# Exercise 6

Due 7/7/2017

## 1 Introduction

This exercise implements the code generation part of compiling StarKist programs into MIPS assembly. The details of the StarKist programming language (including its syntax and semantics) were described previously in exercise 5. The code generation module traverses the AST built by Bison, and builds an intermediate representation (IR) tree. The IR tree is then scanned in a post order fashion to further translate it into MIPS assembly. These two phases are summarized in Table 1. Note, that different flavours of IR exist, and for exam-

| | |
|---|---|
| Phase(1) | AST $\rightarrow$ IR |
| Phase(2) | IR $\rightarrow$ ASM file |

Table 1: The code generation module can be roughly divided into two phases: from AST to IR, and then from IR to the actual assembley file.

ple, gcc alone uses three kinds of intermediate representations:
generic, gimple and rtl. In our project, we will use a (single) low level IR (LIR), which is somewhat reminiscent of gcc's rtl. The term "low level" here indicates that our IR will contain only a limited set of commands, specified in Table 2. Evidently, choosing a low level IR affects the workload between the two phases mentioned above, and makes for an easier second phase on the expense of a harder first phase.

## 2 IR Specification

The entire set of IR commands is given in Table 2. We will briefly describe each of there commands now;

**BINOP** performs the binary operation between its two sub trees, the left sub tree is evaluated first.

**SEQ** indicates a sequence of two sub trees, where the left sub tree is evaluated first. we will use a low level IR (LIR) tree contains a very limited set of commands. Table

| | | |
|---|---|---|
| BINOP | MOVE | MEM |
| CJUMP | JUMP | CONST |
| LABEL | TEMP | CALL |
| FUNC | SEQ | |

Table 2: The LIR tree nodes.

## 3 Translation from LIR tree to ASM file

As mentioned earlier, the advantage of choosing a lower level IR tree is an easier translation from IR tree to assembly, and a harder translation from AST to IR tree. To create a graph visualization of the AST, please install graphviz and run

```
$ dot -Tjpeg -o ./AST_Graph.jpeg ./AST_Graph.txt
```

from `EX5/LINUX_GCC_MAKE`

## 4 Input

The input for this exercise is a single text file, the input StarKist program.

## 5 Output

The output is a MIPS asm text file that contains the compiled StarKist program. Currently, the supplied makefile uses (UBUNTU's native MIPS simulator) spim to run your compiled program and save results in the output directory:

```
spim -file compiledProgram.s > Output.txt
```

Please make sure that *every* StarKist program given to you does what is expected to do (printing primes between 2 and 100, merging sorted lists, checking out of bound array accesses etc.)

## 6 Submission Guidelines

The code for this exercise resides as usual in subdirectory EX6 of the course GitHub. Next, you need to add the relevant derivation rules and AST constructors for classes. Last, you should implement the missing parts of the Starkist semantic analyzer. The semantic analyzer resides in the file semant.c, and this is where most of your changes will occur. Please submit your exercise in your GitHub repository under COMPILATION/EX6, and have a makefile there to build a runnable program called compiler. Make sure that compiler is created in the same level as the makefile: inside EX6. To avoid the pollution of EX6, please remove all *.o files once the target is built. The next paragraph describes the execution of compiler.

**Execution parameters** compiler receives 2 input file names:

InputStarkistProgram.txt
OutputMipsAsm.s