# The Fortress Language Specification

May 28, 2012

# Contents

# Chapter 1

# Introduction

# Chapter 2

# Overview of Fortress

# Chapter 3

# Getting Started

# Chapter 4

# Programs

# Chapter 5

# Types

This chapter should be revised according to Victor's tuple story (his email titled "Tuple story" on 12/07/07).

No dimensions and units.

The Fortress type system supports a mixture of nominal and structural types. Specifically, Fortress provides nominal trait types and some constructs that combine types structurally. There are also a few special types. Most trait types are defined by trait declarations in a program (often in libraries), but a few are built-in.

Victor: This doesn't cover the special types, dimension types, or object expression types. But I think it's okay for this intro, as long as we don't assert that these are all the types.

Every expression has a *static type*, and every value has an *ilk* (also known as a *dynamic or runtime type*). Fortress programs are checked before they are executed to ensure that if an expression $e$ evaluates to a value $v$, the runtime type of $v$ is a subtype of the static type of $e$. Sometimes we abuse terminology by saying that an expression has the runtime type of the value it evaluates to (see Section **??** for a discussion about evaluation of expressions).

Some types may be parameterized by *static parameters*; we call these types *generic types*. See Chapter **??** for a discussion of static parameters. A static parameter may be instantiated with a type or a value depending on whether it is a type parameter.

Victor: I don't know if the follow really belongs here. It isn't complete in any case, because of union/intersection types.

Two types are identical if and only if they are the same kind and their names and static arguments (if any) are identical. Types are related by several relationships as described in Section 5.2.

Syntactically, the positions in which a type may legally appear (i.e., *type contexts*) is determined by the nonterminal *Type* in the Fortress grammar, defined in Appendix **??**.

## 5.1 Kinds of Types

Fortress supports the following kinds of types:

- trait types,
- object expression types,
- tuple types,
- arrow types,
- function types, and
- three special types: $\mathrm{Any}$, $\mathrm{BottomType}$ and $()$.

Object expression types, function types and $\mathrm{BottomType}$ are not first-class types: they cannot be written in a program. However, some values have object expression types and function types and `throw` and `exit` expressions have the type $\mathrm{BottomType}$.

Collectively, trait types and object expression types are *defined* types; the trait and object declarations and object expressions are the types' *definitions*.

Victor:
I don't know if you like this terminology for trait types and object expression types. Putting this all together allowed some repetition to be eliminated, and emphasizes the similarity, which I think you wanted to retain. I still want to separate object expression types from object trait types. I think the difference here is not as evident as it might be if we handled generics properly: it is the type environment of object expression types that make them a pain, and I'm not dealing with that for the most part.

## 5.2 Relationships between Types

We define two relations on types: *subtype* and *exclusion*.

add coercion in full language

The subtype relation is a partial order on types that defines the type hierarchy; that is, it is reflexive, transitive and antisymmetric. $\mathrm{Any}$ is the top of the type hierarchy: all types are subtypes of $\mathrm{Any}$. $\mathrm{BottomType}$ is the bottom of the type hierarchy: all types are supertypes of $\mathrm{BottomType}$. For types $T$ and $U$, we write $T \preceq U$ when $T$ is a subtype of $U$, and $T \prec U$ when $T \preceq U$ and $T \neq U$; in the latter case, we say $T$ is a *strict* subtype of $U$. We also say that $T$ is a *supertype* of $U$ if $U$ is a subtype of $T$. Thus, in the execution of a valid Fortress program, an expression's runtime type is always a subtype of its static type. We say that a value is *an instance of* its runtime type and of every supertype of its runtime type; immediate subtypes of $\mathrm{Any}$ comprises of tuple types, arrow types, $()$, and $\mathrm{Object}$.

The exclusion relation is a symmetric relation between two types whose intersection is $\mathrm{BottomType}$. Because $\mathrm{BottomType}$ is uninhabited (i.e., no value has type $\mathrm{BottomType}$), types that exclude each other are disjoint: no value can have a type that is a subtype of two types that exclude each other.

The converse is technically not true: we can define two traits with method declarations such that no trait or object can extend them both, so no value can have a type that is a subtype of both trait types, but the trait types do not exclude each other. For example: trait A f(self, y: Object) = 1; end trait B f(x: Object, self) = 2; end trait C f() = 3; end
No pair of the three trait types defined above can be extended but none of them exclude any of the others.

Note that $\mathrm{BottomType}$ excludes every type including itself (because the intersection of $\mathrm{BottomType}$ and any type is $\mathrm{BottomType}$), and also that no type other than $\mathrm{BottomType}$ excludes $\mathrm{Any}$ (because the intersection of any type $T$ and $\mathrm{Any}$ is $T$). $\mathrm{BottomType}$ is the only type that excludes itself. Also note that if two types exclude each other then any subtypes of these types also exclude each other.

Fortress also allows *coercion* between types (see Chapter **??**). A coercion from $T$ to $U$ is defined in the declaration of $U$. We write $T \rightarrow U$ if $U$ defines a coercion from $T$. We say that $T$ *can be coerced to* $U$, and write $T \rightsquigarrow U$, if $U$ defines a coercion from $T$ or any supertype of $T$: $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \rightarrow U$.

The Fortress type hierarchy is acyclic with respect to both subtyping and coercion relations except for the following:

- The trait $\mathrm{Any}$ is a single root of the type hierarchy and it forms a cycle as described in Chapter **??**.
- There exists a bidirectional coercion between two tuple types if and only if they have the same sorted form.

These relations are defined more precisely in the following sections describing each kind of type in more detail. Specifically, the relations are the smallest ones that satisfy all the properties given in those sections (and this one).

## 5.3 Trait Types

Syntax:

$$\mathit{Type} \quad ::= \quad \mathit{TraitType}$$

A *trait type* is a named type defined by a trait or object declaration. It is an *object trait type* if it is defined by an object declaration.

Victor: The following text really belongs to traits, not to types. Traits are declared by trait and object declarations (described in Chapter 10 and Chapter 11). A trait has a *trait type* of the same name.

A trait type is a subtype of every type that appears in the `extends` clause of its definition. In addition, every trait type is a subtype of $\mathrm{Any}$.

A trait declaration may also include an `excludes` clause, in which case the defined trait type excludes every type that appears in that clause. Every trait type also excludes every arrow type and every tuple type, and the special types $()$ and $\mathrm{BottomType}$.

### 5.3.1 Object Trait Types

An object trait type is a trait type defined by an object declaration. Object declarations do not include `excludes` clauses, but an object trait type cannot be extended. Thus, an object trait type excludes any type that is not its supertype.

## 5.4 Object Expression Types

An object expression type is defined by an object expression (described in Section **??**) and the program location of the object expression. Every evaluation of a given object expression has the same object expression type. Two object expressions at different program locations have different object expression types.

Like trait types, an object expression type is a subtype of all trait types that appear in the `extends` clause of its definition, as well as the type $\mathrm{Any}$.

Like object trait types, an object expression type excludes any type that is not its supertype.

## 5.5 Tuple Types

Syntax:

$$\begin{array}{lll} \mathit{TupleType} & ::= & (\ \mathit{Type},\ \mathit{TypeList}\ ) \\ \mathit{TypeList} & ::= & \mathit{Type}(,\ \mathit{Type})^* \end{array}$$

Victor: The following two sentences don't belong in this section: A tuple expression is an ordered sequence of expressions separated by commas and enclosed in parentheses. See Section **??** for a discussion of tuple expressions.

A tuple type consists of a parenthesized, comma-separated list of two or more types.

Every tuple type is a subtype of $\mathrm{Any}$. No other type encompasses all tuple types. Tuple types cannot be extended by trait types. Tuple types are covariant: a tuple type $X$ is a subtype of tuple type $Y$ if and only if they have the same number of element types and each element type of $X$ is a subtype of the corresponding element type of $Y$.

A tuple type excludes any non-tuple type other than $\mathrm{Any}$. A tuple type excludes every tuple type that does not have the same number of element types. Also, tuple types that have the same number of element types exclude each other if any pair of corresponding element types exclude each other.

Intersection of nonexclusive tuple types are defined elementwise; the intersection of nonexclusive tuple type $X$ and $Y$ is a tuple type with exactly corresponding elements, where the type in each element type is the intersection of the types in the corresponding element types of $X$ and $Y$. Note that intersection of any exclusive types is $\mathrm{BottomType}$ as described in Section 5.8.

## 5.6 Arrow Types

Arrow types should include `io`-ness.

Arrow types with contracts
Function contracts consist of three parts: a `requires` part, an `ensures` part, and an `invariant` part. All three parts are evaluated in the scope of the function body extended with a special variable *arg*, bound to an immutable array of all function arguments. (This array is useful for describing contracts in the presence of higher-order functions; see Section 5.6).
Is the special variable *arg* reserved? What if one of the function's parameters is named $arg$?
At runtime, when a function is bound to a variable or parameter $f$ whose type includes a contract, the contract is evaluated when the function is called through a reference to $f$. (Note that this contract is distinct from the contract attached to the function bound to $f$, which is also evaluated upon a call to $f$). This contract is evaluated in the enclosing scope of the arrow type, extended with any keyword argument names provided in the arrow type, and the special variable $arg$ bound to an immutable array containing the arguments provided at the call site (in the order provided at the call site). If the `requires` clauses stipulated in the type of $f$ are not satisfied, a $\mathrm{CallerViolation}$ exception is thrown. Otherwise, if the `requires` clauses attached to the function bound to $f$ is not satisfied, a $\mathrm{ContractBinding}$ exception is thrown. If any other part of the contract of the function bound to $f$ is not satisfied, a $\mathrm{CalleeViolation}$ exception is thrown. Otherwise, if any part of the contract stipulated in the type of $f$ is not satisfied, a $\mathrm{ContractBinding}$ exception is thrown.

Syntax:

| | | |
|---|---|---|
| *Type* | ::= | *ArgType* $\rightarrow$ *Type Throws*? |
| *ArgType* | ::= | ( (*Type*, )* *Type*... ) |
| | \| | *TupleType* |

The static type of an expression that evaluates to a function value is an *arrow type* (or possibly an intersection of arrow types). An arrow type has three constituent parts: a *parameter type*, a *return type*, and a set of *exception types*. (Multiple parameters or return values are handled by having tuple types for the parameter type or return type respectively.)

Victor: I'd rather say "an exception type", where that type can be a union type, but I'm avoiding mentioning union types for now.

Syntactically, an arrow type consists of the parameter type followed by the token $\rightarrow$, followed by the return type, and optionally a `throws` clause that specifies the set of exception types.

Victor: What about io? Ignoring that for now.

If there is no `throws` clause, the set of exception types is empty.

The parameter type of an arrow type may end with a "varargs entry", $T\ldots$. The resulting parameter type is not a first-class type; rather, it is the union of all the types that result from replacing the varargs entry with zero or more entries of the type $T$. For example, $(T\ldots)$ is the union of $()$, $T$, $(T,T)$, $(T,T,T)$, and so on.

We say that an arrow type is *applicable* to a type $A$ if $A$ is a subtype of the parameter type of the arrow type. (If the parameter type has a varargs entry, then $A$ must be a subtype of one of the types in the union defined by the parameter type.)

Every arrow type is a subtype of $\mathrm{Any}$. No other type encompasses all arrow types. A type parameter can be instantiated with an arrow type. Arrow types cannot be extended by trait types. Arrow types are covariant in their return type and exception types and contravariant in their parameter type; that is, arrow type "$A \rightarrow B$ `throws` $C$" is a subtype of arrow type "$D \rightarrow E$ `throws` $F$" if and only if:

- $D$ is a subtype of $A$,
- $B$ is a subtype of $E$, and
- for all $X$ in $C$, there exists $Y$ in $F$ such that $X$ is a subtype of $Y$.

For arrow types $S$ and $T$, we say that $S$ is *more specific than* $T$ if the parameter type of $S$ is a subtype of the parameter type of $T$. We also say that $S$ is *more restricted than* $T$ if the return type of $S$ is a subtype of the return type of $T$ and for every $X$ in the set of exception types of $S$, there exists $Y$ in the set of exception types of $T$ such that $X$ is a subtype of $Y$. Thus, $S$ is a subtype of $T$ if and only if $T$ is more specific than $S$ and $S$ is more restricted than $T$.

An arrow type excludes any non-arrow type other than $\mathrm{Any}$ and the function types that are its subtypes (see Section 5.7). However, arrow types do not exclude other arrow types because of overloading as described in Chapter **??**.

Here are some examples:

## 5.7 Function Types

Function types are the runtime types of function values. We distinguish them from arrow types to handle overloaded functions. A function type consists of a finite set of arrow types. However, not every set of arrow types is a well-formed function type. Rather, a function type $F$ is *well-formed* if, for every pair $(S_1, S_2)$ of distinct arrow types in $F$, the following properties hold:

- the parameter types of $S_1$ and $S_2$ are not the same,
- if $S_1$ is more specific than $S_2$ then $S_1$ is more restricted than $S_2$, and
- if the intersection of the parameter types of $S_1$ and $S_2$ is not $\mathrm{BottomType}$ (i.e., the parameter types of $S_1$ and $S_2$ do not exclude each other) then $F$ has some constituent arrow type that is more specific than both $S_1$ and $S_2$ (recall that the more specific relation is reflexive, so the required constituent type may be $S_1$ or $S_2$).

Henceforth, we consider only well-formed function types. The overloading rules ensure that all function values have well-formed function types.

We say that a function type $F$ is *applicable* to a type $A$ if any of its constituent arrow types is applicable to $A$. We extend this terminology to values of the corresponding types. That is, for example, we say that a function of type $F$ is applicable to a value of type $A$ if $F$ is applicable to $A$. Note that if a well-formed function type $F$ is applicable to a type $A$, then among all constituent arrow types of $F$ that are applicable to $A$ (and there must be at least one), one is more specific than all the others. We say that this constituent type is the *most specific* type of $F$ applicable to $A$.

