4. Syntax

This section presents an informal syntax for Joule. For a formal syntax, see Appendix B. Syntactic abstraction, the set of techniques for extending the Joule syntax, is discussed but not specified in this document.

In the Joule syntax presented here, typography is significant: whitespace delimits tokens and italics indicate comments. Boldface is used to denote syntactic keywords in the text, but is not semantically significant. Keywords occupy the same namespace as identifiers.

The Joule syntax presented in this document replaces a former one in which line indentation was significant.

4.1. Lexical Conventions

This section describes the token types for standard Joule programs. These include numerals, identifiers, keywords, labels, operators, special characters, whitespace, comments, literals and quasi-literals. Joule uses UNICODE for its character set.

When the UNICODE committee defines character categories such as numeric characters, identifier characters, and operator characters, Joule will adopt those distinctions. Until then, Joule will use the simplest possible distinctions.

4.1.1. Numerals

Numerals (the textual representations of the send ports to Joule numbers) are composed of the ASCII digits `0-9`. No other UNICODE characters are considered "numerals" in Joule.

4.1.2. Identifiers

Identifiers are sequences of UNICODE letters, digits, and operator characters that begin with a letter, or sequences of any UNICODE characters (including whitespace) enclosed by either straight (' ') or standard (‘ ’) single quotes, with backslash as an escape character. The quotes and escapes are *not* considered part of the identifier. Case is significant. Some examples of legal identifiers are:

```
x               list            a>              'an identifier'
question?       D→38a           δαιμον+         'letter \'a\' '
```

4.1.3. Keywords

Keywords are identifiers that are treated specially. They are shown in the text as **bold**, but this representation is not syntactically significant. The syntax extension system to be described in future versions of the docu-

ment will cover this in detail. Keywords provide syntactic structure. Examples of keywords are:

Server	**If**	**•**	**Syntax+**
Case	**Define**	Δ	'Right Here'

(The send keyword •, which does not begin with a letter, is an exception to the rule that keywords must be identifiers.)

4.1.4. Operators

Operators are sequences of UNICODE letters, digits, and operator characters that begin with an operator character. Some examples of legal operators are:

+	→	!=	<-than-3

4.1.5. Labels

Labels are identifiers followed by colons (":"), or the double-colon by itself. Labels, along with operators, are used as operations (message names).

sort:	::	delta+3:	'there now\':'

4.1.6. Characters

When the UNICODE operator character declarations are finalized, the set of Joule operator characters will be extended to include additional characters like ± and ÷.

Operator characters include:
`+ - * / < = > ! @ $ % ^ & | \ / ? ~ _ → ⇒`

Some characters are treated specially. These include:
`. ; , : # ' " ` { } [ ] ( )`

"." and ";" are handled specially to support a syntax that doesn't require character attributes: identifiers that end in "." are considered keywords (without the "."), and ";" begins a comment that consumes the rest of the line. ":" is used generally as the indicator of an operation label. "#" introduces an arbitrary quasi-literal; "##" introduces an arbitrary literal.

4.1.7. Whitespace and Comments

Whitespace includes: space, tab, linefeed, CR, and form-feed.

Comments are arbitrary characters written with italic character attributes. Joule treats comments just like whitespace. Comments cannot be embedded within a single identifier.

4.1.8. Literals and Quasi-literals

Two special token types are *literals* and *quasi-literals*. Though directly supported by the syntax, these both represent expression values and are described in the next section.

Common literal types like numbers, strings, and characters are defined. Joule also supports general literals: arbitrary user-defined objects embedded in the source code by multi-media editors. The support for this is beyond the scope of this document.

4.2. Expressions

At the bottom, Joule syntax is imperative, and therefore statement-based. However, because expressions are used so frequently for math and comparison operations, a rich expression syntax is supported which transforms cleanly into the relational syntax underneath.

