The Alonzo Functional Programming Language

John D. Ramsdell*
The MITRE Corporation
Bedford, MA 01730

Abstract

Alonzo is a programming language for specifying an output stream of characters as a function of a given input stream of characters. It is a non-strict language based on the untyped λ-calculus, and it has been enriched by adding syntax for local bindings and mutual recursion. Primitive data and their operators have been included along with strict vectors.

Contents

1 Design Goals 1

2 Basic Concepts 2
- 2.1 True and False . . . 2
- 2.2 Pairs and Lists . . . 2

3 Expressions 2
- 3.1 Variable Reference . . . 2
- 3.2 Literal Expressions . . . 2
- 3.3 The List Expression . . . 2
- 3.4 The Vector Expression . . . 2
- 3.5 Lambda Expressions . . . 2
- 3.6 Parenthesized Expressions . . . 3

4 Program Structure 3

5 Standard Functions 3
- 5.1 Type Predicates . . . 3
- 5.2 Integer Operators . . . 3
- 5.3 Character Operators . . . 4
- 5.4 Vector Operators . . . 4

6 Formal Definition 4
- 6.1 Syntax . . . 5
- 6.2 Semantics . . . 5

A Editing Alonzo Programs 6

*This work was supported by the MITRE Sponsored Research Program.

1 Design Goals

Alonzo was designed for programming multicomputers [AS88]. It was designed under the assumption that communication to and from the multicomputer is slow, so that communication is limited to streams of characters.

Another design goal was to display a useful functional programming language in which both the syntax and the semantics are simple and regular, and the syntax closely reflects the semantics. This is in contrast with many new functional programming languages, such as Haskell [HW88], which were designed to be intuitively obvious to a programmer with a mathematical background at the risk of simplicity. An example of a useful feature of Alonzo syntax concerns the ease in which definitions may be introduced into expressions. All parenthesized expressions may begin with local definitions. A benefit of having simple syntax and semantics is that it was straightforward to implement a compiler for Alonzo in Alonzo.

The language was designed to allow a simple embedding in the Scheme dialect of Lisp [RC86]. Many of the standard functions have been taken from Scheme along with much of Scheme's syntactic conventions. Even the form of this report was greatly influenced by the Scheme report [RC86].

There exists two implementations of Alonzo. A Scheme implementation was used to bootstrap an implementation built on a fast reduction machine written in ANSI C. An existing Alonzo compiler image for the C machine allows the use of Alonzo without a Scheme system. Both implementations use Turner's combinators [Tur79], however, both provide facilities for linking supercombinators [Hug82] or other optimized combinators.

A major challenge in producing a successful parallel implementation of Alonzo will be finding an algorithm which discovers much of the potential opportunities for parallel reduction of Alonzo expres-

sions. If we limit ourselves to the reduction of expressions known to be needed, we require a strictness analysis of a dynamically typed variant of the untyped λ-calculus. One approach to such a task is to type fragments of Alonzo programs, and use existing strictness analysis algorithms [BHA86] limited to just these fragments. The interest in strictness analysis is one important reason Alonzo has a formal definition.

2 Basic Concepts

The lexical conventions used in writing Alonzo programs are nearly identical to Scheme's conventions, which, in turn, are much like most dialects of Lisp. The notation for strings, whitespace, comments, and quotation is as is Scheme's. Identifier notation is like Scheme's, except the list of syntactic keywords which may not be used as variables has been changed to:

`define`, `lambda`, `list`, `quote`, and **`vector`.**

2.1 True and False

The boolean values are the functions which select between two alternatives, with true being associated with selecting the first alternative and false selecting the second alternative.

```
(define (true x y) x)
(define (false x y) y)
```

2.2 Pairs and Lists

The pairing function was chosen to be non-strict, so its definition is the obvious one [Bar84, pages 132–135]. The pairing of two expressions is the function which applies its argument to the first expression, and applies that result to the second expression.

```
(define (pair x y)
   (lambda (z) z x y))
```

Lists are built out of pairs using the special value, **null**, to mark the empty list.

