The Fortress Language Specification

May 31, 2012

Contents

Chapter 1

Introduction

Chapter 2

Overview of Fortress

Chapter 3

Getting Started

Chapter 4

Programs

The following sentences really belong in a chapter describing this document rather than this chapter.

This document stipulates *static constraints* that must be satisfied by a *valid* program, and specifies how a valid program *executes*. Violation of a static constraint is called a *static error*.

Sometimes we say that a program must not have some property. This is equivalent to saying that it is a static error if it has that property.

Chapter 5

Lexical Structure

A program is of a finite sequence of Unicode 5.0 abstract characters. This sequence is partitioned into *input elements*, as described in this chapter.

To aid in program entry and facilitate interaction with legacy tools, Fortress specifies an ASCII encoding for programs, described in Section 5.8. This encoding is deciphered by an idempotent *preprocessing step*, described in Appendix A, which occurs before any other processing is done Outside of Section 5.8 and Appendix A, all constraints and properties of Fortress programs stipulated in this specification apply to the programs after preprocessing unless otherwise specified.

Fortress also specifies standard ways to *render* (i.e., display) input elements in order to approximate conventional mathematical notation. These are described in Section **??**.

5.1 Characters

Unicode 5.0 specifies a representation for each character as a sequence of *code points*.[1] Fortress programs must not contain characters that map to multiple code points or to sequences of code points of length greater than 1. Thus, every character in a valid program is associated with a single code point, designated by a hexadecimal numeral preceded by "U+". Unicode 5.0 assigns each such character a unique name and a general category.

We designate certain sets of characters as follows:

- *letters*: characters in Unicode general category Lu, Ll, Lt, Lm or Lo, and ∞, $\top$ and $\bot$ (U+221E, U+22A4–5);
- *uppercase letters*: characters in Unicode general category Lu;
- *digits*: characters in Unicode general category Nd;
- *prime characters*: ′ ″ ‴ (i.e., U+2032–4);
- *word characters*: letters, digits, prime characters, and ' and _ (i.e., apostrophe and underscore);
- *operator characters*, which are listed in Appendix **??**;
- *spaces*: characters in Unicode general category Zs;
- *whitespace characters*, as designated by Unicode 5.0: spaces, line separator (U+2028), paragraph separator (U+2029), and six control characters (U+0009–D and U+0085);

  Victor: Removed information separators 1–4, and added U+0085, to be compatible with Unicode. Is there any reason not to do so?

[1]Unicode 5.0 also specifies several encodings for characters. Fortress does not distinguish different encodings of the same character, and treats canonically equivalent characters as identical. See the Unicode Standard [**?**] for a full discussion of encoding and canonical equivalence.

- *digit-group separator*: the narrow no-break space (U+202F);
- *character-literal delimiters*: ' ` ‘ ’
- *string-literal delimiters*: " “ ”
- *special non-operator characters*: & ′ ( ) , . ; \ ... ∀ ∃ := ⟦ ⟧
- *special operator characters*: * / : < = > [ ] ^ { | } → ↦ ⇒
- *recognized characters*: characters in any of the sets above.

Note that excepting apostrophe, the recognized characters are partitioned into word characters, operator characters, whitespace character, character-literal delimiters, string-literal delimiters and special non-operator characters. Apostrophe is both a word character and a character-literal delimiter.

It is a static error for a program to contain any control character (Unicode general category Cc) other than whitespace characters, except that “control-Z” (U+001A) is allowed and ignored if it is the last character of a program. It is a static error for the horizontal and vertical tab characters (U+0009 and U+000B) to occur outside a comment. It is a static error for \, any space with code point in U+2000–8, and any character other than a recognized character to appear outside of comments and string and character literals.

To ensure the idempotence of preprocessing, the use of the ampersand outside of comments and string and character literals is also severely restricted. This restriction is described precisely in Appendix A.

Do we support ampersand yet?

5.2 Words and Chunks

In a sequence of characters, a *chunk* is a nonempty contiguous subsequence. A *word* is a maximal chunk consisting of only word characters (letters, digits, underscore, prime characters, and apostrophe); that is, a word is one or more consecutive word characters delimited by characters other than word characters (or the beginning or end of the entire sequence).

5.3 Lines, Pages and Position

The characters in a Fortress program are partitioned into *lines* and into *pages*, delimited by *line terminators* and *page terminators* respectively. A *page terminator* is an occurrence of the character `FORM_FEED`. A *line terminator* is an occurrence of any of the following:

- `LINE FEED` (U+000A),
- `CARRIAGE RETURN` (U+000D) not immediately followed by `LINE FEED`,
- `NEXT LINE` (U+0085),
- `LINE SEPARATOR` (U+2028), or
- `PARAGRAPH SEPARATOR` (U+2029).

A character is on page n (respectively line n) of a program if exactly $n - 1$ page terminators (respectively line terminators) precede that character in the program. Thus, for example, the nth line terminator of a program is the last character on line n. A character is on line k of page n if it is on page n and is preceded by exactly $k - 1$ line terminators on page n. A character is at line position k on line n if it is preceded by exactly $k - 1$ characters on line n other than page terminators. Note that a page terminator does *not* terminate a line; thus, the character immediately following a page terminator need not be at line position 1.

