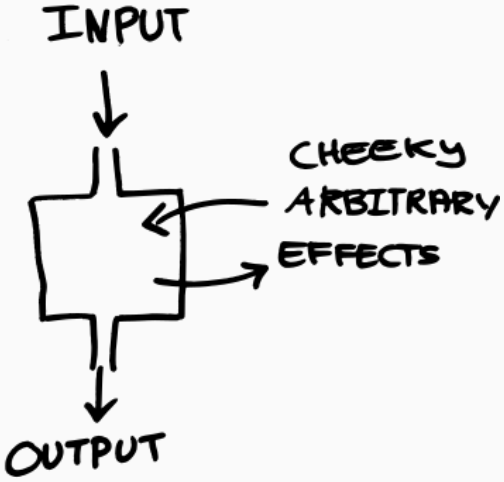
FunC: Functional Programming in C

Fan Gao (fg2432)
May 15, 2019

Introduction

Functions

Procedures



Source: FPComplete

Motivation:

- C is easy to learn and compile
- But C is imperative and some cool language features are missing in C
- Functions in C are second-class citizens as one can only write higher-order functions through function pointers

Goals:

- Fuse imperative and functional programming in C-like syntax
- Provide syntactic sugar to aid functional programming

Language Overview

General Attributes:

- Statically typed
- Lexical scoping
- Pass-by-value
- One main in global scope

Unique Features:

- Variable-length arrays
- Anonymous functions
- First-class functions
- Higher-order functions
- Map & reduce
- Piping

Keywords & Data Types

bool	else	false
float	for	func
if	int	map
new	print	printf
prints	reduce	return
true	void	while

Primitive	Derived
int	string
bool	array
float	func
void	

Operators

(), [], identifier(typ id1, typ id2, ...)	Parenthesized expr, array access, function call
-, !	Unary minus, logical negation
*, /, %, +, -	Multiplicative, additive
<, >, <=, >=, ==, !=, &&, ||	Relational, equality, logical and, logical or
new, =>	Malloc, lambda
|>, |	Pipe & bar
=, ,	Assignment, comma

Control Flow & Functions

if … else ...

```
bool pass(int score)
{
    if (score >= 60)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

while

```
void count_ten()
{
    int cnt;
    cnt = 0;

    while (cnt < 10)
    {
        cnt = cnt + 1;
    }
}
```

for

```
void print_num(int num)
{
    int i;

    for (i = 1; i <= num; i = i + 1)
    {
        print(i);
    }
}
```

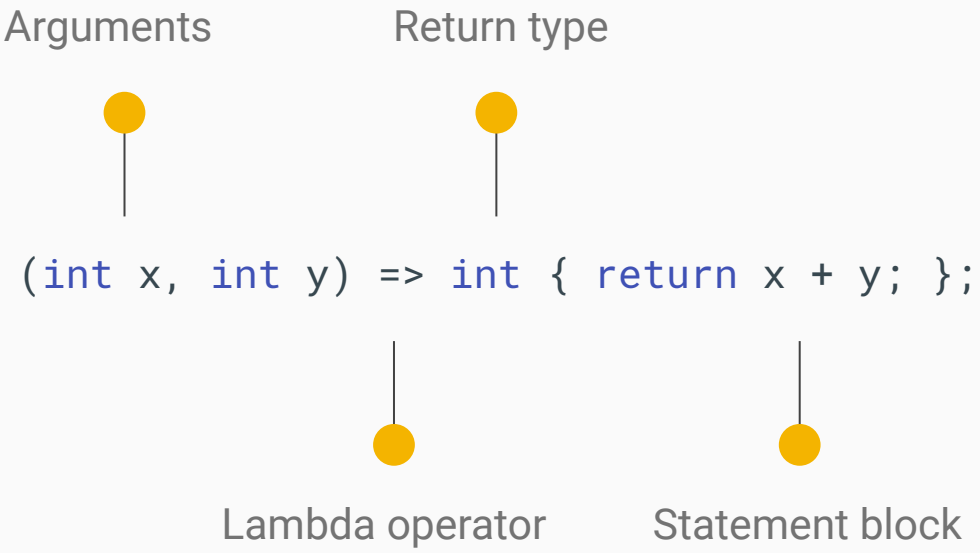
```
void square(int[] arr, int length)
{
    int i;
    for (i = 0; i < length; i = i + 1)
        arr[i] = arr[i] * arr[i];
}

int[] square2(int[] arr, int length)
{
    int i;
    new_arr = new int[length];

    for (i = 0; i < length; i = i + 1)
        new_arr[i] = arr[i] * arr[i];

    return new_arr;
}

int main()
{
    int[3] arr;
    int[3] new_arr;

    arr = [1, 2, 3];
    /* In-place */
    square(arr, 3); /* arr = [1, 4, 9] */

    /* Return a new array */
    new_arr = square2(arr, 3); /* arr = [1, 4, 9] */
    /* new_arr = [1, 16, 81] */

