AFR-0
The Amlantis System Overview

Version 0.11

Markéta Lisová

November 6, 2022

Special Thanks To

Sabina Eva Lisová

Marie Strnadová

Ondřej Profant

Vojtěch Biberle

Tuan Tran

Contents

Part I

Introduction

Chapter 1

A Brief Introduction to the Amlantis System

Amlantis System is a collection of specifications of programming and data definition languages and their related tools, runtimes and libraries.

Those specifications do not always require particular implementations, but attempt to avoid undefined behaviours as much as possible.

Amlantis also provides an open source reference implementation of those specifications, but it is indeed possible for other people to write their own implementations or forks[1] of this default implementation, probably focusing on optimising other aspects of the system, maybe exploring new options of future development to be pull-requested into the default implementation.

A Few Notes on the Name

Amlantis' name has quite some history. The project started being named *Coral*, but that collided with another language of a similar name, *CORAL 66*. Then it got renamed to *Gear*, but that again collided with another language of the same name, which seemed inactive at the time, *zippers/gear*. Than an idea was born and Aml was named *Amlantis*, which is whatever you want it to be. It could be a misspelling of *Atlantis*[2], it could be an acronym like *A ML Language*, or maybe even something like *A ML Language And Neat Technology Improvement System*, or maybe *Caml* without the *C*. For the meaning of the cryptic *ML* part, search for the *Standard ML* or *OCaml*.

[1]Forks and pull requests are preferred way of help!

[2]Intentionally – because otherwise, it would be named Atlantis, but there is already a city of that name.

1.1 The Amlantis Languages

The Amlantis System is a home to more than only one programming language – in fact, it can be home to any number of languages, ones that Amlantis users can either extend from existing languages create or even create new ones entirely. Every language build within the Amlantis System shares a common type system and value models, and thus are interoperable.

The minimal Amlantis System contains these languages:

- **`Aml/Base`**, a minimal Lisp-like[3] language that is easy to parse and limited in functionality.
- **`Aml/Core`**, an ML-like[4] language that is easier to read and harder to parse, fully featured in functionality.
- **`Aml`**, an ML-like[5] language that extends **`Aml/Core`**, is easier to read and the hardest one to parse, also fully featured in functionality, and introducing more grammar than its parent **`Aml/Core`**.

More languages may be added to this list over time.

[3]One might say that **`Aml/Base`** is a grandchild of Racket and Clojure.
[4]One might say that **`Aml/Core`** is a grandchild of OCaml, SML and F╳.
[5]One might say that **`Aml`** is a grandchild of **`Aml/Core`**, Haskell and Ada.

Part II

The Amlantis System Environment

Chapter 2

Languages

Amlantis System provides multiple interfaces to talk to it with. Those interfaces would be:

- A REPL console, where user's text input is immediately read, evaluated, printed and requested again.
- Source code, where user's text input is stored for reuse in general, usually to either do some scripting, or building modules and complex applications.
- Bytecode or native (machine) code files, which originate from textual source code.

More ways to interact with the Amlantis System might be possible in the future.[1]

Either way, for now, text-based interface to all Amlantis System tools require it to have a component that is able to read the text. We call them *Language Readers* (or shortly just *Readers* for the rest of this chapter). After reading the text, it has to be given meaning and transformed into a system of values that the Amlantis System understands by *Language Expanders* (or shortly just *Expanders* for the rest of this chapter). The result of that can be eventually evaluated.

2.1 Syntax Model

The syntax of programs in the Amlantis System is determined by the following process:

- A *read* pass, which comprise processing a text source port into a syntax object. Such syntax object does not contain any bindings. The read pass is implemented by Readers (§2.2).

[1]Some ideas for that would include natural speech interface (at least in English).

- An *expand* pass, where the syntax object from previous pass is transformed into a syntax object with full program information. All bindings are defined in such syntax object. The expand pass is implemented by Expanders (§2.3).

A fully-featured language provides implementation for both read and expand passes. Such implementation may as well reuse implementation of another language. A language may as well provide just its Reader, in case the resulting value can be expanded by the expander of `Aml/Base`. Alternatively, a language may provide just its Expander, providing bindings that may differ from those in `Aml/Base`.

