Wellesley College ◊ CS301 Compiler Design ◊ December 6, 2000
Handout #12

PROBLEM SET 7
The Final Revision!
Due on Friday, December 8

This is the final version of PS7. It supersedes all previous versions. There are a few details not yet worked out -- namely clever register allocators and contest details -- that will be posted to the CS301 conference later.

READING

- Appel, Chapters 6 through 11. A few notes:
- In Chapter 6, you may ignore the details of the Frame abstraction.
- In Chapter 7, Appel's Tree language is a subset of the IRL language we have studied in class.
- The stages used in Chapter 8 are those used in the Bobcat compiler, modulo the handling of constructs in IRL that are not in Appel's Tree language (e.g., `RETURN`, `TAILCALL`, `HALT`).
- In Chapter 9, Appel's instruction selection algorithm translates from Tree into `Assem`, an abstract assembly language. In contrast, the Bobcat instruction selector translates from IRL into a subset of IRL.

OVERVIEW

The purpose of this problem set is to help you gain familiarity with the theory and practice of compiler back ends, particularly intermediate representation languages and transformations on these languages (e.g. optimization, instruction selection, register allocation, etc). You will get hands-on experience with these in the context of a back end for Bobcat.

This assignment has three problems worth a total of 250 points. There is also a contest aspect to the problem set: teams can win tasty food prizes based on the efficiency of code produced by their back ends.

COLLABORATION DETAILS

As usual, you should work in three teams of two members each. You should work with the same partner with whom you worked on Problem Set 6.

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 `BobcatToIRL.sml` from Problem 1; (2) your final version of the file `CanonToNormal.sml` from Problem 2; and (3) transcripts of your test cases for Problems 1 and 2. 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 version of the `bobcat/codegen` program directory described in the Getting Started section.

THE STRUCTURE OF THE BOBCAT BACK END

The Bobcat **front end** process a Bobcat source program and produces a Bobcat abstract syntax tree (AST). For the remainder of this discussion, we shall assume that the AST is well-typed. There is a Bobcat evaluator that can be used to evaluate such ASTs; see Appendix A.

The Bobcat **back end** transforms a well-typed Bobcat AST into executable MIPS code with the same behavior. The Bobcat back end has the following stages:

- The **Input Stage** translates Bobcat ASTs into programs in the tree-structured IRL intermediate representation language. The IRL language is summarized in Appendix B. It uses psuedo-register and label abstractions that a supplied by the `Temp` structure, described in Appendix C. Support code for IRL includes a pretty printer (Appendix D) and an evaluator (Appendix E). This stage corresponds to Chapter 7 of Appel's book. You will implement this stage via the `BobcatToIRL` translation in Problem 1.

- The **Canonicalization Stage** translates IRL programs into a restricted **canonical** form, which is defined in Appendix F. This stage, which is described in Chapter 8 of Appel, has been implemented for you; details on the functions for this stage can also be found in Appendix F.

- The **Instruction Selection Stage** translates canonical IRL programs into programs **in MIPS normal form**, a restricted form of IRL that corresponds closely to MIPS machine code. MIPS normal form is defined in Appendix G. Instruction selection is covered in Chapter 9 of Appel. You will implement this stage via the `CanonToNormal` translator in Problem 2. (The `CanonToNormal` stage only implements part of instruction selection; the other part is implemented by the `NormalToMIPS` stage in Problem 3.)

- The **Register Allocation Stage** translates normal-form IRL programs using arbitrarily many pseudo-registers into normal-from IRL programs that use the small number of registers available on a MIPS processor. This stage is described in Chapter 11 of Appel; it makes use of a liveness analysis described in Chapter 10 of Appel. Various algorithms for this stage will be implemented for you.

- The **Output Stage** translates the result of register allocation into executable MIPS code, accomplishing the second part of instruction selection while performing code generation. This stage also makes explicit the details of the function calling convention. You will implement this stage via the `NormalToMIPS` translator described in Problem 3.

GETTING STARTED

Begin this assignment by connecting to your local cs301 directory and executing the following in a shell:

```
cvs update -d
```

This will install the `cs301/bobcat/codegen` directory and make updates to several bobcat and `irl` files. All the files you will need to edit can be found in the `codegen` directory.

