

# 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 language files to type check them. Options can be specified to change the default behaviour.

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

```
cd bin/  
java Compiler path/to/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 stdout.
- `-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
```

The latest provided tests are in `accept/provided` for all `*_valid.ul` files and `reject/provided` for all `*_invalid.ul` files.

## 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() {}
```

## Example

---

This following code finds the factorial of 8.

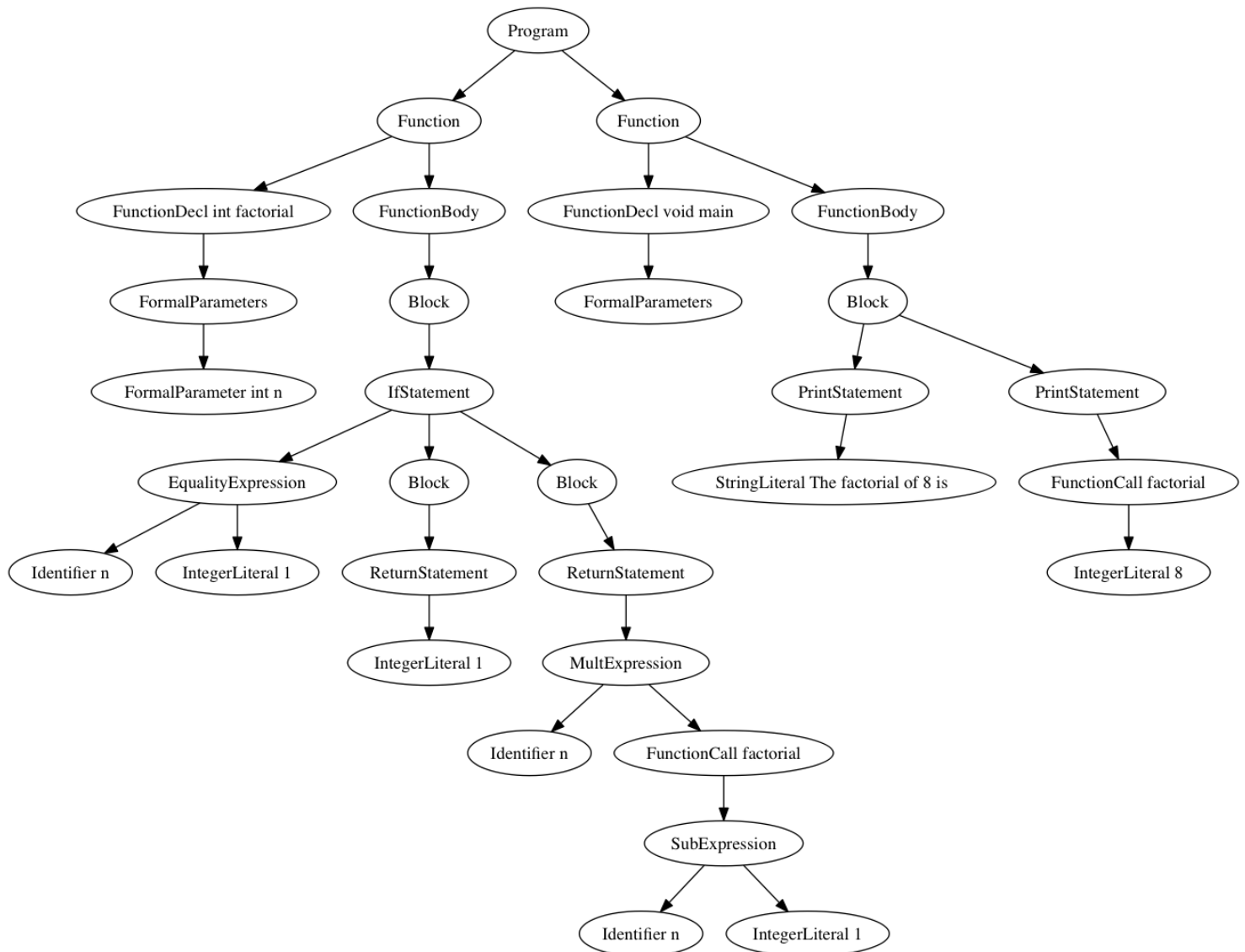
```

int factorial (int n) {
    if (n == 1) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}

void main () {
    print "The factorial of 8 is ";
    println factorial(8);
}

```

The following AST is produced.



# 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";
}
```