2.1.1 Syntax Objects

A *syntax object* is a combination of an arbitrary Amlantis System value with lexical information, source-location information and syntax properties.[2]

Although syntax objects may contain any value, the expand pass (§2.1) will need to be able to handle the value and any value it itself contains. Therefore, it is recommended to limit the values embedded into syntax objects to the following types:

- Atoms:
  - Booleans.
  - Numbers.
  - Characters.
  - Strings.
  - Bytes and byte strings.
  - Symbols, Labels and Keywords.
  - Unit and Undefined.
- Basic collection and composite objects:
  - Tuples.
  - Pairs and Lists.
  - Vectors.
  - Maps.

[2]Racket, where inspiration for this mechanism comes from, also has tamper status within those syntax objects. We may decide to add that later.

2.2 Readers

Every Reader needs a source port to read from. Such a port could be console input for the REPL, a file on local filesystem, a TCP connection, socket, or just an in-memory string[3]. Reading program source starts with a top-most Reader, which we may call the *Root Reader*, which is implicitly (unless defined otherwise) `Aml/Base.Lang.Reader`.

Readers are required to implement two methods: `read` and `read-syntax`.

The `read` method must accept a source port as its only argument, and it can return any type of value.

The `read-syntax` method must accept a source name argument along with second argument of the source port, and it must return either the `System/IO.Eof` value, or a `System/Lang.Syntax_Object` value.

The Root Reader's `read-syntax` method is used on the top-most initial port.[4]

2.2.1 Root Reader Abilities

The Root Reader can recognise basic delimiters, such as all whitespace and newlines, which aid it in switching to another source code language. Given that the Root Reader is in fact `Aml/Base.Lang.Reader`, its abilities are not limited to these.

Delimited by those, it can recognise the following forms:

- Shebang, such as `#!/usr/bin/env aml`, to point to the entry-level Amlantis System tool.
- Reader switch, `#reader` *name* `...`, which specifies a reader to use for the following forms, and switches to it. The *name* should be a simple identifier.[5]
- Language switch, `#lang` *name* `...`, which expands right away into `#reader` *name*`.Lang.Reader ...`.[6] The *name* should be a simple identifier.[7]

2.3 Expanders

[3]More about strings later.
[4]Might be different in future, when we allow it to use `read` instead. That is not very useful now though.
[5]Specification for identifiers will be provided later.
[6]It may expand to other forms as well in certain order, due to be defined later.
[7]Specification for identifiers will be provided later.

Part III

The `Aml/Base` Language

Chapter 3

Syntax

The syntax of `Aml/Base` is based on Lisp-1, the so-called “S-expressions” (shorter for “symbolic expressions”[1]) and blatantly borrowing from languages such as Scheme, Racket or Clojure, while maintaining similarities with the `Aml` and `Aml/Core` languages.

3.1 The `Aml/Base.Lang.Reader`

3.1.1 Delimiters and Dispatch

Along with whitespace (as defined by Unicode “White_Space” property), the following characters are delimiters:

```
( ) [ ] { } " ' ` ~ ;
```

The `#` character has got a special meaning, determined by the following character or characters; see below for details.

After skipping whitespace, the `Aml/Base` reader dispatches based on the next character or characters in the source port this way:

[1]Which we will not abbreviate further as “sexprs”.