PROBLEM 1: BobcatToIRL [80]

In this problem, you will translate Bobcat ASTs into IRL code by implementing the following function in the file `BobcatToIRL.sml`:

```
val translate : AST.prog -> IRL.pgm
```

Returns an IRL program that has the same behavior as the given Bobcat AST.

Once you define the `translate` function, the following functions will also be defined:

```
val translatePgm : AST.prog -> IRL.pgm
```

Returns the result of translating the given Bobcat AST into IRL after it has been extended with the Bobcat standard library.

```
val translateString : string -> IRL.pgm
```

Returns the result of `translatePgm` on the Bobcat program written as the given string.

```
val translateFile : string -> IRL.pgm
```

Returns the result of `translatePgm` on the Bobcat program that is the contents of the given file.

You can use the IRL pretty-printer (Appendix D) and IRL evaluator (Appendix E) to test the results of your translations on individual test files (such as those in the `bobcat/test` directory). Since the tree structure of the code produced by translate is somewhat difficult to read, it is recommended tat you first linearize or canonicalize the IRL code before pretty-printing it; see Appendix F for details. When using the IRL evaluator, setting the `debug` flag to true displays *lots* of information, some of which may help you pinpoint bugs.

A more thorough test of the `BobcatToIRL` translator can be accomplished via the following functions in the `BobcatToIRLTest` structure:

```
val test : unit -> unit
```

Runs the BobcatToIRL translator on test cases in a benchmark suite, and compares the results of running the Bobcat evaluator on the pre-translated program to the results of running the IRL evaluator on the post-translated program. Whenever the results differ, the differences are printed via the Unix `diff` command.

```
val testVerbose : unit -> unit
```

Like `test`, but also displays the output of the two evaluators on each test case.

You should approach this problem in stages. First, implement the most straightforward translation from Bobcat ASTs to IRL that you can think of, and make sure it works. Two details to watch out for:

- When compiling a function body, make sure that every control path ends in a `RETURN` or a `TAILCALL`. For this purpose, the program body should be treated as the body of a nullary function. (Note that the IRL evaluator will complain if it encounters an empty sequence of statements -- something that will happen if a `RETURN` or `TAILCALL` is missing).

- The translation must represent global variables by global memory locations; global variables cannot correctly be represented as temporaries (i.e., locations in a stack frame).

The straightforward translation is rather inefficient and can be improved in many ways. Chapter 7 of Appel describes some improvements that you may wish to implement in a second pass. An important improvement not considered by Appel is ensuring that all tail-recursive function invocations in the Bobcat source program are translated to the `TAILCALL` statement in IRL.

PROBLEM 2: CanonToNormal [90]

In this problem, you will implement instruction selection for Bobcat by translating canonical IRL programs into IRL programs in MIPS normal form. (See Appendix G for a definition of MIPS normal form.) Modulo the issues of (1) pseudo-registers and (2) implicit function calling conventions, IRL programs in MIPS normal form are very close to MIPS machine code.

Your task in this problem is to implement the following function in the file `CanonToNormal.sml`:

```
val normalize : IRL.pgm -> IRL.pgm
```
Given an IRL program in canonical form, returns an IRL program in MIPS normal form that has the same behavior as the given program.

Once you define the `normalize` function, the following functions will also be defined:

```
val normalizePgm : AST.prog -> IRL.pgm
```
Returns the result of translating, canonicalizing, and normalizing the given Bobcat AST after it has been extended with the Bobcat standard library.

```
val normalizeString : string -> IRL.pgm
```
Returns the result of `normalizePgm` on the Bobcat program written as the given string.

```
val normalizeFile : string -> IRL.pgm
```
Returns the result of `normalizePgm` on the Bobcat program in the given file.

As in Problem 1, you can use the IRL pretty-printer and evaluator to test your `normalize` function on test cases. The following function in the `Normal` structure is handy for verifying that the result of your select function is indeed in normal form:

```
val normalCheck' : IRL.pgm -> unit
```
Returns unit if the given program is in MIPS normal form. Otherwise, raises an `Abnormal` exception describing why the program is not in normal form.

```
val normalCheck : IRL.pgm -> unit
```
Like normalCheck', but prints out an error in exceptional cases.

