CMU-CS-78-133

TARTAN

Language Design for the Ironman Requirement: Reference Manual

Mary Shaw
Paul Hilfinger
Wm. A. Wulf

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1978

Abstract: Tartan is an experiment in language design. The goal was to determine whether a "simple" language could meet substantially all of the Ironman requirement for a common high-order programming language.

We undertook this experiment because we believed that all the designs done in the first phase of the DOD effort were too large and too complex. We saw that complexity as a serious failure of the designs: excess complexity in a programming language can interfere with its use, even to the extent that any beneficial properties are of little consequence. We wanted to find out whether the requirements inherently lead to such complexity or whether a substantially simpler language would suffice.

Three ground rules drove the experiment. First, no more than two months -- April 1 to May 31 -- would be devoted to the project. Second, the language would meet all the Ironman requirements except for a few points at which it would anticipate Steelman requirements. Further, the language would contain no extra features unless they resulted in a simpler language. Third, simplicity would be the overriding objective.

The resulting language, Tartan, is based on all available information, including the designs already produced. The language definition is presented here; a companion report provides an overview of the language, a number of examples, and more expository explanations of some of the language features.

We believe that Tartan is a substantial improvement over the earlier designs, particularly in its simplicity. There is, of course, no objective measure of simplicity, but the syntax, the size of the definition, and the number of concepts required are all smaller in Tartan.

Moreover, Tartan substantially meets all of the Ironman requirement. (The exceptions lie in a few places where we anticipated Steelman requirements and where details are still missing from this report.) Thus, we believe that a simple language can meet the Ironman requirement. Tartan is an existence proof of that.

We must emphasize again that this effort is an experiment, not an attempt to compete with DOD contractors. Tartan is, however, an open challenge to the Phase II contractors: The language can be at least this simple! Can you do better?

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 (monitored by the Air Force Office of Scientific Research).

Tartan Reference Manual

1. Basic Concepts and Philosophy

A **program** is a piece of text that describes a sequence of actions intended to effect a computation. The process of "executing a program" to obtain this effect is called **elaboration** of the text.[1]

Programming languages are used for communicating programs, both between people and between people and machines. Although the program text is static, the concepts being communicated are dynamic. This dynamic nature of a computation can make it difficult to communicate the ideas underlying a program, and especially to communicate these ideas between people. To expedite the communication, we impose structure on the way languages are used. Although this structure restricts what can be written, it results in regular patterns for expressing decisions. The human reader benefits from this by developing expectations about how these ideas will be expressed.

Programming languages encourage the imposition of structure by providing notations for the structures whose use their designers wish to promote. During the process of language design, our beliefs about programming methodology and the state of language processing technology lead us to formulate concepts and structural rules. We then select syntactic forms and structuring features to emphasize these concepts. We expect that this will simplify the task of describing programs with the attributes we view as "good structure" and that programmers will, as a result, be encouraged to organize their programs this way.

We distinguish three dominant structures in Tartan programs: (1) the **lexical structure**, which organizes the static program text, (2) the **control structure**, which organizes the dynamic execution, and (3) the **data structure**, which organizes the information on which computations are performed.

- **Lexical structure** is a property of the program text. Programs are divided hierarchically into sections, called **lexical scopes**, that share information about data. Scope determines the interpretation of identifiers, so all the text in a given lexical scope shares the same vocabulary -- definitions, variables, etc. **Scope rules** permit some identifiers to be used with the same interpretation in several lexical scopes.

- The **control structure** of the program determines the order in which its statements are executed.

- The structure imposed on **data** involves the concepts of type, values, and variables. Ultimately, computations are performed on values; we take that notion to be primitive: values exist, and each has exactly one type, which determines the legal operations on the value. Values are stored in variables, which are objects produced by elaborating type definitions. Variables, too, have types; these types determine the sets of values that may legally be stored in the variables.

These fundamental structures interact in a number of ways. Two major interactions appear as the concepts of **extent** and **binding**. The control and lexical structures interact to determine extent. The extent of a variable is its lifetime -- the time during which it affects or is affected by the elaboration of the program. Binding rules are invoked by both lexical and control structures; they associate identifiers with program entities (objects, modules, routines, types, labels, and exceptions).

In Tartan, programs are composed of **definitions**, **declarations**, and **executable statements**. A definition binds an identifier to a module, routine (procedure, function, or process), type, or exception; it is processed during translation. A declaration binds an identifier to an **object** (i.e., a variable or value); it is processed at run time, usually to allocate storage. Executable statements are elaborated at run time to effect actual computations -- manipulation of values.

Lexical structure is imposed on Tartan programs by **blocks** and **modules**, which delimit lexical scopes. These scopes may be nested arbitrarily. Both constructs may use identifiers defined in other scopes; both may define identifiers that can be used in other scopes. Blocks and modules differ only

---

[1] We use the word "elaboration", in preference to "execution", to connote actions taken during translation as well as during execution. Elaboration may be thought of as an idealized, direct execution of the textual version of the program.

in their scope rules and in their effects on the extent of variables. Tartan has two scope rules:

- An **open** scope inherits (imports automatically) all the identifiers that are defined in its enclosing scope. It may not export any identifiers. Blocks are open scopes except when used as routine bodies.

- A **closed** scope inherits all identifiers that are defined in its enclosing scope except those for labels and nonmanifest objects.[1] It may explicitly **import** identifiers for objects, provided they have global extent. All modules are closed scopes, as are blocks when they are used as routine bodies. A closed scope that is a module may **export** identifiers that name variables, modules, routines, types, or exceptions.

Identifiers that are exported from an inner scope or imported from an outer scope have the status of identifiers defined in the scope. Redefinition of identifiers within a scope is not permitted; however, this does not prohibit **overloading** of routine names. In addition, the same identifier may be imported with different meanings from two different scopes. Such identifiers are qualified with the names of the modules in which they were defined; thus they are not duplicate definitions. Similarly, literals and constructors are qualified with their types to prevent ambiguity. In either case, the module or type qualifier may be omitted if no ambiguity arises.

In Tartan, extent is controlled exclusively by blocks. Only instantiated objects (variables, constants) have extent. Variables are instantiated by the elaboration of declarations (for named variables) and by **explicit construction** of variables having **dynamic types** (dynamically created variables). Named variables have extent coincident with the surrounding block. Dynamically created variables have extent coincident with the block containing the definitions of their dynamic types. Formal parameters of routines are considered to have extent coincident with the routine body.

