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 example, 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.

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 assembly file.

2 IR Specification

The entire set of IR commands is given in Table 2. We will briefly describe each of the commands now. **BINOP** performs the binary operation between its two sub trees, the left sub tree is evaluated first. **MOVE** stores a source value (sub tree) to a destination which is either a memory address (**MEM** sub tree) or a temporary (**TEMP** sub tree, which is actually a leaf). The destination is evaluated first. **SEQ** indicates a sequence of its two sub trees, and as before, the left sub tree is evaluated first. **CONST** is a leaf containing a single integer as its value. **JUMP** and **LABEL** are (self explanatory) leaves too.

CJUMP evaluates its left and right sub trees in order, and conditionally jumps to one of two possible labels according to the result of the comparison. **FUNC** should ideally abstract away both prologue/epilogue and calling convention, and contain only the signature of the function and its body. However, this property is somewhat violated in our project, and for code readability reasons, the current implementation contains both prologue/epilogue and the (standard) calling convention. Note in addition, that unlike most programming languages, a function in StarKist can *not* return from multiple places, so there is no need for a **RET** IR command.

BINOP	MOVE	MEM
CJUMP	JUMP	CONST
LABEL	TEMP	CALL
FUNC	SEQ	

Table 2: The IR tree nodes.

3 Inspecting the IR Tree

To create a graph visualization of the IR tree, please install graphviz on your machine and run

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

from the relevant directory containing the intermediate files.

4 Running Example

Consider the simple program in Table 3. It contains a single local variable, a

```
let
    type address = {Apt:int, rent:int, ZIP:int}
    var home:address := address{19, 7500, 6701}
in
    PrintInt(home->ZIP)
end
```

Table 3: A simple StarKist program that contains one local variable, one allocation, one field access and one function call.

single allocation, one field access, and one function call. Note that allocations are done by calling the corresponding library functions: AllocateRecord and AllocateArray. The IR tree for this program is shown in Figure 1, and for example, it is immediate to find the variable "home" there:

```
MEM[BINOP(PLUS)(TEMP(fp),-4)]
```

An observant reader might also notice that the initializing fields sent to allocate array are in reverse order. This is done in order to enable an easier translation that uses the standard calling convention. Another point worth mentioning is that allocate array is a variadic function, and so the actual number of parameters sent must be passed too. This is the last CONST(3) in the call to allocate record.

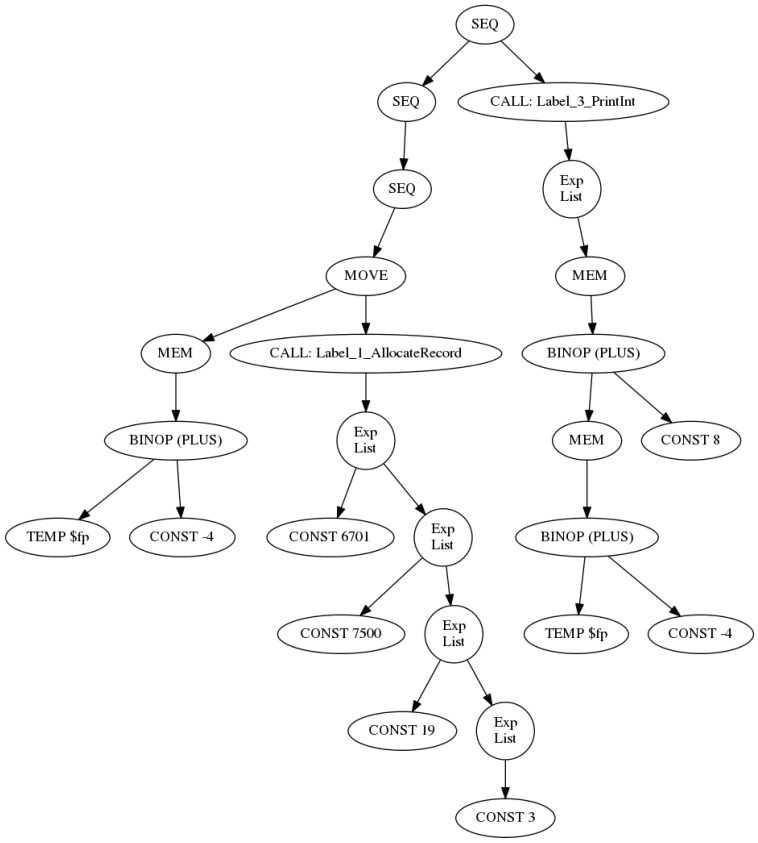

Figure 1: IR tree for the StarKist program in Table 3.

5 Translation from IR to ASM

Translation from IR to ASM involves two parts too. Initially, the IR tree is scanned in a post order fashion, to produce a pseudo ASM text file with an unbounded number of temporaries. Then, the register allocation phase replaces those temporaries with the actual registers of the chosen ASM language. These phases are explained thoroughly in class.

6 Programming Assignment

Your entire assignment is within the creation of the IR tree. The translation from IR to MIPS assembly already exists in the code repository. You will need to implement the runtime mechanism that checks out of bounds array accesses and null dereferences. In addition, you will need to implement all the details to support classes and inheritance (namely, accessing data members of classes, and virtual function calls)

7 Input

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

8 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.)

9 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