The Unnamed Language

Programming language for the UVic CSC 435 Compiler Construction class. The grammar for the language can be found [here](#).

View this repo on [Github](#).

Compiling

The compiler is built with `make`.

- `make grammar` runs Antlr to generate lexer and parser classes
- `make compiler` compiles the compiler
- `make clean` removes compile and build files

To do a full clean and build run,

```
make clean; make
```

Running

The compiler can be run against langauge files to generate [Jasmin assembler code](#). The generated `.j` files can be run through Jasmin to create `.class` files. The `.class` files can be run with `java`. By default, a `.j` file with the same base name as the input file is generated.

The built compiler is located in the `bin/` directory.

```
cd bin/
java Compiler path/to/file.ul
java jasmin.Main file.j
java file
```

A small shell script has been created to make running unnamed language code easier. In the project root directory you can use the `run.sh` to compile and run an unnamed language file.

```
./run.sh file.ul
```

Options

- `-o outfile` Specify a file to save the pretty printed output or dot mode output. If no file is given, output is sent to a file with the same name as the input file (different extension). Output can be sent to stdout if the outfile is `<stdout>`.
- `-ir 1|0` IR generation mode. If `1` then the IR text will be generated to the file and sent to `outfile`. *(Default 0)*
- `-p 1|0` Pretty print mode. If `1` then file will be pretty printed after type checking. *(Default 0)*
- `-d 1|0` Dot mode. If `1` then the output is in the DOT language. *(Default 0)*

Testing

There are a bunch of .ul language files that can be tested against the compiler. Throughout the course I will update the test script to check whether the latest requirements are met. Use the script `test.sh` to run all tests.

```
# Run the tests
./test.sh
```

`.ul` files in the `tests/output` directory have a corresponding `.txt` file. When the tests are run, the IR is sent through `./codegen` and `jasmin`. The output of the Java `.class` file is compared to the `.txt` file. If they are different then an error is thrown. These tests allow me to ensure the behaviour of the programs the compiler generates is expected.

Examples

Fibonacci Numbers

```
// fib.ul
int fib(int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  } else {
    return (fib(n - 1) + fib(n - 2));
  }
}

void print_fib(int x) {
  print "The ";
  print x;
  print "th Fibonacci number is ";
  println fib(x);
}

void main() {
  int x;
  x = 0;

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

```
$ ./run.sh fib.ul
The 0th Fibonacci number is 0
The 1th Fibonacci number is 1
The 2th Fibonacci number is 1
The 3th Fibonacci number is 2
The 4th Fibonacci number is 3
The 5th Fibonacci number is 5
The 6th Fibonacci number is 8
The 7th Fibonacci number is 13
The 8th Fibonacci number is 21
The 9th Fibonacci number is 34
```

Factorial

```
// fact.ul
int factorial(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

void main() {
  int x;
  x = 1;

  while (x < 10) {
    print "The factorial of ";
    print x;
    print " is ";
    println factorial(x);

    x = x + 1;
  }
}
```

```
$ ./run.sh fact.ul
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 6 is 720
The factorial of 7 is 5040
The factorial of 8 is 40320
The factorial of 9 is 362880
```

Differences From Default Spec

My compiler has a few changes from the default specification. These are

- Variables can be declared anywhere in a function and their use is scoped to the current block
- int < float subtype relationship
- Functions with same name but different type signature are allowed

Variable Declarations and Scopes

I have changed by grammar to treat variable declarations as statements. This means that variables can be declared anywhere in the function. Their use is scoped to the current block. My environment creates a new scope when a function or block is entered. For example, the following code was previously not in the language, now it is.

```
void main() {
    print "hello"
    int x;

    if (true) {
        int y;
        x = y + 1;
    }
}
```

If a scope is exited, then the variable becomes out of scope and is not allowed. For example, this would result a "Variable is not declared" error.

```
void main() {
    int x;

    if (true) {
        int y;
    }
    x = y; // not allowed because y is not in scope.
}
```

Subtype Relationship

With the single subtype relationship, this code is allowed

```
void main() {
    int a;
    float b;
    float c;

    a = 1;
    b = 1.1;
    c = a + b;
}
```

Because of this relationship, test wSt3.2.2.ainvalid.ul has been removed from the test set.

Function Declarations

Type signatures are used when comparing function declarations. Two functions can have the same name as long as their signatures are different. For example, this is allowed.