A function type is a subtype of each of its constituent arrow types, and also of $\mathrm{Any}$. Like object trait types and object expression types, a function type excludes every type that is not its supertype.

## 5.8 Special Types

Fortress provides three special types: $\mathrm{Any}$, $\mathrm{BottomType}$ and $()$. The type $\mathrm{Any}$ is the top of the type hierarchy: it is a supertype of every type. The only type it excludes is $\mathrm{BottomType}$.

The type $()$ is the type of the value $()$. Its only supertype (other than itself) is $\mathrm{Any}$, and it excludes every other type.

Fortress provides a special *bottom type*, $\mathrm{BottomType}$, which is an uninhabited type. No value in Fortress has $\mathrm{BottomType}$; `throw` and `exit` expressions have $\mathrm{BottomType}$. $\mathrm{BottomType}$ is a subtype of every type and it excludes every type (including itself). As mentioned above, $\mathrm{BottomType}$ is not a first-class type: programmers must not write $\mathrm{BottomType}$.

Victor's comments The complex numbers have precision like the integers, but are they intended to be only integral complex numbers? That seems odd to me. Also, why say "imaginary and complex", since we don't provide a separate type for just the imaginary numbers? Guy's going to provide APIs for complex numbers.

## 5.9 Types in the Fortress Standard Libraries

The Fortress standard libraries define simple standard types for literals such as $\mathrm{BooleanLiteral}[\![b]\!]$, $()$ (pronounced "void"), $\mathrm{Character}$, $\mathrm{String}$, and $\mathrm{Numeral}[\![n, m, r, v]\!]$ for appropriate values of $b$, $n$, $m$, $r$, and $v$ (See Section 12.1

for a discussion of Fortress literals). Moreover, there are several simple standard numeric types. These types are mutually exclusive; no value has more than one of them. Values of these types are immutable.

The numeric types share the common supertype $\mathrm{Number}$. Fortress includes types for arbitrary-precision integers (of type $\mathbb{Z}$), their unsigned equivalents (of type $\mathbb{N}$), rational numbers (of type $\mathbb{Q}$), real numbers (of type $\mathbb{R}$), complex numbers (of type $\mathbb{C}$), fixed-size representations for integers including the types $\mathbb{Z}8$, $\mathbb{Z}16$, $\mathbb{Z}32$, $\mathbb{Z}64$, $\mathbb{Z}128$, their unsigned equivalents $\mathbb{N}8$, $\mathbb{N}16$, $\mathbb{N}32$, $\mathbb{N}64$, $\mathbb{N}128$, floating-point numbers (described below), intervals (of type $\mathrm{Interval}[\![X]\!]$, abbreviated as $\langle\!|X|\!\rangle$, where $X$ can be instantiated with any number type), and imaginary and complex numbers of fixed size (in rectangular form with types $\mathbb{C}n$ for $n = 16, 32, 64, 128, 256$ and polar form with type $\mathrm{Polar}[\![X]\!]$ where $X$ can be instantiated with any real number type).

The Fortress standard libraries also define other simple standard types such as $\mathrm{Any}$, $\mathrm{Object}$, $\mathrm{Exception}$, $\mathrm{Boolean}$, and $\mathrm{BooleanInterval}$ as well as low-level binary data types such as $\mathrm{LinearSequence}$, $\mathrm{HeapSequence}$, and $\mathrm{BinaryWord}$. See Parts **??** and **??** for discussions of the Fortress standard libraries.

## 5.10 Intersection and Union Types

For every finite set of types, there is a type denoting a unique *intersection* of those types. The intersection of a set of types $S$ is a subtype of every type $T \in S$ and of the intersection of every subset of $S$. There is also a type denoting a unique *union* of those types. The union of a set of types $S$ is a supertype of every type $T \in S$ and of the union of every subset of $S$. Neither intersection types nor union types are first-class types; they are used solely for type inference (as described in Chapter 17) and they cannot be expressed directly in programs.

The intersection of a set of types $S$ is equal to a named type $U$ when any subtype of every type $T \in S$ and of the intersection of every subset of $S$ is a subtype of $U$. Similarly, the union of a set of types $S$ is equal to a named type $U$ when any supertype of every type $T \in S$ and of the union of every subset of $S$ is a supertype of $U$. For example:

```
trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end
```

because of the `comprises` clauses of $S$ and $T$ and the `excludes` clause of $U$, any subtype of both $S$ and $T$ must be a subtype of $V$. Thus, $V = S \cap T$.

Intersection types (denoted by $\cap$) possess the following properties:

- Commutativity: $T \cap U = U \cap T$.
- Associativity: $S \cap (T \cap U) = (S \cap T) \cap U$.
- Subsumption: If $S \preceq T$ then $S \cap T = S$.
- Preservation of shared subtypes: If $T \preceq S$ and $T \preceq U$ then $T \preceq S \cap U$.
- Preservation of supertype: If $S \preceq T$ then $\forall U.\ S \cap U \preceq T$.
- Distribution over union types: $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$.

Union types (denoted by $\cup$) possess the following properties:

- Commutativity: $T \cup U = U \cup T$.
- Associativity: $S \cup (T \cup U) = (S \cup T) \cup U$.
- Subsumption: If $S \preceq T$ then $S \cup T = T$.
- Preservation of shared supertypes: If $S \preceq T$ and $U \preceq T$ then $S \cup U \preceq T$.
- Preservation of subtype: If $T \preceq S$ then $\forall U.\ T \preceq S \cup U$.
- Distribution over intersection types: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$.

## 5.11 Type Aliases

Syntax:

*TypeAlias* ::= `type` *Id StaticParams*? = *Type*

Fortress allows names to serve as aliases for more complex type instantiations. A *type alias* begins with `type` followed by the name of the alias type, followed by optional static parameters, followed by $=$, followed by the type it stands for. Parameterized type aliases are allowed but recursively defined type aliases are not. Here are some examples:

`type` $\mathrm{IntList} = \mathrm{List}[\![\mathbb{Z}64]\!]$
`type` $\mathrm{BinOp} = \mathrm{Float} \times \mathrm{Float} \rightarrow \mathrm{Float}$
`type` $\mathrm{SimpleFloat}[\![\texttt{nat}\ e, \texttt{nat}\ s]\!] = \mathrm{DetailedFloat}[\![\mathrm{Unity}, e, s, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}]\!]$

All uses of type aliases are expanded before type checking. Type aliases do not define new types nor nominal equivalence relations among types.

# Chapter 6

# Lexical Structure

## 6.1 Reserved Words

The following tokens are *reserved words*:

| | | | | | | |
|---|---|---|---|---|---|---|
| `BIG` | `FORALL` | `SI_unit` | `absorbs` | `abstract` | `also` | `api` |
| `asif` | `at` | `atomic` | `bool` | `case` | `catch` | `coerce` |
| `coerces` | `component` | `comprises` | `default` | `dim` | `do` | `elif` |
| `else` | `end` | `ensures` | `except` | `excludes` | `exit` | `export` |
| `extends` | `finally` | `fn` | `for` | `forbid` | `from` | `getter` |
| `hidden` | `if` | `import` | `int` | `invariant` | `io` | `juxtaposition` |
| `label` | `most` | `nat` | `native` | `object` | `of` | `opr` |
| `or` | `outcome` | `override` | `private` | `property` | `provided` | `requires` |
| `self` | `settable` | `setter` | `spawn` | `syntax` | `test` | `then` |
| `throw` | `throws` | `trait` | `try` | `tryatomic` | `type` | `typecase` |
| `typed` | `unit` | `value` | `var` | `where` | `while` | `widens` |
| `with` | `wrapped` | | | | | |

Victor: I don't think 'or" should be reserved. It is only so for its occurrence in 'widens or coerces" but we can recognize it specially in that context, which is never ambiguous because 'widens" and 'coerces" are reserved.

The following operators on units are also reserved words:
cubed cubic in inverse per square squared

To avoid confusion, Fortress reserves the following tokens:
`goto` `idiom` `public` `pure` `reciprocal` `static`

They do not have any special meanings but they cannot be used as identifiers.

Victor: Some other words we might want to reserve: subtype, subtypes, is, coercion, function, exception, match

# Chapter 7

# Names and Declarations

# Chapter 8

# Variables

# Chapter 9

# Functions

# Chapter 10

# Traits

```
export Executable
run() = println("Hello, world!")
```

| | | |
|---|---|---|
| File | ::= | CompilationUnit |
| | \| | $\text{Imports}^{?}$ Exports $\text{Decls}^{?}$ |
| | \| | $\text{Imports}^{?}$ AbsDecls |
| | \| | Imports $\text{AbsDecls}^{?}$ |
| CompilationUnit | ::= | Component |
| | \| | Api |

# Chapter 11

# Objects

# Chapter 12

# Expressions

## 12.1 Literals

A literal (Section **??**) denotes a fixed, unchanging value.

| | | |
|---|---|---|
| Literal | ::= | ( ) |
| | \| | BooleanLiteral |
| | \| | CharacterLiteral |
| | \| | StringLiteral |
| | \| | NumericLiteral |

The type of the literal $()$ is $()$. The type of a Boolean literal is $\mathrm{Boolean}$. The type of a character literal is $\mathrm{Character}$. The type of a string literal is $\mathrm{String}$. The type of a numeric literal is $\mathbb{Q}$ if it contains a '.' character, and otherwise is $\mathbb{N}$.

Evaluation of a literal always completes normally and produces the value represented by the literal.

All literals represent value objects; therefore their values do not have object identity.

## 12.2 Identifier References

## 12.3 Dotted Field Accesses

## 12.4 Function Calls

## 12.5 Operator Applications

## 12.6 Tuple Expressions

## 12.7 Aggregate Expressions

## 12.8 Function Expressions

## 12.9 Blocks

A block is a series of one or more declarations and expressions that are to be evaluated sequentially. A block is not in itself syntactically an expression, but appears as part of a `do` expression Section 12.10, `label` expression Section 12.11, `while` expression Section 12.12, `for` expression Section 12.14, `if` expression Section 12.18, `case` expresion Section 12.19, `typecase` expression Section 12.20, or `try` expression Section 12.23. For purposes of type-checking and evaluation, it may be regarded as an expression.

```
Block           ::=  BlockElement+
BlockElement    ::=  LocalVarFnDecl
                 |   Expr ( , GeneratorList )?
```

It is a static error if the last BlockElement in a Block is an expression of the form "$a = b$" that cannot be regarded as a local declaration. It is a static error if any block element other than the last one is an expression whose type is not $()$. If the last block element of a block is a declaration, then the type of the block is $()$; if the last block element of a block is an expression, then the type of the block is the type of that expression.

Every block introduces a new scope. For the scoping behavior of local declarations within a block, see Section **??** and Section **??**.

Not shown in the grammar is the fact that adjacent block elements may be separated by a semicolon ';' if desired. This is necessary if the block elements appear in the same line of source code. If a newline occurs between two adjacent block elements, then no semicolon is necessary.

A block is evaluated by evaluating its block elements sequentially, in order from left to right; evaluation of each block element must complete before evaluation of the next can begin. If evaluation of any block element completes abruptly for some reason, then evaluation of the block completes abruptly for the same reason, and no further block elements are evaluated. If the evaluations of all block elements complete normally, then the value of the block is the value of the last block element.

If it is desired to have, as a non-final block element, an expression $e$ whose type is not $()$, one may instead write the local variable declaration $\_ = e$, thus signifying explicitly that the value of $e$ is to be discarded (by binding an anonymous variable to the value).

If it is desired to have, as a final block element, an equality test expression of the form $a = b$, simply enclose the expression in parentheses.

Here is an example of a `do` expression that contains a single block having six block elements:

```
do
  f(w: ℤ32) = w + 1      ⊛ Local function declaration
  y = x + 1              ⊛ Local variable declaration (immutable)
  println("y is" y)      ⊛ Expression (has a useful side effect)
  var z: ℝ64 = 0         ⊛ Local variable declaration (mutable)
  z += f(y)              ⊛ Compound assignment
  |z|                    ⊛ Expression (the value of the block)
end
```

## 12.10 Do and Do-Also Expressions

In the simplest case, the keywords `do` and `end` surround a single block to be executed. In the more general case, a `do` expression can contain multiple blocks to be executed independently (perhaps but not necessarily, concurrently). Each block may be governed by its own `atomic` modifier.

```
DoExpr      ::=  ( DoFront also )* DoFront end
DoFront     ::=  ( at Expr )? atomic? do Block?
```

If a `do` expression contains a single Block, then the type of the `do` expresssion is the type of the block. If a `do` expression contains two or more blocks (separated by `also`), then the type of the `do` expression is $()$, and it is a static error if any of the blocks has a type that is not $()$.

If the `do` keyword preceding any block of a multi-block `do` expression is preceded by an `atomic` modifier, it is as if (a) the following block were enclosed by a new pair of `do` and `end` keywords to form a single-block `do` expression, and (b) the `atomic` modifier appeared instead before that single-block `do` expression, thus forming an `atomic` expression. Thus, allowing `atomic` to appear as part of a multi-block `do` expression is merely a syntactic convenience that allows programs to be slightly shorter and perhaps read more naturally. For example:

```
atomic do
  x += 1
  y += 2
also atomic do
  b += 1
  y += 3
end
```

is simply an abbreviation of

```
do
  atomic do
    x += 1
    y += 2
  end
also do
  atomic do
    b += 1
    y += 3
  end
end
```

A `do` expression with a single block is evaluated by evaluating the block. If the block completes abruptly for some reason, then the `do` expression completes abruptly for the same reason. Otherwise, the value of the `do` expression is the value of the block.