If a character (or any other syntactic entity) x precedes another character y in the program, we say that x is *to the left of* y and that y is *to the right of* x, regardless of how they may appear in a typical rendered display of the program. Thus, it is always meaningful to speak, for example, of the left-hand and right-hand operands of a binary operator, or the left-hand side of an assignment expression.

5.4 Input Elements and Scanning

After preprocessing (see Section 5.8 and Appendix A), a program is partitioned into *input elements* by a process called *scanning*. Scanning transforms a program from a sequence of Unicode characters to a sequence of input elements. Input elements are always chunks: the characters that comprise an input element always appear contiguously in the program. Input elements are either *whitespace elements* (including *comments*) or *tokens*. A token is a *reserved word*, a *literal*, an *identifier*, an *operator token*, or a *special token*. There are five kinds of literals: boolean literals, character literals, string literals, the void literal, and numerals (i.e., numeric literals).

Conceptually, we can think of scanning as follows: First, the comments, character literals and string literals are identified. Then the remaining characters are partitioned into words (i.e., maximal chunks of letters, digits, prime characters, underscores and apostrophes), whitespace characters, and other characters. In some cases, words separated by a single `.` or digit-group separator (and no other characters) are joined to form a single numeral (see Section **??**). Words that are not so joined are classified as reserved words, boolean literals, numerals, identifiers, or operator tokens, as described in later sections in this chapter. It is a static error if any word in a program is not part of one of the input elements described above. All remaining whitespace characters, together with the comments, form whitespace elements, which may be *line-breaking*. Finally, chunks of symbols (and a few other special cases) are checked to see whether they form void literals (see Section **??**) or multicharacter operator tokens (see Section **??**). Every other character is a token by itself, either a special token (if it is a special character) or an operator token.

5.5 Comments

Fortress supports two kinds of comments, block comments and end-of-line comments. When not within string literals, occurrences of "$(*$", "$*)$" and "$(*)$" are *opening comment delimiters*, *closing comment delimiters* and *end-of-line comment delimiters* respectively. Opening and closing comment delimiters must be properly balanced,

In a valid program, every opening comment delimiter is balanced by a closing comment delimiter; it is a static error if comment delimiters are not properly balanced. All the characters between a balanced pair of comment delimiters, including the comment delimiters themselves, comprise a *comment*. Comments may be nested. For example, the following illustrates three comments, one of which is nested:

(* This is a comment. *) (* As is this (* nested *)
comment *)

5.6 Whitespace Elements

A whitespace element is a maximal chunk consisting of comments, whitespace characters that are not within string or character literals or numerals, and ampersands (U+0026) that are not within string or character literals.

We distinguish *line-breaking whitespace* from *non-line-breaking whitespace* using the following terminology:

- A *line-terminating comment* is a comment that encloses one or more line terminators. All other comments are called *spacing comments*.
- *Spacing* refers to any chunk of spaces, `FORM FEED` characters, whitespace characters other than line terminators, and spacing comments.
- A *line break* is a line terminator or nonnested line-terminating comment that is not immediately preceded by an ampersand, possibly with intervening spacing.

- *Whitespace* refers to any nonempty sequence of spacing, ampersands, line terminators, and line-terminating comments.
- *Line-breaking whitespace* is whitespace that contains at least one line break.

It is a static error if an ampersand occurs in a program (after ASCII conversion) unless it is within a character or string literal or a comment, or it is immediately followed by a line terminator or line-terminating comment (possibly with intervening spacing).

5.7 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

5.8 ASCII Encoding

To support program entry and legacy tools, Fortress provides an ASCII encoding; that is, for any valid program, which may include non-ASCII characters, there is an equivalent program consisting of only ASCII characters.[2] Before undergoing any other processing, a program is *preprocessed* to convert ASCII-encoded chunks into Unicode characters. This section describes the ASCII encoding informally; the full details follow from the preprocessing step, which is described precisely in Appendix A.

To allow string literals that evaluate to arbitrary strings, preprocessing does *not* convert chunks within string literals. Thus, preprocessing must determine, and maintain, the boundaries of string literals. (This is why string literals must be escaped within character literals.) Also, because unescaped string-literal delimiters in comments do not actually delimit string literals of the program (and need not even be balanced), preprocessing must also determine comment boundaries.

First, replace any next line character (U+0085) with a line feed (U+000A), and replace any unescaped left or right double quotation mark (i.e., U+201C–D), and any left or right double quotation mark in a comment (escaped or not), by the ASCII quotation mark (U+0022). After this step, if the program is valid, there are no non-ASCII control

[2]See Appendix A for the precise notion of equivalence guaranteed here.

characters, and a left or right double quotation mark can occur only escaped within a character or string literal. (This step could make an invalid program valid, but it will not make a valid program invalid.)

For every character, Unicode 5.0 specifies a name (sometimes it specifies several alternatives) that consists of only ASCII letters, hyphens and spaces. Within a character literal, replace any non-ASCII character with its Unicode 5.0 name. Within a string literal, replace any non-ASCII character with its name in a quoted-character escape sequence (i.e., a character literal of that character immediately preceded by backslash; see Section **??**).

After this step, a non-ASCII character must not be part of a string or character literal, and it must not be a string-literal delimiter or a control character (the only control character allowed in a valid program is the next line character). Proceeding from left to right, replace each such character with the *Fortress variant* of a name, which is the string resulting from replacing hyphens and spaces in the name with underscores. (For consistency, it is also usually permissible to use the Fortress variant of a name instead of the Unicode 5.0 name when replacing non-ASCII characters in character and string literals.) If the immediately preceding character is a letter, digit or underscore, then insert an ampersand between them. (This prevents adjacent names from running together and being interpreted as a single name.)

