The C to TDF Producer

Issue 2.1.0 (June 1993)

Defence Research Agency
St Andrews Rd
Malvern
Worcestershire
WR14 3PS
United Kingdom

Notice to Readers

TDF is a portability technology and an architecture neutral format for expressing software applications which was developed by the United Kingdom's Defence Research Agency (DRA). DRA has demonstrated that the TDF technology can support ANSI C on MIPS®, Intel 386™, VAX™, SPARC™ and Motorola® 680x0.

Requests for information about TDF should be directed to:

Dr. N E Peeling
Defence Research Agency
St. Andrews Road
Malvern
Worcestershire
United Kingdom WR14 3PS

Tel. +44 684 895314
Fax +44 684 894303

Internet peeling%hermes.mod.uk@relay.mod.uk

This document is for advanced information. It is not necessarily to be regarded as a final or official statement by the Defence Research Agency.

June 1993

CONTENTS

1 Introduction

The C to TDF compiler has been designed specifically to allow and aid the construction of portable software. There are two main areas where programmers have to tackle the problems of portability, one *internal* to the compiler the other *external*.

Internal portability considerations are where there is room for the TDF produced by the compiler to behave differently on different machines. This can occur if the TDF produced is undefined, or implementation defined, leaving the possibility of different installers installing them differently.

External portability considerations are where different programs are required to represent the same application on different systems/platforms. This occurs, for example, when different algorithms are required for different platforms, or when the application has to interface with different software. In such cases, the programmers' aim is usually to write their software in such a manner as to separate the portable from the non-portable program. The most obvious example of such a division is the writing of an application sitting on top of an API, such as POSIX. The basic idea is to use the same program on all machines/systems with the non-portable components being provided by the implementations of the APIs on each system.

1.1 Error detection

The compiler cannot detect all programs that are not portable; to do so, whilst still retaining a sufficiently general programming language, is not possible.

1.1.1 External Error detection

The compiler, by the explicit representation of the interfaces between the portable and non-portable parts of a program, is able to detect *external* portability errors which are usually missed. The representation of APIs, using these interfaces, provides a powerful mechanism for detecting non-conformance of applications to specific APIs. It simultaneously provides a mechanism for detecting non-conformance of implementations of APIs. The description and method of use of these interfaces is described in Section 2.

1.1.2 Internal Error detection

The compiler is capable of performing a selection of checks to detect the use of C code that will compile to produce undefined or implementation defined TDF. These checks are controlled by the inclusion of pragmas which are described in Section 3.

1.1.3 Programming Error detection

By the inclusion of pragmas as detailed in Section 5, the compiler is capable of detecting some common programming errors. These are not errors associated particularly with portability, more with general consistency of the program. They can be thought of as being on the same footing as Lint checks, but the checks are performed during the compilation process by the compiler.

1.2 Levels of Portability

Different programs can require different degrees of portability. For example it is reasonable for a program to be written so that it is portable to 32-bit architectures, but not 16-bit architectures. The compiler allows the programmers to include in their programs pragmas that control the degree of portability against which internal checks are made see Section 1.1.2.

A typical use of different portability levels occurs with the separation of a program into portable and non-portable components. The portable part of the program may be designed to run on all machines which architectures of 16 bits or greater, whereas the non-portable part may be designed only to run on 32 bit architectures. The separation of the compilation process described in Section 2 allows the appropriate checks to made on the different parts of the program.

1.3 ANSI C Compatibility

The compiler, except as detailed in Section 5, is a strictly conforming ANSI C compiler, Ref [1]. However, by the use of pragmas, the compiler has been extended to:

1. allow program separation, as described in Section 2.
2. to provide information for portability checking, as described in Section 3.
3. to instruct the compiler to accept other dialects of C, as described in Section 4.
4. to provide consistency checking, as described in Section 5.

Many of the dialects of C can be viewed as extensions of ANSI C, however, some are inconsistent with ANSI C. The compiler, by the use of pragmas, accommodates both classes of variant, with the possibility of providing warnings whenever the use of a variant is required. The details of the pragmas that effect the non-ANSI variants are described in Section 4.

1.4 Implementation details

1.4.1 *Startup* and *Termination* files

The compiler is designed to be flexible, that is it can be customised to meet a variety of needs as introduced in the previous sections. However, most programmers would prefer to set up the compiler once and use that setup thereafter. This is performed by using *startup* and *termination* files with *tcc* environments (Ref [4]).

A *startup* file is a file that is included by the compiler prior to the compilation of the main source file. It usually consists of pragmas for customising the compiler and is usually supplied automatically by *tcc*.

A *termination* file is a file that is included by the compiler after the compilation of the main source file. There is no standard use of this facility, but it can be used for uniformly providing information about objects declared in the main body, for example identifying the initialization function produced as a result of executing C front on a C++ source file (see Section 6.2).

1.4.2 Error Messages

The error messages produced by the compiler have been designed to provide as much information as is practical about the reason for the message. To this end an error message often consists of several sub-messages which correspond to the conditions that exist to necessitate the message. Associated with each sub-message is a reference to the documentation that provides a more complete explanation for the message.

There are only two documents referenced by these error messages, this one and the ANSI C Standard, Ref [1]. In the case of the ANSI Standard the precise section number has been identified, but, currently, in the case of this document only the general sections are referred to.

The references included in the error messages can be indentified thus:

1. **ANSI[*section_no*]:** is a semantic error and refers to the ANSI C standard with *section_no* being the relevant section.

2. **Syntax:** is a syntax error, violating the syntax as described in the ANSI C standard.

3. **Implementation:** is a semantic error which refers to Section 4 of this document.

4. **Portability:** is a semantic error which refers to Section 3 of this document.

5. **Tokenisation:** is an semantic error which refers to Section 2 of this document.

1.5 Typing convention

The only adoption of a typing convention is in the description of syntax. The convention is as follow:

1. Italics refer to syntactic classes

2. Bold italics refer to syntactic classes described in the ANSI C standard (Ref [1]).

3. Straight bold letters and symbols are keywords or basic symbols.

1.6 Prior Knowledge

The compiler is based upon the ANSI C standard (Ref [1]) and many of the implemented features are based on concepts which are introduced in the standard. Unless readers have a sound knowledge of ANSI C or access to the standard, they may find parts of the document difficult to comprehend.

2 Program Construct Tokens

```
#pragma token TYPE FILE#
#pragma token EXP rvalue:FILE*:stderr#

int fprintf(FILE *, const char *, ...);

void f(void){
  fprintf(stderr,"hello world\n");
  }
```

Figure 2.1

2.1 Interfaces and Tokens

The major difference between the program in Figure 2.1 and conventional C programs is the presence of the *pragma*s occurring on the first two lines. They, by means of the *token syntax* (see Section 2.3) introduce *tokens* into the C program.

In the same manner as a function prototype introduces a reference to a function which is defined in a separate compilation module, the token syntax introduces a reference to a component of the program that can be defined in a separate compilation module. These references to program components are referred to as *tokens*, because they are a token for the program which they reference. The token syntax is said to specify a *token interface*.

The first line, of Figure 2.1, introduces a token for a type. This token being identified by *FILE*. By using its identifier *FILE*, the token introduced on the first line can be used in the remainder of the program wherever a type would normally be used. This makes it possible to compile, to TDF, a program which makes use of a type even though its definition is unknown. This is fundamental to the construction of portable software where the application developer does not know the definitions of many of the types that are used because they are potentially different on each machine/system.

The second line is more complex. It introduces a token for an expression identified by *stderr*. However, to make use of an expression, it is necessary to know what its type is and whether or not it is an *lvalue*, i.e. whether or not it can be assigned to. But, as this example illustrates, it is not necessary to know precisely the type of the expression because it also can include a token for a type.

The compiler makes no assumptions about the definitions that may be used to define a token, and exits with a constraint error whenever a program requires information about an undefined token. This detects many external errors (see Section 1.1.1) where applications have inadvertently made use of a definition that is present on the system on which the application is being developed. For example often the writers of applications assume that the type *FILE* is implemented by a structure type, but the ANSI Standard API permits *FILE* to be implemented by any type. The program in Figure 2.1 would not compile if there was an attempt to access a member of *stderr*.

```
# pragma token token-introduction token-identification
```

Figure 2.2

2.2 Construction phases

Traditionally, program construction using the C language has two phases, compilation and linking.

Compilation involves mapping source text, written in the C language, to an object code format. In general this is not an executable program because it is incomplete. For example, the program may use the function *fprintf* but not define it. In order to produce an executable program, it is necessary to *link* the object code format with other program segments, expressed in the same format, which provide the definitions of all the undefined objects. This process allows separation in the compilation of a program.

Program construction using tokens requires an extra phase where the undefined tokens using in one program segment are linked to their definition in another. TDF is fundamental to this process because it possesses the ability to represent the tokens introduced into the C language.

In consequence, program construction, via TDF, has four basic operations:

1. Source file compilation to TDF. The TDF produced may be incomplete in the sense that it may use tokens that are not defined.
2. TDF linking. This is analogous to object file linking, but is the linking of TDF and is performed to provide the definitions of the undefined tokens.
3. TDF translation. This is the conversion of the TDF into standard object file format for a particular machine/system
4. Object file linking. This corresponds directly to standard object file linking.

The program *tcc* (Ref [4]) is a harness for coordinating these phases in a similar fashion to that of *cc*. Its documentation provides details on all the operations in the complete compilation process.

In order for the TDF linker to operate, it is necessary to associate identifiers with the undefined tokens and their definitions. These identifiers perform a similar function to the global names of an object file format, that is, during linking, tokens represented in different pieces of TDF with the same identifier are treated as the same token.

2.3 The token syntax

The token syntax is an addition to the ANSI C Standard language to allow tokens to be used for program constructs. This section, (i.e. Section 2), gives details of all the various kinds of program construct that can be represented as tokens by the compiler, and describes how to introduce, use and define these tokens.

The basic token syntax is given in Figure 2.2. The syntax is introduced as a pragma so that other compilers can ignore it. In many cases, if the pragmas are ignored, the program will not compile, because they exist in place of definitions required by the compiler. However, it is possible to have both tokens and definitions. In this case the tokens serve to check the correctness of the use and definition of the defined objects, and the pragma can then be legitimately removed. There is, though, the possibility that the program will have different semantics depending on whether or not the token introduction is present (see Figure 2.51). This change in semantics is to force a program to respect the given token interface where it would otherwise fail to do so.

```
token-identification:
  name-space_opt identifier # ext-identifier_opt

name-space:
  TAG
```

Figure 2.3

The *token-introduction* introduces a token; it defines the kind of token, i.e. whether it is a token for a type, an expression, etc., and any additional information associated with that kind of token (for example an expression token is characterised by its type and whether or not it is an lvalue).

The *token-identification* provides a means of identification of the introduced token.

2.4 Token Identification

The *token-identification* (see Figure 2.3) provides the method of referring to the token, both internally, that is, within the program being compiled, and externally, that is, for the purposes of linking, as discussed in Section 2.2.

2.4.1 Internal token identification

The first identifier of the *token-identification* provides the internal identification of the token. The optional *name-space* that precedes the internal identification, provides the name space in which it resides.

2.4.1.1 Name spaces

The ANSI standard, Ref [1], identifies five name spaces. These are:

1. The *label* space, in which all label identifiers reside.

2. The *tag* space, in which structure, union and enumeration tags reside.

3. The *member* space, in which structure and union member selectors reside.

4. The *ordinary* space, in which all other identifiers reside.

5. The *macro* name space, in which all macro definitions reside.

It is possible for a token identifier to reside in any of these name spaces. However, there are specific restrictions depending on the kind of object that a token represents. For example, types cannot reside in the *member* name space.

If a token identifier is placed in the *macro* name space, then, unlike other macro names, not having a definition, it is not expanded. However, in all other respects it behaves like a macro identifier, i.e.:

1. The result of *defined* within a *#if* conditional expression is *1*, with the corresponding consequences for *#ifdef* and *#ifndef*.

2. It is possible to terminate the scope of the identifier with *#undef*.

3. It hides all other declarations of the same identifier in all other name spaces.

At present, it is only possible to explicitly require that an identifier be placed in the *tag* name space. All other name spaces are entered by using the default name space, see Section 2.4.1.2. It is anticipated that, in future, it will be possible to specify all of the possible name spaces.

2.4.1.2 Default name space

If there is no name space provided with a token identification, then a default name space will be assumed. This name space will depend on the kind of token being introduced. For example, if the token being introduced is for an expression,

```
token-introduction:
  exp-token
  statement-token
  type-token
  selector-token
  procedure-token
```

Figure 2.4

then the default name space is the *macro* name space; that is, it is as if the expression had been introduced with *#define*. If the introduced token was for a type, then the default name space would be the *ordinary* name space, as if the type had been introduced with *typedef*.

With every kind of token which can be introduced, there is associated with it a default name space. This name space is detailed in the section describing the particular kind of token.