Tartan provides a facility for making **generic definitions** of routines and modules. This allows the programmer to write a single textual definition that serves as an abbreviation for many closely-related specific definitions. The definitions may accept parameters; the parameters are completely processed during translation. The effect of using a generic definition is that of lexically substituting the definition in the program at the point of use.

The syntactic definition of Tartan uses conventional BNF with the following additions and conventions:

- Key words (reserved words) and symbols are denoted with **boldface**.

- Metasymbols are denoted by lower-case letters enclosed in angular brackets, e.g., "<stmt>".

- The symbols { and } (not in boldface) are meta-brackets and are used to group constructs in the meta-notation.

- Three superscript characters, possibly in combination with a subscript character, are used to denote the repetition of a construct (or a group of constructs enclosed in {}):
  "*" denotes "zero or more repetitions of"
  "+" denotes "one or more repetitions of"
  "#" denotes "precisely zero or one instance of".
  Since it is often convenient to denote lists of things that are separated by some single punctuation mark, we denote this by placing the punctuation mark directly below the repetition character.

The semantics of the language are described in English. In the interest of a compact and regular syntax, we have allowed syntactic constructs that are disallowed on semantic grounds. This is consistent with standard practice with respect to, for example, undeclared identifiers.

---

[1] Literals and identifiers for variables that are declared **manifest** are manifest objects; hence they are inherited.

2. Basic Structures

2.1. Primitive Expressions

```
<const>       ::= <digit>+ { . <digit>+ }* | true | false | nil | closed | open | mint | empty
                | <constructor> | <id> | <qual id> ' <const> | <type> ' <const> | <expr>
<constructor> ::= ( <expr>,* ) | { { <option> -> <expr> },+ ) | " <char>* "
```

Some examples are:

```
123.456
Color'green
true
Person'("Sam",21,male)
"efg"
(1..2->0.1, 3..4->0.5, others->1.0)
```

Primitive expressions form the basis for the recursive defintion of expressions. They are the elements referred to as constants, literals, and constructors in programming languages and as generators in algebras.

Constants and literals denote values. The type of a constant is determined by its declaration. The types of literals are determined as follows:

- A sequence of digits containing no decimal point is of type Int. Type Int is defined in terms of type fixed for each machine as described in Appendix I.1.

- A sequence of digits containing a decimal point is of type Real. Type Real is defined in terms of type float for each machine as described in Appendix I.1.

- If a sequence of digits, with or without a decimal point, is qualified by a fixed or float type or by a defined type that is ultimately defined in terms of fixed or float, the type of the literal is determined by the qualifier.

- True and false denote boolean values. Nil denotes the null value for any dynamic type. Open and closed denote values for latches. Empty denotes the empty set. Mint denotes an activation of any process in mint state.

- A character string containing one character is a literal of type char. Any other character string is a constructor of type string.

Literals and manifest expressions are evaluated during translation with the same algorithms and accuracy as are used during execution.

If an <id> is to be a <const>, it must have been declared const or be a member of an enumerated type. If an <expr> is to be a <const>, it must be a manifest expression.

The type of a constructor may be indicated by a prefixed qualifier. If the qualifier is omitted, the constructor is assumed to give the value of an array indexed with integers beginning at 1. Constructors are provided for composite and dynamic types.

- If the constructor has a record type, the <expr>s in parentheses give the field values in the order of their declaration.

- If the constructor has an array type, the parenthesized list gives the element values. If the constructor is a simple expression list, it gives the values in order from lowest index to highest. If the constructor uses the form with options, the expressions in the <option>s indicate the array position to which each value corresponds. The special constant others may appear as the last <option>; it will match any constant that is not included in any other <option>. The constructor form with options is legal only for arrays and for types ultimately defined in terms of arrays; the expressions in the <option>s must be manifest.

- If the constructor has a variant type, the first expression in the parenthesized list is the tag and the remainder of the list is a constructor for the corresponding variant.

- If the constructor has dynamic type, the result is a pointer to a new variable having the attributes supplied in the type qualifier and the value given by the parenthesized list.

A constructor containing no <expr> provides an uninitialized instance of the indicated type.

2.2. Identifiers

```
<var id>    ::= <qual id> | <var id> ( <actuals> ) | <var id> . <id> | <var id> ( <range> ) | Rep' <id>
<range>     ::= <expr> . . <expr> | <type>
<option>    ::= { <const> | <range> },+
<qual id>   ::= { <id> '}* <id>
<id>        ::= <letter> <letter or _ or digit>*
```

Some examples are:

```
Animal'Cat
V(3)
V(1..N)
Sam.Age
Ident_with_mark
```

Identifiers have no inherent meanings. They are associated with objects, routines, modules, types, statements, and exceptions. Declarations and definitions establish the meanings of identifiers within particular scopes.

Identifiers may be simple, or they may be qualified with module or type names in order to resolve ambiguity among names exported from several modules.

Identifiers that name objects are <var id>s. They may be simple identifiers, they may be qualified to indicate where they were defined, or they may name elements or substructures of composite structures.

- Simple <var id>s (i.e., <qual id>s used as <var id>s) are identifiers declared in variable declarations or by the <formals> in a routine header.

- The form <var id>(<actuals>), where <var id> denotes an array, denotes the element of that array indexed by the <actual>s. The types of the actuals must match the index types for the array.[1]

- The form <var id>(<actuals>), where <var id> denotes a variable of a variant type and the <actual>s consist of a single <expr>, indicates that the tag field of the <var id> must be <expr> and denotes the value of that option of the variant type. On the left side of an assignment, this form has the effect of setting the tag field; the expression on the right side of the assignment must be of compatible type.

- The form <var id>(<range>) denotes a subarray. The <var id> must denote an array and the limits of the <range> must match the declared type of the array's index set and be a subrange of the declared range. The subarray consists of the indicated elements of the <var id>, in the same order as they appear in the <var id>. If the index type of the array is fixed or defined in terms of fixed, the subarray is indexed by integers beginning with 1; otherwise it is indexed from the minimum value of the index set of the array.

- If <var id> denotes a record object, the form <var id>.<id> denotes the field named <id> in that record object. If <var id> denotes an object of dynamic type, then <var id>.<id> denotes the field named <id> in the record object pointed to by the value of <var id>; <var id> must not have the value nil. This form is also used to access the value of a variant tag or the attributes associated with the type of a value or variable. In addition, if T is a variable of dynamic type, T.all is the complete value (all components) of the object associated with T.

---

[1] Note that the index types include range restrictions.

- The form Rep'<id> is used in the same scope as the definition of the <id>'s type to indicate that the <id> is to be regarding as having the underlying type. This permits operations on the underlying type to be used for defining operations on the new type.

