
          in  (t, e″)
    | struct identifier
        — (e(identifier), e)
    | identifier            /* typedef name */
        — (e(identifier), e)
```

4.3. *Data and* `typedef` *declarations.* In data declarations like

```
    struct complex z, *zp;
```

note that the types of identifiers like `z` and `zp` cannot influence the type of a structure tag or `typedef` name.[5] We can therefore ignore declarators containing identifiers like `z` and `zp` while determining the types of structure tags and `typedef` names. However, the type specifier in a data declaration may contain a structure tag so the effect on the environment of type specifiers must be propagated.

A `typedef` declaration will change the environment by making the identifier embedded in the declarator a synonym for the type yielded by the declarator.

```
de⟦declaration⟧(e):  En
    | type_specifier init_declarator ;
        — e′  where  (t, e′) = te⟦type_specifier⟧(e)
    | typedef type_specifier declarator ;
        — let    (t, e′) = te⟦type_specifier⟧(e);
                 (id, t′) = ⟦declarator⟧(t);
          in  e′[t′/id]

de⟦declaration_s⟧(e):  En
    | /* empty */
        — e
    | declaration_s  declaration
        — de⟦declaration⟧( de⟦declaration_s⟧(e) )
```

[5] Except through `sizeof` within constant expressions which might specify array bounds. Since the syntax of declarators in §3.1 does not allow constant expressions in "array" declarators, for the purposes of this paper there is no interaction between identifiers like `z` and `zp` and structure or `typedef` declarations.

4.4. *An example.* The structure tag **x** below, is used to declare one of the structure members so the type of **x** is circularly defined.

```
struct x {
    int count;
    struct x *p;
    }
```

Syntactically, the above program fragment is a type specifier, so valuation **te** will be applied to it. From the rules for **te** in §4.2, a type specifier maps an environment e to a type t and a new environment e''. Let us suppose that the starting environment e is such that $e(\mathbf{x}) = tx$.

The members of the structure do not affect the environment since they do not contain the declaration of any structure tags. The pair list yielded by the members will therefore be:

$$(\mathtt{count}, \mathit{integer})(\mathtt{p}, \mathit{point}(tx))$$

This pair list will be used to construct the type

$$t = \mathit{str}(\mathbf{x}, (\mathtt{count}, \mathit{integer})(\mathtt{p}, \mathit{point}(tx)))$$

Then, the environment $e'' = e[t/\mathbf{x}]$ will be determined and t and e'' will be the result.

If tx is an approximation to the type of **x**, then t above is a better approximation to the type of **x**. In fact, if $t_0 = \perp$, and

$$t_{i+1} = \mathit{str}(\mathbf{x}, (\mathtt{count}, \mathit{integer})(\mathtt{p}, \mathit{point}(t_i)))$$

then the least upper bound of the chain $t_0, \cdots, t_i, \cdots$ is the desired type of **x**.

There are two observations that are relevant at this point: the first hinges on the remark, "if tx is an approximation to the type of **x**"; the second concerns the fact that the types of two structure tags may be interdependent, so the types of all structure tags and `typedef` names will have to be determined simultaneously.

There is no guarantee that tx will indeed be an approximation to the type of **x**. After all **x** may be used for some other purpose outside the current C-function and we have not yet provided a mechanism for ensuring that on entering a function, the types of any structure tags declared in the current function are "reset". The valuation **dz** in §4.5 will "reset" environments.

4.5. *Initial environments in a "block".* For each identifier with which a type is associated in the current set of declarations we will enter $\perp$ as the type of the identifier. Types are associated with identifiers by `typedef` declarations, so we need a valuation **dz** for declarations. Structure tags occur within type specifiers, so we need a valuation for type specifiers. Furthermore, a type specifier may be part of a member declaration, so in order to pick up nested structure tags we need a valuation for member declarations.

Since the purpose of each of these valuations is just to enter $\perp$ for certain identifiers, we will use the same letters **dz** for all of them.

