`Super Serial`
Language Reference Manual

Douglas Kaminsky (dk2848)

Fall 2014

1 Introduction

This manual describes the proposed standard for the `Super Serial` programming language, including all syntax, grammar and meta-level constructs that define the language.

2 Lexical Conventions

A program consists of one or more files, divided into logical *namespaces*. Namespaces begin on the next line after their declaration and continue until the end of file unless a new namespace is declared first, at which point it becomes the "active" namespace. A namespace called *Main* acts as the entry point into a program.

2.1 Tokens

Tokens can be loosely categorized into: identifiers, keywords, literals (numeric and otherwise), operators and structural elements. Any consecutive non-newline whitespace, including blank spaces, horizontal and vertical tabs are collapsed and treated as a token separators. Newlines, either as a single newline character or a newline with carriage return, are used to separate statements and type definition elements. Two newlines in a row signify the end of a definition (i.e. a newline at the end of the definition and a blank line after it).

When scanning an input stream, the next token will be produced using the token rule matching the greatest possible number of characters from the stream.

2.2 Comments

There are no comments in this language. This is a deliberate decision based on the belief of the language author that comments are of little to no value beyond automated documentation, which is outside the scope of this language. Rather, the language syntax and resulting semantics should be self-descriptive and read fluently, like a natural language.

2.3 Identifiers

$$
\begin{array}{lcl}
identifier_{capitalized} & = & \textbf{['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']}+ \\
identifier_{uncapitalized} & = & \textbf{['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9']}+ \\
 & & \\
identifier_{namespace} & = & identifier_{namespace}\textbf{'.'}identifier_{capitalized} \\
identifier_{namespace} & = & identifier_{capitalized} \\
identifier_{type} & = & identifier_{namespace}\textbf{'.'}identifier_{capitalized} \\
identifier_{type} & = & identifier_{capitalized} \\
identifier_{function} & = & \textbf{['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9' '\_']}+ \\
identifier_{extension} & = & \textbf{['a'-'z']}+ \\
identifier_{field} & = & \textbf{'"'}identifier_{uncapitalized}\textbf{'"'}
\end{array}
$$

There are several types of identifiers in this language, with slightly different construction rules:

- *Namespace identifiers* begin with a capital letter and can contain only letters and numbers. A namespace identifier can have multiple "segments" separated by a **'.'** character.
- *Type identifiers* begin with a capital letter and can contain only letters and numbers. A *fully qualified type name* can be constructed by connecting a namespace identifier with a type idenifier using a '.' character.
- *Function identifiers* begin with a lowercase letter and can contain letters, numbers and underscores.

- *Extension identifiers* consist solely of lowercase letters.
- *Field identifiers* follow the same rule as function identifiers but are bounded on both sides by quotation marks.

2.4 Keywords

The following keywords are reseved and may not be used as identifiers:

and	can	false	is	option	type
any	constraint	float64	literal	or	unique
apply	each	function	matches	some	using
array	escape	input	namespace	string	void
as	exp	int32	none	to	with
boolean	expect	int64	numeric	true	where

2.5 Literals

$$
\begin{array}{lcl}
sign & = & \textbf{['-', '+']} \\
digit & = & \textbf{['0'-'9']} \\
exp & = & \textbf{['e' 'E']}sign?digit+ \\
 & & \\
literal_{int} & = & sign?digit+ \\
literal_{float} & = & sign?digit + exp \\
literal_{float} & = & sign?digit + \textbf{'.'}exp? \\
literal_{float} & = & sign?digit * \textbf{'.'}digit + exp? \\
literal_{boolean} & = & \textbf{"true"} \\
literal_{boolean} & = & \textbf{"false"} \\
literal_{string} & = & \textbf{'''[\^{} ']'''} \\
literal_{option} & = & \textbf{"none"} \\
literal_{option} & = & \textbf{"some"}\textbf{('}expr\textbf{')} \\
literal_{void} & = & \textbf{"void"}
\end{array}
$$

The language supports several built-in types which can be represented by literals within code:

- *Integer literals* are represented by an optional sign and a series of digits representing the integer value

- *Float literals* allow representation of a subset of floating point values. Positive or negative numbers with or without a decimal component and with or without an exponent (scientific notation) are permitted.
- *Boolean literals* are "true" and "false"
- *String literals* are a sequence of characters surrounded by single quotes. Internally, strings are represented as a series of bytes corresponding to the ASCII encoded values of each character in the string, accompanied by a length counter that expresses the length of the string.
- *Optional literals* are either the keyword "none" or "some" accompanied by some expression describing the actual underlying value.
- *Void literal* is a single keyword, "void" that is used to represent the lack of a formal type, e.g. for operations with side effects and no return value.

3 Syntax Notation

Grammar rules described in this document use a subset of regular expression syntax, including:

- character classes (denoted by square braces, e.g. $[a-z]$)
- negative character classes (character classes beginning with a caret, indicating the inverse set of symbols is accepted, e.g. $[\hat{}abc]$)
- literal strings enclosed in single quotes (e.g. 'string')
- the Kleene closure operator (*)
- the one-or-more operator (+)
- the zero-or-one operator (?)

Alternatives are listed on separate lines. Associativity and precedence of operations is not expressed in this grammar, and in practice will be managed by the scanner generator used to create the lexical scanner.

4 Basic Language Constructs

4.1 Namespaces

Namespaces are the core unit of separation within a program. Aside from designating the entry point of the program by specifying the *Main* namespace, namespaces create logical separations between types. In addition, the type system also allows the specification of a field that can accept "any" member of a namespace. A namespace consists of declarations and a block of code that is executed after the remaining content of the namespace is processed, including references to other namespaces.

4.2 Types

Types are used to encapsulate data and methods. They are defined by specifying a number of fields and their corresponding types, along with a set of methods. Methods can be specified via contract (i.e. what things this type can do) or via implementation (i.e. how to do something), or both. A separate contract can be specified per datatype, or the datatype can be omitted and the type of the input inferred from the content of the function body. Types containing a 'can' declaration without a 'to' declaration are implicitly abstract.

Fields on a type instance are instantiated in order by specifying actual parameters either in a type extension clause (e.g. type B is type A with some predetermined value for some fields) or object instantiation (e.g. object a is an instance of type A with some value for some fields).

5 Type System

The type system is static and strongly typed, so there is no implicit conversion between types. Types can inherit from a single parent type, and potentially pre-specify individual values within the parent type, creating a form of closure over the parent type.

6 Expressions

6.1 Constant Expressions

$$
\begin{aligned}
expr &= \textbf{'input'} \\
expr &= literal_{int} \\
expr &= literal_{float} \\
expr &= literal_{string} \\
expr &= literal_{boolean} \\
expr &= literal_{option} \\
expr &= literal_{void}
\end{aligned}
$$

Literals and the **input** token constitute the constant expressions in this language. The **input** token is used to represent the currently examined datum when iterating over a data set or reading data of for a stream. It is also the implicit argument to certain other constructs.

6.2 Arithmetic Expressions

$$
\begin{aligned}
expr &= \textbf{'-'}expr \\
expr &= expr\textbf{'+'}expr \\
expr &= expr\textbf{'-'}expr \\
expr &= expr\textbf{'/'}expr \\
expr &= expr\textbf{'*'}expr \\
expr &= expr\textbf{'\%'}expr \\
expr &= expr\textbf{'exp'}expr
\end{aligned}
$$

Arithmetic can be performed on numeric types using numeric expression operators. This includes unary negation and binary operations addition, subtraction, division, multiplication, modulus and exponent application.

6.3 Boolean Expressions

$$
\begin{array}{lcl}
expr & = & \textbf{'!'}expr \\
expr & = & expr\textbf{'!='}expr \\
expr & = & expr\textbf{'='}expr \\
expr & = & expr\textbf{'>'}expr \\
expr & = & expr\textbf{'>='}expr \\
expr & = & expr\textbf{'<'}expr \\
expr & = & expr\textbf{'<='}expr
\end{array}
$$

Boolean logic is expressed using boolean operators, including logical negation, equality, inequality, greater than, greater than or equal to, less than and less than or equal to.