2.4.2 External token identification

The *ext-identifier* in the *token-identification*, if present, provides the external identification of the token which is used for TDF linking. It can be any sequence of characters except for new line. There is only one name space for the external identification of tokens, and this is different from the external name space of functions and objects. This means that, for example, it is possible to have a function and a token both identified externally by *putc*. If no external identification is provided, then it is assumed that the internal and external identification are the same.

2.5 Kinds of token

There are, currently, five kinds of token. That is, there are five different kinds of program construct that can be represented by tokens. These kinds are Expression, Statement, Type, Selector and Procedure. These tokens are introduced using the *token-introduction* syntax of Figure 2.2. They approximately correspond to *expression*, *statement*, *type-name*, *member-designator* and *function-like macro* as described in the ANSI standard, Ref [1]. The introduction, use and definition of each of these kinds of token will be described in the following sections.

It is known that this set of kinds of token, and the ways of introducing them are not complete, and it is intended to perform a review of the existing syntax. However, the set of constructs permitted by the compiler at the moment have allowed the expression of most of the existing APIs, see Ref [3].

2.6 The Expression token

The following properties are associated with expression tokens:

1. Designation: *Designation* is a classification of the value delivered by evaluating an expression, see Section 2.6.1.

2. Type: *Type* is the type of the expression ignoring any type qualification.

3. Constancy: *Constancy* is the property of being a constant expression as described in ANSI 3.4.

The validity of programs that use expression tokens can depend on the properties associated with the expression. For example, the unary **&** operator is invalid on a value designation, such as **(3+4)**.

```
exp-token:
  EXP exp-storage : type-name :
  NAT

exp-storage:
  rvalue
  lvalue
```

Figure 2.5

```
#pragma token EXP lvalue:int:errno#

void f(void){
  errno = 5;
  errno += 2;
  }
```

Figure 2.6

2.6.1 Expression Designations

The designation of an expression describes the nature of the result of evaluation of the expression represented by the token.

There are three designations, implied by ANSI 3.3.

1. Value Designation, where the expression describes the computation of a value
2. Object Designation, where the expression designates an object. Associated with an object designator is a *type-qualifier* giving the access conditions of the object.
3. Function Designation, where the expression designates a function.

2.6.2 General Expression Introduction

The general method for introducing a token for an expression is (see Figure 2.5):

```
EXP e_s : type :
```

e_s is an *exp-storage* which is either **rvalue** or **lvalue**. If it is **rvalue**, then the introduced token is either a value or function designation, depending on whether or not *type* is a function type. If *e_s* is **lvalue**, then the token is an object designation without any type qualification.

type is the type of the expression to which the token refers.

Any expression introduced with a general introduction is assumed not to be constant.

The expression introduction in Figure 2.6 is a typical use of an expression token. It introduces an object designator of type int, which, since it has no type qualifiers, is modifiable.

2.6.3 Constant Expression Introduction

At the moment there is very limited support for introducing tokens for constant expressions. The token introduction

```
NAT
```

can be used for introducing constant integral expressions. This is a constant value designator of type **int**.

2.6.4 Name Spaces

Internal expression token identifiers can be placed in the *macro* name space only. The

```
primary-expression:
  identifier
  constant
  string-literal
  ( expression )
  exp-token-name
```

Figure 2.7

```
#pragma token EXP rvalue:int:x#

int f(int y){return y*x;}
```

Figure 2.8

default name space is, consequently, the *macro* space.

There is no reason why expression token identifiers cannot be placed in the *ordinary* name space. However, this has not yet been implemented.

2.6.5 Using expression tokens

In order to make use of tokenised expressions, a new symbol, *exp-token-name*, has been introduced into the syntax analysis at phase 7 of translation, Ref [1]. The pre-processor, on encountering an identifier for an expression token passes an *exp-token-name* symbol through to the syntax analyser. An *exp-token-name* is very similar to a ***typedef-name***, Ref [1], but whereas the **typedef-name** provides information about the type that has been introduced with a **typedef**, the *exp-token-name* provides information about the token for the expression. The only place where this symbol can occur is as part of a ***primary-expression*** (ANSI C Standard, Section 3.3.1), Figure 2.7. The expression resulting from the use of *exp-token-name* will have the type, designation and constancy specified at the introduction of the token.

Consider the program in Figure 2.8. The pragma introduces a token for an expression with internal and external name *x*. This token is then used in the definition of *f* to compute the result of the function. The use of *x*, after its declaration, is very similar to that of an ordinary identifier. However, because it is in the *macro* name space:

1. It is impossible to hide it by the use of an inner scope.
2. From the point of view of *#ifdef* etc., *x* is defined.
3. Its scope can be terminated by *#undef x*.
4. *#define x*, which would normally be treated as a re-definition of *x*, has another meaning, see Section 2.6.6.

2.6.6 Defining expression tokens

The definition of an expression token is similar to that of the definition of a macro, see Figure 2.9, and the example Figure 2.10.

2.6.6.1 Constraints

1. If the *exp-token-name* refers to an expression that is constant, then the assignment-expression must be a constant expression as specified in the ANSI C Standard, Section 3.4.

```
expression-definition:
  # define exp-token-name assignment-expression
```

Figure 2.9

```
#pragma token EXP rvalue:int:x#

extern int z;

#define x z+1
```

Figure 2.10

```
#pragma token EXP lvalue:int:i#

extern short k;

#define i 6 /*Violation of Constraint 2*/
#define i k /*Violation of Constraint 2*/
```

Figure 2.11

2. If the *exp-token-name* refers to an expression that has an object designation, Section 2.6.1, then the defining expression must designate an object; the type of the expression token must be resolvable to that of the defining expression (Section 2.8.8.1)and all of the type qualifiers of the defining expression must appear in the object designation of the token expression.

3. If the *exp-token-name* refers to an expression that has function designation, then the type of the expression token must be resolvable to that of the defining expression (Section 2.8.8.1).

Figure 2.11 provides two examples of the violation of the second constraint. The first violation occurs because the expression, *6*, does not designate an object. The second violation is because the type of expression, *k*, i.e. *short*, is incompatible with that of the token expression, i.e. *int*.

2.6.6.2 Semantics

If the *exp-token-name* refers to an expression that designates a value, then the defining expression is converted, as if by assignment (ANSI C Standard, Section 3.3.16.1), to the type of the expression token using the *assignment resolution operator* (Section 2.8.8.1); but with all the other designations the defining expression is left unaltered. The resulting expression serves as the definition of the expression token.

Consider the example in Figure 2.12. The definition of *li* causes the defining expression, *6*, to be converted from a value of type *int* to a value of type *long*. This is essential for the separation of the use from the definition of *li*. However, as a direct consequence of this, it is apparent that the introduction of a token for *li* has subtly changed the meaning of the program. If the parameter *x* is non-zero, with the token introduction, the function *f* returns

```
#pragma EXP rvalue:long:li#

#define li 6

int f(int x){
  if(x)return li;
  return sizeof(li);
  }
```

Figure 2.12

```
#pragma token EXP rvalue:int:X#

#define X M + 3

#define M sizeof(int)

int f(int x){return x+X;}
```

Figure 2.13

```
#pragma token EXP rvalue:int X#

#define M sizeof(int)
#define X M + 3                                              /*line 4*/
#undef M

int M(x){return x+X;}                                        /*line 7*/
```

Figure 2.14

sizeof(long), whereas without it *f* returns *sizeof(int)*.

2.6.6.3 Phases of translation

A major difference between the defining of a macro and the defining of an expression token is that, whereas a macro is defined by any ***pre-processing tokens***, an expression token is defined by an ***assignment-expression***. Pre-processing tokens are computed in phase 3 of translation and ***assignment-expression***s are computed in phase 7. Consequently *expression-definition*s are evaluated in phase 7 whereas macro definitions are evaluated in phase 4.

One of the consequences of this difference is illustrated by the program in Figure 2.13. In the absence of the token expression introduction, *X*, this program will compile. At the time the definition of *X* is interpreted, in *return x+X*, both *M* and *X* are in scope. In the presence of token introduction, the definition of *X*, being part of translation phase 7, is interpreted when it is encountered, and at this point *M* is not defined. This problem can readily be rectified by reversing the order of the definitions of *X* and *M*.

```
#pragma token EXP lvalue:int:errno#

extern int errno;
```

Figure 2.15

```
#pragma token EXP lvalue:int:errno#

extern int _errno;

#define errno _errno;
```

Figure 2.16

```
statement-token:
  STATEMENT
```

Figure 2.17

```
#pragma token STATEMENT init_globs#

int g(int);
int f(int x){init_globs return g(x);}
```

Figure 2.18

However, the reverse is true of the example in Figure 2.14. This example shows how token definitions can be used to relieve some of the pressures on name-spaces by undefining the macros that are only used in their definitions. The definition of *X* is computed on line 4, when *M* is in scope, not on line 7 where it is used. Care should be taken if using this facility since it may not be a straightforward matter to convert the program back to a conventional C program.

2.6.7 Defining Expressions with Externals

An alternative method of defining token expressions is by declaring the *exp-token-name* that references the token to be an object with external linkage. A typical example is given in Figure 2.15.

2.6.7.1 Semantics

If the object declared by the declaration has external linkage, then the expression that designates this object is used as the defining expression for the token expression. That is the semantics for Figure 2.15 are almost represented by Figure 2.16 except that *_errno* is replaced with *errno* after the definition of the token expression has been computed.

2.7 The Statement Token

The syntax for introducing a statement token is very straightforward and given in Figure 2.17,

```
statement:
  labelled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  stat-token-name
```

Figure 2.19

```
statement-definition:
  # define stat-token-name statement
```

Figure 2.20

an example of a declaration and use being given in Figure 2.18.

2.7.1 Name Spaces

Internal statement token identifiers can be placed in the *macro* name space only. It follows that the default name space is also the *macro* space.

2.7.2 Using statement tokens

The use of statement tokens is analogous to the use of expression tokens see Section 2.6.5.

A new symbol, *stat-token-name*, has been introduced into the syntax analysis at phase 7 of translation. This token is passed through to the syntax analyser whenever the pre-processor encounters the identifier referring to the statement token. The only place where *stat-token-name* can occur is as part of the ***statement*** syntax (ANSI C Standard, Section 3.6) as illustrated in Figure 2.19.

2.7.3 Defining statement tokens

As with the defining of expression tokens (Section 2.6.6) the definition of a statement token is similar to that of the definition of a macro (see Figure 2.20).

2.7.3.1 Constraints

The statement that forms the definition of the statement token is subject to the following constraints:

1. Unless the definition of the statement occurs at the outer level, that is not inside the ***compound statement*** (ANSI C Standard, Section 3.6.2) forming a function definition, there shall be no use of labels.

2. The use of **return** within the defining statement is disallowed.

It is anticipated that both of these restrictions will be removed in future releases.

2.7.3.2 Semantics

The semantics of the defining statement are identical to that of the *compound statement* forming the definition of a function with no parameters and ***void*** result.

2.7.3.3 Phases of translation

There are precisely the same implications on the phases of translation with the definition of statement tokens as there are for the expression tokens (Section 2.6.6.3).

```
type-token:
  TYPE
  VARIETY
  ARITHMETIC
  STRUCT
  UNION
```

Figure 2.21

2.8 Type tokens

The ANSI C Standard, Section 3.1.2.5, identifies the following classification of types:

1. the type ***char***
2. signed integer types
3. unsigned integer types
4. floating types
5. character types
6. enumeration types
7. array types
8. structure types
9. union types
10. function types
11. pointer types

These form part of other type classifications:

1. integral types - consisting of the signed integer types, the unsigned integer types, and the type ***char***.
2. arithmetic types - consisting of the integral types and the floating types.
3. scalar types - consisting of the arithmetic and the pointer types.
4. aggregate types - consisting of structure and array types.
5. derived declarator types - consisting of array, function and pointer types.

The classification of a type is critical in the determination of which operations are allowed to be performed on objects of that type. For example, the operator ***!***, in the ANSI C Standard, Section 3.3.3.3, can only operate on objects with scalar type. For this reason, type token introductions allow the type being introduced to be classified, thereby allowing the compiler to perform the semantic checking required in the application of operators.

The syntax for a *type-token* introduction is given in Figure 2.21.

2.8.1 General type tokens

The most general type-token introduction is ***TYPE***. This introduces a type of unknown classification and can be defined to be any C type. This means that only a few generic operations, like procedure token application, see Section 2.10, can be applied to objects of such an introduced type. The main uses of such a token are in the construction of derived declarator types, Section 2.8.2, and abstract data types.

There are few truly generic operations in ANSI C. Even the assignment operation is not permitted if the left operand has array type. However, in the cases of assignment and function call, it has been possible to provide an extension of their semantics to allow their use with general type tokens. In practice, these operations are not generic only because of the anomalous treatment of arrays.

Consider the program in Figure 2.22, ignoring initially the commented out typedef. It intro-

```
#pragma token TYPE t_t#

/*typedef int t_t[4];*/

int f(t_t)
t_t g(void);

int h(void){
  t_t x = g();
  return f(x);
  }
```

Figure 2.22

```
#pragma token TYPE t_t#
#pragma token TYPE t_p#
#pragma token NAT n#

typedef t_t *ptr_type; /*introduces a pointer type*/

typedef t_t fn_type(t_p); /*introduces a function type*/

typedef t_t arr_type[n]; /*introduces an array type*/
```

Figure 2.23

duces a general type token, *t_t*, and then, in the function *h*, assigns a value of type *t_t* to *x* and then passes an object of type *t_t* to the function *f*.

