View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

An algebraic specification for the static semantics of Pascal

A. van Deursen

Computer Science/Department of Software Technology

CS-R9129 1991

An Algebraic Specification for the Static Semantics of Pascal

A. van Deursen

CWI, Department of Software Technology,

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, email: *arie@cwi.nl*

Abstract

Over the last few years, several formal specifications for the static semantics of Pascal have been given. Thus far, however, no *algebraic* specification has been published. In this document we discuss an algebraic specification we have made for the complete static semantics of Pascal as defined by the International Standardization Organization (ISO). We explain how the specification has been set up, and how several details have been dealt with in a convenient way. Finally, we relate the specification to algebraic specifications for other aspects of (other) programming languages, and we briefly compare the algebraic approach with the other formal specifications for Pascal.

Key Words & Phrases: Formal specifications, algebraic specifications, Pascal, programming language semantics, static semantics, type checking.

1991 CR Categories: D.2.1 [Software engineering]: Requirements/specifications - Languages D.3.1 [Programming languages]: Formal definitions and theory - syntax, semantics F.3.2 [Logics and meanings of programs]: Semantics of programming languages - Algebraic approaches to semantics

1991 Mathematics Subject Classification: 68N15 [Software]: Programming languages; 68Q42 [Theory of Computing]: Rewriting Systems; 68Q55 [Theory of Computing]: Semantics; 68Q65 [Theory of Computing]: Algebraic Specification;

Note: Partial support has been received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators.*

Contents

1 Introduction 3

2 Algebraic Specifications 3
- 2.1 The Signature 4
- 2.2 Terms 4
- 2.3 Equations 4
- 2.4 Term Rewriting Systems 5
- 2.5 The Initial Algebra 5
- 2.6 Conditions and Negations . . . 6
- 2.7 Modularization 6
- 2.8 Syntactic Freedom 6
- 2.9 Signatures as Grammars 7

3 ASF+SDF 7
- 3.1 Lexical Syntax 7
- 3.2 List functions 8
- 3.3 Disambiguation 8
- 3.4 Sections 8

4 Pascal 9

5 Pascal in ASF+SDF 10
- 5.1 Points of departure 10
- 5.2 Overview 10

6 Basic Structures 10
- 6.1 Syntax 10
- 6.2 Contexts 11
- 6.3 Type Representation 11

7 Constraints 13
- 7.1 Error Messages 13
- 7.2 Communication 13
- 7.3 Constraints 13
- 7.4 An Example 14

8 Type check functions 14
- 8.1 Declarations 14
- 8.2 Expressions 16
- 8.3 Statements 18

9 Related Work 18

10 Concluding Remarks 19

References 22

A Import Graphs 23
- A.1 Syntax 23
- A.2 Miscellaneous 23
- A.3 Contexts 23
- A.4 Environments 24
- A.5 Type checking 24

B Module Descriptions 25
- B.1 Syntax 25
- B.2 Miscellaneous 25
- B.3 Contexts 25
- B.4 Environments 26
- B.5 Type checking 26

C Text of the Specification 27
- C.1 Syntax 27
- C.2 Miscellaneous 32
- C.3 Contexts 37
- C.4 Environments 50
- C.5 Type Checking 60

D An Example 92

1 Introduction

A formal specification [LB79, Win90] of a programming language precisely describes the syntax and semantics of the language in question. The syntax of a language can easily be specified, but describing the semantics precisely may prove more difficult. Formal language specifications are important for the design and description of new programming languages, and for the automatic generation of tools (parsers, compilers).

Formal specifications for various languages have been published, including several specifications for Pascal [HW73, BJ82, KHZ82, Duk87], but no *algebraic* specification for Pascal has been presented. We have made an algebraic specification for the static semantics of Pascal, in accordance with the definition of the International Standardization Organization [ISO83]. We discuss the specification, show how it has been set up, explain how difficult details have been handled conveniently, and describe the measures taken to keep the specification simple and easy to read, write, and adapt. We conclude the paper with a brief comparison of the Pascal specification with other algebraic language specifications, and with Pascal specifications in other formalisms.

We assume the reader is familiar with formal specifications, and is interested in language specifications. We introduce algebraic specifications in Section 2. The language used in this document, the ASF+SDF formalism, is explained in Section 3. Some features of Pascal are covered in Section 4.

The formal specification for the static semantics of Pascal is presented in detail in Sections 5, 6, 7 and 8. Finally, related work is discussed in Section 9, and some conclusions are drawn in Section 10. Detailed descriptions of each module are given in the appendices, just as the complete text of the specification. We conclude the paper with a demonstration how a simple Pascal program is type checked.

We have constructed the specification because of our interest in programming environments. We aim at generating a programming environment from an algebraic specification for the relevant aspects of the programming language. With the generator built so far [Kli90, Hen91] it is possible to generate, in an incremental way, a parser, syntax-directed editor, and type checker for Pascal from the specification presented in this document.

2 Algebraic Specifications

Algebraic specifications can be classified by the fact that the object to be specified is an *algebra* [GTW78]. An algebra is nothing more than a set of values (called the *carrier* set), and a set of operations (or *functions*) over these values. As an example, the algebra of integers has all integers as its values, and operations like $+$ and $-$ as its functions.

More complex algebras may need several sorts of values; for instance an algebra for a “stack” requires values of sort `STACK`, sort `INTEGER`, and perhaps of sort `BOOLEAN`. We therefore introduce a *many-sorted* algebra which has several *carriers*, and a unique name (the *sort*) for each carrier. Note that sorts in algebras are similar to types in programming languages.

The problem is, of course, how to describe the algebra we need. Algebras may be infinite objects (infinitely many integers), so we need some compact notation. We will present such a notation using equations, and we will call a description in such a notation an *algebraic specification.*

An algebraic specification consists of a *signature* and a set of *equations*.

2.1 The Signature

In the *signature* a list of the sorts is given, and the functions involved are declared.

For example, we may have a signature declaring the sort `BOOLEAN`[1] and declaring a function `and` over this sort, having two arguments of sort `BOOLEAN` and returning a `BOOLEAN` value:

```
sorts
  BOOLEAN
functions
  and: BOOLEAN # BOOLEAN -> BOOLEAN
```

By considering constant values as functions without arguments, we can introduce values `T` and `F`:

```
sorts
  BOOLEAN
functions
  T:                        -> BOOLEAN
  F:                        -> BOOLEAN
  and: BOOLEAN # BOOLEAN -> BOOLEAN
```

We may have different sorts, and the functions may have arguments of different sorts. As an example, consider the signature for a `STACK`, containing declarations for one constant, the `empty-stack`, and a function `top`, returning the top element, a function `pop`, removing the top element, and a function `push` to add an element.

```
sorts
  ELEMENT STACK
functions
  empty-stack:           -> STACK
  top: STACK             -> ELEMENT
  pop: STACK             -> STACK
  push: STACK # ELEMENT -> STACK
```

2.2 Terms

If we want to "use" a function, we will have to write something like `and(T,F)`, or `and(T,and(F,T))`. We will call such usages *closed terms*. In general, infinitely many closed terms can be constructed over a signature, e.g., over the Booleans we can construct: `and(T,F)`, `and(T, and(T,F))`, `and(and(T,F),and(F,T))`, etc.

If we allow the declaration of variables, we can construct even more terms:

```
sorts
  BOOLEAN
functions
  T:                        -> BOOLEAN
  F:                        -> BOOLEAN
  and: BOOLEAN # BOOLEAN -> BOOLEAN
variables
  p,q:                      -> BOOLEAN
```

Terms with variables, like `and(p,q)`, are called *open terms*. Open terms and closed terms together are called just *terms*.

2.3 Equations

From a signature we can construct terms, but the meaning of the functions is not yet clear. How do we know, for instance, that the result of `and(T,F)` should be equal to `F` (which is of course the behavior of `and` we expect)?

To that end we introduce *equations*:

```
sorts
  BOOLEAN
functions
  T:                        -> BOOLEAN
  F:                        -> BOOLEAN
  and: BOOLEAN # BOOLEAN -> BOOLEAN
variables
  p,q:                      -> BOOLEAN
equations:
  [1]  and(F, p) = F
  [2]  and(T, p) = p
  [3]  and(p, q) = and(q, p)
```

An equation relates two terms over a signature. Equation `[1]` states that if one of the arguments of the `and` function is `F`, the entire term equals `F`. Equation `[2]` states that if one argument is `T`, the result is equal to the value of

[1]We usually will CAPITALIZE sort names

the other argument. Equation `[3]` states that the `and` operator is commutative.

By substituting closed terms for the variables, we may "compute" complex closed terms. Consider for instance the term `and(T, and(F,T))`. By equation `[2]` this is equal to `and(F,T)`, which in turn is equal to `F` by equation `[1]`.

2.4 Term Rewriting Systems

It would be nice if there were some mechanical way to "compute" complex closed terms. The previous example already indicates that such an automatic way may be hard to find. Especially equation `[3] and(p, q) = and(q, p)` causes danger, because it can be applied infinitely many times. Indeed, it can be proved that the question how to "compute" terms is undecidable; There is no mechanical way to test whether two terms are equal.

There is a more restricted alternative, which nevertheless is attractive. We can view each equation as a *rewrite rule* explaining how a term can be rewritten into another term. We "direct" equations from left to right, and take care that we rewrite left-hand sides to "smaller" right-hand sides. This automatically excludes infinite loops like those caused by equation `[3] and(p, q) = and(q, p)`. If we take these precautions, we can mechanically reduce closed terms to smaller ones by applying rewrite rules.

```
sorts
  BOOLEAN
functions
  T:                         -> BOOLEAN
  F:                         -> BOOLEAN
  and: BOOLEAN # BOOLEAN -> BOOLEAN
variables
  p,q:                       -> BOOLEAN
equations:
  [1]  and(F, p) = F
  [2]  and(T, p) = p
  [3]  or(T, p)  = T
  [4]  or(F, p)  = p
```

In this example the term `and(or(T,F),F)` can be reduced by applying rewrite rule `[3]` and `[2]`: `and(or(T,F),F)` $\rightarrow$ `and(T,F)` $\rightarrow$ `F`. There is no way to reduce `F` any further: such terms are called *normal forms*.

A system of rewrite rules is called a *term rewriting system*. Term rewriting systems provide means to "execute" an algebraic specification. The "meaning" of a term is its normal form. Although it is possible that this normal form is not unique, we will require the rewrite rules to be defined in such a way that any normal form is unique (i.e., we require the term rewriting system to be *confluent*).

For a more thorough treatment of term rewriting systems in general, see [Klo91].

2.5 The Initial Algebra

A term rewriting system provides an intuitively clear meaning for an algebraic specification. The term rewriting interpretation, however, puts limitations on the equations. In this section we will return to our point of departure, and explain how an *algebra* can be derived from the specification rather than a term rewriting system, without imposing limitations on the equations. Such an algebra is called a *model* of the specification.

The equations are now undirected again, and can be applied in both directions. Consequently, a term is not reduced to one single normal form, but to an equivalence class of terms that are all equal to each other. All these equal terms are put into an *equivalence class*. In the term rewriting interpretation, with every term a normal form is associated, while in the algebraic interpretation with every term an equivalence class is associated.

The elements of the algebra over the specification are precisely these equivalence classes. The functions of the initial algebra are the functions defined in the signature. These operate on equivalence classes of closed terms.

Since it can be proved that the equations not only define an equivalence relation, but also a *congruence* (the operators respect the equivalence classes) we can use the equations as definitions for the behavior of the functions.

Various models differ in their definition for equal terms. In the initial model two terms are equal if and only if their equality can be proven from the equations, whereas in the final model two terms are equal unless their inequality can be proved. We will use the initial algebra approach.

The initial algebra has two important properties: there are no other elements in it than the terms over the signature (*no junk*), and elements are only equal if this can be proved from the equations (*no confusion*).

Thus, we can obtain an (infinite) algebra from a (finite) algebraic specification consisting of a signature and a number of equations.. The values in the algebra correspond to terms over the signature, and the functions in the algebra are the functions in the signature operating on equivalence classes of terms.

A far more detailed and formal treatment of initial algebras can be found in [GTW78]

2.6 Conditions and Negations

We can obtain a more powerful (i.e., needing less equations to obtain the same specified object) specification by adding conditional equations. In that case, equations of the form `T1 = T2 when T3 = T4` or `T1 = T2 when T3 != T4` become possible.

For instance, suppose we specify the natural numbers in the classical algebraic way: `0`, `succ(0)`, `succ( succ(0) )`, etc., then we may need a check (i.e., a function returning `T` or `F`) whether a number is zero or not. We can specify this function using a negative condition:

```
sorts
  NAT BOOLEAN
functions
  T:                -> BOOLEAN
  F:                -> BOOLEAN
  0:                -> NAT
  succ: NAT         -> NAT
  is-null: NAT      -> BOOLEAN
  add: NAT # NAT    -> BOOLEAN
variables
  x, y:             -> NAT
equations:
  [1]   add(x, 0) = x
  [2]   add(x, succ(y)) = succ( add(x,y) )
  [3]   is-null(0) = T
  [4]   is-null(x) = F when x != 0
```

2.7 Modularization

In order to keep large specifications manageable, modularization operations are needed for splitting large specification into several small ones.

This allows to distinguish sorts and functions that are *exported* by the module from the *hidden* sorts and functions that are only used to define some properties of the exported functions.

2.8 Syntactic Freedom

In large specifications, we do not want to be annoyed by clumsy notations. For instance, we do not want to write `and(T,F)` but `T and F`, and we do not want to write `push(S,I)` but something like `push I on S`. Thus, we do not want to be restricted to prefix operations, but would like to have postfix or mixfix as well. We want to have as much notational freedom as possible.

To allow this, we will use another notation for signatures:

```
sorts
  ELEMENT STACK
functions
  "[]"                            -> STACK
  "pop from" STACK                -> STACK
  "what is the top of" STACK "?" -> ELEMENT
  "push" ELEMENT "on" STACK       -> STACK
```

Although the use of free syntax may have been a little exaggerated in this example, it will be clear that this notational freedom does provide benefits.

We can construct terms over such a signature in the same way, we can write down equations over the signature, and we can eventually derive a term rewriting system from a signature with equations. Thus, we have obtained notational freedom without introducing any problems.

2.9 Signatures as Grammars

An interesting result is obtained when reading these free-syntax function declarations as $X ::= \alpha$ instead of α `->` X, and when considering the sorts as nonterminals. In that case each function declaration turns out to be a grammar rule of a BNF grammar for the language of closed terms over the signature.

3 ASF+SDF

Thus far, we only introduced general theory concerning algebraic specifications. By now we have developed enough machinery to introduce the ASF+SDF [Hen91] algebraic specification formalism smoothly.

Originally only ASF, an acronym for Algebraic Specification Formalism existed [BHK89, Hen88]. ASF is a formalism supporting modularization and conditional equations. A system, the ASF system, was able to generate rewriting systems from the specification, and to execute them.

At almost the same time, SDF, Syntax Definition Formalism, had been developed [HHKR89]. SDF supports the definition of lexical, context-free and abstract syntax at the same time, and incremental parsers can be generated from it.

These two formalisms have been combined into one algebraic specification formalism called ASF+SDF. The ASF part takes care of the (conditional) equations, while the SDF part provides complete syntactic freedom for the specifier.

Specifying in the ASF+SDF formalism is supported by the ASF+SDF *system* [Kli90] [Hen91, Chapter 5]. This system is able to generate parsers from ASF+SDF specifications and to derive term rewriting systems for specifications in ASF+SDF. Moreover, it can generate syntax-directed editors for both modules in the specification as well as terms over the signature. The system is able to perform several static and semantic checks on the specifications, and supports testing of specifications.

3.1 Lexical Syntax

SDF allows the definition of lexical syntax, i.e., the definition of the elementary "words" of the syntax. There are two important corollaries for the specifier: First of all the `LAYOUT` for the equations must be defined, i.e., the specifier himself can (and must) define what symbols constitute white space, and how comments can be recognized. A typical layout definition is:

```
exports
  lexical syntax
    [ \t\n]           -> LAYOUT
    "%%" ~[\n]* "\n"  -> LAYOUT
```

A space, tab, or new line is a layout symbol, as well as everything between two percent signs and a new line. If tokens of sort `LAYOUT` are detected in a text, they are ignored.

Secondly, variable declarations are treated as declarations of lexical syntax. This implies that any constructs allowed in the lexical syntax definition are allowed in the variable definition section as well. Consequently it is possible to define the variables `p1`, `p2`, `p3`, ... all at once:

```
variables
  "p"[0-9]*    -> SORTNAME
```

This declares all words starting with a `p` followed by zero or more characters in the range `0-9` to be variables of sort `SORTNAME`.

3.2 List functions

An important feature of the ASF+SDF formalism is the existence of list functions and list variables. List functions have a variable number of arguments, and list variables may range over any number of arguments of a list function.

As an example, suppose we would like to have a function `[ ]` for the empty set, `[ E1 ]` for a set with one element, `[ E1, E2 ]` for a set with two elements, and so on.

The way to define this in ASF+SDF is as follows:

```
exports
  sorts SET ELEMENT
context-free syntax
  "[" {ELEMENT ","}* "]"       -> SET
```

The asterisk * says that we want zero or more `ELEMENT`s, while the comma says that these should be separated by commas. Thus, a set consists of `ELEMENT`s, separated by commas and delimited by `[` and `]`.

This list notation is simply an abbreviation for the declaration of infinitely many functions `[ ... ]`, each with a different number of arguments. Likewise, the same (concrete) syntax could have been obtained by the following "normal" BNF grammar rules:

```
context-free syntax
   "[" ELEMENTS "]"      -> SET
   "[" "]"               -> SET
   ELEMENT               -> ELEMENTS
   ELEMENTS "," ELEMENT  -> ELEMENTS
```

In order to define equations over list functions, we need list variables:

```
exports
  sorts SET ELEMENT
context-free syntax
  "[" {ELEMENT ","}* "]"       -> SET
variables
  "Elts"[123]  ->  {ELEMENT ","}*
  "Elt"        ->  ELEMENT
equations
  [1]  [Elts1, Elt, Elts2, Elt, Elts3] =
          [Elts1, Elt, Elts2, Elts3]
```

`Elts1`, `Elts2`, and `Elts3` are list variables, ranging over list of zero or more `ELEMENT`s separated by commas.

Here we have specified in one single equation that elements of sets do not have multiplicity: Any set containing element `Elt` at least two times is equal to the set containing one occurrence less of `Elt`.

3.3 Disambiguation

SDF supports *priority* and *associativity* declarations, which are used to disambiguate terms which can otherwise be parsed in several ways. For instance,

```
exports
  sorts BOOL
context-free syntax
  "T"             ->  BOOL
  BOOL "and" BOOL  ->  BOOL {left}
  BOOL "or" BOOL   ->  BOOL {left}
priorities
  or < and
```

Here `and` has a higher priority than `or`, thus stating that, for instance, the term `T or F and F` should be interpreted as `T or (F and F)` rather than as `(T or F) and F`.

Moreover, both `and` and `or` are defined to be left associative, that is `T and T and T` is interpreted as `(T and T) and T` rather than as `T and (T and T)`.

3.4 Sections

A module in the ASF+SDF formalism may contain the following keywords:

- `imports`, listing the names of imported modules;
- `exports` or `hiddens`, listing the exported or hidden items;
- `sorts`, listing the sorts;
- `lexical syntax`, giving lexical syntax;
- `context-free syntax`, listing the function declarations. Attributes, between `{}` brackets, may be associated with functions, stating for instance that the function is `left` of `right` associative;
- `priorities`, giving priorities between functions;
- `variables`, declaring the variables used in the equations;
- `equations`, listing the (conditional) equations of the module.

4 Pascal

Pascal [JW86] is one of the best known and most influential programming languages. It has been standardized by the International Standardization Organization ISO [ISO83]. The Pascal specification described in this document conforms to that standard, except for conformant array parameters and gotos.

We loosely define the static semantics of a Pascal program to be all properties of the program that can be checked without executing it. Most of these properties deal with the type and scope rules of Pascal. We will assume that the reader is more or less familiar with these rules. We will, however, briefly mention some aspects of the static semantics that may be less familiar, yet have significant influence on the design of the specification.

First of all, each identifier, denoting for instance a type, procedure, or function, must be declared before it can be used. The so-called *required* identifiers (eof, char, integer, etc.) are declared implicitly. There is, however, one exception to this rule. The component of a pointer type definition may be a type identifier that has not yet been declared:

```
type   CellPtr = ^Cell;
       Cell    = record
                     Info: InfoType;
                     Next: CellPtr;
                 end;
```

In the similar situation for procedures (or functions), the identifiers may be used before the complete declaration if there is a *forward* declaration:

```
function  ParseTerm(var S:T): Boolean; forward;

function  ParseFactor(var S:T): Boolean;
  begin
    { ... call ParseTerm ... }
  end

function  ParseTerm;
  begin
    { ... call ParseFactor ... }
  end
```

Furthermore, type equivalence in Pascal is not based on structure. Each occurrence of a type constructor denotes a type that is distinct from any other type denoted by a constructor [ISO83, p.12]. Consequently, `A1` and `A2` are not of the same type, whereas `B1` and `B2` are:

```
type BType = array [1..10] of char;

var  A1: array [1..10] of char;
     A2: array [1..10] of char;
     B1: BType;
     B2: BType;
```

Finally, redefinition of identifiers, including the required identifiers, is always possible. The old definitions of symbols using an identifier *Id* within their definition should not change, however, if the definition of *Id* changes.

5 Pascal in ASF+SDF

5.1 Points of departure

While constructing the specification for the static semantics of Pascal, the following points of departure played an important role.

- Use the official document of the International Standardization Organization (ISO) defining Pascal [ISO83] as the source reference for Pascal, rather than a definition of some Pascal dialect,
- Optimize the specification with respect to readability, i.e., concentrate on ease of reading, writing and adapting the specification rather than on execution speed.
- Do not only specify whether the static semantics of Pascal program are legal or illegal, but also specify what is wrong with an incorrect program (i.e., the specification should produce error messages).

5.2 Overview

The ultimate goal is a function mapping a Pascal program to a possibly empty list of error messages. The static semantics of a program is correct if and only if this list is empty. Type checking does not stop after the first error has been found. A very simple error recovery mechanism is used.

The specification can be divided in three layers.

The top layer (Section 8) covers the "real" type check functions, i.e., the functions that operate on some syntactic construct and call the appropriate functions to check the constraints for that particular construct.

The bottom layer (Section 6) defines several basic properties of Pascal programs, like the syntax, contexts for representing declared variables, and several context-sensitive properties (what is the type of this identifier, is this type compatible with that other type, and so on).

The middle layer (Section 7) defines functions checking the constraints required by the top layer, using the properties defined in the bottom layer. The functions check some property, and if the check fails they generate an error message. This allows a separate treatment of error handling; neither the top nor the bottom level need to know that error messages are produced.

In order to keep the specification manageable, it has been split up into 34 modules. While discussing the specification, we will not cover the details of this modularization. More detailed descriptions of single modules can be found in the appendices.

6 Basic Structures

6.1 Syntax

First, we define what Pascal programs are. To do so, we define terms over the sort `PROGRAM`, which are constructed using terms over several other sorts, like `EXPR`, `STATEMENT`, and `BLOCK`.

These terms are the syntactically correct Pascal programs. They are defined by translating the BNF grammar rules given in the ISO standard directly into corresponding SDF rules in the signature. In some cases, we slightly changed the grammar, since the ISO grammar is not entirely unambiguous, and since SDF allows more elegant notations using, for instance, priority declarations.

Terms over these syntactic sorts will be the input for several type check functions.

6.2 Contexts

A Pascal programmer may define several entities (such as types, variables, or procedure) and may assign symbolic names (identifiers) to them. At each point in a Pascal program, some symbols have been declared, and only these symbols may be used. This list of declared symbols may, of course, be different at different points in the program. We will call such a list a *context* of a point in the program.

The context contains both the *required* identifiers (such as `integer` or `eof`) and the identifiers declared by the programmer.

The entries for the definitions of the user-defined identifiers are copies of the program texts giving their declarations. Thus, a context simply is a list of Pascal declarations.

The required identifiers that can be described using the conventional Pascal declaration constructs, like `type Boolean = (false, true)`, are represented in that way in the context. Those that cannot, like the procedure `readln` which may have any number of parameters, are described by the entries `predefined-function`, `predefined-procedure` or by the type definition `type <required-type-name> = <required-type-name>`.

As an example, consider the following skeleton of a part of a Pascal program

```
procedure proc1(par1, par2: t1);
   var v1: t1;
       v2: t2;
   procedure subproc(var p3: t3);
      const MAX = 30;
   begin
      ...  (* P1 *)
   end;
begin
   ...
end;
```

The context at point `P1` is shown below. Definitions for some of the required identifiers are also given:

```
[  predefined-procedure readln,
   type integer = integer,
   type Boolean = (false, true),
   function sin( x: real ): real,
   ...
   procedure proc1(par1, par2: t1),
   block-mark,
     var v1: t1,
     var v2: t2,
     procedure subproc(var p3: t3),
     block-mark,
       const MAX = 30 ]
```

To allow easy processing of scope rules, the `block-mark` entry is introduced. It is used to define a series of nested blocks. If a new block is entered, a `block-mark` is added to the context. If the end of a block is reached, all symbols added to the context since the most recent `block-mark`, as well as the `block-mark` itself, can be removed.

Note that it is not strictly necessary to define contexts. All information in the constructed `CONTEXT` term can be retrieved from the `PROGRAM` term itself. We introduced contexts because they are more practical. Searching the declaration of an identifier in a a `PROGRAM` term is much more complicated than looking it up in the corresponding `CONTEXT` term.

6.3 Type Representation

It should be clear by now that the `CONTEXT` is an essential structure in the specification. Moreover, we will use `CONTEXT`s as the representation for the various data types in the program.

A considerable part of checking the static semantics of a Pascal program is concerned with checking types. In an expression, the type of the operands of `+` must be `integer` or `real`, and the types of the actual parameters must match the types of the formal parameters of a procedure. In order to check these requirements, some representation of types is needed.

Single type declarations cannot be used as the representation for types, since type equivalence is not based on structure (see Section 4).

The identifiers used in a type declaration may be redefined later on. Moreover, textual expansion of the type identifiers is not possible either, since the type definitions for pointers may be recursive.

Type equivalence depends on the place where the type was defined, or, to say the same, depends on the context in which the type was defined. Therefore, we have chosen to represent a type by a CONTEXT. If a CONTEXT is used as a type, the entry entered last should be a type definition, and the entries before it constitute the context of the type. As a consequence, equivalence of Pascal types can be checked simply by the = operator applied to terms of the sort CONTEXT.

In order to allow practical processing of types, several simple abstract functions have been defined on contexts. There are functions to classify types (is a type an array, a subrange, a numerical type, etc.), or to retrieve information from types (what is the component type of an array, of a set, of a pointer, etc.). All information needed to answer these questions is available in the type representation (the CONTEXT) itself. The only type operation that needs extra information is the retrieval of the component of a pointer type, since the definition of the component type need not be given *before* the definition of the pointer type itself.

We illustrate definitions of these functions by two examples. A context denotes a subrange type if its most recent entry is a subrange type definition:

```
context-free syntax
  "is-subrange-type?" "(" CONTEXT ")"  -> BOOL

variables
  "_const"[12]*     -> CONST
  "_id"             -> IDENT
  "Prefix"          -> { ENTRY "," }*

equations
  is-subrange-type?([Prefix,
      type _id1 =  _const1 .. _const2  ])
           =
          TRUE
```

Secondly, in order to obtain the type representation for the index-type of an array, the definition for the index-type subterm should be retrieved in the preceding entries of the context:

```
context-free syntax
  CONTEXT "." "index-type"      -> CONTEXT

variables
  [IC]"-type"       -> TYPE-DENOTER
  "_id1"            -> IDENT
  "Prefix"          -> { ENTRY "," }*

equations
  [Prefix, type _id1 = array [I-type] of C-type]
          . index-type
               =
       get-type of I-type in [Prefix]
```

As an example of the practical use of these abstract functions, consider the following part of the specification of type compatibility. The equation is the algebraic alternative to the ISO sentence “Types T1 and T2 shall be designated *compatible* if both T1 and T2 are subranges of the same host type” [ISO83, p.20] Abstract functions is-subrange-type and .host-type are used:

```
context-free syntax
 CONTEXT "and" CONTEXT "are compatible?" -> BOOL

variables
 "T"[12]*    -> CONTEXT

equations

 is-subrange-type?(T1) = TRUE,
 is-subrange-type?(T2) = TRUE,
 T1.host-type the same as T2.host-type? = TRUE
 ===========================================
 T1 and T2 are compatible? = TRUE
```

These functions provide an abstract interface to retrieve all information necessary from Pascal data types.