Although the above encoding always works, it tends to greatly expand the program, and in particular, to produce very long lines, because most Unicode 5.0 names for characters are long (to avoid clashes). The ASCII encoding of Fortress mitigates these problems in two ways.

First, it allows a word to broken across a line by putting an ampersand immediately after the character before the break, and one immediately before the character after the break (with no intervening whitespace characters between the word characters and the ampersands).

Second, it specifies many short alternatives to the standard Unicode names. For example, a "digit-group separator" in a numeral (see Section **??**) can be replaced by an apostrophe. Similarly, many non-ASCII operators, including most of the common ones, have *shorthands* specified in Appendix **??**, and Appendix A specifies other short names for some common non-ASCII non-operator characters, such as ∞, $\top$ and $\perp$ and the Greek letters. (With the Greek letters, it is even possible to omit adjacent ampersands in some cases.) Also, when there is no ambiguity, Fortress allows Fortress variants of the names to be shortened by omitting any of the following common words (and the underscore immediately following them): LETTER, DIGIT, RADICAL and NUMERAL.

Chapter 6

Types

Fortress provides a rich type system for describing expressions and values. Fortress includes mechanisms for defining both nominal and structural types. The type system supports *parametric polymorphism* (i.e., *generic types*) and *quantified types* (i.e., *universal types* and *existential types*). Types are partially ordered by *subtyping* (also called *conformance*) and can be combined using *union* and *intersection*. Two types are *equivalent* if each is a subtype of the other; equivalent types are the same type.

Victor: This is necessary for subtyping to be a partial order. (This is why conformance is only a pre-order in Scala.)

In addition to subtyping and type equivalence, types may be related by *exclusion*, *coercion*, or *covering*.

Fortress types represent sets of values (i.e., types are first-order); a value is an *instance* of a type to which it belongs. Fortress is *strongly typed*: Every expression has a *static type*; every value has an *ilk*,[1] which is the most specific type to which the value belongs; and the ilk of a value that results from evaluating an expression is always a subtype of the static type of the expression.

This is redundant with other parts of this chapter.

6.1 Kinds of Types

Fortress supports the following kinds of types:

- the types Any, Bottom and $()$ (pronounced "void");
- *trait types*, some of which are *object trait types*;
- *tuple types*;
- *arrow types*;
- *function types*;
- *combined types*, which comprise *union types* and *intersection types*;
- *quantified types*, which comprise *universal types* and *existential types*.

Function types, quantified types, combined types and the type Bottom are not *expressible*: they cannot appear in a program. See Section 6.10 for a precise specification of which types are expressible.

A type is *simple* if it is Any, Bottom, $()$, a trait type, a tuple type or an arrow type.

[1]The notion of *ilk* corresponds to what is sometimes called the *class* or *run-time type* of a value. We prefer *ilk* because this utility of this because the notion—and usefulness—of the most specific type to which a value belongs is not confined to run time. We prefer it to the term "class", which is used in The Java Language Specification [7], because not every language uses the term "class" or requires that every value belong to a class. For those who like acronyms, we offer the mnemonic retronyms "implementation-level kind" and "intrinsically least kind".

```
Type ::=
      SimpleType
    | FunctionType
    | UniversalType
    | ExistentialType
    | UnionType
    | IntersectionType
SimpleType ::=
      Any
    | Bottom
    | ()
    | TraitType
    | TupleType
    | ArrowType
TraitType ::=
      Id
    | Id[\ StaticArgList \]
TupleType ::= ( Type, TypeList )
ArrowType ::= io? SimpleType -> SimpleType Throws?
FunctionType ::= fn{ GeneralizedArrowTypeList }
GeneralizedArrowType ::=
      ArrowType
    | forall[ TypeParamList ] GenericArrowType
UniversalType ::= forall[ TypeParamList? ] GenericSimpleType
ExistentialType ::= exists[ TypeParamList? ] GenericSimpleType
UnionType ::= Union{ SimpleTypeList? }
IntersectionType ::= Intersect{ SimpleTypeList? }

Throws ::= throws UnionType
StaticArg ::= Type | Value | Opr
TypeParam ::= Id <: IntersectionType

GenericArrowType ::= io? GenericSimpleType -> GenericSimpleType GenericThrows?
GenericThrows ::= throws GenericUnionType
GenericUnionType ::= union{ GenericSimpleTypeList }
GenericSimpleType ::=
      SimpleType
    | GenericTraitType
    | GenericTupleType
    | GenericArrowType
    | TypeVar

TypeList ::= Type (, Type)*
StaticArgList ::= StaticArg (, StaticArg)*
GeneralizedArrowTypeList ::= GeneralizedArrowType (, GeneralizedArrowType)*
TypeParamList ::= TypeParam (, TypeParam)*
SimpleTypeList ::= SimpleType (, SimpleType)*
GenericSimpleTypleList ::= GenericSimpleType (, GenericSimpleType)*
```

Figure 6.1: Draft of a grammar for types. It is buggy in several ways, but I don't think it makes sense to write out a full grammar because it seems like it will be quite repetitious, so as to distinguish generic types, expressible types, etc.