Identifiers that refer to definitions (e.g., of routines, types, or modules) are <qual id>s.

When an identifier is exported from a module, in the scope to which it is exported it is referred to by a <qual id> or <var id> constructed by prefixing the identifier with the name of the module from which it is exported. The qualifier is separated from the identifier with an apostrophe. Qualifiers may be omitted if no ambiguity results.

A <type> used as a range must be fixed, an enumerated type, or a defined type that is ultimately defined in terms of fixed or an enumeration.

2.3. Lexical Considerations

Spaces may be inserted freely between lexemes without altering the meaning of the program. An end-of-line is equivalent to a space and may not be part of a lexeme. At least one space must appear between any two adjacent lexemes composed of letters, digits, underbar, and decimal points. In identifiers, all characters are significant, but alphabetic case is not.

Comments are introduced by the character "!" and terminated by the next following end-of-line. They have no effect on the elaboration of the program.

Although the language as presented in this report takes advantage of characters that are not in the 64-character ASCII subset, simple substitution to map programs into that alphabet are defined in Appendix I.

3. Expressions

```
<expr>      ::= <unop>* <var id> | <unop>* <const> | <unop>* <func call>
              | <unop>* ( <expr> ) | ( <expr> ) . <id> | <expr> <binop> <expr>
<unop>      ::= - | ¬
<binop>     ::= * | / | + | - | < | ≤ | > | ≥ | = | ≠ | ∧ | cand | ∨ | cor | ↑
<func call> ::= <qual id> ( <actuals> )
<actuals>   ::= <expr>,*
```

Some examples are:

```
x
x + y
sin(x)
-(x*y + z*w)
(Root.Ptr).all
```

Expressions describe computations that yield values. The elaboration of an expression produces an object containing the value of the expression. The type of the object is determined by the following rules:

- The type of an <expr> that is a <var id>, <const>, <func call>, or selection of a field from a computed composite value is determined by the appropriate declaration (or rule for literals).

- The type of a parenthesized expression is the type of the expression inside the parentheses.

- The type of a binary infix expression or a unary expression is determined by the definition of the appropriate binary or unary operator definition. These operators represent invocations of functions that may be overloaded. The appropriate operator definition must therefore be determined on the basis of the types of the operands.

The usual operations are associated with the operators +, -, *, /, ↑, ¬, ∧, ∨, cand, cor, <, ≤, ≥, >, =, and ≠. The programmer may overload these function names, but the added definitions must be unary or binary to conform to the established syntax. Precedence rules for the unary and binary operators are given by the following table, in which operators on a single line have the same precedence and operators higher in the table bind more tightly than operators lower in the table. Unary operators have the highest precedence.

```
¬  -
↑
*  /
+  -
<  ≤  ≥  >  =  ≠
∧  cand
∨  cor
```

Within precedence levels, associativity is left-to-right.

For all operators except cand and cor, elaboration of an expression proceeds as if the expression were written in functional form (see section 3.1). For cand and cor, the left operand is elaborated first and the right operand is elaborated only if necessary.

A manifest expression is a literal, a value of an enumeration type, an identifier declared with manifest binding, a generic parameter, a manifest type attribute, a constructor involving only manifest expressions, or any expression involving only these expressions and language-defined operations. The value of a manifest expression is known during translation.

3.1. Invocations

Some examples are:

```
F(5)
Sequence'Insert(S,5)
P()
```

An invocation causes the elaboration of a procedure or function body with the elements of the <formals> list of the routine bound to the elements of the <actuals> list provided by the invocation. If a routine name is overloaded, the definition whose formal parameter types match the types of the actual parameters is selected. Procedure and function invocations (<proc call> and <func call>) differ in that procedure invocations are statements, whereas function invocations are expressions having values. An invocation consists of the following steps:

- Elaborate each of the <actuals> in an unspecified order, yielding a sequence of objects.
- For each result formal, create a variable having the same type and attributes as the corresponding actual. Bind the result formals to these variables.
- For each const or manifest formal, create an object of the specified type with the same attributes as the corresponding actual. Copy the value of the actual into the new object.[1]
- Bind each var formal to the corresponding actual, which must be a variable (i.e., a <var id>). Thus var formals are passed by reference.
- With the bindings established, elaborate the body of the routine.
- For each result formal, copy the final value of the variable bound to that formal back into the corresponding actual, which must be a variable (i.e., a <var id>). Note that this actual is determined before the elaboration of the routine (i.e., for the actual A(i), it is the initial and not the final value of i that determines the variable that receives the result).

The result of a function is treated as a result parameter instantiated at the call site with extent as described above and passed as an implicit parameter to the function. During the elaboration of the function, its value is developed in this result parameter.

During elaboration of a function, assignment to a variable that is not local to the function body (or to the body of a routine it invokes, directly or indirectly) is permitted only if the function is never invoked in a scope where such a change is made to a variable or component that is directly accessible by the caller.

Actual parameters are matched with formal parameters positionally. They must satisfy restrictions on type, binding and aliasing.

- The type of an actual parameter is acceptable if its <type name> exactly matches the <type name> of the corresponding formal parameter. Type attributes (instantiation parameters of a type) play no role in type checking. Chapter 5 gives rules for determining <type name>s.
- The binding of the actual parameter is acceptable if it matches the <binding> of the corresponding formal parameter according to the following rules:

If the formal parameter is	then the actual parameter may be
var	<var id> declared var
const	<expr>
manifest	any manifest <expr>
result	<var id>

- Finally, the set of actual parameters must satisfy the following nonaliasing restriction: A variable may not be used in more than one var or result position of a single procedure or

[1] Note that for dynamic types, this is a pointer copy.

process call. For the purpose of testing this restriction, imported variables are considered to be actual parameters bound as specified in the import list.

3.2. Dynamic Allocation

Each use of the constructor for a dynamic type creates a distinct element of the type. Each such element remains allocated as long as there is an access path to it.

Attributes of the dynamic type are provided when the constructor is used. Thus it is possible to associate objects with different attributes with the same dynamic variable at different times.