A more thorough test of the `CanonToNormal` normalization can be accomplished via the following functions in the `CanonToNormalTest` structure:

```
val test : unit -> unit
```
Runs the `BobcatToIRL`, `Canon`, and `CanonToNormal` stages on test cases in a benchmark suite. For each test case in the suite, it first verifies that the result of these stages is in normal form, and then compares the results of running the Bobcat evaluator on the pre-translated program to the results of running the IRL evaluator on the post-translated program. Whenever the results differ, the differences are printed via Unix `diff`.

```
val testVerbose : unit -> unit
```
Like `test`, but also displays the output of the two evaluators on each test case.

As in Problem 1, you should approach this problem in stages. First, implement the most straightforward instruction selection algorithm that you can think of and test that it works. Your translation should not introduce any "real" registers, except perhaps for `$zero`. (Real registers like $a0, $v0, $sp, etc. will be introduced in the NormalToMIPS stage of Problem 3.) Second, consider using the **maximal munch** algorithm described in Chapter 9 of Appel to improve your instruction selection. Finally, consider optimizations that will improve the results of the instruction selection process (e.g., replace multiplying by a small power of 2 by a left-shift operation).

PROBLEM 3: NormalToMIPS [80]

In this problem, you will translate register-reduced MIPS normal form IRL programs into executable MIPS programs. (See Appendix H for the defintiion of register-reduced MIPS normal form.) Your task is to flesh out the following function in the file `NormalToMIPS.sml`.

`val gen : IRL.pgm -> MIPS.instr list`
Given an IRL program in register-reduced MIPS normal form, returns a sequence of MIPS instructions that has the same behavior as the given program.

Once you define the `gen` function, the following functions will also be defined:

`val genPgm : AST.prog -> IRL.pgm`
Returns the result of translating, canonicalizing, normalizing, register allocating, and generating MIPS code for the given Bobcat AST after it has been extended with the Bobcat standard library.

`val genString : string -> IRL.pgm`
Returns the result of `genPgm` on the Bobcat program written as the given string.

`val genStringPrint : string -> unit`
Prints the result of `genPgm` on the Bobcat program written as the given string.

`val genStringToFile : string -> string -> unit`
`(genStringToFile pgmString outFile)` writes to `outFile` the result of `genPgm` on the Bobcat program written as `pgmString`.

`val genFile : string -> IRL.pgm`
Returns the result of `genPgm` on the Bobcat program that is the contents of the given file.

`val genFilePrint : string -> unit`
Prints the result of `genPgm` on the Bobcat program that is the contents of the given file.

`val genFileToFile : string -> string -> unit`
`(genFileToFile inFile outFile)` writes to `outFile` the result of `genPgm` on the Bobcat program that is the contents of `inFile`.

As in Problem Set 4, you can use the tools in the `MIPS` structure and `spim` program to test the results of your `gen` function. Recall that you can use `trace`/`notrace` to turn detailed instruction tracing on/off in the text-based interface. (Sorry, there is no equivalent facility in the windows-based interface). After executing a program, use the `freq` command to print out the frequencies of MIPS instructions used in the previous execution.

A more thorough test of `NormalToMIPS` code generation can be accomplished via the following functions in the `NormalToMIPSTest` structure:

`val test : unit -> unit`
Runs the `BobcatToIRL`, `Canon`, `CanonToNormal`, `RegAlloc`, and `NormalToMIPS` stages on test cases in a benchmark suite. For each test case in the suite compares the results of running the Bobcat evaluator on the pre-translated program to the results of running the `spim` on the post-translated program. Whenever the results differ, the differences are printed via the Unix `diff` command.

`val testVerbose : unit -> unit`
Like `test`, but also displays the output of the two evaluators on each test case.

