Wellesley College ◊ CS301 Compiler Design ◊ October 24, 2000
Handout #08

PROBLEM SET 4

Due on Friday, October 27

This is the final version of the Problem Set 4 description, which supersedes the previous description (posted on October 18). Some details have changed. In particular:

1. *Static validity checking is no longer a part of this assignment. It has been delayed until Problem Set 5. So PS4 consists solely of the code generation problem, which is still worth 100 points.*

2. *The set of files you have to copy to your local directory has be enlarged. See the "What You Need" section. Since the contents of some of these files have changed recently, you should re-download all files mentioned in that section.*

3. *You have now been provided with a testing program that should greatly simplify testing of your code generator. Your hardcopy packet should include a transcript of running* `Codegen.test()`.

OVERVIEW

The purpose of this problem set is to get experience with code generators by writing a MIPS code generator for Kitty. You should carefully study the three code generators presented in class (`CodegenTwoPass.sml`, `CodegenOnePass.sml`, and `CodegenStack.sml`) before attempting this problem set.

This assignment has a single problem worth a total of 100 points. The result will be a single file `Codegen.sml` implementing the Kitty code generator.

COLLABORATION DETAILS

You should break up into three teams of two members each, subject to the following constraints:

- You should not work with the same partner as on PS1, PS2, or PS3

- Each team must consist of one person who has complete CS240 and one person who has not completed CS240.

SUBMISSION DETAILS

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The packet should include (1) your final version of the file `Codegen.sml` and (2) any auxiliary files you created; (3) a transcript of the result of running `Codegen.test()`. Your hardcopy submission packet should also include a header sheet for each team member (see the end of this assignment for the header sheet), and should indicate where the softcopy submission can be found.

Your softcopy submission should consist of your local versions of the `kitty/codegen` and `kitty/test` program directories described below.

THE PROBLEM [100]

Overview

Your problem is to write a code generator for Kitty. Your code generator should have the following behavior:

> Given an AST for a statically valid Kitty program, the code generator should produce a list of MIPS assembly instructions whose execution has the same observable effect as directly executing the Kitty AST.

A Kitty program is **statically valid** if every context that expects a valued expression can be statically (i.e., at compile-time) shown to be supplied with a valued expression, and every context that expects a valueless expression can be statically shown to be supplied with a valueless expression. For the definition of valued and valueless expressions, consult the Kitty Reference Manual (Handout #03).

In this assignment, you can assume that all Kitty programs given as input to the code generator are statically valid. If the code generator is provided with a program that is not statically valid, its behavior is unspecified. You will write a program to detect statically valid Kitty programs in Problem Set 5.

Code generation requires implementing the following function in the `Codegen` structure in the file `kitty/codegen/Codegen.sml`:

```
genProgram : AST.exp -> MIPS.instr list
```

`genProgram(exp)` the returns a list of MIPS instructions that has the same observable behavior as executing `exp` directly in an interpreter.

You can generate code using any patterns and subroutines that you find useful. The only constraint is this: you must store all Kitty variables (both named variables and temporaries) either on the stack or in registers. For storing variables on the stack, you will find it helpful to use the static environment functor in `utils/StaticEnv.sml` for computing lexical addresses.

Use the code generators discussed in class as a guide for thinking about this problem. Many of the details for compiling Kitty are similar to those for compiling SLiP. You will be able to reuse some of the SliP patterns, but will have to develop some new ones as well (e.g. for compiling `let`, `if`, and `while`).

You will need to decide what type you want for `codegen`. There are many possibilities; studying the existing code generators will help you make a decision. You will also need combinators that manipulate entities of type `codegen`. You may be able to reuse existing combinators, but in any case you will probably want to define some new ones on your own.

You should test your code generator on a wide range of test examples, including the “tricky” variable scoping tests from Problem Set 1.

What You Need

Start the problem by making copying the following directories and files to your local file system. You should preserve the relative positions of directories when you make your local copies.

- Copy the modified file `~cs301/download/kitty/AST.sml` to your existing local version of this directory.

4. Copy the files `~cs301/download/kitty/EvalTest.{sml,cm}` to your existing local version of this directory.

- Copy all the files in `~cs301/download/kitty/utils` to your existing local version of this directory.

- Copy all the files in `~cs301/download/kitty/test` to your existing local version of this directory.

- Make a local copy of the directory `~cs301/download/kitty/codegen.`

It is also assumed that:

- Your local version of the `~cs301/download/kitty` directory contains your working evaluator from that problem set.

- Your local version of the `~cs301/download/kitty/scanner` directory contains your working Kitty lexical analyzer (`Kitty.lex.sml`) from Problem Set 2.

- Your local version of the `~cs301/download/kitty/parser` directory contains your working Kitty parser (`Parser.sml`) from Problem Set 3.