4. Statements

```
<stmt>        ::= <proc call> | <id> : <stmt> | <empty> | <block>
                | <var id> := <expr>
                | if <expr> then <stmt>;* { elif <expr> then <stmt>;* }* { else <stmt>;* }* fi
                | case <expr> { on <option> -> <stmt>;* }+ esac
                | while <expr> do <stmt>;* od | for <id> in <range> do <stmt>;* od
                | goto <id>
                | signal <qual id> | resignal | assert <expr>
                | <stmt> { { on <id>,+ -> <stmt>;* }+ }
                | create <var id> ( <actuals> )
<proc call>   ::= <qual id> ( <actuals> )
<block>       ::= <code body>
<code body>   ::= begin { <def-decl> ; }* <stmt>;* end
```

Statements designate actions to be performed. Their elaboration results in changes in the execution state of the program. The <empty> statement has no effect. Labels are used by **goto** statements in altering the flow of control in a program. A label is accessible only within the <stmt> it labels and within a compound statement (sequence of <stmt>s separated by semicolons) of which it is a <stmt>.

4.1. Blocks

Some examples are:

```
begin var x: boolean; x := true end
begin x := y; y := z; end
```

Blocks control extent. A <block> is elaborated when control flows into it, either because the <block> is the body of a routine that has been invoked or because the elaboration of another <stmt> has transferred control to it. First, all declarations and the texts of all module definitions are elaborated, in lexical order. Next, the <stmt>s are elaborated as described elsewhere in this chapter. Finally, the <block> is exited or terminated. If it is exited, control waits for all activations declared in this <block> to become dead or mint, then the extent defined by the <block> is closed and all nondynamic variables instantiated in the <block> are deallocated. If the <block> is terminated, all activations declared in the <block> are forcibly terminated, and then the <block> is exited. The choice between exiting and terminating the block depends on how control arrived at the end of the block. If the block came to an end because a handler completed or an enclosing process was terminated, the block is terminated. Otherwise, it is exited.

A <block> is not permitted to export identifiers. Except when used as a routine body, it is an open scope and has no need to import any.

4.2. Sequenced Statements

Some examples are:

```
x := 1; y := 2; z := 3
SumSq := 0; for i in 1..10 do SumSq := SumSq + V(i)↑2 od
```

Sequenced statements are elaborated in the order given, except when that order is interrupted by a **goto** or an exception.

4.3. Assignment Statement

Some examples are:

```
V(5).Sum := 0
x := (3 + u) * y
```

The assignment statement "V := E" is a procedure call on an appropriate assignment operator, defined

```
proc ":="(var LHS:T, const RHS:T)
```

for arbitrary type T. The value of the second parameter is assigned to the object named by the first parameter. The parameters are of the same type, and the normal type-checking rules apply.

Assignment operators are defined for all primitive types. Assignment operators are defined for arrays, records, variants, and programmer-defined types if and only if they have no components that are declared const or are nonassignable by virtue of this rule. An assignment operator that copies the whole value is automatically supplied for each user-defined type. For dynamic types this is a pointer copy. Although assignment may be invoked with any variable and value of the type, it requires that the attributes of its left and right operands be identical, and signals the BadAssign exception if they are not. The BadAssign exception is also signalled if an assignment involving mismatched array, string, or set sizes or an activation not in mint state is attempted.

4.4. Conditional Statements

Some examples are:

```
if A < 3 then x := y fi
if x ≠ 0 cand y/x > 0 then z := ut(y/x) else u := 1.0; q := y/x fi
case Tint
   on fuchsia -> Hue := cool; Description := "Purplish-red"
   on puce -> Hue := warm; Description := "Brownish-purple"
   esac
```

In the statement "if B then S1 else S2 fi", B must have type boolean. First, B is elaborated. If the resulting value is true, S1 is elaborated; otherwise S2 is elaborated. In the absence of an else clause, S2 is taken to be the empty statement, which has no effect.

The expression

```
if B1 then S1 elif B2 then S2 ... elif Bn then Sn else S* fi
```

is equivalent to

```
if B1 then S1 else
   if B2 then S2 else
      . . .
         if Bn then Sn else S* fi
      . . .
   fi
fi
```

In the statement

```
case E0
   on E11,....,E1k -> S1
   on E21,....,E2l -> S2
      . . .
   on En1,....,Enm -> Sn
   on others -> S*
   esac
```

The E's must all be expressions of the same type, and all except E0 must be manifest. The type of the E's must be fixed, an enumerated type, or a defined type that is ultimately defined in terms of fixed or an enumeration. Any of the E's except E0 may be a <range>; such an Eij is treated as the sequence of values in the range. First, E0 is elaborated. The Eij are elaborated and the results are compared to E0 (in unspecified order). If E0 is equal to some Eij, the corresponding Si is elaborated. If all comparisons yield false, S* is elaborated. Exactly one Si is elaborated for each correct elaboration of the case statement. If the special constant others does not appear as the last <option> and no match is found, an exception (CaseFailed) is signalled.

4.5. Loop Statements

Some examples are:

```
while x < 2.5 do x := F(y,x); y := G(y,x) od
for i in 1..10 do V(i) := i od
for hue in color do Tint(hue) := hue od
```

The loop while E do S od repeatedly elaborates if E then S fi until E becomes false. If E is initially false, the loop has no effect (other than the possible hidden effects or exceptions caused by the elaboration of E.)

The for statement for i in R do S od repeats the steps

- Bind i (as a constant) to a value in the range R.

- Elaborate S.

once for each element of the range R, in order. If R has no elements, the loop has no effect. The scope of the loop constant is restricted to the loop.

4.6. Unconditional Control Transfer

An example is:

```
goto L
```

The effect of a goto statement is to force control to the beginning of the statement with the given label. Since the scope rules prevent inheritance of labels across closed scope boundaries and importation of labels, a goto can not be used to transfer out of a routine or module.

4.7. Exceptions

Some examples are:

```
signal TooBig
assert x < 0

read(file,x) { on EOF -> goto Exit }
x := x+1 { on Overflow -> x := 0 }
```

Exceptions are processed by handler clauses associated with individual statements. Each handler clause associates processing code with given exceptions. The special identifier others may appear as the last <id> list of a handler clause; it matches any exception that is not named in some other exception <id> list of the same clause.

When an exception is signalled, control is transferred to the nearest dynamically enclosing handler clause that handles the exception, either explicitly or through an others clause; the elaboration of the handler replaces the elaboration of the remainder of the statement. If this handler is not in the currently-executing block, all intervening blocks will be terminated. If a handler is not found within the currently-executing routine, that routine is terminated and the exception is resignalled at the point of call of the routine. If a handler is not found within the currently-executing process, that process is terminated and the exception is resignalled at the end of the block in which the process activation was declared after waiting for control to reach that point and for all other activations declared in that block to terminate. If no handler is found in the scope of the exception name, a default handler will be supplied to terminate that block.

Exiting a handler causes termination of the <stmt> with which it is associated. If the handler resignals the same exception or raises a new one, the normal rules for exception processing apply.

