An Intermediate Form for C° Programs

This document describes the intermediate representation that your compilers will use for C° programs. The scheme we will use is based on the idea of syntax trees. Therefore, much of this document will be concerned with the structure of trees for C° programs. Syntax trees typically include some pointers into the compiler's symbol table. Accordingly, this document also includes a partial specification of a symbol table organization for a C° compiler. In fact, since the syntax trees depend upon the symbol table, we will begin with a summary of the organization of the symbol table.

1 Symbol Table Organization Overview

The symbol table maintained by your compiler will consist of two main components. The first is a collection of dynamically allocated structures containing one element for each distinct identifier used in the program. We will refer to this collection of structures as the *identifier table* and to its elements as *identifier descriptors*. Each of these elements will contain a pointer to the character string representation of the identifier with which it is associated and several other link fields. This table will be created by the scanner.

The second component is a collection of dynamically allocated structures including one element for each distinct declaration or definition in the program. We will refer to this collection of structures as the *declaration table* and its elements as *declaration descriptors*. These entries will be created and initialized by the semantic processing phase of your compiler. Each of these elements will include a pointer to the identifier descriptor for the identifier with which it is associated; attribute fields describing important characteristics of this declaration of the identifier (such as whether it is a function, a type or a variable); and several additional link fields.

In the syntax trees produced as output of the syntactic analysis phase, identifiers will be represented by pointers to their identifier descriptors. During the semantic processing phase, references to identifiers within the syntax tree will be modified so that references to identifier descriptors are either replaced by or augmented with references to declaration descriptors.

The construction of the symbol table will depend on two hash tables. The first of these hash tables is used by the scanner to locate the appropriate identifier descriptors as it encounters identifiers in the source program. You will not be concerned with the details of this hash table (I will provide you with the object code of a scanner including the hash table).

The second hash table is used to locate the declaration descriptors for structure component names. Given a component name and a pointer to the declaration descriptor for a structure type, this table will enable one to locate the declaration descriptor for the named component of the type if the named component is indeed a component of the specified structure type. This search structure will be created and maintained by the semantic processing phase of your compiler.

The format of the structures used to hold identifier and declaration descriptors is discussed below, after the specification of the syntax trees for C°.

2 Syntax Tree Organization

As discussed in class, there is a significant difference between the internal nodes of a syntax tree and its leaves. Within an internal node, one finds a phrase type and pointers to sub-trees. The leaves, on

the other hand, hold information about identifiers and constants. In fact, in class I have suggested that rather than actually having separate nodes for the leaves, one could use symbol table entries for leaf nodes.

We will not actually do this in the compilers you build. The reason is a simple, practical one. To generate good error messages, one needs to keep information about where in the source program the text that corresponds to each sub-tree of the syntax tree can be found. We will do this by storing in each node the line number on which the first token that belonged to the phrase the node represents was found. This can not be done for identifiers if all occurrences of an identifier are represented by a single symbol table entry. So, we instead represent identifiers by nodes that contain the line number on which they were found and a pointer to the appropriate symbol table entry. Similar nodes will be used for constants.

As part of semantic processing, you will rewrite the trees that the parser produces for variable references. Basically, while the parser creates trees based on the syntactic structure of the source code, the code generator would prefer trees corresponding closely to the capabilities of the underlying hardware. Variable references, particularly subscripted variables and component selections, can be reconstructed by the semantic processing routines so that they explicitly describe much of the addressing arithmetic required by the variable references they represent.

Two special node types are used to support this translation of variable reference subtrees. The first is an internal node type used to represent the root of a variable reference subtree. These nodes will each hold a single pointer to the subtree that describes the variable reference. The other is a node type used to represent references to pointers to function activation records. Such nodes do not appear in the trees produced by the parser but are needed to translate variable reference subtrees into a form that explicitly describes the required address arithmetic. These nodes will always appear as leaves in the tree.