7 Constraints

7.1 Error Messages

Error messages are terms over a signature of function names that are normal English sentences (using the syntactic freedom of the formalism). These functions may be parameterized by language constructs. For example:

```
context-free syntax
 "Identifier" IDENT "not declared."   -> ERROR
 "Parameter lists are not congruous." -> ERROR
```

If a more detailed description of the place where the error occurred is desired, the language construct in which the error occurred can be set. The signature for this is shown, including the specification of lists of errors.

```
context-free syntax
  CONSTRUCT ":" ERROR      -> CONSTR-ERROR
  CONSTR-ERROR*            -> ERRORS
```

7.2 Communication

Many tc-functions will be needed to check all kinds of Pascal constructs. These tc-functions will have to communicate information. In order to provide communication between tc-functions in a uniform way, we introduce *environments*.

The environment provides the information needed to check a particular construct, and it provides a way to represent results of a check.

The most important fields of an environment are the `context` field, representing the `CONTEXT` of the construct in question, the `errors` field, representing the list of errors encountered so far, and the `result` field, which can be used to pass results of a type check, for example the type of an expression. Operations to manipulate these environments are provided.

Thus, a typical tc-function will retrieve its information from the (`context` of the) incoming environment, and will return its information by an adapted environment (either an error added to the `errors`, or a result in the `result` field).

7.3 Constraints

Many tc-functions will have to check constraints on a construct or context of a construct. If that constraint is met, no special action needs to be taken, but if it is not met, an error message should be added to the environment. Different tc-functions may have to check the same constraints.

We will call the class of functions checking these constraints *should-be* functions. The names of these functions are like normal English sentences, and typically have the words "should be" in their name.

The should-be functions operate on an environment. This environment is returned unchanged if the constraint is met, but it is returned with an error message added to it if the constraint is not met. The constraint is typically tested by a Boolean function.

We give an example of the specification of a typical should-be function:

```
context-free syntax
  CONTEXT "should be ordinal in" ENV  -> ENV

variables
  "T"    -> CONTEXT
  "Env"  -> ENV

equations

 is-ordinal-type?(T) = TRUE
 ===============================
 T should be ordinal in Env = Env

 is-ordinal-type?(T) != TRUE
 ===============================
 T should be ordinal in Env =
  add error [Ordinal type expected.] to Env
```

7.4 An Example

We illustrate the use of environments and should-be functions by a typical tc-function, the function `stat-tc` checking the Pascal `if`-statement.

```
context-free syntax
  "stat-tc(" STATEMENT "," ENV ")"  -> ENV

variables
  "_stat"     -> STATEMENT
  "_expr"     -> EXPR
  "Env"[1-4]  -> ENV

equations
  expr-tc( _expr, Env1 ) = Env2,
  stat-tc( _stat, Env2 ) = Env3,
  Env2.result should be Boolean in Env3 = Env4
  ============================================
  stat-tc( if _expr then _stat, Env1 ) = Env4
```

The tc-functions `stat-tc` and `expr-tc` have an environment as argument and return a (possibly changed) environment. The should-be function is used to check the constraint that the expression is of type Boolean. It takes the result of type checking an expression, that is the `result` field of the environment `Env2`, and returns an adapted version of `Env3`.

8 Type check functions

The contexts, errors, environments, and should-be functions provide enough machinery to specify the functions that really check a Pascal language construct (called the *tc-functions*).

The tc-functions together perform a single pass over the syntax, and a uniform way of checking the static semantic of the program. A tc-function may check a language construct by calling one or more should-be functions in order to check constraints, or by calling other tc-functions for the syntactic parts of the construct, or by a combination of these. The tc-functions operate on a language construct and an `ENV` structure, and return a (possibly changed) `ENV` structure.

Thus, checking an `if` statement, (Section 7.4) implies calling the tc-functions for `EXPR` and `STAT`, and checking the constraint that the expression `should be Boolean`.

The top tc-function checks an entire Pascal program. It is shown below. This function operates on a program only (not on an `ENV` structure), and returns a list of errors.

```
context-free syntax
 "tc(" PROGRAM ")"                          -> ERRORS
 "header-tc(" PROG-HEADER "," ENV ")" -> ENV
 "block-tc"(" BLOCK "," ENV ")"       -> ENV

variables
  "Env"[123]   -> ENV
  "Header"     -> PROG-HEADER
  "Block"      -> BLOCK

equations
  header-tc( Header, new-env) = Env1,
  block-tc(  Block,  Env1)    = Env2
  ===================================
  tc( Header; Block. )  =  Env2.errors
```

8.1 Declarations

Checking the declarations (labels, constants, types, variables, procedures, and functions) of a Pascal program involves (1) checking that the declarations themselves are correct, and (2) representing the declarations in the `context` field of the `environment`, so that the proper use of the defined identifiers can be checked if they appear later on in the program.

The most important checks of the declarations themselves deal with the correct use of identifiers. Within one block, no identifier should be declared more than once. If a declaration uses another identifier, this identifier should have been declared already, and it should be used according to its definition. For instance:

```
context-free syntax
 "const-tc"(" CONST-DEF "," ENV ")"  -> ENV
```

```
variables
 "_id"[12]    -> IDENT
 "E"[1-3]     -> ENV
 "Sign"       -> SIGN
 "Type"       -> CONTEXT

equations
 _id1 should not be declared in block of E1 = E2,
 _id2 should be a constant in E2 = E3,
 const-type of _id2 in E2.context = Type,
 Type should be a number in E3 = E4
 ============================================
 const-tc( _id1 = Sign _id2, E1 ) =
   add decl "const _id1 = Sign _id2" to E4
```

Thus, for the definition `const id1 = -id2;` it should be checked that `id1` has not yet been declared within the current block, and that `id2` has been defined as a numerical constant. The should-be functions required for these constraints can be defined easily.

Constants are the only items in a Pascal program for which *values* are maintained during type checking. For variables or expressions this is not needed, since their values depend on the execution of the program. For constants of an ordinal type (`integer`, `char`, or enumeration), the ordinal values, as defined by the ISO standard, are used to check subranges. Empty subranges, like `1 .. -1` are forbidden.

Correct handling of procedures (functions) is obtained by first adding the procedure-heading to the context (so that the procedure is known and can be called), then adding a `block-mark` to the context, followed by the addition of all formal parameters as local variables of the procedure. This allows easy checking of the proper use of the parameters in the body of the procedure. As soon as the end of the procedure body is reached, all declarations within this block, including the `block-mark`, are removed from the context. The procedure heading, however, entered before the `block-mark`, is not removed, because the procedure still can be called by other procedures.

We give an example of the functions to check a type denoter. The checks for a type denoter have to be specified for each type constructor (array, record, enumeration, etc.). We show the checks for the array declaration. Multidimensional arrays can be defined in two ways, as `array [i1,i2] of c`, or as `array [i1] of array [i2] of c`. These types are equivalent according to the standard [ISO83, p.14]. The first equation (below) makes this equivalence explicit. The tc-function for the arrays (in the second equation) then only has to check the one-dimensional case, which is done by checking the index type and the component type of the array, and by checking that the index type is an ordinal type. Notice the use of the variable name `_i1.._i2`; a common mathematical notation for one or more items separated by comma's.

```
context-free syntax
 "type-denot-tc" "(" TYPE-DENOTER "," ENV ")"
                                          -> ENV
variables
 "Env"[1-4]  -> ENV
 [IC]"-type" -> TYPE-DENOTER
 "_i1"       -> TYPE-DENOTER
 "_i2.._iN"  -> { TYPE-DENOTER "," }+

equations
 array [ _i1, _i2.._iN ] of C-type =
  array [_i1] of array [_i2.._iN] of C-type

 type-denot-tc( I-type, Env1 ) = Env2,
 type-denot-tc( C-type, Env2 ) = Env3,
 Env2.result should be ordinal in Env3 = Env4
 ==========================================
 type-denot-tc( array [I-type] of C-type,
        Env1 ) = Env4
```

Both pointer component identifiers and forwarded procedures or functions (see Section 4) deal with delayed definitions for identifiers. Therefore, in spite of several differences, they are handled in a similar way. We will briefly describe the pointer case.

For each pointer type definition, add the identifier in the component to a list of pointer-component-identifiers. Check at the end of the

checks for the type definition part, whether a definition has been given for all identifiers in that list. Consequently, a list of pointer identifiers must be passed between type definition check functions. This list can be made part of the environment.

8.2 Expressions

Type checking an expression (constants, variables, function calls, arithmetic, Boolean, etc.) always results in a type for that expression, which is returned in the `result` field of the `environment`. This allows to check easily whether the expression has been used in a legal way. Remember for example the tc-function for the `if` statement (Section 7.4), which involved checking whether the type of the expression was `Boolean`.

It is easy to find a type for most of the simple expressions (numbers, strings, constants, etc.) of Pascal. For the empty set `[ ]` this is more difficult. Since the empty set should be compatible with each other set, it is impossible to associate a type already available in Pascal with the empty set. Therefore, we introduce a type not available in Pascal itself, which we call `empty-set`, and we define the `is-compatible?` function in such a way that `empty-set` is compatible with any set. For similar reasons we have introduced the `nil-pointer` as a new type. Type checking simple expressions is illustrated below:

```
context-free syntax
 "sim-expr-tc(" EXPR "," ENV ")" -> ENV

variables
  "_id"          -> IDENT
  "Env"[1-3]*    -> ENV
  "Type"         -> CONTEXT

equations
  _id should be a constant in Env1 = Env2,
  const-type of _id in E1.context =  Type
  ======================================
  sim-expr-tc( _id, Env1 ) =
    set-result of Env2 to Type

 sim-expr-tc( nil, Env ) =
  set-result of Env to [nil-pointer]

 sim-expr-tc( [], Env ) =
  set-result of Env to [empty-set]
```

The type of an expression denoting a variable access is simply the type of the variable as it can be found in the context. Retrieval of components of variables of structured types (e.g. `a[i]`, `a^` or `a[i]`) results in the type of the component of the variable. First, of course, it is checked that the type of the variable allows the retrieval of a component.

The type of an expression denoting a function call is the type returned by the function, which is found in the context given the function identifier. The most complex part of checking a function call is checking that the actual parameters match the formal parameters. Since parameters of functions and procedures are checked in a similar way, we will discuss both here, even though procedures are not expressions in Pascal.

The requirements for the normal, user-defined procedures are easily catched by the should-be approach. The constraints include that there must be neither too few, nor too many actual parameters, that the types of the actual parameters must be compatible (value parameter), or assignment-compatible (`var` parameter) with the formal parameters, and that for procedure or function parameters, the parameter lists must be congruous. In addition, for `var` parameters, the actual parameters should not be components or fields of `packed` structured types. Finally, the context-free syntax allows all actual parameters to be postfixed by a format, e.g. the `:3:4` in the required procedure `write(a:3:4)`. This is only allowed for the procedure `write`. Since it is impossible to "wire" this in the context-free syntax (the programmer may redefine the identi-

fier `write`), the correct use of the format has to be checked for each actual parameter.

Slightly different is the specification for the required procedures, a part of which is shown below. Many input/output procedures have an optional file parameter. If this parameter is omitted, the procedures operate on standard `input` or `output`. Besides, any number of parameters may be given. The ISO standard specifies these procedures by summarizing the equivalences between the various cases, and by giving the constraints for one of them. For example, the call `readln(a,b)` is equivalent to (if `a` is not a file) `read(input,a); read(input,b); readln(input)`. The requirement for `readln` with only one parameter then is that the parameter should denote a file. In the algebraic specification we follow the same approach, the specification for calls of the procedure `readln` is shown below.

```
context-free syntax
 "std-io-tc("  IDENT PAR-LIST "," ENV ")" -> ENV
 "file-io-tc(" IDENT PAR-LIST "," ENV ")" -> ENV

variables
  "Env"        -> ENV
  "_par"      -> ACTUAL-PAR
  "_file"     -> ACTUAL-PAR
  "_pars"     -> { ACTUAL-PAR "," }*
  "_par+"     -> { ACTUAL-PAR "," }+

equations
  std-io-tc( readln, Env) =
    file-io-tc( readln(input), Env)

  is-file-variable?(_par, Env) != TRUE
  ===================================
  std-io-tc( readln(_par, _pars), Env) =
    file-io-tc(readln(input, _par, _pars), Env)

  is-file-variable?(_par, Env) = TRUE
  ==================================
  std-io-tc( readln(_par, _pars), Env) =
    file-io-tc(readln(_par, _pars), Env)

  file-io-tc( readln(_file, _par+), Env ) =
    file-io-tc( readln(_file),
      file-io-tc(read(_file, _par+), Env))

  file-io-tc( readln(_file), Env ) =
    _file should be a text file variable in Env
```

It would be possible to give a specification in should-be style for each arithmetic operator (`+`, `and`, `<` etc.). However, there are 16 operators which all may have operands of several types, and this may result in many large and unreadable equations.

ISO [ISO83, p.39] solves the problem by giving tables summarizing the various operand types allowed for each operator. The question whether `real` and `integer` are allowed as operands for the `+` operator, can be answered simply by looking in the table for the `+` and checking the operand types in that entry.

But if this is the most practical way to specify operand types, why not do it that way in our algebraic specification?

```
sorts
  TABLE-HEADER TABLE-ENTRY TABLE

context-free syntax
 "Operator | Operand1 | Operand2 | Result "
 " --------+----------+----------+--------"
                             -> TABLE-HEADER
 OPERATOR "|" OP-TYPE "|" OP-TYPE "|" OP-TYPE
                             -> TABLE-ENTRY
 TABLE-HEADER TABLE-ENTRY*  -> TABLE
 relational-op-table        -> TABLE
 arithm-op-table            -> TABLE

equations
 arithm-op-table =
  Operator | Operand1 | Operand2 | Result
  ---------+----------+----------+--------
    +      | int      | int      | int
    +      | int      | real     | real
    +      | real     | int      | real
    +      | real     | real     | real
    +      | set      | set      | set
    ...
 { similar for - * / div mod or not and}

 relational-op-table =
  Operator | Operand1 | Operand2 | Result
  ---------+----------+----------+--------
    in     | ordinal  | set      | boolean
    <      | string   | string   | boolean
    <      | simple   | simple   | boolean
```

```
  <=     | simple   | simple   | boolean
  <=     | string   | string   | boolean
  <=     | set      | set      | boolean
  ...
{ similar for > >= = <> }
```

8.3 Statements

We have already seen how procedure statements (Section 8.2) and `if` statements are checked. Most of the tc-functions for statements (`if then else`, `while do`, `repeat until`, assignment, and `begin .. end`), follow the same approach as the tc-function for the `if` statement. For the `case` statement the uniqueness of the case constants has to be checked. For a `with` statement a new block can be created, in which all fields of the record in question are entered as local variables.

We will only describe the `for` statement, for which the type check equation is given in the next figure. It is the most complex statement to be checked. Moreover, it is the most complex equation of the entire specification, and we think it is interesting to see how even the most complex equation is still manageable.

Type checking the `for` statement mainly deals with checking properties of the control variable. The constraints for the control variable, illustrated clearly in the equation itself, are: the control variable should be declared in the same block in which the `for` statement is used (line 9), the control variable should possess an ordinal type (line 15), and both the initial (`_expr1`) and the final (`_expr2`) should be assignment-compatible with the type of the control variable (lines 18 and 21).

The notions of lines 10 and 11, control and threatening variables, may need an explanation. In Pascal [ISO83, p.46], the loop variable of a `for` statement (called the *control* variable) should not be at the same time a control variable of an outer `for` statement. This is checked in line 10.

Moreover, the control variable should not be *threatened* by assignments to it within the body of the `for` statement. This is checked in line 11.

To implement the functions of line 10 and 11, we have used an idea of Welsh and Hay [WH86, Chapter 10]. The definitions of variable declarations has been changed to allow marking variables. Marks can be `possibly-threatening`, `control-variable`, or no mark at all. During the checks for the body of the `for` statement, the control variable is marked `control-variable`. Variables that may become control variables, and are assigned some where are marked `possibly-threatening`. If a `possibly-threatening` variable is used as a control variable, or if a `control-variable` is assigned somewhere, an error is raised.

The mark is set in line 13, and reset in line 24.

9 Related Work

Thanks to the importance of Pascal and to the influence of the language on other languages, several formal specifications for Pascal have been made over the past few years. All recent Pascal specifications are based on the ISO standard [ISO83], which is written in formal English but not in a formalism with mathematically defined semantics.

Attribute grammar specifications are given for the synthesizer generator in the Synthesizer Specification Language [RT89] and for the GAG system in the ALADIN formalism [KHZ82]. From these, just as from ASF+SDF specifications, parsers, syntax-directed editors, and type checkers can be derived. Both the attribute grammars and our algebraic specification take about 60 pages, but the ASF+SDF specification seems to be better readable. Moreover, it does not depend on predefined notions like integers or strings, but

```
 1.  variables
 2.    "_id"        -> IDENT
 3.    "E"[0-9]*    -> ENV
 4.    "_expr"[12]  -> EXPR
 5.    "_stat"      -> STATEMENT
 6.    "(Down)To"   -> DOWN-TO
 7.
 8.  equations
 9.    _id should be a variable in inner block of E1 = E2,
10.    _id should not be a control variable in E2 = E3,
11.    _id should not be a possibly threatening variable in E3 = E4,
12.
13.    mark-variable(_id, control-variable, E4) = E5,
14.      var-access-tc(_id, E5) = E6,
15.      E6.result should be ordinal in E6 = E7,
16.
17.      expr-tc(_expr1, E7) = E8,
18.      E8.result should be assignment-compatible with E6.result in E8 = E9,
19.
20.      expr-tc(_expr2, E9) = E10,
21.      E10.result should be assignment-compatible with E6.result in E10 = E11
22.
23.      stat-tc(_stat, E11) = E12,
24.    mark-variable(_id, , E12) = E13,
25.    ======================================================================
26.    stat-tc( for _id := _expr1 (Down)To _expr2 do _stat, E1) = E13
```

defines them all explicitly.

Watt's extended attribute grammar for Pascal [Wat79] is very compact, but not yet based on the ISO standard. Duke [Duk87] gives a specification in *predicate rules* [DJR87], an approach similar to attribute grammars in a notation akin to the Z specification language. Neither of these specifications can be executed.

A very early formal specification of the semantics of Pascal is given by Hoare and Wirth [HW73], but they "consider such topics as rules about the scope or validity of names as belonging to the realm of syntax" [HW73, p.336], so they hardly cover the static semantics of Pascal.

A denotational specification, also according to the ISO standard, of both the static and dynamic semantics for Pascal is given by Andrews and Henhapl [BJ82, Chapter 7], in 75 pages using the VDM formalism.

Algebraic specifications of (smaller) Pascal-like languages can be found in [BHK89, Meu88]. They are written in the ASF or ASF+SDF formalism. Like the current Pascal specification, they are partitioned in modules for the syntax, contexts, environments, and functions for type checking. Unlike the current specification, the contexts are not used for the representation of types, since the specified languages use other type equivalence rules. Likewise, these specifications can use a less rigorous approach to error handling and constraints than required for the real-life language Pascal.

10 Concluding Remarks

We have given the first algebraic specification for the static semantics of Pascal. We aimed at readability and clearness by carefully choosing

the names of sorts, functions, and variables, by providing abstract interfaces, like the functions operating on types, by looking for a good modularization, and by striving for as much uniformity as possible.

The uniform way of dealing with constraints and errors, using the *should-be* functions, raises the question whether the processing of errors can be mechanized. Can clear error messages be derived from the names of the functions checking particular constraints? If so (as suggested, among others, also by [BLM89, JJWW90]) we would have an algebraic specification of the correct static semantics of a language, and a system (typically an adapted term rewriting system) capable of producing adequate error messages for the incorrect programs analogous to the generation of a syntax checker from a BNF grammar. This promises to be an interesting area for further research.

Another question concerns the style of the specification. Inspired by the objective of readability, a *software engineering* specification style has been used, resulting sometimes in verbose names or superfluous functions. It would also be interesting to construct a specification using a more *mathematically* oriented style, e.g., with short names and the absolute minimum number of functions.

We have constructed the algebraic specification in software-engineering style with *validation* against the ISO standard in mind. The names of functions and structures in the algebraic specification precisely match the names of the ISO concepts they intend to formalize.

Alternatively, we could have written the specification in a more mathematical style if we had intended to *verify* the specification with another formal definition (e.g. VDM, [BJ82, Chapter 7]) of Pascal. Although this would be very interesting, proving some mathematical relation or equivalence between the algebraic and denotational specification probably would be quite complex.

Another interesting question is which concepts or modules can be reused in specifications for other programming languages. It may be clear that for the static semantics of other block structured languages like Modula-3, Ada, or C, many aspects of the Pascal specification can be reused. Many of them probably can be reused for the dynamic semantics as well; candidates are the environments, functions on types, and the should-be approach for error handling.

Algebraic specifications for programming languages may become more and more important, especially when they will be used as the input for programming environment generators. To allow easy construction of large specifications, generic modules for important structures, development methods, and specification heuristics are needed. The only way to obtain these is by writing, reading, and exchanging many complex and realistic algebraic specifications.

Acknowledgements

Paul Hendriks and Emma van der Meulen introduced me to ASF+SDF very patiently. Jasper Kamperman, Emma van der Meulen, Jan Rekers, and Machteld Vonk commented on (parts of) the manuscript. Jan Heering and Paul Klint carefully read the entire paper, suggesting many useful improvements. Thank you all.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BJ82] D. Bjørner and C.B. Jones. *Formal Specification and Software Development.* Prentice-Hall, 1982.

[BLM89] M.G.J. van den Brand, J.J. Langeveld, and H. Meijer. Syntactical and semantical error reporting in automatically generated backtrack parsers. In *Conference Proceedings of Computing Science in the Netherlands, CSN'89*, pages 57–69. SION, 1989.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DJR87] R. Duke, D. Johnston, and G.A. Rose. Specifying the static semantics of block structured languages. *The Australian Computer Journal*, 19(2):99–104, 1987.

[Duk87] R. Duke. Predicate rules for Pascal static semantics. Technical Report 86, The University of Queensland, Department of Computer Science, September 1987.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[Hen88] P.R.H. Hendriks. ASF system user's guide. Report CS-R8823, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988. Extended abstract in: Conference Proceedings of Computing Science in the Netherlands, CSN'88 1, pp. 83-94, SION (1988).

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[ISO83] International organization for standardization. *International standard ISO 7185, Programming languages - PASCAL, Ref. No. 7185-1983(E)*, first edition, 1983.

[JJWW90] P. de Jager, W. Jonker, A. Wammes, and J. Wester. On the generation of interactive programming environments - A LOTOS case study. Technical report, PTT Research Tele-informatics, Groningen, 1990. To appear in Software Engineering and its Applications.

[JW86] K. Jensen and N. Wirth. *Pascal User Manual and Report, ISO Pascal Standard.* Springer-Verlag, third edition, 1986.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator.* Lecture Notes in Computer Science 141. Springer Verlag, 1982.

[Kli90] P. Klint. A meta-environment for generating programming environments. Report CS-R9064, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990. To appear in: Proceedings of the METEOR workshop on Methods Based on Formal Specification, ed. J.A. Bergstra and L.M.G. Feijs, Lecture Notes in Computer Science, Springer-Verlag.

[Klo91] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabby, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol I.* Oxford University Press, 1991. To appear.

[LB79] B.H. Liskov and V. Berzins. An appraisal of program specifications. In P. Wegner, editor, *Research Directions in Software Technology*, pages 276–301. MIT Press, 1979. Also in *Software specification techniques*, edited by N. Gehani and A.D. McGettrick, 1986, Addison Wesley pp. 3-23.

[Meu88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[Wat79] D.A. Watt. An extended attribute grammar for Pascal. *SIGPLAN Notices*, 14(2):60–74, 1979.

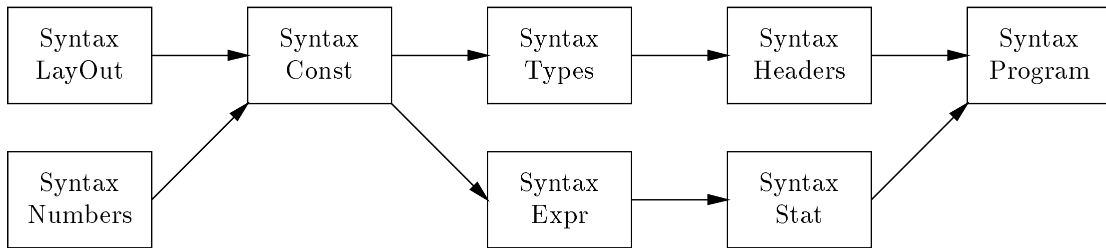
[WH86] J. Welsh and A. Hay. *A Model Implementation of Standard Pascal.* Prentice-Hall, 1986.

[Win90] J. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, 1990.

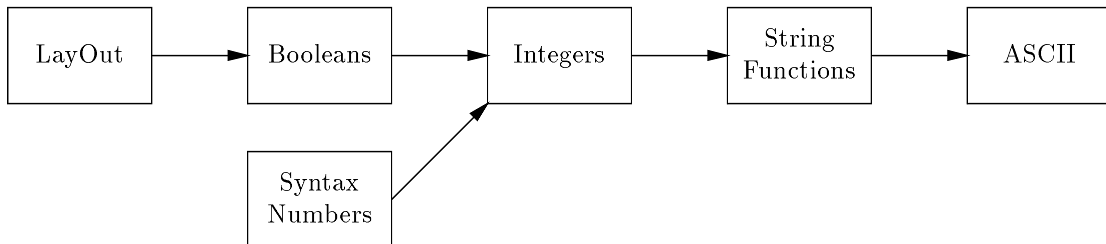
A Import Graphs

The import graph for the specification is given. Several modules are grouped in 5 groups: Syntax, Miscellaneous, Contexts, Environments, and Type Checking. An arrow from module M_1 to M_2 implies that M_1 is imported by M_2.