6.2 Any, Bottom and the Type Hierarchy

Types are partially ordered by subtyping (see Section 6.11.1), which defines the *type hierarchy*. Any is the top of the type hierarchy: every type is a subtype of Any and every value belongs to Any. The only type that Any excludes is Bottom.

Bottom is the bottom of the type hierarchy: every type is a supertype of Bottom. It is *uninhabited*: no value belongs to Bottom, and Bottom excludes every type (including itself). Bottom is the only type that excludes itself. Bottom is also inexpressible: it cannot be written in a program (see Section 6.10)

This is redundant with Section 6.11.1.

Victor: Commented out definition of *leaf type*.

6.3 Type $()$

The type $()$ (pronounced "void") is the ilk of the value $()$; no other value belongs to $()$. Its only strict subtype is Bottom and its only strict supertype is Any. $()$ excludes every type other than Any.

6.4 Trait Types and the Type Object

Syntax:

TraitType	::=	*Id*
	|	*Id*⟦ *StaticParamList* ⟧

A *trait type* is a named type defined by a trait or object declaration (see Chapters 10 and 11). It is an *object trait type* if it is defined by an object declaration.

There is a special trait type Object, which is a supertype of every trait type, arrow type and function type, and of Bottom. Objectexcludes $()$ and every tuple type, and is a subtype of Any.

Should Object be a trait type, or its own special type?

A trait or object declaration may have *static parameters*, in which case it defines a *generic type* with those parameters. A generic type is *not* a (first-order) type; rather, it is a *type schema* whose parameters must be *instantiated* with *static arguments* to produce a trait type. The resulting type is an *instantiation* of the generic type. The trait or object declaration may impose conditions on the arguments with which its static parameters may be instantiated. See Chapter 10 for more of static parameters and how they may be instantiated.

For convenience, we consider a trait or object declaration without static parameters to define a generic type (with no parameters) whose unique instantiation is the trait type defined by the declaration. Thus, properties of instantiations of generic types also apply to trait types defined by non-parameterized trait and object declarations.

We say that one generic type G *determines* another generic type G' if every parameter of G' is a parameter of G. Given an instantiation I of G, the *corresponding instantiation* of G' is the one in which the parameters of G' are instantiated with the corresponding arguments of I.

Every trait type is a subtype of Object and Any, and a supertype of Bottom. A trait or object declaration may have an `extends` clause, which specifies a set of types and generic types determined by the generic type defined by the declaration. An instantiation of the generic type defined by such a declaration is a subtype of every type in this set, and of the corresponding instantiation of each generic type in this set.

A static parameter of a trait declaration may be declared with the `covariant` modifier. In this case, it is a *covariant parameter* of the generic type defined by the trait declaration. One instantiation of a generic type is a subtype of another instantiation of the same generic type if the argument instantiating each covariant parameter in the first instantiation is a subtype of the corresponding argument in the second instantiation, and the argument instantiating each non-covariant parameter is the same in both instantiations.

Every trait type excludes $()$, Bottom and every tuple type. Every trait type other than Object also excludes every arrow type and function type. A trait declaration may have an `excludes` clause, which also specifies a set of types and generic types determined by the generic type defined by the declaration. An instantiation of the generic type defined by such a declaration excludes every type in this set and the corresponding instantiation of each generic type in this set.

Fortress also imposes a rule called *instantiation exclusion*: Two instantiations of a generic trait type exclude each other if the corresponding arguments for any covariant parameter exclude each other, or if the corresponding arguments for any non-covariant parameter are not type equivalent.

I believe this exactly characterizes exclusion among different instantiations of a generic type, but the rule we impose is stronger: Any trait that extends (possibly indirectly) two instantiations of a generic type must extend an *expressible* instantiation of that generic type that is a subtype of both. In other words, among all instantiations of a generic type that are supertypes of some expressible trait type, one must be a subtype of all the others, and it must be expressible. This section should say only which instantiations exclude each other, as it does. It perhaps should make it clearer that what is said here is a consequence of instantiation exclusion, which is a rule applied to trait declarations.

An object trait type has no strict subtype other than Bottom, and it excludes any type that is not its supertype.

A trait declaration may have a `comprises` clause, which also specifies a set of types and generic types determined by the generic type of the declaration. An instantiation of the generic type defined by such a declaration is covered by the union of the types in this set and the corresponding instantiations of the generic type in this set.

A trait or object declaration may provide a *coercion* from a type or a generic type. (See Section **??** and Chapter 16.) If a trait or object declaration provides a coercion from a type, then any instantiation of the generic type defined by that declaration coerces that type. If it provides a coercion from a generic type, then an instantiation of the generic type defined by that declaration coerces the corresponding instantiation of the generic type of the coercion.

6.5 Tuple Types

Syntax:

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

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

Every tuple type is a supertype of Bottom, and a subtype of Any. No other type is a supertype of all tuple 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 Bottom, $()$, and every trait type, arrow type or function type. A tuple type excludes every tuple type that does not have the same number of element types. Also, tuple types with the same number of element types exclude each other if any pair of corresponding element types exclude each other.

Tuple types X and Y that do not exclude each other must have the same number of elements, and their intersection is defined elementwise: the intersection of X and Y is equivalent to a tuple type with the same number of elements in which each element type is the intersection of the corresponding element types of X and Y. (The intersection of tuple types—or any types—that exclude each other is Bottom.)

