Propeller

Isra Ali, Gwendolyn Edgar, Randy Price, Chris Xiong

Why Propeller?

- Simple, small, easy-to-learn reactive programming language
- Applications in UI, sensor devices, and many more

Language Principles

- User-defined data structures (objects)
- Reactive programming
- Clean, concise, unambiguous syntax

Language Basics

Program Structure

- Object definitions
- Global variables
- Function definitions
    - Program execution begins at special function “main”

Control Flow

- If/elif/else statements
  - Each if statement may be followed by 0 or more elif statements, and one optional else statement.
- For loops
  - Designed to iterate a given number of times
  - Looping variable typed/declared/initialized automatically
- While loops
- Continue/break statements for loops

Scoping

- Local variables, formal parameters, and global variables are all visible from within function bodies
- Scoping follows C-like precedence (local, formal, global)

Types

Primitive Types

Type	Memory
int	4 bytes
float	4 bytes
bool	1 bit
str	varies
void	n/a

Objects

- Users can create custom types called objects
- Similar to records/structs
- Object properties (akin to fields) may have functions bound to them such that these functions are called upon assignment to the property

Standard Library

Printing

- print floats = printf
- print ints = print
- print bool = printb
- print string = prints

Notable Features

Objects

- Users can create custom types, similar to structs/records
- The PERIOD operator ('.') is used to initialize properties of objects, retrieve their values, and bind functions to them

```
objdef Jumbo
{
  str name;
  int age;
  float gpa;
}
```

Binding

- When a function is bound to an object's property, assignment to that property will result in a call to that function

```
fn celebrate(int old, int new) -> void
{
  if new == old + 1
  {
    prints('Happy birthday!');
  }
  elif new == old
  {
    prints('Not your birthday :(');
  }
  else
  {
    prints('Illegal aging!');
  }
}

fn init() -> int
{
  Jumbo jeff;

  jeff.name = 'Jeff';
  jeff.age  = 24;
  jeff.gpa  = 3.72;

  # celebrate is called when jeff's age is changed
  bind(jeff.age, celebrate);

  # 'Happy birthday!' is printed
  jeff.age = 25;

  # celebrate is no longer called when jeff's age is changed
  unbind(jeff.age, celebrate);

  # nothing happens!
  jeff.age = 24;
}
```

Lists, Strings

- Immutable array type of primitive types
- 0-indexed
- int list, float list, str list, etc

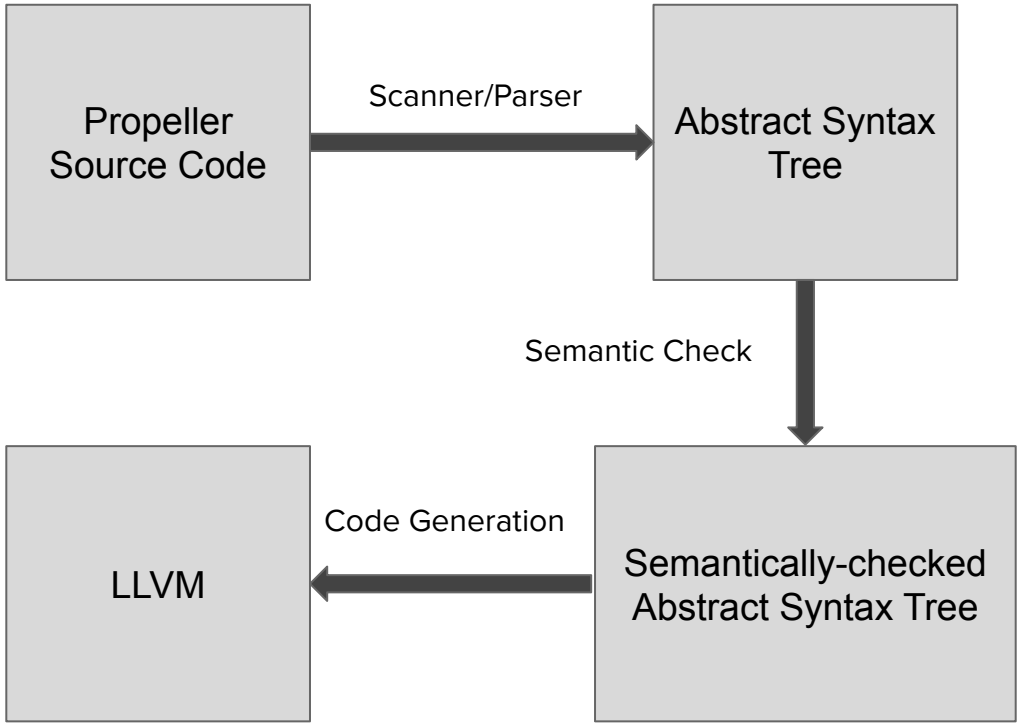
```
fn init() -> int
{
  int list A;
  str list S;

  A = [3, 2, 1];
  print(A[2]);          # prints 2

  S = [‘hi’, ‘howdy’];
  prints(S[1]);         # prints ‘howdy’

  return 0;
}
```

Architecture


Propeller
Source Code
Scanner/Parser
Abstract Syntax
Tree
Semantic Check
LLVM
Code Generation
Semantically-checked
Abstract Syntax Tree

Syntacic Sugar

If/Elif/Else

- Elif chains are syntactic sugar for nested if statements with one optional terminating else statement
- If/elif/else statements are converted to nested if statements during code generation

For Loops

- Syntactic sugar for while loops
- Looping variable internally declared as integer, incremented after each iteration of loop

SAST Construction

Additional Passes

- Find for loops and the names of looping variables
- No duplicate function definitions, object definitions

Code Generation

Object Definitions

- Each object definition has its own LLVM type
- LLVM type determined by number/types object propertites

Bind/Unbind

- Function binding occurs during code generation
- List of bound functions to object variables' properties is generated at the beginning of and updated during the construction of the LLVM function

Challenges

- OCAML LLVM API documentation is dense
- Not many resources easily available outside the ones suggested in class
- End of year rush

Demo Time!