Team:

Shankara Pailoor (sp3485)
Panchampreet Kaur (pk2506)
Graham Barab (gmb2154)

The Cimple Programming Language

We hope to build a statically typed language based off C with features that enable the programmer to write expressive server side programs with the same ease as he or she would with a language like python or ruby. These features include inheritance, interface types, anonymous functions and closures.

We have chosen C as the basis of our language due to its status as a "low-level" language. The ideas expressible in the language have close relationships with the resulting CPU instructions the compiler generates. This proximity of the language to the computer hardware in this sense promotes fast execution speed. Therefore, our language would benefit programs that require fast execution speed, such as any software that processes or renders signals in real time, such as digital audio production workstations and plugins, video editing and rendering, real-time (or otherwise) 3D graphics, video games, etc.

Additionally, we are introducing some higher level language features, including object-oriented principles such as structs with inheritance and methods, anonymous functions with closures, and interfaces. This extends the ways a programmer may organize code, and increase the speed with which more abstract concepts can be represented in code. Types of programs that can benefit from this include those that rely on asynchronous communication. An example would be client software interacting with a server, and vice verse.

Identifiers:

Identifiers consist of letters and digits [a-zA-Z][0-9*] along with underscores. There are a set of reserved identifiers described below.

Scope and Delimiters:

We keep to the C style using `{}` to specify scope

Keywords:

We add the following: `function`, `interface`, `extends`, `s uper`, `implements`, `make` and `clean`.

Constants:

We keep the same integer, character and float constants as C.

Comments:

We support /* */ block comments and // line comments

Primitive Types:

We keep the same primitive types as C but also include `string`.

Operators:

We use the same binary and unary operators as C but also support '+' for string types which indicates concatenation and '==' which indicates the values are the same.

Objects and Structs:

We don't have a new container type for a Class but instead keep the same struct objects from C. We don't support anonymous struct definitions within the struct body. Accessing members of a struct has the same syntax for pointers and non-pointers.

Private vs. Public

We don't have a concept of public or private variables or methods.

Methods:

Methods are functions which either retrieve state or modify state of a struct. There are two implicit methods which are associated with every class 1) the constructor, 2) destructor. For instance suppose we had a struct Person object defined below.

```
struct Person {
      string name;
}
```

Then to allocate and assign this struct to a variable, one would write

```
Struct Person p;      /* Stack variable - default constructor takes
no parameters */

Struct Person p("Joe");     /* Stack variable - constructor with
string parameter */

struct Person *p = make Person(); /* Heap variable */
```

Our language uses the make keyword to allocate space for the struct on the heap and then invokes the default constructor Person(). The default constructor can be overridden, by defining one within the body of the struct. As an example,

```
struct Person {
      string name;
      Person(string name)
      {
            /* Constructor */
            name = name;
      }

      ~Person()
      {
            /* Destructor */
      }
}
```

Other methods are defined outside as follows

```
string (Person *p) getName()
{
      return p.name;
}
```

Like the constructor there are default destructors which are invoked with the `clean` keyword if the instance is stored on the heap (i.e. was instantiated with the “make” keyword), or is called implicitly when the object goes out of scope.

```
struct Person *p = make Person(“myname”);
printf(“%s\n”, p.getName());
clean p;
```

A stretch goal of this project might be to include a garbage collecting runtime.

Inheritance:

Inheritance is denoted by specifying the keyword `extends` along with the name of the parent struct after the struct name. We only support single inheritance. For example

```
      struct Person {
            string name;
            Person(string theName)
            {
                  name = theName;
            }
      }
      struct Student extends Person{
            string uni
            Student(string name, string theUni)
            {
                  super(name);
                  uni = theUni;
            }
      }
```

The `super` keyword allows one to access a parent struct's methods and can only be used within the struct definition. In the above example we are invoking the constructor of the parent class by calling `super(name);`. If we wanted to access a method then we could write `super.getName();`

Overriding and Accessing Base Class Methods From Derived Classes

Methods can override a parent struct's method simply by defining a new function with the same signature. For example, suppose Person had a method getId(). Then we could override the method in Student by defining:

```
String (Student *s) getId()
{
      Return Person(s).getId() + s.uni();
}
```

One thing to note in the above example is we are calling `Person(s)` which casts the struct Student to a struct Person and invokes the getId() since `super` can only be used within the struct definition. If Person were not a parent to Student then this would cause the compiler to throw an error.

Interfaces:

An interface is defined as follows

Example:

```
interface pet {
      Void feed();
      Void walk();
}

struct dog implements pet {
      eat()  { /* Dog-specific method */}
      feed() { /* Pet interface */}
      walk() { /* Pet interface */}
}
```

Anonymous Functions

In our initial design we decided anonymous functions should have a different syntax in order to support closures.

