Language Reference Manual for Propeller
A Property Manipulation Language

Isra Ali
Tester
isra.hamid_ali@tufts.edu

Gwendolyn Edgar
Manager
gwendolynedgar@tufts.edu

Randy Price
Language Guru
edward.price@tufts.edu

Chris Xiong
System Architect
lxiong01@tufts.edu

February 28, 2022

1 Overview

Propeller is a language designed to write programs using the reactive programming paradigm. Programs written in Propeller usually act upon external events such as user inputs and environmental changes. It has a binding system: functions can be bound to properties of objects (akin to members or fields in other languages), so that they get called whenever the value of these properties change.

Propeller has several possible runtime environments: a simple runtime with a fixed entry point, one with GUI capabilities, and one that monitors certain local parameters of the computer (e.g. CPU temperature sensor readings, files creation and deletion, load average and so on).

As of now, the syntax and semantics described in this document is tentative. The final version may contain revisions to address unforeseen design flaws.

1.1 Notation used in this document

This document uses a variation of Backus-Naur form to describe the syntax of the language. The most significant additions are the character range notation (`"a"-"d"` rather than `"a"|"b"|"c"|"d"`), and items repeated zero or more times (`{<characters>}` means `<characters>` repeated zero or more times).

2 Lexical Conventions

2.1 Comment

Propeller supports single-line comments. Anything following a hash (`#`) that is not part of a string literal will be treated as a comment. Comments automatically terminate at the end of line.

2.2 Identifiers

```
    <letter> ::= "A"-"Z" | "a"-"z"
<identifier> ::= <letter> ("?" | "")
               | <letter> ("_" | "")
                 {(<letter> | <digit>) |
                  ((<letter> | <digit>) "_")}
                 (<letter> | <digit>) ("?" | "")
```

To put this rather cryptic identifier rule in plain English: only letters, digits, underscores and question mark are allowed in identifier names. They must start with a letter, and must not end with an underscore. The question mark can only appear at the end, and two punctuation marks cannot appear consecutively.

```
# This is a comment.

# valid identifiers
x
ready?
aBc123
o_o
AsDfG_1234_5?

# invalid identifiers
8eight
two__underscores
propeller_
too?early
huh_?
```

2.3 Keywords

Propeller has 25 reserved keywords:

```
and   bind     break  continue  elif
else  external float  fn        for
from  if       int    list      not
obj   objdef   or     return    str
to    unbind   void   while     xor
```

2.4 Separators

Propeller has 10 separators used to construct literals, define functions, separate statements, and more:

```
( ) [ ] { }
, . ;   ->
```

2.5 Operators

Propeller has 15 operators for comparison, logic, and arithmetic.

```
# comparison  logic  arithmetic
  =           and    +
  !=          or     -
  >           not    *
  <           xor    /
  >=                 %
  <=
```

2.6 Literals

```
            <digit> ::= "0"-"9"
      <int-literal> ::= {<digit>}
    <float-literal> ::= {<digit>} "." {<digit>}
     <bool-literal> ::= "true" | "false"
<string-characters> ::= character or escape sequence
   <string-literal> ::= "'" {<string-characters>} "'"
```

For strings, escape sequences are interpreted the same way OCaml interprets them (as in `Scanf.unescaped`).

```
# int literals     float literals     boolean literals       string literals
1                  3.14               true                   'hello'
23                 0.78               false                  'h0wdy!'
0                  12.345                                    '\twhat\'s up?\n'
5839407430         0.999
```

3 Syntax

3.1 Expressions

```
<binary-operators> ::= "+" | "-" | "*" | "/" | "%"
                     | "==" | "!=" | "<" | "<=" | ">" | ">="
                     | "and" | "xor" | "or"
 <unary-operators> ::= "-" | "not"
       <expr-list> ::= <expression> | <expr-list> "," <expression>
        <arg-list> ::= "" | <expr-list>
      <expression> ::= <int-literal>
                     | <float-literal>
                     | <bool-literal>
                     | <string-literal>
                     | <list>
                     | <identifier>
                     | <expression> <binary-operators> <expression>
                     | <unary-operators> <expression>
                     | <identifier> "=" <expression>
                     | <identifier> "[" <expression> "]"
                     | "(" <expression> ")"
                     | <identifier> "(" <arg-list> ")"
                     | <identifier> "." <identifier>
                     | <identifier> "." <identifier> = <expression>
```

Operator precedence follows standard conventions (e.g. C-style precedence).

```
# the following are all syntactically correct expressions
1
abc
abc + def
alist[6]
zzz = true
(1 + 2) * 3
o.k = 'ok'
not false
zebra % horse
[true, false]
```

3.2 List Literals

```
 <list> ::= "[" arg-list "]"
```

```
# int list
[1, 2, 3, 4]

# float list
[1.2, 3.4, 5.6]

# bool list
[true, false, true]

# str list
['joyce', 'cummings', 'center']

# int list list
[[5], [6]]

# empty list
[]
```