A `do` expression with multiple blocks block is evaluated by evaluating all the blocks independently (perhaps, but not necessarily, concurrently). If more blocks complete abruptly for some reason, then the `do` expression completes abruptly for one of those reasons (evaluation of other blocks may or may not be completed, but evaluation of the `do` expression is not complete until the evaluations of all blocks have been terminated). Otherwise, evaluation of the `do` expression completes only after evaluation of all blocks is complete, and the value of the `do` expression is $()$.

## 12.11 Label and Exit

## 12.12 While Loops

The `while` statement evaluates a test and a `do` expression, sequentially and repeatedly, until the test fails. The test may be either a Boolean expression or a generator binding.

```
WhileExpr          ::=  while Generator DoExpr
Generator          ::=  GeneratorBinding
                    |   Expr
GeneratorBinding   ::=  Id ← Expr
                    |   ( Id , IdList ) ← Expr
```

If the Generator is an expression, it is a static error if the type of the expression does not conform to $\mathrm{Boolean}$. If the Generator is a generator binding, it is a static error if the type of the expression in the generator binding does not conform to $\mathrm{Condition}[\![T]\!]$ for some $T$. The type of a `while` expression is $()$.

A `while` expression with an expression is evaluated by first evaluating Expr. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason. Otherwise, evaluation continues by making a choice based on the resulting value $v$. If $v$ is $\mathit{false}$, no further action is taken; evaluation of the `while` expression completes normally with value $()$. But if $v$ is $\mathit{true}$, then the `do` expression is evaluated. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason; otherwise, the entire `while` expression is evaluated again (beginning by re-evaluating Expr). For example:

```
do
  ⊛ Print the numbers 0 through 9 on separate lines.
  $k\colon \mathbb{Z} := 0$
  while $(k < 10)$ do
    $\mathit{println}(k)$
    $k \mathrel{+}= 1$
  end
end
```

A `while` expression with a generator binding is evaluated by first evaluating the Expr in the generator binding. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason. Otherwise, evaluation continues by making a choice based on the resulting value $v$. If $v$ does not contain a value, no further action is taken; evaluation of the `while` expression completes normally with value $()$. If $v$ contains a value $w$, then the pattern in the generator binding is matched to that value $w$ and then the `do` expression is evaluated, and variables bound by the pattern are visible within the `do` expression. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason; otherwise, the entire `while` expression is evaluated again (beginning by re-evaluating Expr). For example:

```
object $\mathrm{Chain}\big(\mathit{item}\colon \mathbb{Z}, \mathit{link}\colon \mathrm{Maybe}[\![\mathrm{Chain}]\!]\big)$ end
$\mathit{printValues}\big(x\colon \mathrm{Maybe}[\![\mathrm{Chain}]\!]\big)\colon () =$ do
    $\mathit{cursor}\colon \mathrm{Maybe}[\![\mathrm{Chain}]\!] = x$
    while $(\mathit{next} \leftarrow \mathit{cursor})$ do
      $\mathit{println}(\mathit{next}.\mathit{item})$
      $\mathit{cursor} := \mathit{next}.\mathit{link}$
    end
  end
end
$\mathit{run}() = \mathit{printValues}\big(\mathrm{Just\ Chain}(1, \mathrm{Just\ Chain}(2, \mathrm{Just\ Chain}(3, \mathrm{Just\ Chain}(4, \mathrm{None}))))\big)$
```

## 12.13 Generators

## 12.14 For Expressions

## 12.15 Ranges

## 12.16 Reduction Expresions

## 12.17 Comprehensions

## 12.18 If Expressions

The `if` statement allows conditional evaluation of a block or conditional evaluation of at most one of a set of blocks. The choice is made by sequentially executing one or more tests and choosing the first statement for which a test succeeds. Each test may be either a Boolean expression or a generator binding.

| | | |
|---|---|---|
| IfExpr | ::= | `if` Generator `then` Block ElifClause$^*$ ElseClause$^?$ `end` |
| | \| | [(] `if` Generator `then` Block ElifClause$^*$ ElseClause `end`$^?$ [)] |
| Generator | ::= | GeneratorBinding |
| | \| | Expr |
| GeneratorBinding | ::= | Id $\leftarrow$ Expr |
| | \| | ( Id , IdList ) $\leftarrow$ Expr |
| ElifClause | ::= | `elif` Expr `then` Block |
| ElseClause | ::= | `else` Block |

An `if` expression consists of `if` followed by a Generator clause (discussed in Section 12.13), followed by `then`, a Block, a possibly empty sequence of `elif` clauses (each consisting of `elif` followed by a Generator clause, `then`, and a Block), an optional `else` clause (consisting of `else` followed by a Block), and finally `end`. The reserved word `end` may be elided if the `if` expression is immediately enclosed by parentheses; in such a case, the `else` clause is required, not optional.

Each Block is a series of one or more block elements (declarations and expressions). See Section **??** for a description of the various syntactic and semantic properties of blocks.

For each Generator, if it is an expression, it is a static error if the type of the expression does not conform to $\mathrm{Boolean}$; if it is a generator binding, it is a static error if the type of the expression in the generator binding does not conform to $\mathrm{Condition}[\![T]\!]$ for some $T$. If the `if` expression has no `else` clause, it is a static error if the type of the block after the first `then`, or in any `elif` clause, is not $()$. The type of an `if` expression is the union of the types of all its blocks.

An `if` expression whose first Generator is an expression is evaluated by first evaluating Expr. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, evaluation continues by making a choice based on the resulting value $v$. If $v$ is $\mathit{true}$, then the first block is evaluated. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, the value of the `if` expression is value of this block. If $v$ is $\mathit{false}$, then `elif` clauses are considered (see below).

An `if` expression whose first Generator is a generator binding is evaluated by first evaluating the Expr in the generator binding. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, evaluation continues by making a choice based on the resulting value $v$. If $v$ contains a value $w$, then the pattern in the generator binding is matched to that value $w$ and then the first block is evaluated and variables bound by the pattern are visible within the block. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, the value of the `if` expression is value of this block. If $v$ does not contain a value, then `elif` clauses are considered (see below).

If evaluation of an `if` expression must consider `elif` clauses, they are examined sequentially, working from left to right, treating each **??** and each block in exactly the same manner as the first generator and first block of the `if` expression. As soon as a generator is found that produces the value $true$ or a condition value $v$ that contains another value $w$, the corresponding block is evaluated, and its value becomes the value of the `if` expression; but if evaluation of any Generator or block completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason. If consideration of `elif` clauses does not result in evaluating an block (possibly because no `elif` clauses are present), then any `else` clause is considered.

If evaluation of an `if` expression must consider any `else` clause, there are two cases. If an `else` clause is present, then its block is evaluated, and its value becomes the value of the `if` expression; but if evaluation of the block completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason. If no `else` clause is present, then evaluation of the `if` expression completes normally with value `()`.

Examples:

```
if $x > 0$ then $println\ x$ end

if $x > 0$ then
  $println(x$ “is positive”$)$
else
  $println(x$ “is nonpositive”$)$
end

$println\big(x$ “is” (if $v > 0$ then “positive” else “nonpositive”$)\big)$

$z =$ if $x < 0$ then $0$
     elif $x \in \{1, 2, 3\}$ then $3$
     elif $x \in \{4, 5, 6\}$ then $6$
     else $9$ end
```

## 12.19 Case Expressions

## 12.20 Typecase Expressions

## 12.21 Atomic Expressions

## 12.22 Throw Expressions

## 12.23 Try Expressions

## 12.24 Type Ascription

## 12.25 Asif Expressions

# Chapter 13

# Exceptions

# Chapter 14

# Operators

Multifix operators and dimensions and units are not yet supported.

Synonym Operators: Victor's email titled "Synonyms [Fwd: Re: What the heck are 0-width spaces for?]" on 09/12/07

Operators are like functions or methods; operator declarations are described in Chapter **??** and operator applications are described in Section **??**, Section **??**, and Section 12.17. Operators are not values; when an operator is passed as an argument to a function, it should be eta expanded. Just as functions or methods may be overloaded (see Chapter **??** for a discussion of overloading), so operators may have overloaded declarations of the same fixity. Operator declarations with the same operator name but with different fixities are valid declarations because it is always unambiguous which declaration shall be applied to an application of the operator. Calls to overloaded operators are resolved first via the fixity of the operators based on the context of the calls. Then, among the applicable declarations with that fixity, the most specific declaration is chosen.

Most operators can be used as prefix, infix, postfix, or nofix operators as described in Section 14.4 (nofix operators take no arguments); the fixity of an operator is determined syntactically, and the same operator may have declarations for multiple fixities. A simple example is that '$-$ ' may be either infix or prefix, as is conventional. As another example, '! ' may be a postfix operator that computes factorial when applied to integers. These operators might not be used as enclosing operators.

Several pairs of operators can be used as enclosing operators. Any number of '$|$' (vertical line) can be used as both infix operators and enclosing operators.

Some operators are always postfix: a '^' followed by any ordinary operator (with no intervening whitespace) is considered to be a superscripted postfix operator. For example, '$^\wedge *$ ' and '$^\wedge +$ ' and '$^\wedge ?$ ' are available for use as part of the syntax of extended regular expressions. As a very special case, '$^\wedge T$ ' is also considered to be a superscripted postfix operator, typically used to signify matrix transposition.

Finally, there are special operators such as juxtaposition and operators on dimensions and units. Juxtaposition may be a function application or an infix operator in Fortress. When the left-hand-side expression is a function, juxtaposition performs function application; when the left-hand-side expression is a number, juxtaposition conventionally performs multiplication; when the left-hand-side expression is a string, juxtaposition conventionally performs string concatenation. Fortress provides several operators on dimensions and units as described in Chapter **??**.

## 14.1 Operator Names

## 14.2 Operator Names

Victor: This should refer to the appropriate sections of Lexical Structure Operator names are either operator tokens or special operator characters, or a few reserved words.

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as these characters:

```
!    @    #    $    %    *    +    -    =    |    :    <    >    /    ?    ^    ~
->    -->    =>    ==>    <=    >=    =/=    !!    ||    |||
<<    <<<    >>    >>>    <->    <-/-    -/->    <=>    ===
```

In addition, a token that is made up of a mixture of uppercase letters and underscores (but no digits), does not begin or end with an underscore, and contains at least two different letters is also considered to be an operator:

```
MAX   MIN
```

The above operators are rendered as: MAX MIN. (See Section **??** and Appendix **??** for detailed descriptions of operator names in Fortress.)

## 14.3 Operator Precedence and Associativity

Fortress specifies that certain operators have higher precedence than certain other operators and certain operators are associative, so that one need not use parentheses in all cases where operators are mixed in an expression. (See Appendix **??** for a detailed description of operator precedence and associativity in Fortress.) However, Fortress does not follow the practice of other programming languages in simply assigning an integer to each operator and then saying that the precedence of any two operators can be compared by comparing their assigned integers. Instead, Fortress relies on defining traditional groups of operators based on their meaning and shape, and specifies specific precedence relationships between some of these groups. If there is no specific precedence relationship between two operators, then parentheses must be used. For example, Fortress does not accept the expression $a + b \cup c$; one must write either $(a + b) \cup c$ or $a + (b \cup c)$. (Whether or not the result then makes any sense depends on what definitions have been made for the $+$ and $\cup$ operators—see Chapter **??**.)

Here are the basic principles of operator precedence and associativity in Fortress:

- Member selection (`.`) and method invocation ($.name(\ldots)$) are not operators. They have higher precedence than any operator listed below.
- Subscripting (`[ ]` and any kind of subscripting operators, which can be any kind of enclosing operators), superscripting (`^`), and postfix operators have higher precedence than any operator listed below; within this group, these operations are left-associative (performed left-to-right).
- *Tight juxtaposition*, that is, juxtaposition without intervening whitespace, has higher precedence than any operator listed below. The associativity of tight juxtaposition is type-dependent; see Section 14.9.
- Next, *tight fractions*, that is, the use of the operator '/' with no whitespace on either side, have higher precedence than any operator listed below. The tight-fraction operator has no precedence compared with itself, so it is not permitted to be used more than once in a tight fraction without use of parentheses.
- *Loose juxtaposition*, that is, juxtaposition with intervening whitespace, has higher precedence than any operator listed below. The associativity of loose juxtaposition is type-dependent and is different from that for tight juxtaposition; see Section 14.9. Note that *lopsided juxtaposition* (having whitespace on one side but not the other) is a static error as described in Section 14.4.
- Prefix operators have higher precedence than any operator listed below. However, it is a static error for an operand of a loose prefix operator to be an operand of a tight infix operator.
- The infix operators are partitioned into certain traditional groups, as explained below. They have higher precedence than any operator listed below.

- The equal symbol '$=$' in binding context, the assignment operator '$:=$', and compound assignment operators ($+=$, $-=$, $\wedge=$, $\vee=$, $\cap=$, $\cup=$, and so on as described in Section **??**) have lower precedence than any operator listed above. Note that compound assignment operators themselves are not operator names.

The infix binary operators are divided into four general categories: arithmetic, relational, boolean, and other. The arithmetic operators are further categorized as multiplication/division/intersection, addition/subtraction/union, and other. The relational operators are further categorized as equivalence, inequivalence, ordering, and other. The boolean operators are further categorized as conjunctive, disjunctive, and other.

The arithmetic and relational operators are further divided into groups based on shape:

- "ordinary" operators: $+ - \cdot \times / \pm \mp \oplus \ominus \odot \otimes \oslash \boxplus \boxminus \boxdot \boxtimes < \leq \geq > \ll \lll \ggg \gg \nless \nleq \ngeq \ngtr$ etc.