Syntax for defining an anonymous function:

```
func(<returnType>)(...arguments) {
      //define function
}
```

An example:

```
func(int)(int i) {
      // Do something with i
      return i;
}
```

Syntax for naming a function parameter which is an anonymous function:

```
func(void)(int) *function_name
```

To be explicit, `function_name` is an alias for the anonymous function which takes an int parameter and returns a void.

Example:

```
void some_function(int x, func(int)(int) *fn)
{
      int result = fn(x);
      printf("%d\n", result);
}

int main(int argc, string* argv)
{
      for (int i = 0; i < 5; ++i) {
            some_function(i, func(int)(int j) {
                  return j + 1;
            });
      }
}
```

Example Program - Processing an Audio Signal

```
struct Signal {
        float* samples
        int size;

        Signal(int signalLength, func(void)(float*, int) *defineSignal) {
                size = signalLength;
                samples = make float[size];
                defineSignal(samples, size);
        }
        ~Signal() {
                clean samples;
        }
}

interface SignalProcessor {
        struct Signal* processSignal(struct Signal*);
}

struct FilterKernel extends Signal implements SignalProcessor {

        FilterKernel(int signalLength, func(void)(float *, int) *defineSignal) {
                Super.make(signalLength, defineSignal);
        }

        struct Signal* processSignal(struct Signal* sig) {
                /* Create a new signal which is the convolution of this signal with "sig" */

                struct output = make Signal(size + sig.size,
                        func(void)(float* x, int N) {
                                for (int i = 0; i < size + sig.size; ++i) {
                                        for (int j = 0; j < size; ++j) {
                                                if (j > i)
                                                        x[i] = 0;
                                                else
                                                        x[i] = samples[j] * sig.samples[i - j]];
                                        }
                                }
                        }
                });
                return output;
        }
}

const float pi =  3.14159265359;

int main(int, string*) {
        float duration = 1.0;
        int sampleRate = 44100;
        int signalSize = sampleRate * duration;
        float startFreq = 20.0;               // 20hz
        float endFreq = 20000.0;      // 20kHz

        /* Create an sine sweep - a sine wave
        whose frequency sweeps through the
        audible spectrum of 20Hz - 20kHz
        over 1 second */

        struct Signal* sinesweep = make Signal(signalSize,
                func(void)(float* x, int sigSize) {
                /* This is the beginning of an anonymous function */
                int i = 0;
                while (i < sigSize) {
                        float freqModulator = exp((i / sigSize)
                                        * log(endFreq / startFreq)) - 1;
                        x[i] = sin(((duration * (2 * pi * startFreq)
                                        / (log(endFreq / startFreq))) * freqModulator));
                        ++i;
                }
        });

    /* Create a low pass filter convolution kernel */
        struct Signal* lowPassFilter = make FilterKernel(signalSize,
                func(void)(float *x, int sigSize) {
                        int i = 0;
                        float cornerFreq = 0.25;

                        while (i < sigSize) {
                                if (i == sigSize / 2) {
                                        x[i] = sin(2 * pi * cornerFreq)
                                        * (0.42 - (0.5 * cos(2 * pi * i / sigSize)
                                                + (0.08 * cos(4 * pi * i / sigSize))));
                                } else {
                                        x[i] = (sin(2 * pi * cornerFreq * (i - sigSize / 2.0))
                                                / (i - sigSize / 2.0))
                                               * (0.42 - 0.5
                                               * cos(2 * pi * i / sigSize)
                                                + 0.08 * cos(4 * pi * i / M));
                                }
                        }
        });

        /* Now use the lowpass filter to remove frequencies > 11,025Hz from
        the sine sweep (should begin to go silent ~halfway through the sweep). */

        struct Signal* filteredSweep = lowPassFilter->processSignal(sineSweep);

        /* Do something with the filtered signal, then delete it */

        clean filteredSweep;
        clean lowPassFilter;
        clean sinesweep;

        return 0;
}
```

References

- Smith, Steven W. The Scientist and Engineer's Guide to Digital Signal Processing. San Diego, CA: California Technical Pub., 1997. Print.
    - Code example uses formula for a windowed-sinc low pass filter on page 290.

- Stan, Guy-Bart, Jean-Jacques Embrechts, and Dominique Archambeau. "Comparison of different impulse response measurement techniques." Journal of the Audio Engineering Society 50.4 (2002): 249-262.
    - Code example uses formula for logarithmic sine sweep presented in this paper.