2.1 Representing Syntax Tree Nodes

This leads to a syntax tree with five distinct node types. As a result, to specify the general "type" of a syntax tree node, we use the C union type `node` described below.[1]

```
      /* Union type that combines the 5 structure types used to describe */
      /* tree nodes */
typedef union nodeunion {
  struct unknode unk;
  struct internalnode internal;
  struct identnode ident;
  struct constnode constant;
  struct displaynode display;
  struct refvarnode var;
} node;
```

Each of the five node types present in the tree include two common fields: the field specifying the node's phrase type[2] (`type`) and the field specifying the line of the source code on which the first token of the phrase represented by the node's subtree occurred (`line`). The type `unknode`, whose definition is shown below, allows one to reference these fields in situations where the actual type of the node is not yet known. For example, if '`root`' is a pointer to a `node` of unknown type one can use the expression:

```
    root->unk.type
```

[1] The C type declarations shown in this handout will all be made available in '.h' files on the Dells.

[2] Node phrase types are specificed using an enumeration type named 'nodetype'. The elements of this type and the details of nodes of each phrase type are described below.

```
          /* The type 'unknode' provides a template that can be used to */
          /* access the common components found in all node types when  */
          /* the actual type of the node is unknown.                    */
struct unknode {
  nodetype type;
  int line;
};
```

to determine its phrase type. One could also use the expression 'root->internal.type' or 'root->ident.type', but these expressions mis-leadingly suggest that the type of the node is already known. The type `unknode` is provided to support clear coding.

The structure type `internalnode` describes the nodes used to represent almost all of the internal nodes of the tree. In addition to the common `type` and `line` components found in all nodes, a node

```
          /* Tree nodes of type 'internal' are used for all nodes that */
          /* are internal to the tree produced by the parser except    */
          /* for the roots of variable reference subtrees.             */
struct internalnode {
  nodetype type;
  int line;
  union nodeunion *child[MAXKIDS]; /* pointers to the node's sub-trees */
};
```

of this type includes a component `child` which is an array of pointers to its children. The number of children of a given node can be determined from its node type. The syntactic analysis routines I will provide conserve memory by only allocating space for the child pointers actually used by a given internal node. Thus, if a node should only have 2 children, its third child pointer should not be used for any purpose.

The structure types `identnode` and `constnode` are used to represent the leaves of the syntax trees produced by the parser. Declarations of the structure types are shown below:

```
         /* Nodes of type 'ident' are used for leaf nodes corresponding */
         /* to identifiers in the source code.  The value in the        */
         /* 'type' component of such a node will always be 'Nident'.    */
struct identnode {
  nodetype type;
  int line;
  identdesc *ident;   /* Pointer to associated identifier descriptor */
  decldesc *decl;     /* Pointer to associated declaration descriptor */
};

         /* Nodes of type 'constant' are used for leaf nodes corresponding */
         /* to constants in the source code.  The value in the 'type'      */
         /* component of such a node will always be 'Nconst'.              */
struct constnode {
  nodetype type;
  int line;
  int value;              /* Integer value of the constant  */
  int ischar;             /* True if this was a character constant */
};
```

Identifiers are represented by nodes of type `identnode`. The `type` component of such nodes will always be `Nident`. The `ident` and `decl` components of an `identnode` are pointers to the appropriate identifier descriptor and declaration descriptor for the identifier being referenced. The `decl` components of `identnode` nodes are set to NULL (the value 0) by the syntactic analyzer. During semantic analysis, the correct values should be stored in these fields.

There is one special group of `identnode`s produced by the syntactic analyzer. These are `identnode`s for the keyword **integer**. Technically, **integer** is a keyword rather than an identifier in C°. Treating it as an identifier that has been declared as a type, however, will simplify various parts of the compiler. Accordingly, occurrences of **integer** will be represented by special `identnode`s in the syntax tree.

Each `constnode` contains two fields beyond the common `type` and `line` fields. One is named `value`. It holds the integer value of the constant. The second is a field named `ischar` which is used as a boolean flag indicating whether the constant found in the source code was a character or an integer. The `type` component of all such nodes will be `Nconst`.