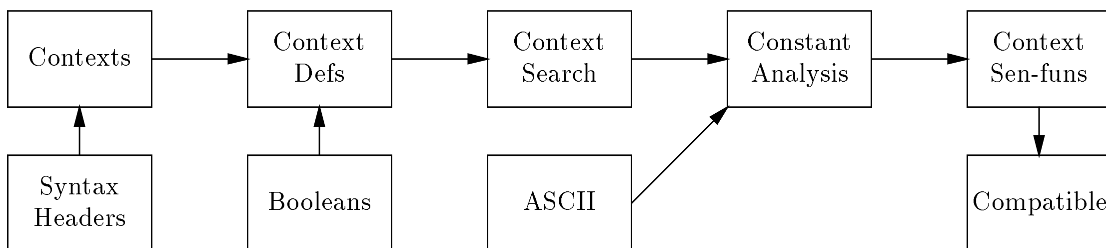
A.1 Syntax



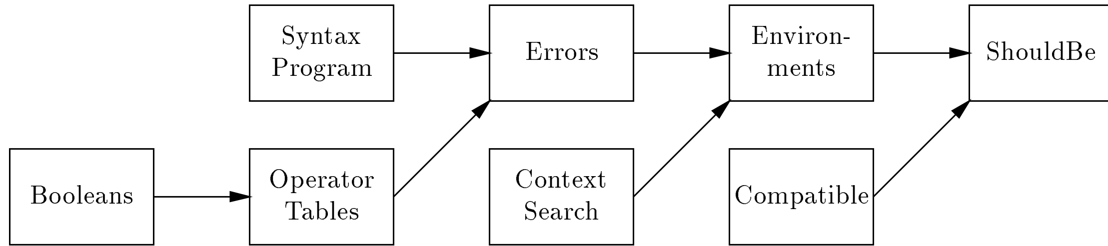
A.2 Miscellaneous



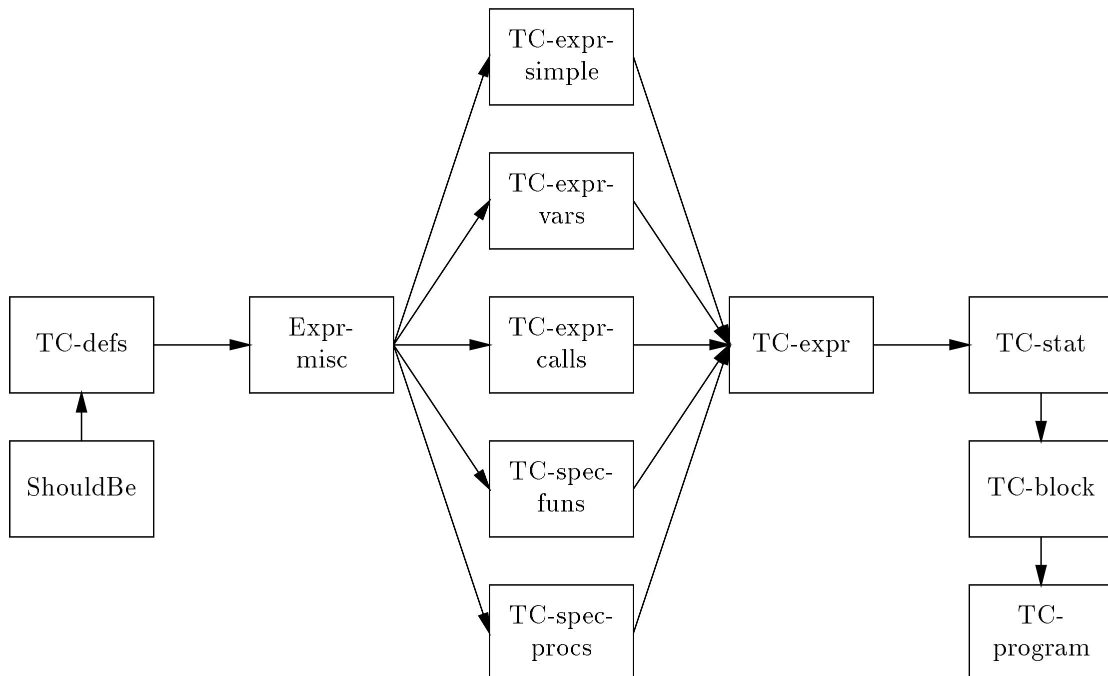
A.3 Contexts



A.4 Environments



A.5 Type checking



B Module Descriptions

Each module in the specification is briefly described, in the order in which they appear in the import graphs. This appendix is intended for reference purposes.

B.1 Syntax

SyntaxLayout Define the lexical syntax of Pascal layout symbols, e.g. white space and comment brackets `{` and `}`.

SyntaxNumbers Define the lexical syntax of Pascal unsigned numbers, e.g. `123`, `123.45`, or `123e-2`.

SyntaxConst Define the syntax of basic constant values, i.e., signed numbers, strings and labels.

SyntaxTypes Define the syntax of the type constructors (array, record, etc.) of Pascal.

SyntaxHeaders Define the syntax of procedure and function headers, and of the various sections (label, const, type, var, procedure or function) in a block.

SyntaxExpr Define the syntax of expressions, e.g. constants, variable accesses, Boolean operators, sets, arithmetic operators, relational operators, etc.

SyntaxStat Define the syntax of all statements (while, repeat, if, etc.).

SyntaxProgram Define the syntax of the program header and of the Pascal block.

B.2 Miscellaneous

Layout Define the Layout as used in the equations.

Booleans Define the syntax and semantics of the Booleans as needed within the specification. These Booleans are *not* used within a Pascal program, in which the Booleans simply are a predefined enumeration type. In order to avoid confusion, the values `TRUE` and `FALSE` in the specification are CAPITALIZED.

Integers Define the syntax and semantics of the Integers. This module imports the module SyntaxNumbers, thus allowing to perform arithmetic on numbers used in a Pascal program.

StringFunctions Define string functions, at the moment only a function to determine the length of a string.

ASCII Define the ordinal values of elements of the Pascal predefined data type "char". This module has been generated by a Pascal program.

B.3 Contexts

Remember that the context plays a major role in the specification, since it is used as the representation for types.

Contexts Define the contexts as tables of declarations, and define the initial block containing the definitions for the required identifiers.

ContextDefs Define functions which investigating how the identifier in the top entry of the context is defined, e.g. as a type, as a variable, as a function, etc.

ContextSearch Define functions searching a context, i.e., returning the most recent definition for a given identifier.

ConstantAnalysis Though ISO defines constants as single lexical entities, they still

may have properties depending on the contexts (e.g. the type of `const a = -b;` depends on the declaration of `b`). These context operations are defined in this module.

Context-sen-fun In this module, all context sensitive definitions of the ISO standard are given. These include classification of several types, retrieval of components of types, etc.

Compatible Define type compatibility.

B.4 Environments

OperatorTables Define the tables and functions on them, used when type checking expressions. Redefinition of operator names is necessary, since only lexical items can appear in associativity and priority declarations.

Errors Define the syntax of the error messages. Modules providing sorts needed in the messages are imported.

Environments Define the structure for the environments and the basic operations on environments.

ShouldBe Define the should-be functions

B.5 Type checking

TC-defs Define how to type check the declarations of a block.

Expr-misc Define several functions needed when type checking expressions.

TC-expr-simple Define how to type check simple (constants) expressions.

TC-expr-vars Define how to type check variable accesses.

TC-expr-calls Define how to type check user-defined function and procedure calls.

TC-spec-funs Define how to type check calls to predefined functions (eof, eoln, ord, etc.)

TC-spec-procs Define how to type check calls to predefined procedures (readln, write, pack, etc.).

TC-expr Merge all expression type check functions.

TC-stat Define how to type check statements.

TC-block Define how to type check an entire block, including forwarded functions and procedures.

TC-prog Define how to type check an entire program, including the program header.

C Text of the Specification

In this appendix the complete text of the specification for the static semantics of Pascal is given.

C.1 Syntax

```
module SyntaxLayOut
%% Define lay out symbols of Pascal programs. Note that in this specification
%% no * between comment delimitters (* and *) is allowed.

exports
  lexical syntax
    [ \t\n]                                               -> LAYOUT
    "{" ~[}]* "}"                                         -> LAYOUT
    "(*" ~[*]* "*)"                                       -> LAYOUT
end module SyntaxLayOut


module SyntaxNumbers
%% Specify the syntax of real and integer numbers
exports
  sorts UNS-NUMBER UNS-INT UNS-REAL DIGIT

  lexical syntax
    [ \t\n]                                                -> LAYOUT
    [0-9]                                                  -> DIGIT
    DIGIT+                                                 -> UNS-INT

    UNS-INT "." UNS-INT                                    -> UNS-REAL
    UNS-INT "." UNS-INT [eE] UNS-INT                       -> UNS-REAL
    UNS-INT "." UNS-INT [eE] [+\-] UNS-INT                 -> UNS-REAL
    UNS-INT [eE] UNS-INT                                   -> UNS-REAL
    UNS-INT [eE] [+\-] UNS-INT                             -> UNS-REAL

  context-free syntax
    UNS-INT                                                -> UNS-NUMBER
    UNS-REAL                                               -> UNS-NUMBER

  variables
    "_uns-int"[0-9]*                                       -> UNS-INT
    "_uns-real"[0-9]*                                      -> UNS-REAL
    "_uns-number"[0-9]*                                    -> UNS-NUMBER
end module SyntaxNumbers


module SyntaxConsts
%% Define constants and  values, ISO section 6.1.3, 6.1.5 6.1.7 and 6.3.
%% Empty strings are not allowed in ISO, but are tolerated in this definition.
imports SyntaxLayOut SyntaxNumbers

exports
  sorts IDENT STRING STRING-PART CONST NEW-CONST SIGN LABEL
```

```
  lexical syntax
    [a-zA-Z] [_A-Za-z0-9]*                                    -> IDENT
    "'" ~['\n]* "'"                                           -> STRING-PART
    STRING-PART+                                              -> STRING
    DIGIT                                                     -> LABEL
    DIGIT DIGIT                                               -> LABEL
    DIGIT DIGIT DIGIT                                         -> LABEL
    DIGIT DIGIT DIGIT DIGIT                                   -> LABEL

  context-free syntax
    "+"                                                       -> SIGN
    "-"                                                       -> SIGN
    STRING                                                    -> NEW-CONST
    UNS-NUMBER                                                -> NEW-CONST
    SIGN UNS-NUMBER                                           -> NEW-CONST
    SIGN IDENT                                                -> CONST
    IDENT                                                     -> CONST
    NEW-CONST                                                 -> CONST

  variables
    "_const"[0-9]*     -> CONST      "_const"[0-9]*"*"  -> {CONST ","}*
    "_id"[0-9]*        -> IDENT      "_id"[0-9]*"+"     -> {IDENT ","}+
    "_sign"[0-9]*      -> SIGN       "_id"[0-9]*"*"     -> {IDENT ","}*
    "_new-const"[0-9]* -> NEW-CONST  "_string"[0-9]*    -> STRING
    "_label"[0-9]*     -> LABEL      "_label"[0-9]*"*"  -> {LABEL ","}*
    "_const"[0-9]*"+"  -> {CONST ","}+

end module SyntaxConsts


module SyntaxTypes
%% The definitions in this module are according to the syntax as
%% specified in section 6.4.1, 6.4.2 and 6.4.3 in the ISO standard.
imports SyntaxConsts

exports
  sorts NEW-TYPE TYPE-DENOTER STRUCTURED-TYPE PACKED-STATUS FIELD-LIST
    FIXED-PART VAR-PART VARIANT RECORD-SECTION OPT-SEMI-COLON SELECTOR
    FIELDS-DESCRIPTION
  context-free syntax
                                                              -> PACKED-STATUS
    packed                                                    -> PACKED-STATUS
    IDENT                                                     -> TYPE-DENOTER
    NEW-TYPE                                                  -> TYPE-DENOTER
    CONST ".." CONST                                          -> NEW-TYPE
    "(" { IDENT "," }+ ")"                                    -> NEW-TYPE
    "^" IDENT                                                 -> NEW-TYPE
    PACKED-STATUS STRUCTURED-TYPE                             -> NEW-TYPE
    array "[" { TYPE-DENOTER "," }+ "]" of TYPE-DENOTER -> STRUCTURED-TYPE
    record FIELD-LIST end                                     -> STRUCTURED-TYPE
    set of TYPE-DENOTER                                       -> STRUCTURED-TYPE
    file of TYPE-DENOTER                                      -> STRUCTURED-TYPE

                                                              -> FIELD-LIST
    FIELDS-DESCRIPTION OPT-SEMI-COLON                         -> FIELD-LIST
```

```
                                                        -> OPT-SEMI-COLON
    ";"                                                 -> OPT-SEMI-COLON
    FIXED-PART                                          -> FIELDS-DESCRIPTION
    VAR-PART                                            -> FIELDS-DESCRIPTION
    FIXED-PART ";" VAR-PART                             -> FIELDS-DESCRIPTION
    { RECORD-SECTION ";" }+                             -> FIXED-PART
    { IDENT "," }+ ":" TYPE-DENOTER                     -> RECORD-SECTION
    case SELECTOR of { VARIANT ";" }+                   -> VAR-PART
    IDENT                                               -> SELECTOR
    IDENT ":" IDENT                                     -> SELECTOR
    { CONST ","}+ ":" "(" FIELD-LIST ")"                -> VARIANT

  variables
    "_type-den"[0-9]*                                   -> TYPE-DENOTER
    "_type-den"[0-9]*"*"                                -> {TYPE-DENOTER ","}*
    "_type-den"[0-9]*"+"                                -> {TYPE-DENOTER ","}+
    "_fields-descr"[0-9]*                               -> FIELDS-DESCRIPTION
    "_new-type"[0-9]*   -> NEW-TYPE     "_PS"           -> PACKED-STATUS
    "_fixed"[0-9]*      -> FIXED-PART   "_var-part"     -> VAR-PART
    "_variant"[0-9]*    -> VARIANT      "_variant+"     -> {VARIANT ";"}+
    "_selector"         -> SELECTOR     "_rec-section"  -> RECORD-SECTION
    "_field-list"       -> FIELD-LIST   "_rec-section+" -> {RECORD-SECTION ";"}+
                                        "opt;"          -> OPT-SEMI-COLON

end module SyntaxTypes


module SyntaxHeaders
%% Define Syntax formal parameters, ISO section 6.6.3.
%% (i.e. level 0).
imports SyntaxTypes

exports
  sorts FORMAL-PAR-LIST FORMAL-PAR-SECTION HEADING PROCEDURE-HEADING
    FUNCTION-HEADING FUNCTION-IDENTIFICATION LABELS CONST-DEF CONSTS
    TYPE-DEF TYPES VAR-DECL VARS

  context-free syntax
    PROCEDURE-HEADING                                   -> HEADING
    FUNCTION-HEADING                                    -> HEADING
    procedure IDENT FORMAL-PAR-LIST                     -> PROCEDURE-HEADING
    function  IDENT FORMAL-PAR-LIST ":" IDENT           -> FUNCTION-HEADING
    function  IDENT                                     -> FUNCTION-IDENTIFICATION
                                                        -> FORMAL-PAR-LIST
    "(" {FORMAL-PAR-SECTION  ";"}+ ")"                  -> FORMAL-PAR-LIST
    { IDENT "," }+ ":" IDENT                            -> FORMAL-PAR-SECTION
    var { IDENT "," }+ ":" IDENT                        -> FORMAL-PAR-SECTION
    PROCEDURE-HEADING                                   -> FORMAL-PAR-SECTION
    FUNCTION-HEADING                                    -> FORMAL-PAR-SECTION
                                                        -> LABELS
    label { LABEL "," }+ ";"                            -> LABELS
                                                        -> CONSTS
    const {CONST-DEF ";"}+ ";"                          -> CONSTS
    IDENT "=" CONST                                     -> CONST-DEF
                                                        -> TYPES
```

```
    type { TYPE-DEF ";" }+ ";"                        -> TYPES
    IDENT "=" TYPE-DENOTER                            -> TYPE-DEF
                                                      -> VARS
    var { VAR-DECL ";" }+ ";"                         -> VARS
    { IDENT "," }+ ":" TYPE-DENOTER                   -> VAR-DECL

  variables
    "_form-par-list"[0-9]*                            -> FORMAL-PAR-LIST
    "_form-par"[0-9]*                                 -> FORMAL-PAR-SECTION
    "_form-par"[0-9]*"+"                              -> {FORMAL-PAR-SECTION ";"}+
    "_form-par"[0-9]*"*"                              -> {FORMAL-PAR-SECTION ";"}*

end module SyntaxHeaders


module SyntaxExpr
%% Specify Pascal Expressions, ISO section 6.7.
%% A sligtly more general specification is given, allowing expressions like
%% 1 * -1 (instead of " 1 * (-1) " in ISO) as well.
imports SyntaxConsts

exports
  sorts VARIABLE-ACCESS MEMBER-DESIGNATOR SET-CONSTRUCTOR
 ACTUAL-PAR-LIST NON-EMPTY-ACT-PAR-LIST ACTUAL-PAR EXPR FORMAT

  context-free syntax
    IDENT                                             -> VARIABLE-ACCESS
    VARIABLE-ACCESS "[" { EXPR "," }+ "]"             -> VARIABLE-ACCESS
    VARIABLE-ACCESS "." IDENT                         -> VARIABLE-ACCESS
    VARIABLE-ACCESS "^"                               -> VARIABLE-ACCESS

    EXPR                                              -> MEMBER-DESIGNATOR
    EXPR ".." EXPR                                    -> MEMBER-DESIGNATOR
    "[" {MEMBER-DESIGNATOR ","}* "]"                  -> SET-CONSTRUCTOR

                                                      -> FORMAT
    ":" EXPR                                          -> FORMAT
    ":" EXPR ":" EXPR                                 -> FORMAT
    EXPR FORMAT                                       -> ACTUAL-PAR
    "(" { ACTUAL-PAR "," }+ ")"                       -> NON-EMPTY-ACT-PAR-LIST
    NON-EMPTY-ACT-PAR-LIST                            -> ACTUAL-PAR-LIST
                                                      -> ACTUAL-PAR-LIST

    VARIABLE-ACCESS                                   -> EXPR
    UNS-NUMBER                                        -> EXPR
    STRING                                            -> EXPR
    nil                                               -> EXPR
    IDENT NON-EMPTY-ACT-PAR-LIST                      -> EXPR
    SET-CONSTRUCTOR                                   -> EXPR
    "(" EXPR ")"                                      -> EXPR bracket

    not EXPR                                          -> EXPR
    "+" EXPR                                          -> EXPR
    "-" EXPR                                          -> EXPR
```

```
    EXPR "*" EXPR                                       -> EXPR left
    EXPR "/" EXPR                                       -> EXPR left
    EXPR div EXPR                                       -> EXPR left
    EXPR mod EXPR                                       -> EXPR left
    EXPR and EXPR                                       -> EXPR left
    EXPR "+" EXPR                                       -> EXPR left
    EXPR "-" EXPR                                       -> EXPR left
    EXPR or EXPR                                        -> EXPR left

    EXPR "=" EXPR                                       -> EXPR non-assoc
    EXPR "<>" EXPR                                      -> EXPR non-assoc
    EXPR "<" EXPR                                       -> EXPR non-assoc
    EXPR ">" EXPR                                       -> EXPR non-assoc
    EXPR "<=" EXPR                                      -> EXPR non-assoc
    EXPR ">=" EXPR                                      -> EXPR non-assoc
    EXPR in EXPR                                        -> EXPR non-assoc

  variables
  "_expr"[0-9]*    -> EXPR                 "_expr"[0-9]*"+"     -> {EXPR ","}+
  "_var-acc"[0-9]* -> VARIABLE-ACCESS      "_var-acc"[0-9]*"+"  -> {VARIABLE-ACCESS ","}+
  "_format"[0-9]*  -> FORMAT               "_act-par"[0-9]*     -> ACTUAL-PAR
  "_member"[0-9]*  -> MEMBER-DESIGNATOR    "_act-par"[0-9]*"+"  -> {ACTUAL-PAR ","}+
  "_act-par"[0-9]*"*"                      -> {ACTUAL-PAR ","}*
  "_act-par-list"[0-9]*                    -> ACTUAL-PAR-LIST
  "_NE-act-par-list"[0-9]*                 -> NON-EMPTY-ACT-PAR-LIST
  "_member"[0-9]*"*"                       -> { MEMBER-DESIGNATOR ","}*

priorities
  { not, "+" EXPR -> EXPR , "-" EXPR -> EXPR }  >
  { left: "*", "/", div, mod, and}  >
  { left: EXPR "+" EXPR -> EXPR, EXPR "-" EXPR -> EXPR, or } >
  { non-assoc: "=", "<>", "<", ">", "<=", ">=", in }

end module SyntaxExpr


module SyntaxStats
%% Specify statements of Pascal, ISO section 6.8.
%% module SyntaxStats
imports SyntaxExpr

exports
  sorts UNL-STAT SIMPLE-STAT STRUCT-STAT STATEMENT
    CASE-LIST-ELT COMP-STAT DOWN-TO
  context-free syntax
    LABEL ":" UNL-STAT                                  -> STATEMENT
    UNL-STAT                                            -> STATEMENT
    SIMPLE-STAT                                         -> UNL-STAT
    STRUCT-STAT                                         -> UNL-STAT

    VARIABLE-ACCESS ":=" EXPR                           -> SIMPLE-STAT
    IDENT ACTUAL-PAR-LIST                               -> SIMPLE-STAT
    goto LABEL                                          -> SIMPLE-STAT
                                                        -> SIMPLE-STAT
```

```
    COMP-STAT                                             -> STRUCT-STAT
    begin { STATEMENT ";" }+ end                          -> COMP-STAT
    if EXPR then STATEMENT                                -> STRUCT-STAT
    if EXPR then STATEMENT else STATEMENT                 -> STRUCT-STAT

    case EXPR of { CASE-LIST-ELT ";" }+ OPT-SEMI-COLON end -> STRUCT-STAT
    { CONST ","}+ ":" STATEMENT                           -> CASE-LIST-ELT

    repeat { STATEMENT ";"}+ until EXPR                   -> STRUCT-STAT
    while EXPR do STATEMENT                               -> STRUCT-STAT
    "to"                                                  -> DOWN-TO
    downto                                                -> DOWN-TO
    for IDENT ":=" EXPR DOWN-TO EXPR do STATEMENT         -> STRUCT-STAT
    with { VARIABLE-ACCESS "," }+ do STATEMENT            -> STRUCT-STAT

priorities
    if EXPR then STATEMENT else STATEMENT -> STRUCT-STAT
    >
    if EXPR then STATEMENT -> STRUCT-STAT
end module SyntaxStats


module SyntaxProgram
%% Define programs and blocks, according to ISO section 6.2.1 and 6.10
imports SyntaxHeaders SyntaxStats

exports
  sorts PROC-OR-FUN PROCS-OR-FUNS  DEFS DECLS BLOCK PROGRAM PROGRAM-HEADER
    PROGRAM-PARAMS
  context-free syntax
    HEADING ";" forward                                    -> PROC-OR-FUN
    HEADING ";" BLOCK                                      -> PROC-OR-FUN
    FUNCTION-IDENTIFICATION ";" BLOCK                      -> PROC-OR-FUN

                                                           -> PROCS-OR-FUNS
    { PROC-OR-FUN ";" }+  ";"                              -> PROCS-OR-FUNS

    DEFS DECLS COMP-STAT                                   -> BLOCK
    LABELS CONSTS TYPES                                    -> DEFS
    VARS PROCS-OR-FUNS                                     -> DECLS

                                                           -> PROGRAM-PARAMS
    "(" { IDENT "," }+ ")"                                 -> PROGRAM-PARAMS
    program IDENT PROGRAM-PARAMS                           -> PROGRAM-HEADER
    PROGRAM-HEADER ";" BLOCK "."                          -> PROGRAM
end module SyntaxProgram
```