3.3 Declarations

3.3.1 Variable Declaration

```
              <types> ::= "int" | "float" | "bool" | "str" | "void" |
                          <identifier> | <types> "list"
     <variable-decl> ::= <types> <identifier> ";"
<variable-decl-list> ::= "" | <variable-decl-list> <variable-decl>
```

3.3.2 Function Declaration

```
    <formal-list> ::= <types> <identifier>
                    | <formal-list> "," <types> <identifier>
<formal-list-opt> ::= "" | <formal-list>
  <function-decl> ::= "fn" <identifier> "(" <formal-list-opt> ")" "->"
                      <types> "{" <variable-decl-list>
                      <statement-list> "}"
```

3.3.3 Object Type Declaration

```
         <obj-def> ::= "objdef" <identifier> "{" <variable-decl-list> "}"
<external-obj-def> ::= "external" <obj-def>
```

```
# defining an object type "Patient"
objdef Patient
{
  str   name;
  int   age;
  float height;
  float weight;
}
# an external object type
external objdef ExternObj
{
  str name;
  int value;
}
# declaring a variable of type Patient
Patient p;

# declaring variables of other types
int x;
str name;
float pi;
bool is_ready?;
float list color;

# a minimal function that does nothing
fn do_nothing() -> void
{
  return;
}
```

3.4 Statements

3.4.1 Sequencing

```
<statement-list>  ::= "" | <statement-list> <statement>
<statememt-block> ::= "{" <statement-list> "}"
```

3.4.2 Control Flow

- Branching

```
     <if-statement> ::= "if" <expression> <statement-block>
                        <elif-statements> <else-clause>
   <elif-statements> ::= {"elif" <expression> <statement-block>}
       <else-clause> ::= "" | "else" <statement-block>
```

  Else clauses are attached to the closest unmatched if clause before it.

- Loops

```
    <for-statement> ::= "for" <identifier> "from" <expression>
                        "to" <expression> <statement-block>
  <while-statement> ::= "while" <expression> <statement-block>
```

- Jumps

```
    <break-statement> ::= "break" ";"
 <continue-statement> ::= "continue" ";"
   <return-statement> ::= "return" ("" | <expression>) ";"
```

3.4.3 Special Statements

Bind and unbind has special syntactical rules. These rules exist only to simplify the implementation. For users of Propeller they can be treated as if they are normal built-in functions.

```
   <bind-statement> ::= "bind" "(" <identifier> "." <identifier> ","
                        <identifier> ")" ";"
 <unbind-statement> ::= "unbind" "(" <identifier> "." <identifier> ","
                        <identifier> ")" ";"
```

3.4.4 All Statements

```
 <statement> ::= <expression> ";"
               | <if-statement>
               | <for-statement>
               | <while-statement>
               | <break-statement>
               | <continue-statement>
               | <return-statement>
               | <bind-statement>
               | <unbind-statement>
```

```
# example for statements
if (x > 0)
{
  print_str('Positive');
}
elif (x == 0)
{
  print_str('Zero');
}
else
{
  print_str('Negative');
}

while x < 10
{
  print_str('Less than 10')
  x = x + 1;
}

sum = 0;
for ii from 1 to 1000000
{
  sum = sum + ii;
  if (sum < 0)
  {
    break;
  }
}
```

4 Semantics

Propeller is heavily inspired by languages in the C family, including its semantics. For the sake of brevity, semantics that are commonly found in other widely-adopted languages will be omitted.

4.1 Data Types

Primitive types use the following internal representations:

Type	Size (bytes)	Description
`int`	4	Integer
`float`	8	IEEE 754 floating point
`bool`	1	Boolean
`str`	varies	Syntactic sugar for integer lists; stores UTF-8 encoded characters of a string

Propeller has a list type. Lists are immutable. All items in one list must be of the same type. List members are stored sequentially in memory.

Custom types can be defined. They can have a number of properties in them. A runtime can have several predefined custom types, which still must be declared in the program before use, but prefixed by the `external` keyword.

4.2 Operations

In binary operations of numbers, if one of the numbers is a float and the other is an integer, the integer will be automatically promoted to float. Division between integers gives the integer portion of the quotient. The modulo operation doesn't support float operands.

Lists can be indexed using the `[]` operator.

```
# comparison
2.3 == 1;   # false
4 != 5;     # true
8.34 > 3;   # true
9 < 10;     # true
5.5 >= 5.6; # false
100 <= 100; # true

# arithmetic
4 + 3;    # 7
2.0 - 1;  # 1.0
-3.3 * 3; # -9.9
5 / 2;    # 2
5.0 / 2;  # 2.5
6 % 4;    # 2

# boolean comparison
true != false; # true
false == true; # false

# boolean operations
not false;      # true
true and false; # false
true or false;  # true
true xor true;  # false
```

```
# list indexing
int list l = [1, 2, 3];
l[1];           # 2
```

4.3 Statements

Most of the control flow in Propeller works like their C counterpart.