3 Expressions

Computation is associated with finding a value associated with an Alonzo expression. That process will be called *reduction*, and an expression is said to *reduce to* ($\Longrightarrow$) a value. Some expressions will be defined by giving another equivalent replacement expression. In that case, the original expression is said to *expand to* ($\longrightarrow$) the replacement expression. The notation $\langle \text{thing}_1 \rangle \ldots$ means zero or more occurrences of ⟨thing⟩.

3.1 Variable Reference

⟨variable⟩

Naming an identifier reduces to the value bound to the identifier. A reference to an unbound identifier is an error.

3.2 Literal Expressions

`(quote` ⟨datum⟩`)` | `'`⟨datum⟩ | ⟨constant⟩

`(quote` ⟨datum⟩`)` reduces to ⟨datum⟩, which is either an integer, a character, the empty list, or the pairing function applied to two ⟨datum⟩'s. A ⟨string⟩ is a list of characters, and a ⟨symbol⟩ is a list of characters in which all upper case characters have been translated to lower case. A parenthesized quoted expression is converted to a ⟨datum⟩ using the pairing function.

```
(quote ())        ⟹  null.
(quote 'Ab)       ⟶  (list #\a #\b).
(quote (0 1 2))   ⟶  (list 0 1 2).
```

`(quote` ⟨datum⟩`)` may be abbreviated as `'`⟨datum⟩. The two notations are equivalent in all respects. Integers, characters, and strings reduce to themselves and need not be quoted.

3.3 The List Expression

`(list` $\langle \text{expression}_1 \rangle \ldots$`)`

The `list` syntax expands to a **null** terminated list containing as many expressions as given. The list is linked using the pairing function of Section 2.2

3.4 The Vector Expression

`(vector` $\langle \text{expression}_1 \rangle \ldots \langle \text{expression}_n \rangle$`)`

The `vector` syntax expands to the application of a vector constructor to the n expressions. The constructor is the result of applying the `make-vector` function described in Section 5.4 to n.

3.5 Lambda Expressions

`(lambda (`$\langle \text{variable}_1 \rangle \ldots$`)` ⟨body⟩`)`

A lambda expression of one variable reduces to a function. As in the λ-calculus, the application of the function to an expression is equivalent to substituting the expression for all occurrences of the variable in the body of the function. As Alonzo is a non-strict language, the actual argument given to a function need not be reduced.

A lambda expression with more than one formal parameter expands to a lambda expression of the first formal parameter in which the body is a lambda expression of the rest of the formal parameters. A lambda expression with an empty formal parameter list is equivalent to a parenthesized expression.

3.6 Parenthesized Expressions

(⟨body⟩)

A ⟨body⟩ is a possibly empty sequence of definitions followed by an expression or an application. Juxtaposition of two expressions gives an application. The left expression must reduce to a function, and the right expression will be given to the function as its actual argument. Application is left associative.

`(a b c)` $\longrightarrow$ `((a b) c)`.

Definitions bind expressions to variables. The scope of the bound variables is the entire ⟨body⟩ in which the variables are defined. Mutual recursive function can be defined, as the variables will be visible to the expressions defining all other variables. It is an error if the definitions at the begining of a body define the same variable more than once.

The syntax of a definition follows.

`(define` ⟨variable⟩ ⟨body⟩`)`
`(define (`⟨variable$_0$⟩ ⟨variable$_1$⟩ ...`)` ⟨body⟩`)`

In the first instance, the variable is bound to the expression (⟨body⟩). In the second instance, ⟨variable$_0$⟩ is bound the the expression

`(lambda (`⟨variable$_1$⟩ ...`)` ⟨body⟩`)`.

As an example, another equivalent way of writing the pairing function of Section 2.2 is

`(define (pair x y z) z x y)`.

4 Program Structure

An Alonzo program maps an input stream of characters to an output stream of characters. A stream of characters is like a list of characters in that the glue used to hold them together is the pairing function of Section 2.2. Unlike lists, streams never terminate. That is, a stream is always a function which can be applied to a function of the form

`(lambda (c s)` ⟨body⟩`)`,

and `c` will be bound to an expression which reduces to a character, and `s` will be bound to another stream.