The semantics of the assignment and argument passing of *t_t* must be defined for all possible substituted types. The commented out typedef is to encourage a comparison between the use of type token *t_t* and the corresponding use of an array type.

2.8.1.1 Conversions

The only standard conversion (ANSI C Standard, Section 3.2) that is performed on an object of general token type is the *lvalue* conversion. When performing the *lvalue* conversion (ANSI C Standard, Section 3.2.2.1) of an object with a general token type, say *t_t*, the *lvalue* is converted to the object stored in the designated object.

This produces a fundamentally different semantics between defining *t_t* to be an array type and defining *t_t* to be a general token type, and later defining that token to be an array type. In the former case *lvalue* conversion will deliver a pointer to the first element, whereas in the latter case, *lvalue* conversion delivers the components of the array.

The definition of the *lvalue* conversion of objects with general tokenised types, leads the way for the semantics of assignment and function argument passing. In the cases where the type token is defined to be an array, then the *components* of the array are assigned and passed as arguments to the function call; in all other cases, the assignment and function call are as if the defining type had been used directly.

```
#pragma token VARIETY i_t#

short f(void){
  i_t x_i = 5;                                        /*line 4*/
  return x_i;                                         /*line 5*/
  }

short g(void){
  long x_i = 5;                                       /*line 8*/
  return x_i;                                         /*line 9*/
  }
```

Figure 2.24

2.8.1.2 Name Spaces

The local identifier of a general type token can only reside in the *ordinary* name space. Consequently, this is the default name space. The local identifier behaves exactly as if the type had been introduced with ***typedef*** and consequently is treated as a ***typedef-name*** by the syntax analyser.

2.8.2 Derived Declarator types

Derived declarator types are constructed using the conventional type declarators. Examples of their construction are given in Figure 2.23.

2.8.3 Integral type tokens

The type token introduction ***VARIETY*** introduces a token representing the *integral* types. It can only be defined as an integral type and consequently can be used wherever an integral type is valid.

2.8.3.1 Name Spaces

As with general type tokens, Section 2.8.1.2, integral type tokens only reside in the *ordinary* name space. They also behave as if they had been introduced with ***typedef***.

2.8.3.2 Conversions

Values which have integral type token, can be converted to any scalar type. Similarly, any value with scalar type can be converted to a value with an integral token type. The semantics of the conversion are identical to those where the type defining the token is used, instead of the integral type token, in the conversion.

For example, consider the program in Figure 2.24. Within the definition of function *f*, there is a conversion from *int*, the type of *5*, to *i_t*, the integral type token, on line 4, and a conversion from *i_t* to *short*, the return type of *f*, on line 6. If, at some stage, *i_t* is defined as *long*, then the function *f* will be equivalent to the function *g*.

2.8.3.3 Integral promotions

Integral promotions (ANSI C Standard, Section 3.2.1.1) are defined according to the rules introduced in Section 4.1, and are applied to all integral type tokens where required by the ANSI C Standard.

2.8.3.4 Arithmetic Conversions

The usual arithmetic conversions (ANSI C Standard, Section 3.2.1.5) are defined on integral type tokens. These conversions are applied where required by the ANSI C Standard.

The integral promotions, Section 2.8.3.3, are first applied to the integral type token and then the usual arithmetic conversions, are applied to the resulting integral type. When the integral promotion is an integral type token the semantics of the conversion are as if the conversion were being applied to the defining integral type, c.f. Section 2.8.3.2.

```
#pragma token STRUCT n_t#                                   /*line 1*/
#pragma token STRUCT TAG s_t#                               /*line 2*/
#pragma token UNION TAG u_t#                                /*line 3*/

void f(){
  n_t x1;            /*n_t is the compound type on line 1*/
  struct n_t x2;     /*This is illegal, n_t is not in the tag space*/
  s_t x3;            /*This is illegal, s_t is not in the ordinary space*/
  struct s_t x4;     /*struct s_t is the compound type on line 2*/
  union u_t x5;      /*union u_t is the compound type on line 3*/
  struct u_t x6;     /*This is illegal, u_t is introduced as a union tag*/
  }
```

Figure 2.25

2.8.4 Floating type tokens

At present floating type tokens are not implemented. This should be rectified in future releases.

2.8.5 Arithmetic type tokens

The type token introduction ***ARITHMETIC***, introduces an *arithmetic* type token. In theory arithmetic type tokens can be defined by any arithmetic type, but this implementation of the compiler only permits their definition by integral types. In consequence, in all respects, arithmetic type tokens behave as if they were integral type tokens, Section 2.8.3.

It is also intended to rectify this limitation in future releases.

2.8.6 Compound type tokens

A *compound type*, as defined in this document, is a type which describes objects that have components that are accessible by *member selectors*. Clearly, all *structure* and *union* types are *compound* types. However, compound types, unlike structure and union types, do not necessarily have an ordering on their member selectors. This means that, in particular, it is not possible to initialize, with an ***initializer-list,*** (ANSI C Standard, Section 3.5.7) all objects with compound type.

Compound type tokens are either introduced by ***STRUCT***, or by ***UNION***. It is assumed that programmers will introduce compound types with non-overlapping member selectors as ***STRUCT*** and those with overlapping member selectors as ***UNION***. In spite of this, no matter how the compound type token is introduced, it can be defined with any compound type.

2.8.6.1 Name Spaces

The local token identifiers of compound type tokens can reside in either the *ordinary* space or the *tag* space. The default name space is the *ordinary* name space. If the identifiers are placed in the *ordinary* name space, it is as if the type had been declared with ***typedef***. If the local identifiers are placed in the *tag* name space, it is as if the type had been declared with ***struct*** *id* or ***union*** *id*, where *id* is the local identifier.

Examples of compound type token introductions are given in Figure 2.25. Although the use of the type *struct u_t* is marked as illegal, it is possible, by the use of a pragma (see Section 4.2), to allow this type specification.

2.8.6.2 The Use of Compound Types

Values and objects whose type is a compound type token are valid everywhere that a structure or union type is valid.

The compound type token introduction does not introduce the member selectors of the compound type. They are added afterwards, see Section 2.9.

```
#pragma token TYPE t_t#                                    /*line 1*/

typedef t_t *ptr_t_t;                                      /*line 3*/
typedef int **ptr_t_t;                                     /*line 4*/
```

Figure 2.26

```
#pragma token EXP rvalue:int:N#

typedef int arr[N];
typedef int arr[4*sizeof(int)];
```

Figure 2.27

2.8.7 Type token compatibility

A type represented by a type token is incompatible (ANSI C Standard, Section 3.1.2.6) with all other types, except for itself, unless the type-token has been provided with a definition, in which case it is compatible with everything that is compatible with its definition.

2.8.8 Defining type tokens

There is, currently, only one way of providing a definition for a type token. This method is referred to as *type resolution*. Type resolution is an operation, similar to type compatibility (ANSI C Standard, Section 3.1.2.6), operating on two types. Essentially, the operation of type resolution is identical to that of type compatibility, except in the case where an *undefined* type token is found to be incompatible with the type with which it is being compared. In this case the type token is defined by the type with which it is being compared, thereby making the types compatible (see Section 2.8.7).

In order to allow the convenient definition of type tokens by resolution, the compiler allows the consistent redefinition of a type. In order to accommodate the ANSI C Standard (see Section 4.3) which disallows the redefinition of a type, a resolution of the types in the two definitions is performed and the redefinition is allowed if:

1. There is a resolution of the two types.
2. As a result of the resolution, there is at least one definition of a token.

Consider the program in Figure 2.26. The second definition of ptr_t_t cause a resolution of the types *(t_t *)* and *(int **)*. This resolution, following the normal rules of compatibility, i.e. two pointers are compatible if their dependent types are compatible, requires the resolution of *t_t* and *(int *)*. This resolution results in the definition of *t_t* as *(int *)* and consequently the definition is allowed.

The program in Figure 2.27 illustrates how type resolution can be used to define expression tokens. The resolution of the two types defining *arr* causes the expression token *N* to be defined as *(4*sizeof(int))*.

2.8.8.1 The resolution operators

The resolution operator, unlike type compatibility, is not symmetric. If the two ***typedef***s in Figure 2.26 are reversed, no resolution would take place and the two types would be incompatible. A *resolution* of two types, *A* and *B*, is an attempt to *resolve* the type *A to* the type *B*, that is only the undefined tokens of *A* can be defined as a result of the application of the resolution operator.

In the case of a re-definition of a type, the first definition is *resolved to* the second, thereby only allowing any definition of tokens present in the first type definition.

```
selector-token:
  MEMBER type-name : type-name :
```

Figure 2.28

```
#pragma token STRUCT TAG lconv#
#pragma token MEMBER char*:struct lconv:negative_sign#

struct lconv f(void){
  struct lconv res;
  res.negative_sign="-";
  return res;
  }
```

Figure 2.29

Similarly, if an identifier for a compound type token is introduced into the *tag* name space and that tag is later defined to be a compound type, then the compound type token is resolved to the defining compound type.

There is another operator, the *assignment resolution* operator. This is similar to the *resolution* operator, but whereas the resolution operator is used when checking the compatibility of two types, the *assignment resolution* operator is used when converting an object of one type to another for the purposes of assignment (ANSI C Standard, Section 3.3.16.1). In this case, if the conversion is not possible, but would be possible if an undefined token, of the type to which the assignee is being converted, were appropriately defined, then that token is defined appropriately. There are, in some cases, more than one possible definition of such a token, but, in these cases, there is always one definition that does not cause any conversion. This is the definition that is chosen.

2.9 Selector Tokens

The use of selector tokens is the primary method of adding member selectors to compound type tokens (Section 2.8.6); the only other method being to provide a structure or union definition of the token. They can also be used to add new member selectors to existing structure and union types.

The syntax for introducing a selector token is given in Figure 2.28. The first ***type-name*** is the type of the object selected by the introduced selector, and the second ***type-name*** is the compound type to which the selector is being added.

A selector token can be defined by any ***member-designator*** (ANSI C Standard, Section 4.1.6) of the compound type to which it belongs. In consequence, when the compound type is a compound type token, the token member selectors cannot be defined until the compound type token has been defined. Often, the definition of the compound type token and its member selector tokens occur together.

2.9.1 Member Selector Ordering

In the declaration of structure and union types within the ANSI C standard, there is an implied ordering of the member selectors. This is generally insignificant except in two cases.

1. Initialization with ***initializer-list***. The identified members of a structure are initialized in the order in which they are declared. The first identified member of a union is initialized.

```
#pragma token TYPE t_t#                                 /*line 1*/
#pragma token STRUCT s_t#                               /*line 2*/
#pragma token MEMBER t_t:s_t:mem_x#                     /*line 3*/
#pragma token MEMBER t_t:s_t:mem_y#                     /*line 4*/

struct s_tag {int a, mem_x, b;};                        /*line 6*/
typdef struct s_tag s_t;                                /*line 7*/
```

Figure 2.30

2. Comparison of pointers. The addresses of members of structures will increase in the order in which they are declared

The member selectors introduced as selector tokens are not related, until defined, to any other member selector. In particular, this means that if a compound type only has undefined token selectors, as is the case for most structure types of APIs, then the composite type cannot be initialized with an initializer list.

The decision to allow unordered member selectors is not accidental. It enables the separation of the decision as to what members belong to a structure and that of where those member components lie.

2.9.2 Name Spaces

The local identifiers of the introduced member selectors can only reside in the member space of the compound type to which they belong. This is also the default name space.

2.9.3 The use of selector tokens

An example of the use of a selector token is given in Figure 2.29. This example is taken from the ANSI C Standard Library Section 4.4.2.1. The standard does not specify all the members of *struct lconv* or which order they appear in, and so cannot be represented naturally by existing C types.

The introduction of new member selector tokens can occur at any point in the program, thereby making it possible to represent those extensions to APIs that require extra member designators.

2.9.4 Defining token selectors

There are two methods for defining token selectors, one implicit and the other explicit.

2.9.4.1 Implicit selector token definition

The implicit definition of a token selector occurs when the token is a selector, identified by, say, *id*, for an undefined token compound type, say *A*. When *A* is defined by another compound type, say *B*, and *B* has a member selector with identifier *id*, then *A.id* is defined to be *B.id*.