The resignal command may be used in any handler body to resend the signal that caused that handler to be invoked.

The assert statement raises the assertion exception if the <expr> is false. It is exactly equivalent to the statement "if ~ <expr> then signal assertion fi".

There is one exception to the rule that an exception must be handled by the block in which it is signalled or by a caller of that block: the Notify operation on activations or actnames. The effect of a Notify is as if the Terminate exception were signalled in the currently-executing statement of the activation named by the Notify command.

4.8. Parallel Process Control

Some examples are:

```
create P(5)
activate(P1)
if IsBlocked(P1) then . . .
```

The create command instantiates a process, P, as an object of type activation-of-P. The <var id> in a create must name an object of type activation-of-P that is in mint state. If a process takes any var parameters, the corresponding actual parameters must have extent at least as great as the activation variable. The effect of the create is to instantiate an activation of P, bind the actuals of the create to the formals of P, and set the activation in suspended state.

Each activation has a unique identifying token value of type actname, and it may be named by one or more objects of type actname. Except for create, all operations that control parallelism are special routines that operate on either actnames or activations. These routines control the processes and parallelism by changing and interrogating the states of individual activations; they are described in Appendix I.2.

Note that the extent rules require an activation to be dead or mint before the block in which it is declared can be exited. This provides an implicit join operation. A fork can be obtained with a series of creates and activates.

5. Types

```
<type>        ::= fixed( <actuals> ) | float( <actuals> ) | boolean | latch | char | file( <actuals> )
                | enum[ <id>,+ ] | enum[ { " <char> " },+ ] | <expr> . . <expr>
                | set( <actuals> ) | string( <actuals> )
                | array ( <range>,+ ) of <type> | record [ <declaration>,+ ]
                | variant <declaration> [ { on <option> -> <type> }+]
                | dynamic <type> | activation of <qual id> | actname
                | <type name> [ ( <actuals> ) ]*
<type name>   ::= fixed | float | boolean | latch | char | file | set | string
                | enum[ <id>,+ ] | enum[ { " <char> " },+ ]
                | array [ <type name>,+ ] of <type name> | record [ [ <id>,+ : <type name> ],+ ]
                | variant [ <type name> { on <option> -> <type name> }* ]
                | dynamic <type name> | activation [ <qual id> ] | actname
                | <qual id> { [ <qual id>,+ ] }*
```

In Tartan, a <type name> may be either a simple identifier or an identifier inflected with additional type names. The <type name> so formed captures all the information needed for type checking.

- The <type name>s for the primitive scalar and simple nonscalar types are the keywords used to declare them: **fixed, float, boolean, latch, char, set, string, actname, file.**

- The <type name> for an array declared "array(a..b) of D" is "array[I,D]", where I is the <type name> of a and b.

- The <type name> for an enumeration declared enum[L1,L2,...Ln] is enum[L1,L2,...,Ln].

- The <type name> for an activation declared activation of P is activation[P].

- The <type name> for a dynamic type declared dynamic T is dynamic T.

- The <type name> for a record type is based on the sequence of field names and <type name>s in its declaration. For a record declared "record[F1:T1, F2:T2, ..., Fn:Tn]" the <type name> is "record[F1:TN1, F2:TN2, ..., Fn:TNn]", where the Fi are lists of field names, the Ti are types, and the TNi are type names. Bindings in the declaration do not appear in the type name.

- The <type name> for a variant is "variant[TT,T1->V1,T2->V2,...,Tn->Vn]", where TT is the <type name> of the tag, Ti is the i^{th} value of the tag type, and Vi is the <type name> that corresponds to the i^{th} value of the tag type. As a result, two variant <type>s are the same if they specify the same <type>s for all values of the tag.

- The <type name> for a defined type is the name given in the type definition.

5.1. Scalar Types

Some examples are:

```
Real
1..10
enum[fuchsia, ochre, puce, saffron]
```

Built-in scalar types include **fixed, float, boolean, latch,** and character. Integer and real must be constructed as special cases of **fixed** and **float**. Ordered scalar enumerated types are defined by providing an ordered list of values.

Types **fixed** and **float** require <actuals> lists to provide range, scale, and precision when they are used in declarations. These are attributes and do not affect the type. Although bindings for attributes may in general be **const** or **manifest**, the specifications of **fixed** and **float** require **manifest** attributes.

To define a type, the <expr>s in an explicit range must be **const** or **manifest**.

5.2. Composite Structures

Some examples are:

```
array(1..10) of Color
array(Color) of Real
string(10)
record(Name:string(35), Age:Int)
variant b:boolean (on true -> Int on false -> char)
```

Nonscalar data structures may be built up in three ways: with arrays (homogeneous indexed linear structure), with records (nonhomogeneous structures with named fields), and with variants (structures whose substructure may vary with time). In addition, the nonscalar types set, string, and file are defined.

Legal bindings for fields of records and variants are var, const, and manifest. If a <binding> is empty, it is taken to be var.

A variant type must have exactly one tag field. The special constant others may appear as the last <option> of a <variant type>; it matches any constant that does not appear in any other <option>.

The syntax for arrays provides an abbreviation for a set of types pre-defined as "array[IxType,EltType](r)" where IxType is the index type, EltType is the element type, and r is a (sub)range of IxType. Thus "array(1..10) of float" means "array[int,float](1..10)". Its type name, "array[int,float]", is written "array[int] of float". As for any type, when an <array type> is used as a formal parameter, the attributes are not supplied. The type "array(A,B) of T" is an abbreviation for "array(A) of array(B) of T". Similarly, the array accessor "V(i,j)" is an abbreviation for "V(i)(j)".

5.3. Dynamic Types

Some examples are:

```
dynamic Real
dynamic record (Data: Int, Next: ListElt, const Index: Int := K)
```

Values of a dynamic type are pointers to variables whose structure corresponds to the type definition. They are initialized to nil. The extent of these variables covers the entire scope of the type definition. Elaborating a constructor for the dynamic type yields a pointer to a new variable distinct from all others. The constructor supplies the attributes for this variable; they are not supplied in the declaration of the named variable of the dynamic type. Thus a named variable of dynamic type may at different times point to several different variables having different attributes.

5.4. Process Control Types

Some examples are:

```
activation of P
actname
```

Parallel processes are controlled with data of two types -- activations of processes and actnames, or names of activations. Activations are instantiations of a given process; an activation may contain at most one process activation during its lifetime and then only of the process given in its <type>. An actname value is a pointer to an activation. Actname variables may contain pointers to activations of any processes; an actname variable may refer to different instantiations of different processes from time to time.