C.2 Miscellaneous

```
module LayOut
exports
  lexical syntax
```

```
    [ \t\n]     -> LAYOUT              "%%" ~[\n]* [\n] -> LAYOUT
end module LayOut


module Booleans
imports LayOut

exports
  sorts BOOLEAN
  context-free syntax
    "TRUE"                                          -> BOOLEAN
    "FALSE"                                         -> BOOLEAN
    BOOLEAN "|" BOOLEAN                             -> BOOLEAN assoc
    BOOLEAN "&" BOOLEAN                             -> BOOLEAN assoc
    "(" BOOLEAN ")"                                 -> BOOLEAN bracket

priorities  "|" < "&"

hiddens
  variables  b -> BOOLEAN

equations
  [1]   TRUE | b  = TRUE                            [2]   FALSE | b = b
  [3]   b | TRUE = TRUE                             [4]   b | FALSE = FALSE
  [5]   TRUE & b  = b                               [6]   FALSE & b = FALSE
  [7]   b & TRUE = b                                [8]   b & FALSE = FALSE
end module Booleans


module Integers
imports Booleans SyntaxNumbers

exports
  sorts NAT INT

  context-free syntax
    UNS-INT       -> NAT              NAT            -> INT
    "+" NAT       -> INT              "-" NAT        -> INT
    INT "+" INT   -> INT {left}       INT "-" INT    -> INT {left}
    INT "*" INT   -> INT {left}       INT ">" INT    -> BOOLEAN
    INT "<=" INT  -> BOOLEAN          "(" NAT ")"    -> NAT {bracket}
    "(" INT ")"   -> INT {bracket}

hiddens
  sorts UNS-INT-LIST
  context-free syntax
    "{" { UNS-INT "," } * "}"                       -> UNS-INT-LIST
    UNS-INT-LIST "right" DIGIT                      -> UNS-INT-LIST
    UNS-INT-LIST "left" DIGIT                       -> UNS-INT-LIST
    pred "(" DIGIT ")"                              -> DIGIT
    non-zero "(" NAT ")"                            -> BOOLEAN

  variables
    n [0-9]*      -> UNS-INT          uns-ints [0-9]* -> {UNS-INT ","} *
    c [0-9]*      -> CHAR             d [0-9]*       -> DIGIT
```

```
   x [0-9]*       -> CHAR *            y [0-9]*        -> CHAR +
   Nat [0-9']*    -> NAT               Int [0-9']*     -> INT

priorities
  {left: INT "+" INT -> INT, INT "-" INT -> INT }  <  "*"

equations

  [01]  pred(0) = 0             [02] pred(1) = 0        [03] pred(2) = 1
  [04]  pred(3) = 2             [05]  pred(4) = 3       [06] pred(5) = 4
  [07]  pred(6) = 5             [08] pred(7) = 6        [09] pred(8) = 7
  [10]  pred(9) = 8

  %% -- left and right rotation --
  [11]  d  = 0 ==> {uns-ints} left d = {uns-ints}
  [12]  d != 0 ==> {n, uns-ints} left d = {uns-ints, n} left pred(d)

  [13]  d  = 0 ==> {uns-ints} right d = {uns-ints}
  [14]  d != 0 ==> {uns-ints, n} right d = {n, uns-ints} right pred(d)
  [15]  n != 0 ==> non-zero(n) = TRUE
  [16]  n = 0  ==> non-zero(n) = FALSE

  [17]  uns-int("0" y) = uns-int(y)

  %% -- addition --
  [18]  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}
          left digit(c1) left digit(c2) = {n, uns-ints}
        ===============================================
        uns-int(c1) + uns-int(c2) = n

  [19]  uns-int(c1) + uns-int(c2) = uns-int(x c),
        uns-int ("0" x1) + uns-int("0" x2) + uns-int("0" x) = uns-int(y)
        ===============================================================
        uns-int(x1 c1) + uns-int(x2 c2) = uns-int(y c)

  %% -- multiplication --
  [20]  n * 0 = 0
  [21]  pred(digit(c1)) = digit(c2)
        ====================================
        n * uns-int(c1) = n + n * uns-int(c2)

  [22]  uns-int(y1) * uns-int (y2) = uns-int(y)
        ===================================================================
        uns-int(y1) * uns-int(y2 c) = uns-int(y "0") + uns-int(y1) * uns-int(c)

  %% -- greater than --
  [23]  {0,1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0}
          left digit(c1) right digit(c2) = {n, uns-ints}
        =====================================================
        uns-int(c1) > uns-int(c2) = non-zero(n)

  [24]  uns-int(y1 c1) > uns-int(c2) = TRUE
  [25]  uns-int(c1) > uns-int(y2 c2) = FALSE
  [26]  uns-int(y c1) > uns-int(y c2) = uns-int(c1) > uns-int(c2)
```

```
  [27]  uns-int(y1) > uns-int(y2) = TRUE
        ======================================
        uns-int(y1 c1) > uns-int(y2 c2) = TRUE

  [28]  uns-int(y1) > uns-int(y2) = FALSE, uns-int(y1) != uns-int(y2)
        ============================================================
        uns-int(y1 c1) > uns-int(y2 c2) = FALSE

  %% -- less than or equal --
  [l1]  Nat1 <= Nat2 = TRUE when Nat2 > Nat1 = TRUE
  [l2]  Nat1 <= Nat1 = TRUE


  %% --  subtraction of naturals --
  [28]  {10,11,12,13,14,15,16,17,18,19,0,1,2,3,4,5,6,7,8,9}
          left digit(c1) right digit(c2) = {n, uns-ints}
        ====================================================
        uns-int("1" c1) - uns-int(c2) = n

  [29]  {0,1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0}
          left digit(c1) right digit(c2) = {n, uns-ints}
        ================================================
        uns-int(c1) - uns-int(c2) = n

  [30]  uns-int(x1 c1) > uns-int(x2 c2) = TRUE,
        uns-int("1" c1) - uns-int(c2) = uns-int(x c),
        (uns-int("0" x) + uns-int("0" x1)) - uns-int("0" x2) - 1 = uns-int(y)
        ===================================================================
        uns-int(x1 c1) - uns-int(x2 c2) = uns-int(y c)

  [31]  Nat1 > Nat2 = FALSE, Nat2 - Nat1 = Nat  ==> Nat1 - Nat2 = -Nat


  %% -- addition and subtraction of integers
  [34]  Int1 = +Nat1                            ==> Int1 = Nat1

  [35]  Int1 =  Nat1, Int2 =  Nat2              ==> Int1 + Int2 = Nat1 + Nat2
  [36]  Int1 =  Nat1, Int2 = -Nat2              ==> Int1 + Int2 = Nat1 - Nat2
  [37]  Int1 = -Nat1, Int2 =  Nat2              ==> Int1 + Int2 = Nat2 - Nat1
  [38]  Int1 = -Nat1, Int2 = -Nat2, Nat1 + Nat2 = Nat
                                                ==> Int1 + Int2 = -Nat

  [39]  Int1 =  Nat1, Int2 =  Nat2              ==> Int1 - Int2 = Nat1 - Nat2
  [40]  Int1 =  Nat1, Int2 = -Nat2              ==> Int1 - Int2 = Nat1 + Nat2
  [41]  Int1 = -Nat1, Int2 =  Nat2, Nat1 + Nat2 = Nat
                                                ==> Int1 - Int2 = -Nat
  [42]  Int1 = -Nat1, Int2 = -Nat2              ==> Int1 - Int2 = Nat2 - Nat1

  [43]  Int1 =  Nat1, Int2 =  Nat2              ==> Int1 * Int2 = Nat1 * Nat2
  [44]  Int1 =  Nat1, Int2 = -Nat2, Nat1 * Nat2 = Nat
                                                ==> Int1 * Int2 = -Nat
  [45]  Int1 = -Nat1, Int2 =  Nat2, Nat1 * Nat2 = Nat
                                                ==> Int1 * Int2 = -Nat
  [46]  Int1 = -Nat1, Int2 = -Nat2              ==> Int1 * Int2 = Nat1 * Nat2
end module Integers
```

```
module StringFunctions
imports Integers SyntaxConsts

exports
  context-free syntax
    length "(" STRING ")"                              -> INT

hiddens
  variables
    a-char      -> CHAR                         chars   -> CHAR*

equations

  [s1]  length( '' ) = 0

  [s2]  length( string( "'" "'" "'" chars "'" ))  =
          length(string("'" chars "'")) + 1

  [s3]  string( "'" a-char "'") != string("'" "'" "'")
        ===================================================
        length(string("'" a-char chars "'" )) =
          length(string("'" chars "'")) + 1
end module StringFunctions


module ASCII
imports Integers SyntaxConsts StringFunctions

exports
  context-free syntax
    ascii-val "(" STRING ")"                           -> INT

equations

  { Ordinal numbers for characters. }
  %% program GenerateAsciiTable(input,output);
  %% var i: integer;
  %% begin
  %%     for i := 32 to 127 do
  %%     begin
  %%         write('  [',i:3,']',' ascii-val(''', chr(i), ''')=', i:3);
  %%         if (i+1) mod 3 = 0 then writeln;
  %%    end
  %% end.

  [ 32] ascii-val(' ')= 32  [ 33] ascii-val('!')= 33  [ 34] ascii-val('"')= 34
  [ 35] ascii-val('#')= 35  [ 36] ascii-val('$')= 36  [ 37] ascii-val('%')= 37
  [ 38] ascii-val('&')= 38  [ 39] ascii-val('''')=39  [ 40] ascii-val('(')= 40
  [ 41] ascii-val(')')= 41  [ 42] ascii-val('*')= 42  [ 43] ascii-val('+')= 43
  [ 44] ascii-val(',')= 44  [ 45] ascii-val('-')= 45  [ 46] ascii-val('.')= 46
  [ 47] ascii-val('/')= 47  [ 48] ascii-val('0')= 48  [ 49] ascii-val('1')= 49
  [ 50] ascii-val('2')= 50  [ 51] ascii-val('3')= 51  [ 52] ascii-val('4')= 52
  [ 53] ascii-val('5')= 53  [ 54] ascii-val('6')= 54  [ 55] ascii-val('7')= 55
```

```
  [ 56] ascii-val(’8’)= 56  [ 57] ascii-val(’9’)= 57  [ 58] ascii-val(’:’)= 58
  [ 59] ascii-val(’;’)= 59  [ 60] ascii-val(’<’)= 60  [ 61] ascii-val(’=’)= 61
  [ 62] ascii-val(’>’)= 62  [ 63] ascii-val(’?’)= 63  [ 64] ascii-val(’@’)= 64
  [ 65] ascii-val(’A’)= 65  [ 66] ascii-val(’B’)= 66  [ 67] ascii-val(’C’)= 67
  [ 68] ascii-val(’D’)= 68  [ 69] ascii-val(’E’)= 69  [ 70] ascii-val(’F’)= 70
  [ 71] ascii-val(’G’)= 71  [ 72] ascii-val(’H’)= 72  [ 73] ascii-val(’I’)= 73
  [ 74] ascii-val(’J’)= 74  [ 75] ascii-val(’K’)= 75  [ 76] ascii-val(’L’)= 76
  [ 77] ascii-val(’M’)= 77  [ 78] ascii-val(’N’)= 78  [ 79] ascii-val(’O’)= 79
  [ 80] ascii-val(’P’)= 80  [ 81] ascii-val(’Q’)= 81  [ 82] ascii-val(’R’)= 82
  [ 83] ascii-val(’S’)= 83  [ 84] ascii-val(’T’)= 84  [ 85] ascii-val(’U’)= 85
  [ 86] ascii-val(’V’)= 86  [ 87] ascii-val(’W’)= 87  [ 88] ascii-val(’X’)= 88
  [ 89] ascii-val(’Y’)= 89  [ 90] ascii-val(’Z’)= 90  [ 91] ascii-val(’[’)= 91
  [ 92] ascii-val(’\’)= 92  [ 93] ascii-val(’]’)= 93  [ 94] ascii-val(’^’)= 94
  [ 95] ascii-val(’_’)= 95  [ 96] ascii-val(’‘’)= 96  [ 97] ascii-val(’a’)= 97
  [ 98] ascii-val(’b’)= 98  [ 99] ascii-val(’c’)= 99  [100] ascii-val(’d’)=100
  [101] ascii-val(’e’)=101  [102] ascii-val(’f’)=102  [103] ascii-val(’g’)=103
  [104] ascii-val(’h’)=104  [105] ascii-val(’i’)=105  [106] ascii-val(’j’)=106
  [107] ascii-val(’k’)=107  [108] ascii-val(’l’)=108  [109] ascii-val(’m’)=109
  [110] ascii-val(’n’)=110  [111] ascii-val(’o’)=111  [112] ascii-val(’p’)=112
  [113] ascii-val(’q’)=113  [114] ascii-val(’r’)=114  [115] ascii-val(’s’)=115
  [116] ascii-val(’t’)=116  [117] ascii-val(’u’)=117  [118] ascii-val(’v’)=118
  [119] ascii-val(’w’)=119  [120] ascii-val(’x’)=120  [121] ascii-val(’y’)=121
  [122] ascii-val(’z’)=122  [123] ascii-val(’{’)=123  [124] ascii-val(’|’)=124
  [125] ascii-val(’}’)=125  [126] ascii-val(’~’)=126

  [127]  ascii-val(_string) = -1 when length(_string) != 1

end module ASCII
```

C.3 Contexts

```
module Contexts
%% Specify the context containing all declarations found so far.
imports  SyntaxHeaders

exports
  sorts CONTEXT ENTRY SYMBOL VAR-STATUS

  context-free syntax
    possibly-threatening                             -> VAR-STATUS
    control-variable                                 -> VAR-STATUS
                                                     -> VAR-STATUS

    label LABEL                                      -> ENTRY
    const CONST-DEF                                  -> ENTRY
    type TYPE-DEF                                    -> ENTRY
    var VAR-DECL VAR-STATUS                          -> ENTRY
    HEADING                                          -> ENTRY
    predefined-function IDENT                        -> ENTRY
    predefined-procedure IDENT                       -> ENTRY
    empty-set                                        -> ENTRY
    nil-pointer                                      -> ENTRY
```

```
    block-mark                                              -> ENTRY
    error-entry                                             -> ENTRY

    IDENT                                                   -> SYMBOL
    LABEL                                                   -> SYMBOL
    block-mark                                              -> SYMBOL

    "[" { ENTRY "," }* "]"                                  -> CONTEXT
    "(" CONTEXT ")"                                         -> CONTEXT bracket
    predefined-block    -> CONTEXT      init-block      -> CONTEXT
    integer-type        -> CONTEXT      boolean-type    -> CONTEXT
    char-type           -> CONTEXT      text-type       -> CONTEXT
    real-type           -> CONTEXT

  variables
    Context[0-9]*       -> CONTEXT      Entry[0-9]*     -> ENTRY
    [TC][0-9]*          -> CONTEXT      Entry[0-9]*"*"  -> { ENTRY ","}*
    "_symbol"[0-9]*     -> SYMBOL       Prefix[0-9]*    -> { ENTRY ","}*
    Postfix[0-9]*       -> {ENTRY ","}*

equations

  [i0]  init-block = []

  [i1]  predefined-block =
         [block-mark,

          predefined-procedure rewrite,     predefined-procedure put,
          predefined-procedure reset,       predefined-procedure get,
          predefined-procedure read,        predefined-procedure write,
          predefined-procedure readln,      predefined-procedure writeln,
          predefined-procedure page,        predefined-procedure new,
          predefined-procedure dispose,     predefined-procedure pack,
          predefined-procedure unpack,

          predefined-function abs,          predefined-function sqr,
          predefined-function trunc,        predefined-function round,
          predefined-function ord,          predefined-function succ,
          predefined-function pred,         predefined-function eof,
          predefined-function eoln,

          type integer = integer,           type real = real,
          type char = char,                 type Boolean = (false, true),
          type text = file of char,

          function sin (x: real): real,     function cos (x: real): real,
          function exp (x: real): real,     function ln (x: real): real,
          function sqrt (x: real): real,    function arctan (x: real): real,
           function chr (x: integer): char,  function odd (x: integer): Boolean,

          const maxint = 32768,

          block-mark
         ]
end module Contexts
```

```
module Context-defs
imports Contexts Booleans

exports
  sorts ENUM-OUT FIXED-OUT VAR-OUT
  context-free syntax
    CONTEXT is definition-point for SYMBOL "?"          -> BOOLEAN
    CONTEXT defines SYMBOL as a label "?"               -> BOOLEAN
    CONTEXT defines SYMBOL as a constant "?"            -> BOOLEAN
    CONTEXT defines SYMBOL as a type "?"                -> BOOLEAN
    CONTEXT defines SYMBOL as a variable "?"            -> BOOLEAN
    CONTEXT defines SYMBOL as a function "?"            -> BOOLEAN
    CONTEXT defines SYMBOL as a procedure "?"           -> BOOLEAN

    CONTEXT defines SYMBOL as a normal constant "?"     -> BOOLEAN
    CONTEXT defines SYMBOL as an enumeration constant "?" -> BOOLEAN
    CONTEXT defines SYMBOL as a normal function "?"     -> BOOLEAN
    CONTEXT defines SYMBOL as a predefined function "?" -> BOOLEAN
    CONTEXT defines SYMBOL as a normal procedure "?"    -> BOOLEAN
    CONTEXT defines SYMBOL as a predefined procedure "?"  -> BOOLEAN
    CONTEXT defines SYMBOL as a block-mark "?"          -> BOOLEAN

    "<" BOOLEAN "," ENTRY ">"                           -> ENUM-OUT
    CONTEXT defines SYMBOL as an enum const             -> ENUM-OUT
    VAR-OUT defines SYMBOL as an enum const in CONTEXT -> ENUM-OUT
    FIXED-OUT defines SYMBOL as an enum const in CONTEXT -> ENUM-OUT
    ENUM-OUT "|" ENUM-OUT                               -> ENUM-OUT {left}

    is-record "?" "(" CONTEXT ")"                       -> BOOLEAN
    CONTEXT "." fixed                                   -> FIXED-OUT
    CONTEXT "." variant                                 -> VAR-OUT
    FIXED-PART                                          -> FIXED-OUT
    VAR-PART                                            -> VAR-OUT

hiddens
  variables
    Enum[?]       -> BOOLEAN            E-out           -> ENUM-OUT

equations

  [dp1] C is definition-point for _symbol? =
        C defines _symbol as a label?     | C defines _symbol as a constant?  |
        C defines _symbol as a type?      | C defines _symbol as a variable?  |
        C defines _symbol as a function?  | C defines _symbol as a procedure? |
        C defines _symbol as a block-mark?

  [de1] [Prefix, label _label] defines _label as a label? = TRUE
  [de2] [Prefix, const _id = _const] defines _id as a normal constant? = TRUE
  [de3] [Prefix, type _id = _type-den] defines _id as a type? = TRUE
  [de4] [Prefix, var _id : _type-den] defines _id as a variable? = TRUE

  [de5] [Prefix, function _id _form-par-list: _id2]
        defines _id as a normal function? = TRUE
```

```
[de6] [Prefix, predefined-function _id]
        defines _id as a predefined function? = TRUE
[d11] C defines _id as a function? =
        C defines _id as a normal function? |
        C defines _id as a predefined function?

[de7] [Prefix, procedure _id _form-par-list]
        defines _id as a procedure? = TRUE
[de8] [Prefix, predefined-procedure _id]
        defines _id as a procedure? = TRUE
[d12] C defines _id as a procedure? =
        C defines _id as a normal procedure?    |
        C defines _id as a predefined procedure?

[de9] C defines _id as a constant? =
        C defines _id as a normal constant?     |
        C defines _id as an enumeration constant?

[d13] [Prefix, block-mark] defines block-mark as a block-mark? = TRUE


{ Enumeration constants: New enumeration constants may be defined in each
  occurence of a _new-type in the entry. }
[eo1] <TRUE, Entry> | E-out  = <TRUE, Entry>
[eo2] E-out | <TRUE, Entry> = <TRUE, Entry>


[en1] C defines _id as an enum const = <Enum?, Entry>
      ============================================
      C defines _id as an enumeration constant? = Enum?

[en2] [Prefix, type _id1 = (_id1*, _id2, _id2*)] defines _id2 as an
        enum const = <TRUE, type _id1 = (_id1*, _id2, _id2*)>

[en3] [Prefix, var _id2: _new-type] defines _id1 as an enum const =
        [Prefix, type Generated = _new-type] defines _id1 as an enum const

[en4] [Prefix, type _id2 = _PS array [_type-den1] of _type-den2]
        defines _id1 as an enum const =
         [Prefix, type Generated = _type-den1] defines _id1 as an enum const |
         [Prefix, type Generated = _type-den2] defines _id1 as an enum const

[en5] [Prefix, type _id2 = _PS file of _new-type]
        defines _id1 as an enum const =
          [Prefix, type Generated = _new-type] defines _id1 as an enum const

[en6] [Prefix, type _id2 = _PS set of _new-type]
         defines _id1 as an enum const =
           [Prefix, type Generated = _new-type] defines _id1 as an enum const

[en7] T1 defines _id as an enum const =
        T1.fixed defines _id as an enum const in T1  |
        T1.variant defines _id as an enum const in T1
          when is-record?(T1) = TRUE
```

```
  [fi1] _id+ : _new-type defines _id as an enum const in [Prefix] =
          [Prefix, type Generated = _new-type] defines _id as an enum const

  [fi2] _rec-section; _rec-section+ defines _id as an enum const in Context =
          _rec-section defines _id as an enum const in Context |
            _rec-section+ defines _id as an enum const in Context

  [va1] case _selector of _const+: ( _field-list )
          defines _id as an enum const in [Prefix, Entry] =
            [Prefix, type Generated = record _field-list end]
              defines _id as an enum const

  [va2] case _selector of _variant; _variant+
          defines _id as an enum const in Context =
            case _selector of _variant defines _id as an enum const in Context
            | case _selector of _variant+ defines _id as an enum const in Context

  [re1] is-record?([Prefix, type _id = _PS record _fields-descr opt; end]) = TRUE
  [re2] [Prefix, type _id = _PS record _fixed ; _var-part opt; end] . variant = _var-part
  [re3] [Prefix, type _id = _PS record _fixed ; _var-part opt; end] . fixed = _fixed
  [re4] [Prefix, type _id = _PS record _fixed opt; end] . fixed = _fixed
  [re5] [Prefix, type _id = _PS record _var-part opt; end] . variant = _var-part
end module Context-defs


module Context-search
%% Define search functions on contexts.
imports Context-defs

exports
  sorts COMPLETE-CONTEXT
  context-free syntax
    "<" CONTEXT "," ENTRY "," CONTEXT ">"              -> COMPLETE-CONTEXT
    complete find SYMBOL in COMPLETE-CONTEXT           -> COMPLETE-CONTEXT
    complete find SYMBOL in CONTEXT                    -> COMPLETE-CONTEXT

    add ENTRY "to" CONTEXT                             -> CONTEXT
    delete inner block of CONTEXT                      -> CONTEXT
    find SYMBOL in CONTEXT                             -> CONTEXT
    find SYMBOL in inner block of CONTEXT              -> CONTEXT

    SYMBOL exists in CONTEXT "?"                       -> BOOLEAN
    SYMBOL exists in inner block of CONTEXT "?"        -> BOOLEAN


equations

  [a1]  add Entry to [Prefix] = [Prefix, Entry]

  [cf0] complete find _symbol in <C, error-entry, []> = <C1, Entry, C2>,
        [Entry] is definition-point for _symbol? = TRUE
        ===============================================
        complete find _symbol in C = <C1, Entry, C2>

  { Avoid having to take the predefined identifiers always with you,
```

```
  only look in predefined identifiers if normal search fails. }
[cf0'] complete find _symbol in <[Entry1*], error-entry, []> =
              <[], error-entry, [Entry1*]>,
      complete find _symbol in predefined-block =
              <[Entry2*], Entry, [Entry3*]>
      ======================================================================
      complete find _symbol in [Entry1*] =
          <[Entry2*], Entry, [Entry3*, Entry1*]>

[cf1] [Entry] is definition-point for _symbol? = TRUE
      =============================================
      complete find _symbol in <[], Entry, C2> = <[], Entry, C2>

[cf2] [Entry] is definition-point for _symbol? != TRUE
      =============================================
      complete find _symbol in <[], Entry, C2> = <[], error-entry, C2>

[cf3] [Prefix, Entry] is definition-point for _symbol? = TRUE
      ==========================================
      complete find _symbol in <[Prefix], Entry, [Postfix]> =
              <[Prefix], Entry, [Postfix]>

[cf4] [Prefix, Entry2, Entry1] is definition-point for _symbol? != TRUE
      =======================================================
      complete find _symbol in <[Prefix, Entry2], Entry1, [Postfix]> =
        complete find _symbol in <[Prefix], Entry2, [Entry1, Postfix]>


[f1]  complete find _symbol in C = <[Prefix], Entry, C2>
      ==================================================
      find _symbol in C = [Prefix, Entry]

[dl1] complete find block-mark in C = <[Prefix], Entry, C2>
      ====================================================
      delete inner block of C = [Prefix]

[fi1] complete find block-mark in C1 = <[Prefix], block-mark, C2>,
      find _symbol in C2 = [Entry*]
      ================================
      find _symbol in inner block of C1 = [Prefix, block-mark, Entry*]

[ex1] find _symbol in C = [Prefix, Entry],
      Entry != error-entry
      ======================
      _symbol exists in C? = TRUE

[ex2] find _symbol in inner block of C = [Prefix, Entry],
      Entry != error-entry
      ======================
      _symbol exists in inner block of C? = TRUE


[pd1] integer-type = find integer in init-block
[pd2] boolean-type = find Boolean in init-block
[pd3] char-type = find char in init-block
```

```
  [pd4] text-type = find text in init-block
  [pd5] real-type = find real in init-block
end module Context-search


module ConstantAnalysis
%% Analyze elementary constants.
imports Context-search ASCII

exports
  context-free syntax
    new-const-type of NEW-CONST                          -> CONTEXT
    new-const-val of NEW-CONST                           -> INT
    id-number of IDENT in "(" { IDENT ","}+ ")"          -> INT
    is-string-const "?" "(" CONST ")"                    -> BOOLEAN

hiddens
  context-free syntax
    id-number "(" "(" { IDENT ","}* ")" "," IDENT "," INT ")"  -> INT

equations

  [v1] new-const-val of   _uns-int = _uns-int
  [v2] new-const-val of + _uns-int = _uns-int
  [v3] new-const-val of - _uns-int = -1 *_uns-int
  [v4] new-const-val of   _uns-real = -1
  [v5] new-const-val of + _uns-real = -1
  [v6] new-const-val of - _uns-real = -1
  [v7] new-const-val of _string = ascii-val(_string)

  [t1] new-const-type of       _uns-int  = integer-type
  [t2] new-const-type of _sign _uns-int  = integer-type
  [t3] new-const-type of       _uns-real = real-type
  [t4] new-const-type of _sign _uns-real = real-type
  [t5] new-const-type of       _string   = char-type when length(_string) = 1

  [t6] length(_string) = _uns-int,
       _uns-int != 1
       ==========================
       new-const-type of _string = add
         type Generated = packed array [ 1 .. _uns-int ] of char
        to init-block

  [i1] id-number of _id in (_id1, _id*) = id-number((_id1, _id*), _id, 0)
  [i2] id-number((), _id, _uns-int) = -1
  [i3] id-number((_id, _id*), _id, _uns-int) = _uns-int
  [i4] id-number((_id1, _id*), _id, _uns-int) =
                id-number((_id*), _id, _uns-int + 1 )
            when _id != _id1

  [i5] is-string-const?(_string) = TRUE
end module ConstantAnalysis


module Context-sen-funs
```

```
%% Specify contexts sensitive definitions
imports Context-search ConstantAnalysis

exports
  sorts FORMAL-PAR-LIST-OUT
  context-free syntax
    is-enumeration-type "?" "(" CONTEXT ")"            -> BOOLEAN
    is-subrange-type "?" "(" CONTEXT ")"               -> BOOLEAN
    is-array-type "?" "(" CONTEXT ")"                  -> BOOLEAN
    is-string-type "?" "(" CONTEXT ")"                 -> BOOLEAN
    is-set-type "?" "(" CONTEXT ")"                    -> BOOLEAN
    is-empty-set-type "?" "(" CONTEXT ")"              -> BOOLEAN
    is-non-empty-set-type "?" "(" CONTEXT ")"          -> BOOLEAN
    is-file-type "?" "(" CONTEXT ")"                   -> BOOLEAN
    is-record-type "?" "(" CONTEXT ")"                 -> BOOLEAN
    is-pointer-type "?" "(" CONTEXT ")"                -> BOOLEAN
    is-nil-pointer-type "?" "(" CONTEXT ")"            -> BOOLEAN
    is-nonnil-pointer-type "?" "(" CONTEXT ")"         -> BOOLEAN
    is-packed "?" "(" CONTEXT ")"                      -> BOOLEAN
    is-simple-type "?" "(" CONTEXT ")"                 -> BOOLEAN
    is-ordinal-type "?" "(" CONTEXT ")"                -> BOOLEAN
    is-number-type "?" "(" CONTEXT ")"                 -> BOOLEAN
    is-possibly-threatening "?" "(" CONTEXT ")"        -> BOOLEAN
    is-control-variable "?" "(" CONTEXT ")"            -> BOOLEAN
    not-permissable-as-file-comp "?" "(" CONTEXT ")"   -> BOOLEAN
    has-variant-part "?" "(" CONTEXT ")"               -> BOOLEAN
    selector-field "?" "(" CONTEXT "," IDENT ")"       -> BOOLEAN

    CONTEXT "." record-of-variant-part                 -> CONTEXT
    CONTEXT "." selector-type                          -> CONTEXT
    CONTEXT "." entry                                  -> ENTRY
    FORMAL-PAR-LIST                                    -> FORMAL-PAR-LIST-OUT
    CONTEXT "." formal-parameters                      -> FORMAL-PAR-LIST-OUT
    CONTEXT "." index-type                             -> CONTEXT
    CONTEXT "." comp-type                              -> CONTEXT
    CONTEXT "." host-type                              -> CONTEXT
    CONTEXT "." type                                   -> CONTEXT
    CONTEXT "." canonical                              -> CONTEXT
    CONTEXT "." value                                  -> INT
    CONTEXT "." nr-of-elements                         -> INT
    pointer-component of CONTEXT in CONTEXT            -> CONTEXT
    CONTEXT is the set-component of CONTEXT "?"        -> BOOLEAN

    get-type of TYPE-DENOTER in CONTEXT                -> CONTEXT
    const-type of CONST in CONTEXT                     -> CONTEXT
    const-val of CONST in CONTEXT                      -> INT

    type-to-var "(" CONTEXT ")"                        -> CONTEXT
    set var-status of IDENT in CONTEXT "to" VAR-STATUS  -> CONTEXT
    IDENT is the name of the inner function-block in CONTEXT "?" -> BOOLEAN

    CONTEXT is the same as CONTEXT "?"                 -> BOOLEAN
    same-set-of-T "?" "(" CONTEXT "," CONTEXT ")"      -> BOOLEAN

hiddens
```

```
  variables
    Index-type  -> TYPE-DENOTER         Comp-type        -> TYPE-DENOTER
    Struct-type -> STRUCTURED-TYPE      VS[0-9]*         -> VAR-STATUS

equations

  { Classification  of new types }
  [nt1] is-subrange-type?([Prefix, type _id = _const1 .. _const2]) = TRUE
  [nt2] is-enumeration-type?([Prefix, type _id = (_id*)]) = TRUE
  [nt3] is-array-type?(
          [Prefix, type _id = _PS array[_type-den1] of _type-den2]) = TRUE
   [nt4] is-non-empty-set-type?([Prefix, type _id = _PS set of _type-den])= TRUE
  [nt5] is-empty-set-type?([Prefix, empty-set]) = TRUE
  [nt5] is-set-type?(T) = is-non-empty-set-type?(T) | is-empty-set-type?(T)
  [nt6] is-file-type?([Prefix, type _id = _PS file of _type-den]) = TRUE
  [nt7] is-nonnil-pointer-type?([Prefix, type _id = ^_id]) = TRUE
  [nt8] is-nil-pointer-type?([Prefix, nil-pointer]) = TRUE
  [nt9] is-pointer-type?(T) =
          is-nil-pointer-type?(T) | is-nonnil-pointer-type?(T)
  [ip1] is-packed?([Prefix, type _id = packed Struct-type]) = TRUE

  { Accessing components of types: }
  [ac1] [Prefix, type _id = _PS array[ Index-type ] of Comp-type].index-type =
          get-type of Index-type in [Prefix]
  [ac2] [Prefix, type _id = _PS array[ Index-type ] of Comp-type].comp-type =
          get-type of Comp-type in [Prefix]
  [ac3] [Prefix, type _id = _PS set of Comp-type].comp-type =
          get-type of Comp-type in [Prefix]
  [ac4] [Prefix, type _id = _PS file of Comp-type].comp-type =
          get-type of Comp-type in [Prefix]
  [ac5] [Prefix, type _id = _const1 .. _const2].host-type =
          const-type of _const1 in [Prefix]
  [ac6] pointer-component of [Prefix, type _id1 = ^_id2] in Context =
          get-type of _id2 in Context
  [ce]  [Prefix, Entry].entry = Entry

  [ct1] T.canonical = T.host-type when is-subrange-type?(T) = TRUE
  [ct2] T.canonical = T when is-subrange-type?(T) != TRUE

  [no1] T.nr-of-elements =  2 *  const-val of maxint in init-block
          when T = integer-type

  [no2] [Prefix, type _id = _const1 .. _const2] . nr-of-elements =
          const-val of _const2 in [Prefix]  -
          const-val of _const1 in [Prefix]  +  1

  [no3] [Prefix, type _id1 = (_id2)] . nr-of-elements = 1
  [no4] [Prefix, type _id1 = (_id2, _id*)] . nr-of-elements =
          1 + [Prefix, type _id1 = (_id*)] . nr-of-elements

  [no5] T.nr-of-elements = ascii-val('~') - ascii-val(' ') + 1
          when T = char-type


  { Grouping types }
```

```
[st1] is-simple-type?(T)  = TRUE when T = real-type
[st2] is-simple-type?(T)  = TRUE when is-ordinal-type?(T) = TRUE
[ot1] is-ordinal-type?(T) = TRUE when T = integer-type
[ot2] is-ordinal-type?(T) = TRUE when T = char-type
[ot3] is-ordinal-type?(T) = TRUE when T = boolean-type
[ot4] is-ordinal-type?(T) = TRUE when is-subrange-type?(T) = TRUE
[ot5] is-ordinal-type?(T) = TRUE when is-enumeration-type?(T) = TRUE
[nt1] is-number-type?(T) = TRUE when T = real-type
[nt2] is-number-type?(T) = TRUE when T = integer-type
[sr1] is-string-type?(T) = TRUE when
        is-array-type?(T) = TRUE,     is-packed?(T) = TRUE,
        T.host-type = char-type
[re1] has-variant-part?([Prefix,type _id= record _var-part opt; end]) = TRUE
[re2] has-variant-part?([Prefix,type _id= record _fixed ; _var-part opt; end])
        = TRUE
[re3] selector-field?(T1, _id1) = TRUE
        when T1.variant = case _id1 : _id2 of _variant+
[re4] [Prefix, Entry] . record-of-variant-part =
        [Prefix, type Generated = record _var-part end]
          when [Prefix, Entry].variant = _var-part
[re5] T1.selector-type = find _id2 in T1
        when T1.variant = case _id1: _id2 of _variant+
[re6] T1.selector-type = find _id in T1
        when T1.variant = case _id of _variant+

{ Get the type definition of a type-denoter in a context. }
 [gt1] get-type of _new-type in [Prefix] = [Prefix, type Generated = _new-type]

[gt2] find _id in Context = [Prefix, type _id = _new-type]
      ===================================================
      get-type of _id in Context = [Prefix, type _id = _new-type]

[gt3] find _id1 in Context = [Prefix, type _id1 = _id2],
      _id1 != _id2
      ===================================================
      get-type of _id1 in Context = get-type of _id2 in [Prefix]

[gt4] find _id1 in Context = [Prefix, type _id1 = _id1]
      ===================================================
      get-type of _id1 in Context = [Prefix, type _id1 = _id1]


{ Operations on constants }
[ct1] const-type of _new-const in Context = new-const-type of _new-const

[ct2] const-type of _sign _id in Context = const-type of _id in Context

[ct3] find _id in Context = [Prefix, const _id = _const]
      ==========================================================
      const-type of _id in Context = const-type of _const in [Prefix]

[ct4] find _id in Context = [Prefix, Entry],
       [Prefix, Entry] defines _id as an enum const = <TRUE, type _id2 = (_id*)>
      ==============================================================
      const-type of _id in Context = [Prefix, type _id2 = (_id*)]
```

```
[cv1] const-val of _new-const in Context = new-const-val of _new-const
[cv2] const-val of - _id in Context = -1 * const-val of _id in Context
[cv3] const-val of + _id in Context = const-val of -id in Context

[cv4] find _id in Context = [Prefix, const _id = _const]
      ================================================
      const-val of _id in Context = const-val of _const in [Prefix]

[cv5] find _id in Context = [Prefix, Entry],
        [Prefix, Entry] defines _id as an enum const = <TRUE, type _id2= (_id*)>
      ================================================================
      const-val of _id in Context = id-number of _id in (_id*)


[ct5] [Prefix, const _id = _const].type = const-type of _const in [Prefix]
[ct6] [Prefix, type _id = (_id*)].type = [Prefix, type _id = (_id*)]
[cv7] [Prefix, const _id = _const].value = const-val of _const in [Prefix]


{ Operations on variables }
[ov1] [Prefix, var _id: _type-den VS].type =
        get-type of _type-den in [Prefix]
[ov2] type-to-var([Prefix, type _id = _type-den]) =
        [Prefix, var Generated: _type-den]
[ov3] is-control-variable?(
        [Prefix, var _id: _type-den control-variable]) = TRUE
[ov4] is-possibly-threatening?(
        [Prefix, var _id: _type-den possibly-threatening]) = TRUE
[ov5] set var-status of _id in Context to VS1 =
        [Prefix, var _id: _type-den VS1, Postfix]
          when  complete find _id in Context =
            <[Prefix], var _id: _type-den VS1, [Postfix]>


{ Operations on functions and procedures }
[ft1] [Prefix, function _id _form-par-list: _id2] . formal-parameters =
        _form-par-list
[pt1] [Prefix, procedure _id _form-par-list] . formal-parameters =
        _form-par-list

[ft2] [Prefix, function _id _form-par-list: _id2] . type =
        get-type of _id2 in [Prefix]

[fi1] find block-mark in Context = [Prefix, Entry, block-mark],
      [Prefix, Entry] defines _id as a function? = TRUE
      ============================================
      _id is the name of the inner function-block in Context? = TRUE


[pf1] not-permissable-as-file-comp?(T) = TRUE
        when is-file-type?(T) = TRUE

[pf2] not-permissable-as-file-comp?(T) = TRUE
```

```
        when is-array-type?(T) = TRUE,
          is-file-type?(T.comp-type) = TRUE


[cs1] is-empty-set-type?(T2) = TRUE
      =============================
      T1 is the set-component of T2? = TRUE

[cs1] is-empty-set-type?(T2) = TRUE,
      T2.comp-type is the same as T1? = TRUE
      =====================================
      T1 is the set-component of T2? = TRUE


{ Both types should be set-of-T. Empty set matches all other sets. }
[sT1] is-empty-set-type?(T1) = TRUE,
      is-set-type?(T2) = TRUE
      ========================
      same-set-of-T?(T1, T2) = TRUE

[sT2] is-empty-set-type?(T2) = TRUE,
      is-set-type?(T1) = TRUE
      ========================
      same-set-of-T?(T1, T2) = TRUE

[sT3] is-non-empty-set-type?(T1) = TRUE,
      is-non-empty-set-type?(T2) = TRUE,
      T1.comp-type.canonical is the same as T2.comp-type.canonical? = TRUE
      ==================================================================
      same-set-of-T?(T1, T2) = TRUE


{ Types are the same if they are constructed by the same NEW-TYPE operator.
  Two types are constructed by the same NEW-TYPE operator, if this NEW-TYPE
  is defined in the same place in both contexts of the type.
  Context equivalence can be checked by checking whether the CONTEXT-term
  is the same.
  In addition, the empty-set is the same as any other set, and the
  nil-pointer is the same as any other pointer.  }

[sa1] T1 is the same as T1? = TRUE

[sa2] is-empty-set-type?(T1) = TRUE,
      is-set-type?(T2) = TRUE
      =============================
      T1 is the same as T2? = TRUE

[sa3] is-empty-set-type?(T2) = TRUE,
      is-set-type?(T1) = TRUE
      =============================
      T1 is the same as T2? = TRUE

[sa4] is-nil-pointer-type?(T1) = TRUE,
      is-pointer-type?(T2) = TRUE
      ==============================
```

```
       T1 is the same as T2? = TRUE

  [sa5] is-nil-pointer-type?(T2) = TRUE,
        is-pointer-type?(T1) = TRUE
        ==============================
        T1 is the same as T2? = TRUE
end module Context-sen-funs


module Compatible
%% Define compatible types.
imports Context-sen-funs

exports
  context-free syntax
    CONTEXT and CONTEXT are compatible "?"              -> BOOLEAN
    CONTEXT is assignment-compatible with CONTEXT "?"   -> BOOLEAN

equations

  [c1]  T1 is the same as T2? = TRUE
        =======================================
        T1 and T2 are compatible? = TRUE

  [c2]  is-subrange-type?(T1) = TRUE,
        T1.host-type is the same as T2? = TRUE
        ======================================
        T1 and T2 are compatible? = TRUE

  [c3]  is-subrange-type?(T2) = TRUE,
        T2.host-type is the same as T1? = TRUE
        ======================================
        T1 and T2 are compatible? = TRUE

  [c4]  is-subrange-type?(T1) = TRUE,
        is-subrange-type?(T2) = TRUE,
        T1.host-type is the same as T2.host-type? = TRUE
        ================================================
        T1 and T2 are compatible? = TRUE

  [c5]  is-non-empty-set-type?(T1) = TRUE,
        is-non-empty-set-type?(T2) = TRUE,
        is-packed?(T1) = is-packed?(T2),
        T1.comp-type and T2.comp-type are compatible? = TRUE
        ====================================================
        T1 and T2 are compatible? = TRUE

  [c6]  is-empty-set-type?(T1) = TRUE,
        is-set-type?(T2) = TRUE
        =============================
        T1 and T2 are compatible? = TRUE

  [c7]  is-empty-set-type?(T2) = TRUE,
        is-set-type?(T1) = TRUE
        =============================
```

```
       T1 and T2 are compatible? = TRUE

  [c8]  is-string-type?(T1) = TRUE,
        is-string-type?(T2) = TRUE,
        T1.index-type.nr-of-elements = T2.index-type.nr-of-elements
        ======================================
        T1 and T2 are compatible? = TRUE


  [ac1] T1 is the same as T2? = TRUE,
        not-permissable-as-file-comp?(T1) != TRUE
        =====================================
        T2 is assignment-compatible with T1? = TRUE

  [ac2] T1 = real-type,
        T2 = integer-type
        =================
        T1 is assignment-compatible with T2? = TRUE

  [ac3] is-ordinal-type?(T1) = TRUE,
        is-ordinal-type?(T2) = TRUE,
        T1 and T2 are compatible? = TRUE
        ================================
        T2 is assignment-compatible with T2? = TRUE

  [ac4] is-set-type?(T1) = TRUE,
        is-set-type?(T2) = TRUE,
        T1 and T2 are compatible? = TRUE
        ================================
        T2 is assignment-compatible with T2? = TRUE

  [ac5] is-string-type?(T1) = TRUE,
        is-string-type?(T2) = TRUE,
        T1 and T2 are compatible? = TRUE
        ================================
        T2 is assignment-compatible with T2? = TRUE
end module Compatible
```

