The Darwin Language
Version 3d

Department of Computing, Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.

Last Revised: Monday, 15 September 1997

Contents

Preliminaries

Syntax

The syntax of Darwin is given in a variant of traditional BNF:-

- Non-terminal identifiers are shown without angles brackets. UPPERCASE letters are not significant in non-terminal identifiers, but serve as commentary.
- Terminal identifiers are shown in bold e.g. **inst**.
- Syntactic comments follow a // and continue to the end of the line.
- The following conventions are used for repetitions:

```
[1   ... ]   one-of
[01  ... ]   zero or one occurrences of
[1+  ... ]   one or more occurrences of
[0+  ... ]   zero or more occurrences of
```

Lexical Conventions

Darwin follows the lexical conventions of IDL for tokens, comments and identifiers. Some additional remarks follow:

Keywords

The following identifiers are reserved for use as keywords:

```
keyword =
   assert | bind | component | dyn | export | forall | import |
   inst | interface | portal | provide | require | spec | to | when |

   int | double | string | boolean | true | false
```

Identifiers

In Darwin identifiers are used for defining and naming component types, interface types, parameters, constants, portals, instances, forall iterators etc. Identifiers must be unique within the scope in which they are defined:

- the scope of a forall identifier extends from its defining point to the end of its forall declaration block.
- the scope of other identifers extends from their defining point to the end of the immediately enclosing declaration block (i.e. enclosing { } pair).

```
id = identifier
```

No identifier may have the same spelling as a Darwin keyword. The case of letters is significant within an identifier.

Expressions

Literals and expressions are the same as those found in IDL.

```
expression = IDL-const_exp    // + function calls
```

Predefined Types & Constants

Darwin has a number of predefined types (**int**, **double**, **string** and **boolean**) and constants (**true**, **false**) whose meaning is taken from similarly named types in IDL.

Function Calls

Darwin extends IDL expressions to support function calls. The types of parameters and function calls is currently inferred.

Darwin Declarations

A Darwin declaration is a collection of component declarations, interface declarations, constant declarations and external declarations.

Syntax

```
darwin-declaration =
   [1+ component-declaration  | interface-declaration  | const-declaration  |
         external-declaration  ]
```

Component Declarations

Component declarations define a component type from which one or more component instances can be created.   Component types can either be defined explicitly (see `component-block` below) or can be fully or partially typed from an existing component type (`see partial-component-declaration` below). The component type identifier is used to name the component type within instance declarations, partial component type declarations and parameter lists that require a component type parameter.

Examples

```
component alphaType (int A, string B) {}

component betaType (string x, <T>)
      @ family (TV_SET)
      @ draw (circle,x,y)
{
      portal ...
      inst ...
      bind ...
}

component betaType2 = betaType ("hello", alphaType);
      // note betaType2 is a component type, not a component instance.
```

Syntax

```
component-declaration =
   component COMPONENT-id
         [1 component-block | partial-component-declaration ]

component-block =
   [01 formal-parameter-list ]
         [0+ tag ]
   "{"
         [0+ declaration ";"]
   "}"

declaration =
   assert-declaration |
   bind-declaration |
   component-declaration |
   const-declaration |
   external-declaration |
   forall-declaration |
   inst-declaration |
   interface-declaration |
   portal-declaration |
   when-declaration

partial-component-declartion =
   "=" BASE-COMPONENT-type
         [01 BASE-COMPONENT-TYPE-partial-argument-list ]
         [0+ tag ]
   ";"

argument-list =
   "(" expression [0+ "," expression ] ")"

partial-argument-list =
   "(" [01 expression] [0+ "," [01 expression ] ] ")"
         // need to allow <T> syntax for some argument-lists.
```


Partial Component Declarations

Component types can be used to fully or partially evaluate other component types. Partial component types can be defined by omitting one or more actual parameters. Actual parameters that are supplied must correspond in order and type to the formal parameters of the component type. Partial component types are considered sub-types of the base component type that they were defined from.

Example

```
component beta1 = betaType ( "abc" );          // 2nd parameter is omitted
component beta2 = betaType ( , alphaType);     // 1st parameter is omitted
```

Generic Types

Generic types act as placeholders for predefined types, component types, or interface types and allow the specification of generic component types and generic interface types.   Generic type identifiers are always enclosed within "<" and ">".