6.6 Arrow Types

Syntax:

$$\begin{array}{lll} \textit{ArrowType} & ::= & \texttt{io}\,?\ \textit{SimpleType} \rightarrow \textit{SimpleType Throws}? \\ \textit{Throws} & ::= & \texttt{throws}\ \textit{UnionType} \end{array}$$

An *arrow type* may be *io* or not, and has three constituent types: a *parameter type*, a *return type* and an *exception type*. An arrow type is written as an optional `io` modifier, followed by the *parameter type* followed by $\rightarrow$, followed by the *return type*, followed by an optional `throws` clause, which specifies the *exception type*. It is *io* if it has the `io` modifier. Its parameter type and return type must be simple types (see Section 6.1); its exception type must be a union type (see Section 6.8). If an arrow type has no `throws` clause, its exception type is Bottom; that is, an arrow type with no `throws` clause is equivalent to one with a `throws` $\{\,\mathrm{Bottom}\,\}$ clause (Bottom is equivalent to $\bigcup\{\mathrm{Bottom}\}$).

Every arrow type is a supertype of Bottom, and a subtype of Object and Any. No other type is a supertype of all arrow types. An arrow type X is a subtype of another arrow type Y if the following conditions hold:

- X is not io or Y is io;
- they have equivalent parameter types;
- the return type of X is a subtype of the return type of Y; and
- the exception type of X is a subtype of the exception type of Y.

Thus, arrow types are covariant in their return type and exception type and invariant in their parameter type.

An arrow type excludes Bottom, $()$, every trait type other than Object, every tuple type, and every function type that is not its subtype (see Section 6.7). However, arrow types do not exclude other arrow types because of overloading as described in Chapter 15.

An arrow type Y coerces another arrow type X if the following conditions hold:

- X is not io or Y is io;
- the parameter type of X is a strict supertype of the parameter type of Y;
- the return type of X is a subtype of the return type of Y; and
- the exception type of X is a subtype of the exception type of Y.

Note that all but the second of these conditions are the same as the conditions for X to be a subtype of Y. This coercion replaces what in some languages is handled by contravariance.

6.7 Function Types

Syntax:

FunctionType	::=	`fn` $\{$*UniversalArrowTypeList*$\}$
UniversalArrowTypeList	::=	*UniversalArrowType*(, *UniversalArrowType*)*
UniversalArrowType	::=	*ArrowType*
	|	$\forall$[*TypeParamList*] *GenericArrowType*

A function type is the ilk of a function value. We distinguish them from arrow types to handle overloaded functions (see Chapter 15). We represent a function type by the keyword `fn` followed by a set of arrow types and *universal arrow types*, where a universal arrow type is a universal type whose constituent generic type is a generic arrow type (see Section 6.9).

More intuition of a function type?

Function types are *not* expressible (see Section 6.10), and restrictions on overloading severely restrict what function types arise in a program. See Chapter 15 for details.

Victor: Commented out discussion about well-formed function types and applicability. That should be in the overloading chapter. Or do we really want to talk about a function *type* being applicable to an argument, rather than function declarations.

A function type is a subtype of each of its constituent types, and all of their supertypes, including Object and Any. The only strict subtype of a function type is Bottom, and a function type excludes every type that is not its supertype.

6.8 Combined Types

Syntax:

UnionType	::=	*SimpleType* $\cup$ *SimpleType*
	|	$\bigcup$\{ *SimpleTypeList*? \}
IntersectionType	::=	*SimpleType* $\cap$ *SimpleType*
	|	$\bigcap$\{ *SimpleTypeList*? \}
SimpleTypeList	::=	*SimpleType*(, *SimpleType*)*

Any finite set of types may be *combined* by *intersection* or *union* to produce an *intersection type* or *union type* respectively. Informally, an intersection type is the "largest" type that is a subtype of all its constituent types; a union type is the "smallest" type that is a supertype of all its constituent types.

Intersection and union types are not expressible (see Section 6.10).

This is tricky: technically, it is not a type that is expressible or not, but the form of the type term. Perhaps it is better to do as Scala does, and use "type" to refer to the form, and so have subtyping define a pre-order on types, and a partial order on equivalence classes of types.

An intersection type is a subtype of each of its constituents, and it is a supertype of any type that is a subtype of all its constituents. A union type is a supertype of each of its constituents, and it is a subtype of any type that is a supertype of all its constituents.

If one constituent of an intersection type is a subtype of all the constituents, then the intersection type is equivalent to that constituent. If one constituent of a union type is a supertype of all the constituents, then the union type is equivalent to that constituent.

An intersection type excludes a type T if any of its constituents excludes T. A union type excludes a type T if all its constituents exclude T.

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, $V = S \cap T$.

6.9 Quantified Types

Syntax:

UniversalType	::=	$\forall$[*StaticParamList*?] *GenericSimpleType*
ExistentialType	::=	$\exists$[*StaticParamList*?] *GenericSimpleType*
GenericSimpleType	::=	*SimpleType*
	|	*GenericTraitType*
	|	*GenericTupleType*
	|	*GenericArrowType*
	|	*TypeVar*

Fortress supports a restricted form of *universal types* and *existential types*, collectively called *quantified types*. Even this restricted form is more general than necessary for Fortress: the only quantified types that appear in the static type of an expression or the ilk of a value, or in any of their supertypes, are universal arrow types, which may be constituents of a function type, and the type system could be formulated without these kinds of types entirely. However, we extend the type system with these kinds of types as a convenience for describing static type checking (and dispatch?).