There are two additional node types related to subtrees representing references to variables. Their declarations are shown in figures 1 and 2. Tree nodes of type `displaynode` are used to refer to the address of a function's stack frame. Other than the standard `type` and `line` fields, the only member

```
        /* Nodes of type 'display' are used for leaf nodes corresponding  */
        /* to points where the address of the activation record of a      */
        /* function is needed.  Such nodes are not included in the tree  */
        /* produced by the parser.  They are inserted during semantic     */
        /* processing. */
struct displaynode {
  nodetype type;
  int line;
  int level;          /* The nesting level of the function whose activation record
                                  address should be used. */
 };
```

Figure 1: The type displaynode

of a display reference node is a `level` field use to store the nesting level of the function whose frame address is to be used.

Finally, nodes of type `refvarnode` are used to designate places where a value should be loaded from a calculated memory address. The `baseaddr` field of a `refvarnode` points to a subtree that describes the computation of the base address. The value of the `displacement` field gives a constant value to be added to the base address before accessing memory. This field is initialized to 0 in trees created by the parser. The `vardesc` field is intended to point to a declaration descriptor for the variable referenced. This field is set to NULL by the parser. As the semantic processor translates variable reference subtrees into a form that more explicitly describes addressing arithmetic, it should set each `refvarnode`'s `vardesc` field to point to the appropriate declaration descriptor.

2.2 Node Phrase Types

As mentioned above, the phrase types `Nident` and `Nconst` are used to label nodes representing the leaves of the syntax tree. The phrase types `Nrefvar` and `Ndisplay` are used to identify the two special node types used to encode variable reference subtrees. All of the other node phrase names defined in the enumeration type `nodetype` are used to label internal nodes. All of these other node phrase names are listed and described below.

```
        /* Refvar nodes are included by the parser as the roots of all     */
        /* variable reference subtrees.  When created by the parser, the   */
        /* "baseaddr" field will either point to an Nselect, Nsubs or      */
        /* Nident node.  During semantic analysis the "baseaddr" subtree   */
        /* will be converted into a subtree describing the calculation of  */
        /* the memory address for the variable.  A "displacement" field is */
        /* included to hold a constant offset from the base address to the */
        /* variable.  Finally, to preserve information about the symbolic  */
        /* variable being used, the semantic analyzer should set the       */
        /* "vardesc" field to point to the declaration descriptor of the   */
        /* variable being used.     */
struct refvarnode {
  nodetype type;
  int line;
  union nodeunion
      * baseaddr;       /* Subtree describing base address calculation     */
  int displacement;     /* Displacement to variable relative to base addr  */
  decldesc *vardesc;    /* Declaration descriptor for referenced variable  */
};
```

Figure 2: The refvarnode type

There are several important subgroups of node phrase types. One important group is the group of "list" phrases including `Nstmtlist`, `Ntypelist`, `Nvarlist` and all of the other phrase types whose names end with "list". These nodes are used to represent lists of items in the program. In all cases, such nodes take 2 children. The left child (`child[0]`) of a list node points to the first element of the list (i.e. a statement, type definition, variable definition or whatever element type is appropriate). The right child (`child[1]`) points to the remainder of the list. Its value is either NULL (= 0) or a pointer to another list node of the same type.

Other important groups of phrase types include the statement phrase types (`Nasgn`, `Ncall`, `Nretn`, `Nif` and `Nwhile`) the variable phrase types (`Nident`, `Nselect`, and `Nsubs`) and the expression phrase types (which includes the `Nrefvar` phrase type in addition to all the "unaries" and "binaries" mentioned in the table below).

All of the phrase names used in internal tree nodes are described in the list below. This list is organized so that node labels for phrase types occur in roughly the same order as the corresponding rules of the C° grammar in the *Revised Report on the C° Programming Language* handout.