A program is syntactically a ⟨body⟩, i.e. a sequence of definitions followed by an expression or an application. The output stream is defined to be the result of applying the expression (⟨body⟩) to the input stream. Given an output stream σ, the output is computed as follows:

1. Let c be the value associated with reducing the expression `(`σ `(lambda (c s) c))`.
2. If c is not a character, an error must be signaled.
3. Print c.
4. Set σ to `(`σ `(lambda (c s) s))`.
5. Go to step 1.

The reduction in step 1 may not pause if the initial segment of the input stream currently available is sufficient to allow the computation of c.

5 Standard Functions

Standard Alonzo functions are patterned after many essential Scheme procedures defined in [RC86].

5.1 Type Predicates

Every object in Alonzo is either an integer, a character, the null list, or a function. Some functions are also vectors.

`int?` Reduces one argument and returns true if the argument is an integer, otherwise false is returned.

`char?` Reduces one argument and returns true if the argument is a character, otherwise false is returned.

`null?` Reduces one argument and returns true if the argument is the empty list, otherwise false is returned.

`func?` Reduces one argument and returns true if the argument is a function, otherwise false is returned.

`vector?` Reduces one argument and returns true if the argument is a vector, otherwise false is returned. All vectors are functions.

5.2 Integer Operators

It is an error to apply any function in this subsection to non-integral arguments.

`zero?, even?, odd?` Each reduces one argument and returns true if the argument is zero, even, or odd, respectively, otherwise false is returned for any other integer.

`+, -, *, max, min` Each reduces two arguments and returns the sum, difference, product, maximum, or minimum, respectively, if both arguments are integers.

`quotient, remainder` Each reduces two arguments and returns the quotient or remainder, respectively, such that for integer `i` and `j` with $j \neq 0$

```
(= i (+ (* j (quotient i j))
        (remainder i j))) ⟹ true.
```

`=, <, >, <=, >=` Each reduces two arguments and returns the appropriate boolean value if the first argument is equal to, less than, greater than, less or equal to, or greater or equal to, respectively, the second argument.

`abs` Reduces one argument and returns the magnitude value of the argument.

`int->char` Reduces one argument and returns the character corresponding to the integer. **`int->char`** is the inverse of **`char->int`**. Application of **`int->char`** to an integer which has no corresponding character is an error.

5.3 Character Operators

It is an error to apply any functions in this subsection to non-characters.

`char=?, char<?, char>?, char<=?, char>=?` Each reduces two arguments and returns the appropriate boolean value if the first argument is equal to, less than, greater than, less or equal to, or greater or equal to, respectively, the second argument. These functions impose a total ordering on the set of characters. It is guaranteed that under this ordering:

- The upper case characters are in order.
- The lower case characters are in order.
- The digits are in order.
- Either all the digits precede all the upper case letters, or vice versa.
- Either all the digits precede all the lower case letters, or vice versa.

`char->int` Reduces one argument and returns the integer representation of a character such that the integer ordering is consistent with the character ordering. That is, for any characters `c` and `d`, `(char=<? c d)` has the same boolean value as

```
(=< (char->int c) (char->int d))
```

`quit?` Reduces one argument and returns true if that character is the quit character. Adding the quit character to the output stream may have some implementation defined behavior, such as saving an image of an Alonzo program or simply exiting.

⟨body⟩ ::= ⟨definitions⟩ ⟨application⟩.
⟨definitions⟩ ::= **`(define`** ⟨pattern⟩ ⟨body⟩)*.
⟨application⟩ ::= ⟨expression⟩
| ⟨application⟩ ⟨expression⟩.
⟨expression⟩ ::= ⟨variable⟩ | ⟨constant⟩
| '⟨datum⟩
| **`(quote`** ⟨datum⟩)
| **`(list`** ⟨expression⟩*)
| **`(vector`** ⟨expression⟩*)
| **`(lambda`** (⟨variable⟩*) ⟨body⟩)
| (⟨body⟩).
⟨pattern⟩ ::= ⟨variable⟩
| (⟨pattern⟩ ⟨variable⟩*).
⟨constant⟩ ::= ⟨integer⟩ | ⟨character⟩
| ⟨string⟩.
⟨datum⟩ ::= ⟨constant⟩ | ⟨symbol⟩
| (⟨list⟩).
⟨list⟩ ::= ⟨datum⟩*
| ⟨datum⟩* . ⟨datum⟩.