An activation is used to control parallel or pseudo-parallel execution of a process. At any time it may be in one of four states: mint, active, suspended, and dead. The extent of an activation variable coincides with its scope. The immediately enclosing block cannot be exited until all activations declared within it are dead or mint. An activation is associated with exactly one process, which must be named by the <qual id>.

An actname may refer to any instantiated process. A newly-declared actname or activation variable is initialized to mint.

5.5. Defined Types

Some examples are:

```
T(n)
Sequence[int](50)
```

Programmers may define new types. See section 6.5 on Type Definitions.

6. Definitions and Declarations

```
<def-decl>      ::= <declaration> | <mod def> | <routine def> | <type def> | <generic def> | <empty>
                  | imports <qual id>+ | exports <qual id>,+ | exception <id>,+ | disable <id>,+
                  | prag <proc call>;+ ;* garp
<declaration>   ::= <binding> { <id>,* { : <type> }* [ := <expr> }* },+ | <binding> { <id>,+ : <type name> },+
<mod def>       ::= module <id> <mod text>
<mod text>      ::= ; <code body> | <remote inst>
<routine def>   ::= proc <id> <proc text> | func <id> <func text> | process <id> <proc text>
                  | func " { <unop> | <binop> } " <func text>
<func text>     ::= ( <formals> ) <id> : <type> ; <block> | <remote inst>
<proc text>     ::= ( <formals> ); <block> | <remote inst>
<type def>      ::= type <type name> { ( <formals> ) }* = <type>
<generic def>   ::= generic module <id> [ <formals> ] <mod text> | generic func <id> [ <formals> ] <func text>
                  | generic proc <id> [ <formals> ] <proc text> | generic process <id> [ <formals> ] <proc text>
<remote inst>   ::= is <qual id> [ <actuals> ] | is assumed ( <id> )
<formals>       ::= { <binding> <id>,* : <type name> },*
<binding>       ::= <empty> | var | const | manifest | result
```

6.1. Declarations

Some examples are:

```
var     x: Real
const   y:= true
var     Hue1, Hue2, Hue3: Color
var     Tint := enum[saffron, puce, fuchsia, ochre]
var     V: array(5..7) of Int
var     M1:Mark(5), M2:Mark(7)
manifest   PI: Real := 3.14
```

The syntax for declarations allows three kinds of abbreviations. If the initialization expression appears, the type of the variable is evident from the <expr> and the ":<type>" may be omitted. In addition, lists of <id>s with the same types or bindings may be condensed. These abbreviations are illustrated by the following five declarations, all of which have the same effect:

```
var x,y := 0
var x,y:Int := 0
var x := 0, y := 0
var x:Int := 0, y:Int := 0
var x:Int := 0; var y:Int := 0
```

Elaboration of a declaration causes instantiation of an object which is the variable. Each variable has a type and a value. The type is determined when it is instantiated, but the value may be changed by further elaboration of the program. A variable may be restricted to be const (value fixed at block entry) or manifest (value fixed during translation).

Elaboration of a declaration proceeds as follows:

- Evaluate the <expr>, if present. It must be present in manifest or const declarations. It must be manifest in manifest declarations.

- If the <binding> is manifest, bind the value of the <expr> to the identifier(s).

- If the <binding> is const or var, elaborate any <actual>s in the <type> and instantiate a new variable with the indicated type and attributes for each identifier. If there was an <expr>, assign its value to each of the new variables.

When the type is dynamic, the declaration supplies the <type name> only (no attributes). In this case, only the pointer is allocated at block entry; the attributes are supplied when the dynamic type is actually (dynamically) allocated.

6.2. Modules

An example is:

```
module CounterDef;
    begin
    exports Counter, Reset, Incr, Value;
    type Counter = Int;
    proc Reset(result C:Counter); begin C := 0 end;
    proc Incr(var C:Counter); begin C := C + 1 end;
    func Value(const C:Counter)x:Counter; begin x := C end
    end
```

The elaboration of a module takes place during the elaboration of declarations for the block in which the module is defined. This elaboration consists of elaborating the declarations of the module in lexical order, then elaborating the statements of the module.

A module or routine inherits identifiers for definitions (modules, routines, exceptions, and types) from its enclosing scope. It may explicitly **import** identifiers of objects from that scope, provided the objects have global extent. A module, but not a routine, may **export** definition and object identifiers to its enclosing scope. Types, named routines, field accessors for records, and variables are exported by including their names in the **exports** list of the module. The right to apply infix operators, constructors, subscripts, ".all", or the **create** command for a type T are exported by including the special names T'infix, T'constr, T'subscr, T'all, and T'create, respectively, in the exports list. Literals of enumerated types are exported automatically if the types are exported.

6.3. Routines

Some examples are:

```
proc F(var x:Int); begin x := - x; end
proc G is GenG(5)
func IsNil(x:DynT)y:boolean; begin y := (x = nil) end
func "+"(a,b:gorp)c:gorp;
    begin
    imports Bias;
    c := gorp'(a.left+b.left+Bias, a.right+b.right+Bias)
    end
```

A routine is a closed scope whose body is a block. Thus its body controls extent for local declarations, but does not inherit identifiers for (non-manifest) objects or labels. The <formals> list declares the identifiers for parameters.

A routine may be a function (**func**), which returns a value and has no visible side effects; it may be a procedure (**proc**), which can modify its parameters but must be called as a statement; or it may be a **process**, which is a potentially-parallel procedure. Special type-specific routines are described in Appendix I.2.

Routine names may be overloaded by binding the same identifier to several definitions with different numbers or types of parameters. The functions for which special infix notation is provided are obvious candidates for overloading.

If a <binding> in a routine header is omitted, it is assumed to be **const**. The result binding may be used only in procedures. No duplication of identifiers within the <formals> list is permitted, and parameter names may not conflict with declarations or imports in the routine body.

6.4. Exceptions

Some examples are:

```
exception TooBig, TooSmall, Late, Singular
disable TooBig, TooSmall
```

The scope of an exception name is the block in which it is declared. A **disable** declaration in an inner block suppresses detection of the exceptions it names. A handler clause associates recovery code with a statement that may generate an exception (see section 4.7).

The **disable** declaration permits exceptions to be individually suppressed within a given scope. Should an exception occur when its detection is suppressed, the consequences are not defined. An exception must not be signalled or redeclared in a scope in which it is suppressed. Note that suppression of an exception is not an assertion that the condition that gives rise to the exception will not occur.

Standard exceptions will be declared in the global extent.

