1 Using the Compiler

A quick overview of all commands can be triggered by running the compiler with “-h” or “—help”.

1.1 Compiling a file

This obviously shall be the simplest part.

```
java -jar dist/compiler488.jar -i ../a1/A.488
Parsing input file... [DONE]
Running semantic analysis... [DONE]
```

The option “-i” expects an input file. That’s it…

If you pass an invalid file the output is more interesting:

```
java -jar dist/compiler488.jar -i ./tests/semantics/S49/fail/2.488
Parsing input file... [DONE]
Running semantic analysis...
[ERROR (S49)]: Forward declaration and method definition shall have the same signature.
=> compiler488.ast.decl.RoutineDeclaration:
File "/mnt/hgfs/compilerCourse/a3/tests/semantics/S49/fail/2.488":
[Method definition] in line 4 to 7 >
     es)\n \n  proc b2str(boolean yes) \n begin \n return \n end \n \n b2st
               ~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
=> compiler488.ast.decl.ForwardDeclaration:
File "/mnt/hgfs/compilerCourse/a3/tests/semantics/S49/fail/2.488":
[Forward-declaration] in line 2 >
     begin\n  forward proc b2str(integer yes) \n \n proc b2str(bo
              ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[FATAL-ERROR]: Input file contains semantic errors!
```

Note that with only the input file as parameter, nothing really happens. The compiler will only validate the file but nothing else!

1.2 Code generation

A very interesting feature of the compiler is that it can generate compliable code. Currently, it can dump out 488 code. This is useful for validating the AST in a bootstrap process. To do this, you take one huge input file assessing hopefully all possible AST node configurations. Then you let the compiler dump out a 488 code file. It is virtually equivalent, if the AST is correct, but you need to verify this manually. Once you have done this (which can be laborious) you can passed the dumped file as input file and now do binary equivalence tests which are automatic. So once you have a bootstrap file, all future AST tests can

be automated this way. To generate a 488 dump, you need to specify the input file and also an output file using the "-o" command. Since the format defaults to 488, you don't need to specify it:

```
 -i ../a1/A.488 -o ./tests/488-dump/A.488
Parsing input file... [DONE]
Running semantic analysis... [DONE]
Generating output file "/mnt/hgfs/compilerCourse/a3/tests/488-dump/A.488"... [DONE]
```

Another format is C++11. This is interesting because it generated compliable C++ code that is semantically equivalent, easy to read and debuggable. This is especially useful when it comes to Assignment 5 and code generation. Basically it allows you to compare the output of a 488 execution run with the one of a binary program generated by a C++ compiler. This will be a huge advantage later, even if it does not help much right now. For this, we will add the capability of running and comparing both programs automatically. In contrast to above, you need to tell the compiler that it should generate C++ code:

```
-i ../a1/A.488 -o ./tests/c++-dump/A.cpp -f=cpp
Parsing input file... [DONE]
Running semantic analysis... [DONE]
Generating output file "/mnt/hgfs/compilerCourse/a3/tests/c++-dump/A.cpp"... [DONE]
```

The compiler also supports generation of binary executables. This will only work if you have a recent version of GCC (4.5 or higher) available. On Windows, the Intel Compiler 2013 is used through a BAT script called "run-icl.bat" that you will find next to the compiler's JAR. You may need to modify the exact path to your Intel Compiler's "bin" directory. The following demonstrates a binary compilation:

```
java -jar dist/compiler488.jar -i ../a1/A.488 -o ../a1/A -f=x86
Checking for working C++11 compiler... [DONE]
Parsing input file... [DONE]
Running semantic analysis... [DONE]
Generating executable file...
Compiling code with C++ compiler... [DONE]
Generating executable file... [DONE]
```

As long as your system is configured properly this is a seamless operation and generates an executable file called "A" in directory "../a1/". If the compilation fails, you may need to explicitly pass a compatible C++ compiler via command line:

```
java -jar dist/compiler488.jar -i ../a1/A.488 -o ../a1/A -f=x86 -cpp clang++
```

The "-cpp" option may have one argument, which defaults to "g++" on all other systems than Windows. This argument shall provide the path to a C++ compiler. In the above example, we switched the compiler to "clang++". Note that you need a quite recent version of Clang, preferably 3.2 or higher, otherwise compilation will likely fail.

1.3 Code execution

Right now, you need a working C++ compiler (see above) to execute code. The compiler is able to generate C++ code in the background and also compile it to native code using an external C++ compiler. If you also want to run this file immediately or not even bother to have that binary file in a persistent location, you can just invoke:

```
java -jar dist/compiler488.jar -i ../a1/A.488 -cpp
Checking for working C++11 compiler... [DONE]
Parsing input file... [DONE]
Running semantic analysis... [DONE]
Generating executable file...
Compiling code with C++ compiler... [DONE]
Generating executable file... [DONE]
Running generated code:
1 + 2 = 3
1 - 1 = 0
1 * 2 = 2
10 / 2 = 5
-1 + 2 = 1
```

