# CakeML Reference Manual

Scott Owens
University of Kent
https://cakeml.org

May 27, 2018

# Preface

This manual is intended to describe the CakeML language, compiler, and associated proof technologies. We hope that it will be useful for anyone who wants to understand CakeML, either with an aim to collaborate with main CakeML team,[1] or to work with CakeML independently.

We have attempted to make this reference manual self-contained. However, our aim is not to explain the basics of functional programming, programming language semantics, or interactive theorem proving, and so we expect of the reader at least some familiarity with a functional programming language, and with basic discrete mathematics: logic, inductively defined relations, etc.

[1] Project ideas are listed at `https://cakeml.org/projects.html`.

# Contents

# Part I

# The CakeML Language

# Chapter 1

# An informal description of CakeML

CakeML is a functional language in the ML family. Its syntax and semantics resembles Standard ML, and the concepts will also be largely familiar to OCaml and Haskell programmers. This chapter describes the CakeML language. Although it uses example programs to illustrate its various features, its focus is on what the language is, and not on how to effectively use it. However, these pragmatics should be clear to programmers with experience in other functional languages.

CakeML's syntax has a two-level structure with declarations and expressions. A program is a sequence of top-level declarations that define variables, functions, types, and modules. Computations are performed by expressions that form the bodies of functions and initialise variables.

Expression are built up from constants, variables, function applications, and special forms for conditionals, pattern matching, exception handling, anonymous functions, and local definitions. Each expression has a type, and computes a value of that type, unless it infinite loops or raises an exception.

CakeML supports the following types of values:

- atomic values – booleans, integers, words, strings, and characters (§1.2 and §2);
- functions (§1.5);
- immutable containers – lists, tuples, user-defined algebraic data types (also called variant types) (§1.6), and vectors (§2.7); and
- mutable containers – references (§1.8) and arrays (§2.1).

## 1.1 Comments and white space

All comments and white space act as delimiters, but are otherwise ignored. Comments are surrounded by `(* *)`. They nest properly, so that the following idiom works.

```
(*
  commented out code
  (* comment inside of commented out code *)
  commented out code
*)
```

The first `*)` does not end the entire comment, but only the corresponding `(*`.

## 1.2 Atomic values

**Booleans** The boolean constants are written as `true` and `false`. The type of booleans is `bool`. The conjunction and disjunction logical operators are written as infix `andalso` and `orelse`. They evaluate the left argument first, and only evaluate the right one if necessary (short-circuit evaluation). The `not` function negates a boolean value.

**Integers** Integers constants are written in decimal notation. The `~` character acts as the minus sign for writing negative numbers. So negative one is `~1` instead of `-1`. The type of integers is `int`.

CakeML provides the following infix arithmetic operators on integers: `+`, `-`, `*`, `div`, `mod`. Integers are represented internally as arbitrary precision numbers (*bignums*) so they can never overflow, and there is no maximum or minimum representable integer. The `div` and `mod` operations will raise the `Div` exception if given a divisor of 0. `div` returns the least integer less than the real quotient, so `~10 div 3` returns `~4`, and not `~3`. `mod` is consistent with this, so that `~10 mod 3` returns `2`, not `1`.

Integers are compared with the infix operators `=`, `<>`, `<`, `>`, `<=`, `>=`, each of which returns a `bool`.

**Words** CakeML supports both 64-bit and 8-bit words. All word constants are 64-bit, but 8-bit words can be created using functions in the standard library's Word8 module (§2.9). Word constants in decimal notation are prefixed with `0w` (for example, `0w256`). Words constants in

hexadecimal notation are prefixed with `0wx` (for example, `0wx3fA`). 64-bit words have the type `word`, whereas the type of 8-bit words is `Word8.word` from the standard library. See §2.8 and §2.9 for the operations on words.

**Strings** String constants are enclosed in double quotation marks (`"`). The type of strings is `string`.

The escape character is `\`, and it supports the following:

- `\n` : a newline character
- `\"` : a double quotation (`"`) character
- `\\` : a single backslash (`\`) character, and
- `\t` : a tab character.

The `^` infix operator concatenates strings. See §2.6 for other operations on strings.

**Characters** A character constant is just a string constant of length 1, with a `#` prefix, for example `#"c"` for the character 'c'. This means that characters use the same escape sequences as strings. The type of characters is `char`. See §2.2 for the operations on characters.