Node Type	Num. of Children	Description
Nprogram	2	Represents an entire program. Child[0] is a (possibly NULL) list of Ntypelist nodes. Child[1] is an Nbody sub-tree.
Nbody	3	Represents the body of a program or function. Child[0] is a (possibly NULL) list of Nvarlist nodes. Child[1] is a (possibly NULL) list of Nfunclist nodes. Child[2] is a list of Nstmtlist nodes.
Ntypelist	2	List header used to build lists of Ntypedefn nodes.
Ntypedefn	2	Used to represent a single type definition. Child[0] will be an Nident node for the name of the type. Child[1] will be an Narray or Nstruct node describing the type itself.
Narray	2	Represents an array type specification. Child[0] is an Nconst specifying the array's size. Child[1] is an Nident node for the type name specified for the elements of the array.
Nstruct	1	Represents a struct specification. Child[0] is a Nfieldlist containing the field specifications for all the components of the structure.
Nfieldlist	2	Used to represent lists of structure field specifications. Child[0] will be an Ndecl.
Ndecl	2	Used to represent variable declarations and structure field specifications. Both children should be of type Nident. Child[0] is the identifier being declared. Child[1] is the type name. Remember that a special symbol table entry is created during initialization to allow uniform treatment of the type `integer`.
Nvarlist	2	Used to represent lists of variable declarations. Child[0] will be an Ndecl.
Nfunclist	2	Used to represent lists of function definitions. Child[0] will be an Nprocdefn or an Nfuncdefn.
Nprocdefn	3	Used to represent the definition of a **void** function. Child[0] is an Nident node for the function's name. Child[1] is a (possibly NULL) list of Nformallist nodes. Child[2] is an Nbody node for the function's body.
Nfuncdefn	3	Used to represent the definition of a function that returns a value. The use of the children is identical to that of an Nprocdefn.
Nformallist	2	Used to represent lists of formal parameter specifications. Child[0] will be an Nvarparm or an Nvalparm.
Nvarparm	2	Used to represent the specification of a call-by-reference parameter. Child[0] is an Nident node for the formal parameter name. Child[1] is an Nident node for the parameter type.
Nvalparm	2	Used to represent the specification of a call-by-value parameter. The children are similar to those of an Nvarparm node. Child[1] will always be an Nident node for the pseudo-identifier **integer**.

Node Type	Num. of Children	Description
Nstmtlist	2	Used to represent statement lists. Child[0] will be one of the following five "statement" phase types or another Nstmtlist node.
Nasgn	2	Represents an assignment statement. Child[0] will be a node of type Nrefvar pointing to a subtree that describes the target of the assignment. Child[1] will be a node whose type is classified as an "expression".
Ncall	2	Represents a call statement or a function call expression. Child[0] will be an Nident node for the function's name. Child[1] points to a (possibly NULL) list of Nactuallist nodes.
Nretn	1	Represents a return statement. If an expression was included in the statement, child[0] points to a sub-tree representing the expression. Otherwise, child[0] is NULL.
Nif	3	Represents an if statement. Child[0] points to a sub-tree representing the "boolean" expression. Child[1] points to a list of Nstmtlist nodes that represents the then part. If an else part was included, child[2] points to the list of Nstmtlist nodes representing the else part. Otherwise, child[2] is NULL. Note that the last two children will be list nodes even if only a single statement is included for either the then or else part.
Nwhile	2	Represents a while statement. Child[0] points to a tree representing the loop termination condition. Child[1] points to a statement list representing the loop body. For loops are rewritten to appear as Nwhile subtrees by the parser.
Nactuallist	2	Used to represent lists of actual parameters. Child[0] will be a node of one of the expression phrase types.
Nrefvar	1	Nrefvar nodes are stored using the `refvarnode` type rather than the `internalnode` type. They do, however, appear as internal nodes in the tree. They appear as the roots of variable subtrees pointed to by Nasgn nodes and nodes that are expected to point to nodes representing expressions. In the trees produced by the parser, the `baseaddr` of such a node will point to either an Nident, Nselect or Nsubs node. After semantic processing, an Nrefvar node will point to a node of some expression phrase type.
Nselect	2	Represents a variable (or expression) formed by selecting a component from some structure variable. Child[0] describes the struct sub-variable. Child[1] is an Nident node for the name of the component being selected.
Nsubs	2	Represents a variable (or expression) formed by subscripting an array variable. Child[0] represents the array sub-variable. Child[1] points to an expression sub-tree for the subscript expression.