```
dz⟦declaration⟧(e): En
    | type_specifier init_declarator ;
        — dz⟦type_specifier⟧(e)
    | typedef type_specifier declarator ;
        — let   e′ = dz⟦type_specifier⟧(e);
                (id,t) = ⟦declarator⟧(⊥);     /* see note below */
          in  e′[⊥/id]

dz⟦declaration_s⟧(e): En
    | declaration
        — dz⟦declaration⟧(e)
    | declaration_s declaration
        — dz⟦declaration⟧( dz⟦declaration_s⟧(e) )
```

Note. A declarator maps a type to an identifier and a type, so we have supplied ⊥ as a "dummy" type to the declarator just in order to extract the identifier. The type t returned by the declarator will be ignored, since we enter ⊥ for *id* in the environment. □

```
dz⟦type_specifier⟧(e): En
    | basic_specifier
        — e
    | struct identifier  member_decl_p  ;
        — let   e′ = dz⟦member_decl_p⟧(e);
          in  e′[⊥/identifier]
    | struct identifier
        — e
    | identifier /*typedef name */
        — e
```

The remaining rules merely propagate the effect of embedded type specifiers.

```
dz⟦member_decl⟧(e): En
    | type_specifier declarator ;
        — dz⟦type_specifier⟧(e)

dz⟦member_decl_p⟧(e): En
    | member_decl
        — dz⟦member_decl⟧(e)
    | member_decl_p member_decl
        — dz⟦member_decl⟧( dz⟦member_decl_p⟧(e) )
```

4.6. *Conclusion.* The ingredients for determining the types of structure tags and `typedef` names have all been assembled, but the details will have to wait until function declarations are discussed in §6. On entering a function with environment *e* the **dz** valuations are used to reset the types for all identifiers that represent types in the function. The valuations **de**, **te** and **me** are then used repeatedly to determine the final environment. This final environment is the starting point for storage management.

5. Storage Management

The value of an identifier y can be changed either by an explicit assignment to y, or by an indirect assignment through a pointer to y. The presence of pointers makes it convenient to have a two-stage mapping from identifiers to their values. It is the purpose of this section to give semantics for the "storage" aspects of data declarations.

For an identifier y representing an integer, we will first find the location of y, and then look up the value in this location. For identifiers representing arrays, structures, and unions, we need enough locations to hold all the members. For example, after the declaration

```
struct pair {int hd, tl;} z;
```

locations will be reserved for `z.hd` and `z.tl`. Let these locations be l_1 and l_2. The identifier z, by itself, has a list of two locations associated with it.

Locations, §5.1, are analogous to storage cells. Every location is included in the class of lvalues, §5.1, but lvalues include for example a function mapping `hd` to l_1 and `tl` to l_2, which is clearly not a location. In general, the members of a structure or an array need not be of basic type, so rather than there being locations, there will be lvalues for the members.

As the next example suggests, lvalues for the members of an array must be contiguous, at least conceptually. The declaration

```
int ab[7];
```

will reserve 7 locations for members of the array ab, where each location will hold an integer. After ab is declared, the assignment

```
pab = &ab[0];
```

leaves pab pointing to the location for the first member of the array. The locations for the array ab have to be contiguous since the expression

```
pab + 1
```

points to ab[1], the next member of the array. In general, the members of an array can be of any type, so we will want the lvalues for the members of the array to be contiguous.

5.1. *Locations and lvalues.* Let L be a domain containing a countable number of locations $l_1, l_2, \cdots$. There will also be a special location FREE, distinct from l_i, for any $i \geqslant 0$, which will be used during storage "allocation". Informally, for each state *s*, *s*(FREE) will be the first free location relative to *s*.

$$L = \{\text{FREE}, l_1, l_2, \cdots\}_\perp$$

The function *succ* is used to obtain the "next" location: $succ(l_i) = l_{i+1}$, for all $i \geqslant 0$.

An identifier in a data declaration will be mapped by an environment *e* to an lvalue. For an identifier of basic or pointer type, this lvalue will be a location. Rather than mapping an array identifier to a list of lvalues for the members of the array, the identifier will be mapped to a function from integers to lvalues. (This function makes it easy to determine the lvalues of the array members e.g. if the lvalue for ab is *l*, then the lvalue for ab[5] will be *l*(5).) The lvalue for a structure or union identifier will be a function from member identifiers to member lvalues. There is no distinction between the lvalue for a structure and a union.

$$\mathbf{Lv} = \mathbf{L} + [\mathbf{N} \rightarrow \mathbf{Lv}] + [\mathbf{Ide} \rightarrow \mathbf{Lv}]$$

5.2. *Allocation.* Given a type t and a state s, the auxiliary function *new* returns an lvalue, and changes the state to s''. In the changed state, all locations in the returned lvalue are initialized to the special garbage value GRB, and $s''(\text{FREE})$ is the first location following the locations in the returned lvalue.

