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 27, 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 Syntax

2.1 Comment

Anything following a hash (`#`) that is not part of a string literal will be treated as a comment.

2.2 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`).

2.3 Identifiers

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

In addition to the rules listed above, identifiers cannot contain two consecutive underscores, or end with `_?`.

2.4 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>
                     | <expression> <binary-operators> <expression>
                     | <unary-operators> <expression>
                     | <identifier> "=" <expression>
                     | "(" <expression> ")"
                     | <identifier> "(" <arg-list> ")"
                     | <identifier> "." <identifier>
                     | <identifier> "." <identifier> = <expression>
```

Operator precedence follows normal conventions.

2.5 Lists

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

2.6 Declarations

2.6.1 Variable Declaration

```
    <primitive-type> ::= "int" | "float" | "bool" | "str" | "void"
     <compound-type> ::= "obj" | <primitive-type> "list"
     <variable-decl> ::= <compound-type> <identifier> ";"
<variable-decl-list> ::= "" | <variable-decl-list> <variable-decl>
```

2.6.2 Function Declaration

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

2.6.3 Object Type Declaration

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

2.7 Statements

2.7.1 Sequencing

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

2.7.2 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>) ";"
```

2.7.3 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> ")" ";"
```

2.7.4 All Statements

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

3 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.

3.1 Data Types

Propeller has 5 primitive types: `int` is a signed 32-bit integer type; `float` is a 64-bit IEEE 754 floating point type; `bool` is a boolean type, stored in a single byte; `str` is a syntactic sugar for integer lists, which stores UTF-8 encoded characters of a string; and finally a `void` type.

Propeller has a list type. Lists are immutable. All items in one list must be of the same type. Only values of primitive types are allowed in lists.

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 a `external` keyword.

3.2 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.

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.)

3.3 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.

4 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, terminating execution with exit code, printing values of primitive types, as well as basic list operations.

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.

5 Examples

5.1 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;
}
```

5.2 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 t) -> void
{
  if (t > 100)
  {
    print_str('Boiling!');
  }
}

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