Node Type	Num. of Children	Description
unaries	1	The node labels Nnot and Nneg are used to represent expressions formed using the logical not operator (`!`) and the arithmetic negation operator (unary `-`). Child[0] points to a sub-tree representing the expression to whose value the operator should be applied.
binaries	2	The node labels Nor, Nand, Nlt, Ngt, Neq, Nle, Nge, Nne, Nplus, Nminus, Ntimes, Ndiv and Nmod are used to represent expressions formed using the logical, relational and arithmetic binary operators. The sub-expressions to whose values the operator should be applied are pointed to by child[0] and child[1].
Nerror	0	Inserted in tree at points where an error was detected in the syntax of a phrase. Actually, the only place that such nodes ever appear is as elements of "lists". So, the only place you need to check for them is when processing statement lists, parameter lists, etc.

3 Symbol Table Details

Now, to complete the discussion of our scheme for representing C° programs, we must discuss more details of the types used in the symbol table. As explained in the overview presented above, the symbol table is composed of identifier descriptors and declaration descriptors.

3.1 Identifier Descriptors

Identifier descriptors are actually quite simple. There is one such descriptor for each distinct identifier used in the program.[3] A C language structure specification for the type `identdesc` used to store identifier descriptors is shown in figure 3. The `name` field is just a pointer to the characters that

```
typedef struct iddesc {
  char *name;               /* The character string for the identifier */
  struct iddesc *hashlink;  /* Link for hash chains used by scanner    */
  union dcldesc *declstack; /* Head pointer for stack of declarations  */
                            /*    in open scopes.                      */
} identdesc;
```

Figure 3: Declaration for Type 'identdesc'

form the identifier. The `hashlink` field is used to maintain lists of identifier with the same hash value when building the hash table used by the scanner. It will not be of concern to you when doing semantic processing. The `declstack` component is to be used as a pointer to the head of the linked list representing the stack of declarations of the identifier found in scopes that are still open. The scanner initializes this field to NULL.

[3]Including the pseudo-identifier **integer** as discussed above

3.2 Declaration Descriptors

Declaration descriptors are more complex than identifier descriptors. Depending on the type of declaration involved (a type definition or a function definition or a variable declaration, etc.) different structures must be used. Accordingly, as with tree nodes, the type used to describe declaration descriptors is a union type. The C declaration for this union type is shown in figure 4.

```
    /* This union describes the type of all declaration descriptors */
typedef union dcldesc {
  struct unkdesc unk;

  struct funcdesc func;

  struct vardesc unkvar;
  struct vardesc var;
  struct formaldesc formal;
  struct fielddesc field;

  struct unktypedesc unktype;
  struct arraydesc array;
  struct structdesc structure;
} decldesc;
```

Figure 4: Definition of the type 'decldesc'

While many distinct structure types are used as declaration descriptors they share several common fields. The declarations of these common fields is grouped in a `#define` named `COMMONFIELDS`. This `#define` is used to include the fields in each of the distinct structure type definitions. As in the syntax tree definitions, all declaration descriptors contain a common `type` field used to determine the actual format of a member of the union type. The value of this `type` field will be an element of the enumerated type `decltype`. Also, a structure type `unkdesc` is provided to allow one to reference the common fields of a declaration descriptor before the actual type of the descriptor involved can be determined. The declarations of `COMMONFIELDS`, `decltype` and `unkdesc` are shown in figure 5.