```
new(t,s) =
    $t \in \mathbf{Tb} \vee t = point(t') \rightarrow$
        $let\ l = s(\text{FREE});\ s' = s[\text{GRB}/l];\ l' = succ(l);\ in\ (l, s'[l'/\text{FREE}]),$
    $t = arr(n,t') \rightarrow$
        $let\ f_0 = \lambda i.\perp;\ in\ newa(0,n,t',f_0,s),$
    $t = str(id,pairlist) \vee t = uni(id,pairlist) \rightarrow$
        $let\ f_0 = \lambda id'.\perp;\ in\ news(pairlist,f_0,s)$

newa(i,n,t',f,s) =
    $i \geqslant n \rightarrow (f,s),$
    $i < n \rightarrow let\ (l',s') = new(t',s);\ in\ newa(i+1,n,t',f[l'/i],s')$

news(pairlist,f,s) =
    $pairlist = () \rightarrow (f,s),$
    $pairlist = (id,t')\cdot pairs \rightarrow let\ (l,s') = new(t',s);\ in\ news(pairs,f[l/id],s')$
```

5.3. *Data declarations.* The type of each identifier in a data declaration will be determined just like the types of structure members, §4.2, and `typedef` names, §4.3, were determined. Given a type, the function *new* will be used to associate an lvalue with the identifier. Allocation and initialization of the lvalue will actually be done as part of the meaning of *init_declarator*, which is given at the end of §5.4.

```
$\mathbf{ds}[\![declaration]\!](e,s)$: (En,S)
    | type_specifier init_declarator ;
        — let    $(t,e') = \mathbf{te}[\![type\_specifier]\!](e);$
          in  $[\![init\_declarator]\!](t,e',s);$

$\mathbf{ds}[\![declaration\_s]\!](e,s)$: (En,S)
    | /* empty */
        — (e,s)
    | declaration_s declaration
        — $\mathbf{ds}[\![declaration]\!](\ \mathbf{ds}[\![declaration\_s]\!](e,s)\ )$
```

5.4. *Initialization.* An initialized declaration is the only way of specifying initial values for identifiers declared externally to all functions. An "initializer" consists of an expression or a list of expressions nested in braces.

init_declarator:
declarator
declarator = initializer

initializer:
expression
{ *initializer_pc* }

initializer_pc:
initializer
initializer , *initializer_pc*

When an initializer applies to a *scalar* (a pointer or a basic type), it consists of a single expression. When the declared identifier is an *aggregate* (a structure or an array), then the initializer consists of brace enclosed, comma-separated list of initializers for the members of the aggregate, in increasing subscript or member order. If the member contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers than there are members in the aggregate, then the aggregate is padded with 0's.

It is not permitted to initialize unions.

For example, in

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

1, 3, and 5 initialize the first row of the array y[0], namely y[0][0], y[0][1], and y[0][2]. Likewise the next two lines initialize y[1] and y[2]. The initializer ends early and therefore y[3] is initialized with 0's.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

Precisely the effect of the example above is achieved by

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for y begins with a left brace, but that for y[0] does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for y[1] and y[2]. Also, viewing y as a two dimensional array,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column and leaves the rest of the array 0.

Since braces may be elided, the type of an identifier must be known before expressions in an initializer can be associated with locations to be initialized. In associating expressions with locations it is more convenient to follow the structure of the type of the identifier being initialized than the syntactic structure of the initializer.

Given an environment and a state, we will extract a list (or tree) of values from an initializer. Informally, if the expressions in an initializer were evaluated, and the braces indicated the nesting structure, then the resulting structure would be the list of values extracted. This list will be an element of the domain Vt given by

$$\mathbf{Vt} = \mathbf{V} + \mathbf{Vt}^{+}$$

The list is constructed as follows.