As part of translating register-reduced MIPS normal form IRL statements to MIPS instructions, the NormalToMIPS translator also makes explicit the function calling convention that is implicit in function bodies and in the following IRL statements:

- `MOVE(TEMP(rd), CALL(`*`fcn`*`,`*`args`*`))`
- `TAILCALL(n,`*`fcn`*`,`*`args`*`)`
- `RETURN(n,TEMP(rs))`

where *`fcn`* has the form `NAME(`*`label`*`)` or `TEMP(`*`reg`*`)`, with *`reg`* a real machine register, and *`args`* is a list of elements of the form `TEMP(`*`reg`*`)`. In register-reduced form, the *n* formal parameters of an IRL function are named with *argument temporaries* whose indices range from 1 to n. (See Appendix C for the interface to argument temporaries.)

You should assume that the first four arguments of a function are passed in machine registers `$a0`, `$a1`, `$a2`, and `$a3`, and that any additional arguments are passed on the stack. Try not to hardwire these assumptions into your code --- you should write your code in such a way that it is easy to change which and how many registers are used for passing arguments.

As part of implementing the function calling convention, you will need to distinguish `CALL` from `TAILCALL`. Recall that:

- `CALL` invokes the called function via `jal` or `jalr`, and pushes a new stack frame on top of the current one. This stack frame (including any stack-based arguments to the called function) are popped when RETURN is executed in the body of the called function.

- `TAILCALL` pops the current stack frame before pushing a new one, and invokes the called function via `j` or `jr`.

As part of compiling a function body, you will somewhere need to allocate space for local variables, which are denoted by *spill temporaries* in register-reduced IRL. There are many ways to allocate local variables, reference them, and later deallocate them. Think carefully about your strategy before you implement it. Reviewing the function calling convention lectures from class and skimming Chapter 6 of Appel may be helpful here.

The register allocator guarantees that the *n* spill temporaries used in a single function body have indices from 1 to *n*; this fact is very useful for allocating space for local variables. You do *not* have to worry about saving caller- and callee- saves machine registers (conventionally, the `$t`*`n`* and `$s`*`n`* registers, respectively); the register allocator will do this for you (see Appendix H). However, *you* are responsible for saving the `$ra` register appropriately.

Here are some other details that are important:

- You may generate MIPS code in any way you want, but it is strongly recommended that you use the code generation monad that was discussed in lecture and is provided in `NormalToMIPS.sml`.

- As in the SLIP code generator studied in class, you will also need to allocate global variables in the data section of the MIPS code for Bobcat variables declared at top-level.

- It is possible to reference all information in the stack activation frame via the stack pointer (`$sp`) register. However, you may wish to use the frame pointer (`$fp`) register in conjunction with the stack pointer register, as discussed in Chapter 6 of Appel. It's up to you.

- For determining properties of programs like "what's the maximal index of a spill temporary in this function body" or "what global variables are used in this program", you probably want to use the way cool expression/statement enumerator described in Appendix I.

- For determining properties such as those mentioned above, you may also wish to manipulate sets whose elements have type `temp`, or tables whose keys have type `temp`. These are provided by the structures `TempSet` and `TempTable` exported by the `Temp` structure. The interface for sets is described at

  http://cm.bell-labs.com/cm/cs/what/smlnj/doc/smlnj-lib/Manual/ord-set.html .

  The interface for tables can be found in the cs301 CVS repository in `~cs301/utils/Table.sml`.

  For nested structures, you use nested qualification to access an element. For instance, the empty set can be written `Temp.TempSet.empty`. You can use local structure abbreviations to simplify the qualified names. For example:

  ```
  structure TS = Temp.TempSet

  ... TS.empty ...
  ```

  There are analagous structures `Temp.LabelSet` and `Temp.LabelTable` for manipulating sets with label elements and tables with label keys.

  Note: In previous versions of the code, `TempSet` was defined in `NormalToMIPS.sml`. It has been moved to `Temp` to make it more accessible for all phases of compilation.

- The label named `main` is problematic in the `NormalToMIPS` translation. MIPS expects the entry point to the program to be the label `main`. However, syntactic sugar for Bobcat programs can introduce a function named `main` that is distinct from the top-level entry point to the program. To finesse this problem, it is recommended that you prefix all labels arising in code generation with a special header (e.g. "`user.`") so that the label for the Bobcat function named `main` does not clash with the `main` entry label for the MIPS program.