`(`	starts a pair or list; see Reading Pairs, Lists & Arrays.
`[`	starts a pair, list, or array; see Reading Pairs, Lists & Arrays.
`{`	starts a record; see Reading Records.
`)`	matches `(` or raises error.
`]`	matches `[` or raises error.
`}`	matches `{` or raises error.
`"`	starts a string; see Reading Strings.
`'`	starts a quote; see Reading Quotes.
`` ` ``	starts a symbol or a quote; see Reading Symbols and Reading Quotes.
`~`	starts an unquote; see Reading Quotes.
`^`	starts a metadata; see Reading Metadata.
`;`	starts a line comment; see Reading Comments.
`#y` or `#Y`	true; see Reading Booleans.
`#n` or `#N`	false; see Reading Booleans.
`#t` or `#T`	true; see Reading Booleans.
`#f` or `#F`	false; see Reading Booleans.
`#(`	starts a set; see Reading Sets.
`#[`	starts a vector; see Reading Vectors.
`#{`	starts a map; see Reading Maps.
`#\`	starts a character; see Reading Characters.
`#"`	starts a byte string; see Reading Strings.
`#%`	starts a symbol; see Reading Symbols.
`#:`	starts a keyword; see Reading Keywords.
`#'`	starts a syntax quote; see Reading Quotes.
`` #` ``	starts a syntax quasi-quote; see Reading Quotes.
`#~`	starts a syntax (possibly splicing) unquote; see Reading Quotes.
`##`	starts a parameterized read; see Reading Parameterized Reads.
`#e` or `#E`	starts a number; see Reading Numbers.
`#i` or `#I`	starts a number; see Reading Numbers.
`#<<`	starts a string; see Reading Strings.
`#|`	starts a block comment; see Reading Comments.
`#;` or `#_`	starts an S-expression comment; see Reading Comments.
“`#! `”	starts a line comment; note that the space is necessary; see Reading Comments.
`#!/`	starts a line comment; see Reading Comments.
`#!`	may start a reader extension use; see Reading via an Extension.
`#reader`	starts a reader extension use; see Reading via an Extension.
`#lang`	starts a reader extension use; see Reading via an Extension.
`#`*other*	starts a handler defined in current readtable or raises error.
otherwise	starts a symbol; see Reading Symbols.

Table 3.2: Table of Dispatches recognized by `Aml/Base.Lang.Reader`.

3.1.2 Reading Symbols

A sequence of source characters that does not start with any of the sequences defined in Delimiters and Dispatch is parsed as either a symbol or a number. The sequence “`#%`” also starts a symbol. A successful number parse takes precedence over symbol parse, see Reading Numbers.

A sequence of source characters that starts with “``” is parsed as a (verbatim) symbol until a matching “``” is found. If a “``” should appear inside the symbol, it has to be preceded by a “\”, and that character is the only character omitted from the resulting symbol.[2]

Symbols that consist entirely of “operator” characters (Unicode character classes Sm, So, except for `U+0060`) are treated specially, TBD[3].

Example 3.1.1 A few examples of parsing a symbol:

`Plum`	reads equal to `(string->symbol "Plum")`
`Plu#m`	reads equal to `(string->symbol "Plu#m")`
`Pl.um`	reads equal to `(string->symbol "Pl.um")`
`#%Plum`	reads equal to `(string->symbol "#%Plum")`
`string->symbol`	reads equal to `(string->symbol "string->symbol")`
``` ``the \`` is included`` ```	reads equal to ``` (string->symbol "the `` is included") ```

[2]This is why quasiquote must not be followed by another backtick.

[3]This is yet to be properly defined, but generally a property will be added to the form marking it “operator”.

3.1.3 Reading Numbers

A sequence of source characters that does not start with any of the sequences defined in Delimiters and Dispatch is parsed as a number when it matches the following grammar, in which *n* is a variable.

A number is optionally prefixed by an exactness specifying form, case-insensitively, `#e` (exact) or `#i` (inexact), which specifies its parsing as an exact or inexact number. Such form can be followed by whitespace, which is simply ignored.

In the grammar below, each literal lowercase letter stands for both itself and its uppercase form.

Grammar:

‹*number*$_n$›	::= ‹*prefix*$_n$› ‹*exact*$_n$› | ‹*inexact*$_n$›
‹*exact*$_n$›	::= ‹*exact-rational*$_n$› | ‹*exact-complex*$_n$›
‹*exact-rational*$_n$›	::= ‹*sign*›opt ‹*unsigned-rational*$_n$›
‹*unsigned-rational*$_n$›	::= ‹*unsigned-integer*$_n$›
	| ‹*unsigned-integer*$_n$› “/” ‹*unsigned-integer*$_n$›
‹*exact-integer*$_n$›	::= ‹*sign*›opt ‹*unsigned-integer*$_n$›
‹*unsigned-integer*$_n$›	::= ‹*digits*$_n$›+
‹*exact-complex*$_n$›	::= ‹*exact-rational*$_n$›opt ‹*sign*› ‹*unsigned-rational*$_n$› “`i`”
‹*inexact*$_n$›	::= ‹*inexact-real*$_n$› | ‹*inexact-complex*$_n$›
‹*inexact-real*$_n$›	::= ‹*sign*›opt ‹*inexact-simple*$_n$› (‹*exp*$_n$› ‹*exact-integer*$_n$›)opt
‹*inexact-unsigned*$_n$›	::= ‹*inexact-simple*$_n$›
‹*inexact-simple*$_n$›	::= ‹*digits*$_n$› “`.`” ‹*digits*$_n$›
	| ‹*digits*$_n$› “/” ‹*digits*$_n$›
‹*inexact-complex*$_n$›	::= ‹*inexact-real*$_n$›opt ‹*sign*› ‹*inexact-unsigned*$_n$› “`i`”
‹*sign*›	::= “+” | “–”
‹*general-number*$_n$›	‹*exactness*›opt ‹*number*$_n$›
‹*exactness*›	::= “`#e`” | “`#i`”

In the table below, we define the parameterized *digits*$_n$, *exp*$_n$ and *prefix*$_n$ for all supported *n*s.

Grammar:

‹*digit*$_{16}$›	::= ‹*digit*$_{12}$› | “`c`” | “`d`” | “`e`” | “`f`”
‹*prefix*$_{16}$›	::= “`0x`”
‹*digit*$_{12}$›	::= ‹*digit*$_{10}$› | “`a`” | “`b`”
‹*prefix*$_{12}$›	::= “`0d`”
‹*digit*$_{10}$›	::= ‹*digit*$_8$› | “`8`” | “`9`”
‹*prefix*$_{10}$›	::= empty sequence
‹*digit*$_8$›	::= ‹*digit*$_2$› | “`2`” | “`3`” | “`4`” | “`5`” | “`6`” | “`7`”
‹*prefix*$_8$›	::= “`0o`”
‹*digit*$_2$›	::= “`0`” | “`1`”
‹*prefix*$_2$›	::= “`0b`”

$$\begin{aligned}
\langle \mathit{exp}_{16} \rangle \quad &::= \text{“}\texttt{p}\text{”} \\
\langle \mathit{exp}_{12} \rangle \quad &::= \langle \mathit{exp}_{10} \rangle \\
\langle \mathit{exp}_{10} \rangle \quad &::= \text{“}\texttt{e}\text{”} \\
\langle \mathit{exp}_{8} \rangle \quad &::= \langle \mathit{exp}_{10} \rangle \\
\langle \mathit{exp}_{2} \rangle \quad &::= \langle \mathit{exp}_{10} \rangle \\
\langle \mathit{digits}_{n} \rangle \quad &::= \langle \mathit{digit}_{n} \rangle\ (\text{“}\texttt{\_}\text{”}\text{opt}\ \langle \mathit{digit}_{n} \rangle)^{*}
\end{aligned}$$

In the grammar below, we define the special case of sexagesimal numbers. Some of the rules here override those for non-sexagesimal numbers.

Grammar:

$$\begin{aligned}
\langle \mathit{digit}_{60} \rangle \quad &::= (\text{“}\texttt{0}\text{”} \mid \text{“}\texttt{1}\text{”} \mid \text{“}\texttt{2}\text{”} \mid \text{“}\texttt{3}\text{”} \mid \text{“}\texttt{4}\text{”} \mid \text{“}\texttt{5}\text{”})\text{opt}\ \langle \mathit{digit}_{10} \rangle \\
\langle \mathit{prefix}_{60} \rangle \quad &::= \text{“}\texttt{0s}\text{”} \\
\langle \mathit{exp}_{60} \rangle \quad &::= \langle \mathit{exp}_{10} \rangle \\
\langle \mathit{digits}_{60} \rangle \quad &::= \langle \mathit{digit}_{60} \rangle\ (\text{“}\texttt{,}\text{”}\ \langle \mathit{digit}_{60} \rangle)^{*} \\