## 1.3 Variables and other names

**Variable syntax** There are two kinds of variable names: alphanumeric and symbolic. An alphanumeric variable must start with a lower case letter and be a sequence of letters, digits and the `_` and `'` characters. A symbolic variable is a sequence of the following characters: `! % & $ # + - / : < = > ? @ \ ~ ^ | ‘ *`.

Names of modules (§1.13) and constructors (from algebraic data types §1.6 and exceptions §1.11) are alphanumeric and start with an upper case letter. Additionally, the names `true`, `false`, `ref`, and `nil` are constructors, rather than variables.

Names of types are alphanumeric and start with a lower case letter. Type variables must be alphanumeric and start with `'`.

No name can be any of the following reserved words:

```
and  andalso  case  datatype  else  end  exception
fn  handle  if  in  let  op  of  orelse  raise
ref  sig  struct  structure  then  type  val
:>  =>  :  |
```

Make this list and CakeML's lexer agree.

**Declaration of variables** A `val` declaration introduces a new variable name with a given value. We will follow tradition and refer to these as variables, although their values cannot be changed once initialised.

```
val x = 1 + 2
val my_string = "Happy"
```

Variable declarations may optionally be followed by a semi-colon (`;`).

## 1.4 Conditionals

Conditionals are expressions that compute a value. They use `if`, `then`, and `else`. For example,

```
if x < 0 then 0 - x else x
```

This tests if `x` is less than `0`, and if so, evaluates to `0-x`, and if not evaluates to `x`. The `else` and associated sub-expression cannot by omitted. The test expression (following `if`) must be of `bool` type, and the result expressions must both have the same types.

## 1.5 Functions

**Declaration of functions** A function is declared with `fun`, followed by the function's name, its parameters, `=`, and finally its body expression, as follows:

```
fun abs x =
  if x < 0 then 0 - x else x
```

A function returns whatever its body expression evaluates to, in this example the absolute value of its argument. Functions are called by placing their argument expression(s) immediately after the function's name:

```
val x = abs ~100
val y = abs (if x mod 2 = 0 then x - 2 else x - 5)
```

Functions can be recursive, and can have multiple arguments/parameters which are all placed in order after the function name without any punctuation, for both calls and declarations.

```
fun gcd a b =
  if b = 0 then
    a
  else
    gcd b (a - b * (a div b));
```

Mutually recursive functions are separated by `and` instead of `fun`:

```
fun even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1)
```

Function call expressions evaluate their argument in right-to-left order, before calling the function.

Function declarations may optionally be followed by a semi-colon (`;`).

**Anonymous and partially applied functions** Functions are first-class values. This means that a function can be returned from another functions, or passed as an argument to another function, and that a variable can have a function as its value. In fact, the function name in a `fun` declaration is just a normal variable that has a function value.

The `->` type operator gives the type of a variable or expression that has a function value. In the above examples, `abs`, `gcd`, `even`, `odd` have types `int -> int`, `int -> int -> int`, `int -> bool`, and `int -> bool` respectively.

An anonymous function can be created directly inside of an expression with `fn` as follows:

```
val f =
  if should_inc then (fn a => fn b => a + b + 1)
  else (fn a => fn b => a + b)
val x = f 1 2
```

Here `f` is a function with type `int -> int -> int`.

The type of a function with more than one argument has multiple arrows `->`, and in the above, we used `fn` twice to build a two argument anonymous function. In CakeML, functions are curried, meaning that they all take exactly one argument, and multiple argument functions

are simulated by passing in the arguments one after the other. This means that we do not have to apply a function to all of it's arguments at once.[1] Using the same `f`:

```
val g = f 8
val x = g 1
val y = g 2
```

Here `g` is a one argument function with type int -> int, and thus `x` and `y` are integers.

**Lexical scope** A function value can be created inside of another function and returned from it. If the returned function refers to variables from the function that created it, then it will continue to do so after it is returned. In the following `x` is equal to `5` both times that `do_add` is called.

```
fun mk_add x =
  if should_inc then (fn y => x + y + 1)
  else (fn y => x + y)
val do_add = mk_add 5
val x = do_add 10
val y = do_add 11
```

## 1.6 Compound values and pattern matching

**Tuples** A tuple expression is surrounded by parentheses and its constituent expressions are separated by commas. For example, `("1", 3+5, true)` is a three element tuple. The empty tuple `()` is the (only) member of the type `unit`. The type of a tuple is formed from the types of its members, separated by `*`, so our first tuple example has type `string * int * bool`.