There is one constraint:

1. The type of *A.id* must be resolved to the type of *B.id* (see Section 2.8.8).

An example of implicit selector token definition is given in Figure 2.30. The type definition on line 7 is a redefinition of the type definition of *s_t* implicitly introduced on line 2 (Section 2.8.6.1). This redefinition caused a resolution between the token introduced on line 2 and the structure, *struct s_tag*, introduced on line 6. This resolution causes the token, of line 2, to be defined. As a direct consequence of this definition, the token for the selector *mem_x*, introduced on line 3 is defined as the second member of *struct s_tag*, and the consequential resolution of the type of *s_t.mem_x* to the type of *struct s_tag.mem_x*, causes the type token *t_t*, introduced on line 1, to be defined as *int*. Since *mem_y* is not a member of *struct s_tag*, the token selector introduced on line 4 is not defined.

```
#pragma DEFINE MEMBER type-name identifier : member-designator

member-designator:
  identifier
  identifier . member-designator
```

Figure 2.31

```
#pragma token STRUCT s_t#
#pragma token MEMBER int : s_t : mem#

typedef struct {int x; struct{char y; int z;} s;} s_t;

#pragma DEFINE MEMBER s_t : mem s.z
```

Figure 2.32

```
typedef struct {int x; struct{char y; int z;} s;} s_t;

#define mem s.z

extern s_t mem;                                          /*line 5*/
```

Figure 2.33

2.9.4.2 Explicit selector token definition

The explicit definition of a selector token is performed by using a pragma. The syntax of this pragma is given in Figure 2.31.

The ***type-name*** provides the compound type, say *C*, of which the selector to be defined is a member. The ***identifier*** provides the identification of the member selector within that type. The *member-designator* provides the definition of the selector.

A member-designator identifies a selector, say *S*, of a compound type.

If the *member-designator* is an ***identifier***, then:

1. The identifier must be a selector of the compound type *C*.

The selector identified is the designated member, *S*.

If the *member-designator* is an identifier, say *id*, followed by a further *member-designator*, say *M*, then:

1. The identifier *id* must be a member, identifying a selector *S1*, of the compound type *C*.

2. The type of the selector identified by *id*, must have a compound type, say *M_C*.

3. The member designator, *M*, must identify a selector, say *S2*, of *M_C*.

The selector, *S*, identified by the *member-designator* as a whole is the combination of the selector *S1* followed by *S2*.

```
#define SWAP(T,A,B){T x; x=B;B=A;A=x;}

void f(int i1,int i2,char *c1,char *c2){
  SWAP(int,i1,i2)
  SWAP(char *,c1,c2)
  }
```

Figure 2.34

```
procedure-token:
  general-procedure
  simple-procedure
  function-procedure
```

Figure 2.35

As with implicit token definitions, there is one further constraint:

1. The type of the selector token must be resolved to the type of the selector identified by the *member-designator*.

An example of explicit selector token definition is given in Figure 2.32, which defines the token selector *mem* by the double selection *s.z*.

The nearest equivalent, in conventional ANSI C, is illustrated in Figure 2.33, but this has the disadvantage that *mem* is placed in the global *macro* name space, thereby corrupting line 5. Note that

```
#define mem s.z
```

could not be used to define the member selector of Figure 2.32 because *mem* is not in the *macro* name space.

2.10 Procedure tokens

Consider the program in Figure 2.34. The macro SWAP is essentially a statement that is parameterised by a type and two expressions. Procedure tokens are based on the same idea. Procedure tokens represent, and are defined by, program components that are parameterised by other program components.

The syntax for the introduction of procedure tokens is given in Figure 2.35. As is made clear by the syntax, there are three methods for the introduction of procedure tokens. These are described in the following sections.

2.10.1 General Procedure Introduction

The syntax for the general procedure token introduction is given in Figure 2.36 and an example use is given in Figure 2.37. The intention is that the definition of *dderef* should be a double dereference.

The final *token-introduction* of the *general-procedure* syntax characterises the kind of program construction being parameterised. In the example of Figure 2.37, the program construction that is being parameterised is an *rvalue* expression. The meaning of the type of the expression, *t*, is discussed in Section 2.10.1.1. At the moment the only kinds of program construct that can be parameterised are expressions and statements. Future developments should allow the parameterisation of types to give abstract type constructors.

```
general-procedure:
  PROC { bound-toks_opt | prog-pars_opt } token-introduction

bound-toks:
  bound-token
  bound-token , bound-toks

bound-token:
  token-introduction name-space_opt identifier

prog-pars:
  program-parameter
  program-parameter , prog-pars

program-parameter:
  EXP identifier
  STATEMENT identifier
  TYPE type-name
  MEMBER type-name : identifier
  PROC identifier
```

Figure 2.36

```
#pragma token PROC{TYPE t,EXP rvalue:t**:e|EXP e}EXP rvalue:t:dderef#
```

Figure 2.37

The local identifier of the procedure token is placed in a name space as if it were the construct being parameterised. In the example, this is the default space for expressions, which is the *macro* name space.

That part of the syntax enclosed between the braces describes the parameters to the procedure token. These are divided into two components, the *bound-toks* and the *prog-pars*.

2.10.1.1 Bound token dependencies

The bound token dependencies, *bound-toks*, describe the program constructs on which the token depends. They should not be confused with the parameters of the token which are described in Section 2.10.1.2. The procedure token in the example is dependent on both the expression, to be dereferenced twice, and the type of that expression. However, the token has only one argument, namely the expression to be dereferenced twice, from which it can deduce both dependencies required by the procedure token.

The dependencies are introduced in an identical fashion to the straightforward introduction of tokens, as described in the previous sections. The *name-space* is also as in the previous sections with the ***identifier*** corresponding to the local-identifier. However, the scope of the identifier terminates at the end of the procedure token introduction and, whilst in scope, hides all other identifiers in the same name space. The tokens are referred to as bound because they are local to the procedure token.

Once a dependency has been introduced, it can be used in the construction of any of the remaining components of the procedure token introduction. In the example of Figure 2.37, the expression dependency, *e*, is dependent on the type *t*, as is the expression constructed by the application of the procedure token.

```
#pragma token PROC{TYPE t,EXP rvalue:t**:e|EXP e}EXP rvalue:t:dderef#

char f(char **c_ptr_ptr){
  return dderef(c_ptr_ptr);                                  /*line 4*/
  }
```

Figure 2.38

```
#pragma token PROC{VARIETY v1,VARIETY v2,EXP rvalue:v1:e|TYPE v2,EXP e}\
                      EXP rvalue:v2: convert#

long f(char c){
  return convert(long,c);                                    /*line 5*/
  }
```

Figure 2.39

2.10.1.2 Program parameters

Program parameters, *prog-pars*, describe the parameters with which the token procedure is called. The procedure dependencies are deduced from the program parameters.

Each program parameter is introduced with a keyword to express the kind of program construct it represents, as follows:

1. The keyword *EXP* describes a parameter that is an expression. The identifier following *EXP* must be the identification of a bound token for an expression. The corresponding argument to the procedure token, which must be an ***assignment-expression***, is considered to be a definition of the bound token, thereby providing definitions for all the dependencies related to that definition. The semantics for the definition of expression tokens is given in Section 2.6.6. An example use is given in Figure 2.38. Here, the call of *dderef*, on line 4, with argument *c_ptr_ptr*, causes *c_ptr_ptr* to be treated as a definition of *e*. The consequential assignment conversion, using *assignment resolution* (Section 2.8.8.1), causes *(t **)* to be resolved to *(char **)* which in turn causes the dependency *t,* to be defined as *char*. This definition of *t* also provides the type of expression obtained from the application of the procedure token.

2. The keyword *STATEMENT* describes a parameter that is a statement. Its semantics correspond directly to those of *EXP*.

3. The keyword *TYPE* describes a parameter that is a type. The corresponding argument must be a ***type-name***. The parameter type is resolved to the argument type in order to define any related dependencies. An example of a *TYPE* parameter is given in Figure 2.39. The first argument of the call of the procedure token *convert* causes the bound token *v2*, by *resolution*, to be defined as *long*. The second argument causes *e* to be defined as *c* and *v1* to be defined as *char*.

4. The keyword *MEMBER* describes a member selector parameter. The ***type-name*** is the composite type to which the member belongs, and the *identifier* is the identification of the member of that composite type that is to be defined by the argument (which must be a *member-designator*). A typical example is *offsetof*, shown in Figure 2.40. The argument *struct s*, defines the bound type token *s*, The argument *st.i* defines the

```
#pragma token VARIETY size_t#
#pragma token PROC{STRUCT s,TYPE t,MEMBER t:s:m|TYPE s,MEMBER s:m}\
                     EXP rvalue:size_t:offsetof#

struct s {char c; struct{long l;int i;} st;};

size_t f(void){
  return offsetof(struct s,st.i);                        /*line 8*/
  }
```

Figure 2.40

```
#pragma token PROC{TYPE t,EXP lvalue:t:e1,EXP lvalue:t:e2|\
                    TYPE t,EXP e1,          EXP e2\
                    }STATEMENT SWAP#
```

Figure 2.41

simple-procedure:
  PROC **(** *simp_tok_pars*$_{opt}$ **)** *token-introduction*

simp_tok_pars:
  simple-token
  simple-token , *simp_tok_pars*

simple-token:
  token-introduction name-space$_{opt}$ ***identifier***$_{opt}$

Figure 2.42

member selector *s.m*. The type *s* has already been defined by the first argument to be *struct s*. In consequence, the second argument, *st.i*, which defines the selector, *m*, of type *s*, is the selector *st.i* of *struct s*. A side-effect of the defining of the selector *m* is the defining, by resolution, of the type *t* to be *int*.

5. The keyword *PROC* describes a procedure token parameter. This has not been implemented yet.

2.10.2 Simple procedure tokens

The example in Figure 2.41 is the representation, as a procedure token, of the macro *SWAP* defined in Figure 2.34. The token introduction has been laid out to emphasise the similarity between the bound tokens and the program parameters.

This is such a frequent occurrence that a simpler form of the token syntax has been designed to cater for it. This syntax is shown in Figure 2.42. The semantics of the simple procedure syntax is based on that of the general procedure syntax, only the *simple-token* syntax

```
SIMPLE-TOKEN        BOUND-TOKEN             PROGRAM-PARAMETER

EXP rv:t:           EXP rv:t:_h             EXP _h
STATEMENT           STATEMENT _h            STATEMENT _h
TYPE                TYPE _h                 TYPE _h
VARIETY             VARIETY _h              TYPE _h
ARITHMETIC          ARITHMETIC _h           TYPE _h
STRUCT              STRUCT _h               TYPE _h
UNION               UNION _h                TYPE _h
MEMBER t:s:         MEMBER t:s:_h           MEMBER s:_h
```

Figure 2.43

```
#pragma token PROC(TYPE t,EXP lvalue:t:,EXP lvalue:t:)STATEMENT SWAP#
```

Figure 2.44

```
int putchar(int);

#pragma token PROC(EXP rvalue:int)EXP rvalue:int:putchar#
```

Figure 2.45

introduces both a bound token and a program parameter.

The *simple-token* syntax is almost identical to that of the *bound-token* syntax (Figure 2.36). The bound token, introduced by the *simple-token* syntax, is defined as if it had been defined with a *bound-token* syntax. If the final identifier is missing, then there must be no name space specified, and the bound token is not identified; there is, in effect a local hidden identifier. In addition to the bound token, a program parameter is also introduced. The generation of bound tokens and program parameters is shown in Figure 2.43, with *_h* as the hidden identifier. Figure 2.44 shows the expression of the *SWAP* procedure token in the new syntax. It can readily be shown to be equivalent to the introduction in Figure 2.41, with the introduction of two hidden identifiers for the two expressions.

2.10.3 Function procedure tokens

One of the commonest uses of simple procedure tokens is the representation of function in-lining. In this case the procedure token represents the in-lining of the function with the function parameters being the program arguments of the procedure token call and the program construct resulting from the call of the procedure token being the corresponding in-lining of the function. This is a direct parallel to the use of macros to represent functions.

An example of such a use is given in Figure 2.45. Since the syntax for the procedure token can be deduced from the type of the function, a *function-procedure* syntax has been designed to take advantage of it. This syntax is given in Figure 2.46 and the representation of the example introduced in Figure 2.45 is shown in Figure 2.47.

```
function-procedure:
  FUNC type-name :
```

Figure 2.46

```
#pragma token FUNC int(int): putchar#
```

Figure 2.47

```
#define identifier( id-list_opt ) assignment-expression
#define identifier( id-list_opt ) statement

id-list:
  identifier
  identifier id-list
```

Figure 2.48

The ***type-name*** of the function-procedure syntax must be a prototyped function type without an ellipsis. It is used to declare a function of that type with external linkage (ANSI C Standard, Section 3.1.2.2). It is also used to introduce the procedure token suitable for an in-lining of the function. Every parameter type and result type, say *ty*, is mapped onto the token introduction

```
EXP rvalue:ty
```

2.10.4 Defining procedure tokens

The process of defining of all kinds of procedure token, i.e. general, simple and function, is identical. Since simple and function procedure tokens can be transformed into general procedure tokens, the definition will be explained in terms of general procedure tokens.

As mentioned in Section 2.10.1, currently it is only possible to parameterise expressions and statements. The syntax for such definitions is given in Figure 2.48 and is based upon the standard parameterised macro definition. As with the definitions of expressions and statements (Section 2.6.6 and Section 2.7.3), these definitions are evaluated in phase 7 of translation, not phase 4.