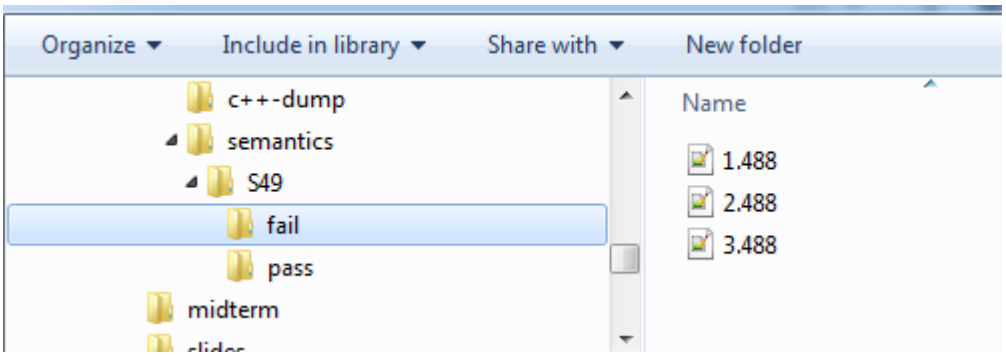
The compiler will manage and remove all temporary files automatically and run the executable immediately, as you can see above. It also supports command line input, if your program needs it. Again you can specify the C++ compiler explicitly, since the “-cpp” option is the same as above. If you want to keep the executable and immediately run the code, you can do:

```
-i ../a1/A.488 -o ../a1/A -f=x86 -cpp --exec
Checking for working C++11 compiler... [DONE]
Parsing input file... [DONE]
Running semantic analysis... [DONE]
Generating executable file...
Compiling code with C++ compiler... [DONE]
Generating executable file... [DONE]
Running generated code:
1 + 2 = 3
1 - 1 = 0
1 * 2 = 2
10 / 2 = 5
-1 + 2 = 1
true | false = true
true & false = false
!true = false
1 = 2 ? false
```

This is basically the same as code generation, just that you specify the sub-option “—exec” to its parent option “-cpp” (this is why it must immediately follow it, so “—exec –cpp” would fail!). “—exec” tells the compiler to run the C++ binary even though it generated a persistent executable. You may again specify the compiler explicitly like “-cpp clang++ --exec”.

1.4 Running a test-suite

During development one often runs into the issue of having a set of files that should pass and a set of files that should fail. The compiler has supported for that integrated. To use this feature you need to setup a directory structure whose layout is free for you to choose. The only requirement is that all files that are to be processed reside either in a “fail” or “pass” directory. Here is an example:



Our test-suite directory is “semantics” and all files in “fail” are meant to, well, fail compilation. If they don’t it is an error. To run this test-suite, just invoke the compiler with something like

```
java -jar dist/compiler488.jar -i ./tests/semantics/ --test-suite
```

In this case the input file is the root directory of the test-suite. “—test-suite” is a dependent option that must immediately follow the directory path. If you run this, the output may look like this:

```
Running test-suite on 4 out of 4 files...
TEST-SUITE RESULTS:

 -> S49 -> pass -> 1.488 [FAILED]

 -> S49 -> fail -> 1.488 [OK]
 -> S49 -> fail -> 2.488 [OK]
 -> S49 -> fail -> 3.488 [OK]
```

In this case all files that should fail have failed, this is why they are green and OK. But the red entry is a file that failed even though it was expected to pass. This is not really insightful is it? So to get an idea of what went wrong you could either run that file alone, which can be cumbersome especially when a whole bunch of stuff went wrong. Instead you can pass additional options to the compiler for each file. Right now only one option is supported. In this case you’d either want to pass something like “—verbose” to display all output of each test run.

```
java -jar dist/compiler488.jar -i ./tests/semantics/ --test-suite @-v
```

This parameter needs to be preceded by a special character “@”, since otherwise there is no way for the command line parser to determine if your argument should be passed to the compiler itself or each test run. Running the test-suite with “-v” reveals the following:

```
Running test-suite on 4 out of 4 files...
Parsing input file... [DONE]
Running semantic analysis...
[ERROR (S49)]: Forward declaration and method definition shall have the same signature.
=> compiler488.ast.decl.RoutineDeclaration:
File "/mnt/hgfs/compilerCourse/a3/tests/semantics/S49/pass/1.488":
[Method definition] in line 11 to 14 >
     \n \n  integer b2str2(boolean yes) \n begin \n result 1 \n end \n \n i
            ~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
=> compiler488.ast.decl.ForwardDeclaration:
File "/mnt/hgfs/compilerCourse/a3/tests/semantics/S49/pass/1.488":
[Forward-declaration] in line 9 >
     eturn\n  end\n \n  forward proc b2str2(boolean yes) \n \n integer b2st
                      ~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[FATAL-ERROR]: Input file contains semantic errors!
```

So that's the problem in the file that should pass. It was really an issue in the file, not the compiler, so we fix it and run the suite again without verbose:

```
TEST-SUITE RESULTS:

 -> S49 -> pass -> 1.488 [OK]

 -> S49 -> fail -> 1.488 [OK]
 -> S49 -> fail -> 2.488 [OK]
 -> S49 -> fail -> 3.488 [OK]
```

Everything alright! Or is it? Still, files that should fail have the problem that you don't know WHY they failed. Currently there is no way to assist you with that. You need to add something like "@-ll=error" which will only print errors but not annoy you with files that pass or any other non-important output during each test-run. A good idea is to keep files that should fail as small as possible, so that the chance that compilation fails for any reason other than the desired one is as low as possible. Then you just need to verify that all errors are the ones that should occur in each file.

```
java -jar dist/compiler488.jar -i ./tests/semantics/ --test-suite @-ll=error
```

In future, an arbitrary command line will be supported for each test-run allowing you to even compare execution of generated code with a C++ execution of the code and stuff like that.