\langle \mathit{inexact\text{-}simple}_{60} \rangle \quad &::= \langle \mathit{digits}_{60} \rangle\ \text{“}\texttt{;}\text{”}\ \langle \mathit{digits}_{60} \rangle
\end{aligned}$$

3.1.4 Reading Booleans

The forms “`#yes`”, “`#y`”, “`#true`”, “`#t`” (case-insensitive) followed by a delimiter are the complete input syntax for the `System.Boolean.Yes` constant, and “`#no`”, “`#n`”, “`#false`”, “`#f`” (case-insensitive) followed by a delimiter are the complete input syntax for the `System.Boolean.No` constant.

3.1.5 Reading Pairs, Lists & Arrays

Sequences of input characters that start with “`(`” or “`[`” begin parsing characters to form a pair or a list.[4]

Parsing a pair or list form entails recursively parsing sub-forms until a matching “`(`” or “`[`” (respectively, these are not interchangeable) is found. A delimited “`.`” is handled specially.

If there is no delimited “`.`” among the forms between parentheses, then a list is formed, containing the results of the recursive reads, in the same order as they appear in the source text.

If there is exactly one delimited “`.`”, it's not the first and also not the last sub-form, and there are two other sub-forms (the first and the last one), then a pair is formed. More generally, if there are more sub-forms before the delimited “`.`”, a pair is formed with the last of them, and recursively, a pair is formed with the preceding sub-form and the previously formed pair, thus forming nested pairs.

[4]Note that in Lisp languages, not excluding `Aml/Base`, evaluating list forms does not necessarily produce a list - instead, many list forms are a function application form. To get a list data structure, one either needs to construct it with function applications, or quote the list form.

If there are exactly two delimited “`.`”, none of which is the first and the last sub-form, and there is at least one other sub-form in between them, then the middle sub-form is put first in the resulting list, then the sub-forms before the first delimited “`.`”, and finally the sub-forms after the second delimited “`.`”.

If the sequences of input characters start with “`[|`” and end with “`|]`”, then an array is formed instead of a list.

Example 3.1.2 A few examples of parsing lists, pairs and arrays:

`()`	reads equal to `(list)`
`(1 2 3)`	reads equal to `(list 1 2 3)`
`[1 2 3]`	reads equal to `(list 1 2 3)`
`[|1 2 3|]`	reads equal to `(array 1 2 3)`
`(1 2 (3))`	reads equal to `(list 1 2 (list 3))`
`(1 . 2)`	reads equal to `(pair 1 2)`
`(1 2 3 . 4)`	reads equal to `(pair 1 (pair 2 (pair 3 4)))`
`(1 2 . ())`	reads equal to `(pair 1 (pair 2 (list)))`
`(1 . 2 . 3)`	reads equal to `(list 2 1 3)`

3.1.6 Reading Records

Sequences of input characters that start with “`{`” begin parsing characters to form a record.

Parsing a record entails recursively parsing sub-forms, until a matching “`}`” is found.

There are 3 different possible orderings of sub-forms:

- Pairs of sub-forms, where first item is a record field, second is any form.
- First sub-form is any form, followed by keyword “`#:with`”, followed by pairs of sub-forms, where first item is a record field, second is any form. This sequence of sub-forms can then optionally be followed by a keyword “`#:without`”, followed by record field sub-forms.
- First sub-form is any form, followed by keyword “`#:without`”, followed by record field sub-forms.

Alternatively, whenever a pair of two sub-forms is expected, a pair form can be used, see Reading Pairs, Lists & Arrays.[5]

A record field is such a form *f* for which “`(record-field-name?` *f*`)`” returns “`#yes`”.[6]

If the sequence of input characters start with “`{|`” and end with “`|}`”, then an anonymous record is formed instead of a (regular) record.

[5]This is the basic form of record expressions.

[6]Record fields are usually symbols, in case of ambiguities, such symbol may refer to a fully or partially qualified record field.

Example 3.1.3 A few examples of parsing records:

```
{ Fraction.numerator 17 denominator 3 }
{ fr #:with numerator 16 }
{ fr #:with numerator 16 #:without denominator }
{ fr #:without denominator }
{ (Fraction.numerator . 17) (denominator . 3) }
```

3.1.7 Reading Strings

Sequences of input characters that start with "`"`" begin parsing characters to form a string, until another "`"`" is found (that is not escaped by a single "\").

Within a string sequence, the following "escape sequences" are recognized:

- "`\'`": single quote
- "`\"`": double-quotes (does not terminate the string form)
- "`\\`": backslash (the second is not an escaping backslash)
- "`\0`": binary zero (NUL, `U+0000`)
- "`\a`": BEL (alarm) character (`U+0007`)
- "`\b`": backspace (`U+0008`)
- "`\e`": escape (`U+001B`)
- "`\f`": form feed (`U+000C`)
- "`\n`": end-of-line (`U+000A`)
- "`\r`": carriage return (`U+000D`)
- "`\t`": horizontal tab (character tabulation, `U+0009`)
- "`\v`": vertical tab (line tabulation, `U+000B`)
- "`\x`*nn*": Unicode for the hexadecimal number specified by *nn*, where each *n* is a ‹*digit*$_{16}$›.
- "`\u`*nnnn*": Unicode for the hexadecimal number specified by *nnnn*, where each *n* is a ‹*digit*$_{16}$›, the character `U+`*nnnn*.
- "`\U`*nnnnnnnn*": Unicode for the hexadecimal number specified by *nnnnnnnn*, where each *n* is a ‹*digit*$_{16}$›, the character `U+`*nnnnnnnn*.
- "`\u{`$n_1 \ldots n_m$`}`": Unicode for the hexadecimal number specified by $n_1 \ldots n_m$, where each n_i is a ‹*digit*$_{16}$› and $m < 8$, the character `U+`$n_1 \ldots n_m$.

- “`\`‹*newline*›”: elided, where ‹*newline*› is either an end-of-line, carriage return, or carriage return end-of-line combination. This sequence is not added to the string form.

When the sequence of input characters starts with “`#"`” instead of just “`"`”, the parsed string form is a byte string.

Sequences of input characters that start with “`#<<`” begin parsing characters to form a string, in this form called *here string*. The characters following “`#<<`” until a newline character define a terminator for the string. The content of the resulting string form includes all characters between the line containing “`#<<`” and a line whose entire content is the specified terminator (possibly preceded by whitespace). The first and last line of such form are not included in the resulting string form. No escape sequences are recognized in *here strings*.

3.1.8 Reading Characters

Sequences of input characters that start with “`#\`” begin parsing characters to form a character constant. The following forms are recognized:

- “`#\nul`”, “`#\null`” or “`#\0`”: NUL (ASCII 0), the next character must not be alphabetic.
- “`#\backspace`”: backspace (`U+0008`), the next character must not be alphabetic.
- “`#\tab`” or “`#\htab`”: horizontal tab (`U+0009`), the next character must not be alphabetic.
- “`#\newline`” or “`#\linefeed`”: end-of-line (`U+000A`), the next character must not be alphabetic.
- “`#\vtab`”: vertical tab (`U+000B`), the next character must not be alphabetic.
- “`#\page`”: form feed (page break, `U+000C`), the next character must not be alphabetic.
- “`#\return`”: carriage return (`U+000D`), the next character must not be alphabetic.
- “`#\space`”: space (`U+0020`), the next character must not be alphabetic.
- “`#\delete`” or “`#\rubout`”: delete (`U+007F`), the next character must not be alphabetic.
- “`#\quote`”, “`#\squote`” or “`#\'''`”: single quote (apostrophe, `U+0027`), the next character must not be alphabetic after the former two forms.
- “`#\x`*nn*”: Unicode for the hexadecimal number specified by *nn*, where each *n* is a ‹*digit*$_{16}$›.
- “`#\u`*nnnn*”: Unicode for the hexadecimal number specified by *nnnn*, where each *n* is a ‹*digit*$_{16}$›, the character `U+`*nnnn*.
- “`#\U`*nnnnnnnn*”: Unicode for the hexadecimal number specified by *nnnnnnnn*, where each *n* is a ‹*digit*$_{16}$›, the character `U+`*nnnnnnnn*.

- "`#\u{`$n_1 \ldots n_m$`}`": Unicode for the hexadecimal number specified by $n_1 \ldots n_m$, where each n_i is a ‹*digit*$_{16}$› and $m < 8$, the character `U+`$n_1 \ldots n_m$.
- "`#\'`*c*`'`": the character *c*, which must be a single Unicode codepoint.

3.1.9 Reading Quotes

Upon reading a "`'`", the reader recursively reads next form a new list is formed containing the symbol "`#'quote`" and the following form.

The reader recognizes a few other forms in the same way:

`'`	adds `'quote`
`` ` ``	adds `'quasiquote`
`~`	adds `'unquote`
`~@`	adds `'unquote-splicing`
`#'`	adds `'syntax-quote`
``#` ``	adds `'syntax-quasiquote`
`#~`	adds `'syntax-unquote`
`#~@`	adds `'syntax-unquote-splicing`

Example 3.1.4 A few examples of parsing quotes:

`'(apple)`	reads equal to `(list 'quote 'apple)`
`` `(1 #,2) ``	reads equal to `(list 'quasiquote (list 1 (list 'unquote 2)))`

3.1.10 Reading Comments

A "`;`" starts a line comment. When the reader encounters it, it discards all characters following it until the next new line.

A "`#|`" starts a nestable block comment. When the reader encounters it, it discards all characters following it until a closing "`|#`". When discarding characters within the nestable block comment, another "`#|`" begins recursive discarding, so that pairs of matching "`#|`" and "`|#`" can be nested (hence nestable block comments).

A "`#;`" or a "`#_`" starts an S-expression comment. When the reader encounters it, it recursively reads one following form, and discards it.

A "`#! `" (a "`#!`" followed by a whitespace) or "`#!/`" starts a line comment, much like "`;`" does, except if the line ends with "\", the discarding continues to the next line.

Example 3.1.5 A few examples of comments:

`; comment`	reads equal to *nothing*
`#| a |# 1`	reads equal to `1`
`#| #| a |# 1 |# 2`	reads equal to `2`
`#_1 2`	reads equal to `2`
`#!/usr/bin/env amlc`	reads equal to *nothing*
`#! /usr/bin/env amlc`	reads equal to *nothing*

3.1.11 Reading Vectors

The sequence “`#[`” starts a vector. Following sub-forms are recursively read until a matching “`]`” is found, same as for lists.

Example 3.1.6 An example of a vector:

`#[1 2 3]`	reads equal to `(vector 1 2 3)`

3.1.12 Reading Sets

The sequence “`#(`” starts a set. Following sub-forms are recursively read until a matching “`)`” is found, same as for lists. The sequence “`#(|`” starts a bag. Following sub-forms are recursively read until a matching “`|)`” is found.

Example 3.1.7 Examples of sets:

`#(1 2 3)`	reads equal to `(hash-set 1 2 3)`
`#(|1 2 3|)`	reads equal to `(hash-bag 1 2 3)`

3.1.13 Reading Maps

The sequence “`#{`” starts a map. Following sub-forms are recursively read in pairs until a matching “`}`” is found, similar to records. Sub-form pair can also be explicit pairs, same as for records. The sequence “`#(|`” starts a multi-map. Following sub-forms are recursively read until a matching “`|)`” is found, same as for maps.

Example 3.1.8 Examples of maps:

`#{"a" 5 "b" 7}`	reads equal to `(hash-map ("a" . 5) ("b" . 7))`
`#{|"a" 5 "b" 7|}`	reads equal to `(hash-multi-map ("a" . 5) ("b" . 7))`

3.1.14 Reading Keywords

The sequence "`#:`" starts a keyword. The parsing is the same as for a symbol preceded by "`#%`", except for this initial sequence, and the part after this initial sequence is never parsed as a number.[7]

Example 3.1.9 A few examples of parsing a keyword:

`#:Plum`	reads equal to `(string->keyword "Plum")`
`#:42`	reads equal to `(string->keyword "42")`

3.1.15 Reading Metadata

The sequence "`^`" starts a metadata. When the reader encounters it, it recursively reads next form, which is expected to be a map without the leading "`#`", or a plain symbol. In case of a plain symbol *s*, the metadata is constructed as `#{(#:metadata . `*s*`)}`. After the metadata *m* is read, the reader attaches it to the next recursively read form *f*: `(with-metadata `*f m*`)`. If there is no non-whitespace read after the first "`^`", an ordinary symbol is parsed with the contents being the "`^`".

Example 3.1.10 Examples of parsing metadata:

`^{#:a 1 #:b 2} [1 2]`	reads equal to `(with-metadata [1 2] #{#:a 1 #:b 2})`
`^str x`	reads equal to `(with-metadata x #{#:metadata 'str})`
`(^ x)`	reads equal to `(list (string->symbol "^") x)`

3.1.16 Reading Parameterized Reads

(TBD)

3.1.17 Reading via an Extension

When the reader dispatches on the `#reader` form, it recursively applies another reader to the current source port.

First, the reader recursively reads the next datum after `#reader`, and uses it as path to the another reader. Such reader is then loaded, and `read` is used when this reader is in `read` mode, or else, `read-syntax` is used when this reader is in `read-syntax` mode.

The `#lang` reader form is similar. It must be followed by a single whitespace character (preferably a single space, ASCII 32), and then followed by an identifier form. The complete form

[7]Once we verify that it does not interfere with the rest of the language, keywords could be prefixed just with a double colon, without the leading hash.

is then terminated by a new line, or end-of-file. A sequence `#lang` *name* is equivalent to `#reader` *name*`.Lang.Reader`.

For compatibility with e.g. R^{6}RS, `#!` is an alias for `#lang` followed by a space when it is followed by alphanumeric ASCII, `+`, `-` or `_`.

3.1.17.1 S-expression Reader Language