Generic type names occur:

- as formal and actual parameters for component types and interface types,
- as portal types in portal declarations and portal member declarations. Note: portal declarations need not specify a portal type at all. In such cases an implicit and unique generic type is assumed.
- as component types in instance declarations.

Examples

```
component A (int y, <T>) { ... }
interface B (int y, <T1>, <T2>) { ... }
portal p : <T>;                    // <T> can be omitted here
inst X : <T1> (12, <T2>);
```

Syntax

```
type-name =
   id |
   "<" TYPE-id ">"                   // generic-type
```

Namespaces

Constants, interface types and nested component types can be accessed within other component types by prefixing the defining component type and the name resolution operator ("."), e.g. ComponentType.ConstantID.

Formal Parameters

Both component types and interface types can be parameterised. Such parameters can either be value parameters or type parameters.

```
formal-parameter-list =
   "("
         formal-parameter [0+ "," formal-parameter ]
   ")"

formal-parameter =
   value-parameter | type-parameter
```

```
value-parameter
   PREDEFINED-TYPE-id
         PARAMETER-id

type-parameter
   "<" GENERIC-TYPE-id ">"
```

Value Parameters

Value parameters take one of the predefined types (e.g. short, double, char, string) and can be used within expressions, for example:

- to set the upper bound for a portal or instance array,
- within *when* declarations to describe variant configurations or for evaluating the base case for recursive configurations,
- within *forall* declarations to iterate over a range of values,
- within *inst*ance declarations to index instances,
- within *bind* declarations to index instances and portals,
- within *tags* to define tag arguments.

Type Parameters

Type parameters allow both generic component types and generic interface types to be defined in Darwin.

Portal Declarations

Portal declarations define a set of component portals that can be bound internally to the portals of encapsulated sub-components or externally to the portals of peer components.

Portal declarations consist of an optional direction (e.g. **provide** or **require** or **import** or **export**), an optional type (e.g. fileIO) and a mandatory name (e.g. F):

```
provide portal F : fileIO;
require portal D : deviceIO;
```

Portal types can be

- composite interface types defined in Darwin (see below) or
- generic portal parameter types or
- omitted, in which case, Darwin will generate an implicit generic type name for the portal type.

Examples

```
portal P;
provide portal S          @ port(int);
require f : FileIO (2);
require portal S : <T>    @ tcp(192,12,43,43) @ entry(int,double);
export  Y                 // implicit generic type assumed
```


The Darwin Language Version 3

Syntax

```
portal-declaration  =
   [01 provide | require | import | export] [01 portal]
   PORTAL-id
         [0+ array-subscript]
         [01 interface-call ]
   [0+ tag ]

interface-call =
   ":" INTERFACE-type-name  [01 INTERFACE-TYPE-argument-list ]

array-subscript  = "[" INT-expression "]"
```

Portal Arrays

Portal arrays can be declared by suffixing the portal name with one or more integer expressions that specify the number of elements for each dimension of the array. Each bound must be >= 1. The lower bound of a portal array is always zero.

Portal Directions

Four portal directions are available:

- **Provide** declarations declare portals that are being provided by the defining component to other encapsulating components.
- **Require** declarations declare portals that are being provided by other encapsulating or external components to the defining component.
- **Export** declarations declare portals that are being provided by the defining component to an external nameserver/trader. Exported portals are similar to provided portals. One or more tags are typically used to register exported portals into a nameserver. Exported portals can be bound via import declarations.
- **Import** declarations declare portals that are being provided to the defining component by an external nameserver/trader. Imported portals are similar to required portals. One or more tags are typically used to locate and bind to portals registered with a nameserver or trader.

Interface Declarations

Interface declarations allow portals to be grouped together to form a composite portal type. Such interface types can be used in portal declarations to declare a composite portal with several nested portal members. Interface types can be parameterised and derived (by inheritance) from one or more base interface types. The portal type for an interface member can be omitted, in which case Darwin will generate an implicit generic portal type for the interface member.