C.4 Environments

```
module OperatorTables
%% Specify Operator Tables
imports Booleans

exports
  sorts OP-TYPE BAR OPERATOR  TABLE-HEADER TABLE-ENTRY LOOK-FOR-OUT
    OPERATOR-TABLE

  lexical syntax
    [\-]+ "+" [\-]+ "+" [\-]+ "+" [\-]+                 -> BAR

  context-free syntax
```

```
    "not"   -> OPERATOR   "+"     -> OPERATOR   "-"     -> OPERATOR
    "*"     -> OPERATOR    "/"    -> OPERATOR   "div"   -> OPERATOR
    "mod"   -> OPERATOR   "and"   -> OPERATOR   "or"    -> OPERATOR
    "="     -> OPERATOR   "<>"    -> OPERATOR   "<"     -> OPERATOR
    ">"     -> OPERATOR   "<="    -> OPERATOR   ">="    -> OPERATOR
    "in"    -> OPERATOR
    int     -> OP-TYPE    real    -> OP-TYPE    boolean -> OP-TYPE
    undef   -> OP-TYPE    set     -> OP-TYPE    ordinal -> OP-TYPE
    string  -> OP-TYPE    simple  -> OP-TYPE    pointer -> OP-TYPE

    "Operator" "|" "Operand1" "|" "Operand2" "|" "Result"
                      BAR                               -> TABLE-HEADER
    OPERATOR "|" OP-TYPE  "|" OP-TYPE  "|"  OP-TYPE     -> TABLE-ENTRY
    TABLE-HEADER TABLE-ENTRY*  ";"                      -> OPERATOR-TABLE

    look-for "(" OPERATOR ","
       OP-TYPE "," OP-TYPE "," OPERATOR-TABLE ")"       -> LOOK-FOR-OUT
    "<" BOOLEAN "," OP-TYPE ">"                         -> LOOK-FOR-OUT

hiddens
  context-free syntax
    entry-matches "?" "(" OPERATOR "," OP-TYPE "," OP-TYPE  ","
      OPERATOR "," OP-TYPE "," OP-TYPE ")"              -> BOOLEAN
  variables
    [_]Optor[12]*       -> OPERATOR     [_]Op[123]*     -> OP-TYPE
    [_]Res              -> OP-TYPE      [_]Header       -> TABLE-HEADER
    [_]OpEntries        -> TABLE-ENTRY*

equations

  [lf1] look-for(_Optor, _Op1, _Op2, _Header;) = <FALSE, undef>

  [lf2] look-for(_Optor, _Op1, _Op2,
          _Header _OpEntries _Optor|_Op1|_Op2|_Res ;)   =  <TRUE, _Res>

  [lf3] entry-matches?(_Optor1, _Op11, _Op12, _Optor2, _Op21, _Op22) != TRUE
        ==================================================================
        look-for(_Optor1, _Op11, _Op12,
          _Header _OpEntries _Optor2|_Op21|_Op22|_Res;) =
            look-for(_Optor1, _Op11, _Op12, _Header _OpEntries;)

  [ma1] entry-matches?(_Optor1, _Op1, _Op2, _Optor1, _Op1, _Op2) = TRUE
end module OperatorTables


module Errors
%% Module Errors
imports SyntaxProgram OperatorTables

exports
 sorts ERROR ERRORS CONSTRUCT  CONSTR-ERROR IDENT-LIST

 context-free syntax
    TYPE-DEF                                            -> CONSTRUCT
    CONST-DEF                                           -> CONSTRUCT
```

```
VAR-DECL                                           -> CONSTRUCT
HEADING                                            -> CONSTRUCT
STATEMENT                                          -> CONSTRUCT
program IDENT PROGRAM-PARAMS                       -> CONSTRUCT

CONSTRUCT ":" ERROR                                -> CONSTR-ERROR
CONSTR-ERROR*                                      -> ERRORS
"[" { IDENT ","}* "]"                              -> IDENT-LIST

"Identifier" SYMBOL not "declared."                -> ERROR
"Identifier" IDENT multiple "declared."            -> ERROR
"Identifier" IDENT is not a "constant."            -> ERROR
"Identifier" IDENT is not a type "."               -> ERROR
"Identifier" IDENT is not a "variable."            -> ERROR
"Identifier" IDENT is not a "function."            -> ERROR
"Identifier" IDENT is not a "procedure."           -> ERROR
"Identifier" IDENT is not a "constant," "variable," or "function." -> ERROR
"Empty string not allowed."                        -> ERROR
"Operator" "^" only allowed on pointer or file "variables."        -> ERROR
"Non-number" types not "allowed."                  -> ERROR
"Types" are not the "same."                        -> ERROR
"Types" are not "assignment-compatible."           -> ERROR
"Type must be ordinal."                            -> ERROR
"Type must be an array-type."                      -> ERROR
"Type must be a packed array."                     -> ERROR
"Type must be an unpacked array."                  -> ERROR
"Type must be a pointer."                          -> ERROR
"Type must be a file-type."                        -> ERROR
"Type must" "be Boolean."                          -> ERROR
"Type must be a record."                           -> ERROR
"Type must be a variant record."                   -> ERROR
"Variants must be given for all possible values."  -> ERROR
"Variant selector cannot be passed as var-parameter." -> ERROR
"Constants in case statement or record variant should be unique."  -> ERROR
"Real expected."                                   -> ERROR
"Integer expected."                                -> ERROR
"Empty" range not "allowed."                       -> ERROR
"No" body given for forwarded identifiers ":" IDENT-LIST "."       -> ERROR
"Case" constants should be "unique."               -> ERROR
"Illegal" use of control variable IDENT "."        -> ERROR
"Too" many actual "parameters."                    -> ERROR
"Too" few actual "parameters."                     -> ERROR
"Reference" "to" packed component in VARIABLE-ACCESS not "allowed."-> ERROR
"Illegal use of formatted write parameter."        -> ERROR
"Parameter" lists are not "congruous."             -> ERROR
 ACTUAL-PAR is not a file "variable."              -> ERROR
"Number-type or character-type expected."          -> ERROR
"Number, character, boolean or string type expected."              -> ERROR
"Illegal" pointer type "definition."               -> ERROR
"Illegal" file "component."                        -> ERROR
"Illegal" operand types for operator OPERATOR "."  -> ERROR
"Type" incompatibility in "set-component."         -> ERROR
"Incompatible set-types."                          -> ERROR
ACTUAL-PAR "is not a text file."                   -> ERROR
ACTUAL-PAR "is not a file variable."               -> ERROR
```

```
    ACTUAL-PAR ":" must be a variable "access."         -> ERROR
    ACTUAL-PAR ": Function" identifier "expected."      -> ERROR
    ACTUAL-PAR ": Procedure" identifier "expected."     -> ERROR
    "Types" of OPERATOR are not "compatible," "numbers," or of the
        same set-of-T type "."                          -> ERROR

  variables
    Err[0-9]*      -> CONSTR-ERROR      Err[0-9]*"*"    -> CONSTR-ERR*

equations

  [Er1] Err1* Err Err2* Err Err3* = Err1* Err Err2* Err3*
end module Errors


module Environments
imports Context-search Errors

exports
  sorts ENV FIELD

  context-free syntax
    "(" context "," CONTEXT ")"                         -> FIELD
    "(" errors "," ERRORS ")"                           -> FIELD
    "(" result "," CONTEXT ")"                          -> FIELD
    "(" construct "," CONSTRUCT ")"                     -> FIELD
    "[" { FIELD "," }* "]"                              -> ENV
    "(" ENV ")"                                         -> ENV bracket
    new-env                                             -> ENV
    ENV "." result                                      -> CONTEXT
    ENV "." context                                     -> CONTEXT
    ENV "." construct                                   -> CONSTRUCT
    ENV "." errors                                      -> ERRORS
    ENV "+" ENTRY                                       -> ENV
    leave inner block of ENV                            -> ENV
    add-error "[" ERROR "]" "to" ENV                    -> ENV
    set-result of ENV "to" CONTEXT                      -> ENV
    set-construct of ENV "to" "\"" CONSTRUCT "\""       -> ENV
    set-context of ENV "to" CONTEXT                     -> ENV

  variables
    Env[0-9]*      -> ENV                 E[0-9]*      -> ENV

hiddens
  variables
    Field               -> FIELD           Fields[12]* -> {FIELD ","}*
    [_]c[onstr]*[12]*   -> CONSTRUCT       Error       -> ERROR

equations

  [d2]  [ Fields1, (errors, Err*), Fields2 ].errors = Err*
  [d3]  [ Fields1, (result, C), Fields2 ].result = C
  [d4]  [ Fields1, (context, C), Fields2 ].context = C
  [d5]  [ Fields1, (construct, _c), Fields2 ].construct = _c
```

```
  [a1]  set-result of Env1 to [error-entry] = Env2,
        Env2 = [Fields1, (errors, Err*), Fields2]
        =======================================
        add-error [Error] to Env1 =
          [Fields1, (errors, Err* Env1.construct : Error), Fields2]

  [a2]  leave inner block of [Fields1, (context, C), Fields2] =
          [Fields1, (context, delete inner block of C), Fields2]
  [a3]  [Fields1, (context, C), Fields2] + Entry =
          [Fields1, (context, add Entry to C), Fields2]
  [a4]  set-result of [Fields1, (result, C1), Fields2] to C2 =
          [Fields1, (result, C2), Fields2]
  [a5]  set-construct of [Fields1, (construct, _c1), Fields2] to "_c2" =
          [Fields1, (construct, _c2), Fields2]
  [a6]  set-context of [Fields1, (context, C1), Fields2] to C2 =
          [Fields1, (context, C2), Fields2]

  [c1]  new-env = [(construct,),(context,init-block),(errors,),(result,[])]
end module Environments


module ShouldBe
%% Define several basic should-be functions
%% These functions operate on an Environment. With each should-be function
%% a boolean condition is associated.
%% If the boolean condition is met, an unchanged environment is returned.
%% Otherwise, the environment wiht an error message added to it is returned.
imports Environments Compatible

exports
  sorts VAL-LIST

  context-free syntax
    IDENT "should not be declared in inner block of" ENV              -> ENV
    IDENT "should be declared in inner block of" ENV                  -> ENV
    SYMBOL "should be declared in" ENV                                -> ENV
    IDENT "should be a constant in" ENV                               -> ENV
    IDENT "should be a type in" ENV                                   -> ENV
    IDENT "should be a type in inner block of" ENV                    -> ENV
    IDENT "should be a variable in" ENV                               -> ENV
    IDENT "should be a variable in inner block of" ENV                -> ENV
    IDENT "should be a function in" ENV                               -> ENV
    IDENT "should be a function in inner block of" ENV                -> ENV
    IDENT "should be a procedure in" ENV                              -> ENV
    IDENT "should be a procedure in inner block of" ENV               -> ENV
    STRING "should not be an empty string in" ENV                     -> ENV
    CONTEXT "should be allowed as file-component in" ENV              -> ENV
    CONTEXT "should be a number in" ENV                               -> ENV
    CONTEXT "should be integer in" ENV                                -> ENV
    CONTEXT "should be real in" ENV                                   -> ENV
    CONTEXT "should be Boolean in" ENV                                -> ENV
    "(" { IDENT ","}* ")" "should not have double names in" ENV       -> ENV
    CONTEXT and CONTEXT "should be the same type in" ENV              -> ENV
    CONTEXT "should be assignment-compatible with" CONTEXT in ENV     -> ENV
    CONTEXT "should be ordinal in" ENV                                -> ENV
```

```
    CONTEXT "should be an array in" ENV                              -> ENV
    CONTEXT "should be a packed array in" ENV                        -> ENV
    CONTEXT "should be an unpacked array in" ENV                     -> ENV
    CONTEXT "should be a pointer in" ENV                             -> ENV
    CONTEXT "should be a file in" ENV                                -> ENV
    INT "should be less than" INT in ENV                             -> ENV
    CONTEXT "should be the set-component of" CONTEXT in ENV          -> ENV
    CONTEXT and CONTEXT "should be the same set-of-T type in" ENV    -> ENV
    CONTEXT OPERATOR CONTEXT "should be compatible," "numbers,"
      "or of the same set in" ENV                                    -> ENV
    CONTEXT should be a pointer or a file in ENV                     -> ENV
    CONTEXT "should be a record in" ENV                              -> ENV
    CONTEXT "should be a variant record in" ENV                      -> ENV
    all elements of CONTEXT should occur in VAL-LIST in ENV          -> ENV
    INT should not be a member of VAL-LIST in ENV                    -> ENV
    "[" { INT "," }* "]"          -> VAL-LIST

  variables
    Value[0-9]*     -> INT          Value[0-9]*"*"      -> { INT "," }*
    Val-list[0-9]*  -> VAL-LIST


hiddens
  context-free syntax
    check-double-names of "(" {IDENT "," }* ")" in ENV             -> ENV
    length "(" VAL-LIST ")"                                        -> INT
    INT is member of VAL-LIST "?"                                  -> BOOLEAN

  variables
    Optor  -> OPERATOR           V[0-9]*  -> INT

equations

  [md1] _id exists in inner block of Env.context? != TRUE
        ===================================================
        _id should not be declared in inner block of Env = Env

  [md2] _id exists in inner block of Env.context? = TRUE
        ==================================================
        _id should not be declared in inner block of Env =
          add-error [Identifier _id multiple declared.] to Env


  [md3] _id exists in inner block of Env.context? = TRUE
        ===================================================
        _id should be declared in inner block of Env = Env

  [md4] _id exists in inner block of Env.context? != TRUE
        ==================================================
        _id should be declared in inner block of Env =
          add-error [Identifier _id not declared.] to Env


   { If _id is not yet defined, raise error "Identifier not declared",
     Otherwise set result to corresponding entry. }
```

```
[f1]  find _symbol in Env.context = [Prefix, Entry],
      Entry != error-entry
      ==================================================
      _symbol should be declared in Env =
        set-result of Env to [Prefix, Entry]

[f2]  find _symbol in Env.context = [error-entry]
      =============================================================
      _symbol should be declared in Env =
        add-error [Identifier _symbol not declared.] to Env


[co1] _id should be declared in E1 = E2,
      E2.result defines _id as a constant? = TRUE
      ===========================================
      _id should be a constant in E1 = E2

[co2] _id should be declared in E1 = E2,
      E2.result defines _id as a constant? != TRUE
      ===========================================
      _id should be a constant in E1 =
        add-error [Identifier IDENT is not a constant.] to E2


[ty1] _id should be declared in E1 = E2,
      E2.result defines _id as a type? = TRUE
      ===========================================
      _id should be a type in E1 = E2

[ty2] _id should be declared in E1 = E2,
      E2.result defines _id as a type? != TRUE
      ===========================================
      _id should be a type in E1 =
        add-error [Identifier IDENT is not a type.] to E2

[ty3] _id should be a type in inner block of E =
        _id should be a type in (_id should be declared in inner block of E)

[va1] _id should be declared in E1 = E2,
      E2.result defines _id as a variable? = TRUE
      ===========================================
      _id should be a variable in E1 = E2

[va2] _id should be declared in E1 = E2,
      E2.result defines _id as a variable? != TRUE
      ===========================================
      _id should be a variable in E1 =
        add-error [Identifier IDENT is not a variable.] to E2

[va3] _id should be a variable in inner block of E =
        _id should be a variable in (_id should be declared in inner block of E)

[fu1] _id should be declared in E1 = E2,
      E2.result defines _id as a function? = TRUE
      ===========================================
```

```
      _id should be a function in E1 = E2

[fu2] _id should be declared in E1 = E2,
      E2.result defines _id as a function? != TRUE
      ===========================================
      _id should be a function in E1 =
        add-error [Identifier IDENT is not a function.] to E2

[fu3] _id should be a function in inner block of E =
        _id should be a function in (_id should be declared in inner block of E)

[pr1] _id should be declared in E1 = E2,
      E2.result defines _id as a procedure? = TRUE
      ===========================================
      _id should be a procedure in E1 = E2

[pr2] _id should be declared in E1 = E2,
      E2.result defines _id as a procedure? != TRUE
      ===========================================
      _id should be a procedure in E1 =
        add-error [Identifier IDENT is not a procedure.] to E2

[pr3] _id should be a procedure in inner block of E =
        _id should be a procedure in (_id should be declared in inner block of E)

[ft1] not-permissable-as-file-comp?(T) != TRUE
      ========================================
      T should be allowed as file-component in E = E

[ft2] not-permissable-as-file-comp?(T) = TRUE
      ======================================
      T should be allowed as file-component in E =
         add-error [Illegal file component.] to E

[cn1] T should be a number in E = E when is-number-type?(T) = TRUE
[cn2] T should be a number in E =
        add-error [Non-number types not allowed.] to E
          when is-number-type?(T) != TRUE

[bt1] T should be Boolean in E = E when T = boolean-type
[bt2] T should be Boolean in E = add-error [Type must be Boolean.] to E
        when T != boolean-type

[in1] T should be integer in E = E when T = integer-type
[in2] T should be integer in E = add-error [Integer expected.] to E
        when T != integer-type

[ra1] T should be real in E = E when T = real-type
[ra2] T should be real in E = add-error [Real expected.] to E
        when T != real-type

[dn1] E1 + block-mark = E2,
      check-double-names of (_id*) in E2 = E3,
      leave inner block of E3 = E4
      ===================================
```

```
      (_id*) should not have double names in E1 = E4

[dn2] check-double-names of () in E =  E
[dn3] check-double-names of (_id, _id*) in E1 =
        check-double-names of (_id*) in
          ((_id should not be declared in inner block of E1) + var _id: dummy)

[ts1] T1 and T2 should be the same type in E = E
        when T1 is the same as T2? = TRUE
[ts2] T1 and T2 should be the same type in E =
        add-error [Types are not the same.] to E
          when T1 is the same as T2? != TRUE

[ac1] T1 should be assignment-compatible with T2 in E = E when
        T1 is assignment-compatible with T2? = TRUE
[ac1] T1 should be assignment-compatible with T2 in E =
        add-error [Types are not assignment-compatible.] to E
          when T1 is assignment-compatible with T2? != TRUE

[or1] T should be ordinal in E = E when is-ordinal-type?(T) = TRUE
[or2] T should be ordinal in E = add-error [Type must be ordinal.] to E
        when is-ordinal-type?(T) != TRUE

[ar1] T should be an array in E = E when is-array-type?(T) = TRUE
[ar2] T should be an array in E = add-error [Type must be an array-type.] to E
        when is-array-type?(T) != TRUE

[ar3] T should be a packed array in E = E
        when is-array-type?(T) & is-packed?(T) = TRUE
[ar4] T should be a packed array in E =
        add-error [Type must be a packed array.] to E
          when is-array-type?(T) & is-packed?(T) = TRUE

[ar5] T should be an unpacked array in E = E
        when is-array-type?(T)  = TRUE, is-packed?(T) != TRUE
[ar6] T should be an unpacked array in E =
        add-error [Type must be an unpacked array.] to E
          when is-array-type?(T) = TRUE, is-packed?(T) != TRUE

[re1] T should be a record in E = E when is-record?(T) = TRUE
[re2] T should be a record in E = add-error [Type must be a record.] to E
         when is-record?(T) != TRUE
[re3] T should be a variant record in E = E when has-variant-part?(T) = TRUE
[re4] T should be a variant record in E =
        add-error [Type must be a variant record.] to E
          when has-variant-part?(T) != TRUE

[fi1] T should be a file in E = E when is-file-type?(T) = TRUE
[fi2] T should be a file in E = add-error [Type must be a file-type.] to E
        when is-file-type?(T) != TRUE

[fi1] T should be a pointer in E = E when is-pointer-type?(T) = TRUE
[fi2] T should be a pointer in E = add-error [Type must be a pointer.] to E
        when is-pointer-type?(T) != TRUE
```

```
[lt1] V1 should be less than V2 in E = E when V1 <= V2 = TRUE
[lt2] V1 should be less than V2 in E =
        add-error [Empty range not allowed.] to E
          when V1 <= V2 != TRUE

[ct1] T1 should be the set-component of T2 in E = E
        when T1 is the set-component of T2? = TRUE
[ct2] T1 should be the set-component of T2 in E =
        add-error [Type incompatibility in set-component.] to E
          when T1 is the set-component of T2? != TRUE

[sT1] T1 and T2 should be the same set-of-T type in E = E
        when same-set-of-T?(T1, T2) = TRUE
[sT1] T1 and T2 should be the same set-of-T type in E =
        add-error [Incompatible set-types.] to E
          when same-set-of-T?(T1, T2) != TRUE

[cn1] T1 and T2 are compatible? |
        ( is-number-type?(T1) & is-number-type?(T2) )  |
           same-set-of-T?(T1, T2)  = TRUE
      ==================================================================
      T1 Optor T2 should be compatible, numbers, or of the same set in E = E

[cn2] T1 and T2 are compatible? |
        ( is-number-type?(T1) & is-number-type?(T2) )  |
           same-set-of-T?(T1, T2)  != TRUE
      ==================================================================
      T1 Optor T2 should be compatible, numbers, or of the same set in E =
        add-error [Types of Optor are not compatible, numbers, or of the
          same set-of-T type.] to E

[sb1] is-pointer-type?(T1) | is-file-type?(T1) = TRUE
      ==================================================
      T1 should be a pointer or a file in E = E

[sb2] is-pointer-type?(T1) | is-file-type?(T1) != TRUE
      ==================================================
      T1 should be a pointer or a file in E =
        add-error [Operator ^ only allowed on pointer or file variables.] to E

[es1] length(_string) != 0
      ========================================
      _string should not be an empty string in E = E

[es2] length(_string) = 0
      ========================================
      _string should not be an empty string in E =
        add-error [Empty string not allowed.] to E

[va1] Value is member of [Value1*, Value, Value2*]? = TRUE
[va2] length([]) = 0
[va3] length([Value, Value*]) = length([Value*]) + 1

[va4] Value should not be a member of Val-list in E = E
        when Value is member of Val-list? != TRUE
```

```
  [va5] Value should not be a member of Val-list in E =  add-error
          [Constants in case statement or record variant should be unique.] to E

  [va6] all elements of T1 should occur in Val-list in E = E
          when length(Val-list) = T1.nr-of-elements

  [va7] all elements of T1 should occur in Val-list in E = add-error
          [Variants must be given for all possible values.] to E
            when length(Val-list) != T1.nr-of-elements
end module ShouldBe
```