  The arithmetic operations in this group are further subdivided into "plain" ($+ - \cdot \times / \pm \mp$ etc.), "circled" ( $\oplus \ominus \odot \otimes \oslash$ etc.), "boxed" ($\boxplus \boxminus \boxdot \boxtimes$ etc.), and so on; any of these groups may be used with the plain relational operators ($< \leq \geq > \ll \lll \ggg \gg \nless \nleq \ngeq \ngtr$ etc.), but the groups might not be mixed.

- "rounded horseshoe" or "set" operators: $\cap \Cap \cup \Cup \uplus \subset \subseteq \supseteq \supset \Subset \Supset \not\subset \nsubseteq \nsupseteq \not\supset$ etc.

- "square horseshoe" operators: $\sqcap \sqcup \sqsubset \sqsubseteq \sqsupseteq \sqsupset \not\sqsubseteq \not\sqsupseteq$ etc.

- "curly" operators: $\curlywedge \curlyvee \prec \preceq \succeq \succ \nprec \npreceq \nsucceq \nsucc$ etc.

- "triangular" relations: $\lhd \unlhd \unrhd \rhd \ntriangleleft \ntrianglelefteq \ntrianglerighteq \ntriangleright$ etc.

- "chickenfoot" relations: <--> etc.

The principles of precedence for binary operators are then as follows:

- A multiplication or division or intersection operator has higher precedence than any addition or subtraction or union operator that is in the same shape group.

- Certain addition and subtraction operators come in pairs, such as $+$ and $-$, or $\oplus$ and $\ominus$, which are considered to have the same precedence and so may be mixed within an expression and are grouped left-associatively. These addition-subtraction pairs are the *only* cases where two different operators are considered to have the same precedence.

- An arithmetic operator has higher precedence than any equivalence or inequivalence operator.

- An arithmetic operator has higher precedence than any relational operator that is in the same shape group.

- A relational operator has higher precedence than any boolean operator.

- A conjunctive boolean operator has higher precedence than any disjunctive boolean operator.

While the rules of precedence are complicated, they are intended to be both unsurprising and conservative. Note that operator precedence in Fortress is not always transitive; for example, while $+$ has higher precedence than $<$ (so you can write $a + b < c$ without parentheses), and $<$ has higher precedence than $\vee$ (so you can write $a < b \vee c < d$ without parentheses), it is *not* true that $+$ has higher precedence than $\vee$—the expression $a \vee b + c$ is not permitted, and one must instead write $(a \vee b) + c$ or $a \vee (b + c)$.

Another point is that the various multiplication and division operators do *not* have "the same precedence"; they may not be mixed freely with each other. For example, one cannot write $u \cdot v \times w$; one must write $(u \cdot v) \times w$ or (more likely) $u \cdot (v \times w)$. Similarly, one cannot write $a \odot b / c \odot d$; but juxtaposition does bind more tightly than a loose (whitespace-surrounded) division slash, so one is allowed to write $a\,b \;/\; c\,d$, and this means the same as $(a\,b)/(c\,d)$. On the other hand, loose juxtaposition binds less tightly than a tight division slash, so that $a\;b/c\;d$ means the same as $a\;(b/c)\;d$. On the other other hand, tight juxtaposition binds more tightly than tight division, so that $(n+1)/(n+2)(n+3)$ means the same as $(n+1)/((n+2)(n+3))$.

There are two additional rules intended to catch misleading code: it is a static error for an operand of a tight infix or tight prefix operator to be a loose juxtaposition, and it is a static error if the rules of precedence determine that a use of infix operator $a$ has higher or equal precedence than a use of infix operator $b$, but that particular use of $a$ is loose and that particular use of $b$ is tight. Thus, for example, the expression $\sin\ x\ +\ y$ is permitted, but $\sin\ x{+}y$ is not permitted. Similarly, the expression $a\ \cdot\ b\ +\ c$ is permitted, as are $a{\cdot}b\ +\ c$ and $a{\cdot}b{+}c$, but $a\ \cdot\ b{+}c$ is not permitted. (The rule detects only the presence or absence of whitespace, not the amount of whitespace, so $a\quad\cdot\ b\ +\ c$ is permitted. You have to draw the line somewhere.)

When in doubt, just use parentheses. If there's a problem, the compiler will (probably) let you know.

## 14.4 Operator Fixity

Multifix operators are not yet supported.

Most operators in Fortress can be used variously as prefix, postfix, infix, nofix, or multifix operators. (See Section 14.5 for a discussion of how infix operators may be chained or treated as multifix operators.)

Victor: I think that we shouldn't list "multifix" separately here. (Note that multifix does not appear in any of the tables.) Rather multifix is a way to interpret infix operators.

Some operators can be used in pairs as enclosing (bracketing) operators—see Section 14.6. The Fortress language dictates only the rules of syntax; whether an operator has a meaning when used in a particular way depends only on whether there is a definition in the program for that operator when used in that particular way (see Chapter **??**).

The fixity of a non-enclosing operator is determined by context. To the left of such an operator we may find (1) a *primary tail* (described below), (2) another operator, or (3) a comma, semicolon, or left encloser. To the right we may find (1) a *primary front* (described below), (2) another operator, (3) a comma, semicolon, or right encloser, or (4) a line break. A primary tail is an identifier, a literal, a right encloser, or a superscripted postfix operator (exponent operator). A primary front is an identifier, a literal, or a left encloser.

A primary expression is an identifier, a literal, an expression enclosed by matching enclosers, a field selection, or an expression followed by a postfix operator.

Considered in all combinations, this makes twelve possibilities. In some cases one must also consider whether or not whitespace separates the operator from what lies on either side. The rules of operator fixity are specified by Figure 14.1, where the center column indicates the fixity that results from the left and right context specified by the other columns.

A case described in the center column of the table as an **error** is a static error; for such cases, the fixity mentioned in parentheses is the recommended treatment of the operator for the purpose of attempting to continuing the parse in search of other errors.

The table may seem complicated, but it all boils down to a couple of practical rules of thumb:

1. *Any* operator can be prefix, postfix, infix, or nofix.
2. An infix operator can be *loose* (having whitespace on both sides) or *tight* (having whitespace on neither side), but it mustn't be *lopsided* (having whitespace on one side but not the other).
3. A postfix operator must have no whitespace before it and must be followed (possibly after some whitespace) by a comma, semicolon, right encloser, or line break.

## 14.5 Chained and Multifix Operators

Certain infix mathematical operators that are traditionally regarded as *relational* operators, delivering boolean results, may be *chained*. For example, an expression such as $A \subseteq B \subset C \subseteq D$ is treated as being equivalent to $(A \subseteq B) \wedge (B \subset C) \wedge (C \subseteq D)$ except that the expressions $B$ and $C$ are evaluated only once (which matters only if they have side effects such as writes or input/output actions). Similarly, the expression $A \subseteq B = C \subset D$ is treated as being equivalent to $(A \subseteq B) \wedge (B = C) \wedge (C \subset D)$, except that $B$ and $C$ are evaluated only once. Fortress restricts such chaining to a mixture of equivalence operators and ordering operators; if a chain contains two or more ordering operators, then they must be of the same kind and have the same sense of monotonicity; for example, neither

| **left context** | **whitespace** | **operator fixity** | **whitespace** | **right context** |
|---|---|---|---|---|
| primary tail | yes | **infix** | yes | primary front |
| primary tail | yes | **error** (infix) | no | primary front |
| primary tail | no | **postfix** | yes | primary front |
| primary tail | no | **infix** | no | primary front |
| primary tail | yes | **infix** | yes | operator |
| primary tail | yes | **error** (infix) | no | operator |
| primary tail | no | **postfix** | yes | operator |
| primary tail | no | **infix** | no | operator |
| primary tail | yes | **error** (postfix) | | `,` `;` right encloser |
| primary tail | no | **postfix** | | `,` `;` right encloser |
| primary tail | yes | **infix** | | line break |
| primary tail | no | **postfix** | | line break |
| operator | | **prefix** | | primary front |
| operator | | **prefix** | | operator |
| operator | | **error** (nofix) | | `,` `;` right encloser |
| operator | | **error** (nofix) | | line break |
| `,` `;` left encloser | | **prefix** | | primary front |
| `,` `;` left encloser | | **prefix** | | operator |
| `,` `;` left encloser | | **nofix** | | `,` `;` right encloser |
| `,` `;` left encloser | | **error** (prefix) | | line break |

Figure 14.1: Operator Fixity (I)

$A \subseteq B \leq C$ nor $A \subseteq B \supset C$ is permitted. This transformation is done before type checking. In particular, it is done even if these operators do not return boolean values, and the resulting expression is checked for type correctness. (See Section **??** for a detailed description of which operators may be chained.)

Any infix operator that does not chain may be treated as *multifix*. If $n-1$ occurrences of the same operator separate $n$ operands where $n \geq 3$, then the compiler first checks to see whether there is a definition for that operator that will accept $n$ arguments. If so, that definition is used; if not, then the operator is treated as left-associative and the compiler looks for a two-argument definition for the operator to use for each occurrence. As an example, the cartesian product $S_1 \times S_2 \times \cdots \times S_n$ of $n$ sets may usefully be defined as a multifix operator, but ordinary addition $p+q+r+s$ is normally treated as $((p+q)+r)+s$.

## 14.6 Enclosing Operators

These operators are always used in pairs as enclosing operators:

```
                (/   /)      (\   \)
[   ]           [/   /]                    [*   *]
{   }           {/   /}      {\   \}       {*   *}
                </   />      <\   \>
               <</   />>    <<\   \>>
```

(ASCII encodings are shown here; they all correspond to particular single Unicode characters.) There are other pairs as well, such as $\lfloor\ \ \rfloor$ and $\lceil\ \ \rceil$ and multicharacter enclosing operators described in Section **??**. Note that the pairs `(  )` and `[\  \]` (also known as $[\![\ \ ]\!]$) are not operators; they play special roles in the syntax of Fortress, and their behavior cannot be redefined by a library. The bracket pairs that may be used as enclosing operators are described in Section **??**.

Any number of '`|`' (vertical line) may also be used in pairs as enclosing operators but there is a trick to it, because on the face of it you can't tell whether any given occurrence is a left encloser or a right encloser. Again, context is

| left context | whitespace | operator fixity | whitespace | right context |
|---|---|---|---|---|
| primary tail | yes | **infix** | yes | primary front |
| | yes | **left encloser** | no | |
| | no | **right encloser** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **infix** | yes | operator |
| | yes | **left encloser** | no | |
| | no | **right encloser** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **error** (right encloser) | | `,` `;` right encloser |
| | no | **right encloser** | | |
| primary tail | yes | **infix** | | line break |
| | no | **right encloser** | | |
| operator | | **error** (left encloser) | yes | primary front |
| | | **left encloser** | no | |
| operator | | **error** (left encloser) | yes | operator |
| | | **left encloser** | no | |
| operator | | **error** (nofix) | | `,` `;` right encloser |
| operator | | **error** (nofix) | | line break |
| `,` `;` left encloser | | **left encloser** | | primary front |
| `,` `;` left encloser | | **left encloser** | | operator |
| `,` `;` left encloser | | **nofix** | | `,` `;` right encloser |
| `,` `;` left encloser | | **error** (left encloser) | | line break |

Figure 14.2: Operator Fixity (II)

used to decide, this time according to Figure 14.2.

This is very similar to Figure 14.1 in Section 14.4; a rough rule of thumb is that if an ordinary operator would be considered a prefix operator, then one of these will be considered a left encloser; and if an ordinary operator would be considered a postfix operator, then one of these will be considered a right encloser.

In this manner, one may use `|...|` for absolute values and `||...||` for matrix norms.

## 14.7 Conditional Operators

If a binary operator other than '`:`' and '`::`' is immediately followed by a '`:`' then it is *conditional*: evaluation of the right-hand operand cannot begin until evaluation of the left-hand operand has completed, and whether or not the right-hand operand is evaluated may depend on the value of the left-hand operand. If the left-hand operand throws an exception, then the right-hand operand is not evaluated.

The Fortress standard libraries define several conditional operators on boolean values including $\wedge$`:` and $\vee$`:`.

See Section **??** for a discussion of how conditional operators are declared.

## 14.8 Big Operators

A big operator application is either a *reduction expression* described in Section **??** or a *comprehension* described in Section 12.17.

The Fortress standard libraries define several big operators including $\sum$, $\prod$, and `BIG` $\wedge$.

See Section **??** for a discussion of how big operators are declared.

## 14.9 Juxtaposition

| Qualified names are not yet supported. |
|---|

Juxtaposition in Fortress may be a function call or a special infix operator. The Fortress standard libraries include several declarations of a `juxtaposition` operator.

When two expressions are juxtaposed, the juxtaposition is interpreted as follows: if the left-hand-side expression is a function, juxtaposition performs function application; otherwise, juxtaposition performs the `juxtaposition` operator application.

The manner in which a juxtaposition of three or more items must be associated requires type information and awareness of whitespace. (This is an inherent property of customary mathematical notation, which Fortress is designed to emulate where feasible.) Therefore a Fortress compiler must produce a provisional parse in which such multi-element juxtapositions are held in abeyance, then perform a type analysis on each element and use that information to rewrite the n-ary juxtaposition into a tree of binary juxtapositions.

All we need to know is whether each element of a juxtaposition has an arrow type. There are actually three legitimate possibilities for each element of a juxtaposition: (a) it has an arrow type, in which case it is considered to be a function element; (b) it has a type that is not an arrow type, in which case it is considered to be a non-function element. (c) it is an identifier that has no visible declaration, in which case it is considered to be a function element (and everything will work out okay if it turns out to be the name of an appropriate functional method).

The rules below are designed to forbid certain forms of notational ambiguity that can arise if the name of a functional method happens to be used also as the name of a variable. For example, suppose that trait $T$ has a functional method of one parameter named $n$; then in the code

```
do
  $a{:}T = t$
  $n{:}\mathbb{Z} = 14$
  $z = n\ a$
end
```

it might not be clear whether the intended meaning was to invoke the functional method $n$ on $a$ or to multiply $a$ by $14$. The rules specify that such a situation is a static error.