- Your `.cm` files reference utility files in the `kitty/utils` directory rather than in the `kitty` directory. (These have moved from kitty to `kitty/utils` since, and references to old utility files in `kitty` will cause CM problems.)

Documentation

Your task is to flesh out the skeleton file `codegen/Codegen.sml` so that it implements a Kitty code generator. This skeleton file contains a few predefined functions that will simplify testing. These are described in the next subsection.

I have extended the SPIM simulator (just the terminal interface program, not the X-windows program) with (1) additional system calls ; (2) file input/output to allow you to compile and test the Kitty I/0 primitives; and (3) tracing functionality allowing you to trace instructions and determine instruction counts and frequencies. The extensions are described in Appendix A.

Compiling, Testing, and Debugging

You can compile your Kitty parser by executing the following in the SML interpreter.

```
CM.make'("Codegen.cm");
```

As usual, you may want to name this command via a small function name to avoid retyping the whole command. E.g.:

```
fun c() = CM.make'("Codegen.cm");
```

You can test your Kitty code generator via any of the following functions within the Codegen structure:

```
val genStringPrint : string -> ()
```
`genStringPrint kittyExp` prints the MIPS code sequence resulting from the code generation of `kittyExp`, a literal string representation of a Kitty expression.

```
val genStringToFile : string -> string -> ()
```

`genStringToFile outfile kittyExp` writes to the file named `outfile` the MIPS code sequence resulting from the code generation of `kittyExp`, a literal string representation of a Kitty expression.

`val` `genFilePrint : string -> ()`
`genFilePrint infile` prints the MIPS code sequence resulting from the code generation of the Kitty expression in the file named `infile`.

`val` `genFileToFile : string -> string -> ()`
`genFileToFile infile outfile` writes to the file named `outfile` the MIPS code sequence resulting from the code generation of the Kitty expression in the file named `infile`.

All of the above functions will use the parser that is in `parser/Parser.sml`, which presumably is your working parser from Problem Set 3. Please contact me if your PS3 parser is not working, as it is necessary to have a working parser in order to test your code generator.

The fact that your code generator produces MIPS code that must itself be executed makes testing and debugging more complex than usual. It is difficult by visual inspection to determine if the MIPS instruction sequence produced by your code generator for a particular Kitty source file is correct. In general, it will be necessary to execute the MIPS code in the SPIM simulator to determine if it has the right behavior. Using the `infile` and `outfile` commands in the extended terminal-based simulator will allow you test Kitty programs that read input from or write input to files. See Appendix A for details.

To simplify testing, you have been provided with a file `CodegenTest.sml` that will automatically compare the results of running your evaluator and running the MIPS code generated by your code generator on a suite of benchmarks. The `CodegenTest` structure in this file provides the following top-level variables and testing functions:

`val testCases : (string * string list) list`
This is the list of benchmarks tested in the testing functions. Each benchmark consists of a pair of (1) the name of a file containing a Kitty program and (2) a list of input files on which to test the Kitty program. If the input file list is empty, it is assumed that the Kitty program does not read any input; for any Kitty program that reads input, the input file list should contain at least one file. It is assumed that all filenames refer to files in the `kitty/test` directory.

You are encouraged to add new benchmarks to the suite.

`val test : unit -> unit`
Invoking this function tests each benchmark in `testCases`. Testing a benchmark means: (1) evaluating the Kitty program in the evaluator for all input files; (2) compiling the Kitty program to MIPS; (3) executing the resulting MIPS program on all input files; and (4) comparing the results of evaluating the results of steps (1) and (3). (For programs not encountering a static or dynamic error, the results should be identical.) The test function reports on the results of each step for all combintations of Kitty program and input file.

`val testVerbose : unit -> unit`
Like `test`, except also displays the results of evaluating the Kitty program and the results of executing the MIPS program compiled from the Kitty program.

For example, here is a snippet of the transcript produced by `test()`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Testing Kitty program words.kty on input file wd40.txt
Evaluating words.kty, writing output to words_wd40.eval.out
Generating MIPS for words.kty, writing output to words.mips
```

```
Executing words.mips in SPIM, writing output to words_wd40.mips.out
spim -infile "wd40.txt" -outfile "words_wd40.mips.out" -file "words.mips"
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes file I/O extensions (version 1.0) and tracing extensions (version 1.0)
by Franklyn Turbak (fturbak@wellesley.edu).
Using wd40.txt as the input file
Using words_wd40.mips.out as the output file
Loaded: /usr/share/spim/trap/trap.handler
diff -a words_wd40.eval.out words_wd40.mips.out