In brief, complex expressions become separate statements with the distributor of an implicit results channel. The site of the original expression is replaced by a reference to the acceptor of the results channel. This means that nested expressions still compute completely concurrently with their embedding statement. This transformation is described in detail in **Section 5.4**.

Below is a simplified BNF for expressions. Multiple lines in the production definition are disjunctive; thus, a `simpleExpr` is an `Identifier`, or a `Literal`, or a `Quasiliteral`, etc.

Production	Production Definition	Example

`simpleExpr`	`Identifier`
	`Literal`
	`Quasiliteral`
	`tuple`
	`'('` `nestExpr` `')'`
		`bank>`
		`17.5`
		
		`oper: arg`

`opExpr`	`simpleExpr` `simpleExpr` *Operator* `opExpr`	`17` `3 + 17`
`nestExpr`	`simpleExpr` `simpleExpr opExpr`	`12` `bank deposit: chk`
`tuple`	*Operator* `opExpr*` *Label* `opExpr*`	`+ b` `get: i - 1 result>`

In BNF representations, `foo*` means zero or more instances of `foo` and `foo?` means zero or one instances of `foo`. Braces are used for grouping.

The apparent shift/reduce ambiguities in the grammar must be resolved by reducing (as YACC would).

Simple expressions designate particular values. These are:

4.2.1. Identifiers

Identifiers name communication ports on which messages can be sent to other Joule receivers, and which can be included in messages. These are just single tokens.

4.2.2. Literals

A literal expression statically designates a specific value that will be made available at run-time. It is represented as a single token to the compiler. Examples include numbers, shared immutable strings, and user-defined, embedded receivers (shared icons, shared print servers, etc.):

```
1234.5 ##"this is a test"
```

4.2.3. Quasi-literals

A quasi-literal expression designates a value which will be copied at run-time. These copies may incorporate literals, and run-time values. Examples include quasi-quoted lists as in Lisp, strings computed from formats as in C `printf` statements, and user-defined, embedded receivers.

For example, `#"This is a $t1"` where `t1` is `"Test"` would result in a copy of `"This is a Test"`.

4.2.4. Tuples

Tuples are used as messages in Joule. A tuple has a statically-available name, called an *operation*, and any number of arguments (including zero) which are other expressions. A tuple expression can be thought of as a special and extremely common kind of quasi-literal. *Operations* are either operators or labels:

```
+ 4 result>
req: arg1 arg2
```

Tuple expressions include all operator expressions following them, so they must be enclosed in parentheses (as a nested expression) in order for another expression to follow them in a line of source code.

4.2.5. Operator Expressions (`opExpr`)

Computation in Joule proceeds by sending and processing messages. The operator expression syntax supports conveniently sending messages commonly used in mathematical and relational expressions.

Operator expressions combine simple expressions into complex expressions using operators. Precedence is right-to-left—the first operator is applied to the first argument and the rest of the line, which will be the second operator applied to the second argument and the rest of the line.

The expression:	is interpreted as:
`3 + 4 + 5 - 12`	`3 + (4 + (5 - 12))`
`a <= b * c`	`a <= (b * c)`

4.2.6. Nested Expressions (`nestExpr`)

Nested expressions are enclosed by parentheses and can be used anywhere simple expressions are allowed. They support the explicit grouping as shown in the example above, and allow expressions that use label-based messages rather than just operator messages. For example, this statement

```
• x max: (y min: z)
```

sends the server `x` the `max:` operation with a single argument (the minimum of `y` and `z`). Without the parentheses, the `max:` request would be sent with two arguments: `y` and the tuple `min: z`.

4.3. Program Structure

Joule programs are composed of a sequence of forms. Each *form* starts with a keyword that identifies the syntactic type of that statement. Forms with the "•" keyword are used for the most frequent operation, message sending.

Syntactic extension tools allow users to associate new syntactic forms with keywords. The syntactic extension system is not presented in this

version of the manual, but many of the forms presented in later sections are actually syntactic abstractions built out of more primitive forms.