The rules for reassociating a loose juxtaposition are as follows:

- First the loose juxtaposition is broken into nonempty chunks; wherever there is a non-function element followed by a function element, the latter begins a new chunk. Thus a chunk consists of some number (possibly zero) of functions followed by some number (possibly zero) of non-functions.
- It is a static error if any non-function element in a chunk is an unparenthesized identifier $f$ and is followed by another non-function element whose type is such that $f$ can be applied to that latter element as a functional method.
- The non-functions in each chunk, if any, are replaced by a single element consisting of the non-functions grouped left-associatively into binary juxtapositions.
- What remains in each chunk is then grouped right-associatively.

  (Notice that, up to this point, no analysis of the types of newly constructed chunks is needed during this process.)
- It is a static error if an element of the original juxtaposition was the last element in its chunk before reassociation, the chunk was not the last chunk (and therefore the element in question is a non-function element), the element was an unparenthesized identifier $f$, and the type of the following chunk after reassociation is such that $f$ can be applied to that following chunk as a functional method.
- If any element that remains has type String, then it is a static error if there is any pair of adjacent elements within the juxtaposition such that neither element is of type String.

- What remains is considered to be a binary or multifix application of the juxtaposition operator; if more than two elements remain, the application is handled in the same way as for any other multifix operator (see Section 14.5).

Here is an example: $n\;(n+1)\;\sin\;3\;n\;x\;\log\;\log\;x$. Assuming that $\sin$ and $\log$ name functions in the usual manner and that $n$, $(n+1)$, and $x$ are not functions, this loose juxtaposition splits into three chunks: $n\;(n+1)$, $\sin\;3\;n\;x$, and $\log\;\log\;x$. The first chunk has only two elements and needs no further reassociation. In the second chunk, the non-functions $3\;n\;x$ are replaced by $((3\;n)\;x)$. In the third chunk, there is only one non-function, so that remains unchanged; the chunk is the right-associated to form $(\log\;(\log\;x))$. Assuming that no multifix definition of juxtaposition has been provided, the three chunks are left-associated, to produce the final interpretation $((n\;(n+1))\;(\sin\;((3\;n)\;x)))\;(\log\;(\log\;x))$. Now the original juxtaposition has been reduced to binary juxtaposition expressions.

A tight juxtaposition is always left-associated if it contains any dot (i.e., ".") not within parentheses or some pair of enclosing operators. If such a tight juxtaposition begins with an identifier immediately followed by a dot then the maximal prefix of identifiers separated by dots (whitespace may follow but not precede the dots) is collected into a "dotted id chain", which is subsequently partitioned into the longest prefix, if any, that is a qualified name, and the remaining the dots and identifiers, which are interpreted as selectors. If the last identifier in the dotted id chain is part of a selector (i.e., the entire dotted id chain is not a qualified name), and it is immediately followed by a left parenthesis, then the last selector together with the subsequent parenthesis-delimited expression is a method invocation.

A tight juxtaposition without any dots might not be entirely left-associated. Rather, it is considered as a nonempty sequence of elements: the front expression, any "math items", and any postfix operator, and subject to reassociation as described below. A math item may be a subscripting, an exponentiation, or one of a few kinds of expressions. It is a static error if an exponentiation is immediately followed by a subcripting or an exponentiation.

The procedure for reassociation is as follows:

- For each expression element (i.e., not a subscripting, exponentiation or postfix operator), determine whether it is a function.
- If some function element is immediately followed by an expression element then, find the first such function element, and call the next element the argument. It is a static error if either the argument is not parenthesized, or the argument is immediately followed by a non-expression element. Otherwise, replace the function and argument with a single element that is the application of the function to the argument. This new element is an expression. Reassociate the resulting sequence (which is one element shorter).
- If there is any non-expression element (it cannot be the first element) then replace the first such element and the element immediately preceding it (which must be an expression) with a single element that does the appropriate operator application. This new element is an expression. Reassociate the resulting sequence (which is one element shorter).
- Otherwise, the sequence necessarily has only expression elements, only the last of which may be a function. If any element that remains has type String, then it is a static error if there is any pair of adjacent elements within the juxtaposition such that neither element is of type String.
- What remains is considered to be a binary or multifix application of the juxtaposition operator; if more than two elements remain, the application is handled in the same way as for any other multifix operator (see Section 14.5).

(Note that this process requires type analysis of newly created chunks all along the way.)

Here is an (admittedly contrived) example: $\mathit{reduce}(f)(a)(x+1)\mathit{atan}(x+2)$. Suppose that $\mathit{reduce}$ is a curried function that accepts a function $f$ and returns a function that can be applied to an array $a$ (the idea is to use the function $f$, which ought to take two arguments, to combine the elements of the array to produce an accumulated result).

The leftmost function is $\mathit{reduce}$, and the following element $(f)$ is parenthesized, so the two elements are replaced with one: $(\mathit{reduce}(f))(a)(x+1)\mathit{atan}(x+2)$. Now type analysis determines that the element $(\mathit{reduce}(f))$ is a function.

The leftmost function is $(\mathit{reduce}(f))$, and the following element $(a)$ is parenthesized, so the two elements are replaced with one: $((\mathit{reduce}(f))(a))(x+1)\mathit{atan}(x+2)$. Now type analysis determines that the element $((\mathit{reduce}(f))(a))$ is not a function.

The leftmost function is $atan$, and the following element $(x+2)$ is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)(atan(x+2))$. Now type analysis determines that the element $(atan(x+2))$ is not a function.

There are no functions remaining in the juxtaposition. Assuming that no multifix definition of juxtaposition has been provided, the remaining elements are left-associated:

$$(((reduce(f))(a))(x+1))(atan(x+2))$$

Now the original juxtaposition has been reduced to binary juxtaposition expressions.

## 14.10 Operator Declarations

Multifix operators and keyword parameters are not yet supported.

An operator declaration may appear anywhere a top-level function or method declaration may appear. Operator declarations are like other function or method declarations in all respects except that an operator declaration has `opr` and has an operator name (see Section 14.2 for a discussion of valid operator names) instead of an identifier. The precise placement of the operator name within the declaration depends on the fixity of the operator. Like other functionals, operators may have overloaded declarations (see Chapter **??** for a discussion of overloading). These overloadings may be of the same or differing fixities.

Syntax:

| | | |
|---|---|---|
| *FnDecl* | ::= | *FnMods*? *FnHeaderFront FnHeaderClause* (= *Expr*)? |
| *FnHeaderFront* | ::= | *OpHeaderFront* |
| *OpHeaderFront* | ::= | `opr` `BIG`? ({↦ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params*? (*RightEncloser* \| *Encloser*) |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr` `BIG`? (*Op* \| ˆ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |
| *MdDecl* | ::= | *MdDef* |
| | \| | `abstract`? *MdMods*? *MdHeaderFront FnHeaderClause* |
| *MdHeaderFront* | ::= | *OpMdHeaderFront* |
| *OpMdHeaderFront* | ::= | `opr` `BIG`? ({↦ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params*? (*RightEncloser* \| *Encloser*) (:= ( *SubscriptAssignParam* ))? |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr` `BIG`? (*Op* \| ˆ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |
| *SubscriptAssignParam* | ::= | *Varargs* |
| | \| | *Param* |

An operator declaration has one of eight forms: multifix operator declaration, infix operator declaration, prefix operator declaration, postfix operator declaration, nofix operator declaration, bracketing operator declaration, subscripting operator method declaration, and subscripted assignment operator method declaration. Each is invoked according to specific rules of syntax. An operator method declaration must be a functional method declaration, a subscripting operator method declaration, or a subscripted assignment operator method declaration.

### 14.10.1 Multifix Operator Declarations

A multifix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must not have any keyword parameters, and must be capable of accepting at least two arguments. It is permissible to use a varargs parameter; in fact, this is a good way to define a multifix operator. Static parameters (described in Chapter **??**) may also be present, between the operator and the parameter list.

An expression consisting of a multifix operator applied to an expression will invoke a multifix operator declaration. The compiler considers all multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. The invocation will pass more than two arguments.

A multifix operator declaration may also be invoked by an infix, a prefix, or nofix (but not a postfix) operator application if the declaration is applicable.

Example:

| Example is commented out because it is not supported nor run by the interpreter yet. |
|---|

### 14.10.2 Infix Operator Declarations

An infix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have two value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of an infix operator applied to an expression will invoke an infix operator declaration. The compiler considers all infix and multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Note that superscripting (`^`) may be defined using an infix operator declaration even though it has very high precedence and cannot be used as a multifix operator.

Example:

### 14.10.3 Prefix Operator Declarations

A prefix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of a prefix operator applied to an expression will invoke a prefix operator declaration. The compiler considers all prefix and multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

### 14.10.4 Postfix Operator Declarations

A postfix operator declaration has `opr` where a functional declaration would have an identifier; the operator name itself *follows* the parameter list. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between `opr` and the parameter list.

An expression consisting of a postfix operator applied to an expression will invoke a postfix operator declaration. The compiler considers all postfix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

### 14.10.5 Nofix Operator Declarations

A nofix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have no parameters.

An expression consisting only of a nofix operator will invoke a nofix operator declaration. The compiler considers all nofix and multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Uses for nofix operators are rare, but those rare examples are very useful. For example, if the @ operator is used to construct subscripting ranges, and it is the nofix declaration of @ that allows a lone @ to be used as a subscript:

### 14.10.6 Bracketing Operator Declarations

A bracketing operator declaration has `opr` where a functional declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by the brackets being defined. A bracketing operator declaration may have any number of parameters, keyword parameters, and varargs parameters in the value parameter list. Static parameters may also be present, between `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets.

An expression consisting of zero or more comma-separated expressions surrounded by a bracket pair will invoke a bracketing operator declaration. The compiler considers all bracketing operator declarations for that type of bracket pair that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. For example, the expression $\langle p, q\rangle$ might invoke the following bracketing method declaration:

## 14.11 Overview of Operators in the Fortress Standard Libraries

The operators in this section are not tested nor run by the interpreter yet.

This section provides a high-level overview of the operators in the Fortress standard libraries. See Appendix **??** for the detailed rules for the operators provided by the Fortress standard libraries.

### 14.11.1 Prefix Operators

For all standard numeric types, the prefix operator $+$ simply returns its argument and the prefix operator $-$ returns the negative of its argument.

The operator $\neg$ is the logical NOT operator on boolean values and boolean intervals.

The operator ⫬ computes the bitwise NOT of an integer.

Big operators such as $\sum$ begin a *reduction expression* (Section **??**). The big operators include $\sum$ (summation) and $\prod$ (product), along with $\bigcap$, $\bigcup$, $\bigwedge$, $\bigvee$, $\underline{\bigvee}$, $\bigoplus$, $\bigotimes$, $\biguplus$, $\boxplus$, $\boxtimes$, `MAX`, `MIN`, and so on.

### 14.11.2 Postfix Operators

The operator $!$ computes factorial; the operator $!!$ computes double factorials. They may be applied to a value of any integral type and produces a result of the same type.

When applied to a floating-point value $x$, $x!$ computes $\Gamma(1+x)$, where $\Gamma$ is the Euler gamma function.

### 14.11.3 Enclosing Operators

When used as left and right enclosing operators, $|\quad|$ computes the absolute value or magnitude of any number, and is also used to compute the number of elements in an aggregate, for example the cardinality of a set or the length of a list. Similarly, $\|\quad\|$ is used to compute the norm of a vector or matrix.

The floor operator $\lfloor\quad\rfloor$ and ceiling operator $\lceil\quad\rceil$ may be applied to any standard integer, rational, or real value (their behavior is trivial when applied to integers, of course). The operators hyperfloor $⟦\lfloor x\rfloor⟧ = 2^{\lfloor\log_2 x\rfloor}$, hyperceiling $⟦\lceil x\rceil⟧ = 2^{\lceil\log_2 x\rceil}$, hyperhyperfloor $⦀\lfloor x\rfloor⦀ = 2^{⟦\lfloor\log_2 x\rfloor⟧}$, and hyperhyperceiling $⦀\lceil x\rceil⦀ = 2^{⟦\lceil\log_2 x\rceil⟧}$ are also available.

### 14.11.4 Exponentiation

Given two expressions $e$ and $e'$ denoting numeric quantities $v$ and $v'$ that are not vectors or matrices, the expression $e^{e'}$ denotes the quantity obtained by raising $v$ to the power $v'$. This operation is defined in the usual way on numerals.

Given an expression $e$ denoting a vector and an expression $e'$ denoting a value of type $\mathbb{Z}$, the expression $e^{e'}$ denotes repeated vector multiplication of $e$ by itself $e'$ times.

Given an expression $e$ denoting a square matrix and an expression $e'$ denoting a value of type $\mathbb{Z}$, the expression $e^{e'}$ denotes repeated matrix multiplication of $e$ by itself $e'$ times.

Eric: Do we also want to support the matrix exponential via this operator? Doing so requires having a special constant value for e.

### 14.11.5 Superscript Operators

The superscript operator $\hat{}\,T$ transposes a matrix. It also converts a column vector to a row vector or a row vector to a column vector.