Examples

```
interface ABCD {A; B; C; D;}

interface fileIO (int q)  @ qos(q)
{
      u : unix_fileIO     @ proto (nfs, tcp);
      m : mac_fileIO      @ proto (appleshare, localtalk);
}
```

```
interface ExtFileIO (int q) : fileIO (q) {
      p : PPP;
}

interface NestedFileIO (int q) {
      myFileIO : ExtFileIO (q);
}

interface GenericFileIO (<FileIO>, int q) {
      myFileIO : <FileIO> (q);
}

interface deviceio (int vector1, int vector2, <T>)  {
      k : keyboard @ interrupt (vector1);
      m : <T>      @ interrupt (vector2) @ assert (vector2#vector1);
      error;       // implicit generic type is assumed
}
```

Interface members are selected by suffixing the name of the portal with a dot and then the name of the portal member.

If an interface type has parameters, corresponding actual parameters (arguments) must be specified. Actual parameters must correspond in order and type to the formal parameters of the portal class.

Syntax

```
interface-declaration  =
   [1 interface] INTERFACE-TYPE-id
         [01 formal-parameter-list ]
         [0+ tag ]
         [0+ BASE-INTERFACE-interface-call ]
   "{"
         [1+ member-declaration ";"]
   "}"

member-declaration =
   MEMBER-id
         [0+ array-subscript]
         [01 interface-call ]
   [0+ tag ]
```

Instance Declarations

Instance declarations are used to create component instances from component types.

Examples

```
inst f : fileman ("/nd")      @ loc(64);
inst filter : <T>;
```

Syntax

```
instance-declaration =
   inst
         INSTANCE-id
                [0+ array-subscript ]                        // array
   elements
         ":" COMPONENT-type-name  [01 COMPONENT-TYPE-argument-list ]
         [0+ tag ]
```

The portals of a instance are selected by prefixing the name of the component instance and a dot character.

If the component type has parameters, corresponding actual parameters (arguments) must be specified. Actual parameters must correspond in order and type to the formal parameters of the component type.

Instance Arrays

Array instance elements are declared by suffixing the name of the instance array with one or more subscripts that define the specific instance array element. An instance array element may only be instanstiated once, and each element of an instance array must have the same number of subscripts. Unlike traditional arrays, Darwin instance arrays need not be explicitly bounded and can be sparse.

Examples

```
inst t [63] : transputer  @ arrange (circle);
inst t [-31] : transputer  @ arrange (circle);
```

Binding Declarations

Within a Darwin program, binding declarations are used to establish potential interactions between instances. Import declarations can be used to bind portals across one or more Darwin programs and also to bind portals to non-Darwin programs.

Examples

```
bind t.requirement -- s.provision@ qos (95.5);
bind x[2].tvsignals -- dyn window@ channels (1,10);
```

Syntax

```
bind-declaration =
   bind
         endpoint  "--"  endpoint
         [0+ tag ]

end-point =
   portal-name | dyn COMPONENT-type-name

portal-name =
   INSTANCE-OR-PORTAL-OR-MEMBER-id
         [0+ array-subscript]                    // instance or portal arrays
   [0+ "." portal-name ]                         // nested members
```

Dynamic Instances and Binding

Bindings can also be made to **dyn** components. Such bindings cause a new anonymous instance of the component to be instantiated each time the component is "invoked" by a bound portal. Parameters to the newly created instance are supplied from the invoking portal. The method for dynamic component invocation is implementation-dependent.

Dyn Portal Types

Portals bound to a **dyn** component have a special Darwin-generated and implementation-dependent **dyn** portal type that defines the parameters of the dyn component type.

Directionality Constraints

In Darwin, only the directions in the table below are allowed for bindings. Note that required, provided, exported and imported are optional in the table.

Binding	Binding Form	Picture
Peer	**bind** instance.requiredPortal -- instance.providedPortal	[]o -- •[]
Outward	**bind** instance. requiredPortal -- requiredPortal	[]o -- o]
Import	**bind** instance. requiredPortal -- importedPortal	[]o -- o
Inward	**bind** providedPortal -- instance. providedPortal	[• -- •[]
Export	**bind** exportedPortal -- instance. providedPortal	• -- •[]
Switch	**bind** providedPortal -- requiredPortal	[• -- o]
Dyn Component	**bind** instance. requiredPortal -- **dyn** componentType **bind** providedPortal -- **dyn** componentType	[]o -- [*] [• -- [*]
Dyn Instance	**bind dyn** componentType. requiredPortal - instance. providedPortal **bind dyn** componentType. requiredPortal - requiredPortal	[*]o -- •[] [*]o -- o]

Connectivity Constraints

The limits to the number of bindings that can be made to or from different categories of portal are summarised in the following table:

