Contents

formal:
  formal-tag identifier formal-type$_{opt}$*tp*

3 Organizapion

This specification is organized as follows:

Section 1, Scope, describes the scope of this specification.

Section 25, Input and Output, describes support for input and output in Chapel, including file input and output..

Section 26, Standard Modules, describes the standard modules that are provided with the Chapel language.

Acknowledgments

```
  // isEmpty? method: true if the stack is empty; otherwise false
  def isEmpty? return numItems == 0;
}
```

6 Lexical Structure

This section describes the lexical components of Chapel programs.

6.4.2 Keywords

The following keywords are reserved:

```
atomic      begin       bool        break       by
class       cobegin     complex     config      const
```

Type	Minimum Value	Maximum Value

9 Conversions

A conversion allows an expression of one type to be converted into another type. Conversions can be either implicit or explicit.

Implicit conversions can occur during an assignment (from t

9.1.3 Implicit Class Conversions

10 Expressions

This section defines expressions in Chapel. Forall expressions are described in §22.2.

The syntax for an expression is given by:

```
expression:
  literal-expression
```

A *call-expression* is resolved to a particular function according to the algori

10.5 Casts

A cast is specified with the following syntax:

cast -

```
    def -(a: uint
def *(a: real(128), b: real(128)): real(128)

def *(a: imag(32), b: imag(32)): real(32)
def *(a: imag(64), b: imag(64)): real(64)
def *(a: imag(128), b: imag(128)): real(128)

def *(a: complex(64), b: complex(64)): complex(64)
def *(a: complex(128), b: complex(128)): complex(128)
def *(a: complex(256), b: complex(256)): complex(256)

def *(a: real(32), b: imag(32)): imag(32)
def *(a: imag(32), b: real(32)): imag(32)
def *(a: real(64), b: imag(64)): imag(64)
def *(a: imag(64), b: real(64)): imag(64)
def *(a: real(128), b: imag(128)): imag(128)
def *(a: imag(128), b: real(128)): imag(128)

def *(a: real(32), b: complex(64)): complex(64)
def *(a: complex(64), b: real(32)): complex(64)
def *(a: real(64), b: complex(128)): complex(128)
def *(a: complex(128), b: real(64)): complex(128)
def *(a: real(128), b: complex(256)): complex(-2.25145())-2.25145]TJ /R346 7.97011 Tf -229.561 -18.95
```

10.9.8 Exponentiation Operators

The exponentiation operators are predefined as follows:

```
def **(a: int(32), b: int(32)): int(32)
def **(a: int(64), b: int(64)): int(64)
def **(a: uint(32), b: uint(32)): uint(32)
def **(a: uint(64), b: uint(64)): uint(64)
def **(a: uint(64), b: int(64))
def **(a: int(64), b: uint(64))

def **(a: real(32), b: real(32)): real(32)
def **(a: real(64), b: real(64)): real(64)
def **(a: real(128), b: real(128)): real(128)
```

For each of these definitions that return a value, the result is the value of the first operand raised to thedpower of the second operand.

ie i290.849(a)-1198998(i)0.965521(t)-239.93u(e)-2(d)-52(p)66.65115(n)-5.88872(t)8006691(i)0.96552y(p)-5.89115

enitl ii

```
[illegible] Td [(()12.7965(a)[illegible]&011 Tf -18-2.24992(*)-2.24992]TJ 9.6 1.43984 Td [(()12.7965(a)&011 Tf -18-2.24992(*)-2.24992]TJ 9
    de&011 Tf -18-2.24992(*)-2.24992]TJ 9.6 1.43984 Td [(()12.7965(a)
f -18-2.24992(*)-2.24992]TJ 9.6 1.43984 Td [(()12.7965(a)
```

Expressions

The result of `a > b` is true if `a` is greater than `b`; otherwise her result is false.

The result of `a == b` is true if `a` and `b`

11 Statements

11.8 The For Loop

The for loop iterates over sequences, domains, arrays, iterators, or any class that implements the structural iterator interface. The syntax of the for loop is given by:

```
for
```

12.2.2 Command-Line Arguments

A predefined variable is used to capture arguments to the execution of a program. It has this declaration:

```
var argv: seq of string;
```

Default expressions allow for the omission of actual arguments at the call site, resulting in the imp.89115(l)0.965521(t)0icit p.89
of default value. Default values are discussed in §13.4.2.

Functions can ike a variable number of arguments. Such func tions are discussed in §

named argument passing is used to map the actual arguments to

arity	**operators**
unary	

Valid Mapping A function call is mapped to a function according to the following steps:

Each actual argument that is passed by name is matched to the formal argument with that name. If there is no formal argument with that name, there is no valid mapping.

The remaining actual arguments are mapped in order to the remaining formal arguments in order. If

14 Classes

Classes are an abstraction of a data structure where the stor

```
class Actor {
  var name: string;
  var age: uint;
}
```

14.7.5 Inheriting from Multiple Classes

15 Records

A record is a data structure that is like a class but that has va

16 Unions

17.5.1 Declaring Homogeneous Tuples

18.5 Iteration over Sequences

18.7.2 Sequence Indexing by Tuples

If `s` is a sequence and `t`

18.8.2 Tensor Product Promotion

If the function `f` were called by using square brackets instead of parentheses

18.12.3 The *spread* Function

```
def spread(s: seq, length: int, dim: int = 1)
```

Indefinite arithmetic sequences can be iterated over with zi

19.5.1 Changing the Indices in Indefinite Domains

20 Iterators

An iterator is a function that conceptually returns a sequen

20.4 The Structural Iterator Interface

21 Generics

Chapel supports generic functions and types that are parameterizable over both types and parameters. The generic functions and types look similar to non-generic functions and types already discussed.

21.1 Generic Functions

21.1.2 Formal Parameter Arguments

If a formal argument is specified with intent `param`, then a parameter must be passed to the function at the call site. A copy of the function is instantiated for each unique parameter that is passed to this function at a call site. The formal argument is a parameter.

Example

```
def fillTuple(param p: int, x: ?t) {
  var result: p*t;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

21.3.3Fids without Ty.s

21.4 fihere Expressions

22.1.2 The Ordered Forall Loop

22.7.4 Synchronization Variables of Record and Class Types

Inside the atomic statement is a sequential implementation of removing a particular object de-

```
forall d in D {
  // body
}
```

the body of the loop is executed in the locale where the index `d` maps to by the distribution of `D`.

23.3.2 Distributed Arrays

Arrays defined over a distributed domain will have the element variables stored on the locale detersined by the distribution. Thus, if `d` is an index of distributed domain `D` and `A` is an array defined over that domain, then `A(d).locale` is the locale to which `d` maps to according to `D`.

23.3.3 Undistributed Domains and Arrays

If a domain or an array does not have a distributed part, the domain or array is not distributed and exists only on the locale on which it is defined.

24 Reductions and Scans

Chapel provides a set of built-in reductions and scans with parallel semantics, a mechanism for defining more reductions and scans with efficient implementations, and syntact support to make reductions and scans easy to use.

25.2 Standard files *stdout*, *stdin*, and *stderr*

The files `stdout`

25.7 Default

Exits the program if `test` is false and prints to standard error the location in the Chapel code of

Index