```
[[initializer]](e,s):  (Vt,S)
    | exp
        — [[exp]](e,s)
    | { initializer_pc }
        — [[initializer_pc]](e,s)

[[initializer_pc]](e,s):  (Vt,S)
    | initializer
        — [[initializer]](e,s)
    | initializer , initializer_pc
        — let   (list1,s′) = [[initializer]](e,s);
                (list2,s″) = [[initializer_pc]](e,s′);
          in  (list1 · list2,s″)
```

If an identifier of type *t* has lvalue *l*, then the auxiliary function *enter* will initialize the lvalue *l* using values from a list *initlist*. Like *new*, §5.2, *enter* examines the structure of type *t* and extracts elements from *initlist* as needed and enters them into locations. Entering a value into a location changes the state, so *enter* yields a new state and the remainder of the *initlist*.

```
enter(l,t,s,initlist) =
    $t \in \mathbf{Tb} \vee t = point(t') \rightarrow enterb(l,t,s,initlist)$,
    $t = arr(n,t') \rightarrow entera(0,n,l,t',s,initlist)$,
    $t = str(id,pairlist) \rightarrow enters(pairlist,l,s,initlist)$

enterb(l,t,s,initlist) =
    $initlist = () \rightarrow ((),s[0/l])$,
    $initlist = v \in \mathbf{V} \rightarrow ((),s[v/l])$,[6]
    $initlist = (v,list_2,\ldots,list_k) \wedge v \in \mathbf{V} \rightarrow ((list_2,\ldots,list_k),s[v/l])$[6]
```

[6] In general, *v* need not be of type *t*, so an auxiliary function *cast* may be needed to determine *v′* from *v*, where *v′* is of type *t*, and *v′* is entered in location *l*.

```
$entera(i,n,l,t,s,initlist) =$
    $i \geqslant n \rightarrow (initlist,s),$
    $initlist = (v,list_2, \ldots, list_k) \wedge v \in \mathbf{V} \rightarrow$
        let  $(initlist',s') = enter(l(i),t,s,initlist);$
        in   $entera(i+1,n,l,s',initlist'),$
    $initlist = (list_1,list_2, \ldots, list_k) \rightarrow$
        let  $(rest,s') = enter(l(i),t,s,list_1);$
        in   $entera(i+1,n,l,t,s',(list_2, \ldots, list_k))$

$enters(pairlist,l,s,initlist) =$
    $pairlist = () \rightarrow (initlist,s),$
    $initlist = (v,list_2, \ldots, list_k) \wedge v \in \mathbf{V} \rightarrow$
        let  $pairlist = (id,t) \cdot more;$
             $(initlist',s') = enter(l(id),t,s,initlist);$
        in   $enters(more,l,s',initlist'),$
    $initlist = (list_1,list_2, \ldots, list_k) \rightarrow$
        let  $pairlist = (id,t) \cdot more;$
             $(rest,s') = enter(l(id),t,s,list_1);$
        in   $enters(more,l,s',(list_2, \ldots, list_k)$
```

An *init_declarator* consists of a declarator with a possible intializer. All locations in the lvalue returned by *new* are intialized to the special garbage value GRB. If an intializer is present, then this garbage value is overwritten.

```
$[\![init\_declarator]\!](t,e,s)$: (En,S)
    | declarator
        — let  $(id,t') = [\![declarator]\!](t);$
               $(l,s') = new(t',s);$
               $e' = e[l/id];$
          in  $(e',s')$
    | declarator initializer
        — let  $(id,t') = [\![declarator]\!](t);$
               $(l,s') = new(t',s);$
               $e' = e[l/id];$
               $(initlist,s') = [\![initializer]\!](e,s);$
               $(initlist',s'') = enter(l,t',s',initlist);$
          in  $(e',s'')$
```

6. Function Declarations

A function declaration, like the following from §1.1, conforms to the syntax given below.