### 14.11.6 Subscript Operators

Subscripting of arrays and other aggregates is written using square brackets:

| | | | |
|---|---|---|---|
| `a[i]` | *is displayed as* | $a_i$ | $i$th element of one-dimensional array $a$ |
| `m[i,j]` | *is displayed as* | $m_{ij}$ | $i,j$th element of two-dimensional matrix $m$ |
| `space[i,j,k]` | *is displayed as* | $\mathit{space}_{ijk}$ | $i,j,k$th element of three-dimensional array $\mathit{space}$ |
| `a[3] := 4` | *is displayed as* | $a_3 := 4$ | assign $4$ to the third element of mutable array $a$ |
| `m["foo"]` | *is displayed as* | $m_{\text{“}\texttt{foo}\text{”}}$ | fetch the entry associated with string “`foo`” from map $m$ |

### 14.11.7 Multiplication, Division, Modulo, and Remainder Operators

For most integer, rational, floating-point, complex, and interval expressions, multiplication can be expressed using any of ‘$\cdot$ ’ or ‘$\times$ ’ or simply juxtaposition. However, the $\times$ operator is used to express the shape of matrices, so if expressions using multiplication are used in expressing the shape of a matrix, it may be necessary to avoid the use of ’$\times$ ’ to express multiplication, or to use parentheses.

For integer, rational, floating-point, complex, and interval expressions, division is expressed by $/$. When the operator $/$ is used to divide one integer by another, the result is rational. The operator $\div$ performs truncating integer division: $m \div n = \mathit{signum}\left(\frac{m}{n}\right) \left\lfloor \left|\frac{m}{n}\right| \right\rfloor$. The operator $\mathtt{REM}$ gives the remainder from such a truncating division: $m\,\mathtt{REM}\,n = m - n(m \div n)$. The operator $\mathtt{MOD}$ gives the remainder from a floor division: $m\,\mathtt{MOD}\,n = m - n\left\lfloor \frac{m}{n} \right\rfloor$; when $n > 0$ this is the usual modulus computation that evaluates integer $m$ to an integer $k$ such that $0 \le k < n$ and $n$ evenly divides $m - k$.

The special operators $\mathtt{DIVREM}$ and $\mathtt{DIVMOD}$ each return a pair of values, the quotient and the remainder; $m\,\mathtt{DIVREM}\,n$ returns $(m \div n, m\,\mathtt{REM}\,n)$ while $m\,\mathtt{DIVMOD}\,n$ returns $(\left\lfloor \frac{m}{n} \right\rfloor, m\,\mathtt{MOD}\,n)$.

Multiplication of a vector or matrix by a scalar is done with juxtaposition, as is multiplication of a vector by a matrix (on either side). Vector dot product is expressed by ‘$\cdot$ ’ and vector cross product by ‘$\times$ ’. Division of a matrix or vector by a scalar may be expressed using ‘$/$ ’.

The syntactic interaction of juxtaposition, $\cdot$, $\times$, and $/$ is subtle. See Section 14.3 for a discussion of the relative precedence of these operations and how precedence may depend on the use of whitespace.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an $\mathrm{IntegerOverflow}$. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

The handling of division by zero depends on the type of the number produced. For integer results, division by zero throws a $\mathrm{DivisionByZero}$. For rational results, division by zero produces $1/0$. For floating-point results, division by zero produces a $\mathrm{NaN}$ value according to the rules of IEEE 754. For intervals, division by zero produces an appropriate containing interval (which under many circumstances will be the interval of all possible real values and infinities).

Wraparound multiplication on fixed-size integers is expressed by $\dot{\times}$. Saturating multiplication on fixed-size integers is expressed by $\boxdot$ or $\boxtimes$. These operations do not overflow.

Ordinary multiplication and division of floating-point numbers always use the IEEE 754 “round to nearest” rounding mode. This rounding mode may be emphasized by using the operators $\otimes$ (or $\odot$) and $\oslash$. Multiplication and division in “round toward zero” mode may be expressed with $\boxtimes$ (or $\boxdot$) and $\boxslash$. Multiplication and division in “round toward positive infinity” mode may be expressed with $\bigtriangleup\!\!\!\!\!\times$ (or $\bigtriangleup\!\!\!\!\!\cdot$) and $\bigtriangleup\!\!\!\!\!/$. Multiplication and division in “round toward negative infinity” mode may be expressed with $\bigtriangledown\!\!\!\!\!\times$ (or $\bigtriangledown\!\!\!\!\!\cdot$) and $\bigtriangledown\!\!\!\!\!/$.

### 14.11.8 Addition and Subtraction Operators

Addition and subtraction are expressed with $+$ and $-$ on all numeric quantities, including intervals, as well as vectors and matrices.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an $\mathrm{IntegerOverflow}$. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

Wraparound addition and subtraction on fixed-size integers are expressed by $\dot{+}$ and $\dot{-}$. Saturating addition and subtraction on fixed-size integers are expressed by $\boxplus$ and $\boxminus$. These operations do not overflow.

Ordinary addition and subtraction of floating-point numbers always use the IEEE 754 "round to nearest" rounding mode. This rounding mode may be emphasized by using the operators $\oplus$ and $\ominus$. Addition and subtraction in "round toward zero" mode may be expressed with $\boxplus$ and $\boxminus$. Addition and subtraction in "round toward positive infinity" mode may be expressed with $⨹$ and $\triangle$. Addition and subtraction in "round toward negative infinity" mode may be expressed with $\bigtriangledown\!\!\!\!+$ and $\bigtriangledown$.

The construction $x \pm y$ produces the interval $\langle\!|x - y, x + y|\!\rangle$.

### 14.11.9 Intersection, Union, and Set Difference Operators

Sets support the operations of intersection $\cap$, union $\cup$, disjoint union $\uplus$, set difference $\setminus$, and symmetric set difference $\ominus$. Disjoint union throws $\mathrm{DisjointUnionError}$ if the arguments are in fact not disjoint.

Intervals support the operations of intersection $\cap$, union $\cup$, and interior hull $\underline{\cup}$. The operation $\Cap$ returns a pair of intervals; if the intersection of the arguments is a single contiguous span of real numbers, then the first result is an interval representing that span and the second result is an empty interval, but if the intersection is two spans, then two (disjoint) intervals are returned. The operation $\Cup$ returns a pair of intervals; if the arguments overlap, then the first result is the union of the two intervals and the second result is an empty interval, but if the arguments are disjoint, they are simply returned as is.

### 14.11.10 Minimum and Maximum Operators

The operator `MAX` returns the larger of its two operands, and `MIN` returns the smaller of its two operands.

For floating-point numbers, if either argument is a $\mathrm{NaN}$ then $\mathrm{NaN}$ is returned. The floating-point operations `MAXNUM` and `MINNUM` behave similarly except that if one argument is $\mathrm{NaN}$ and the other is a number, the number is returned. For all four of these operators, when applied to floating-point values, $-0$ is considered to be smaller than $+0$.

### 14.11.11 GCD, LCM, and CHOOSE Operators

The infix operator `GCD` computes the greatest common divisor of its integer operands, and `LCM` computes the least common multiple. The operator `CHOOSE` computes binomial coefficients: $n \,\mathtt{CHOOSE}\, k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The expression $e \neq e'$ is semantically equivalent to the expression $\neg(e = e')$.

### 14.11.12 Comparisons Operators

Unless otherwise noted, the operators described in this section produce boolean (*true*/*false*) results.

The operators $<$, $\leq$, $\geq$, and $>$ are used for numerical comparisons and are supported by integer, rational, and floating-point types. Comparison of rational values throws $\mathrm{RationalComparisonError}$ if either argument is the rational infinity $1/0$ or the rational indefinite $0/0$. Comparison of floating-point values throws $\mathrm{FloatingComparisonError}$ if either argument is a $\mathrm{NaN}$.

The operators $<$, $\leq$, $\geq$, and $>$ may also be used to compare characters (according to the numerical value of their Unicode codepoint values) and strings (lexicographic order based on the character ordering). They also use lexicographic order when used to compare lists whose elements support these same comparison operators.

When $<$, $\leq$, $\geq$, and $>$ are used to compare numerical intervals, the result is a boolean interval. The functions *possibly* and *certainly* are useful for converting boolean intervals to boolean values for testing. Thus $possibly(x > y)$ is true if and only if there is some value in the interval $x$ that is greater than some value in the interval $y$, while $certainly(x > y)$ is true if and only if $x$ and $y$ are nonempty and every value in $x$ is greater than every value in $y$.

The operators $\subset$, $\subseteq$, $\supseteq$, and $\supset$ may be used to compare sets or intervals regarded as sets.

The operator $\in$ may be used to test whether a value is a member of a set, list, array, interval, or range.

### 14.11.13 Logical Operators

The following binary operators may be used on boolean values:

| | |
|---|---|
| $\wedge$ | AND |
| $\vee$ | inclusive OR |
| $\veebar$ or $\oplus$ or $\neq$ | exclusive OR |
| $\equiv$ or $\leftrightarrow$ or $=$ | equivalence (if and only if) |
| $\rightarrow$ | IMPLIES |
| $\overline{\wedge}$ | NAND |
| $\overline{\vee}$ | NOR |

These same operators may also be applied to boolean intervals to produce boolean interval results.

The following operators may be used on integers to perform "bitwise" operations:

| | |
|---|---|
| $\bigwedge\!\!\!\!\wedge$ | bitwise AND |
| $\bigvee\!\!\!\!\vee$ | bitwise inclusive OR |
| $\underline{\bigvee\!\!\!\!\vee}$ | bitwise exclusive OR |

The prefix operator $\urcorner$ computes the bitwise NOT of an integer.

### 14.11.14 Conditional Operators

If $p(x)$ and $q(x)$ are expressions that produce boolean results, the expression $p(x) \wedge\!: q(x)$ computes the logical AND of those two results by first evaluating $p(x)$. If the result of $p(x)$ is $true$, then $q(x)$ is also evaluated, and its result becomes the result of the entire expression; but if the result of $p(x)$ is $false$, then $q(x)$ is not evaluated, and the result of the entire expression is $false$ without further ado. (Similarly, evaluating the expression $p(x) \vee\!: q(x)$ does not evaluate $q(x)$ if the result of $p(x)$ is $true$.) Contrast this with the expression $p(x) \wedge q(x)$ (with no colon), which evaluates both $p(x)$ and $q(x)$, in no specified order and possibly in parallel.

# Chapter 15

# Overloading and Dispatch

Keyword and varargs parameters are not yet supported.

Fortress allows functions and methods (collectively called *procedures*) to be *overloaded*. That is, there may be multiple declarations for the same procedure name visible in a single scope (which may include inherited method declarations), and several of them may be applicable to any particular procedure call. This raises the question of which definition will actually be executed for any given procedure invocation. The simple answer is that the *dynamically most specific applicable visible* definition is chosen. That is, one first considers the set of all definitions that are *dynamically visible*; then, among those in the set that are *applicable* to the given argument values, the *most specific* one is chosen.

At compile time, typechecking performs a related series of tests: the type of a procedure call is the return type of the *statically most specific applicable visible* declaration. That is, one first considers the set of all definitions that are *statically visible*; then, among those in the set that are *applicable* to the types of the given arguments, the *most specific* one is chosen.

`trait` $T$ `extends` $\{A, B\}$ `end`

In this chapter, we describe how to determine which declarations are *visible* to a particular procedure call (boths statically and dynamically); how to further determine which of these are *applicable* to that procedure call (boths statically and dynamically); and how the most specific one is chosen. We also introduce some rules for writing procedure declarations that ensure the soundness of the type system, as well as the existence and uniqueness of a most specific applicable declaration (at compile time) or definition (at run time). The result is that if a program successfully compiles, then procedure calls always succeed and are never ambiguous.

Section 15.1 introduces some terminology and notation. In Section 15.3, we show how to determine which declarations are applicable to a *named procedure call* (a function call described in Section 12.4 or a naked method invocation described in Section **??**) when all declarations have only ordinary parameters (without varargs or keyword parameters). We discuss how to handle dotted method calls (described in Section **??**) in Section 15.4, and declarations with varargs and keyword parameters in Section 15.5. Determining which declaration is applied, if several are applicable, is discussed in Section 15.6.

## 15.1 Principles of Overloading

Fortress allows multiple procedure declarations of the same name to be declared in a single scope. However, recall from Chapter **??** the following shadowing rules:

- dotted method declarations shadow top-level function declarations with the same name, and
- dotted method declarations provided by a trait or object declaration or object expression shadow functional method declarations with the same name that are provided by a different trait or object declaration or object expression.

Also, note that a trait or object declaration or object expression must not have a functional method declaration and a dotted method declaration with the same name, either directly or by inheritance. Therefore, top-level functions can overload with other top-level functions and functional methods, dotted methods with other dotted methods, and functional methods with other functional methods and top-level functions. It is a static error if any top-level function declaration is more specific than any functional method declaration. If a top-level function declaration is overloaded with a functional method declaration, the top-level function declaration must not be more specific than the functional method declaration.

Operator declarations with the same name but different fixity are not a valid overloading; they are unambiguous declarations. An operator method declaration whose name is one of the operator parameters (described in Section **??**) of its enclosing trait or object may be overloaded with other operator declarations in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Chapter **??**) with every operator declaration in the same component.

This restriction will be relaxed.

Recall from Chapter 5 that we write $T \preceq U$ when $T$ is a subtype of $U$, and $T \prec U$ when $T \preceq U$ and $U \not\preceq T$.

## 15.2 Visibility to Named Procedure Calls

## 15.3 Applicability to Named Procedure Calls

In this section, we show how to determine which declarations are applicable to a named procedure call when all declarations have only ordinary parameters (i.e., neither varargs nor keyword parameters).

For the purpose of defining applicability, a named procedure call can be characterized by the name of the procedure and its argument type. Recall that a procedure has a single parameter, which may be a tuple (a dotted method has a receiver as well). We abuse notation by using *static call* $f(A)$ to refer to a named procedure call with name $f$ and whose argument has static type $A$, and *dynamic call* $f(X)$ to refer to a named procedure call with We use *call* $f(C)$ to refer to a named procedure call with name $f$ and whose argument, when evaluated, has dynamic type $C$. (Note that if the type system is sound—and we certainly hope that it is!—then $X \preceq A$ for all well-typed calls to $f$.) We use the term *call* $f(C)$ to refer to static and dynamic calls collectively. We assume throughout this chapter that all static variables in procedure calls have been instantiated or inferred.