The *id-list* must directly correspond to the program arguments of the procedure token introduction. That is, there must be precisely one identifier for each program argument. These identifiers are used to identify the program parameters of the procedure token being defined and consequently have a scope that terminates at the end of the procedure token definition. The identifiers are placed in the default name spaces for the kinds of program construction being identified. For example types are placed in the *ordinary* name space and expressions in the *macro* name space.

During the evaluation of the definition of a procedure token, none of the bound token dependencies can be defined, since these definitions are effectively provided at every call of the procedure token by its arguments.

To illustrate how this operates re-consider the *dderef* example in Figure 2.38 and its definition in Figure 2.49. The identifiers *t* and *e* are not in scope during the definition, being merely local identifiers for use in the token introduction. The

```
#pragma token PROC{TYPE t,EXP rvalue:t**:e|EXP e}EXP rvalue:t:dderef#

#define dderef(A)**A
```

Figure 2.49

```
#pragma token VARIETY size_t#
#pragma token PROC{STRUCT s,TYPE t,MEMBER t:s:m|TYPE s,MEMBER s:m|\
                  EXP rvalue:size_t:st_sizeof#

#define st_sizeof(T,M) sizeof( (T *)(NULL)-> M)
```

Figure 2.50

```
#pragma token PROC(TYPE t,EXP lvalue:t:,EXP lvalue:t:)STATEMENT SWAP#

#define SWAP(T,A,B){T x; x=B;B=A;A=x;}

void f(int x,int y){
  SWAP(int,x,y)
  }

/*WITHOUT THE PROCEDURE TOKEN --- {int x; x=y;y=x;x=x;} oops!!!*/
```

Figure 2.51

```
#pragma token FUNC int(int, long):func#

#define func(A,B)(func)(A,B)/*tentative definition*/
```

Figure 2.52

only identifier in scope is *A*. *A* identifies an expression token which is an rvalue whose type is a pointer to a pointer to a type token. The expression token and the type token are provided by the arguments at the call.

A more complex example is the definition of *st_sizeof* in Figure 2.50. It is based on *offsetof*, Figure 2.40,but has a nicer definition. Here, during the definition, the only identifiers in scope are *T* and *M*. *T* is an identifier for a compound type token. *M* identifies a token selector for that compound type, selecting a member which has whose type is a type token. The definitions of these tokens are, once again, provided by the parameters.

As a final example consider the definition of *SWAP* in Figure 2.51. The definition is straightforward enough, and so is the application of the procedure token. However, if this is not compiled in the presence of the procedure token introduction, which separates the definition from the use of the token, but is treated as a straightforward macro, the swap does not take place because the parameter *x* becomes a reference

```
# pragma token-op token-id-list_opt

token-id-list:
  TAG_opt identifier dot-list_opt token-id-list_opt

dot-list:
  . member-designator

token-op:
  define
  no_def
  ignore
  interface
```

Figure 2.53

to the variable in the inner scope. Furthermore, many conventional C compilers would not notice that there was anything wrong with this particular application (because there isn't!).

2.10.4.1 Implicit definition of function tokens

Whenever a function procedure token is introduced, it has a tentative implicit definition. The tentative definition is to define the procedure token as a direct call of the function, effectively removing the in-lining capability.

If at any point after the introduction of the function procedure token, there is a genuine definition, it overrides the tentative definition. An example of the tentative definition for *func* is given in Figure 2.52.

2.11 Token definition states

Associated with every token is a set of states. These states determine what the action of the compiler is in the presence/absence of a definition of a token. The states are as follows:

1. *Defined* - the token has a valid definition. Any further attempt to define the token will cause the compiler to exit with a constraint error. This prevents a token from being defined twice.
2. *Indefinable* - the token has not been defined and must not be defined. This prevents the definition of a token, any attempt to define the token will cause the compiler to exit with a constraint error. It is not possible to move from the state of *defined* to *indefinable*.
3. *Committed* - the token must be defined during the compilation of the program otherwise the compiler exits with a constraint error.
4. *Ignored* - any definition of the token that has been assigned during the compilation of the program will not be output as TDF.
5. *Free* - non of the above states apply, the token can be left defined or undefined. If it is defined then the definition will be output to TDF.

The purpose of these states is to enable programmers to instruct the compiler to check that tokens are defined when and only when they want them to be. This is critical in the separation of program into portable and unportable software. Programmers of applications do not want to accidentally define a token that is expressing an API. Implementors of APIs do not want to inadvertently fail to define a token expressing that API. At their point of introduction, tokens are in the *free* state.

There are a collection of pragma which control the state of a token. These pragmas have the syntax shown Figure 2.53. The syntax for *member-designator* is given in Figure 2.31.

```
# pragma implement interface header
# pragma extend interface header
```

Figure 2.54

The *token-op* is the operation to be applied to a token and the *token-id-list* is the list of tokens to which the operation is to be applied.

The tokens in the *token-id-list* are identified by the ***identifier***, which is optionally preceded by **TAG**. If **TAG** is present the identifier refers to the *tag* name space otherwise the *macro* and *ordinary* name spaces are searched for the identifier. If there is no *dot-list* present, the identifier must identify a token, otherwise it must identify a compound type. If there is a *dot-list* present, then the member-designator must identify a member selector token (Section 2.9) of that compound type.

2.11.1 The token operators

There are three literal token operators and one context dependent token operators.

The three literal token operators are:

1. The token operator *define*, which causes the state of a token to move to *committed*.
2. The token operator *no_def*, which causes the state of the token to move to *indefinable*.
3. The token operator *ignore*, which causes the state of the token to move to *ignored*.

The context dependent operator is *interface* which is designed specifically with APIs and interfaces in mind.

2.11.2 The interface operators

Interfaces are collections of tokens that represent the program constructs of APIs. For example *FILE* and *stderr* of Figure 2.1 can be thought of interface tokens for the ANSI C Standard *<stdio.h>* interface/API.

There are three uses of interface tokens introductions:

1. They are used to by applications that do not wish to assume any knowledge of the implementation of the interface/API. The tokens need to be in the *indefinable* state.
2. They are used to provide the defining implementations of the tokens forming the interface. The tokens need to be in the *committed* state.
3. They are used in the construction of the tokens of other interfaces. A typical example is that the *<stdio.h>* of POSIX (Ref [2]) is an extension of *<stdio.h>* of the ANSI C Standard and uses the same tokens to represent the common part of the interface. When using, within an application, the ANSI tokens of *<stdio.h>* via the POSIX API for <stdio.h>: the tokens must be in a state of *indefineability* since nothing can be assumed about the implementation of the POSIX or the ANSI implementations. However, when the POSIX tokens are being implemented, the ANSI implementations can be assumed and therefore the tokens can be accordingly defined, but, the implementation of the ANSI tokens must not output the definitions as this will already have taken place as part of the definition of the ANSI API: in this case the tokens are in an *ignored* state.

Associated with each file that is compiled by the compiler is one of three compilation states. The compilation state determines the interpretation of the *interface* token operator. The three compilation states are:

1. *Standard* - This is the initial compilation state and the *interface* operator is interpreted as the *no_def* operator. It is the standard state for the compilation of applications in the presences of APIs
2. *Implementation* - In this state the *interface* operator is interpreted as the define operator. It is the state for defining the components of an API.

3. *Extension* - In this state the *interface* operator is interpreted as the ignore operator and it is the state for the compilation of interfaces that are used as the basis for other interfaces.

The compilation state is changed by the two pragmas shown in Figure 2.54. These two pragmas behave in an identical fashion to

 # include *header*

but they potentially alter the compilation state of the file being included. The syntax class *header* corresponds to any valid set of pre-processing tokens after **# include**.

The inclusion of a file with **#include** causes the compilation state of the included file to be the same as the file from which it was included.

The inclusion of a file with **#pragma implement interface** causes the compilation state of the included file to be *implementation*.

The inclusion of a file with **#pragma extend interface** causes the compilation state of the included file to be:

1. *Standard* - if the file from which it was included was in the state *standard*.

2. *Extension* - if the file from which it was included was in the state *extension*.or *implementation*.

3 Portability Checks

char_bits	*decimal-integer*
short_bits	*decimal-integer*
int_bits	*decimal-integer*
long_bits	*decimal-integer*
signed_range	*range-type*
char_type	*sign-specification*
ptr_int	**none**
ptr_fn	**yes**
non_prototype_checks	**yes**
multibyte	**0**

Figure 3.1

3.1 Introduction

The internal portability checks (see Section 1.1.2) that have been introduced into the compiler are a collection of checks that are both convenient and thought useful to perform. The main thrust in the compiler development has been towards the performing of external portability checks (see Section 1.1.1), and the internal portability checks, described in this section, are in the process of being developed to complement the external checks.

The use of these checks is controlled by pragmas and a portability table.

The pragmas and portability table serve two purposes:

1. To describe the portability level required of the program, see Section 1.2.

2. To indicate which checks are to be performed.

3.2 The portability table

The portability table is the main mechanism for describing the portability level required of the program being compiled. It describes the minimum assumptions about the representation of the integer types. The portability table is passed as a parameter to the compiler, see Ref [4], and its format is shown in Figure 3.1.

The portability table format must be in precisely the format shown, with all the information provided, and provided in the order shown. The last four lines of the table, i.e. from **ptr_int** downwards, are no longer used by the compiler, and so, in order to complete the table, it is suggested that the default options shown in the table are used.

3.2.1 Integer sizes

The *decimal-integer* associated with **char_bits**, **short_bits**, **int_bits** and **long_bits** indicates that the compiler should check for portability to all targets whose **char**, **short**, **int** and **long** types are represented by at least the number of bits expressed in the *decimal-integer*. For example, if **int_bits** is set to *16*, the compiler will per-

```
char_bits              8
short_bits             16
int_bits               16
long_bits              32
signed_range           symmetric
char_type              either
ptr_int                none
ptr_fn                 yes
non_prototype_checks   yes
multibyte              0
```

Figure 3.2

```
char_bits              8
short_bits             16
int_bits               32
long_bits              32
signed_range           maximum
char_type              either
ptr_int                none
ptr_fn                 yes
non_prototype_checks   yes
multibyte              0
```

Figure 3.3

form its portability checks assuming that the **int** types might be represented by as a few as 16 bits. If **int_bits** is set to 32, then the compiler will perform its checks in the knowledge that the program would not be used on a machine whose **int** types were represented by 16 bits, although they may be represented by 32 or 64 bits.

3.2.2 Integer ranges

The minimum range of values that can be represented by an integer type is deduced from the minimum number of bits that type is specified to have.

In the case of the unsigned integer ranges, the minimum range is from 0 to (2^{b}-1), where *b* is the minimum number of bits for the integer type.

In the case of signed integer ranges, the upper limit of the minimum range is ($2^{(b\text{-}1)}$-1), but the lower limit depends on whether **signed_range** is set to **maximum** or **symmetric**. If it is set to **maximum**, the lower limit is (-$2^{(b\text{-}1)}$); if it is set to **symmetric**, then the lower limit is the negation of the upper limit.

The minimum range of the type **char**, which is not specified to be either signed or unsigned, is determined by **char_type**. **char_type** is either **signed**, **unsigned** or **either** indicating that the compiler should check for portability to targets where the type **char** is respectively signed, unsigned or of unknown signedness. If **char_type** is **signed** or **unsigned**, then the minimum assumed range of **char** is identical to that of **signed char** or **unsigned char** respectively, otherwise it is the intersection of these two ranges.

The portability table of Figure 3.2 reflects the minimal requirements of a target system as laid down in the ANSI C Standard Section 2.2.4.2.1. The portability table of Figure 3.3 reflects the implementation of most modern 32 bit machines

```
unsigned int f(){
  return 0x10001UL
  }
```

Figure 3.4

 # pragma no implicit conversion warning$_{opt}$

 # pragma no explicit conversion warning$_{opt}$

Figure 3.5

with 32-bit integer types and with maximal as opposed to symmetric ranges.

Consider the example in Figure 3.4. The answer returned by *f* is different depending on whether or not the program is compiled with 16 or 32 bit integers. If the program is compiled on a target with 16 bit integers, then the answer delivered is *0x1*; if it is compiled on a target with integers represented by more than 16 bits the answer delivered is *0x10001*. With the portability table of Figure 3.2, the answer is not consistent across the targets in the portability table, since it is different for say 16 and 32 bit machines. However, the program is consistent across the targets expressed by the portability table of Figure 3.3. If the appropriate portability checks are requested, see Section 3.3, then the program of Figure 3.4, will be flagged with the table of Figure 3.2, but not with the table of Figure 3.3.

3.3 Integer Conversions

The conversion from one integer type, say the *from-type*, to another, say the *to-type*, potentially depends on the representation sizes of the two types concerned. Therefore, integer conversion is, in general, target dependent. The compiler has an *integer-conversion check* which, when in operation, causes all integer conversions to be flagged, except in the following circumstances:

1. When all the values of the *from-type* are included in the *to-type*. This occurs only when the *from-type* is known to have the same signedness as the *to-type* and the *to-type* is known to be a larger range than the *from-type*.
2. When the value to be converted can be computed at compile time and is known, by reference to the portability table, to lie in the range of the *to-type*.
3. The *from-type* is ***char***.