A quantified type is a quantifier (either $\forall$ or $\exists$) followed by a bracketed list of static parameter bindings, followed by a generic type, which must have the form of a trait type, a tuple type or an arrow type (or it can be degenerate; that is, it must not have any parameters). As mentioned in Section 6.4, a generic type is not a type, but a type schema with parameters that must be instantiated. Every parameter of the constituent generic type of a quantified type must be declared in the list of static parameter bindings.

Do we allow degenerate quantified types (i.e., ones with an empty static parameter list)? I'm assuming we do for now.

If a static parameter of a quantified type does not appear in the type's constituent generic type, then the quantified type is equivalent to one in which the binding for that static parameter is removed. A quantified type with no static parameters is equivalent to the (unique) instantiation of its constituent generic type.

Each static parameter binding specifies a constraint on the argument that may instantiate that parameter. For a type parameter, it is an upper bound on the type argument (i.e., the argument must be a subtype of the specified bound), and this upper bound must be a simple type or an intersection type.

Do we allow other kinds of static parameters?

A type is a *valid instantiation* of a quantified type if it is an instantiation of the quantified type's constituent generic type with arguments that satisfy the specified constraints. Informally, we can think of a universal type as the intersection of all its valid instantiations, and an existential type as the union of all its valid instantiations. However, quantified types are not special cases of combined types because a quantified type may have infinitely many valid instantiations.

A universal type is a subtype of any of its valid instantiations (and of any supertype of such types). It is a supertype of a type T if all its valid instantiations are supertypes of T. A universal type U is a subtype of another universal type V if for every valid instantiation I_V of V, there is a valid instantiation of U that is a subtype of I_V. A universal type excludes a non-quantified type T if any of its valid instantiations exclude T. Two universal types exclude each other if any valid instantiation of one excludes any valid instantation of the other.

An existential type is a supertype of any of its valid instantiations constraints (and of any supertype of such types). It is a subtype of a type T if all its valid instantiations are subtypes of T. An existential type E is a subtype of another existential type F if for every valid instantiation I_E of E, there is a valid instantiation of F that is a supertype of I_E. An existential type excludes a non-quantified type T if all its valid instantiations exclude T. Two existential types exclude each other if every valid instantiation of one excludes every valid instantiation of the other.

From the above, we can already derive that a universal type is a subtype of an existential type if some valid instantiation of the universal type is a subtype of some valid instantiation of the existential type. An existential type is a subtype of a universal type if every valid instantiation of the existential type is a subtype of every valid instantiation of the universal type. A universal type and an existential type exclude each other if every valid instantiation of the existential type excludes some valid instantiation of the universal type.

6.10 Expressible Types

Function types and Bottom 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 Bottom.

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 **??**.

6.11 Relations on Types

6.11.1 Subtyping

Types are partially ordered by the *subtype relation* (i.e, the subtype relation is reflexive, transitive and antisymmetric), which defines the *type hierarchy*. 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 *(strict) supertype* of U if U is a (strict) subtype of T.

We define three fundamental relations on types: *subtype*, *exclusion* and *coercion*.

The *subtype relation* is a partial order on types that defines the *type hierarchy*; that is, it is reflexive, transitive and antisymmetric. 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. Any is the top of the type hierarchy: every type is a subtype of Any. Bottom is the bottom of the type hierarchy: every type is a supertype of Bottom.

The *exclusion relation* is a symmetric relation between two types whose intersection is Bottom.

Because $\mathrm{Bottom}$is uninhabited (i.e., no value has type Bottom), types that exclude each other are disjoint: no value can have a type that is a subtype of two types that exclude each other. Note that Bottomexcludes every type including itself (because the intersection of $\mathrm{Bottom}$and any type is Bottom), and also that no type other than Bottomexcludes Any(because the intersection of any type T and $\mathrm{Any}$is T). $\mathrm{Bottom}$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).

6.12 Types in the Fortress Standard Libraries

The Fortress standard libraries define simple standard types for literals such as $\mathrm{BooleanLiteral}[\![b]\!]$, $()$ (pronounced "void"), $\mathrm{Character}$, 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 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 Any, 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.

6.13 Values and Ilks

Every expression has a *static type*, and every value has an *ilk*.

Chapter 7

Names and Declarations

Chapter 8

Variables

Chapter 9

Functions

Chapter 10

Traits

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

$$
\begin{array}{lcl}
\mathsf{File} & ::= & \mathsf{CompilationUnit} \\
 & | & \mathsf{Imports}^{?}\ \ \mathsf{Exports}\ \ \mathsf{Decls}^{?} \\
 & | & \mathsf{Imports}^{?}\ \ \mathsf{AbsDecls} \\
 & | & \mathsf{Imports}\ \ \mathsf{AbsDecls}^{?} \\
\mathsf{CompilationUnit} & ::= & \mathsf{Component} \\
 & | & \mathsf{Api}
\end{array}
$$

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 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 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 $()$.