6.5. Type Definitions

Some examples are:

```
type Counter = Int
type Matrix(n:Int) = array(1..n,1..n) of Real
```

A user may introduce a new type into his program with a type definition. The type definition itself merely introduces the <type name> and defines the representation of the type. Operations are introduced by writing routines whose formal parameters are of the newly-defined type. Scope boundaries, particularly module boundaries, play no role in the definition of the type. There is, as a consequence, no notion of the complete set of operations on a type.

A type definition may be parameterized. The bindings in the formal parameter list must be const or manifest. If a <binding> is omitted, it will be assumed to be const. The names of the formal parameters of the type are available throughout the elaboration of the program as constants, called attributes. They are accessed by treating the <var ident> as a record and the type attribute as a field. Attributes for primitive types are given as part of the type definitions.

Within the scope in which the type is defined, the qualifier Rep may be used to indicate that the object named by the identifier Rep qualifies is to be treated as if it had the underlying type. This allows operations on the new type to be written using operations on its representation. When no ambiguity arises, the Rep qualification may be omitted.

6.6. Generic Definitions

Some examples are:

```
generic proc Reset[T:type](var x:T); begin x := x'min end
proc ResetColor is Reset[Color]
proc ResetX is Reset[Sample]
module Stack is assumed(StackDef)

generic module RingDef[K: Int];
    begin
    exports Ring, Next;
    type Ring = fixed(1,0,0,K-1);
    func Next(R:Ring)N:fixed(1,0,0,K-1); begin N := mod(R+1,K); end
    end
module RS is RingDef[5]
module R9 is RingDef[9]
```

A generic definition is syntactically like the corresponding specific definition except that it is prefixed by the word **generic** and it may have a set of generic parameters (enclosed in square brackets) after the definition name. For generic definitions, type is acceptable as a formal <type name>.

The actual parameters supplied in an instantiation of a generic definition may be any defined identifiers, including those for variables, functions, types, or modules, or any expression. When the generic definition is instantiated, the text of the actual parameters replaces the identifiers that represent the formal parameters. The substitution is done on a lexical, rather than a strictly textual, basis. That is, the identifiers in the generic definition are renamed as necessary to avoid conflicts with the identifiers in the actual parameters.

Both generic definitions and remotely-defined modules or routines may be incorporated in a program as **remote instances**. A remote instance may be an instantiation of a generic definition or a reference to a definition given elsewhere.

A module or routine that is used by the program but whose definition is given elsewhere (e.g., in a library) is incorporated by writing **is assumed**(<id>) as the body of a module or routine definition. The <id> is used by a pragmat to locate the remote definition.

A generic definition is instantiated by referring to it as the body of a module or routine definition. The effect of the instantiation is as if the generic definition were lexically substituted in place of the reference to it. That is, the body of the module or routine being defined becomes a copy of the generic definition.

An instantiation of a generic definition may be used as the body of a specific module or routine. The usual restrictions on defining new identifiers apply to the module or routine being defined in terms of a generic.

Generic type definitions arise from generic modules. They are instantiated when the module is instantiated. Thereafter, they may be used in declarations or definitions.

If the generic definition has generic parameters, the actual parameters supplied with the instantiation must have correponding types and be syntactically suitable for substitution.

If a generic definition is instantiated more than once in a scope, ambiguous names may be introduced. The usual rules for resolving such ambiguities apply.

6.7. Translation Issues

An example is:

```
prag Optimize(space); Listing(Off) garp
```

A program is a <block>. The extent defined by the outer block of the program is the global extent.

The translator may be guided by <pragmat>s. Pragmats have the same syntax as procedure calls. The set of pragmat names and the interpretations of the arguments are determined by each translator. Translators will ignore pragmats whose names they do not recognize.

A program may be broken into separately defined segments. This decomposition must take place in the global extent. The units of separate definition are modules and routines. The definition

```
module Q is assumed(I)
```

in a segment has the effect of making the semantics of the segment the same as if the (separately defined) text of Q had been substituted for "**is assumed**(I)". The identifier I refers to a file, library, or other facility for storing separately defined segments. The relation between the identifier I and that storage facility may be established by a pragmat.

It is a matter of optimization whether the separate definition is included as text or separately translated and linked in. In order to perform independent translation of a separately defined component, it is necessary to embed the module or routine being translated in an environment that supplies definitions for all the names it inherits or imports. This environment must form a complete program. It is assumed that the translation system provides commands for selecting which components of such a translation to save and for determining where and in what form they are to be saved.

I. Standard Definitions

I.1. System-Dependent Characteristics

The translator for each system is assumed to provide a module in the global extent that defines appropriate system constants. Such constants are assumed at various points in the language definition; these and certain others are summarized here in the form of a skeleton module.

```
module Sys;
begin
exports . . .                ! exports all definitions below

type Int = fixed(. . .)      ! appropriate to the machine
                             ! Note Int.Min and Int.Max give range

type Real = float(. . .)     ! appropriate to the machine
                             ! Attributes give range, precision, scale

const . . .                  ! constants that descibe properties of the
                             ! object machine

proc . . .                   ! procedures for accessing facilities of the
                             ! operating and file systems

exceptions . . .             ! System-defined exceptions such as Assertion, BadAssign....

end
```

I.2. Properties of Types

All types have assignment operators and routines for conversion to appropriate other types. In particular, the scalar types have routines for converting to and from character strings. All nonscalar types have constructors. The sections below sketch some important properties of the built-in types.

I.2.1. Fixed

Literals:	digit strings
Attributes:	Min, Max, Precision, Scale
Infix operations:	Arithmetic and relational
Special routines:	rounding, truncation

I.2.2. Float

Literals:	digit strings with decimal point
Attributes:	Min, Max, Radix, Precison, MinExp, MaxExp
Infix operations:	Arithmetic and relational
Special routines:	rounding, truncation

I.2.3. Enumerations

All enumerations are ordered. The literals are assumed to appear in the declaration in increasing order.

Literals:	As given in definition
Attributes:	Min, Max
Infix operations:	Relational
Special routines:	succ, pred

I.2.4. Boolean

Literals:	true, false
Attributes:	none
Infix operations:	logical
Special routines:	none

I.2.5. Characters

Literals:	Quoted characters
Attributes:	Min, Max
Infix operations:	none
Special routines:	as for enumerations

I.2.6. Latches

A latch is a simple spinlock for mutual exclusion. If the latch is open, it is available for siezure; if it is closed, a Lock command will wait on it.