Ignoring exception 3, if the integer-conversion check is in operation, all unflagged conversions are guaranteed to produce the same value on the targets characterised by the portability table.

Exception 3 is permitted purely because the conversion of values from type ***char*** are extremely frequent, mostly portable and would otherwise always be rejected by the *integer-conversion check* when its signedness is unknown.

Since, currently, it is not possible to express the signedness, or the relative ordering to other integer types, of integer type tokens (see Section 2.8.3), the *integer-conversion check* flags all conversions to or from an integer type token.

The *integer-conversion checks* are brought into operation by the use of the pragmas illustrated in Figure 3.5. The first pragma causes the checks to operate on all assignment conversions (ANSI C Standard, Section 3.3.16.1). The second pragma causes the checks to operate

```
# pragma integer literal literal-class lit-class-type-list

literal-class:
  denomination unsigned_opt long_opt

denomination:
  octal
  decimal
  hexadecimal
```

Figure 3.6

on all explicit casts. If the keyword **warning** is present, then any conversion flagged by the check produces a warning, otherwise it causes the compiler to exit with a constraint error.

The program in Figure 3.4 performs an assignment conversion on the operand of ***return*** from **unsigned long** to **unsigned int**. If the program is compiled in the presence of the first pragma of Figure 3.5, then:

1. If the portability table of Figure 3.2 is used then the compiler will flag the conversion because it does not belong to any of the three exceptional cases.

2. If the portability of Figure 3.3 is used then the compiler will not flag the conversion because exception 2 applies.

3.3.1 Conversions errors

In many of the cases where programmers use conversions that are potentially not portable, they tend to fall into three categories

1. Genuine mistake with a straightforward correction

2. Known by the programmer to be portable, even though the compiler cannot guarantee it.

3. Known by the programmer to be unportable, but it is too awkward or difficult to write a portable version, so a *quite* portable solution is adopted.

The compiler provides a handle on the last two cases with its user defined conversions (see Section 3.5). A user defined conversion enables the conversion from one type to another to be defined as a procedure token (see Section 2.10). Any user-defined conversion is accepted by the compiler as a portable conversion.

If the conversions of categories two and three are represented by user defined conversions, then the errors can be removed. In the case of category two: this has the effect of identifying all the conversions that the programmer believes to be portable. In the case of category three, the use of a procedure token enables the programmer to provide a different implementation of the conversion on each target.

In order to take advantage of these facilities, it is necessary to have a much more refined notion of integer types. For example, it is unlikely that a programmer would wish to convert all values of type **int** to type **short**, but, by using integral type tokens (see Section 2.8.3), an arbitrary number of distinguishable integer types can be introduced into a program.

3.4 Integer literals

The type of an integer literal depends on the value of the literal and the ranges of the integer types (ANSI C Standard, Section 3.1.3.2). Since the ranges of the integer types depend on the target for which the program is being compiled, the types of the integer literals, and consequently their semantics, is target dependent.

In order to tighten the semantics in this area another pragma has been introduced as in Fig-

```
lit-class-type-list:
  * int-type-spec
  integer-constant int-type-spec | lit-class-type-list

int-type-spec:
  : type-name
  * warning_opt : identifier
  * * :
```

Figure 3.7

```
#pragma integer literal decimal\
      0x7fff:int|0x7fffffff:long|*:unsigned long       /*line 2*/
```

Figure 3.8

ure 3.6. The purpose of this pragma is to explicitly define the method by which the type, and thereby the semantics, of an integer literal is computed.

The *literal-class* identifies the kind of literal integer for which the types are being defined. There are twelve different classes of integer literal in ANSI C. First integer literals are divided into octal (i.e. those that begin with **0**), decimal (i.e. those that begin with a non-zero digit) and hexadecimal (i.e. those that begin with **0x** or **0X**). Each of these major divisions is then subdivided into a further four divisions according to the presence or absence of the suffices **U** and **L**.

The major division of the literal integers is expressed by the *denomination* and the subdivision of that class is determined by the presence or absence of **unsigned** or **long** which correspond to **U** and **L** respectively.

If an integer literal is to be used in a program, then the method of computing its type must first have been introduced using this pragma.

The *lit-class-type-list*, Figure 3.7, is the specification of the type computation for the integer literal class identified by *literal-class*.

3.4.1 The type computation specification

The values of the integer literals of any particular class are divided into contiguous sub-ranges. The ***integer-constant***s define the upper limits of these sub-ranges. That is the first integer-constant, say *i1*, identifies the range [0,*i1*]; the second integer literal, say *i2*, identifies the range [*i1*+1,*i2*]. The symbol * identifies the unlimited range upwards from the last integer constant. Each integer constant must be strictly larger than the one that precedes it. Associated with each sub-range is an *int-type-spec* which is either a type, a procedure token (see Section 2.10), or a failure. The *int-type-spec* identifies the method by which integer literals, which lie in the sub-range with which it is associated, are computed.

If the *int-type-spec* associated with a sub-range is a ***type-name***, then the type which it specifies must be an integral type and this is the type associated with integers lying in the sub-range. For example, consider line 2 in Figure 3.8, which divides the unsuffixed decimal literals into three ranges. Integer literals in the range [0,0x7fff] are of type **int**, integer literals in the range [0x8000,0x7fffffff] are of type **long**, and the remainder are of type **unsigned long**.

If the *int-type-spec* associated with a sub-range is an ***identifier***, with an optional **warning**, then

```
#pragma token PROC (VARIETY)VARIETY l_i#~lit_int

#pragma integer literal decimal 0x7fff:int|**warning : l_i
```

Figure 3.9

```
#pragma token PROC (VARIETY)VARIETY l_i#~lit_int

#pragma integer literal decimal 0x7fff:int|0xffffffff*warning:l_i|***:
```

Figure 3.10

the type of the integer is computed by a procedure token (see Section 2.10). The procedure token takes the integer value as a parameter and delivers its type. Since the type of the integer is determined by a procedure token, which potentially can be implemented differently on different targets, there is the option of producing a warning whenever the token is applied. The procedure token must have been introduced previously with

```
#pragma token PROC(VARIETY)VARIETY
```

and the identifier in the *int-type-spec* is its local identification in the *macro* name space.

At present, within the language, It is not possible to define any such tokens taking integer parameters and delivering integral types. However, it is possible to provide definitions directly in TDF. Along with the TDF system there should be, in the token definition libraries, a set of predefined TDF token definitions that can be used to compute integer literal types. These definitions can then be TDF linked to the TDF output of the C compiler to provide the definitions of the tokens used in the type computation.

The TDF defined tokens for use in these type computations are defined differently on each target:

1. "~lit_int" is the external identification of a token that returns the integer type according to the rules of ANSI C Section 3.1.3.2 for unsuffixed decimal, taking into account the ranges of the integer types of the target on which the token is defined.
2. "~lit_hex" is the external identification of a token that returns the integer type according to the rules of ANSI C Section 3.1.3.2 for unsuffixed hexadecimal, taking into account the ranges of the integer types of the target on which the token is defined.
3. "~lit_unsigned" is the external identification of a token that returns the integer type according to the rules of ANSI C Section 3.1.3.2 for integers suffixed by **U** only, taking into account the ranges of the integer types of the target on which the token is defined.
4. "~lit_long" is the external identification of a token that returns the integer type according to the rules of ANSI C Section 3.1.3.2 for integers suffixed by **L** only, taking into account the ranges of the integer types of the target on which the token is defined.

An example of the use of a procedure token is given in Figure 3.9. This describes a specification for an integer literal as follows:

1. If it is less than or equal to 0x7fff, it has type **int**.
2. If it is greater than 0x7fff, then its type is computed by the token with external identification "~lit_int" and internal identification "l_i".
3. Since the definition of "~lit_int", bound to the output of the program being compiled, is potentially different on different targets, the compiler will output a warning every time it is called.

```
# pragma accept conversion conv-list_opt

conv-list:
  identifier conv-list_opt
```

Figure 3.11

```
#pragma token TYPE IP#
#pragma token PROC{TYPE t,EXP rvalue:*t:e|EXP e}EXP rvalue:IP:p_to_ip#
#pragma token PROC{VARIETY v,EXP rvalue:v:e|EXP e}EXP rvalue:IP:i_to_ip#
#pragma token PROC{TYPE t,EXP rvalue:IP:e|EXP e}EXP rvalue:*t:ip_to_p#
#pragma token PROC{VARIETY v,EXP rvalue:IP:e|EXP e}EXP rvalue:v:ip_to_i#

#pragma accept conversion p_to_ip i_to_ip ip_to_p ip_to i

void f(void){
  IP ip, *pip=&ip;short s=6;
  ip = s;           /*using i_to_ip*/
  s = ip;           /*using ip_to_i*/
  ip = pip;         /*using p_to_ip*/
  pip = ip;         /*using ip_to_p*/
  }
```

Figure 3.12

If the *int-type-spec* associated with a sub-range is ***:** then any integer literal lying in this range will cause the compiler to terminate with a constraint error. A typical use of this is where some of the envisaged targets for the program can not accept integer literals that are too large. For example the program in Figure 3.9 could be modified to that of Figure 3.10 to prevent the use of integer literals over 0xffffffff.

3.4.2 ANSI C Compliance

This pragma allows considerable freedom in the specification of the types of integer literals, so much so that it is possible to express rules for the computation of integer literal types that are not consistent with the ANSI C Standard Section 3.1.3.2. The compiler is usually called via tcc (Ref [4]) which calls a sequence of start up files consisting of a set of pragmas to give a specification of integer literal types which is consistent with the ANSI C standard.

3.5 User defined conversions

There is a pragma, whose syntax is illustrated in Figure 3.11, for replacing or adding conversion operators between values of different types.

Each identifier in the *conv-list* must be a local identifier for a procedure token (see Section 2.10) which has precisely one program parameter, which must be an expression. The procedure token must deliver an expression. Both the parameter and resulting expressions must be *lvalue*s.

When attempting the conversion of a value, either by assignment or by casting, from one type to another, if that conversion would not normally be permitted, because of the constraints of the language or portability checks, then for each token introduced as a conversion token:

1. An attempt to resolve the type of the token result to the type to which the value is being converted is made.

2. If the result is resolved, and the value to be converted is a suitable argument for the token procedure, the token procedure is applied to implement the conversion.

If no suitable conversion token can be found, then the compiler exist with a constraint error.

An example is shown in Figure 3.12. This introduces a token for a type whose intended implementation is a type that can represent all varieties on integer and all kinds of pointers. Ordinarily conversions to or from the token type would not be allowed, but because of the use of user-defined conversions, both integers and pointers can be converted to and from the token type.

4 C Variants

```
# pragma promote type-name : type-name

# pragma compute promote identifier
```

Figure 4.1

```
# pragma promote char : int
# pragma promote unsigned short : unsigned int
```

Figure 4.2

The C to TDF compiler is designed to be compliant to the ANSI C Standard (Ref [1]). In order to accommodate the many other dialects of C, pragmas have been introduced to allow the programmer to specify the use of facilities in other dialects. Usually these pragmas appear in *startup* files (Section 6.1) automatically supplied by *tcc* (Ref [4]) to make the compiler behave in an identical fashion to the dialect.

Some of the variants are pure extensions to the ANSI C Standard, for example allowing a more generous syntax (Section 4.10). Others provide alternative semantics for existing constructs and concepts of ANSI C, for example different rules for integer promotion (Section 4.1).

In addition to the facilities described in this section, there are, described in other sections, facilities to provide ANSI C extensions and semantic variants. They have not been included in this section because it is not their primary reason for being introduced. They can be found in Section 3.4.2.

4.1 Integer Promotions

There are two common algorithms for the computation of integer promotions (ANSI C Standard Section 3.2.1.1). The one adopted by ANSI C is often called *value preserving*, but a common alternative is *sign preserving*.

The *value preserving* promotions convert all objects of type **char** and **short**, both their signed and unsigned varieties, to **int**, unless **int** cannot represent all the values of the type being converted, in which case it is converted to **unsigned int**.

The *sign preserving* promotions convert the signed varieties of **char** and **short** to **int** and the unsigned varieties to **unsigned int**. The type **char** is converted to **int**.

The compiler allows programmers to specify precisely the promotions they will obtain from the promotion of an object of a particular type. This is done with the two pragmas shown in Figure 4.1. The first pragma provides a *literal* promotion and the second a *computed* promotion.

```
#pragma token VARIETY prom_t#

#pragma promote unsigned short : prom_t
```

Figure 4.3

```
#pragma token PROC(VARIETY)VARIETY comp_p#

#pragma promote char : int
#pragma compute promote comp_p
```

Figure 4.4

4.1.1 Literal Integer Promotion

Literal integer promotion is where the promoted type is provided directly. The first ***type-name*** is the type for which the promotion is being specified, it must be an integral type. The second ***type-name*** is its promoted type; it must also be an integral type.