- To compile the `HALT` statement, you will need to use the exit system call (code = 10).

- The Bobcat standard library functions will automatically be included with your Bobcat program before it is compiled. However, in the NormalToMIPS code generator, you need to handle the five primitive functions specially: `chr`, `ord`, `eof`, `readc`, and `writec`. These can be handled as follows:

  - To be consistent with the Bobcat evaluator, `chr` should mod by 256 the integer value representing the character. (Implementing this without using a new temporary register requires a little bit of cleverness.)

  - `ord` is just the identity operator at the machine level.

  - `eof` can by implemented by a system call with code 17.

  - `readc` can be implemented by a system call with code 14.

  - `writec` can be implemented by a system call with code 11.

BACK END CONTEST

Each team's back end will be entered into a contest. (Don't tell your parents this!) The back end will be exercised on a suite of benchmark programs, and the number of MIPS instructions executed for each program will be measured. (Recall that the spim `freq` command will show you the MIPS instruction frequencies.) A weighted sum of instructions over all benchmark programs will be calculated.

Teams will be ranked first by the correctness of their compiler (how many programs are compiled correctly) and second by the weighted sum of instructions. A smaller weighted sum indicates a more efficient compiler.

Teams will be eligible for valuable Trader Joe's treats that depend on their final ranking:

- *First Place*: Each team member gets one large treat or two small treats.
- *Second Place*: Each team member gets one small treat.
- *Third Place*: Team members share one small treat.

Details about the benchmarks and weighting functions will be posted soon.

APPENDIX A: The Bobcat Evaluator

The CVS-controlled directory `~cs301/bobcat/eval` contains a file `Eval.sml` that defines an `Eval` structure with the following signature:

`exception EvalError of string`
: This exception is raised when an error in Bobcat evaluation is encountered.

`val eval : AST.prog -> unit`
: `(eval pgm)` evaluates the Bobcat AST `pgm` using the console as standard input and output.

`val evalString : string -> unit`
: `(evalString bobcatString)` type checks and evaluates the Bobcat program in `bobcatString`, extended with the standard libraries, using the console as standard input and output.

`val evalStringIn : string -> string -> unit`
: `(evalStringIn bobcatString infile)` is like `(evalString bobcatString)`, but it uses `infile` as standard input.

`val evalStringOut : string -> string -> unit`
: `(evalStringIn bobcatString outfile)` is like `(evalString bobcatString)`, but it uses outfile as standard output.

`val evalStringInOut : string -> string -> string -> unit`
: `(evalStringInOut bobcatString infile outfile)` is like `(evalString bobcatString)`, but it uses infile as standard input and outfile as standard output.

`val evalFile : string -> unit`
: `(evalFile bobcatFile)` type checks and evaluates the Bobcat program in the file named `bobcatFile`, extended with the standard libraries, using the console as standard input and output.

`val evalFileIn : string -> string -> unit`
: `(evalFileIn bobcatFile infile)` is like `(evalFile bobcatFile)` but uses `infile` as standard input.

`val evalFileOut : string -> string -> unit`
: `(evalFileOut bobcatFile outfile)` is like `(evalFile bobcatFile)` but uses `outfile` as standard output.

`val evalFileInOut : string -> string -> string -> unit`
: `(evalFileInOut bobcatFile outfile)` is like `(evalFile bobcatFile)` but uses `infile` as standard input and `outfile` as standard output.

`val withStandardHandler : 'a -> (unit -> 'a) -> 'a`
: `(withStandardHandler default thunk)` evaluates `thunk` in the context of a standard exception handler that prints the messages of `EvalError` exceptions. It returns the value of the `thunk` if no exception is raised, and default if exception is raised.