12.10 Label and Exit

12.11 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 false, no further action is taken; evaluation of the `while` expression completes normally with value $()$. But if v is 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.
  $k{:}\,\mathbb{Z} := 0$
  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 $\mathrm{Chain}\big(\mathit{item}{:}\,\mathbb{Z}, \mathit{link}{:}\,\mathrm{Maybe}[\![\mathrm{Chain}]\!]\big)$ end
$\mathit{printValues}\big(x{:}\,\mathrm{Maybe}[\![\mathrm{Chain}]\!]\big){:}\,() =$ do
    $\mathit{cursor}{:}\,\mathrm{Maybe}[\![\mathrm{Chain}]\!] = x$
    while $(\mathit{next} \leftarrow \mathit{cursor})$ do
      $\mathit{println}(\mathit{next}.\mathit{item})$
      $\mathit{cursor} := \mathit{next}.\mathit{link}$
    end
  end
end
$\mathit{run}() = \mathit{printValues}\big(\mathrm{Just\ Chain}(1, \mathrm{Just\ Chain}(2, \mathrm{Just\ Chain}(3, \mathrm{Just\ Chain}(4, \mathrm{None}))))\big)$
```

12.12 Generators

12.13 For Expressions

12.14 Ranges

12.15 Reduction Expresions

12.16 Comprehensions

12.17 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 12.12), 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 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 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(x “is” (if v > 0 then “positive” else “nonpositive”))

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

12.18 Case Expressions

12.19 Typecase Expressions

12.20 Atomic Expressions

12.21 Throw Expressions

12.22 Try Expressions

12.23 Type Ascription

12.24 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

Fortress operators provide a rich alternate syntax for defining and invoking certain functions and methods. The goal is to support a wide variety of notations traditionally used in mathematics and domain-specific fields of application. There are several distinct kinds of operator syntax:

- Use of a single symbol as a one-operand *prefix* operator (example: $-x$)
- Use of a single symbol or a superscripted symbol as a one-operand *postfix* operator (example: $n!$)
- Use of a single symbol as a zero-operand *nofix* operator (this is rare, but is used to implement the use of colon as a "select entire axis" subscript, as in $x_:$)
- Use of a single symbol as a two-operand *infix* operator (example: $x + y$)
- Use of *juxtaposition* as a two-operand infix operator, typically to express multiplication (example: $(n + 1)(n + 2)$)
- Use of a matching pair of symbols as an *enclosing* operator, taking any number of operands separated by commas (example: $\langle a, b, c\rangle$)
- Use of a matching pair of symbols as a *subscripting* operator, taking a preceding operand plus any number of enclosed operands separated by commas (example: $x[m, n]$)
- Use of a matching pair of symbols plus $:=$ as a *subscripting assignment* operator, taking a preceding operand, any number of enclosed operands separated by commas, and a following operand (example: $x[m, n] := 3$)
- Use of a matching pair of symbols plus a vertical bar $\mid$ as a *comprehension* (example: $\{\, x \mid x \leftarrow a, x > 0 \,\}$)
- Use of a symbol as part of *big operator* syntax (examples: $\sum\limits_{i \leftarrow 0:9} x[\,i\,]$ and $\underset{i \leftarrow 0:0}{\mathtt{MAX}}\ x[\,i\,]$)

In addition, there are special rules for treating clusters of infix operators as if they were either *chained* (example: treating $0 \leq j < k \leq n$ as meaning $(0 \leq j) \wedge (j < k) \wedge (j \leq n)$) or a single *multifix* operator (example: treating $A \times B \times C \times D$ as a single operator named $\times$ with four operands).

The fixity of an operator use 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. In addition, an operator may have multiple (overloaded) declarations of the same fixity, just as functions or methods may be overloaded (see Chapter **??** for a discussion of overloading). An invocation of an overloaded operator is resolved first via the fixity of the operator; then, among the visible declarations for the operator with that fixity, the most specific applicable declaration is chosen.

Special syntax rules, explained in Section 14.4, govern the analysis of any operator consisting of one or more vertical bars, such as $|$ and $\|$ and $|||$; these rules allow vertical bars to be used both as infix operators (as in $k \mid m$) and as enclosing operators (as in $|x|$).

Operators are not values. If it is desired to pass an operator as an argument, the correct approach is usually to use a function expression; for example, to pass the $+$ operation to a function f, one might write $f\big(\mathtt{fn}\ (x, y) \Rightarrow x + y\big)$.

Operator declarations are described in Chapter **??** and operator applications are described in Section **??**, Section **??**, and Section 12.16.

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.

left context	whitespace	operator fixity	whitespace	right context
primary tail	yes	**infix**	yes	primary front
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary tail	yes	**infix**	yes	operator
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary tail	yes	**error** (postfix)		`,` `;` right encloser
	no	**postfix**		
primary tail	yes	**infix**		line break
	no	**postfix**		
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 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 $A \subseteq B \leq C$ 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

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

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 ⌊ ⌋ and ⌈ ⌉ 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 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 12.16.

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 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 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: $((\mathit{reduce}(f))(a))(x+1)(\mathit{atan}(x+2))$. Now type analysis determines that the element $(\mathit{atan}(x+2))$ 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:

$$(((\mathit{reduce}(f))(a))(x+1))(\mathit{atan}(x+2))$$

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 | ^ | Encloser | ∑ | ∏) 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 | ^ | Encloser | ∑ | ∏) 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 Γ 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 $\llfloor x \rrfloor = 2^{\lfloor \log_2 x \rfloor}$, hyperceiling $\llceil x \rrceil = 2^{\lceil \log_2 x \rceil}$, hyperhyperfloor $⦀\!\lfloor x \rfloor\!⦀ = 2^{\llfloor \log_2 x \rrfloor}$, and hyperhyperceiling $⦀\!\lceil x \rceil\!⦀ = 2^{\llceil \log_2 x \rrceil}$ 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	ith element of one-dimensional array a
`m[i,j]`	*is displayed as*	m_{ij}	i,jth element of two-dimensional matrix m
`space[i,j,k]`	*is displayed as*	$space_{ijk}$	i,j,kth element of three-dimensional array $space$
`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 `REM` gives the remainder from such a truncating division: $m\,\mathtt{REM}\,n = m - n(m \div n)$. The operator `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 `DIVREM` and `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 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 $\mathrlap{\bigtriangleup}{\times}$ (or $\mathrlap{\bigtriangleup}{\cdot}$) and $\mathrlap{\bigtriangleup}{/}$. Multiplication and division in "round toward negative infinity" mode may be expressed with $\mathrlap{\bigtriangledown}{\times}$ (or $\mathrlap{\bigtriangledown}{\cdot}$) and $\mathrlap{\bigtriangledown}{/}$.

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 $\mathrlap{\bigtriangleup}{+}$ and $\mathrlap{\bigtriangleup}{-}$. Addition and subtraction in "round toward negative infinity" mode may be expressed with $\mathrlap{\bigtriangledown}{+}$ and $\mathrlap{\bigtriangledown}{-}$.

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 NaN then NaN is returned. The floating-point operations `MAXNUM` and `MINNUM` behave similarly except that if one argument is 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: $n\,\texttt{CHOOSE}\,k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$.

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 ($\mathit{true}/\mathit{false}$) 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 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 $\mathit{possibly}$ and $\mathit{certainly}$ are useful for converting boolean intervals to boolean values for testing. Thus $\mathit{possibly}(x > y)$ 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 $\mathit{certainly}(x > y)$ 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:

$\wedge\!\!\!\wedge$	bitwise AND
$\vee\!\!\!\vee$	bitwise inclusive OR
$\underline{\vee\!\!\!\vee}$	bitwise exclusive OR

The prefix operator $\neg\!\!\neg$ 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 6 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 ith 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 ith 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

Lexical Preprocessing

Preprocessing consists of three steps: “pasting” words across line terminators; replacing chunks of ASCII characters with single Unicode characters; and replacing apostrophes with the “digit-group separator” (see Section **??**).

Appendix B

Simplified Grammar for Application Programmers and Library Writers

B.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)*

B.2 Top-level Declarations

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

B.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

B.4 Object declaration

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

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

B.5 Variable Declaration

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

B.6 Function Declaration

FnDecl	::=	FnDef
	$\mid$	AbsFnDecl
FnDef	::=	FnMods$^?$ FnHeaderFront FnHeaderClause = Expr
AbsFnDecl	::=	FnMods$^?$ FnHeaderFront FnHeaderClause
	$\mid$	Name : ArrowType

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

B.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

B.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

B.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

B.10 Method Parameters

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

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

B.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

B.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

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

B.14 Expressions Enclosed by Keywords

DelimitedExpr	::=	TupleExpr
	$\mid$	ObjectExpr
	$\mid$	DoExpr
	$\mid$	LabelExpr
	$\mid$	WhileExpr
	$\mid$	ForExpr
	$\mid$	IfExpr
	$\mid$	CaseExpr
	$\mid$	TypecaseExpr
	$\mid$	TryExpr
TupleExpr	::=	((Expr ,)* (Expr ... ,)$^{?}$ Binding (, Binding)*)
	$\mid$	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 )*
```