**Lists** A list of values is surrounded by square brackets and separated by commas. Each element of the list must have the same type. For example, the expression `[1, 2+3, 0, 4]` is a list of integers containing four elements. Its type is `int list`. The empty list is written `[]`, and has a polymorphic type `'a list`, indicating that it can be used as an

[1] The compiler optimises multiple argument functions so that the arguments are not actually passed in one at a time when the program is run.

empty list of any type of contents. The cons operator (constructing a list with one more element) is written as `::`, so the above list could also have been written as `1::2+3::0::4::[]`.

The `@` infix operator appends two lists. See §2.4 for more list-related functions.

**Algebraic data types** A new data type is declared with `datatype`, giving the type a name, and listing the constructors that form elements of the new type, separated by `|`. For example, the following introduces a new type `colour` and three new values of type `colour`: `Red`, `Green`, and `Blue`. The type name must begin with a lower case letter, and the constructors with upper case letters.

```
datatype colour = Red | Green | Blue
```

The constructors can contain values of given types specified with `of`. Those values can be of the same type as the datatype itself; that is, the types constructed by `datatype` can be recursive. We can model a binary search tree with a terminal leaf, and interior nodes that contain left and right child trees, and integer keys and values.

```
datatype search_tree =
  Leaf
| Node of search_tree * int * int * search_tree
```

We can create a `search_tree` by applying the constructor to four arguments of the given types using a tuple-like syntax. Here are some example `search_trees`.

```
val empty_tree = Leaf
val two_node = Node (Leaf, 1, 0, Node (Leaf, 2, 100, Leaf))
```

Although the syntax looks like the constructor applied to a tuple expression, it is not as flexible as that, and we cannot apply `Node` to an actual tuple value, as in the following **non-example**.

```
val tup = (Leaf, 2, 100, Leaf)
val error = Node tup
```

We can generalise the tree datatype to contain values of any type by introducing a type parameter, which is written *before* the type name:

```
datatype 'a search_tree =
  Leaf
| Node of 'a search_tree * int * 'a * 'a search_tree
```

The `Leaf` example now has type `'a search_tree`, and our `Node` example has type `int search_tree`. We can create a `char search_tree` or `string list search_tree` as follows.[2]

```
val char_tree =
  Node (Leaf, 1, #"0", Node (Leaf, 2, #"A", Leaf))
val sl_tree =
  Node (Leaf, 1, ["44", "name"], Node (Leaf, 2, [""], Leaf))
```

Types with multiple parameters put them in parentheses and separate them with commas:

```
datatype ('k, 'a) search_tree =
  Leaf
| Node of ('k, 'a) search_tree * 'k * 'a * ('k, 'a) search_tree
```

Similar to functions, data types can be mutually recursive with subsequent types being separated by `and`.

```
datatype ('k, 'a) nary_tree =
  Node of 'k * 'a * ('k, 'a) tree_list
and ('k, 'a) tree_list =
  Empty
| Cons of ('k, 'a) nary_tree * ('k, 'a) tree_list
```

Algebraic data types can also be composed. For example, the above `nary_tree` type could be defined using the built-in list type.

```
datatype ('k, 'a) nary_tree =
  Node of 'k * 'a * ('k, 'a) nary_tree list
```

No `datatype` declaration may use the special constructor names `true`, `false`, `ref`, `::`, and `nil`.

Add this check to the type system.

**Pattern matching** Compound values are de-structured with pattern matching `case` expressions. A pattern has the same shape as the value that it matches. Patterns are separated by `|`, and each pattern is accompanied by a `=>` and expression. The `case` expression finds the first pattern that matches the value given to it, and evaluates that pattern's expression. The patterns must all match values of the same type, and the expression must all have the same type as each other.