The first of the common fields is the `type` field which holds an element of the enumeration type `decltype`. The field `ident` is used by all declaration descriptors to hold a pointer back to the identifier descriptor for the identifier associated with the declaration. The `line` component is used to hold the line number on which the declaration occurred.

The next two common fields are not used in all declaration descriptors. In particular, they are not used in descriptors for structure component names. The first of these two fields is `levellink`. This is used to link all of the declarations found in an open scope together in a linked list. The second is `level` which holds the nesting level of the scope in which the declaration occurred.

The last component in `COMMONFIELDS` is `decllink`. During declaration processing, this field is used as the "next" pointer for the linked list that is used to implement the stack of declaration descriptors associated with a given identifier descriptor. The head pointers for these stacks are stored in the `declstack` components of identifier descriptors.

There are many different kinds of declaration descriptors, but they all fall within three main groups. While all descriptors share the `COMMONFIELDS`, descriptors within each group share additional features.

```
    /* Enumeration type used to label the various type of declaration  */
    /* descriptors that can occur in the symbol table.                 */
typedef enum {
  funcdecl,        /* function declarations       */
  vardecl,         /* global and local variables              */
  formaldecl,      /* Formal parameter names                  */
  fielddecl,       /* Component names of struct type          */
  arraydecl,       /* Type names for array types              */
  structdecl,      /* Type names for struct types             */
  integertype      /* Label for pseudo-declaration of integer */
  } decltype;


    /* All declaration descriptors contain the following components */
    /* (although structure component descriptors don't use them all.) */
#define COMMONFIELDS      decltype type;            /* Type of this declaration descriptor
*/     identdesc *ident;         /* pointer to associated ident. descriptor    */     int
line;                  /* Line number at which declaration occurred. */     union dcldesc
*levellink; /* list of declarations made at nesting level */     int level;                /*
nesting level of this declaration          */     union dcldesc *decllink;  /* stack of active
declarations of this ident */


    /* Generic structure used to access common fields of decl. descriptors. */
struct unkdesc {
   COMMONFIELDS
};
```

Figure 5: Definitions of shared features of declaration descriptors

3.2.1 Type Name Declaration Descriptors

The first group includes all descriptors for names associated with types. This includes struct type descriptors, array type descriptors and a special descriptor for the built-in integer type. The definitions for this group of type descriptors is shown in figure 6.

All type name declaration descriptors share a field named `typesize` which should be set equal to the number of units of memory needed to hold an element of the type. To provide a way to write clear code that accesses this field without determining the particular sort of type involved, a structure named `unktypedesc` is declared and a choice of `unktype` is included in the `decldesc` union type.

Structure types are described by declaration descriptors of type `structdesc`. The only special component of such a descriptor is a pointer, `fields`, to a list of declaration descriptors for the components of the structure type.

Array types are described by declaration descriptors of type `arraydesc`. An array declaration descriptor contains a `size` field in which the compiler will store the number of elements in the array. It also needs to somehow represent the element type of the array. The natural way to accomplish this would be to have the array type's declaration descriptor point to the descriptor for the element type. We will use a somewhat more complex scheme (for reasons that will not become clear until near the end of the project). For each array you will need to create a declaration descriptor for an imaginary variable of the array's element type. The array type's `elmntvar` field will then be set to point to this variable descriptor which will in turn point to the descriptor for the array's element type.

```
    /* Structure used to reference common fields of type descriptor.    */
struct unktypedesc {
   COMMONFIELDS
   int typesize;                  /* Memory required for var. of this type */
 };


    /* Structure used for array type declaration descriptors.     */
struct arraydesc {
   COMMONFIELDS
   int typesize;                /* Memory required for array of this type */
   int size;                    /* Number of elements in array            */
   union dcldesc * elmntvar;    /* decl. descriptor for imaginary variable of
                                       this array's element type      */
 };


    /* Structure used for struct type declaration descriptors.      */
struct structdesc {
   COMMONFIELDS
   int typesize;                /* Memory required for recd. of this type    */
   union dcldesc *fields;       /* Header for list of this type's components */
 };
```

Figure 6: Declarations for type declaration descriptors

3.2.2 Variable and Field Name Descriptors

The second group of related forms of declaration descriptors are those for things that act like variables: actual variables, formal parameter names and structure components. Again, we define a component of the `decldesc` union type that can be used to access the shared components of any one of the members of this group. This component is named `unkvar`. The declaration for the three declaration descriptor formats that fall in this group are shown in figure 7.