```
#lang s-exp path
```

3.1.17.2 Chaining Reader Language

```
#lang reader path
```

Part IV

The `Aml/Bootstrap` Language

Chapter 4

Syntax

The syntax of `Aml/Bootstrap` is a subset of `Aml/Base`. It is not a programming language intended for direct programmer's use, instead, it is meant to be handled by programs, but still readable by programmers. Also, it is meant to bridge the gap between a world where `Aml` does not exist and a world where it begins to exist; being interpreted by code written in a non-`Aml` language. `Aml/Bootstrap` code is used to implement `Aml/Base`, and to interact with the type system, thus bootstrapping any program written in the `Aml` languages.

Part V

The Amlantis System Tools

Chapter 5

The Top-Level System or REPL – `aml`

This chapter will describe the top-level system for the Amlantis System, which serves as universal entry point for running programs designed for this system.

Without input files specified in arguments, it permits interactive use of the Amlantis System through a mechanism known as read-eval-print loop (REPL). With this mode, the system repeatedly reads user's input, evaluates it with the context of every previous input of the same REPL session, prints the inferred type of the result of such evaluation along with the text representation of the result, if any, and repeats.

Chapter 6

The Compiler – `amlc`

In this chapter, we're going to describe the Amlantis System Compiler **`amlc`**, which compiles source code files to bytecode files. The compilation process involves reading through the input source files, expanding them by evaluating the read programs by using the specified language, and serializing the type system's state along with modules' state to bytecode files, which are then to be run by **`amlrun`** (§7) or **`aml`** (§5).

Chapter 7

The Runtime System – `amlrun`

This chapter will describe the **`amlrun`** tool, which reads files output by **`amlc`** (§6) and executes a specified executable[1].

[1]Due to be described later, but those could be e.g. a specific function, or a script file.

Chapter 8

The Debugger – `amldebug`

In this chapter, we'll describe the Amlantis System's own debugger tool.

Chapter 9

The Profiler – `amlprof`

This chapter will describe the Amlantis System's profiling tool.

Chapter 10

The Builder – `amlbuild`

In this chapter, we'll describe the Amlantis System's builder tool.

Chapter 11

The Book Workspace – `amlbook`

This chapter will describe the Amlantis System's "Book Workspace" tool, which is a planned addition to the whole toolchain and supposed to be accompanied by a GUI. The purpose will be to work with source texts that intertwine program source code in snippets with rich text documents, and ability to selectively execute these program source codes, seeing results right below each snippet.