[2] Type application is left associative, so `string list search_tree` is the same as `(string list) search_tree`.

```
fun is_red c =
  case c of
    Red => true
  | Green => false
  | Blue => false
```

Variables inside of patterns match anything, and are bound to the corresponding piece of the value being matched. Underscore _ matches anything, but binds nothing. Patterns can also be nested.

```
fun is_red c =
  case c of
    Red => true
  | _ => false

fun first p =
  case p of
    (x, _) => x

fun first_red p =
  case p of
    (Red, _) => true
  | _ => false
```

Numbers, booleans, lists, tuples, and constructors from algebraic data types and exceptions can all appear in patterns. As a final example, the following function searches for a given element in an `nary_tree`, using patterns for data type constructors and lists.

```
fun nary_mem k tree =
  case tree of
    Node (k1, v, []) => k1 = k
  | Node (k1, v, tree1::trees) =>
    nary_mem k tree1 orelse nary_mem k (Node (k1, v, trees))
```

**Other places patterns can appear** Besides `case` expressions and `handle` expressions (§1.11), patterns can also be used in variable declarations (`val`), and as an anonymous function expression's (`fn`) parameter.

```
val (x, y, z) = if something then (1, 2, 3) else (4, 5, 6)
val _ = print "message"
val f = fn (Node (k, v, ts) => v)
```

**Unmatched patterns** If no pattern in a `case` expression matches the value, then a `Bind` exception is raised. Similarly, if the pattern given in a `val` declaration or `fn` expression does not match the initialisation value, or function argument, a `Bind` exception is raised.

## 1.7 Type annotations

Types are inferred in CakeML, so programs do not need to specify the types of variable declarations or functions. However, we can optionally annotate any pattern or expression with its type using a colon `:`. That type is then enforced by the compiler.

```
val (x : int) = 1
fun f (x : bool) = if x then 0 else 1
```

## 1.8 References

A reference is a mutable structure that contains exactly one value. The type of a reference containing a value of type $\tau$ is written $\tau$ `ref`. References have the following operations, where `:=` is written infix.

- `ref : 'a -> 'a ref`
  Create a new reference with an initial value.
- `! : 'a ref -> 'a`
  Get the value from a reference.
- `:= : 'a ref -> 'a -> unit`
  Replace the value in the reference with a new one.

For example, the following code initialises a reference to `0`, then increments it to contain `1`. Finally, the value of `x` is also `1`.

```
val r = ref 0
val _ = r := !r + 1
val x = !r
```

## 1.9 Infix operators

CakeML supports the set of infix operators listed in Figure 1.1. Except for `=`, `<>`, `o`, and `before`, these all operate on primitive values, and they are described in the listed sections.

| | |
|---|---|
| `* div mod /` | (§1.2, / unimplemented) |
| `+ - ^` | (§1.2) |
| `@ ::` | (§1.6) |
| `< > <= >= <> =` | (§1.2, §1.9 for =) |
| `o :=` | (§1.8, `o` unimplemented) |
| `before` | (unimplemented) |
| `orelse andalso` | (§1.2) |

Figure 1.1: Infix operators from tightest (at the top) to loosest binding

**Equality** The infix equality operator = checks whether two values are equal, returning a boolean. The values must be of the same type. For atomic data (booleans, integers, bytes, and strings), they are equal exactly when they are the same value. For immutable data (tuples, lists, vectors, and algebraic data types), they are equal if they are made up of equal elements, recursively. For mutable data (references, arrays, and byte arrays), they are equal if they are the same structure. Their elements are not inspected. If the equality function encounters a function value while traversing its arguments, it returns `true`.

The <> operator is the negation of =.

**The `before` and `o` operators** The `before` operator sequences two expressions, and returns the value of the first one. The second is evaluate for its side effect. The type of `before` is `'a -> unit -> 'a`. The following function increments a reference, and returns the pre-increment value.

```
fun inc r =
  !r before r := !r + 1
```

The `o` operator composes two functions, first applying the one on the right, then the one on the left. Its type is `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`. In the following example, `a` is bound to the string `"2"`.

```
fun add1 x = x + 1
val a = (Int.toString o add1) 1
```

Implement o and before in the basis. I suspect this will require making before a keyword, due to evaluation order issues.

**Suppressing infix status** Infix status is suppressed with the `op` prefix. For example, we can use `op+` to get a prefix addition function, and then partially apply it to an argument.

```
val add1 = op+ 1
```

The `op` prefix also works in patterns, which lets us define our own versions of these operators, shadowing their global definitions.

```
val op< = String.<
val x = "1" < "2"
```

Since `andalso` and `orelse` are reserved words (§1.3), `op` cannot be used on them.