6.4 Binary Arithmetic

$$
\begin{array}{lcl}
expr & = & expr\textbf{'\&'}expr \\
expr & = & expr\textbf{'|'}expr \\
expr & = & expr\textbf{'\^{}'}expr \\
expr & = & expr\textbf{'»'}expr \\
expr & = & expr\textbf{'»>'}expr \\
expr & = & expr\textbf{'«'}expr
\end{array}
$$

Binary arithmetic operations include binary and, binary or, binary xor, left shift, right shift and right shift arithmetic. Because this language doesn't explicitly differentiate between signed and unsigned data, this language also uses the Java convention of right shift and right shift arithmetic being separate operators.

6.5 Other Expressions

$$
\begin{array}{lcl}
expr & = & \textbf{'('}expr\textbf{')'} \\
expr & = & expr expr \\
expr & = & identifier_{function} expr namedFuncActuals? \\
expr & = & identifier_{field} \\
expr & = & |. + | \\
namedFuncActuals & = & \textbf{"with"}namedFuncActualSeq \\
namedFuncActualSeq & = & namedFuncActualSeq\textbf{','}identifier_{extension} expr \\
namedFuncActualSeq & = & identifier_{extension} expr
\end{array}
$$

Expressions can be manually prioritized by adding parentheses.

Function application occurs by putting an argument next to a function identifier to specify its **input**. Additional arguments to a function must be named and provided using a *with* clause.

Putting any other two expressions next to each other will attempt to concatenate the two as strings. A field identifier on its own is also a valid expression.

File I/O is accomplished by enclosing a file name with '|' characters. This implicitly assigns to the "input" value based on the newline-delimited content of the file.

7 Declarations

7.1 Namespace Declaration

$$
\begin{array}{lcl}
namespaceDecl & = & \textbf{namespace} identifier_{namespace} newline namespaceDef \\
namespaceDef & = & \epsilon \\
namespaceDef & = & namespaceRefs * typeDecl * initBlock \\
namespaceRefs & = & \textbf{using} identifier_{namespace} newline \\
namespaceRefs & = & namespaceRefs \\
initBlock & = & stmt * newline
\end{array}
$$

Namespace declaration begins with the name of the namespace, followed by declarations of any other namespaces referenced by this namespace. Types can then be declared, and then finally a block of code can be specified to be executed when this namespace has been processed.

7.2 Type Declaration

$$
\begin{array}{lcl}
newline & = & \textbf{n} \\
newline & = & \textbf{n}\ \ \textbf{r} \\
 & & \\
typeDecl & = & \textbf{type}identifier_{type}typeDeclExt?newlinetypeDef? \\
typeDeclExt & = & \textbf{"is"}typeDescriptor \\
typeDescriptor & = & \textbf{"literal""'}identifier_{extension}\textbf{"'} \\
typeDescriptor & = & builtInType\textbf{"with"}typeDescriptorExt \\
typeDescriptorExt & = & typeDescriptorExtElement \\
typeDescriptorExt & = & typeDescriptorExt\textbf{','}typeDescriptorExtElement \\
typeDescriptorExtElement & = & \textbf{unique}identifier_{field} \\
typeDescriptorExtElement & = & \textbf{escape}literal_{string} \\
typeDescriptorExtElement & = & \textbf{constraint}userTypeRef \\
userTypeRef & = & identifier_{type}typeRefActuals? \\
typeRef & = & identifier_{type} \\
typeRef & = & builtInType \\
typeRef & = & \textbf{"any"}identifier_{namespace} \\
typeRefExt & = & \textbf{"with"} \\
typeRefActuals & = & \textbf{'('}typeRefActualSeq\textbf{')'} \\
typeRefActualSeq & = & typeRefActualSeq\textbf{','}typeRefActual \\
typeRefActualSeq & = & typeRefActual \\
typeRefActual & = & literal \\
typeDef & = & \textbf{'\{'}newlinefieldDecl * funcDecl * \textbf{'\}'}newline \\
fieldDecl & = & identifier_{field}\textbf{"->"}typeRefnewline \\
funcDecl & = & \textbf{"can"}identifier_{function}\textbf{"->"}typeRefnewline \\
funcDecl & = & \textbf{"to"}identifier_{function}typeRef?\textbf{"->"}newline?stmt + newline
\end{array}
$$