There are two fields shared by the declaration descriptors of variables, formals and structure components. The first is `vartype` which should be set equal to a pointer to the declaration descriptor for the variable or component type. The second is `disp` which should be set equal to the displacement to the field or variable within the structure or activation record in which it is located. These are the only special fields in the struct type used for variable declaration descriptors, `vardesc`.

Descriptors for formal parameters contain two additional fields. The `isVal` field is basically a Boolean flag set to "true" (i.e. 1) if the parameter is to be passed by value and to "false (i.e. 0) if the parameter is passed by reference. In addition, the `formallink` field is used to chain the descriptors for the parameters of a given function into a list.

Structure fields names are associated with declaration descriptors of type `fielddesc`. The `structtype` component of each such descriptor is used to hold a back pointer to the descriptor for the structure type to which the component belongs. The `hashlink` field is used to build the chains of descriptors that form the hash table used to look up the descriptor associated with a reference to a structure field name. Finally, the `fieldlink` component is used to chain all the declaration descriptors for a given structure's fields together in a linked list.

```
    /* Structure used for variable declaration descriptors.     */
struct vardesc {
   COMMONFIELDS
   union dcldesc * vartype;     /* decl. descriptor for the variable's type  */
   int disp;                    /* disp. within activation record or globals */
};

    /* Structure used for formal parameter declaration descriptors.    */
struct formaldesc {
   COMMONFIELDS
   union dcldesc * vartype;     /* decl. descriptor for the formal's type */
   int disp;                    /* disp. within activation record         */
   union dcldesc * formallink;  /* link for list of func's formals.       */
   int isVal;                   /* True if call by value formal           */
 };

    /* Structure used for struct field name declaration descriptors.   */
struct fielddesc {
   COMMONFIELDS
   union dcldesc * vartype;     /* decl. descriptor for component's type  */
   int disp;                    /* displacement to field within structure    */
   union dcldesc * hashlink;    /* link pointer for hash chain            */
   union dcldesc * structtype;  /* decl. descriptor for type to which the */
                                /*       component belong                 */
   union dcldesc *fieldlink;    /* Next pointer for type's list of fields */
 };
```

Figure 7: Declarations for variable/field name declaration descriptors

```
    /* Structure used for function declaration descriptors.    */
struct funcdesc {
   COMMONFIELDS
   union dcldesc * formallist;  /* head of list of this func's formals. */
   int paramcount;              /* Count of parameters func. expects    */
   int isfunc;                  /* True if this is a function.          */
   int localsize;               /* Size of space required for locals    */
   CODELBL * entrylbl;          /* Label placed on first line of func   */
};
```

Figure 8: Declarations for function name declaration descriptors

3.2.3 Function Name Descriptors

The only remaining form of declaration descriptor is that used to represent names defined as functions. The type used for such descriptors is named `funcdesc` and is shown in figure 8.

The `formallist` component of a `funcdesc` structure points to a list of declaration descriptors stored as structures of type `formaldesc`. This list should contain the declaration descriptors for all of the formal parameters associated with the function. This list will be used to type check calls made to the function. The `paramcount` field should be set equal to the number of parameters the function expects.

The `isfunc` field is a Boolean set to true if the name is declared as a function returning an int (rather than void). The number of memory units required for local variables and temporary storage is stored in `localsize`. Finally, the `entrylbl` field is used by the code generator to generate jumps to the function's entry point.