Test succeeded
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

By comparison, here is the corresponding snippet produced by `testVerbose()`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Testing Kitty program words.kty on input file wd40.txt
Evaluating words.kty, writing output to words_wd40.eval.out
-------------------- Contents of file words_wd40.eval.out --------------------
I
bought
cans
of
WD
yesterday
for
Abby
said
Isn
t
that
great
--------------------------------------------------------------------------------
Generating MIPS for words.kty, writing output to words.mips
Executing words.mips in SPIM, writing output to words_wd40.mips.out
spim -infile "wd40.txt" -outfile "words_wd40.mips.out" -file "words.mips"
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes file I/O extensions (version 1.0) and tracing extensions (version 1.0)
by Franklyn Turbak (fturbak@wellesley.edu).
Using wd40.txt as the input file
Using words_wd40.mips.out as the output file
Loaded: /usr/share/spim/trap/trap.handler
-------------------- Contents of file words_wd40.mips.out --------------------
I
bought
cans
of
WD
yesterday
for
Abby
said
Isn
t
that
great
--------------------------------------------------------------------------------
diff -a words_wd40.eval.out words_wd40.mips.out

Test succeeded
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

All filenames mentioned in the transcript by default reside in the `kitty/test` directory. You may wish examine some of them as part of debugging.

Any static or dynamic errors encountered by the testing program are reported. In some cases, they indicate an error in the Kitty program that your evaluator or compiler is required to catch. In other cases, they indicate an error in your evaluator, scanner, parser, or code generator. Below is an example of the former case, where the Kitty program has an unbound variable “x”:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Testing output-only Kitty program unbound.kty
Evaluating unbound.kty, writing output to unbound.eval.out

*** ERROR: EvalError exception:
Unbound variable x
Evaluation did not succeed; attempting compilation.
Generating MIPS for unbound.kty, writing output to unbound.mips

*** ERROR: Fail exception:
StaticEnv: lookup of unbound variable "x"

Neither evaluation no compilation succeeded.
Perhaps it is because they encountered the same error; perhaps not.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Both `test()` and `testVerbose()` do most of their work in the `kitty/test` directory, but they try to leave your SML working directory the same as they encountered it. However, should you use Ctrl-C to abort one of these, you can circumvent the cleanup code and leave the working directory connected to `kitty/test`. In this case, you may need to manually change it back to the original directory (typically, `kitty/codgen`).

Appendix A: SPIM Extensions

I have extended the terminal interface to the SPIM simulator (but not the X-windows interface) with the features described below.

A.1 New Syscalls

SPIM now supports the following additional I/O syscall primitives, which are extremely handy for compiling Kitty.

Mnemonic	Number
`FILEIO_PRINT_CHAR`	11
`FILEIO_PRINT_INT`	12
`FILEIO_PRINT_STRING`	13
`FILEIO_READ_CHAR`	14
`FILEIO_READ_INT`	15
`FILEIO_READ_INT_DEFAULT`	16
`FILEIO_EOF`	17

All of the I/O performed by these primitives takes place on the standard input and output streams, which may have been redirected from/to files via the file redirection commands described in section A.2. File redirection does not work on the existing SPIM I/O syscalls.

The meaning of the above primitives is as follows:

- `FILEIO_PRINT_CHAR`: Print the character in register $a0 to standard out.
- `FILEIO_PRINT_INT`: Print the integer in register $a0 to standard out.
- `FILEIO_PRINT_STRING`: Print the null-terminated string at the address in register $a0 to standard out.
- `FILEIO_READ_CHAR`: Read and consume the next character from standard input into register $v0. The resulting value is the ASCII value of the character, or –1 if the end-of-file has been encountered.
- `FILEIO_READ_INT`: Has the effect of invoking `FILEIO_READ_INT_DEFAULT` with argument 0.
- `FILEIO_READ_INT_DEFAULT`: Register $a0 contains a "default" integer. Consume all whitespace before the next non-whitespace character. If that character can be interpreted as the beginning of an integer, consume the characters of the integer and return the integer in register $v0. Otherwise, consume no non-whitespace characters and return the default integer. Also return the default integer if the end-of-file is encountered while searching for the next integer.
- `FILEIO_EOF`: Sets register $v0 to 1 if the next `FILEIO_READ_CHAR` would fail (return a –1). Sets register $v0 to 0 otherwise.

A.2 File redirection commands

The following commands have been added to the SPIM read-eval-print loop:

`infile` *`filename`*
: Redirects standard input for subsequent SPIM execution to come from the file named *`filename`*, which should be delimited by double quotes. If filename is the empty string, sets standard input to come from the console.

`outfile` *`filename`*

Redirects standard output for subsequent SPIM execution to be written to the file named *`filename`*, which should be delimited by double quotes. If filename is the empty string, sets standard output to go to the console.