8 Statements

$$
\begin{array}{lcl}
stmt & = & identifier_{function}\textbf{"each"}exprnamedFuncActuals?newline \\
stmt & = & identifier_{field}\textbf{"<-"}exprnewline \\
stmt & = & printexprnewline \\
stmt & = & exprnewline
\end{array}
$$

Statements are essentially sequential operations. All statements end with a newline in this language.

Primarily, any expression can be used as a statement. Beyond that, I/O and assignment are the primary statements available. There is also a "list map" construct that is this language's primary mechanic for iteration.

9 More to Come

Due to computer death, this language manual had to be rewritten from scratch, and some features just didn't make the cut in the name of submitting something within a reasonable amount of time.

9.1 First Class Functions

This language will support functions as data, as well as anonymous function (lambda) syntax.

9.2 Extensions

This language will allow simple "typedef"-like functionality to create custom language keywords, e.g. ? predicate as a shortcut for function (? -> boolean).

9.3 Regular Expression Literals

This language will allow regular expressions to be specified as literals

10 Grammar

This section contains the collected grammar as described in the rest of this

$identifier_{capitalized}$ = **['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']**+
$identifier_{uncapitalized}$ = **['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9']**+

$identifier_{namespace}$ = $identifier_{namespace}$**'.'**$identifier_{capitalized}$
$identifier_{namespace}$ = $identifier_{capitalized}$
$identifier_{type}$ = $identifier_{namespace}$**'.'**$identifier_{capitalized}$
$identifier_{type}$ = $identifier_{capitalized}$
$identifier_{function}$ = **['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']**+
$identifier_{extension}$ = **['a'-'z']**+
$identifier_{field}$ = **'"'**$identifier_{uncapitalized}$**'"'**

$sign$ = **['-', '+']**
$digit$ = **['0'-'9']**
exp = **['e' 'E']**$sign?digit+$

$literal_{int}$ = $sign?digit+$
$literal_{float}$ = $sign?digit + exp$
$literal_{float}$ = $sign?digit$ + **'.'**$exp?$
$literal_{float}$ = $sign?digit * $ **'.'**$digit + exp?$
$literal_{boolean}$ = **"true"**
$literal_{boolean}$ = **"false"**
$literal_{string}$ = **'"[^ ']"'**
$literal_{option}$ = **"none"**
$literal_{option}$ = **"some"**'('*expr*')'
$literal_{void}$ = **"void"**

$newline$ = **\n**
$newline$ = **\n \r**

$typeDecl$ = **type**$identifier_{type}$$typeDeclExt?newline typeDef?$
$typeDeclExt$ = **"is"**$typeDescriptor$
$typeDescriptor$ = **"literal""'**$identifier_{extension}$**"'**
$typeDescriptor$ = $builtInType$**"with"**$typeDescriptorExt$
$typeDescriptorExt$ = $typeDescriptorExtElement$
$typeDescriptorExt$ = $typeDescriptorExt$**','**$typeDescriptorExtElement$
$typeDescriptorExtElement$ = **unique**$identifier_{field}$
$typeDescriptorExtElement$ = **escape**$literal_{string}$
$typeDescriptorExtElement$ = **constraint**$userTypeRef$
$userTypeRef$ = $identifier_{type}typeRefActuals?$
$typeRef$ = $identifier_{type}$
$typeRef$ = $builtInType$
$typeRef$ = **"any"**$identifier_{namespace}$
$typeRefExt$ = **"with"**