We also use *function declaration* $f(P) : U$ to refer to a function declaration with function name $f$, parameter type $P$, and return type $U$.

For method declarations, we must take into account the self parameter, as follows:

A *dotted method declaration* $P_0.f(P) : U$ is a dotted method declaration with name $f$, where $P_0$ is the trait or object type in which the declaration appears, $P$ is the parameter type, and $U$ is the return type. (Note that despite the suggestive notation, a dotted method declaration does not explicitly list its self parameter.)

A *functional method declaration* $f(P) : U$ *with self parameter at* $i$ is a functional method declaration with name $f$, with the parameter `self` in the $i$th position of the parameter type $P$, and return type $U$. Note that the static type of the self parameter is the trait or object trait type in which the declaration $f(P) : U$ occurs. In the following, we will use $P_i$ to refer to the $i$th element of $P$.

We elide the return type of a declaration, writing $f(P)$ and $P_0.f(P)$, when the return type is not relevant to the discussion. Note that static parameters may appear in the types $P_0$, $P$, and $U$.

A declaration $f(P)$ is *applicable* to a call $f(C)$ if the call is in the scope of the declaration and $C \preceq P$. (See Chapter **??** for the definition of scope.) If the parameter type $P$ includes static parameters, they are inferred as described in Chapter 17 before checking the applicability of the declaration to the call.

Note that a named procedure call $f(C)$ may invoke a dotted method declaration if the declaration is provided by the trait or object enclosing the call. To account for this, let $C_0$ be the trait or object declaration immediately enclosing the call. Then we consider a named procedure call $f(C)$ as $C_0.f(C)$ if $C_0$ provides dotted method declarations applicable to $f(C)$, and use the rule for applicability to dotted method calls (described in Section 15.4) to determine which declarations are applicable to $C_0.f(C)$.

## 15.4 Applicability to Dotted Method Calls

Dotted method applications can be characterized similarly to named procedure applications, except that, analogously to dotted method declarations, we use $C_0$ to denote the dynamic type of the receiver object, and, as for named procedure calls, $C$ to denote the dynamic type of the argument of a dotted method call. We write $C_0.f(C)$ to refer to the call.

A dotted method declaration $P_0.f(P)$ is *applicable* to a dotted method call $C_0.f(C)$ if $C_0 \preceq P_0$ and $C \preceq P$. If the types $P_0$ and $P$ include static parameters, they are inferred as described in Chapter 17 before checking the applicability of the declaration to the call.

## 15.5 Applicability for Procedures with Varargs and Keyword Parameters

The basic idea for handling varargs and keyword parameters is that we can think of a procedure declaration that has such parameters as though it were (possibly infinitely) many declarations, one for each set of arguments it may be called with. In other words, we expand these declarations so that there exists a declaration for each number of arguments that can be passed to it.

A declaration with a varargs parameter corresponds to an infinite number of declarations, one for every number of arguments that may be passed to the varargs parameter. In practice, we can bound that number by the maximum number of arguments that the procedure is called with anywhere in the program (in other words, a given program will contain only a finite number of calls with different numbers of arguments). The expansion described here is a conceptual one to simplify the description of the semantics; we do not expect a real implementation to actually expand these declarations at compile time. For example, the following declaration:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) : \mathbb{Z}$

would be expanded into:

$f(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}, z_3 : \mathbb{Z}) : \mathbb{Z}$
$\ldots$

A declaration with a varargs parameter is applicable to a call if any one of the expanded declarations is applicable.

## 15.6 Overloading Resolution

Victor: Does this paragraph, other than the last sentence, really belong in this section?

To evaluate a given procedure call, it is necessary to determine which procedure declaration to dispatch to. To do so, we consider the declarations that are applicable to that call at run time. If there is exactly one such declaration, then the call dispatches to that declaration. If there is no such declaration, then the call is *undefined*, which is a static error. (However, see Section **??** for how coercion may add to the set of applicable declarations.) If multiple declarations are applicable to the call at run time, then we choose an arbitrary declaration among the declarations such that no other applicable declaration is more specific than them.

We use the subtype relation to compare parameter types to determine a more specific declaration. Formally, a declaration $f(P)$ is *more specific* than a declaration $f(Q)$ if $P \prec Q$. Similarly, a declaration $P_0.f(P)$ is more specific than a declaration $Q_0.f(Q)$ if $P_0 \prec Q_0$ and $P \prec Q$. (See Section **??** for how coercion changes the definition of "more specific".) Restrictions on the definition of overloaded procedures (see Chapter **??**) guarantee that among all applicable declarations, one is more specific than all the others. If the declarations include static parameters, they are inferred as described in Chapter 17 before comparing their parameter types to determine which declaration is more specific.

# Chapter 16

# Coercion

# Chapter 17

# Type Inference

# Chapter 18

# Components and APIs

# Appendix A

# Simplified Grammar for Application Programmers and Library Writers

## A.1 Components and API

```
File                  ::=  CompilationUnit
                       |   Imports? Exports Decls?
                       |   Imports? AbsDecls
                       |   Imports AbsDecls?
CompilationUnit       ::=  Component
                       |   Api
Component             ::=  component DottedId Imports? Exports Decls? end
Api                   ::=  api DottedId Imports? AbsDecls? end
Imports               ::=  Import+
Import                ::=  import ImportFrom
                       |   import AliasedDottedIds
ImportFrom            ::=  * ( except Names )? from DottedId
                       |   AliasedNames from DottedId
Names                 ::=  Name
                       |   { NameList }
NameList              ::=  Name ( , Name )*
AliasedNames          ::=  AliasedName
                       |   { AliasedNameList }
AliasedName           ::=  Id ( as DottedId )?
                       |   opr Op ( as Op )?
                       |   opr LeftEncloser RightEncloser ( as LeftEncloser RightEncloser )?
AliasedNameList       ::=  AliasedName ( , AliasedName )*
AliasedDottedIds      ::=  AliasedDottedId
                       |   { AliasedDottedIdList }
AliasedDottedId       ::=  DottedId ( as DottedId )?
AliasedDottedIdList   ::=  AliasedDottedId ( , AliasedDottedId )*
Exports               ::=  Export+
Export                ::=  export DottedIds
DottedIds             ::=  DottedId
```

| | | |
|---|---|---|
| | \| | { DottedIdList } |
| DottedIdList | ::= | DottedId ( , DottedId )$^*$ |

## A.2 Top-level Declarations

| | | |
|---|---|---|
| Decls | ::= | Decl$^+$ |
| Decl | ::= | TraitDecl |
| | \| | ObjectDecl |
| | \| | VarDecl |
| | \| | FnDecl |
| AbsDecls | ::= | AbsDecl$^+$ |
| AbsDecl | ::= | AbsTraitDecl |
| | \| | AbsObjectDecl |
| | \| | AbsVarDecl |
| | \| | AbsFnDecl |

## A.3 Trait Declaration

| | | |
|---|---|---|
| TraitDecl | ::= | TraitHeader GoInATrait$^?$ `end` |
| TraitHeader | ::= | TraitMods$^?$ `trait` Id StaticParams$^?$ Extends$^?$ TraitClauses$^?$ |
| TraitClauses | ::= | TraitClause$^+$ |
| TraitClause | ::= | Excludes |
| | \| | Comprises |
| GoInATrait | ::= | GoFrontInATrait GoBackInATrait$^?$ |
| | \| | GoBackInATrait |
| GoFrontInATrait | ::= | GoesFrontInATrait$^+$ |
| GoesFrontInATrait | ::= | AbsFldDecl |
| | \| | GetterSetterDecl |
| GoBackInATrait | ::= | GoesBackInATrait$^+$ |
| GoesBackInATrait | ::= | MdDecl |
| AbsTraitDecl | ::= | TraitHeader AbsGoInATrait$^?$ `end` |
| AbsGoInATrait | ::= | AbsGoFrontInATrait AbsGoBackInATrait$^?$ |
| | \| | AbsGoBackInATrait |
| AbsGoFrontInATrait | ::= | AbsGoesFrontInATrait$^+$ |
| AbsGoesFrontInATrait | ::= | ApiFldDecl |
| | \| | AbsGetterSetterDecl |
| AbsGoBackInATrait | ::= | AbsGoesBackInATrait$^+$ |
| AbsGoesBackInATrait | ::= | AbsMdDecl |
| | \| | AbsCoercion |

## A.4 Object declaration

| | | |
|---|---|---|
| ObjectDecl | ::= | ObjectHeader GoInAnObject$^?$ `end` |
| ObjectHeader | ::= | ObjectMods$^?$ `object` Id StaticParams$^?$ ObjectValParam$^?$ Extends$^?$ FnClauses |
| ObjectValParam | ::= | ( ObjectParams$^?$ ) |

|  |  |  |
|---|---|---|
| ObjectParams | ::= | ( ObjectParam , )$^{*}$ ObjectKeyword ( , ObjectKeyword )$^{*}$ |
|  | \| | ObjectParam ( , ObjectParam )$^{*}$ |
| ObjectKeyword | ::= | ObjectParam = Expr |
| ObjectParam | ::= | FldMods$^{?}$ Param |
|  | \| | `transient` Param |
| GoInAnObject | ::= | GoFrontInAnObject GoBackInAnObject$^{?}$ |
|  | \| | GoBackInAnObject |
| GoFrontInAnObject | ::= | GoesFrontInAnObject$^{+}$ |
| GoesFrontInAnObject | ::= | FldDecl |
|  | \| | GetterSetterDef |
| GoBackInAnObject | ::= | GoesBackInAnObject$^{+}$ |
| GoesBackInAnObject | ::= | MdDef |
| AbsObjectDecl | ::= | ObjectHeader AbsGoInAnObject$^{?}$ `end` |
| AbsGoInAnObject | ::= | AbsGoFrontInAnObject AbsGoBackInAnObject$^{?}$ |
|  | \| | AbsGoBackInAnObject |
| AbsGoFrontInAnObject | ::= | AbsGoesFrontInAnObject$^{+}$ |
| AbsGoesFrontInAnObject | ::= | ApiFldDecl |
|  | \| | AbsGetterSetterDecl |
| AbsGoBackInAnObject | ::= | AbsGoesBackInAnObject$^{+}$ |
| AbsGoesBackInAnObject | ::= | AbsMdDecl |
|  | \| | AbsCoercion |

## A.5 Variable Declaration

|  |  |  |
|---|---|---|
| VarDecl | ::= | VarWTypes InitVal |
|  | \| | VarWoTypes = Expr |
|  | \| | VarWoTypes : TypeRef ... InitVal |
|  | \| | VarWoTypes : SimpleTupleType InitVal |
| VarWTypes | ::= | VarWType |
|  | \| | ( VarWType ( , VarWType )$^{+}$ ) |
| VarWType | ::= | VarMods$^{?}$ Id IsType |
| VarWoTypes | ::= | VarWoType |
|  | \| | ( VarWoType ( , VarWoType )$^{+}$ ) |
| VarWoType | ::= | VarMods$^{?}$ Id |
| InitVal | ::= | ( = \| := ) Expr |
| AbsVarDecl | ::= | VarWTypes |
|  | \| | VarWoTypes : TypeRef ... |
|  | \| | VarWoTypes : SimpleTupleType |

## A.6 Function Declaration

|  |  |  |
|---|---|---|
| FnDecl | ::= | FnDef |
|  | \| | AbsFnDecl |
| FnDef | ::= | FnMods$^{?}$ FnHeaderFront FnHeaderClause = Expr |
| AbsFnDecl | ::= | FnMods$^{?}$ FnHeaderFront FnHeaderClause |
|  | \| | Name : ArrowType |

| | | |
|---|---|---|
| FnHeaderFront | ::= | Id StaticParams$^?$ ValParam |
| | \| | OpHeaderFront |

## A.7 Headers

| | | |
|---|---|---|
| Extends | ::= | `extends` TraitTypes |
| Excludes | ::= | `excludes` TraitTypes |
| Comprises | ::= | `comprises` ComprisingTypes |
| TraitTypes | ::= | TraitType |
| | \| | { TraitTypeList } |
| TraitTypeList | ::= | TraitType ( , TraitType )$^*$ |
| ComprisingTypes | ::= | TraitType |
| | \| | { ComprisingTypeList } |
| ComprisingTypeList | ::= | . . . |
| | \| | TraitType ( , TraitType )$^*$ ( , . . . )$^?$ |
| FnHeaderClause | ::= | IsType$^?$ FnClauses |
| FnClauses | ::= | Throws$^?$ |
| Throws | ::= | `throws` MayTraitTypes |
| MayTraitTypes | ::= | { } |
| | \| | TraitTypes |
| CoercionClauses | ::= | Throws$^?$ |
| UniversalMod | ::= | `private` |
| TraitMod | ::= | `value` |
| | \| | UniversalMod |
| TraitMods | ::= | TraitMod$^+$ |
| ObjectMods | ::= | TraitMods |
| FnMod | ::= | LocalFnMod |
| | \| | UniversalMod |
| FnMods | ::= | FnMod$^+$ |
| VarMod | ::= | `var` |
| | \| | UniversalMod |
| VarMods | ::= | VarMod$^+$ |
| AbsFldMod | ::= | `hidden` \| `settable` \| UniversalMod |
| AbsFldMods | ::= | AbsFldMod$^+$ |
| FldMod | ::= | `var` |
| | \| | AbsFldMod |
| FldMods | ::= | FldMod$^+$ |
| ApiFldMod | ::= | `hidden` \| `settable` \| UniversalMod |
| ApiFldMods | ::= | ApiFldMod$^+$ |
| LocalFnMod | ::= | `atomic` |
| LocalFnMods | ::= | LocalFnMod$^+$ |
| StaticParams | ::= | ⟦ StaticParamList ⟧ |
| StaticParamList | ::= | StaticParam ( , StaticParam )$^*$ |
| StaticParam | ::= | Id Extends$^?$ |
| | \| | `nat` Id |
| | \| | `int` Id |

```
                        |  bool Id
                        |  opr Op
                        |  ident Id
```