Both standard input and output can be redirected from/to files via command-line arguments to `spim` (`-infile` *`filename`* and `–outfile` *`filename`*). In these cases, *`filename`* need not be delimited by double quotes.

A.3 Instruction-level Tracing

The extended terminal interface to SPIM supports a *tracing mode* in which all executed instructions are displayed, along with the contents of any source and destination registers mentioned by the instructions. This is an extremely valuable debugging tool – one that is often more time-effective than using single-stepping (even single-stepping under the X windows interface.) It is controlled by the following commands in the SPIM read-eval-print loop:

`trace`
: Turns tracing mode on. (Tracing mode can also be set via the `–trace` command-line argument to `spim`.)

`notrace`
: Turns tracing mode off.

As an example of tracing, consider the following snippet of the 7th through 15th instructions executed by a MIPS program that copies characters from standard input to standard output:

```
7: [0x00400020] 0x34020011  ori $2, $0, 17                  ; 17: li $v0, 17        # system
call code for fileio_eof
>Reg 0 = 0x00000000 (0)
<Reg 2 = 0x00000011 (17)
8: [0x00400024] 0x0000000c  syscall                         ; 18: syscall
>Reg 2 = 0x00000011 (17)
[System call of fileio_eof]
9: [0x00400028] 0x14400007  bne $2, $0, 28 [done-0x00400028]; 19: bnez $v0, done
>Reg 2 = 0x00000000 (0)
>Reg 0 = 0x00000000 (0)
10: [0x0040002c]        0x3402000e  ori $2, $0, 14                  ; 20: li $v0, 14        #
system call code for file\
io_read_char
>Reg 0 = 0x00000000 (0)
<Reg 2 = 0x0000000e (14)
11: [0x00400030]        0x0000000c  syscall                         ; 21: syscall
>Reg 2 = 0x0000000e (14)
[System call of fileio_read_char]
12: [0x00400034]        0x00022021  addu $4, $0, $2                 ; 22: move $a0, $v0
>Reg 0 = 0x00000000 (0)
>Reg 2 = 0x00000064 (100)
<Reg 4 = 0x00000064 (100)
13: [0x00400038]        0x3402000b  ori $2, $0, 11                  ; 23: li $v0, 11        #
system call code for file\
io_print_char
>Reg 0 = 0x00000000 (0)
<Reg 2 = 0x0000000b (11)
14: [0x0040003c]        0x0000000c  syscall                         ; 24: syscall
>Reg 2 = 0x0000000b (11)
[System call of fileio_print_char]
>d15: [0x00400040]      0x08100008  j 0x00400020 [main]             ; 25: j main
```

Each instruction is the actual MIPS instruction executed. Comments indicate lines in the MIPS source program. Recall that pseudo-instructions may assemble to multiple MIPS instructions.

Each instruction is numbered to indicate its order in all instructions executed for the program. An instruction is followed by information about the contents of registers manipulated by the instruction. Register information preceded with '>' indicates the values of the registers *before* the instruction is executed, while those preceded with '<' indicate the values of registers *after* the instruction is executed.

A.4 Instruction Counts and Frequencies

Sometimes it is helpful to know how many instructions or how many of a particular kind of instruction were executed in a given MIPS program. The extended spim simulator keeps track of this information for every program. The information can be accessed by the following commands in the spim read-eval-print loop:

`count`
: Prints out the number of instructions executed since the simulator was last reinitialized

`freq`
: Prints out a table of instruction frequencies for instructions executed since the simulator was last reinitialized.

The transcript below shows `count` and `freq` used in the context of a program that copies standard input to standard output:

```
(spim) load "cat.mips"
(spim) infile "test.txt"
Setting input file to test.txt.
(spim) run
This is a test.
It is only a test.
(spim) count
Number of instructions executed in previous program: 327
(spim) freq
Instruction frequencies for previous program (only non-zero entries
listed):
addiu:  2
addu:   36
bne:    36
jal:    1
jr:     1
j:      35
lw:     1
ori:    107
sll:    1
syscall:        107
        syscall 10 (exit):      1
        syscall 11 (fileio_print_char): 35
        syscall 14 (fileio_read_char):  35
        syscall 17 (fileio_eof):        36
```

Note that the frequency information for syscalls is further broken down by the primitive called.

The above information can also be obtained by using the `-count` and `-freq` command-line arguments to `spim`.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 4
Due Friday, October 27, 2000

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (*any teams collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the total time each team member spent on the parts of this problem set. (Note that spending less time than your partner does not necessarily imply that you contributed less.) Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	**Time For** _____ **(Team Member #1)**	**Time For** _____ **(Team Member #2)**	**Score**
General Reading			
Code Generation [100]			
Total			