An example of a literal promotions is shown in Figure 4.2. The second promotion is slightly controversial. If the promotion is intended to reflect the ANSI promotion rules, then the promoted type of **unsigned short** will depend on whether the type **int** can represent all the values of type **unsigned short** or not. Programmers are faced with a choice. They can:

1. Restrict the portability of the program to those machines where they know the promoted type of unsigned short.

2. Choose a promotion type for **unsigned short** and produce a program that does not conform the ANSI C Standard on some of the machines to which they port.

3. Represent the promotion of **unsigned short** by a integral type token (see Section 2.8.3) and define this token appropriately on each target to which the program is being ported.

4. Use computed promotions (see Section 4.1.2).

An example of option3 is shown in Figure 4.3

4.1.2 Computed Promotions

If a promotion is required of a value whose promotion has not been specified with **#pragma promote**, then the compiler will exit with a constraint error unless there is a **#pragma compute** directive. This directive allows the programmer to provide the identification of a procedure token (Section 2.10) for computing the promotion type, of any integral type. This token is then called whenever a promotion of a type, without a literal promotion, is required. The procedure token must be declared as

```
PROC(VARIETY)VARIETY
```

and an example is shown in Figure 4.4. The promotion of all types except for **char** are computed using **comp_p**.

Currently, it is not possible to define, within the C language, tokens for performing the integral promotions, although these can be defined directly in TDF. With the TDF system, with token definition libraries are bound the following two tokens for performing integral promotions:

1. “~promote” is the external identification of the token, defined on each target, to provide the *value preserving* promotions appropriate to that target.

2. “~sign_promote” is the external identification of the token, defined on each target, to provide the *sign preserving* promotions appropriate to that target.

 # pragma TAG ignore class warning$_{opt}$

Figure 4.5

```
#pragma TAG ignore class

struct s {int x,y;};

union s co-ord;
```

Figure 4.6

```
struct s {int x,y;};

struct s co-ord;
```

Figure 4.7

 # pragma accept extra type definitions warning$_{opt}$

Figure 4.8

4.2 Tag Name Classifications

Associated with identifiers in the *tag* name space (ANSI C Standard, Section 3.1.2.3) is a classification (**struct**, **union** or **enum**) of the type to which they refer. When any such identifier is used, it is preceded by its classification, which must be identical to the classification associated with the identifier (ANSI C Standard, Section 3.5.2.3). The inclusion of the pragma of Figure 4.5 instructs the compiler to ignore the classification at the use of the tag identifier and assume the classification associated with the identifier. For example, in the presence of the pragma, the program in Figure 4.6 is valid and semantically equivalent to the program in Figure 4.7.

If **warning** is present in the pragma, then whenever the classification of the use of a tag does not agree with that associated with its identifier, a warning message is reported.

4.3 Re-definition of types

The ANSI C Standard (Section 3.1.2.2) states that identifiers corresponding to type definitions have no linkage and consequently cannot be redefined in the same scope. The inclusion of the pragma of Figure 4.8 allow the redefinition of a type, with an optional warning, provided that the new definition is compatible with the old definition. The type associated with the identifier becomes the composite type (ANSI C Standard, Section 3.1.2.6) of the old and the new definitions.

An example of such a re-definition is given by the program in Figure 4.9, which is semantically equivalent to the program in Figure 4.10.

```
#pragma accept extra type definitions

typedef int t[];
typedef int t[6];

t z = {1,2,3,4,5,6};
```

Figure 4.9

```
typedef int t[6];

t z = {1,2,3,4,5,6};
```

Figure 4.10

 # pragma accept char * as void * warning$_{opt}$

Figure 4.11

 # pragma accept argument *type-name* as *type-name*
 # pragma accept argument *type-name* ...
 # pragma accept extra ...

Figure 4.12

4.4 Generic pointers

Historically **char** * was used as the generic pointer of C, but more recently this has been replaced by **void ***. There are some old applications which, because they interact with newer APIs, require the compatibility of **char** * and **void** *. The inclusion of the pragma of Figure 4.11 instructs the compiler to treat **char** * and **void** * as compatible, with the composite type being **void** *. If the pragma includes **warning**, then a warning is produced whenever the compatibility of **char** * and **void** * is required.

4.5 Parameter Compatibility

Many targets/systems pass function arguments, of differing types, in the same way. For example on the majority of systems characters are passed as if they were integers. Some programs take advantage of this. The three pragmas introduced in Figure 4.12 are to specify extra compatibility of function types by providing extra compatibility for the function parameters. They are referred to as *type-type parameter compatibility*, *type-ellipsis compatibility* and *ellipsis compatibility* respectively.

```
#pragma accept argument char as short
#pragma accept argument short as int
#pragma accept argument char * ...

int f1(char,...);
int f1(int,char*,...);

#pragma accept extra ellipsis

int f2(short);
int f2(char,...);
```

Figure 4.13

```
int f1(int,...);
int f2(short,...);
```

Figure 4.14

4.5.1 Type-type parameter compatibility

When comparing two parameter types for compatibility, as part of checking for the compatibility of two function prototypes (ANSI C Standard, Section 3.1.2.6), if the parameter types would otherwise be incompatible, they are treated as compatible if they have previously been introduced with a *type-type parameter compatibility*. In this case, the second type in the *type-type parameter compatibility* pragma is taken as the composite type.

The *type-type parameter compatibility* is transitive, that is if type *A* is compatible with type *B* and type *B* is compatible with type *C*, then type *A* is compatible with type *C*.

4.5.2 Type-ellipsis compatibility

When comparing two function prototypes for compatibility, if the prototypes have different numbers of arguments, they are compatible if the following holds:

1. Both prototypes have an ellipsis

2. Each parameter type common to both prototypes is compatible.

3. Each extra parameter type in the prototype with more parameters, is either specified in a *type-ellipsis compatibility* or is *type-type parameter compatible* to a type that is specified in a *type-ellipsis compatibility*.

The composite type of the two prototypes is the prototype with the fewer arguments.

4.5.3 Ellipsis compatibility

If, when comparing two prototypes for compatibility, one has an ellipsis and the other does not, but otherwise the two types would be compatible, then if *ellipsis compatibility* has been introduced by its pragma (the third pragma of Figure 4.12), the two types are compatible.

The composite type has an ellipsis.

4.5.4 Examples

Figure 4.13 is an example of the use of parameter compatibility, and Figure 4.14 is a program with equivalent semantics.

```
void f(void){
  {  extern void g();                                   /*line 2*/
     g(3);
     }                                                  /*line 4*/
  g(7);                                                 /*line 5*/
  }
```

Figure 4.15

 # pragma warning$_{opt}$ **forward enum declarations**

Figure 4.16

4.6 External Name Space

Consider the program in Figure 4.15, this declares, on line 2, a function *g* with external linkage. The scope of *g* terminates at line 4, and so, according to ANSI C Standard, Section 3.3.2.2, *g* is implicitly declared as:

```
extern int g();
```

This re-introduces *g* as a function with external linkage but now as a function delivering **int** instead of **void**. This is clearly an error because both declarations must relate to the same function because they have external linkage (ANSI C Standard, Section 3.1.2.2).

If the pragma

```
#pragma name space backdrop
```

is included in a program, then after that point in the program, before implicitly declaring a function, it is searched for in the external name space. If it is found, the type in the external name space is assumed.

In the example of Figure 4.15, the second use of *g*, on line 5, would assume the type **void ()** as opposed to **int()**.

4.7 External Volatility

The inclusion of the pragma

```
# pragma external volatile
```

instructs the compiler, thereafter, to treat any object declared with external linkage (ANSI C Standard, Section 3.1.2.2) as if it were volatile (ANSI C Standard, Section 3.5.3).

4.8 Enumeration Constants

The ANSI C Standard, Section 3.5.2.3, requires that the first introduction of an enumeration tag shall declare the constants associated with that tag. This can be relaxed by using the pragma of Figure 4.16 which, when included, permits the declaration and use of an enumeration tag before the declaration of its associated enumeration constants. If **warning** occurs in the pragma, then a warning message is output whenever this facility is used.

4.9 Lvalue Conditionals

The ANSI C Standard, Section 3.2.2.1 requires that the *lvalue* conversion is applied to the operands of most operators including the conditional

 # pragma accept conditional lvalue warning$_{opt}$

Figure 4.17

```
#pragma accept conditional lvalue

struct s {int x,y;};
int f(int c, int sx, struct s *s1, struct s *s2){
  return (c?s1:s2)->x = sx;
  }
```

Figure 4.18

# pragma accept no definitions	/*Section 4.10.1*/
# pragma accept extra ;	/*Section 4.10.2*/
# pragma accept unknown escapes warning$_{opt}$	/*Section 4.10.3*/
# pragma accept weak equal macro warning$_{opt}$	/*Section 4.10.4*/
# pragma accept implicit integer type warning$_{opt}$	/*Section 4.10.5*/

Figure 4.19

operator (ANSI C Standard, Section 3.3.15). From this it follows that the expression resulting from a conditional cannot be an *lvalue*.

There is no difficulty in implementing a conditional that delivers an *lvalue* when both options of the conditional are *lvalue*s of objects of compatible types. However, because of the possible conversions that take place to match the operands of a conditional it is only sensible to return an *lvalue* when the *lvalue* conversion is the only conversion that can be applied. For this reason, the pragma of Figure 4.17 instructs the compiler to deliver an lvalue from a conditional when:

1. Both options of the conditional operator have compatible compound types (see Section 2.8.6).

2. Both options of the conditional are *lvalues*.

in which case the *lvalue* delivered has the composite type of the two options. If **warning** is include in the pragma, then a warning is output whenever this facility is used by the programmer.

An example of the use of this pragma is shown in Figure 4.18.

4.10 Syntactic Extensions

There are a collection of pragmas (Figure 4.19) for allowing non-semantic extensions to the C language. They are described in the following sub-sections.

4.10.1 No external declarations

The inclusion of this pragma instructs the compiler to allow a program with no external declarations/definitions (ANSI C Standard, Section 3.7), that is a program without a ***translation-unit***.

4.10.2 Extra Semi-colons

The inclusion of this pragma allows the inclusion of extra semi-colons before and after the external declarations and definitions that form the ***translation-unit*** (ANSI C Standard, Section 3.7).

```
# pragma weak equal macro

#define A(x)x+2

#define A(y)y+2
```

Figure 4.20

```
# pragma unknown pragma warning_opt                        /*Section 4.11.1*/
# pragma accept directive ignore_opt standard-dir          /*Section 4.11.2*/

standard-dir:
  file
  ident
  assert
  unassert
```

Figure 4.21

4.10.3 Unknown Escape Sequences

The inclusion of this pragma instructs the compiler to allow escape sequences other than those specified in ANSI C Standard, Section 3.1.3.4. In this case the program is compiled as if the '\', marking the start of the escape sequence, was not present. If **warning** is present in the pragma, then a warning message is output whenever this pragma is used.

4.10.4 Macro Equality

The ANSI C Standard, Section 3.8.3, states that: for *function-like* macros to be equal, the spelling of the parameters and the replacement lists must be equal. Since the names of the parameters to *function-like* macros are incidental to their semantics, the inclusion of this pragma allows (with an optional warning) the equality of two function-like macros provided there is a consistent substitution of the parameters of the one macro to obtain the other.

An example of a valid function-like macro redefinition is given in Figure 4.20.

4.10.5 Implicit integer type

External-declarations (ANSI C Standard, Section 3.7) are either ***function-definition***s or ***declaration***s. ***Function-definition***s do not require the presence of ***declaration-specifiers*** whereas ***declaration***s do. The inclusion of this pragma relieves the need for the *declaration-specifiers* (with an optional warning) in the case of *declaration*s, consistently assuming that they consist only of **int**.

4.11 Directive Control

The pragmas of are for controlling the use of pre-processing directives.

4.11.1 Pragma Control

Except for those pragmas described in this document, all pragmas cause the compiler to exit with a constraint error. This pragma instructs the compiler to accept, but ignore (with an optional warning) any pragmas that it does not recognise.

4.11.2 Extra directives

There are four pre-processing (**file**, **ident**, **assert**, **unassert**) directives that are used by programs that are designed to be compiled with C compilers meeting the specification of Ref [5]. These directives are accepted by the compiler by using the second pragma of Figure 4.21. If

ignore is present in the pragma, then the pre-processing directive, to which it refers, is ignored, otherwise it has the semantics expressed in Ref [5].

The compiler also accepts two directives, **#pragma weak** and **#pragma ident** which implement the semantics expressed in Ref [5].

5 Correctness checking

```
int f();

int g(void){
  return f(6);                                        /*line 4*/
  }
```

Figure 5.1

```
int f(i)int i;{return i+1;}
```

Figure 5.2

```
int f(i)long i;{return i+1;}
```

Figure 5.3

The compiler has the capability of providing some checks to help in the detection of common errors made in programming in C. These can be thought of as similar to Lint checks, but rather than being performed separately from the compilation process, they are performed during compilation.

5.1 Non-prototype checking

There is still considerable use of functions defined without prototypes. One of the major reasons being the need to compile an application with as many compilers as possible, some of which do not readily support prototypes.