Figure 1: Alonzo Syntax

5.4 Vector Operators

A vector is a function which when applied to an integer index i, returns the i-th element of the vector. Indexing is zero-based, so legal indices for a vector of length n satisfies $0 \leq i < n$. The application of a vector to an illegal index is an error.

`make-vector` Reduces one argument and if it is the non-negative integer n, creates a function which constructs a vector of length n. If the vector constructor is applied to n arguments, it reduces each argument, and returns an n-vector containing the arguments. The application of **`make-vector`** to other than a non-negative integer is an error.

`vector-length` Reduces one argument and if it is a vector, returns its length. Application of this function to a non-vector is an error.

6 Formal Definition

This section provides formal descriptions of what has already been described informally in previous sections.

6.1 Syntax

The complete grammar for Alonzo is given in Figure 1. The start symbol is ⟨body⟩. The terminal symbols include ⟨variable⟩, ⟨integer⟩, ⟨character⟩, ⟨symbol⟩, and ⟨string⟩. The grammer is extended BNF in that ⟨thing⟩* means zero or more occurrences of ⟨thing⟩.

6.2 Semantics

This section provides a formal denotational semantics for the primative expressions in Alonzo and a few selected standard functions. The concepts and notation used here are described in [Sto77].

6.2.1 Abstract Syntax

$C \in \text{Con}$	integers, characters, and '()
$I \in \text{Ide}$	identifiers (variables)
$E \in \text{Exp}$	expressions
$A \in \text{App}$	applications
$B \in \text{Bod}$	bodies
$P \in \text{Pgm} = \text{Bod}$	programs

E ::= C | I | (**lambda** (I) B) | (B).
A ::= E | A E.
B ::= (**define** I B)* A.
P ::= B.

6.2.2 Domain Equations

C	primitive constants
$H \subset C$	characters and $\perp$
$\rho \in U = \text{Ide} \mapsto E$	environments
$\alpha \in A = E \mapsto K \mapsto H$	applicable values
$\nu \in V = C + A$	values of computations
$\epsilon \in E = K \mapsto H$	values of expressions
$\kappa \in K = V \mapsto H$	continuations
$\sigma \in S = H \times S$	character streams

6.2.3 Semantic Functions

$\mathcal{C} : \text{Con} \mapsto V$
$\mathcal{E} : \text{Exp} \mapsto U \mapsto K \mapsto H$
$\mathcal{A} : \text{App} \mapsto U \mapsto K \mapsto H$
$\mathcal{B} : \text{Bod} \mapsto U \mapsto K \mapsto H$
$\mathcal{P} : \text{Pgm} \mapsto S \mapsto S$

The definition of $\mathcal{C}$ has been deliberately omitted.

$\mathcal{E}[\![\text{C}]\!] = \lambda\rho\kappa.\kappa(\mathcal{C}[\![\text{C}]\!])$
$\mathcal{E}[\![\text{I}]\!] = \lambda\rho\kappa.\kappa(\rho[\![\text{I}]\!])$
$\mathcal{E}[\![(\texttt{lambda}\ (\text{I})\ \text{B})]\!] = \lambda\rho.\mathit{funct}\,\lambda\epsilon.\mathcal{B}[\![\text{B}]\!]\rho[\epsilon/\text{I}]$
$\mathcal{E}[\![(\text{B})]\!] = \mathcal{B}[\![\text{B}]\!]$
$\mathit{funct} = \lambda\alpha\kappa.\kappa(\alpha \text{ in } V)$