```
void foo() {}
void foo(int x) {}
void main() {}
```

Dot Graphs

Dot language programs can be produced with the `-d 1` option to the compiler.

For example, if you have this file

```
// hello.ul
void main() {
    println "hello world";
}
```
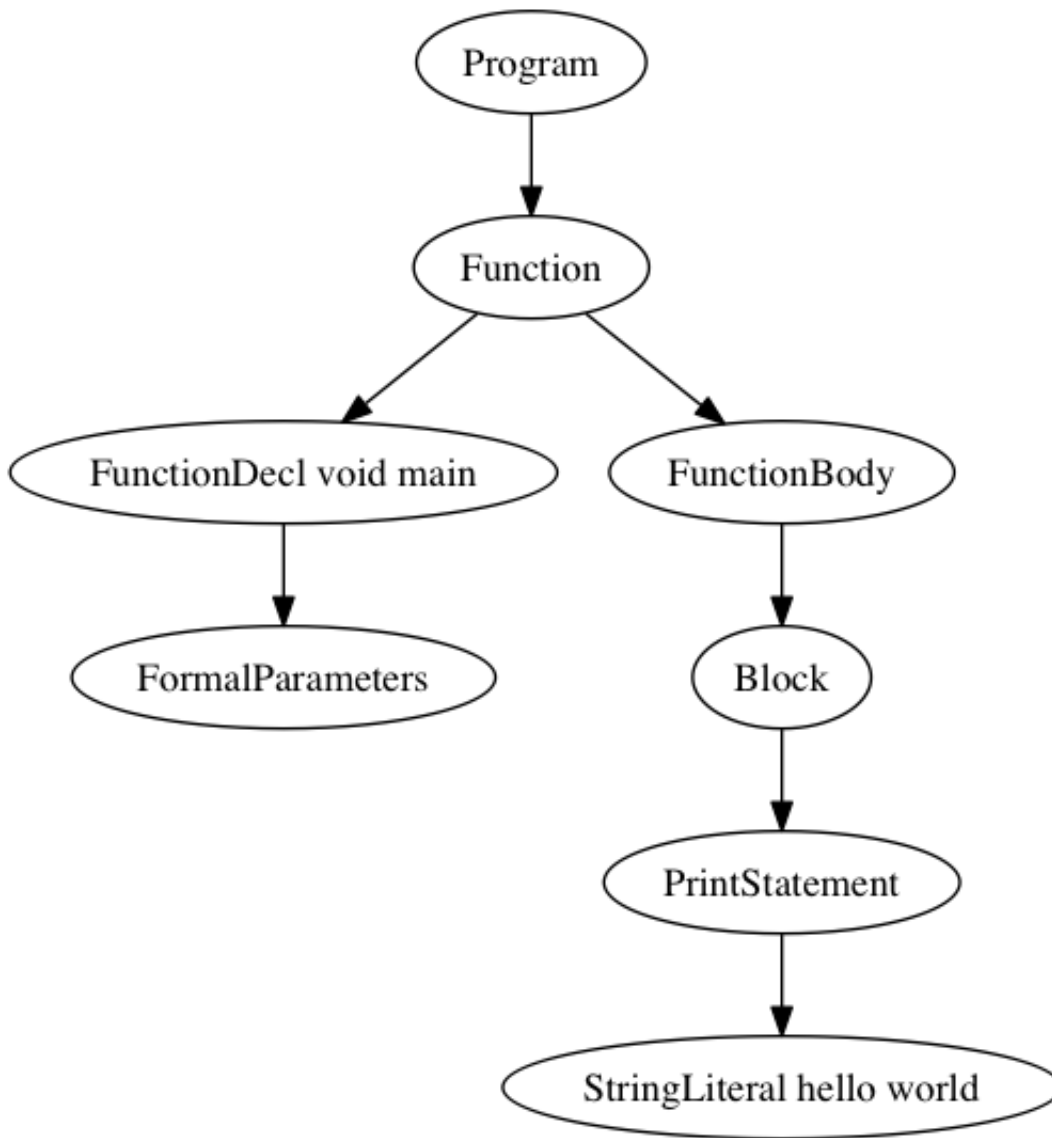
You can compile it with

```
java Compiler -d 1 -ir 0 -o hello.dot hello.ul
```

You can then use the dot program to create a png image file and open it

```
dot -Tpng hello.dot -O && open hello.dot.png
```

The output should be

Licenses

All third party code is referenced in the LICENSES file.

TODO

- [x] Lexer
- [x] Parser and AST generation

- [x] Pretty printing
- [x] Dot output
- [x] Syntax analysis
- [x] Type checking
- [x] Intermediate code generation
- [x] JVM code generation