## 1.10 Sequencing

Expressions are sequenced by enclosing them in parentheses `( )`, and separating them with semi-colons `;`. The sequence is evaluated left-to-right. The value of the last expression is returned; the previous expressions are evaluated only for their side-effects. The following function increments a reference and returns the post-increment value.

```
fun inc r =
  (r := !r + 1; !r)
```

## 1.11 Exceptions

**Exception declarations** The `exception` declaration creates a new kind of exception. It must start with a capital letter, and can have arguments similar to the constructor of an algebraic data type.

```
exception Error
exception Type_error of int * string
```

**Raising exceptions** A `raise` expression raises its argument as an exception. For example,

```
if x < 0 then raise Error
else x
```

```
if t1 = t2 then t1
else raise (Type_error (loc, "type mismatch"))
```

A `raise` expression can have any type, and so can be used anywhere an expression is expected, as in `4 + (raise Error)`.

**Handling exceptions** If an exception is not handled, it will terminate the execution of the program. A `handle` expression allows us to handle exceptions raised in the preceding expression. It is followed by a pattern match with the same syntax as in `case` (§1.6). Each pattern must match either an exception or be `_` to handle all exceptions. If no pattern matches the raised exception, it is not handled and instead raised out of the `handle` expression.

The first declaration below sets `m` to `0`, whereas the second prints the error message from a `Type_error` exception, and a generic message for any other exception.

```
val m =
  1 div 0 handle Div => 0
val x =
  type_check p
  handle
    Type_error (i, s) => print (Int.toString i ^ " " ^ s)
  | _ => print "unknown exception"
```

Update print function call

## 1.12 Local bindings

A `let` expression lets us declare variables in a local scope.

```
let
  val x = 1
  fun f y = x + y
in
  f 10 + x
end
```

The declarations occur between the `let` and `in`. The expression between `in` and `end` becomes the value for the whole `let` expression.

## 1.13 Modules

Write this section

## 1.14 Differences from Standard ML

### 1.14.1 Unsupported features

Here we only list missing features of the language, and not the basis library.

The CakeML module system is heavily restricted compared to SML's. Structures cannot be nested, and functors are not supported. Structures cannot be given alternate names, or opened. Signatures cannot be named, and must appear directly on the structure that they seal. Only opaque sealing is supported (`:>`, but not `:`). Signatures cannot contain `include`, `sharing`, `structure`, `eqtype` or datatype replication specifications.

In contrast, CakeML's core language is more fully featured, with only the following omissions. The `local`, `abstype`, and datatype replication declarations are not supported. Neither are records, type annotations that contain type variables, and `while` loops. Furthermore, `let` expressions are restricted to `val` and `fun` declarations; they cannot contain type or exception declarations. The pattern language does not support the `as` form.

CakeML has a fixed set of infix operators, and does not support user defined parsing precedences (`infix`, `infixr`, and `nonfix`). See Figure 1.1 for the supported infix operators.

Floating point numbers are not supported, nor are integer constants written in hexadecimal notation (`0x`). Lastly, only the most basic escape sequences are supported in strings (just `\n`, `\t`, `\"`, and `\\`).

Syntactic sugar for `fun` declarations with pattern parameters is not provided, instead parameters are variables, and must be matched against with `case` expressions, if desired.

### 1.14.2 Syntactic and semantic differences

All names must follow the OCaml convention where structure and constructor names start with a capital letter, and type and variable names must either start with a lower case letter, or be symbolic names.

CakeML does not support the limited overloading found in the SML Basis Library.

CakeML does not support the equality types that SML uses to ensure that the `=` operation never encounters a function value. Instead, the `=` returns `true` if it does.

The types of bindings in `let` expressions cannot be generalised. This restriction does not affect top level, or structure top level declarations.

The built-in CakeML functions prefer the curried style over SML's preferred tupled style. For example, we write `Vector.sub v 12` to get the $12^{\text{th}}$ element of vector v, rather than `Vector.sub (v,12)`. The infix operators are also curried when used with `op`.

CakeML guarantees right to left evaluation order for function calls.