Forms typically have a **Keyword** followed by any number of operator expression arguments and ending with the corresponding **endKeyword**. In the interior of the form, underneath the keyword statement, is an optional block of more statements. Following the block are optional extension lines that have the same structure but whose keyword identifies them as part of the preceding statement. By convention, keywords that start with uppercase begin a form, keywords that start with lowercase begin extensions. A simple example is:

```
If amount <= balance
    • account withdraw: amount
else
    • account report-bounce:
endIf
```

The entire example is a single form; the **If** clause is the primary clause, the **else** clause is an extension. The **If** clause has one argument following, the operator expression `amount <= balance`. The nested form, `• account withdraw: amount`, uses the •-keyword statement form with a single argument, the tuple `withdraw: amount`. The nested form under the **else** extension is similar: it is a one form block using the •-keyword statement with a single argument, the tuple `report-bounce:`.

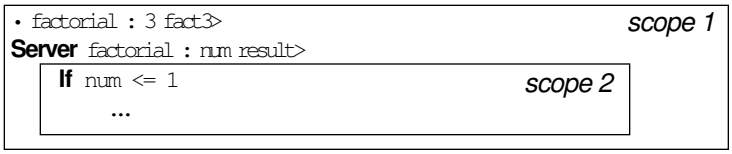
Note that tuple expressions include all operator expressions following them. For another expression to follow a tuple in a line of source code, the tuple must be enclosed in parentheses (as a nested expression).

4.4. Identifier Scoping

Any identifier may name a channel on which servers receive messages. An identifier that does so is said to be *bound* to that channel. Use of the identifier designates use of the channel to which it is bound.

Certain syntactic forms create new channels and bind identifiers to those channels. All bindings are *statically scoped*: the region of the source code in which the binding will be visible is an observable property of the source code. The same identifier may be bound to a different channel in an outer region that includes the inner region; the inner binding shadows the outer one, and is the only binding visible to code in the inner region.

Statements can create bindings in their *inner scope*—the statement itself and everything nested within it—or in their *outer scope*—the block that directly contains the statement, including all its sibling statements. The syntactic form of the statement determines where the statement makes bindings, and with what identifiers. **Server** is a simple construct that shows both kinds of binding:



Server makes a new channel for factorial requests named `factorial` and binds it in the outer scope (labeled *scope 1*). Each request sent to the factorial channel invokes the nested code with `num` and `result>`bound in the invoked code to the two arguments in the factorial request (roughly the channel for the argument and the channel for the revealed result). The two parameters are bound in the inner scope of the procedure statement.

Like Scheme, Joule is a statically scoped language with block structure. The block structure is represented by **Keyword**-**endKeyword** pairs; the binding site for a use of an identifier can be statically determined from the code. Unless hidden by the statement, bindings visible to a statement are also visible to statements nested within it. The multiple clauses of some statement forms may share the same inner scope, or may each introduce nested scopes only visible to that clause.

Strict static scoping allows visibility constraints to contribute to the modularity and security of the language. For instance, the **Define** construct makes a new channel and binds identifiers to its ports. The identifier supplied with **Define** is bound in the outer scope to the acceptor of the channel. This makes it visible both in the outer scope and the inner scope, since bindings in one scope are generally visible to all scopes within it. **Define** also binds a modification of the identifier (by convention, the identifier followed by ">") in the inner scope to the distributor of the channel. This distributor is visible only within the inner scope. The utility of this can be seen in:

```
Define bank
    implementation responding to messages on "bank>"
endDefine
bank clients that send messages to "bank"
```

Any number of clients can share the bank without trusting each other because their messages can only be received by the bank implementation (which they have to trust anyway). The visibility constraints of **Define** guarantee that the binding of the distributor of the channel is not visible outside the nested code within the **Define**-**endDefine** pair.