C.5 Type Checking

```
module TC-defs
%% Type check the definitions and declarations of a Pascal Program.
imports ShouldBe

exports
  sorts VAL-OUT
  context-free syntax
    label-decls-tc "(" {LABEL ","}* "," ENV ")"         -> ENV
    const-defs-tc "(" {CONST-DEF ";"}* "," ENV ")"      -> ENV
    const-tc "(" CONST "," ENV ")"                      -> ENV
    type-defs-tc "(" {TYPE-DEF ";"}* "," ENV ")"        -> ENV
    type-tc "(" TYPE-DENOTER "," ENV ")"                -> ENV
    var-decls-tc "(" { VAR-DECL ";"}* "," ENV ")"       -> ENV
    new-block-with-fields "(" CONTEXT "," ENV ")"       -> ENV
    const-val-list-tc "(" {CONST ","}+ ","
      CONTEXT "," VAL-LIST "," ENV ")"                  -> VAL-OUT
    "<" VAL-LIST "," ENV ">"                            -> VAL-OUT

hiddens
  sorts PTR-OUT ID-LIST PTR-VAL-OUT
  context-free syntax
    "[" { IDENT ","}* "]"                               -> ID-LIST
    "<" ID-LIST "," ENV ">"                             -> PTR-OUT
    type-ptrs-tc "(" { TYPE-DEF ";"}* "," ENV ")"       -> PTR-OUT
    type-denot-tc "(" TYPE-DENOTER "," ENV ")"          -> PTR-OUT
    fields-tc "(" FIELD-LIST "," ENV ")"                -> PTR-OUT
    fixed-tc "(" FIXED-PART "," ENV ")"                 -> PTR-OUT
    variant-tc "(" VAR-PART "," ENV ")"                 -> PTR-OUT
    "<" ID-LIST "," VAL-LIST "," ENV ">"                -> PTR-VAL-OUT
    variant-tc "(" {VARIANT ";"}+ "," CONTEXT "," VAL-LIST "," ENV ")"
                                                        -> PTR-VAL-OUT
    selector-tc "(" SELECTOR "," ENV ")"                -> ENV
    check-pointer-identifiers "(" ID-LIST "," ENV ")"   -> ENV
    ID-LIST should be empty in ENV                      -> ENV

  variables
    V[123]*       -> INT           Component-type       -> TYPE-DENOTER
    Ptr-list[0-9]* -> ID-LIST      Index-type[123]*     -> TYPE-DENOTER
```

```
    Ptr-ids[0-9]*  -> {IDENT ","}* Index-types[+][123]* -> {TYPE-DENOTER "," }+
    Const-def      -> CONST-DEF    Const-defs           -> {CONST-DEF ";"}*
    Type-def       -> TYPE-DEF     Type-defs            -> {TYPE-DEF ";"}*
    Var-decl       -> VAR-DECL     Var-decls            -> {VAR-DECL ";"}*
    Ptr-ids[0-9]*  -> {IDENT ","}*

equations

{ Label declarations}
  [l1]  label-decls-tc(,E) = E
  [l2]  label-decls-tc( _label, _label*, E ) =
          label-decls-tc( _label*, label-decls-tc( _label, E ))
  [l3]  label-decls-tc( _label, E ) = E + label _label


{ Constant declarations }
  [c1]  const-defs-tc(,E) = E
  [c2]  const-defs-tc( Const-def; Const-defs, E ) =
          const-defs-tc(Const-defs, const-defs-tc(Const-def, E))

  [c3]  set-construct of E1 to "_id = _const" = E2,
        _id should not be declared in inner block of E2 = E3 ,
        const-tc( _const, E3 ) = E4
        ==========================================
        const-defs-tc( _id = _const, E1 ) = E4 + const _id = _const

  [c4]  const-tc( _string , E ) =  _string should not be an empty string in E

  [c5]  const-tc( _new-const, E) = E when is-string-const?(_new-const) != TRUE

  [c6]  const-tc( _id , E ) = _id should be a constant in E

  [c7]  _id should be a constant in E1 = E2,
        (const-type of _id in E2.context) should be a number in E2 = E3
        ===================================================================
        const-tc( _sign _id, E1) = E3


{ Type declarations }
  [t1]  type-ptrs-tc(Type-defs, E1) = <Ptr-list, E2>,
        check-pointer-identifiers(Ptr-list, E2) = E3
        ============================================
        type-defs-tc(Type-defs, E1) = E3

  [t2]  type-ptrs-tc(, E) = <[], E>

  [t3]  type-ptrs-tc( Type-def, E1 ) = <[Ptr-ids1], E2>,
        type-ptrs-tc( Type-defs, E2 ) = <[Ptr-ids2], E3>
        ===============================================
        type-ptrs-tc( Type-def; Type-defs, E1 ) = <[Ptr-ids1, Ptr-ids2], E3>

  [t4]   set-construct of E1 to "_id = _type-den" = E2,
        _id should not be declared in inner block of E2 = E3,
        type-denot-tc( _type-den, E3 ) = <Ptr-list, E4>
        ============================================
```

```
      type-ptrs-tc( _id = _type-den, E1 ) =
              <Ptr-list, E4 + type _id = _type-den>

{ Identier as type-denoter }
[t5]  type-denot-tc( _id , E) = <[], _id should be a type in E>

{ Enumeration type constructor }
[te1] type-denot-tc((_id+), E1) = <[], (_id+) should not have double names in E1>

{ Subrange type constructor }
[ts1] const-tc( _const1, E1 ) = E2,
      const-tc( _const2, E2 ) = E3,
      const-type of _const1 in E3.context = T1,
      const-type of _const2 in E3.context = T2,
      T1 and T2 should be the same type in E3 = E4,
      T1 should be ordinal in E4 = E5,
      (const-val of _const1 in E3.context) should be less than
        (const-val of _const2 in E3.context) in E5 = E6
     ==================================================
      type-denot-tc( _const1 .. _const2 , E1 ) = <[], E6>

{ Array type constructor }
[ar1] _PS array [Index-type1, Index-types+] of Component-type =
        _PS array [Index-type1] of
          _PS array [ Index-types+ ] of Component-type

[ar2] type-tc( Index-type, E1 ) = E2,
      type-denot-tc( Component-type, E2 ) = <Ptr-list2, E3>,
      E2.result should be ordinal in E3 = E4
      ======================================
      type-denot-tc( _PS array [ Index-type ] of Component-type , E1 ) =
              <Ptr-list2, E4>

{ Set type constructor }
[se1] type-tc( Component-type, E1 ) = E2,
      E2.result should be ordinal in E2 = E3
      ======================================
      type-denot-tc( _PS set of Component-type, E1) = <[], E3>

{ File type constructor }
[fi1] type-denot-tc( Component-type, E1) = <Ptr-list, E2>,
      E2.result should be allowed as file-component in E2 = E3
      ========================================================
      type-denot-tc( _PS file of Component-type, E1) = <Ptr-list, E3>

{ Pointer type constructor }
[pt1] (find _id in E1.context) defines _id as a type? = TRUE
      ======================================================
      type-denot-tc( ^ _id, E1 ) = <[], E1>

[pt1] (find _id in E1.context) defines _id as a type? != TRUE
      =======================================================
      type-denot-tc( ^ _id, E1 ) = <[_id], E1>
```

```
{ Record type constructor }
[re1] fields-tc( _field-list, E1 + block-mark ) = <Ptr-list, E2>,
      leave inner block of E2 = E3
      ===================================
      type-denot-tc( _PS record _field-list end, E1 ) = <Ptr-list, E3>

[re2] fields-tc( _field-list, E1 + block-mark ) = <Ptr-list, E2>
      ======================================================
      new-block-with-fields([Prefix,  type _id = record _field-list end],E1) = E2

[re3] fields-tc(, E) = <[], E>
[re4] fields-tc(_fixed opt;, E) = fixed-tc(_fixed, E)
[re5] fields-tc(_var-part opt;, E) = variant-tc(_var-part, E)
[re6] fixed-tc( _fixed, E1 ) = <[Ptr-ids1], E2>,
      variant-tc( _var-part, E2 ) = <[Ptr-ids2], E3>
      ============================================
      fields-tc( _fixed; _var-part opt;, E1 ) = <[Ptr-ids1, Ptr-ids2], E3>

[fx1] type-denot-tc( _type-den, E1 ) = <Ptr-list, E2>,
      var-decls-tc( _id+: _type-den, E2 ) = E3
      ====================================================
      fixed-tc( _id+ : _type-den, E1 ) = <Ptr-list, E3>

[fx2] fixed-tc( _rec-section, E1 ) = <[Ptr-ids1], E2>,
      fixed-tc( _rec-section+, E2 ) = <[Ptr-ids2], E3>
      ================================================
      fixed-tc( _rec-section; _rec-section+, E1 ) = <[Ptr-ids1, Ptr-ids2], E3>

[vp1] selector-tc( _selector, E1 ) = E2,
      E2.result.type should be ordinal in E2 = E3,
      variant-tc(_variant+,E2.result.type, [], E3) = <Ptr-list,Val-list,E4>,
      all elements of E2.result.type should occur in Val-list in E4 = E5
      ================================================================
      variant-tc( case _selector of _variant+, E1 ) = <Ptr-list, E5>

[se1] selector-tc( _id, E ) = type-tc( _id, E )
[se2] selector-tc( _id1 : _id2, E ) = var-decls-tc( _id1 : _id2, E )

[va1] const-val-list-tc( _const+, T1, Val-list1, E1 ) = <Val-list2, E2>,
      fields-tc( _field-list, E2 ) = <Ptr-list, E3>
      ===========================================
      variant-tc( _const+:(_field-list), T1, Val-list1, E1 ) =
        <Ptr-list, Val-list2, E3>

[va2] variant-tc( _variant, T1, Val-list1, E1 ) =<[Ptr-ids1], Val-list2, E2>,
      variant-tc(_variant+, T1, Val-list2, E2 ) =<[Ptr-ids2], Val-list3, E3>
      ===================================================================
      variant-tc(_variant ; _variant+, T1, Val-list1, E1 ) =
        <[Ptr-ids1, Ptr-ids2], Val-list3, E3>


{ type-tc may be called when no not-yet-existing pointer identifiers are
  assumed to be used within in the type-denoter, e.g. in the var-decls. }
[ty1] type-denot-tc( _type-den, E1 ) = <Ptr-list, E2>,
```

```
        Ptr-list should be empty in E2 = E3
        ====================================
        type-tc( _type-den, E1 ) = set-result of E3 to
          get-type of _type-den in E3.context

  [pe1] [] should be empty in E = E
  [pe2] [_id+] should be empty in E = add-error
          [Illegal pointer type definition.] to E


  [cp1] check-pointer-identifiers([],E) = E
  [cp2] check-pointer-identifiers([_id], E) =
                _id should be a type in inner block of E
  [cp3] check-pointer-identifiers([_id,_id+],E) =
          check-pointer-identifiers([_id],
            check-pointer-identifiers([_id+],E))


{ Variable declarations }
  [v1]  var-decls-tc(, E) = E

  [v2]  var-decls-tc( Var-decl; Var-decls, E) =
          var-decls-tc(Var-decls, var-decls-tc(Var-decl, E))

  [v3]  var-decls-tc( _id, _id+ : _type-den, E) =
          var-decls-tc( _id: _type-den; _id+: _type-den, E)

  [v4]  set-construct of E1 to "_id : _type-den" = E2,
        _id should not be declared in inner block of E2 = E3,
        type-tc( _type-den, E3 ) = E4
        ========================================
        var-decls-tc( _id: _type-den, E1 ) = E4 + var _id: _type-den


  [cv1] const-tc( _const, E1 ) = E2,
        T1 and E2.result.type should be the same type in E2 = E3,
        E2.result.value should not be a member of [Value*] in E3 = E4
        ============================================================
        const-val-list-tc( _const, T1, [Value*], E1 ) = <[E2.result.value, Value*], E4>

  [cv2] const-val-list-tc( _const, T1, Val-list1, E1 ) = <Val-list2, E2>,
        const-val-list-tc(_const+, T1, Val-list2, E2 ) = <Val-list3, E3>
        ============================================================
        const-val-list-tc(_const, _const+, T1, Val-list1, E1) =
          <Val-list3, E3>
end module TC-defs


module Expr-misc
%% Several general functions needed for typechecking expressions.
imports TC-defs

exports
  context-free syntax
    expr-tc "(" EXPR "," ENV ")"                                -> ENV
```

```
    sim-expr-tc "(" EXPR "," ENV ")"                           -> ENV
    var-access-tc "(" EXPR "," ENV ")"                         -> ENV
    proc-call-tc "(" STATEMENT "," ENV ")"                     -> ENV
    fun-call-tc "(" EXPR "," ENV ")"                           -> ENV
    spec-fun-tc "(" IDENT ACTUAL-PAR-LIST "," ENV ")"          -> ENV
    spec-proc-tc "(" IDENT ACTUAL-PAR-LIST "," ENV ")"         -> ENV

exports
  sorts DESIGNATOR
  context-free syntax
    value                                                      -> DESIGNATOR
    var                                                        -> DESIGNATOR
    procedure                                                  -> DESIGNATOR
    function                                                   -> DESIGNATOR
    FORMAT should be an empty format in ENV                    -> ENV
    act-par-tc "(" DESIGNATOR ACTUAL-PAR "," ENV ")"           -> ENV

    take care of possibly threatening variable ACTUAL-PAR in ENV    -> ENV
    IDENT should not be a control variable in ENV              -> ENV
    IDENT should not be a possibly threatening variable in ENV -> ENV
    mark-variable "(" IDENT "," VAR-STATUS "," ENV ")"         -> ENV
     is-simple-expr "?" "(" EXPR "," CONTEXT ")"               -> BOOLEAN
    is-var-access "?" "(" ACTUAL-PAR "," CONTEXT ")"           -> BOOLEAN
    is-function-call "?" "(" EXPR "," CONTEXT ")"              -> BOOLEAN
    is-a-procedure-identifier "?" "(" EXPR "," CONTEXT ")"     -> BOOLEAN
    is-a-function-identifier "?" "(" EXPR "," CONTEXT ")"      -> BOOLEAN
     ACTUAL-PAR should be a variable-access in ENV             -> ENV
    is-just-an-identifier "?" "(" ACTUAL-PAR ")"               -> BOOLEAN
    is-packed "?" "(" ACTUAL-PAR "," CONTEXT ")"               -> BOOLEAN
    component-of-packed "?" "(" ACTUAL-PAR "," CONTEXT ")"     -> BOOLEAN
    is-file-variable "?" "(" ACTUAL-PAR "," ENV ")"            -> BOOLEAN
    is-text-file "?" "(" ACTUAL-PAR "," ENV ")"                -> BOOLEAN
    is-char-or-number "?" "(" CONTEXT ")"                      -> BOOLEAN
    is-bool-or-string "?" "(" CONTEXT ")"                      -> BOOLEAN
    act-par-is-selector "?" "(" ACTUAL-PAR "," ENV ")"         -> BOOLEAN
    CONTEXT should be a character or a number in ENV           -> ENV
    CONTEXT should be a "number," "character," "bool," or string in ENV -> ENV
    EXPR should be a procedure identifier in ENV               -> ENV
    EXPR should be a function identifier in ENV                -> ENV
    ACTUAL-PAR "should be a text file variable in" ENV         -> ENV
    ACTUAL-PAR should not be a packed component in ENV         -> ENV
    ACTUAL-PAR should be a file variable in ENV                -> ENV
    ACTUAL-PAR "should not be a variant selector field in" ENV -> ENV
    no { ACTUAL-PAR ","}* should be given in ENV               -> ENV

hiddens
  context-free syntax
    is-potential-control-variable "?" "(" IDENT "," CONTEXT ")"-> BOOLEAN
    non-formatted-par-tc "(" DESIGNATOR EXPR "," ENV ")"       -> ENV
  variables
    Definition -> CONTEXT    Status -> VAR-STATUS   Designator -> DESIGNATOR

equations

  { Dealing with control variables. }
```

```
[tr0] take care of possibly threatening variable _act-par in E = E
        when is-just-an-identifier?( _act-par ) != TRUE

[tr1] is-potential-control-variable?(_id, E1.context) != TRUE
      =======================================================
      take care of possibly threatening variable _id in E1 = E1

[tr2] is-potential-control-variable?(_id, E1.context) = TRUE,
      _id should not be a control variable in E1 = E2,
      mark-variable(_id, possibly-threatening, E2) = E3
      =============================================
      take care of possibly threatening variable _id in E1 = E3


[pc1] _id exists in inner block of Context? != TRUE,
      find _id in Context = Definition,
      Definition defines _id as a variable? = TRUE,
      is-ordinal-type?(Definition.type) = TRUE
      ===================================
      is-potential-control-variable?(_id, Context) = TRUE


[ic1] find _id in E.context = Definition,
      is-control-variable?(Definition) = TRUE
      =======================================
      _id should not be a control variable in E =
        add-error [Illegal use of control variable _id.]  to E

[ic2] find _id in E.context = Definition,
      is-control-variable?(Definition) != TRUE
      =======================================
      _id should not be a control variable in E = E


[ic5] find _id in E.context = Definition,
      is-possibly-threatening?(Definition) = TRUE
      =======================================
      _id should not be a possibly threatening variable in E =
        add-error [Illegal use of control variable _id.] to E

[ic6] find _id in E.context = Definition,
      is-possibly-threatening?(Definition) = TRUE
      =======================================
      _id should not be a possibly threatening variable in E = E


[mt1] mark-variable(_id, Status, E) =
        set-context of E to (set var-status of _id in E.context to Status)


{ Actual parameters }
[ap1] non-formatted-par-tc( Designator _expr, E1 ) = E2,
      _format1 should be an empty format in E2 = E3
      ==============================================
      act-par-tc( Designator _expr _format1 , E1 ) = E3
```

```
[ap2] non-formatted-par-tc( value _expr, E1 ) = expr-tc( _expr, E1 )

[ap3] var-access-tc( _expr, E1 ) = E2
      ===============================
      non-formatted-par-tc( var _expr, E1 ) = set-result of E2 to E2.result.type

[ap5] non-formatted-par-tc( procedure _expr, E ) =
              _expr should be a procedure identifier in E

[ap6] non-formatted-par-tc( function _expr, E ) =
              _expr should be a function identifier in E

[ap7] should be an empty format in E = E
[ap8] _format should be an empty format in E =
        add-error [Illegal use of formatted write parameter.] to E
          when _format !=


{ classification of expressions }
[ce0] is-just-an-identifier?( _id ) = TRUE

[ce1] is-simple-expr?( _uns-number, Context ) = TRUE
[ce2] is-simple-expr?( _string, Context ) = TRUE
[ce3] is-simple-expr?( nil, Context ) = TRUE
[ce4] is-simple-expr?( [ _member* ], Context ) = TRUE
[ce5] is-simple-expr?( _id, Context ) = TRUE
        when (find _id in Context) defines _id as a constant? = TRUE

[ce6] is-var-access?( _id, Context ) =  TRUE
        when (find _id in Context) defines _id as a variable? = TRUE
[ce7] is-var-access?( _var-acc, Context ) = TRUE
        when is-just-an-identifier?( _var-acc ) != TRUE

[ce8] _expr should be a variable-access in E = E
        when is-var-access?( _expr, E.context ) = TRUE
[ce9] _expr should be a variable-access in E =
        add-error [_expr: must be a variable access.] to E
          when is-var-access?( _expr, E.context ) != TRUE

[ce0] is-function-call?(_id, Context) = TRUE
        when (find _id in Context) defines _id as a function? = TRUE
[ce1] is-function-call?( _id _NE-act-par-list, Context ) = TRUE

[ce2] is-a-procedure-identifier?(_id, Context) = TRUE
        when (find _id in Context) defines _id as a procedure? = TRUE
[ce3] is-a-function-identifier?(_id, Context) = TRUE
        when (find _id in Context) defines _id as a function? = TRUE

[ce4] _id should be a procedure identifier in E =
        set-result of E to (find _id in E.context)
          when is-a-procedure-identifier?(_id, E.context) = TRUE
[ce5] _expr should be a procedure identifier in E =
        add-error [_expr: Procedure identifier expected.] to E
          when is-a-procedure-identifier?(_expr, E.context) != TRUE
```

```
[ce6] _id should be a function identifier in E =
        set-result of E to (find _id in E.context)
          when is-a-procedure-identifier?(_id, E.context) = TRUE
[ce7] _expr should be a function identifier in E =
        add-error [_expr: Function identifier expected.] to E
          when is-a-procedure-identifier?(_expr, E.context) != TRUE


[ip1] is-packed?(_id, Context) = is-packed?(find _id in Context)
[ip2] is-packed?( _var-acc [ _expr ] , Context ) =
            is-packed?( _var-acc , Context )
[ip3] is-packed?( _var-acc . _id, Context ) =
            is-packed?( _var-acc , Context )

[ip4] component-of-packed?( _var-acc [ _expr ], Context ) =
            is-packed?( _var-acc, Context )
[ip5] component-of-packed?( _var-acc . _id, Context ) =
            is-packed?( _var-acc, Context )

[ip6] component-of-packed?( _act-par, E.context ) != TRUE
      ====================================================
      _act-par should not be a packed component in E = E

[ip7] component-of-packed?( _var-acc, E.context ) = TRUE
      ====================================================
      _var-acc should not be a packed component in E =
        add-error [Reference to packed component in _var-acc not allowed.]
          to E


[np1] no should be given in E = E
[np2] no _act-par+ should be given in E =
        add-error [Too many actual parameters.] to E


[fv1] act-par-tc(var _act-par, E1) = E2,
      E2.result should be a file in E2 = E3
      =======================================
      _act-par should be a file variable in E1 = E3


[if1] is-file-variable?(_act-par, E) =
        is-var-access?(_act-par, E.context) &
          is-file-type?(act-par-tc(var _act-par, E).result)

[cn1] is-char-or-number?(T) = TRUE when T = char-type
[cn2] is-char-or-number?(T) = TRUE when is-number-type?(T) = TRUE
[cn3] T should be a character or a number in E = E
        when is-char-or-number?(T) = TRUE
[cn4] T should be a character or a number in E =
        add-error [Number-type or character-type expected.] to E
          when is-char-or-number?(T) != TRUE

[cn5] is-bool-or-string?(T) = TRUE when T = boolean-type
```

```
  [cn6] is-bool-or-string?(T) = TRUE when is-string-type?(T) = TRUE
  [cn7] T should be a number, character, bool, or string in E = E
          when is-bool-or-string?(T) | is-char-or-number?(T) = TRUE
  [cn8] T should be a number, character, bool, or string in E =
          add-error [Number, character, boolean or string type expected.] to E
            when is-bool-or-string?(T) | is-char-or-number?(T) = TRUE

  [tf1] is-text-file?(_act-par, E) =  TRUE when
                is-var-access?(_act-par, E.context) = TRUE,
                act-par-tc(var _act-par, E).result = text-type

  [tf2] _act-par should be a text file variable in E = E
          when is-text-file?(_act-par, E) = TRUE
  [tf3] _act-par should be a text file variable in E =
          add-error [_act-par is not a text file.] to E
            when is-text-file?(_act-par, E) != TRUE


  [s2]  var-access-tc( _var-acc, E1 ) = E2,
        selector-field?(E2.result, _id1) = TRUE
        ======================================
        act-par-is-selector?(_var-acc : _id1, E1) = TRUE

  [s3]  _act-par should not be a variant selector field in Env = Env
          when act-par-is-selector?(_act-par, Env) != TRUE

  [s4]  _act-par should not be a variant selector field in Env =
          add-error [Variant selector cannot be passed as var-parameter.]
            to Env
              when act-par-is-selector?( _act-par, Env ) = TRUE
end module Expr-misc


module TC-expr-simple
%% Type check simple expressions: Numbers, Strings, Nil, Set-constructor
imports Expr-misc

hiddens
  context-free syntax
    member-tc "(" MEMBER-DESIGNATOR "," ENV ")"         -> ENV

equations

  [s1]  sim-expr-tc( _uns-number, E ) = set-result of E to new-const-type of _uns-number

  [s2]  sim-expr-tc( _string, E ) =  set-result of E to new-const-type of  _string

  [s3]  _id should be a constant in E1 = E2
        ==================================
        sim-expr-tc( _id, E1 ) =  set-result of E2 to const-type of _id in E1.context

  [s4]  sim-expr-tc( nil, E ) = set-result of E to [nil-pointer]

  [s5]  sim-expr-tc( [], E ) = set-result of E to [empty-set]
```

```
  [s6]  member-tc(_member, E1) = E2,
        sim-expr-tc([_member*], E2) = E3,
        E2.result and E3.result should be the same type in E3 = E4
        ===========================================================
        sim-expr-tc([_member, _member*], E1) =
                set-result of E4 to E2.result

  [s7]  member-tc( _expr, E ) = expr-tc( _expr, E )

  [s8]  expr-tc( _expr1, E1 ) = E2,
        expr-tc( _expr2, E2 ) = E3,
        E2.result and E3.result should be the same type in E3 = E4,
        E2.result should be ordinal in E4 = E5
        ========================================
        member-tc( _expr1 .. _expr2 , E1 ) = E5
end module TC-expr-simple


module TC-expr-vars
%% Typecheck a variable access.
imports Expr-misc

hiddens
  context-free syntax
    var-arrow-tc "(" VARIABLE-ACCESS "," ENV ")"        -> ENV

equations

{ var-access-tc returns an environment. The result of the environment is
  set to a variable definition of the appropriate type.
  Type checking an expression consisting of only that variable-access will
  transform that variable definition into a type definition. }

  [v1]  var-access-tc( _id, E ) =  _id should be a variable in E

  [a1]  var-access-tc( _var-acc[ _expr, _expr+ ], E ) =
          var-access-tc( _var-acc[ _expr ] [ _expr+ ], E)

  [v2]  var-access-tc( _var-acc, E1 ) = E2,
        expr-tc( _expr, E2 )  = E3,
        E2.result.type should be an array in E3 = E4,
        E3.result should be assignment-compatible with
          E2.result.type.index-type in E4 = E5
        ==============================================
        var-access-tc( _var-acc [ _expr ], E1 ) =
          set-result of E5 to type-to-var(E2.result.type.comp-type)

  [v3]  var-access-tc( _var-acc, E1 ) = E2,
        is-file-type?( E2.result.type ) = TRUE
        ======================================
        var-arrow-tc( _var-acc^, E1 ) =
          set-result of E2 to type-to-var(E2.result.type.comp-type)

  [v3]  var-access-tc( _var-acc, E1 ) = E2,
        is-pointer-type?( E2.result.type ) = TRUE
```

```
        ======================================
        var-arrow-tc( _var-acc^, E1 ) =
          set-result of E2 to type-to-var(
            pointer-component of E2.result.type in E1.context)

  [v4]  var-access-tc( _var-acc, E1 ) = E2,
        E2.result.type should be a pointer or a file in E2 = E3,
        var-arrow-tc( _var-acc^, E3 ) = E4
        ================================
        var-access-tc( _var-acc^, E1 ) = E4

  [v5]  var-access-tc( _var-acc, E1 ) = E2,
        E2.result.type should be a record in E2 = E3,
        new-block-with-fields(E2.result.type, E3) = E4,
          _id should be declared in inner block of E4 = E5,
        leave inner block of E5 = E6
        ==================================================
        var-access-tc( _var-acc._id, E1 ) =
                set-result of E6 to E5.result.type
end module TC-expr-vars


module TC-expr-calls
%% Type check procedure and function calls.
imports Expr-misc

hiddens
  context-free syntax
    call-tc "(" FORMAL-PAR-LIST-OUT "," ACTUAL-PAR-LIST "," ENV ")" -> ENV
    FORMAL-PAR-LIST-OUT and FORMAL-PAR-LIST-OUT
      should be congruous in ENV                                   -> ENV
    congruous "?" "(" FORMAL-PAR-LIST-OUT
      "," FORMAL-PAR-LIST-OUT "," ENV ")"                          -> BOOLEAN

  variables
    Definition                                                     -> CONTEXT

equations

  [p1]  find _id in E1.context = Definition,
        Definition defines _id as a predefined procedure? = TRUE,
        spec-proc-tc(_id _act-par-list, E1) = E2
        ========================================
        proc-call-tc(_id _act-par-list, E1) = E2

  [p2]  find _id in E1.context = Definition,
        Definition defines _id as a normal procedure? = TRUE,
        call-tc( Definition.formal-parameters, _act-par-list, E1) = E2
        ========================================
        proc-call-tc(_id _act-par-list, E1) = E2

  [f1]  find _id in E1.context = Definition,
        Definition defines _id as a predefined function? = TRUE,
        spec-fun-tc(_id, E1) = E2
        ========================================
```

```
      fun-call-tc(_id, E1) = E2

[f2]  find _id in E1.context = Definition,
      Definition defines _id as a predefined function? = TRUE,
      spec-fun-tc(_id _NE-act-par-list, E1) = E2
      ========================================
      fun-call-tc(_id _NE-act-par-list, E1) = E2

[f3]  find _id in E1.context = Definition,
      Definition defines _id as a normal function? = TRUE,
      call-tc( Definition.formal-parameters, _NE-act-par-list, E1) = E2
      ========================================
      fun-call-tc(_id _NE-act-par-list, E1) =
        set-result of E2 to Definition.type.canonical

[f4]  find _id in E1.context = Definition,
      Definition defines _id as a normal function? = TRUE,
      call-tc( Definition.formal-parameters, , E1) = E2
      =====================================================
      fun-call-tc(_id, E1) = set-result of E2 to Definition.type.canonical


[c0]  call-tc( , , E ) = E

[c1]  call-tc( (_form-par+), , E) = add-error
        [Too few actual parameters.] to E

[c2]  call-tc(  , (_act-par+), E) = add-error
        [Too many actual parameters.] to E

[c3]  call-tc( (_form-par; _form-par+), (_act-par, _act-par+), E) =
        call-tc( (_form-par+), (_act-par+),
          call-tc( (_form-par), (_act-par), E ))


[c4]  act-par-tc(value _act-par, E1) = E2,
      type-tc(_id2, E2) = E3,
      E2.result should be assignment-compatible with E3.result in E3 = E4
      ==================================================================
      call-tc( (_id1: _id2), (_act-par), E1 ) = E4

[c5]  act-par-tc(var _act-par, E1) = E2,
      _act-par should not be a packed component in E2 = E3,
      take care of possibly threatening variable _act-par in E3 = E4,
      { _expr should not be a variant selector field in E4 = E5}
      type-tc(_id2, E4) = E5,
      E2.result and E5.result should be the same type in E5 = E6
      ==================================================================
      call-tc( (var _id1: _id2), (_act-par), E1 ) = E6

[c6]  act-par-tc(procedure _act-par, E1) = E2,
      E2.result.formal-parameters and _form-par-list
        should be congruous in E2 = E3
      ===================================================================
      call-tc( (procedure _id1 _form-par-list), (_act-par), E1 )  = E3
```

```
  [c7]  act-par-tc(function _act-par, E1) = E2,
        E2.result.formal-parameters and _form-par-list
          should be congruous in E2 = E3
        ===================================================================
        call-tc( (function _id1 _form-par-list: _id2), (_act-par), E1 ) = E3


  { Two parameter lists are congruous if they have the same number of
    arguments, and if each parameter pair refers to the same type. }

  [co1] congruous?(,,E) = TRUE
  [co2] congruous?((_form-par1; _form-par1+), (_form-par2; _form-par2+), E) =
          congruous?((_form-par1), (_form-par2), E)   &
          congruous?((_form-par1+), (_form-par2+), E)

  [co3] type-tc(_id1, E1) = E2,
        type-tc(_id2, E2) = E3,
        E2.result is the same as E3.result? = TRUE
        ===========================================
        congruous?((_id11: _id1), (_id22: _id2),E1) = TRUE

  [co4] type-tc(_id1, E1) = E2,
        type-tc(_id2, E2) = E3,
        E2.result is the same as E3.result? = TRUE
        ===========================================
        congruous?((var _id11: _id1), (var _id22: _id2), E1) = TRUE

  [co5] congruous?((procedure _id1 _form-par-list1),
          (procedure _id2 _form-par-list2), E) =
            congruous?(_form-par-list1, _form-par-list2, E)

  [co6] type-tc(_id1, E1) = E2,
        type-tc(_id2, E2) = E3,
        E2.result is the same as E3.result? = TRUE
        ===========================================
        congruous?((function _id11 _form-par-list1 : _id1),
          (function _id22 _form-par-list2 : _id2), E1) =
            congruous?(_form-par-list1, _form-par-list2, E1)

  [co7] congruous?( _form-par-list1, _form-par-list2, E ) = TRUE
        =====================================================
        _form-par-list1 and _form-par-list2 should be congruous in E = E

  [co8] congruous?( _form-par-list1, _form-par-list2, E ) != TRUE
        =====================================================
        _form-par-list1 and _form-par-list2 should be congruous in E =
          add-error [Parameter lists are not congruous.] to E
end module TC-expr-calls


module TC-spec-funs
%% Type check special, predefined, functions.
imports  Expr-misc
```

```
equations

  [sf1] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result and real-type should be the same type in E3 = E4
        =====================================================
        spec-fun-tc( trunc(_act-par, _act-par*), E1 ) =
          set-result of E4 to real-type

  [sf2] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result and real-type should be the same type in E3 = E4
        =====================================================
        spec-fun-tc( round(_act-par, _act-par*), E1 ) =
          set-result of E4 to real-type

  [sf3] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be a number in E3 = E4
        =====================================================
        spec-fun-tc( abs(_act-par, _act-par*), E1 ) =
          set-result of E4 to E2.result

  [sf4] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be a number in E3 = E4
        =====================================================
        spec-fun-tc( sqr(_act-par, _act-par*), E1 ) =
          set-result of E4 to E2.result

  [sf5] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be ordinal in E3 = E4
        =====================================================
        spec-fun-tc( ord(_act-par, _act-par*), E1 ) =
          set-result of E4 to integer-type

  [sf6] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be ordinal in E3 = E4
        =====================================================
        spec-fun-tc( succ(_act-par, _act-par*), E1 ) =
          set-result of E4 to E2.result

  [sf7] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be ordinal in E3 = E4
        =====================================================
        spec-fun-tc( pred(_act-par, _act-par*), E1 ) =
          set-result of E4 to E2.result

  [sf8] act-par-tc( var  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be a file in E3 = E4
        =====================================================
```

```
       spec-fun-tc( eoln(_act-par, _act-par*), E1 ) =
         set-result of E4 to boolean-type

  [sf9] act-par-tc( value  _act-par, E1 ) = E2,
        no _act-par* should be given in E2 = E3,
        E2.result should be a file in E3 = E4
        ===================================================
        spec-fun-tc( eof(_act-par, _act-par*), E1 ) =
          set-result of E4 to boolean-type

  [eo1] spec-fun-tc( eof, E) = spec-fun-tc( eof(input), E)
  [eo2] spec-fun-tc( eoln,E) = spec-fun-tc(eoln(input), E)
end module TC-spec-funs


module TC-spec-procs
%% Check the predefined procedures: input/output, dynamic allocation, (un)pack
imports Expr-misc

hiddens
  context-free syntax
    stc "(" IDENT ACTUAL-PAR-LIST "," ENV ")"           -> ENV
    tcf "(" IDENT ACTUAL-PAR-LIST "," ENV ")"           -> ENV
    "write-par-tc" "(" ACTUAL-PAR "," ENV ")"           -> ENV
    tag-constants-tc "(" {ACTUAL-PAR ","}+ "," CONTEXT "," ENV ")" -> ENV

  variables
    [_]par[0-9]*       -> ACTUAL-PAR    [_]file         -> ACTUAL-PAR
    [_][paiz][0-9]*    -> ACTUAL-PAR    [_]par[0-9]*"*"  -> { ACTUAL-PAR "," }*
    [_]p[ar]*[+][0-9]* -> {ACTUAL-PAR ","}+

equations
  [sp1] spec-proc-tc(_id _act-par-list, E) = stc(_id _act-par-list, E)

  { Procedures rewrite, put, reset, and get all should be called with
    exactly one parameter. This parameter should be a file variable.
    ISO section 6.6.5.2 }

  [re1] stc( rewrite, E ) = add-error [Too few actual parameters.] to E
  [pu1] stc( put    , E ) = add-error [Too few actual parameters.] to E
  [rs1] stc( reset  , E ) = add-error [Too few actual parameters.] to E
  [ge1] stc( get    , E ) = add-error [Too few actual parameters.] to E

  [re2] stc( rewrite(_par), E)  = _par should be a file variable in E
  [pu2] stc( put(_par), E)      = _par should be a file variable in E
  [rs2] stc( reset(_par), E)    = _par should be a file variable in E
  [ge2] stc( get(_par), E)      = _par should be a file variable in E

  [re3] stc(rewrite(_par,_par+),E)=add-error [Too many actual parameters.] to E
  [pu3] stc(put    (_par,_par+),E)=add-error [Too many actual parameters.] to E
  [rs3] stc(reset  (_par,_par+),E)=add-error [Too many actual parameters.] to E
  [ge3] stc(get    (_par,_par+),E)=add-error [Too many actual parameters.] to E


  { The procedure read. If the first parameter is not a file variable,
```

```
  insert file variable "input". If the file parameter is of type text,
  (ISO section 6.9.1), the actual parameters shall be of char-type,