$\mathcal{A}[\![\text{E}]\!] = \mathcal{E}[\![\text{E}]\!]$
$\mathcal{A}[\![\text{A E}]\!] = \lambda\rho.\mathit{apply}(\mathcal{A}[\![\text{A}]\!]\rho)(\mathcal{E}[\![\text{E}]\!]\rho)$
$\mathit{apply} = \lambda\epsilon_0\epsilon_1\kappa.\epsilon_0\lambda\nu.\nu \in A \rightarrow (\nu \mid A)\epsilon_1\kappa, \kappa\perp$

$\mathcal{B}[\![\text{A}]\!] = \mathcal{A}[\![\text{A}]\!]$
$\mathcal{B}[\![(\texttt{define}\ \text{I B})\ \text{A}]\!] = \lambda\rho.\mathcal{A}[\![\text{A}]\!]\rho[\mathit{fix}(\lambda\epsilon.\mathcal{B}[\![\text{B}]\!]\rho[\epsilon/\text{I}])/\text{I}]$
$\mathcal{B}[\![(\texttt{define}\ \text{I}_1\ \text{B}_1)\ldots(\texttt{define}\ \text{I}_n\ \text{B}_n)\ \text{A}]\!] =$
$\lambda\rho.\mathit{tie}_n$
$(\lambda\epsilon_1\ldots\epsilon_n\omega.\omega$
$\mathcal{B}[\![\text{B}_1]\!]\rho[\epsilon_1,\ldots,\epsilon_n/\text{I}_1,\ldots,\text{I}_n]$
$\vdots$
$\mathcal{B}[\![\text{B}_n]\!]\rho[\epsilon_1,\ldots,\epsilon_n/\text{I}_1,\ldots,\text{I}_n])$
$\lambda\epsilon_1\ldots\epsilon_n.\mathcal{A}[\![\text{A}]\!]\rho[\epsilon_1,\ldots,\epsilon_n/\text{I}_1,\ldots,\text{I}_n]$
$\mathit{tie}_n = \lambda\alpha.\mathit{fix}(\lambda\omega.\alpha(\omega\lambda\epsilon_1\ldots\epsilon_n.\epsilon_1)\ldots(\omega\lambda\epsilon_1\ldots\epsilon_n.\epsilon_n))$
$\text{I}_1,\ldots,\text{I}_n$ must be unique identifiers.

ρ_0 is an environment containing only the bindings for the standard functions.
$\mathcal{P}[\![\text{P}]\!] = \lambda\sigma.\mathit{apply}(\mathcal{B}[\![\text{B}]\!]\rho_0)(\mathit{in}\,\sigma)\mathit{out}$
$\mathit{in} = \mathit{fix}(\lambda\psi\sigma.\sigma = \perp \rightarrow \lambda\kappa.\kappa\perp,$
$\mathit{funct}\,\lambda\epsilon.\mathit{apply}(\mathit{apply}\,\epsilon\,(\sigma \downarrow 1))$
$(\psi(\sigma \downarrow 2)))$
$\mathit{out} = \mathit{fix}(\lambda\psi\nu.\nu \notin A \rightarrow \perp, \mathit{put}(\nu \mid A)\psi)$
$\mathit{put} = \lambda\alpha\psi.\mathit{get}\,\alpha = \perp \rightarrow \perp, \langle \mathit{get}\,\alpha, \alpha\ \mathit{rest}\,\psi\rangle$
$\mathit{get} = \lambda\alpha.\alpha\ \mathit{first}\,\lambda\nu.\nu \in H \rightarrow \nu, \perp$
$\mathit{first} = \mathcal{E}[\![(\texttt{lambda (x y) x})]\!]\rho_0$
$\mathit{rest} = \mathcal{E}[\![(\texttt{lambda (x y) y})]\!]\rho_0$

6.2.4 Primitives

$\rho_0[\![\texttt{null?}]\!] = \mathit{funct}\,\lambda\epsilon\kappa.\epsilon\lambda\nu.\nu = \textbf{null} \rightarrow \mathit{true}\,\kappa, \mathit{false}\,\kappa$
$\mathit{true} = \mathcal{E}[\![(\texttt{lambda (x y) x})]\!]\rho_0$
$\mathit{false} = \mathcal{E}[\![(\texttt{lambda (x y) y})]\!]\rho_0$