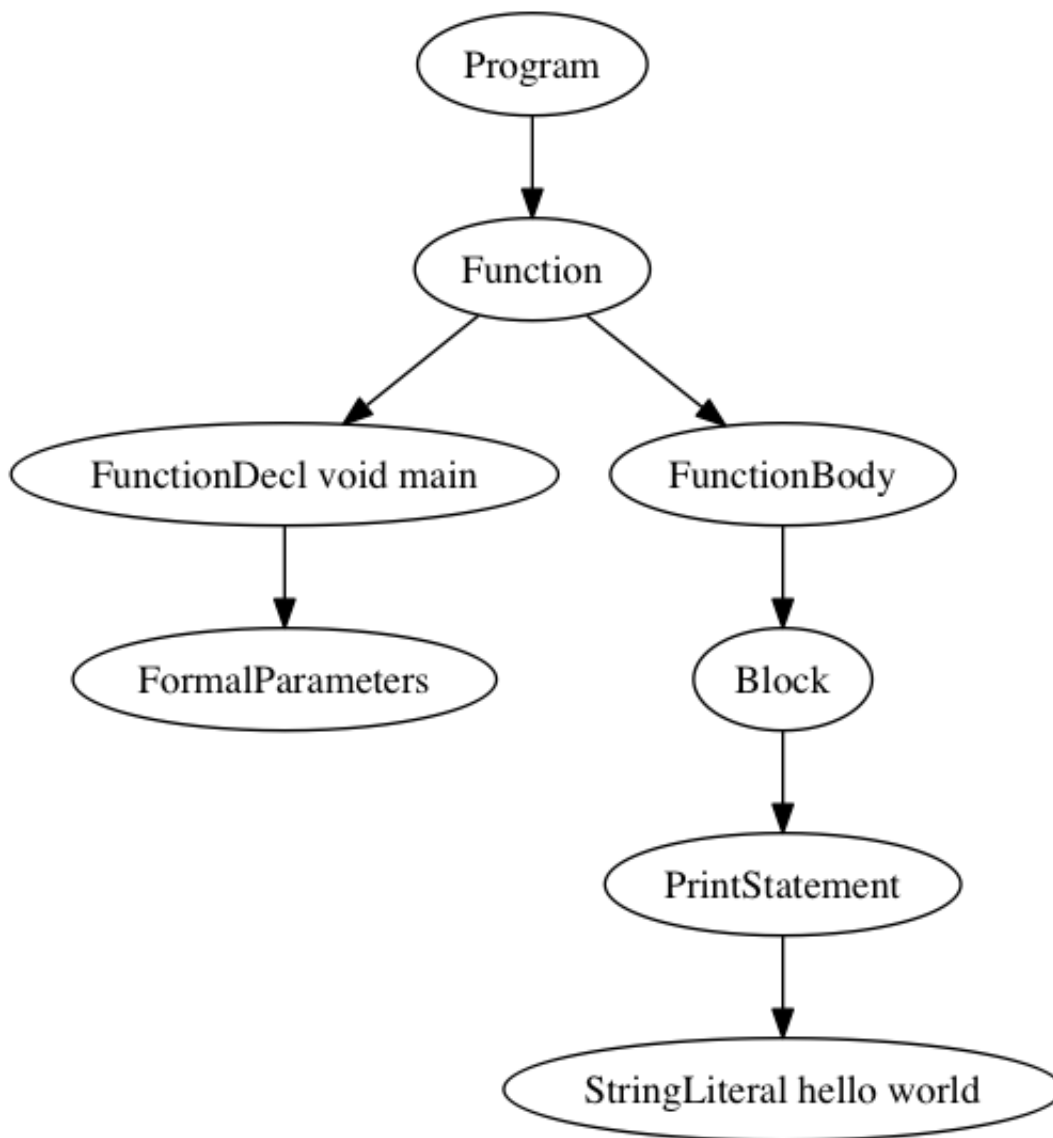
You can compile it with

```
java Compiler -d 1 -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

- ☐ Intermediate code generation
- ☐ Register allocation
- ☐ Machine code generation
- ☐ Assembly and linking