No. of bindings Allowed	Applicable Portals	Picture
0 or 1 (LHS)	*instance.requiredPortal* *providedPortal* *exportedPortal* **dyn** *componentType. requiredPortal*	[]o [• • [*]o
0 or more (RHS)	*instance. providedPortal* *requiredPortal* *importedPortal* **dyn** *componentType*	•[] o] o [*]

Typing Constraints

Two portals can only be bound for the following cases:

LHS Portal type	RHS Portal Type	Allowed
T1	**T2**	if T1 is identical to T2 or T1 is a base type of T2
T1 [ ]	**T2** [ ]	if T1 is identical to T2 or T1 is a base type of T2 and the size of T1 is equal to the size of T2.

Two portal types T1 and T2 are identical if T1 and T2 both have the same type, and for interface types if the members of the interface type are in the same order, have the same types and for interface member arrays the arrays have identical subscripts.

If T1 and T2 are array types then the size of T1 must be equal to the size of T2.

A portal type T1 is a base type of portal type T2 if T2 inherits from T1 and corresponding portal

member arrays in T1 and T2 have identical subscripts

Type Inference Rules for Binding

...

When Declarations

When declarations are used to conditionally elaborate a component type.

Example

```
when row != n-1 {
  bind  processor [ row ].out -- processor [(row+1)%n].in;
}
```

Syntax

```
when-declaration =
   when BOOLEAN-expression
        [0+ tag ]
        "{"
             [0+ declaration “;”]
        "}"
```

Implementations may choose to limit the allowable set of declarations that can placed within a when declaration block.

Forall Declarations

Forall declarations are used to iteratively elaborate a component type.

Example

```
forall row = 0 to n-1 {
       forall col = 0 to m-1 {
              inst t800 processor [row] ;
              bind processor[ row ].out -- processor [(col+1)%m].in;
       }
}
```

Syntax

```
forall-declaration =
   forall FORALL-id “=” INTEGER-expression to INTEGER-expression
              [0+ tag ]
        "{"
              [0+ declaration “;”]
        "}"
```

Implementations may choose to limit the allowable set of declarations that can be placed within a forall-declaration block.

The scope of the forall identifier is restricted to its forall declaration block. Forall indentifiers are typed as **int**.

Constant Declarations

Constant declarations are used to introduce names for constant values or expressions. Constants can be of any of the pre-defined types,.

Examples

```
double pi = 3.14159        @ bits(64);
string alphabet = "abcdef" @ share(true)
```

Syntax

```
const-declaration =
   [1 int | double | string | boolean ]
         CONSTANT-id
          "=" expression
   [0+ tag ]
```

Assert Declarations

Assert declarations are used to perform integrity checks during elaboration.

```
assert-declaration =
   assert BOOLEAN-expression
         [0+ tag ]
```

If an assertion fails then a error is produced and elaboration of the Darwin component is aborted.

Tag Declarations

Darwin declarations can optionally have one or more tags. Tags consist of a identifier and a set of expressions. Tags are a mechanism to attach non-structural information (e.g. resource specifications, constraints) to a Darwin specification.

Examples

```
@ family (TV-set)
@ layout (circle,45, 12,"beta") @ family (TV-set)
@ trader ("trader.doc.ic.ac.uk", 6666);
```

Syntax

```
tag =
   "@" TAG-id [01 TAG-argument-list ]
```

In this document tag identifiers are shown in italics.

Tag Elaboration

Darwin implementations evaluate tag expressions and make them available them to Darwin plug-ins (e.g. the Regis code generator, SAA, IDL-stub generators, Tracta). Darwin implementations allow plug-ins to read and write tags and generate new ones during component elaboration. In addition, Darwin compilers may provide some compiler information as tagged data (e.g. source-code tracking data, component dependencies).

External Declarations

External declarations are used to introduce externally written definitions into Darwin (e.g. IDL and Regis definitions). The handling of externally written definitions is implementation-defined.

Examples

```
spec IDL {
      #include "SQLinterface.h"
      interface myextension : SQLselect {
            long alive (in long id);
      }
}

spec REGIS {
      typdef entry <int, int> OpenT;
}

spec LTS {
      SEMA = (a -> b, b -> c) @ {a, b, c}.
}
```

Syntax

```
external-declaration =
   spec EXTERNAL-id  "{"
         any-characters -excluding  }            // use \} for embedding }
   "}"
```