```
char select(x,c,d)
int x; char c; char d;
{
    char e;
    if (x>n) e = c; else e = d;
    return(e);
}
```

6.1. *Syntax.*

fn_dec:
  type_specifier declarator parameters fn_body

parameters:
  (*identifier_sc*) *parameter_decl_s*

parameter_decl_s:
  /* empty */
  type_specifier declarator ; *parameter_decl_s*

6.2. *Parameters.* Since parameters are called by "value" in C, on entry to a function, the actual parameters, which will be values, will be entered into fresh lvalues allocated for the purpose. Observe that parameter declarations cannot affect the environment. (In §4 embedded structure tags within type_specifiers caused the environment to be changed. Recall from §1.2 and §2.5 that the structure tag becomes part of the type that the tag refers to. Any tag within parameter declarations will be treated as if it was distinct from the same tag external to the function. Since the types of the formal parameters must match the types of the actual parameters, no new tags can be declared within the parameter declarations.)

From the parameter declarations we will extract a list of formal parameters and their associated types.

```
⟦parameters⟧(e): (Ide,Ty)*
  | ( identifier_sc ) parameter_decl_s
      — ⟦parameter_decl_s⟧(e)

⟦parameter_decl_s⟧(e): (Ide,Ty)*
  | /* empty */
      — ()
  | type_specifier declarator ; parameter_decl_s
      — let   (t,e') = te⟦type_specifier⟧(e);
              (pid,pt) = ⟦declarator⟧(t);
              pairs = ⟦parameter_decl_s⟧(e);
        in (pid,pt) · pairs
```

6.3. *Function entry.* Having determined the formal parameters and their types, the function *new* will be used to find fresh lvalues for the formal parameters.[7]

[7] The semantic rules for *fn_dec* assume that the formal parameters are of scalar type i.e. either basic or pointer. If this restriction on types is lifted, then the lvalue for a formal parameter need not be a location.

```
fe⟦fn_dec⟧(e): En
  | type_specifier declarator parameters fn_body
    — let   $(t,e') = \mathbf{te}[\![type\_specifier]\!](e)$;
            $(fid,t') = [\![declarator]\!](t)$;
            $(pid_1,pt_1)\cdot \cdots \cdot(pid_k,pt_k) = [\![parameters]\!](e')$;
            $f = \lambda v_1, \ldots, v_k.\ \lambda s.$
                let  $(l_1,s_1) = new(pt_1,s)$;
                     ...
                     $(l_k,s_k) = new(pt_k,s_{k-1})$;
                     $e'' = e'[l_1/pid_1] \cdots [l_k/pid_k]$;
                     $s' = s_k[v_1/l_1] \cdots [v_k/l_k]$;
                     $(v,s'') = [\![fn\_body]\!](e'',s')$;
                in  $(v,s'')$
        in   $e[f/fid]$
```

6.4. *Function bodies.* The rules for *fn_body* that follow summarize a fair amount of work. (Since several environments and states are involved, the line number in which an environment or state is defined is used as the subscript for the defined environment or state.)

The function body is reached with environment e and state s which already have lvalues for the formal parameters initialized with the actual parameters. The first step is to use the valuation **dz**, §4.5, to enter $\perp$ as the type for all structure tags and `typedef` names in the declarations at the head of the function body. As discussed in §4.4, the valuation de determines a new environment containing better approximations of the types for the structure tags and the `typedef` names. Thus we can get the sequence of environments

$$en_0 = e_1$$

$$en_{i+1} = \mathbf{de}[\![declaration\_s]\!](en_i) \qquad i \geqslant 0$$

The environment e_2 that contains the types of all structure tags and `typedef` names is actually the least upper bound of the chain of environments $en_0, en_1, \cdots$.

$$e_2 = \sqcup\{en_i \mid i \geqslant 0\}$$

The above least upper bound is similar enough to the least upper bounds taken while determining least fixed points that the reader may be tempted to equate e_2 with the least fixed point of $\mathbf{de}[\![declaration\_s]\!]$. Note however that en_0 is not $\perp$ and contains valuable information about external identifiers.

We therefore introduce a new operator *clo* (from fix closure) such that given a function τ: D → D and $x \in$ D,[8]

and a function like *enter*, §5.4, will be needed to enter the actual parameter into the lvalue. The line

$$s' = s_k[v_1/l_1] \cdots [v_k/l_k];$$

will then have to be changed.

Also, the value v returned by *fn_body* need not be of the same type as t', which is the type returned by the function, so an auxiliary function *cast* may be needed to determine v' from v, where v' is of type t'.

[8] P. Cousot pointed out that $clo(x)(\tau)$ is exactly $luis(\tau)(x)$ in the terminology of [cou77]. Some theoretical properties of this operator have been studied in [cou77].

$$clo(x)(\tau) = \bigsqcup\{\ \tau^i(x) \mid i \geqslant 1\}$$

Using *clo*

$$e_2 = clo(e_1)(\mathbf{de}[\![declaration\_s]\!])$$

Having determined e_2 which contains types for structure tags and `typedef` names, we can now allocate and initialize storage for the data declarations at the head of the function body by using valuation **ds**, §5.3, to end up with environment e_3 and state s_3.

The declarations have now been attended to, but we still have labels within the statements in the function body to worry about. A valuation **sz** will enter the continuation $\perp$ for each label in the function body, yielding environment e_4. This time valuation **se** determines a new environment containing better approximations of the continuations for the labels in the function body. Another use of the *clo* operator yields environment e_5 which can finally be used for the statements in the function body.

As discussed in [set79b], starting with an initial continuation c_0 which yields the garbage value GRB and does not change the state, the value v and the final state s_7 returned by the function are determined using $\mathbf{sc}[\![stm\_s]\!]$.