## 1.15 Differences from OCaml

Because CakeML has a significantly different syntax, we focus here on the major features and semantic differences rather than attempt to catalogue every piece of OCaml syntax that lacks a CakeML analogue. Briefly, CakeML does not have floating point numbers, records, objects, polymorphic variants, labels, nested/local modules, or functors.

Strings are immutable; however, the Word8Array structure provides byte arrays.

CakeML guarantees right to left evaluation order for function calls.

The = operator does not traverse mutable data (arrays and references), instead it compares the pointers, SML style.

# Chapter 2

# Standard library

## 2.1 Array

Arrays have a fixed length and support constant-time indexing. Each element of an array must have the same type. The operations on arrays are collected in the `Array` module. If the elements of an array are of some type $\tau$ then the vector's type is $\tau$ `Array.array`.

The following are the supported operations on arrays. If an index is out of bounds (or negative), the `Subscript` exception is raised.

- `Array.array : int -> 'a -> 'a array` Create a new array of given length and default value.

- `Array.length : 'a array -> int` Get the array's length.

- `Array.sub : 'a array -> int -> 'a` Get the $i^{th}$ element.

- `Array.update : 'a array -> int -> 'a -> unit` Set the $i^{th}$ element.

## 2.2 Char

## 2.3 Int

## 2.4 List

## 2.5 Option

An optional value is either `NONE`, indicating that no value is present, or `SOME` $v$ for a value $v$. They type of an optional value that might contain a value of type $\tau$ is $\tau$ `option`. There are no operations for de-structuring options.

## 2.6 String

## 2.7 Vector

Vectors are immutable array-like structures. They have a fixed length and support constant-time indexing. Each element of a vector must have the same type. The operations on vectors are collected in the `Vector` module. If the elements of a vector are of some type $\tau$ then the vector's type is $\tau$ `Vector.vector`.

The following are the supported operations on vectors. If an index is out of bounds (or negative), the `Subscript` exception is raised.

- `Vector.fromList : 'a list -> 'a vector` Convert a list to a vector containing the same elements (in order).
- `Vector.length : 'a vector -> int` Get the vector's length.
- `Vector.sub : 'a vector -> int -> 'a` Get the $i^{th}$ element.

## 2.8 Word64

## 2.9 Word8

## 2.10 Word8Array

Byte arrays have a fixed length and support constant-time indexing. Each element of an array must be a byte (`Word8.word` type). The operations on byte arrays are collected in the `Word8Array` module. The type of a byte array is `Word8Array.array`.

The following are the supported operations on byte arrays. If an index is out of bounds (or negative), the `Subscript` exception is raised.

- `Word8Array.array : int -> Word8.word -> Word8Array.array` Create a new array of given length and default value.
- `Word8Array.length : Word8Array.array -> int` Get the array's length.
- `Word8Array.sub : Word8Array.array -> int -> Word8.word` Get the $i^{th}$ element.
- `Word8Array.update : Word8Array.array -> int -> Word8.word -> unit` Set the $i^{th}$ element.

# Chapter 3

# Formal syntax and semantics

## 3.1 Lexical and context free syntax

| | | | |
|---|---|---|---|
| $\alpha$ | = | `'[a-zA-Z'_]`$^*$ | type variable |
| $i$ | = | `~`$^?$`[0-9]`$^+$ | integer constant |

Lexical syntax for str, tn, v, fqv, C, fqC, S

| | | | |
|---|---|---|---|
| $t$ | ::= | | types |
| | | $\alpha$ | type variable |
| | \| | $t$ `*` $t$ | tuple |
| | \| | $t$ `->` $t$ | function |
| | \| | $tn$ | type name |
| | \| | $t$ $tn$ | type application |
| | \| | `(`$t$`,`$t$`(,`$t$`)`$^*$`)` $tn$ | type application |
| | \| | `(`$t$`)` | |

| | | | |
|---|---|---|---|
| $l$ | ::= | | literal constants |
| | | `true` | |
| | \| | `false` | |
| | \| | `()` | unit |
| | \| | `[]` | empty list |
| | \| | `nil` | empty list |
| | \| | $i$ | integer |
| | \| | *str* | string |