$\rho_0[\![\texttt{make-vector}]\!] =$
$\mathit{funct}\,\lambda\epsilon\kappa.\epsilon\lambda\nu.\nu \geq 0 \rightarrow \kappa(\textbf{mkvec}(v)), \kappa\perp$
$\textbf{mkvec}(n) =$
$\lambda\epsilon_1\ldots\epsilon_n\kappa.\epsilon_1\lambda\nu_1.\ldots\epsilon_n\lambda\nu_n.\mathit{vecref}_n\nu_1\ldots\nu_n\kappa$ in V
$\mathit{vecref}_n = \lambda\nu_1\ldots\nu_n.$
$\mathit{funct}\,\lambda\epsilon\kappa.\epsilon\lambda\nu.\nu = 0 \rightarrow \kappa\nu_1,$
$\vdots$
$\nu = n - 1 \rightarrow \kappa\nu_n, \kappa\perp$

6.2.5 Derived Expression Types

(**define** (⟨pattern⟩ ⟨variable$_1$⟩ ...) ⟨body⟩)
⟶ (**define** ⟨pattern⟩
(**lambda** (⟨variable$_1$⟩ ...) ⟨body⟩))

(**lambda** () ⟨body⟩) ⟶ (⟨body⟩)
(**lambda** (⟨variable$_0$⟩ ⟨variable$_1$⟩ ...) ⟨body⟩)
⟶ (**lambda** (⟨variable$_0$⟩)
(**lambda**(⟨variable$_1$⟩ ...) ⟨body⟩))

`(list)` ⟶ `(quote ())`
`(list` ⟨expression$_0$⟩ ⟨expression$_1$⟩ ...`)`
⟶ `((lambda (x y z) z x y)`
⟨expression$_0$⟩ `(list` ⟨expression$_1$⟩ ...`))`

`(vector` ⟨expression$_1$⟩ ... ⟨expression$_n$⟩`)`
⟶ (*v* *n* ⟨expression$_1$⟩ ... ⟨expression$_n$⟩)
where *v* is the value of **make-vector** described in Section 5.4.

`(quote` ⟨integer⟩`)` ⟶ ⟨integer⟩
`(quote` ⟨character⟩`)` ⟶ ⟨character⟩
`(quote` ⟨string⟩`)` ⟶ ⟨string⟩
`(quote` ⟨symbol⟩`)` ⟶ ⟨list of characters⟩
`(quote ( . ` ⟨datum⟩`)))`
⟶ `(quote` ⟨datum⟩`)`
`(quote (`⟨datum⟩ ⟨list⟩`)))`
⟶ `((lambda (x y z) z x y)`
`(quote` ⟨datum⟩`)) (quote (`⟨list⟩`))))`

`'`⟨datum⟩ ⟶ `(quote` ⟨datum⟩`)`

⟨string⟩ ⟶ ⟨list of characters⟩

A Editing Alonzo Programs

The syntactic conventions of Alonzo programs are much like Scheme's, so editors that are tailored for Scheme are automatically tailored for Alonzo. Experience has shown that the effective use of Scheme pretty printers requires special treatment of conditionals. The trick is to define **if** as follows, and use this **if** for all conditionals.

```
(define (if test cons alt)
  (test cons alt))
```

References

[AS88] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, 21(8):9–24, August 1988.

[Bar84] H. P. Barendregt. *The Lambda Calculus.* North-Holland, Amsterdam, revised edition, 1984.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Hug82] R. J. M. Hughes. Super combinators: A new implementation method for applicative languages. In *1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, Pittsburg, PA, August 1982.

[HW88] Paul Hudak and Philip Wadler et. al. Report on the functional programming language haskell—draft proposed standard. Technical Report YALEU/DCS/RR-666, Yale University, New Haven, CN, December 1988.

[RC86] Jonathan Rees and William Clinger eds. Revised^3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, September 1979.