For loops use a different syntax and semantics in Propeller. The loop variable is initialized to the value the first expression evaluates to, is incremented by 1 after the body of the loop is executed each time, and terminates when the loop variable is greater than the value the second expression evaluates to. The second expression is reevaluated every time in this process.

4.4 Binding

Property of objects can be bound to functions, so that whenever the value of this property changes, functions bound to that property are called.

Let β be the bindings that are currently established during execution of the program. β is one of the environment metavariable of Propeller's operational semantics. $\beta(o, p)$ is a set of functions bound to property p of object o. Note that this way objects of the same custom type don't share bindings.

Function bound to a property must accept 3 parameters: two values of the same type as the property itself, passing the old value of the property and new value of the property, and one of the object's type, which will be set to the object whose property value has changed.

When multiple functions are bound to the same property of an object, their order of execution is not defined and depends on the implementation.

Semi-formal operational semantics of syntactical forms related to binding will be given below. $\rho(o, p)$ retrieves the location where property p of object o is stored, and $\sigma(l)$ is the value at location l.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma, \beta, \cdots\rangle \Downarrow \langle v, \rho, \sigma_0, \beta, \cdots\rangle \\ \text{for each } f_i \in \beta(o,p), i = 1 \ldots n \\ \langle f_i(\sigma_0(\rho(o,p)), v, o), \rho, \sigma_{i-1}, \beta, \cdots\rangle \Downarrow \langle \mathit{void}, \rho, \sigma_i, \beta, \cdots\rangle\end{array}}{\langle \textsc{PropertyAssign}(o, p, e), \rho, \sigma, \beta, \cdots\rangle \Downarrow \langle v, \rho, \sigma_n\{\rho(o,p) \mapsto v\}, \beta, \cdots\rangle} \quad \textsc{PropertyAssign}$$

$$\frac{}{\langle \textsc{bind}(o, p, f), \rho, \sigma, \beta, \cdots\rangle \Downarrow \langle \mathit{void}, \rho, \sigma, \beta\{(o,p) \mapsto \beta(o,p) \cup \{f\}\}, \cdots\rangle} \quad \textsc{Bind}$$

$$\frac{}{\langle \textsc{unbind}(o, p, f), \rho, \sigma, \beta, \cdots\rangle \Downarrow \langle \mathit{void}, \rho, \sigma, \beta\{(o,p) \mapsto \beta(o,p) \setminus \{f\}\}, \cdots\rangle} \quad \textsc{Unbind}$$

(Note: If the object is of an external type in PropertyAssign, the actual behavior will differ a little, but that is the case only due to technical reasons and has no difference semantically.)

4.5 Program Execution

When a program written in Propeller is executed, it will start from a function called `init()`. After `init()` returns, it enters an event loop defined by the runtime library. For the most basic text-mode only runtime, the event loop simply terminates the program.

5 Built-in Functions and Standard Library

This section gives a tentative feature set of built-in functions and the standard library of Propeller.

Functions that are built-in will offer mathematical calculations such as trigonometric functions (`sin`, `cos`, `tan`, `fabs`, `exp`, `log`), terminating execution with exit code (`exit`), printing values of primitive types (`print_int`, `print_float`, `print_str`), as well as basic list operations (`empty?`, `first`, `rest`, the last two behave like `car` and `cdr` in some functional languages).

The standard library, built upon these built-in functions, will provide extra features such as string manipulation, printing values in lists, and some more advanced list operations. The standard library will be implemented using Propeller itself, and included implicitly in every program.

6 Examples

6.1 Minimal test driving program

This is a basic text mode program. It should print 3 and then terminate. It uses the basic text mode runtime.

```
# a simple custom type containing one int property
objdef A
{
  int prop;
}

A a;

fn print_prop(int oldval, int val, obj o) -> void
{
  # built-in function
  print_int(val + 2);
  # function in standard library to terminate execution
  exit(0);
}

fn init() -> void
{
  a.prop = 0;
  bind(a.prop, print_prop)
  # now that a.prop has been modified, print_prop will be called.
  a.prop = 1;
}
```

6.2 Temperature Monitor

A simple temperature monitor that prints a message when the reading exceeds a threshold. This program will require a correct runtime.

```
external objdef Sensor
{
  float temperature;
}

Sensor sensor;

fn print_warning(float oldt, float t, Sensor s) -> void
{
  if (t > 100)
  {
    print_str('Boiling!');
  }
}

fn init() -> void
{
  bind(sensor.temperature, print_warning);
}
```