  integer-type, or real-type. If the file parameter is not of type text,
  (ISO section 6.6.5.2) the type of the actual parameter should be the
  same as the component type of the file. }

[ea1] stc( read, E) = add-error [Too few actual parameters.] to E

[ea2] stc( read(_par), E) = add-error [Too few actual parameters.] to E
        when is-file-variable?(_par,E) = TRUE

[ea3] stc( read(_par, _par+), E) = tcf(read(_par,_par+), E)
        when is-file-variable?(_par,E) = TRUE

[ea4] stc( read(_par, _par*), E) = tcf(read(input,_par,_par*), E)
        when is-file-variable?(_par,E) != TRUE

[ea5] tcf( read(_file,_par,_par+), E) =
        tcf( read(_file,_par), tcf(read(_file,_par+), E))

[ea6] act-par-tc(var _file, E1) = E2,
      E2.result.type = text-type,
      act-par-tc(var _par, E2) = E3,
      E3.result should be a character or a number in E3 = E4
      =====================================================
      tcf( read(_file, _par), E1) = E4

[ea7] act-par-tc(var _file, E1) = E2,
      E2.result.type != text-type,
      act-par-tc(var _par, E2) = E3,
      E3.result should be assignment-compatible with
              E2.result.type.comp-type in E3 = E4
      ===============================================
      tcf( read(_file, _par), E1) = E4


{ The procedure readln, ISO section 6.9.2 }
[ln1] stc( readln, E) = tcf( readln(input), E)

[ln2] stc( readln(_par, _par*), E) = tcf(readln(input, _par, _par*), E)
        when is-file-variable?(_par, E) != TRUE

[ln3] stc( readln(_par, _par*), E) = tcf(readln(_par, _par*), E)
        when is-file-variable?(_par, E) = TRUE

[ln4] tcf( readln(_file, _par+), E) =
        tcf( readln(_file), tcf(read(_file, _par+), E))

[ln5] tcf( readln(_file), E ) = _file should be a text file variable in E


{ The procedure write, ISO section 6.9.3, 6.6.5.2.
  Eventually insert file-variable output. Distinguish between text-files and
  non-text files. Check formats. }
```

```
[wr1] stc( write, E) = add-error [Too few actual parameters.] to E

[wr2] stc( write(_par), E) = add-error [Too few actual parameters.] to E
        when is-file-variable?(_par, E) = TRUE

[wr3] stc( write(_par, _par+), E) = tcf(write(_par, _par+), E)
        when is-file-variable?(_par, E) = TRUE

[wr4] stc( write(_par, _par*), E) = tcf(write(output, _par, _par*), E)
        when is-file-variable?(_par, E) != TRUE

[wr5] tcf( write(_file,_par,_par+), E) =
        tcf( write(_file, _par), tcf(write(_file, _par+), E))

[wr6] act-par-tc(var _file, E1) = E2,
      E2.result.type = text-type,
      write-par-tc(_par, E2) = E3,
      E3.result should be a number, character, bool, or string in E3 = E4
      ===================================================================
      tcf( write(_file, _par), E1) = E4

[wr7] act-par-tc(var _file, E1) = E2,
      E2.result.type != text-type,
      act-par-tc(value _par, E2) = E3,
      E3.result should be assignment-compatible with
              E2.result.type.comp-type in E3 = E4
      ================================================
      tcf( write(_file, _par), E1) = E4


{ The procedure writeln, ISO section 6.9.4. }
[ln6] stc( writeln, E) = tcf( writeln(output), E )

[ln7] stc( writeln(_par, _par*), E) = tcf(writeln(output,_par, _par*),E)
        when is-file-variable?(_par, E) != TRUE

[ln8] stc( writeln(_par, _par*), E) = tcf(writeln(_par, _par*), E)
        when is-file-variable?(_par, E) = TRUE

[ln9] tcf( writeln(_file, _par+), E) =
        tcf( writeln(_file), tcf(write(_file, _par+), E))

[ln0] tcf( writeln(_file), E) = _file should be a text file variable in E


{ The procedure page, ISO section 6.9.5.}
[pa1] stc( page, E ) = stc( page(output), E)

[pa2] stc( page(_file), E) = _file should be a text file variable in E

[pa3] stc( page(_file, _par+), E) = add-error [Too many actual parameters.] to E


[wp1]  write-par-tc(_expr, E) = expr-tc(_expr, E)
```

```
[wp2]  expr-tc( _expr2, E1) = E2,
       expr-tc( _expr1, E2) = E3,
       E3.result should be integer in E3 = E4
       ========================================
       write-par-tc( _expr1 : _expr2, E1 ) = E4

[wp3]  expr-tc( _expr3, E1) = E2,
       expr-tc( _expr2, E2) = E3,
       expr-tc( _expr1, E3) = E4,
       E2.result should be integer in E4 = E5,
       E3.result should be integer in E5 = E6,
       E4.result should be real in E6 = E7
       ========================================
       write-par-tc( _expr1 : _expr2 : _expr3, E1 ) = E5


{ Dynamic allocation procedures new and dispose, ISO section 6.6.5.3.}
[da1] act-par-tc(var _p, E1) = E2,
      E2.result.type should be a pointer in E2 = E3
      =============================================
      stc( new(_p), E1) = E3

[da2] stc( new(_p), E1) = E2,
      tag-constants-tc( _par+, E2.result.type, E2 ) = E3
      ==========================================================
      stc( new(_p, _par+), E1 ) = E3


[da3] T1 should be a variant record in E1 = E2,
      act-par-tc( value _par, E2 ) = E3,
      T1.selector-type and E3.result.type should be the same type in E3 = E4
      =================================================================
      tag-constants-tc( _par, T1, E1 ) = E4

[da4] tag-constants-tc( _par, T1, E1 ) = E2,
      tag-constants-tc( _par+, T1.record-of-variant-part, E2 ) = E3
      ==============================================================
      tag-constants-tc( _par, _par+, T1, E1 ) = E3

[da5] stc( new, E ) = add-error [Too few actual parameters.] to E
[da6] stc( dispose _act-par-list, E ) = stc( new _act-par-list, E )


{ Transfer procedures, pack(a,i,z), unpack(z,a,j), ISO section 6.6.5.4.}
[tp1] act-par-tc(var   _z, E1) = E2,
      act-par-tc(var   _a, E2) = E3,
      act-par-tc(value _i, E3) = E4,
      E2.result.type should be an unpacked array in E4 = E5,
      E3.result.type should be a packed array in E5 = E6,
      E2.result.type.comp-type and  E3.result.type.comp-type
        should be the same type in E6 = E7,
      E4.result should be assignment-compatible with
        E3.result.type.index-type in E7 = E8
      =================================================
      stc(pack(_a, _i, _z), E1) = E8
```

```
  [tp2] stc(unpack(_z, _a, _i), E) = stc(pack(_a, _i, _z), E)

  [tp3] stc(pack, E) = add-error [Too few actual parameters.] to E
  [tp4] stc(pack(_p1), E) = add-error [Too few actual parameters.] to E
  [tp5] stc(pack(_p1,_p2), E) = add-error [Too few actual parameters.] to E
  [tp6] stc(unpack, E) = add-error [Too few actual parameters.] to E
  [tp7] stc(unpack(_p1), E) = add-error [Too few actual parameters.] to E
  [tp8] stc(unpack(_p1,_p2), E) = add-error [Too few actual parameters.] to E
  [tp9] stc(pack(_p1,_p2,_p3,_p+),E) =
          add-error [Too many actual parameters.] to E
  [tp0] stc(unpack(_p1,_p2,_p3,_p+),E) =
             add-error [Too many actual parameters.] to E
end module TC-spec-procs


module TC-expr
imports TC-expr-vars TC-expr-simple TC-expr-calls TC-spec-funs TC-spec-procs

hiddens
  context-free syntax
    arithm-op-table                                   -> OPERATOR-TABLE
    relational-op-table                               -> OPERATOR-TABLE

    unop-tc "(" OPERATOR "," EXPR "," ENV ")"           -> ENV
    binop-tc "(" OPERATOR "," EXPR "," EXPR "," ENV ")" -> ENV
    relop-tc "(" OPERATOR "," EXPR "," EXPR "," ENV ")" -> ENV
    match-ops "(" OPERATOR-TABLE ","
      OPERATOR "," OP-TYPE "," OP-TYPE "," ENV ")"      -> ENV
    check-sets "(" CONTEXT "," CONTEXT ")" in ENV       -> ENV
    get-result "(" CONTEXT "," CONTEXT ")"              -> CONTEXT

    map "(" CONTEXT ")"                                 -> OP-TYPE
    map "(" OP-TYPE ")"                                 -> CONTEXT
    rmap "(" CONTEXT ")"                                -> OP-TYPE
    omap "(" CONTEXT ")"                                -> OP-TYPE
    is-just-an-identifier "?" "(" EXPR ")"              -> BOOLEAN

  variables
    [_]*Op[era]*tor -> OPERATOR       Op[0-9]*          -> OP-TYPE
    Result-type     -> OP-TYPE        OpTable           -> OPERATOR-TABLE

equations

  { Direct the basic expressions:, constants and simple expressions,
    variables accesses, and function calls. }

  [e1]  is-simple-expr?( _expr, Env.context ) = TRUE
        ==========================================
        expr-tc( _expr, Env ) = sim-expr-tc( _expr, Env )

  [e2]  is-var-access?( _expr, Env1.context ) = TRUE,
        var-access-tc( _expr, Env1 ) = Env2
        ========================================
        expr-tc( _expr, Env1 ) = set-result of Env2 to Env2.result.type.canonical
```

```
[e3]  is-function-call?( _expr, Env1.context ) = TRUE
      ===============================================
      expr-tc( _expr, Env1 ) = fun-call-tc( _expr, Env1 )

[e4]  is-simple-expr?( _id, E.context ) != TRUE,
      is-var-access?( _id, E.context ) != TRUE,
      is-function-call?( _id, E.context ) != TRUE
      ===========================================
      expr-tc( _id, E ) = add-error
        [Identifier _id is not a constant, variable, or function.] to E


{ Type check expressions constructed of arithmetic and relational
  operators. }

[ot1] arithm-op-table =
              Operator  | Operand1  | Operand2  | Result
              ----------+-----------+-----------+--------
                 +      | int       | int       | int
                 +      | int       | real      | real
                 +      | real      | int       | real
                 +      | real      | real      | real
                 +      | int       | undef     | int
                 +      | real      | undef     | real
                 +      | set       | set       | set

                 -      | int       | int       | int
                 -      | int       | real      | real
                 -      | real      | int       | real
                 -      | real      | real      | real
                 -      | int       | undef     | int
                 -      | real      | undef     | real
                 -      | set       | set       | set

                 *      | int       | int       | int
                 *      | int       | real      | real
                 *      | real      | int       | real
                 *      | real      | real      | real
                 *      | set       | set       | set

                 /      | int       | int       | real
                 /      | int       | real      | real
                 /      | real      | int       | real
                 /      | real      | real      | real

                div     | int       | int       | int
                div     | int       | int       | int

                mod     | int       | int       | int
                mod     | int       | int       | int

                and     | boolean   | boolean   | boolean
                or      | boolean   | boolean   | boolean
                not     | boolean   | undef     | boolean
```

```
              ;

[ot2] relational-op-table =
              Operator  | Operand1  | Operand2  | Result
              ----------+-----------+-----------+--------
                in      | ordinal   | set       | boolean
                <       | string    | string    | boolean
                <       | simple    | simple    | boolean
                <=      | simple    | simple    | boolean
                <=      | string    | string    | boolean
                <=      | set       | set       | boolean
                >       | string    | string    | boolean
                >       | simple    | simple    | boolean
                >=      | simple    | simple    | boolean
                >=      | string    | string    | boolean
                >=      | set       | set       | boolean
                =       | simple    | simple    | boolean
                =       | pointer   | pointer   | boolean
                =       | set       | set       | boolean
                =       | string    | string    | boolean
                <>      | simple    | simple    | boolean
                <>      | pointer   | pointer   | boolean
                <>      | set       | set       | boolean
                <>      | string    | string    | boolean
              ;

{ Type check the operators in the expressions. }
{ To do this, find the types of the operators, and check in a type-table
  whether these operator types can be matched in that type-table.
  If not raise an error. }

[u1]  expr-tc( +   _expr, E ) = unop-tc( +, _expr, E )
[u2]  expr-tc( -   _expr, E ) = unop-tc( -, _expr, E )
[u3]  expr-tc( not _expr, E ) = unop-tc( not, _expr, E)

[u4]  expr-tc( _expr, E1 ) = E2,
      match-ops(arithm-op-table, _Operator, map(E2.result), undef, E2) = E3
      ================================================================
      unop-tc(_Operator, _expr, E1) = E3


[b1]  expr-tc( _expr1 /   _expr2, E ) = binop-tc( /,   _expr1, _expr2, E )
[b2]  expr-tc( _expr1 div _expr2, E ) = binop-tc( div, _expr1, _expr2, E )
[b3]  expr-tc( _expr1 mod _expr2, E ) = binop-tc( mod, _expr1, _expr2, E )
[b4]  expr-tc( _expr1 and _expr2, E ) = binop-tc( and, _expr1, _expr2, E )
[b5]  expr-tc( _expr1 or  _expr2, E ) = binop-tc( or,  _expr1, _expr2, E )
[b6]  expr-tc( _expr1 +   _expr2, E ) = binop-tc( +,   _expr1, _expr2, E)
[b7]  expr-tc( _expr1 *   _expr2, E ) = binop-tc( *,   _expr1, _expr2, E)
[b8]  expr-tc( _expr1 -   _expr2, E ) = binop-tc( -,   _expr1, _expr2, E)

[b9]  expr-tc( _expr1, E1 ) = E2,
      expr-tc( _expr2, E2 ) = E3,
      match-ops(arithm-op-table,
        _Operator, map(E2.result), map(E3.result), E3) = E4,
      check-sets (E2.result, E3.result) in E4 = E5
```

```
       ========================================================
       binop-tc(_Operator, _expr1, _expr2, E1) = E5


 [r1]  expr-tc( _expr1 =  _expr2, E ) = relop-tc( =,  _expr1, _expr2, E )
 [r2]  expr-tc( _expr1 <> _expr2, E ) = relop-tc( <>, _expr1, _expr2, E )
 [r3]  expr-tc( _expr1 <  _expr2, E ) = relop-tc( <,  _expr1, _expr2, E )
 [r4]  expr-tc( _expr1 >  _expr2, E ) = relop-tc( <,  _expr1, _expr2, E )
 [r5]  expr-tc( _expr1 <= _expr2, E ) = relop-tc( <=, _expr1, _expr2, E )
 [r6]  expr-tc( _expr1 >= _expr2, E ) = relop-tc( >=, _expr1, _expr2, E )

 [r8]  expr-tc( _expr1, E1 ) = E2,
       expr-tc( _expr2, E2 ) = E3,
       E2.result _Operator E3.result should be compatible, numbers,
         or of the same set in E3 = E4,
       match-ops(relational-op-table,
         _Operator, rmap(E2.result), rmap(E3.result), E3) = E4
       ===================================================
       relop-tc(_Operator, _expr1, _expr2, E1) = E4

 [r9]  expr-tc( _expr1, E1 ) = E2,
       expr-tc( _expr2, E2 ) = E3,
       match-ops(relational-op-table, in,
         omap(E2.result), rmap(E3.result), E3) = E4,
       E2.result should be the set-component of E3.result in E4 = E5
       ===========================================================
       expr-tc( _expr1 in  _expr2, E1) = E5


{ If the operators of +, -, or * where sets, extra checks need to be done.
  The sets must be of the same component-type.
  The result should be set (if possible) to the non-empty list argument }

[cs0]  is-set-type?(T1) | is-set-type?(T2) != TRUE
       ==========================================
       check-sets(T1, T2) in E1 = E1

[cs1]  is-set-type?(T1) & is-set-type?(T2) = TRUE,
       T1 and T2 should be the same set-of-T type in E1 = E2
       ===================================================
       check-sets(T1, T2) in E1 = set-result of E2 to get-result(T1, T2)

[cs2]  get-result(T1, T2) = T1 when is-non-empty-set-type?(T1) = TRUE
[cs3]  get-result(T1, T2) = T2 when is-non-empty-set-type?(T2) = TRUE
[cs3]  get-result(T1, T2) = T1 when
         is-empty-set-type?(T1) & is-empty-set-type?(T2) = TRUE


 [f1]  look-for(Optor, Op1, Op2, OpTable) = <TRUE, Result-type>
       =================================================
       match-ops(OpTable, Optor, Op1, Op2, E) =
         set-result of E to map(Result-type)

 [f2]  look-for(Optor, Op1, Op2, OpTable) = <FALSE, Result-type>
       ===========================================================
```

```
        match-ops(OpTable, Optor, Op1, Op2, E) =
          add-error [Illegal operand types for operator Optor.] to E


  [ma1] map(T) = int when T = integer-type
  [ma2] map(T) = boolean when T = boolean-type
  [ma3] map(T) = real when T = real-type
  [ma4] map(T) = set when is-set-type?(T) = TRUE

  [mb1] map(int) = integer-type
  [mb2] map(real) = real-type
  [mb3] map(boolean) = boolean-type

  [rm1] rmap(T) = simple when is-simple-type?(T) = TRUE
  [rm2] rmap(T) = set when is-set-type?(T) = TRUE
  [rm3] rmap(T) = pointer when is-pointer-type?(T) = TRUE
  [rm4] rmap(T) = string when is-string-type?(T) = TRUE

  [om1] omap(T) = ordinal when is-ordinal-type?(T) = TRUE
end module TC-expr


module TC-stat
%% Type check statements; Goto statements are not dealt with.
imports TC-expr

exports
  context-free syntax
    stat-tc "(" {STATEMENT ";"}+ "," ENV ")"            -> ENV


hiddens
  sorts CASE-OUT
  context-free syntax
    lhs-tc "(" VARIABLE-ACCESS "," ENV ")"              -> ENV
    case-lists-tc "(" {CASE-LIST-ELT";"}+ ","
      CONTEXT "," VAL-LIST "," ENV ")"                  -> VAL-OUT

  variables
    [_]stat[123]*   -> STATEMENT       [_]"stat*"       -> { STATEMENT ";"}*
    [_]unl-stat     -> UNL-STAT        FunDef           -> CONTEXT
    VarDef          -> CONTEXT         "Cl-elts+"       -> {CASE-LIST-ELT ";"}+
    Cl-elt          -> CASE-LIST-ELT   "(Down)To"        -> DOWN-TO

equations

  [s2]  stat-tc(, E) = E
  [s3]  stat-tc(_stat; _stat*, E) =
          stat-tc(_stat*, stat-tc(_stat, E))

  { Procedure call }
  [s5]  set-construct of E0 to "_id _act-par-list" = E1,
        proc-call-tc( _id _act-par-list, E1 ) = E2
        ===================================================
        stat-tc( _id _act-par-list, E0 ) = E2
```

```
{ Compound statement }
[s6]  stat-tc(begin _stat* end, E) = stat-tc( _stat*, E)

{ Assignment statement }
[as1] set-construct of E0 to "_var-acc := _expr" = E1,
      lhs-tc(_var-acc, E1) = E2,
      expr-tc(_expr, E2) = E3,
      E2.result should be assignment-compatible with E3.result in E3 = E4
      ===============================================================
      stat-tc(_var-acc := _expr , E0) = E4

[as2] _id is the name of the inner function-block in E.context? = TRUE,
      find _id in E.context = FunDef
      ================================================
      lhs-tc(_id, E) = set-result of E to FunDef.type.canonical

[as3] _id is the name of the inner function-block in E1.context? != TRUE,
      expr-tc(_id, E1) = E2,
      E2.result defines _id as a variable? = TRUE,
      take care of possibly threatening variable _id in E2 = E3
      ==================================================================
      lhs-tc(_id, E1) = set-result of E3 to E2.result.type.canonical

[as4] is-just-an-identifier?(_var-acc) != TRUE
      ====================================
      lhs-tc(_var-acc, E1) = var-access-tc(_var-acc, E1)

{ If statement }
[is1] set-construct of E0 to "if _expr then <STATEMENT>" = E1,
      expr-tc(_expr, E1) = E2,
      stat-tc(_stat, E2) = E3,
      E2.result should be Boolean in E3 = E4
      ===================================
      stat-tc(if _expr then _stat, E0) = E4

[is2] set-construct of E0 to
              "if _expr then <STATEMENT> else <STATEMENT>" = E1,
      expr-tc(_expr, E1) = E2,
      stat-tc(_stat1, E2) = E3,
      stat-tc(_stat2, E3) = E4,
      E2.result should be Boolean in E4 = E5
      ===================================
      stat-tc(if _expr then _stat1 else _stat2, E0) = E5

{ With statement }
[ws1] stat-tc(with _var-acc, _var-acc+ do _stat, E) =
        stat-tc(with _var-acc do with _var-acc+ do _stat, E)

[ws2] set-construct of E0 to "with _var-acc do <STATEMENT>" = E1,
      var-access-tc(_var-acc, E1) = E2,
      E2.result.type should be a record in E2 = E3,
      new-block-with-fields(E2.result.type, E3) = E4,
        stat-tc(_stat, E4) = E5,
      leave inner block of E5 = E6
```

```
      ====================================================
      stat-tc( with _var-acc do _stat, E0 ) = E6

{ Case statement }
[cs1] stat-tc(case _expr of Cl-elts+; end, E) =
        stat-tc(case _expr of Cl-elts+ end, E)

[cs2] set-construct of E0 to "case _expr of <CASE-LIST-ELT> end" = E1,
      expr-tc(_expr1, E1) = E2,
      E2.result should be ordinal in E2 = E3,
      case-lists-tc( Cl-elts+, E2.result, [], E3) = <Val-list, E4>
      ==================================================
      stat-tc(case _expr1 of Cl-elts+ end, E0) = E4

{ All constants in the case statement should be of the same type, and
  should have different Values.}
[cs3] case-lists-tc( Cl-elt, T1, Val-list1, E1) = <Val-list2, E2>
      ==========================================================
      case-lists-tc( Cl-elt; Cl-elts+, T1, Val-list1, E1) =
        case-lists-tc( Cl-elts+, T1, Val-list2, E2)

[cs5] const-val-list-tc( _const+, T1, Val-list1, E1) = <Val-list2, E2>,
      stat-tc( _stat, E2 ) = E3
      =================================
      case-lists-tc( _const+: _stat, T1, Val-list1, E1 ) = <Val-list2, E3>

{ Repetive statements }
[rs1] set-construct of E0 to "repeat <STATEMENT> until _expr" = E1,
      expr-tc( _expr, E1 ) = E2,
      stat-tc( _stat*, E2 ) = E3,
      E2.result should be Boolean in E3 = E4
      =====================================
      stat-tc( repeat _stat* until _expr, E0 ) = E4

[ws1] set-construct of E0 to "while _expr do <STATEMENT>" = E1,
      expr-tc( _expr, E1 ) = E2,
      stat-tc( _stat, E2 ) = E3,
      E2.result should be Boolean in E3 = E4
      =====================================
      stat-tc( while _expr do _stat, E0 ) = E4


[fs2] set-construct of E0 to
              "for _id := _expr1 (Down)To _expr2 do <STATEMENT>" = E1,

      _id should be a variable in inner block of E1 = E2,
      _id should not be a control variable in E2 = E3,
      _id should not be a possibly threatening variable in E3 = E4,

      var-access-tc(_id, E4) = E5,
      mark-variable(_id, control-variable, E5) = E6,
      expr-tc(_expr1, E6) = E7,
      expr-tc(_expr2, E7) = E8,
      stat-tc(_stat, E8) = E9,
      mark-variable(_id, , E9) = E10,
```

```
        E5.result should be ordinal in E10 = E11,
        E7.result should be assignment-compatible with E5.result in E11 = E12,
        E8.result should be assignment-compatible with E5.result in E12 = E13
        =================================================================
        stat-tc( for _id := _expr1 (Down)To _expr2 do _stat, E0) = E13


  { It is only checked whether labels are declared indeed. }
  [ls1] _label should be declared in E1 = E2,
        stat-tc(_unl-stat, E2) = E3
        ===================================
        stat-tc(_label: _unl-stat, E1) = E3

  [ls2] stat-tc(goto _label, E) = _label should be declared in E
end module TC-stat


module TC-block
%% Type check a block of a Pascal Program (ISO section 6.2.1.)
%% Most of the work is done in the checks for procedure and function
%% declarations:
%% The heading must be correct (all parameter names different, all type
%% identifiers legal), and the block must be correct.
%% _All_ forwarded headings should be followed by an identification
%% associating a body with that heading.

imports TC-stat

exports
  context-free syntax
    block-tc "(" BLOCK "," ENV ")"                           -> ENV

hiddens
  sorts FORW-DECLS PROFUN-OUT
  context-free syntax
    labels-tc "(" LABELS "," ENV ")"                         -> ENV
    consts-tc "(" CONSTS "," ENV ")"                         -> ENV
    types-tc "(" TYPES "," ENV ")"                           -> ENV
    vars-tc "(" VARS "," ENV ")"                             -> ENV
    procs-or-funs-tc "(" PROCS-OR-FUNS "," ENV ")"           -> ENV

    profuns-tc"("{PROC-OR-FUN";"}* "," ENV ")"               -> ENV
    profuns-tc"("{PROC-OR-FUN";"}* "," FORW-DECLS "," ENV ")" -> PROFUN-OUT
    block-heading-tc "(" HEADING "," FORW-DECLS "," ENV ")"   -> PROFUN-OUT
    forwarded-profun-tc "(" PROCEDURE-HEADING ","
      FORW-DECLS "," ENV ")"                                 -> PROFUN-OUT
    forwarded-profun-tc "(" FUNCTION-IDENTIFICATION ","
      FORW-DECLS "," ENV ")"                                 -> PROFUN-OUT
    forw-heading-tc "(" HEADING "," ENV ")"                  -> ENV
    param-list-tc "(" FORMAL-PAR-LIST "," ENV ")"            -> ENV

    "[" { HEADING ","}* "]"                                  -> FORW-DECLS
    "<" FORW-DECLS "," ENV ">"                               -> PROFUN-OUT
```

```
  remove IDENT from FORW-DECLS                              -> FORW-DECLS
  get-id-list "(" FORW-DECLS ")"                            -> IDENT-LIST
  is-forward-list-empty "?" "(" FORW-DECLS ")"              -> BOOLEAN
  IDENT is a forwarded procedure in FORW-DECLS "?"          -> BOOLEAN
  forward list FORW-DECLS should be empty in ENV            -> ENV

 variables
  Id-list       -> IDENT-LIST           Block              -> BLOCK
  Labels        -> LABELS               Consts             -> CONSTS
  Const-defs    -> {CONST-DEF ";"}*     [_]Types           -> TYPES
  Type-defs     -> {TYPE-DEF ";"}*      Vars               -> VARS
  Var-decls     -> {VAR-DECL ";"}*      Procs-or-funs      -> PROCS-OR-FUNS
  Profuns       -> { PROC-OR-FUN ";"}*  Profun-decl        -> PROC-OR-FUN
  Comp-stat     -> COMP-STAT            F[orw\-list]*[0-9]* -> FORW-DECLS
  Forwards      -> { HEADING ","}*      Heading            -> HEADING
  Headings[0-9]* -> { HEADING ","}*     Heading-plus       -> {HEADING ","}+
  Fun-heading   -> FUNCTION-HEADING     Proc-heading       -> PROCEDURE-HEADING

equations

 [b1]  block-tc( Labels Consts _Types Vars Procs-or-funs Comp-stat , E ) =
         stat-tc( Comp-stat,
           procs-or-funs-tc( Procs-or-funs,
             vars-tc( Vars,
               types-tc( _Types,
                 consts-tc( Consts,
                   labels-tc( Labels, E ))))))

 [b2]  labels-tc(, E) = E
 [b3]  labels-tc(label _label*;, E) = label-decls-tc(_label*, E)
 [b3]  consts-tc(, E) = E
 [b4]  consts-tc( const Const-defs;, E) = const-defs-tc(Const-defs, E)
 [b5]  types-tc(, E) = E
 [b6]  types-tc( type Type-defs;, E) = type-defs-tc(Type-defs, E)
 [b7]  vars-tc(, E) = E
 [b8]  vars-tc( var Var-decls;, E) = var-decls-tc(Var-decls, E)
 [b9]  procs-or-funs-tc(, E) = E
 [b10] procs-or-funs-tc(Profuns;, E) = profuns-tc(Profuns,E)


 { Type check procedure and function declarations }
 [pf1] profuns-tc(Profuns, [], E1) = <Forw-list, E2>,
       forward list Forw-list should be empty in E2 = E3
       ===========================================
       profuns-tc(Profuns, E1) = E3

 [pf2] profuns-tc(Profun-decl, Forw-list1, E1) = <Forw-list2, E2>,
       profuns-tc(Profuns, Forw-list2, E2) = <Forw-list3, E3>
       ====================================================
       profuns-tc(Profun-decl; Profuns, Forw-list1, E1) = <Forw-list3, E3>

 [pf3] profuns-tc(,F,E) = <F,E>

 { A normally defined procedure or function: }
 [pf5] block-heading-tc( Heading, Forw-list1, E1 ) = <Forw-list2, E2>,
```

```
      block-tc( Block, E2 ) = E3,
      leave inner block of E3 = E4
      ===============================
      profuns-tc(Heading; Block, Forw-list1, E1) = <Forw-list2, E4>

{ A declaration with a forward directive; add heading to Forw-list and to
   environment. }
[pf4] forw-heading-tc( Heading, E1 ) = E2
      ==============================
      profuns-tc( Heading; forward, [Forwards], E1 ) =
        < [Forwards, Heading], E2 + Heading>

{ A function already declared forward, whose body is given now. }
[pf5] forwarded-profun-tc(function _id, Forw-list1, E1) = <Forw-list2, E2>,
      block-tc( Block, E2 ) = E3,
      leave inner block of E3 = E4
      =====================================
      profuns-tc(function _id; Block, Forw-list1, E1) = <Forw-list2, E4>

{ Type check a heading of a forwarded function or procedure.
  Just check the correctness of the parameters and their types. }
[fh1] _id1 should not be declared in inner block of E1 = E2,
      E2 + block-mark = E3,
      param-list-tc(_form-par-list, E3) = E4,
      _id2 should be a type in E4 = E5,
      leave inner block of E5 = E6
      ========================================
      forw-heading-tc(function _id1 _form-par-list: _id2, E1) = E6

[fh2] _id1 should not be declared in inner block of E1 = E2,
      E2 + block-mark = E3,
      param-list-tc(_form-par-list, E3) = E4,
      leave inner block of E4 = E5
      ========================================
      forw-heading-tc(procedure _id1 _form-par-list, E1) = E5

{ Type check a heading of a function or procedure with a block.
  The normal case is a new procedure or function, possibly with parameters.
  In the "procedure _id; block" case, the _id may either refer to an
  already forwarded procedure, or to a new procedure without parameters. }
[bh1] _id1 should not be declared in inner block of E1 = E2,
      E2 + function _id1 _form-par-list : _id2 + block-mark = E3,
      param-list-tc(_form-par-list, E3) = E4,
      _id2 should be a type in E4 = E5
      ==========================================
      block-heading-tc(function _id1 _form-par-list: _id2, Forw-list, E1) =
        <Forw-list, E5>

[bh2] _id is a forwarded procedure in Forw-list? != TRUE,
      _id should not be declared in inner block of E1 = E2,
      E2 + procedure _id _form-par-list + block-mark = E3,
      param-list-tc(_form-par-list, E3) = E4
      ==========================================
      block-heading-tc(procedure _id _form-par-list, Forw-list, E1) =
        <Forw-list, E4>
```

```
[bh3] _id is a forwarded procedure in Forw-list? = TRUE
      ================================================
      block-heading-tc(procedure _id, Forw-list, E) =
        forwarded-profun-tc(procedure _id, Forw-list, E)


{ Check a function or procedure identification: i.e. a function or
  procedure already forwarded, and whose body is given by now. }
[fw1] remove _id from Forw-list1 = Forw-list2,
      _id should be a function in inner block of E1 = E2,
      Fun-heading = E2.result.entry,
      _form-par-list = E2.result.formal-parameters,
      E2 + block-mark + Fun-heading = E3,
      param-list-tc(_form-par-list, E3) = E4
      =====================================
      forwarded-profun-tc(function _id, Forw-list1, E1) = <Forw-list2, E4>

[fw1] remove _id from Forw-list1 = Forw-list2,
      _id should be a procedure in inner block of E1 = E2,
      Proc-heading = E2.result.entry,
      _form-par-list = E2.result.formal-parameters,
      E2 + block-mark + Proc-heading = E3,
      param-list-tc(_form-par-list, E3) = E4
      =====================================
      forwarded-profun-tc(function _id, Forw-list1, E1) = <Forw-list2, E4>

{ Check a formal parameter list.
  Add the parameters as variables to the block: guarantees the uniqueness
  of the names. Check whether the types are OK. }
[pr1] (_form-par1*; _id1, _id+: _id3; _form-par2*) =
        (_form-par1*; _id1: _id3; _id+: _id3; _form-par2*)
[pr2] (_form-par1*; var _id1, _id+: _id3; _form-par2*) =
        (_form-par1*; var _id1: _id3; var _id+: _id3; _form-par2*)


[pl1] param-list-tc(,E) = E
[pl2] param-list-tc((_form-par; _form-par+), E) =
        param-list-tc((_form-par+), param-list-tc((_form-par), E))

[pl3] _id1 should not be declared in inner block of E1 = E2,
      _id2 should be a type in E2 = E3,
      E3.result should be allowed as file-component in E3 = E4
      ==============================================
      param-list-tc((_id1: _id2), E1) = E3

[pl4] _id1 should not be declared in inner block of E1 = E2,
      _id2 should be a type in E2 = E3
      ================================
      param-list-tc((var _id1: _id2), E1) = E3

[pl5] param-list-tc((Fun-heading), E) = forw-heading-tc(Fun-heading, E)
[pl6] param-list-tc((Proc-heading), E) = forw-heading-tc(Proc-heading, E)
```

```
  { Operations on lists of forwarded headings: }
  [fl1] _id is a forwarded procedure in
         [Headings1, procedure _id _form-par-list, Headings2]? = TRUE
  [fl2] remove _id from [Headings1, procedure _id _form-par-list, Headings2] =
         [Headings1, Headings2]
  [fl3] remove _id1 from [Headings1,function _id1 _form-par-list:_id2,Headings2] =
         [Headings1, Headings2]

  [fl4] is-forward-list-empty?([]) = TRUE

  [fl5] is-forward-list-empty?(Forw-list) = TRUE
        ==============================================
        forward list Forw-list should be empty in E = E

  [fl6] is-forward-list-empty?(Forw-list) != TRUE,
        Id-list = get-id-list(Forw-list)
        ==============================================
        forward list Forw-list should be empty in E =
          add-error [No body given for forwarded identifiers: Id-list.] to E


  [fl7] get-id-list([procedure _id _form-par-list]) = [_id]
  [fl8] get-id-list([function _id _form-par-list :_id2]) = [_id]

  [fl9] get-id-list([Heading]) = [_id],
        get-id-list([Heading-plus])= [_id+]
        ======================================
        get-id-list([Heading, Heading-plus]) = [_id, _id+]

  [f20] get-id-list([]) = []
end module TC-block


module TC-program
%% Typecheck Pascal programs, according to ISO section 6.2.1. and 6.10.
imports TC-block

exports
  context-free syntax
    tc "(" PROGRAM ")"                                      -> ERRORS

hiddens
  context-free syntax
    params-tc-before-block "(" PROGRAM-HEADER "," ENV ")"     -> ENV
    params-tc-after-block  "(" PROGRAM-HEADER "," ENV ")"     -> ENV
    params-tc-before-block "(" { IDENT ","}* "," ENV ")"      -> ENV
    params-tc-after-block  "(" { IDENT ","}* "," ENV ")"      -> ENV
    "check-input/output" "(" { IDENT ","}* "," ENV ")"        -> ENV
    "is-input/output-parameter" "?" "(" IDENT ")"             -> BOOLEAN
]
  variables
    Program       -> PROGRAM                  [_]header      -> PROGRAM-HEADER
    Prog-params   -> PROGRAM-PARAMS           Block          -> BLOCK
    [_]pars       -> { IDENT ","}*            [_]par         -> IDENT
```

```
equations

  [pr1] params-tc-before-block( _header, new-env ) = E1,
        block-tc( Block, E1 ) = E2,
        params-tc-after-block(  _header, E2 ) = E3
        ===============================================
        tc( _header; Block. ) = E3.errors


  { Check the parameter list of the program header. }
  [pb1] set-construct of E1 to "program _id (_pars)" = E2,
        (_pars) should not have double names in E2 = E3,
        check-input/output( _pars, E3 ) = E4
        ====================================
        params-tc-before-block( program _id (_pars), E1 ) = E4

  [pb2] params-tc-before-block( program _id, E) = E


  { If input or output are given as program parameters, they are supposed
    to be implicitly defined as variables of type char. Make this explicit
    by adding them to the environment. }
  [io1] check-input/output(_par, _pars, E) =
          check-input/output(_pars, check-input/output(_par, E))
  [io2] check-input/output(input, E) = E + var input: text
  [io3] check-input/output(output, E) = E + var output: text
  [io4] check-input/output( _par, E ) = E
          when is-input/output-parameter?(_par) != TRUE
  [io5] check-input/output(, E ) = E
  [io6] is-input/output-parameter?(input) = TRUE
  [io7] is-input/output-parameter?(output) = TRUE


  { All program parameters should be defined as variables in the
    program block. This is checked after the program block has been checked. }
  [pa0] set-construct of E1 to "program _id (_pars)" = E2,
        params-tc-after-block(_pars, E2) = E3
        =========================================
        params-tc-after-block(program _id (_pars), E1) = E3

  [pa1] params-tc-after-block(_par, _pars, E) =
          params-tc-after-block(_pars, params-tc-after-block(_par, E))
  [pa2] params-tc-after-block(_par, E) =  _par should be a variable in E
  [pa3] params-tc-after-block(,E) = E
end module TC-program
```