B.15 Local Declarations

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

B.16 Literals

Literal	::=	( )
	$\mid$	BooleanLiteral
	$\mid$	CharacterLiteral
	$\mid$	StringLiteral
	$\mid$	NumericLiteral
RectElements	::=	Expr MultiDimCons*
MultiDimCons	::=	RectSeparator Expr
RectSeparator	::=	; +
	$\mid$	Whitespace

B.17 Types

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

```
TupleType          ::=  ( ( TypeRef , )* ( TypeRef ... , )? KeywordType ( , KeywordType )* )
                     |  ( ( TypeRef , )* TypeRef ... )
                     |  SimpleTupleType
KeywordType        ::=  Id = TypeRef
SimpleTupleType    ::=  ( TypeRef , TypeRefList )
TypeRefList        ::=  TypeRef ( , TypeRef )*
StaticArgList      ::=  StaticArg ( , StaticArg )*
StaticArg          ::=  Op
                     |  TypeRef
                     |  ( StaticArg )
ArraySize          ::=  ExtentRange ( , ExtentRange )*
ExtentRange        ::=  StaticArg? # StaticArg?
                     |  StaticArg? : StaticArg?
                     |  StaticArg
```

B.18 Symbols and Operators

```
AssignOp           ::=  :=
                     |  Op =
Accumulator        ::=  ∑ | ∏ | BIG Op
```

B.19 Identifiers

```
IdList             ::=  Id ( , Id )*
Name               ::=  Id
                     |  opr Op
DottedId           ::=  Id ( . Id )*
BindId             ::=  Id
                     |  _
```

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.