The major problem, from a correctness point of view, of the absence of prototypes is that the function declarations provide no information about the types of the arguments. For example, consider the program in Figure 5.1. The call of **f** on line 4 is undefined if **f** is defined as in Figure 5.3 (because 6 is passed through as an **int** not a **long**), but well defined if it is defined as in Figure 5.2 (ANSI C Standard, Section 3.3.2.2).

On a system where **int** and **long** have the same representation and *f* is defined as in Figure 5.3, there are no difficulties. However, if this program were compiled on a machine with 32 bit integers and 64 bit longs then there is no guarantee as to what will be the effect of the call.

In order to get a handle on this problem, the language has been extended to accept a new type constructor to represent the type of a function defined without a prototype. This type constructor is called the *weak prototype constructor.*

5.1.1 The weak prototype constructor

The *weak prototype constructor* is a keyword which is introduced into the language as the

```
# pragma accept weak prototype identifier
```

Figure 5.4

```
# pragma accept weak prototype _WEAK                /*line 1*/

int f _WEAK(long);                                   /*line 3*/

int g(void){
  return f(6);                                       /*line 6*/
  }
```

Figure 5.5

```
#pragma accept weak prototype _WEAK

int f _WEAK(short);

int g(int a){ return f(a);}
```

Figure 5.6

identifier of the pragma of Figure 5.4. This constructor can be placed immediately before the opening round bracket of a parameter type list of a function prototype. The inclusion of the weak prototype constructor in a function prototype is to introduce a type that describes functions defined without prototypes and whose parameters have the types given by the parameter type list.

The program in Figure 5.5 is an example of the use of a weak prototype constructor. The keyword **_WEAK** is introduced on line 1. Then, the declaration of a function **f** occurs on line 3. This function is defined without a prototype and has one parameter of type **long**, that is declaration corresponds to the one defined in Figure 5.3.

5.1.2 Argument checking

If, in a function call, the function has a weak prototype type, then the compiler is in a position to check the correctness of the arguments as follows:

1. The default argument promotions (ANSI C Standard, Section 3.3.2.2) are applied to the argument. The default argument promotions are applied to the corresponding parameter type in the weak prototype. If the promoted parameter type is not compatible with the promoted argument type the compiler exits with a constraint error.
2. The promoted argument is passed as a parameter to the function being called.

The call of **f** on line 6 is illegal because the argument promotion of the argument is **int** and the argument promotion of the parameter is **long** which are incompatible.

5.1.3 Weak Argument conversion

Consider the program in Figure 5.6. In terms of the function call, it is well defined. The promoted type of the argument is **int**, as is the promoted type of the parameter. However, there is an implicit conversion that takes place within the function body to convert the argument back to **short**. If the program is being compiled with

 # pragma no implicit definitions warning$_{opt}$

Figure 5.7

implicit conversion checks this program will flag an error because it is not an allowed conversion under Section 3.3. The rules for the detection of implicit conversions during argument passing are as follows: If

1. The program is being compiled in the presence of implicit conversion checks.
2. The un-promoted argument would, under conversion to the un-promoted weak parameter type cause a conversion check to flag an error.

then the passing of the argument to the function will cause an implicit conversion error to be flagged.

If the function **g** in Figure 5.6 had been defined to take a **short** or **char** parameter, then there would be no implicit conversion error because both **short** and **char** can be converted to **short**. Note that, in such cases, the argument promotion to the type int and its subsequent conversion back to **short** does not affect the portability or the well-definedness of the program.

5.1.4 The type of a function definition

Unless instructed otherwise, by a pragma, the type of a function defined without a prototype is the weak function prototype corresponding to its definition. This enables all the non-prototype checks to operate in the remainder of the program. For example, the function **f** in Figure 5.3 behaves as though it were declared with the weak prototype type of line 3 on Figure 5.5. If the programmer wishes to disable this facility then there is a pragma

```
#pragma no extra prototype checks
```

which causes subsequent function definitions without prototypes to have non-prototype types. That is **f** would subsequently behave as if it were declared

```
int f();
```

This action can be reversed by using the pragma

```
#pragma extra prototype checks
```

5.1.5 Function type compatibility

A weak prototype function type is incompatible with all function prototype types, but in all other respects the compatibility of weak prototype function types is identical to that of prototype function types.

5.2 Implicit declarations

The ANSI C Standard (Section 3.3.2.2) instructs compilers to implicitly declare out of scope identifiers that are used to identify functions in a function call.

This is a common source of error, often occurring when the appropriate header has not been included and frequently leading to the wrong declaration of the function with its inherent problems.

The compiler can be instructed to detect all such implicit declarations and then either produce a warning or exit with a constraint error. The syntax for the appropriate pragma is given in Figure 5.7.

This pragma is vital for proper API checking.

6 Implementation Defined Behaviour

```
void f(void){
  extern int x;
  }

static int x;
```

Figure 6.1

 # pragma warning$_{opt}$ **resolve linkage internal**

Figure 6.2

 # pragma preserve *static-list*

static-list:
  *****
  static-id-list$_{opt}$

static-id-list:
  identifier *static-id-list*$_{opt}$

Figure 6.3

Many aspects of the C language are described as having implementation defined behaviour or undefined behaviour (ANSI C Standard, Section 1.6).

Many of the other sections in this document describe properties of the implementation of this compiler, thereby explaining how the compiler behaves with respect to implementation or undefined behaviour in particular areas. This section provides details of pragmas that are explicitly designed to allow the programmer to control implementation defined / undefined behaviour.

6.1 Linkage Resolution

The ANSI C Standard, Section 3.1.2.2, states that if an identifier has both internal and external linkage, its behaviour is undefined. An example of such an identifier is *x* of Figure 6.1. Ordinarily the compiler would make *x* have external linkage, but in the presence of the pragma of Figure 6.2, the compiler (with an optional warning) makes *x* have internal linkage.

6.2 Static identifications

In the absence of instructions to do otherwise, the compiler does not output any information about the identification of static objects and functions. Since they are not required for linking there is no need of their output. However, there are some programs for example C profiling programs and C front which require the identification of some or all of the static identifiers.

The pragma of Figure 6.3 instructs the compiler to output the static identification of some or all of the static identifiers.

If the *static-list* is *, then the compiler will output the identification of all identifiers with static linkage introduced after the pragma. If the *static-list* is a *static-id-list* then:

1. Every identifier in the *id-list* must refer to an object or function with internal linkage.

2. The identification of every object in the *id-list* is output by the compiler.

7 References

[1] Draft Proposed American National Standard for Information Systems - Programming Language C, ANSI, X3J11/88-002, May 13, 1988.

[2] POSIX, IEEE Standard, Portable Operating System Interface for Computer Environments, IEEE 1003.1-1988.

[3] TDF Release Notes, DRA Document, June 1993

[4] tcc User's Guide, DRA Document, June 1993.

[5] C Language Specification for C Issue 5.0, USL Document, August 16, 1988.

Appendix A: Pragma Syntax

A.1 Basic pragma syntax: # pragma *pragma-syntax*

pragma-syntax:
    token-syntax
    definition-syntax
    token-control
    portability-checks
    c-variants
    correctness-checks
    implementation-control

A.2 Token syntax

(2.3) *token-syntax:*
    token *token-introduction token-identification*

(2.4) *token-identification:*
    name-space$_{opt}$ ***identifier* #** *ext-identifier*

(2.4.1.1) *name-space:*
    TAG

(2.5) *token-introduction:*
    exp-token
    statement-token
    type-token
    selector-token
    procedure-token

A.2.1 Expression token syntax

(2.6) *exp-token:*
  EXP *exp-storage* **:** ***type-name*** **:**
  NAT

(2.6) *exp-storage:*
  rvalue
  lvalue

A.2.2 Statement token syntax

(2.7) *statement-token:*
  STATEMENT

A.2.3 Type token syntax

(2.8) *type-token:*
  TYPE
  VARIETY
  ARITHMETIC
  STRUCT
  UNION

A.2.4 Selector token syntax

(2.9) *selector-token:*
  MEMBER ***type-name*** **:** ***type-name*** **:**

A.2.5 Procedure token syntax

(2.10) *procedure-token:*
  general-procedure
  simple-procedure
  function-procedure

(2.10.1) *general-procedure:*
  PROC **{** *bound-toks*$_{opt}$ **|** *prog-pars*$_{opt}$ **}** *token-introduction*

(2.10.1.1) *bound-toks:*
  bound-token
  bound-token **,** *bound-toks*

(2.10.1.1) *bound-token:*
  token-introduction name-space$_{opt}$ ***identifier***

(2.10.1.2) *prog-pars:*
program-parameter
program-parameter , prog-pars

(2.10.1.2) *program-parameter:*
EXP *identifier*
STATEMENT *identifier*
TYPE *type-name*
MEMBER *type-name* : *identifier*
PROC *identifier*

(2.10.2) *simple-procedure:*
PROC (*simp_tok_pars*$_{opt}$ **)** *token-introduction*

(2.10.2) *simp_tok_pars:*
simple-token
simple-token , simp_tok_pars

(2.10.2) *simple-token:*
token-introduction name-space$_{opt}$ ***identifier***$_{opt}$

(2.10.3) *function-procedure:*
FUNC *type-name* :

A.3 Definition Syntax

definition-syntax:
mem-selector-defn

A.3.1 Member selector definition syntax

(2.9.4.2) *mem-selector-defn:*
DEFINE MEMBER *type-name identifier* : *member-designator*

(2.9.4.2) *member-designator:*
identifier
identifier . *member-designator*

A.4 Token control

token-control:
  token-operation
  interface-control

(2.11) *token-operation:*
  token-op *token-id-list*$_{opt}$

(2.11) *token-id-list:*
  TAG$_{opt}$ ***identifier*** *dot-list*$_{opt}$ *token-id-list*$_{opt}$

(2.11) *dot-list:*
  . *member-designator*

(2.11.1) *token-op:*
  define
  no_def
  ignore
  interface

(2.11.2) *interface-control:*
  implement interface *header*
  extend interface *header*

A.5 Syntax for Portability Checks

portability-checks:
  conversion-checks
  integer-literal-specification
  user-defined-conversion

A.5.1 Conversion checks

(3.3) *conversion-checks:*
  no implicit conversion warning$_{opt}$
  no explicit conversion warning$_{opt}$

A.5.2 Integer literal specifications

(3.4) *integer-literal-specification:*
  integer literal *literal-class* *lit-class-type-list*

(3.4) *literal-class:*
  denomination **unsigned**$_{opt}$ **long**$_{opt}$

(3.4) *denomination:*
  octal
  decimal
  hexadecimal

(3.4.1) *lit-class-type-list:*
  ***** *int-type-spec*
  integer-constant *int-type-spec* **|** *lit-class-type-list*

(3.4.1) *int-type-spec:*
  : ***type-name***
  *** warning**$_{opt}$ **:** ***identifier***
  *** * :**

(3.5) *user-defined-conversion:*
  accept conversion *conv-list*$_{opt}$

(3.5) *conv-list:*
  identifier *conv-list*$_{opt}$

A.6 Syntax for C Variants

c-variants:
  promotion-types
  tag-classes
  type-redefinition
  generic-pointers
  parameter-compatibility
  external-name-space
  object-volatility
  enum-tag-constants
  conditional-lvalues
  syntactic-extensions
  directive-control

(4.1) *promotion-types:*
  promote ***type-name*** **:** ***type-name***
  compute promote ***identifier***

(4.2) *tag-classes:*
    TAG ignore class warning$_{opt}$

(4.3) *type-redefinition:*
    accept extra type definitions warning$_{opt}$

(4.4) *generic-pointers:*
    accept char * as void * warning$_{opt}$

(4.5) *parameter-compatibility:*
    accept argument *type-name* as *type-name*
    accept argument *type-name* as ...
    accept extra ...

(4.6) *external-name-space:*
    name space backdrop

(4.7) *object-volatility:*
    external volatile

(4.8) *enum-tag-constants:*
    warning$_{opt}$ **forward enum declarations**

(4.9) *conditional-lvalues:*
    accept conditional lvalue warning$_{opt}$

(4.10) *syntactic-extensions:*
    accept no definitions
    accept extra ;
    accept unknown escapes warning$_{opt}$
    accept weak equal macro warning$_{opt}$
    accept implicit integer type warning$_{opt}$

(4.11) *directive-control:*
    unknown pragma warning$_{opt}$
    accept directive ignore$_{opt}$ *standard-dir*

(4.11.2) *standard-dir:*
    file
    ident
    assert
    unassert

A.7 Correctness checks

correctness-checks:
    non-prototype-checks
    declaration-checks

(5.1) *non-prototype-checks:*
    accept weak prototype *identifier*
    extra prototype checks
    no extra prototype checks

(5.2) *declaration-checks:*
    no implicit definitions warning$_{opt}$

A.8 Implementation Control

implementation-control:
    linkage-resolution
    static-identification

(6.1) *linkage-resolution:*
    warning$_{opt}$ **resolve linkage internal**

(6.2) *static-identification:*
    preserve *static-list*

(6.2) *static-list:*
    *****
    static-id-list$_{opt}$

(6.2) *static-id-list:*
    identifier *static-id-list*$_{opt}$