D An Example

There is no way to give a formal proof of the correctness of the algebraic specification in this document with respect to the ISO specification in natural language. However, in order to strengthen the idea that our specification is correct, we will give a small example of how it works on a Pascal program containing some static semantic errors. We give a program to compute the greatest common divisor of two natural numbers using Euclid's algorithm (see, e.g., [Dij76]).

```
{ 1}  program Euclid (output);
{ 2}
{ 3}  type positive = 1 .. maxint;
{ 4}  var x,y: positive;
{ 5}
{ 6}  begin
{ 7}    writeln('Enter two naturals: ');
{ 8}    readln(x, y);
{ 9}
{10}    while x <> y do
{11}      if x < y then x := x - 'y'
{12}      else y := y - x;
{13}
{14}    writeln('Greatest Common Divisor:', z);
{15}  end.
```

Note that the programmer has made three errors: in line 1 he forgot mentioning the program parameter `input` although he is using the function `readln`; in line 11 he by accident quoted `y` in `x := x - 'y'`; and in line 14 he typed `writeln( ... z)` instead of `writeln( ... x)`.

Putting the function `tc` around this program, and passing the term to the term rewriting system that can be derived from the entire specification, will have the following result. In each paragraph, we mention the module name, and the tags of the most important equations involved.

`TC-program, [pr1]`: Starting in an empty environment (no errors, no declarations), the program header and the program block are checked.

`TC-program, [pb1], [io2]`: Since the header contains `input` as a program parameter, the declaration `var input: text` is added to the environment.

`TC-block, [b1], [b6], [b8]`: The checks for the block are split up into consecutive checks of the type definitions (line 3), the variable declarations (line 4), and the statements (line 6-15).

`TC-defs, [t4]`: The type definition is broken up into checks for the left and right-hand side, after which the type definition `type positive = 1 .. maxint` is added to the environment.

`ShouldBe, [md1]` and `Context-search, [ex2], [fi1]`: Since the identifier `positive` is not yet declared in this block, the left hand side is correct.

`TC-defs, [ts1], [c5], [c6]`: The right-hand side involves checking a subrange type declaration. Both constants (`1` and `maxint`) are checked, and `maxint` is found in the part of the context of the environment containing the predefined identifiers. Since the types of both constants are the same and ordinal, and since the value of `maxint`, 32768, is more than the value of `1`, 1, the subrange declaration is correct.

`TC-defs, [v3], [v4]`: The checks for the variable declaration of line 4 are split up into checks for left and right hand side of `var x: positive` and `var y: positive`. Since the declaration for `positive` is found in the context of the environment, and neither `x` nor `y` are declared, all these checks succeed, and the declarations for `x` and `y` are added to the environment.

`TC-stat, [s6], [s3]`: The compound statement is thus checked in an environment containing the definitions of the program. The checks for the compound statements are split up into checks for the individual statements.

`TC-stat, [s5]` and `TC-expr-calls, [p1]`: The first statement to be checked is a procedure call, and since the identifier `writeln` is found and detected as a predefined identifier in the

environment, the type check function for special procedures is called.

TC-spec-procs [ln7], [ln9], [wr6]: Since the first argument of writeln is not a file variable, writeln('...') is rewritten to writeln(output, '...'), which in turn is broken up into writeln(output) and write(output, '...'). Because output is a variable of type text, and the argument of write is a string, it is concluded that the statement is correct. Consequently the environment is kept unchanged.

TC-spec-procs [ln2], [ln4] and [ea7]: The readln of line 8 is checked in a similar way: This time, however, variable input is not declared (since it was not mentioned in the program header). Therefore, the checks of the actual parameters produce the messages readln(x,y): Identifier input not declared. and readln(x,y): Identifier input is not a variable. Moreover, no type can be found for the file argument, producing readln(x,y): Types are not assignment-compatible. Finally, the checks for a single readln with one arguments raises [ln5] will add the message readln(x,y): input is not a text file.

So line 8 of the program causes 4 error messages to be added to the environment. Note that these messages are prefixed by the construct field of the environment, which is set in the checks for the statements.

TC-stat, [ws1]: For the while statement, the condition and body are checked, after which it is confirmed that the type of the expression is of type Boolean.

TC-expr, [r2], [r8], [e2], [rm1]: For the expression x <> y, first the two arguments are checked, detecting that they are expressions of type integer. Thus, the arguments are simple types, and the entry <> | simple | simple | boolean is found. Consequently, it is concluded that the expression is correct and of type Boolean.

TC-stat [is2] [as1] [as3], TC-expr-vars [v1] and TC-expr [f2]: The body of the while statement is the if statement of lines 11 and 12. The condition of the if statement is checked similar to the condition of the while statement, just as the expressions on the right hand sides of the assignment statements in the conclusions of the if statement. However, x - 'y' results in trying to lookup an entry that does not exist, so an error message is added: x := x - 'y': Illegal operand types for operator -. Consequently, it cannot be concluded that the types of the first assignment are compatible, resulting in: x := x - 'y': Types are not assignment-compatible.

TC-spec-proc, [wr6], and TC-expr, [e4] In the last writeln statement (the check of which is, of course, similar to the check of the writeln of line 7), the identifier z is not known. The several constraints thus raise messages Identifier z not declared. and Identifier z is not a constant, variable or function. The check for the write does not know one of its argument types and complains with Number, character, boolean, or string type expected.

Finally, since in module Errors equation [er1] eliminates double error messages, the total list of errors resulting from equation [pr1] is:

```
readln(x,y): Identifier input not declared.
readln(x,y): Identifier input is not a variable.
readln(x,y): Types are not assignment-compatible.
readln(x,y): input is not a text file.

x := x - 'y': Illegal operand types for operator -.
x := x - 'y': Types are not assignment-compatible.

writeln('Greatest Common Divisor:', z): Identifier
    z not declared.
writeln('Greatest Common Divisor:', z): Identifier
    z is not a constant, variable, or function.
writeln('Greatest Common Divisor:', z): Number,
    character, boolean, or string type expected.
```