```
$[\![fn\_body]\!](e,s)$:  (V,S)
    | { $declaration\_s$ $stm\_s$ }
        — let   $e_1 = \mathbf{dz}[\![declaration\_s]\!](e)$;
                $e_2 = \mathrm{clo}(e_1)(\mathbf{de}[\![declaration\_s]\!])$;
                $(e_3,s_3) = \mathbf{ds}[\![declaration\_s]\!](e_2,s)$;
                $e_4 = \mathbf{sz}[\![stm\_s]\!](e_3)$;
                $e_5 = \mathrm{clo}(e_4)(\mathbf{se}[\![stm\_s]\!])$;
                $c_0 = \lambda\ s'.\ (\mathrm{GRB},s')$;
                $(v,s_7) = \mathbf{sc}[\![stm\_s]\!](e_5,c_0,s_3)$;
          in  $(v,s_7)$
```

6.5. *Programs.* The semantics of a C program are very similar to the semantics of a C-function body: there are a sequence of data declarations that have to be "processed" and then instead of entering continuations for mutually dependent labels into the environment, the meanings of mutually recursive function declarations have to be entered into the environment. The valuation **fz** specified below enters $\perp$ into the environment for each C-function identifier.

```
$[\![program]\!](e,s)$: S
  | declaration_s fn_dec_p
      — let  $e_1 = \mathrm{dz}[\![declaration\_s]\!](e)$;
             $e_2 = \mathrm{clo}(e_1)(\mathrm{de}[\![declaration\_s]\!])$;
             $(e_3, s_3) = \mathrm{ds}[\![declaration\_s]\!](e_2, s)$;
             $e_4 = \mathrm{fz}[\![fn\_dec\_p]\!](e_3)$;
             $e_5 = \mathrm{clo}(e_4)(fe[\![fn\_dec\_p]\!])$;
             $f = e_5(\mathtt{main})$;
             $(v, s_7) = f(s_3)$;
        in  $s_7$

$\mathrm{fz}[\![fn\_dec]\!](e)$: En
  | type_specifier declarator parameters fn_body
      — let  $(fid, t) = [\![declarator]\!](\perp)$;
        in  $e[\perp/fid]$