Literals:	open, closed
Attributes:	none
Infix operations:	none
Special routines:	Lock, IfLock, Unlock

I.2.7. Arrays

Literals:	none
Attributes:	Range, EltType
Infix operations:	none
Special operations:	subscript, subarray, catenation, upper bound, lower bound

I.2.8. Sets

"Sets" are boolean vectors on which some additional operations are defined.

Literals:	empty
Attributes:	EltType, MaxSize
Infix operations:	logical
Special operations:	subscript

I.2.9. Dynamic Types

Literals:	nil
Attributes:	The named variable does not itself have attributes, but the dynamic variable that it references may.
Infix operations:	none
Special operations:	.all denotes whole value of dynamic object, as distinguished from the reference. A dynamic constructor allocates a new dynamic object.
Special routines:	none

I.2.10. Records

Literals:	none
Attributes:	individually defined with record type
Infix operations:	none
Special operations:	field selection, constructors
Special routines:	none

I.2.11. Variants

Literals:	none
Attributes:	individually defined with variant type
Infix operations:	none
Special operations:	variant must be designated to reference contents
Special routines:	none

I.2.12. Strings

Literals:	Quoted strings
Attributes:	Length
Infix operations:	none
Special operations:	subscript, substring, catenation

I.2.13. Activations

Literals:	mint
Attributes:	none
Infix operations:	none
Special operations:	create
Special routines:	To change state: Activate(A), Suspend(A), UnlockAndSuspend(A,L), UnlockAndActivate(A,L), LockAndSuspend(A,L), LockAndActivate(A,L), Terminate(A) To query state: IsMint(A), IsAct(A), IsSusp(A), IsTerm(A) To obtain actname: NameOf(A), Me() To sent exception: Notify(A) Other: Priority(A), SetPriority(A), Time(A)

where A is an activation or actname and L is a latch

Assignment causes the BadAssign exception if either the value or the variable to which it is being assigned is in a state other than mint.

I.2.14. Actnames

Literals:	mint
Attributes:	none
Infix operations:	none
Special operations:	none
Special routines:	Same as for activations

I.2.15. Files

A minimal input-output facility will be provided.

I.3. Alphabets

The following context-free substitutions reduce the alphabet used in this report to the standard 64-character ASCII subset. Note that some identifiers are pre-empted as a result.

For the publication character:	Substitute the ASCII string:
lower case a..z	upper case A..Z
≤	<=
≥	>=
≠	<>
∧	and
∨	or
{	<<
}	>>

II. Collected Syntax

```
<const>        ::= <digit>+ { . <digit>+ }* | true | false | nil | closed | open | mint | empty
                 | <constructor> | <id> | <qual id> ' <const> | <type> ' <const> | <expr>
<constructor>  ::= ( <expr>,* ) | ( { <option> -> <expr> },+ ) | " <char>* "
<var id>       ::= <qual id> | <var id> ( <actuals> ) | <var id> . <id> | <var id> ( <range> ) | Rep' <id>
<range>        ::= <expr> .. <expr> | <type>
<option>       ::= { <const> | <range> },+
<qual id>      ::= { <id> '}* <id>
<id>           ::= <letter> <letter or _ or digit>*
<expr>         ::= <unop>* <var id> | <unop>* <const> | <unop>* <func call>
                 | <unop>* ( <expr> ) | ( <expr> ) . <id> | <expr> <binop> <expr>
<unop>         ::= - | ¬
<binop>        ::= * | / | + | - | < | ≤ | > | ≥ | = | ≠ | ∧ | cand | ∨ | cor | ↑
<func call>    ::= <qual id> ( <actuals> )
<actuals>      ::= <expr>,*
<stmt>         ::= <proc call> | <id> : <stmt> | <empty> | <block>
                 | <var id> := <expr>
                 | if <expr> then <stmt>;* { elif <expr> then <stmt>;* }* { else <stmt>;* }* fi
                 | case <expr> { on <option> -> <stmt>;* }+ esac
                 | while <expr> do <stmt>;* od | for <id> in <range> do <stmt>;* od
                 | goto <id>
                 | signal <qual id> | resignal | assert <expr>
                 | <stmt> { { on <id>,+ -> <stmt>;* }+ }
                 | create <var id> ( <actuals> )
<proc call>    ::= <qual id> ( <actuals> )
<block>        ::= <code body>
<code body>    ::= begin { <def-decl> ; }* <stmt>;* end
<type>         ::= fixed( <actuals> ) | float( <actuals> ) | boolean | latch | char | file( <actuals> )
                 | enum( <id>,+ ) | enum( { " <char> " },+ ) | <expr> .. <expr>
                 | set( <actuals> ) | string( <actuals> )
                 | array ( <range>,+ ) of <type> | record [ <declaration>,+ ]
                 | variant <declaration> [ { on <option> -> <type> }+]
                 | dynamic <type> | activation of <qual id> | actname
                 | <type name> { ( <actuals> ) }*
<type name>    ::= fixed | float | boolean | latch | char | file | set | string
                 | enum( <id>,+ ) | enum( { " <char> " },+ )
                 | array [ <type name>,+ ] of <type name> | record [ { <id>,+ : <type name> },+ ]
                 | variant [ <type name> { on <option> -> <type name> }* ]
                 | dynamic <type name> | activation [ <qual id> ] | actname
                 | <qual id> { [ <qual id>,+ ] }*
<def-decl>     ::= <declaration> | <mod def> | <routine def> | <type def> | <generic def> | <empty>
                 | imports <qual id>,+ | exports <qual id>,+ | exception <id>,+ | disable <id>,+
                 | prag <proc call>;+ ;* garp
<declaration>  ::= <binding> { <id>,+ [ : <type> ]* { := <expr> ]* },+ | <binding> { <id>,+ : <type name> },+
<mod def>      ::= module <id> <mod text>
<mod text>     ::= ; <code body> | <remote inst>
<routine def>  ::= proc <id> <proc text> | func <id> <func text> | process <id> <proc text>
                 | func " { <unop> | <binop> } " <func text>
<func text>    ::= ( <formals> ) <id> : <type> ; <block> | <remote inst>
<proc text>    ::= ( <formals> ); <block> | <remote inst>
<type def>     ::= type <type name> { ( <formals> ) }* = <type>
<generic def>  ::= generic module <id> [ <formals> ] <mod text> | generic func <id> [ <formals> ] <func text>
                 | generic proc <id> [ <formals> ] <proc text> | generic process <id> [ <formals> ] <proc text>
<remote inst>  ::= is <qual id> [ <actuals> ] | is assumed ( <id> )
<formals>      ::= { <binding> <id>,+ : <type name> },*
<binding>      ::= <empty> | var | const | manifest | result
```