## A.8 Parameters

```
ValParam             ::=  BindId
                      |   ( Params? )
Params               ::=  ( Param , )* Keyword ( , Keyword )*
                      |   ( Param , )*
                      |   Param ( , Param )*
Keyword              ::=  Param = Expr
PlainParam           ::=  BindId IsType?
                      |   TypeRef
Param                ::=  PlainParam
OpHeaderFront        ::=  opr StaticParams? ( LeftEncloser | Encloser ) Params ( RightEncloser | Encloser ) ( :=
                      |   opr StaticParams? ValParam Op
                      |   opr ( Op | Encloser ) StaticParams? ValParam
SubscriptAssignParam ::=  Param
```

## A.9 Method Declaration

```
MdDecl               ::=  MdDef
                      |   AbsMdDecl
MdDef                ::=  FnMods? MdHeaderFront FnHeaderClause = Expr
                      |   Coercion
AbsMdDecl            ::=  abstract ? FnMods? MdHeaderFront FnHeaderClause
MdHeaderFront        ::=  ( Receiver . )? Id StaticParams? MdValParam
                      |   OpHeaderFront
Receiver             ::=  Id
                      |   self
GetterSetterDecl     ::=  GetterSetterDef
                      |   AbsGetterSetterDecl
GetterSetterDef      ::=  FnMods? GetterSetterMod MdHeaderFront FnHeaderClause = Expr
GetterSetterMod      ::=  getter | setter
AbsGetterSetterDecl  ::=  abstract ? FnMods? GetterSetterMod MdHeaderFront FnHeaderClause
Coercion             ::=  widening? coercion StaticParams? ( Id IsType ) CoercionClauses = Expr
AbsCoercion          ::=  widening? coercion StaticParams? ( Id IsType ) CoercionClauses
```

## A.10 Method Parameters

```
MdValParam           ::=  ( MdParams? )
MdParams             ::=  ( MdParam , )* MdKeyword ( , MdKeyword )*
                      |   ( MdParam , )*
                      |   MdParam ( , MdParam )*
```

| | | |
|---|---|---|
| MdKeyword | ::= | MdParam = Expr |
| MdParam | ::= | Param |
| | \| | `self` |

## A.11 Field Declarations

| | | |
|---|---|---|
| FldDecl | ::= | FldWTypes InitVal |
| | \| | FldWoTypes = Expr |
| | \| | FldWoTypes : TypeRef ... InitVal |
| | \| | FldWoTypes : SimpleTupleType InitVal |
| FldWTypes | ::= | FldWType |
| | \| | ( FldWType ( , FldWType )$^{+}$ ) |
| FldWType | ::= | FldMods$^{?}$ Id IsType |
| FldWoTypes | ::= | FldWoType |
| | \| | ( FldWoType ( , FldWoType )$^{+}$ ) |
| FldWoType | ::= | FldMods$^{?}$ Id |

## A.12 Abstract Filed Declaration

| | | |
|---|---|---|
| AbsFldDecl | ::= | AbsFldWTypes |
| | \| | AbsFldWoTypes : TypeRef ... |
| | \| | AbsFldWoTypes : SimpleTupleType |
| AbsFldWTypes | ::= | AbsFldWType |
| | \| | ( AbsFldWType ( , AbsFldWType )$^{+}$ ) |
| AbsFldWType | ::= | AbsFldMods$^{?}$ Id IsType |
| AbsFldWoTypes | ::= | AbsFldWoType |
| | \| | ( AbsFldWoType ( , AbsFldWoType )$^{+}$ ) |
| AbsFldWoType | ::= | AbsFldMods$^{?}$ Id |
| ApiFldDecl | ::= | ApiFldMods$^{?}$ Id IsType |

## A.13 Expressions

| | | |
|---|---|---|
| Expr | ::= | AssignLefts AssignOp Expr |
| | \| | OpExpr |
| | \| | DelimitedExpr |
| | \| | FlowExpr |
| | \| | `fn` ValParam IsType$^{?}$ Throws$^{?}$ $\Rightarrow$ Expr |
| | \| | Expr *as* TypeRef |
| | \| | Expr `asif` TypeRef |
| AssignLefts | ::= | [(] AssignLeft ( , AssignLeft )$^{*}$ ) |
| | \| | AssignLeft |
| AssignLeft | ::= | SubscriptExpr |
| | \| | FieldSelection |
| | \| | BindId |
| SubscriptExpr | ::= | Primary [ ExprList$^{?}$ ] |

```
FieldSelection     ::=  Primary . Id
OpExpr             ::=  EncloserOp OpExpr? EncloserOp?
                     |  OpExpr EncloserOp OpExpr?
                     |  Primary
EncloserOp         ::=  Encloser
                     |  Op
Primary            ::=  Comprehension
                     |  Id ⟦ StaticArgList ⟧
                     |  BaseExpr
                     |  LeftEncloser ExprList? RightEncloser
                     |  Primary [ ExprList? ]
                     |  Primary . Id ( ⟦ StaticArgList ⟧ )? TupleExpr
                     |  Primary . Id ( ⟦ StaticArgList ⟧ )? ( )
                     |  Primary . Id
                     |  Primary ^ BaseExpr
                     |  Primary ExponentOp
                     |  Primary TupleExpr
                     |  Primary ( )
                     |  Primary Primary
                     |  TraitType . coercion ( ⟦ StaticArgList ⟧ )? ( Expr )
FlowExpr           ::=  exit Id? ( with Expr )?
                     |  Accumulator ( [ GeneratorList ] )? Expr
                     |  atomic AtomicBack
                     |  tryatomic AtomicBack
                     |  throw Expr
AtomicBack         ::=  AssignLefts AssignOp Expr
                     |  OpExpr
                     |  DelimitedExpr
GeneratorList      ::=  Generator ( , Generator )*
Generator          ::=  GeneratorBinding
                     |  Expr
GeneratorBinding   ::=  Id ← Expr
                     |  ( Id , IdList ) ← Expr
```

## A.14 Expressions Enclosed by Keywords

```
DelimitedExpr      ::=  TupleExpr
                     |  ObjectExpr
                     |  DoExpr
                     |  LabelExpr
                     |  WhileExpr
                     |  ForExpr
                     |  IfExpr
                     |  CaseExpr
                     |  TypecaseExpr
                     |  TryExpr
TupleExpr          ::=  ( ( Expr , )* ( Expr ... , )? Binding ( , Binding )* )
                     |  NoKeyTuple
NoKeyTuple         ::=  ( ( Expr , )* Expr ... )
```

| | | |
|---|---|---|
| | \| | ( ( Expr , )$^*$ Expr ) |
| ObjectExpr | ::= | object Extends$^?$ GoInAnObject end |
| DoExpr | ::= | ( DoFront also )$^*$ DoFront end |
| DoFront | ::= | ( at Expr )$^?$ atomic$^?$ do Block$^?$ |
| LabelExpr | ::= | label Id Block end Id |
| WhileExpr | ::= | while Generator DoExpr |
| ForExpr | ::= | for GeneratorList DoFront end |
| IfExpr | ::= | if Generator then Block ElifClause$^*$ ElseClause$^?$ end |
| | \| | [(] if Generator then Block ElifClause$^*$ ElseClause end$^?$ [)] |
| ElifClause | ::= | elif Expr then Block |
| ElseClause | ::= | else Block |
| CaseExpr | ::= | case Expr Op$^?$ of CaseClauses CaseElseClause$^?$ end |
| CaseClauses | ::= | CaseClause$^+$ |
| CaseClause | ::= | Expr ⇒ Block |
| CaseElseClause | ::= | else ⇒ Block |
| TypecaseExpr | ::= | typecase TypecaseBindings of TypecaseClauses CaseElseClause$^?$ end |
| TypecaseBindings | ::= | ( BindingList ) |
| | \| | Binding |
| | \| | Id |
| BindingList | ::= | Binding ( , Binding )$^*$ |
| Binding | ::= | BindId = Expr |
| TypecaseClauses | ::= | TypecaseClause$^+$ |
| TypecaseClause | ::= | TypecaseTypeRefs ⇒ Block |
| TypecaseTypeRefs | ::= | ( TypeRefList ) |
| | \| | TypeRef |
| TryExpr | ::= | try Block Catch$^?$ ( finally Block )$^?$ end |
| Catch | ::= | catch Id CatchClauses |
| CatchClauses | ::= | CatchClause$^+$ |
| CatchClause | ::= | TraitType ⇒ Block |
| Comprehension | ::= | { Expr \| GeneratorList } |
| | \| | { Entry \| GeneratorList } |
| | \| | < Expr \| GeneratorList > |
| | \| | [ ArrayComprehensionClause$^+$ ] |
| Entry | ::= | Expr ↦ Expr |
| ArrayComprehensionLeft | ::= | IdOrInt ↦ Expr |
| | \| | ( IdOrInt , IdOrIntList ) ↦ Expr |
| ArrayComprehensionClause | ::= | ArrayComprehensionLeft |
| | \| | GeneratorList |
| IdOrInt | ::= | Id |
| | \| | IntLiteral |
| IdOrIntList | ::= | IdOrInt ( , IdOrInt )$^*$ |
| BaseExpr | ::= | NoKeyTuple |
| | \| | Literal |
| | \| | Id |
| | \| | self |
| ExprList | ::= | Expr ( , Expr )$^*$ |

## A.15 Local Declarations

| | | |
|---|---|---|
| Block | ::= | BlockElement$^{+}$ |
| BlockElement | ::= | LocalVarFnDecl |
| | \| | Expr ( , GeneratorList )$^{?}$ |
| LocalVarFnDecl | ::= | LocalFnDecl$^{+}$ |
| | \| | LocalVarDecl |
| LocalFnDecl | ::= | LocalFnMods$^{?}$ FnHeaderFront FnHeaderClause = Expr |
| LocalVarDecl | ::= | LocalVarWTypes InitVal |
| | \| | LocalVarWTypes |
| | \| | LocalVarWoTypes = Expr |
| | \| | LocalVarWoTypes : TypeRef ... InitVal$^{?}$ |
| | \| | LocalVarWoTypes : SimpleTupleType InitVal$^{?}$ |
| LocalVarWTypes | ::= | LocalVarWType |
| | \| | ( LocalVarWType ( , LocalVarWType )$^{+}$ ) |
| LocalVarWType | ::= | `var` ? Id IsType |
| LocalVarWoTypes | ::= | LocalVarWoType |
| | \| | ( LocalVarWoType ( , LocalVarWoType )$^{+}$ ) |
| LocalVarWoType | ::= | `var` ? Id |

## A.16 Literals

| | | |
|---|---|---|
| Literal | ::= | ( ) |
| | \| | BooleanLiteral |
| | \| | CharacterLiteral |
| | \| | StringLiteral |
| | \| | NumericLiteral |
| RectElements | ::= | Expr MultiDimCons$^{*}$ |
| MultiDimCons | ::= | RectSeparator Expr |
| RectSeparator | ::= | ; + |
| | \| | Whitespace |

## A.17 Types

| | | |
|---|---|---|
| IsType | ::= | : TypeRef |
| TypeRef | ::= | TraitType |
| | \| | ArrowType |
| | \| | TupleType |
| | \| | ( TypeRef ? ) |
| TraitType | ::= | DottedId ( $\llbracket$ StaticArgList $\rrbracket$ )$^{?}$ |
| | \| | { TypeRef $\mapsto$ TypeRef } |
| | \| | $<$ TypeRef $>$ |
| | \| | TypeRef [ ArraySize$^{?}$ ] |
| | \| | TypeRef ˆ IntLiteral |
| | \| | TypeRef ˆ ( ExtentRange ( $\times$ ExtentRange )$^{*}$ ) |
| ArrowType | ::= | TypeRef $\rightarrow$ TypeRef Throws$^{?}$ |

| | | |
|---|---|---|
| TupleType | ::= | ( ( TypeRef , $)^*$ ( TypeRef ... , $)^?$ KeywordType ( , KeywordType $)^*$ ) |
| | $\mid$ | ( ( TypeRef , $)^*$ TypeRef ... ) |
| | $\mid$ | SimpleTupleType |
| KeywordType | ::= | Id = TypeRef |
| SimpleTupleType | ::= | ( TypeRef , TypeRefList ) |
| TypeRefList | ::= | TypeRef ( , TypeRef $)^*$ |
| StaticArgList | ::= | StaticArg ( , StaticArg $)^*$ |
| StaticArg | ::= | Op |
| | $\mid$ | TypeRef |
| | $\mid$ | ( StaticArg ) |
| ArraySize | ::= | ExtentRange ( , ExtentRange $)^*$ |
| ExtentRange | ::= | StaticArg$^?$ # StaticArg$^?$ |
| | $\mid$ | StaticArg$^?$ : StaticArg$^?$ |
| | $\mid$ | StaticArg |

## A.18 Symbols and Operators

| | | |
|---|---|---|
| AssignOp | ::= | := |
| | $\mid$ | Op = |
| Accumulator | ::= | $\sum$ $\mid$ $\prod$ $\mid$ `BIG` Op |

## A.19 Identifiers

| | | |
|---|---|---|
| IdList | ::= | Id ( , Id $)^*$ |
| Name | ::= | Id |
| | $\mid$ | `opr` Op |
| DottedId | ::= | Id ( . Id $)^*$ |
| BindId | ::= | Id |
| | $\mid$ | _ |

# Bibliography

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.