$\mathrm{fz}[\![fn\_dec\_p]\!](e)$: En
  | fn_dec
      — $\mathrm{fz}[\![fn\_dec]\!](e)$
  | fn_dec fn_dec_p
      — $\mathrm{fz}[\![fn\_dec\_p]\!](\ \mathrm{fz}[\![fn\_dec]\!](e)\ )$
```

Acknowledgments.

S. C. Johnson and D. M. Ritchie patiently fielded a number of questions about C. R. D. Tennent patiently fielded a number of questions about his denotational semantics of Pascal [ten77]. There are a number of similarities between C and Pascal at the level of the semantic rules, and it was useful to have [ten77] as a reference.

I want to thank B. W. Kernighan and D. M. Ritchie for their permission to use the material from [ker78]. Comments by A. V. Aho, S. R. Bourne, and M. D. McIlroy are appreciated.

[illegible]

MH-1271-RS-unix   Ravi Sethi

Att.
Appendix A (pgs. 23,24)
References

Appendix A. Abstract Syntax of Declarations

Convention. At several points, lists of items, sometimes separated by commas have to be generated. By convention, the suffixes *_s*, *_p*, *_sc*, and *_pc* mean zero or more, one or more, zero or more separated by commas, and one or more separated by commas, respectively. For example, if *nonterm* is some nonterminal, then the productions for *nonterm_s*, *nonterm_p*, *nonterm_sc*, and *nonterm_pc* are as given below.

```
nonterm_s:
    /* empty */
    nonterm_p

nonterm_p:
    nonterm
    nonterm nonterm_p

nonterm_sc:
    /* empty */
    nonterm_pc

nonterm_pc:
    nonterm
    nonterm , nonterm_pc
```

Productions for nonterminals ending with *_s*, *_p*, *_sc*, and *_pc* will not be given.

```
program:
    declaration_s fn_dec_p

fn_dec:
    type_specifier declarator parameters fn_body

parameters:
    ( identifier_sc ) parameter_decl_s

parameter_decl:
    type_specifier declarator ;

declaration:
    type_specifier init_declarator ;

type_specifier:
    basic_specifier
    struct identifier { member_decl_p }
    struct identifier
    union identifier { member_decl_p }
    union identifier
    identifier

member_decl:
    type_specifier declarator ;

init_declarator:
    declarator
```

```
        declarator = initializer

    declarator:
        identifier
        ( declarator )
        declarator [ constant ]
        declarator ( )
        * declarator

    abstract_declarator:
        /* empty */
        ( abstract_declarator )
        abstract_declarator [ constant ]
        abstract_declarator ( )
        * abstract_declarator

    initializer:
        expression
        { initializer_pc }

    type_name:
        type_specifier abstract_declarator
```

References

The following list combines references mentioned in the text with references from [set79b].

cou77	P. Cousot and R. Cousot, "Constructive versions of Tarski's fixed point theorems," Rapport de Recherche 85, L.A. 7, Universite Scientifique et Medicale de Grenoble, Grenoble, France (September 1977).
dij72	E. W. Dijkstra, "Notes on structured programming," pp.1-82 in O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (eds.) *Structured Programming,* Academic Press, London (1972).
joh78	S. C. Johnson, "A portable compiler: theory and practice," *Fifth ACM Symposium on Principles of Programming Languages,* pp. 97-104 (January 1978).
ker78	B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J. (1978).
mss76	P. D. Mosses, "Compiler generation using denotational semantics," pp.436-441 in *Mathematical Foundations of Computer Science 1976,* Lecture Notes in Computer Science 45, Springer-Verlag, Berlin (1976).
rit78a	D. M. Ritchie, "A tour through the UNIX C compiler," unpublished manuscript, Bell Laboratories, Murray Hill, NJ (1978).
rit78b	D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX time-sharing system: the C programming language," *Bell Sys. Tech. J.* 57(6) pp. 1991-2019 (1978).
scs71	D. Scott and C. Strachey, "Towards a mathematical semantics for computer languages," pp.19-46 in *Proceedings of the Symposium on Computers and Automata,* Polytechnic Press, Brooklyn, N.Y. (April 1971).
set79a	R. Sethi, "Semantics of the C programming language, part 0: prelude," unpublished manuscript, Bell Laboratories, Murray Hill, N.J. (1979).
set79b	R. Sethi, "Semantics of the C programming language, part 1: statements," unpublished manuscript, Bell Laboratories, Murray Hill, N.J. (1979).
sto77	J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, Cambridge, MA (1977).
ten76	R. D. Tennent, "The denotational semantics of programming languages," *Comm. ACM* **19** (8) pp.437-453 (August 1976).
ten77	R. D. Tennent, "A denotational definition of the programming language PASCAL," Technical Report 77-47, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada (July 1977).