| | | | |
|---|---|---|---|
| $p$ | ::= | | patterns |
| | | `_` | wildcard |
| | \| | $v$ | variable |
| | \| | $l$ | constant |
| | \| | $fqC$ | constant constructor |
| | \| | $fqC\ p$ | constructor application |
| | \| | `(`$p$`,`$p$`(,`$p$`)*)` | tuple |
| | \| | $p$`::`$p$ | list (first and rest) |
| | \| | `[`$p$`(,`$p$`)*]` | list (fixed length) |
| | \| | `ref` $p$ | reference |
| | \| | `(`$p$`)` | |

| | | | |
|---|---|---|---|
| $e$ | ::= | | expressions |
| | | $l$ | constant |
| | \| | $fqv$ | variable |
| | \| | $fqC$ | constant constructor |
| | \| | $fqC\ e$ | constructor application |
| | \| | `(`$e$`,`$e$`(,`$e$`)*)` | tuple |
| | \| | `[`$e$`(,`$e$`)*]` | list |
| | \| | `raise` $e$ | exception raising |
| | \| | $e$ `handle` $p$ `=>` $e$ `(\|`$p$ `=>` $e$`)*` | exception handling |
| | \| | `fn` $v$ `=>` $e$ | function |
| | \| | $e\ e$ | function application |
| | \| | $e\ op\ e$ | binary operator |
| | \| | `((`$e$`;)*`$e$`)` | sequencing |
| | \| | `if` $e$ `then` $e$ `else` $e$ | conditional |
| | \| | `case` $e$ `of` $p$ `=>` $e$ `(\|`$p$ `=>` $e$`)*` | pattern matching |
| | \| | `let` $(ld\|;)^*$ `in` $(e;)^*e$ `end` | let |
| | \| | `(`$e$`)` | |
| $ld$ | ::= | | local definition |
| | | `val` $x$ `=` $e$ | value |
| | \| | `fun` $v\ v^+$ `=` $e$ `(and` $v\ v^+$ `=` $e)^*$ | function |
| $op$ | ::= | | infix operators |
| | | `*\|div\|mod` | multipicative |
| | \| | `+\|-` | additive |
| | \| | `@\|::` | list |
| | \| | `=\|<\|<=\|>\|>=\|<>` | comparison |
| | \| | `o\|:=` | |
| | \| | `before` | sequencing |
| | \| | `andalso\|orelse` | logical |

| | | | |
|---|---|---|---|
| $d$ | ::= | | declarations |
| | | `val` $p$ `=` $e$ | value |
| | \| | `fun` $v$ $v^+$ `=` $e$ (`and` $v$ $v^+$ `=` $e$)$^*$ | function |
| | \| | `datatype` $tyd$ (`and` $tyd$)$^*$ | type |
| | \| | `exception` $c$ | exception |
| $c$ | ::= | | constructors |
| | | $C$ | constant |
| | \| | $C$ `of` $t$ | with arguments |
| $ptn$ | ::= | | type names w/parameters |
| | | `(`$\alpha$(`,`$\alpha$)$^*$`)` $tn$ | |
| | \| | $\alpha$ $tn$ | |
| | \| | $tn$ | |
| $tyd$ | ::= | $ptn$ `=` $c$(`\|` $c$)$^*$ | define a single type |

| | | | |
|---|---|---|---|
| $m$ | ::= | `structure` $S$ $s^?$ `=` `struct` ($d$\|`;`)$^*$ `end` | modules |
| $s$ | ::= | `:>` `sig` ($sp$\|`;`)$^*$ `end` | signatures |
| $sp$ | ::= | | specifications |
| | | `val` $v$ `:` $t$ | value |
| | \| | `type` *tyn* | opaque type |
| | \| | `datatype` $tyd$ (`and` $tyd$)$^*$ | type |
| | \| | `exception` $c$ | exception |

| | | | |
|---|---|---|---|
| $top$ | ::= | | top-level declaration |
| | | $m$ | module |
| | | $d$ | declaration |

## 3.2 Abstract syntax

## 3.3 Type system

## 3.4 Operational semantics

# Chapter 4

# Theorems

# Part II

# The CakeML compiler and verification tools

# Chapter 5

# Installing and using the compiler

# Chapter 6

# Extracting pure CakeML programs from HOL4

# Chapter 7

# Verifying effectful CakeML programs with characteristic formulae