    return 0;
}
```

Variable-Length Arrays

- Need a basic data structure to demonstrate some of the features
- Support variable-length arrays
- Arrays are allocated on the heap

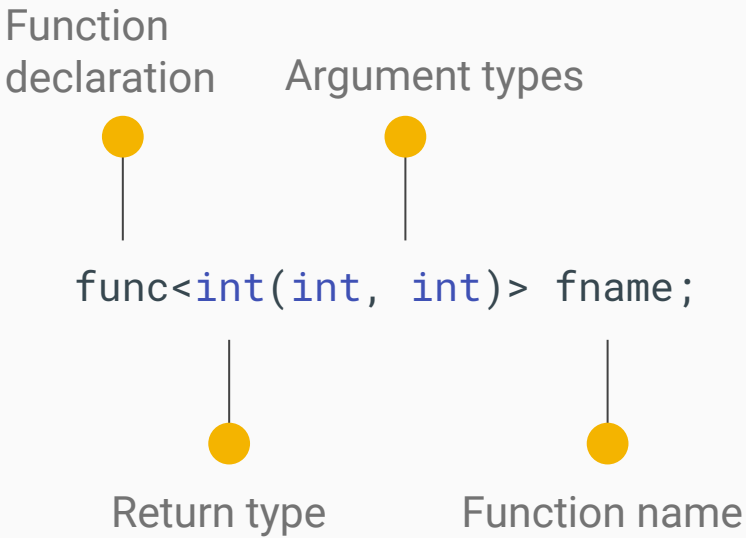
Anonymous Functions

These self-contained inline functions are quick and easy to write to replace local operations that don't need to be named and added to the namespace.



First-Class Functions

- Open the way for functional programming
- Make it easy to pass around units of behavior and work with functions like with any other values



Higher-Order Functions (1)

- A function that accepts another functions as an argument
- Unleash the power of first-class and anonymous functions

```
int twice(func<int(int)> f, int x)
{
    return f(f(x));
}

int main()
{
    func<int(int)> square;
    int r;

    square = (int x) => int { return x * x; };

    int r = twice(square, 2); /* 16 */

    return 0;
}
```

```
func<void()> greet(bool formal)
{
    func<void()> greet_formal;
    func<void()> greet_casual;

    greet_formal = () => void { prints("How are you?"); }
    greet_casual = () => void { prints("What's up?"); }

    if (formal) {
        return greet_formal;
    }

    return greet_casual;
}

int main()
{
    func<void()> greet_casual;

    greet_casual = greet(false);
    greet_casual(); /* What's up? */

    return 0;
}
```

Higher-Order Functions (2)

Not only can functions accept behaviors through arguments but they can also return behaviors.

Map & Reduce

- Avoid writing for loops and managing the states
- FunC's syntactic sugar for writing compact and elegant code

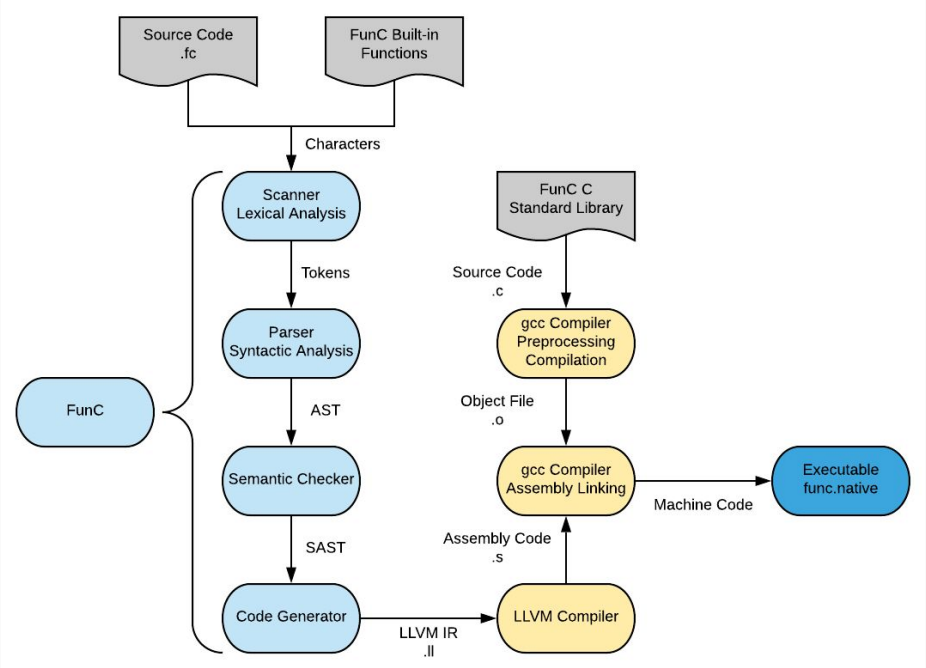
```
int main()
{
    int[] numbers;
    int len;
    func<int(int)> square;
    func<int(int, int)> sum;
    int[] squares;
    int s;

    numbers = [1, 2, 3, 4, 5];
    len = 5;
    square = (int x) => int { return x * x; };
    sum = (int x, int y) => int { return x + y; };

    squares = map(square, len, numbers); /* {1, 4, 9, 16, 25} */
    s = reduce(sum, len, squares); /* 55 */

    return 0;
}
```

Piping

- Unix-like piping
- Allow users to decompose a long series of successive function calls into a functional pipeline

```
int main()
{
    int[] numbers;
    int len;
    func<int(int)> square;
    func<int(int, int)> sum;
    int s;

    numbers = [1, 2, 3, 4, 5];
    len = 5;
    square = (int x) => int { return x * x; };
    sum = (int x, int y) => int { return x + y; };

    s = numbers |> map(square, len) |> reduce(sum, len) |; /* 55 */

    return 0;
}
```

Compiler Architecture



- Follows a traditional compiler architecture design
- 5 modules and 2 interfaces
- Standard library functions written in FunC
- Integration with C standard libraries
- Backed by LLVM and gcc compilers and linker

Testing & Debugging

- Shell script for automated testing
- Include failing and passing cases
- 100% test coverage for all features (110 tests)
- Have a Utility module for debugging

Lessons Learned

- Gained much deeper understanding on language design and the inner working of a compiler
- Do everything in a recursive manner
- Learn to read LLVM IR code
- Debugging a compiler is not easy
- Scope workload appropriately

Demo