APPENDIX B: The IRL Intermediate Representation Language

The CVS-controlled file `~cs301/irl/IRL.sml` defines the `IRL` structure given below. IRL extends Appel's Tree intermediate language from Chapter 7 with (1) a program construct with mutually recursive function declarations; (2) `RETURN`, `TAILCALL`, and `HALT` statements; and (3) larger integers (Int32.int). IRL is good as a target for a flat, first-order language like Bobcat, but cannot easily model Tiger's block structure or ML's higher-order functions.

```
structure IRL = struct

  (* A program is a collection of global mutually recursive function and
     a list of statements to be executed relative to these *)
  datatype pgm = PROG of fundecl list * stm list

  (* A member of the collection of global mutually recursive function decls *)
  and fundecl = FUNDECL of label * Temp.temp list * stm list

  and stm = SEQ of stm * stm
          | LABEL of Temp.label
          | JUMP of exp * Temp.label list
          | CJUMP of relop * exp * exp * Temp.label * Temp.label
          | MOVE of exp * exp
          | EXP of exp
          (* The following are new in IRL *)
          | RETURN of int * exp
          | TAILCALL of int * exp * exp list
          | HALT of string

  and exp = BINOP of binop * exp * exp
          | MEM of exp
          | TEMP of Temp.temp
          | ESEQ of stm * exp
          | NAME of label
          | CONST of Int32.int (* Larger than Appel's int *)
          | CALL of exp * exp list

  and binop = PLUS | MINUS | MUL | DIV | MOD (* MOD is new in IRL *)
            | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR

  and relop = LT | LE  |EQ | NE | GT | GE | ULT | ULE | UGT | UGE

  (* Invert the sense of a relop *)
  fun notRel LT = GE | notRel LE = GT | notRel EQ = NE | notRel NE = EQ
    | notRel GE = LT | notRel GT = LE | notRel ULT = UGE | notRel ULE = UGT
    | notRel UGE = ULT | notRel UGT = ULE

  val NOOP = EXP(CONST(Int.toLarge(0))) (* A "no op" is a "do nothing" instr *)

  (* Convert an IRT.stm list to an IRT stm *)
  fun seqStm([]) = NOOP
    | seqStm([stm]) = stm
    | seqStm(stm::stms) = SEQ(stm,seqStm(stms))

  (* Convert an IRT.exp list to an IRT stm *)
  fun seqExp(es) = seqStm (List.map EXP es)

  fun const(n) = CONST(Int32.toLarge(n))
  fun fundeclLabel(FUNDECL(label,_,_)) = label
  fun fundeclParams(FUNDECL(_,params,_)) = params
  fun fundeclBody(FUNDECL(_,_,body)) = body
end
```

APPENDIX C: The Temp Structure

IRL labels and temporaries are supplied by a `Temp` structure in the CVS-controlled file `cs301/irl/Temp.sml`. The signature for these are given below. (The signature is a modification of that in Chapter 7 of Appel).

The labels are straightforward. For creating labels local to a function body, you should use `newLabel` or `namedNewLabel`; the only difference is that the latter incorporates a name which can help debugging. To generate labels whose global name matters, use `namedLabel`.

```
eqtype label
```
The abstract type of an IRL label.

```
val newLabel : unit -> label
```
Returns a fresh label.

```
val namedNewLabel : string -> label
```
Returns a fresh label whose name extends the given string with a unique integer.

```
val namedLabel : string -> label
```
Returns the label with the given label. While calls to `newTemp` or `namedNewLabel` always return fresh labels, two calls to `namedTemp` with the same name return the same label.

```
val labelToString : label -> string
```
Returns the string name of a label.

```
val labelCompare : label * label -> order
```
Returns the order (`LESS`,`EQUAL`,`GREATER`) of two labels.

```
structure LabelSet : ORD_SET
```
Structure for sets whose elements have type `label`.

```
structure LabelTable : TABLE
```
Structure for tables whose keys have type `label`.

There are four classes of IRL temporaries:

· **Register temporaries** (created via reg) represent real machine registers. Except for the distinguished $zero register, these temporaries are not introduced until the register allocation phase.

· **Argument registers** (created via arg) represent abstract argument registers. These are introduced by the register allocation phase and should be translated into real argument registers or stack offsets as part of the NormalToMIPS translation.

· **Spill registers** (created via newSpill) represent stack locations. These are introduced by the register allocation phase and should be translated into stack offsets as part of the NormalToMIPS translation.

· **Pseudo-registers** (created via newTemp or namedNewTemp) represent one of arbitrarily many abstract locations. These are completely removed by the register allocation stage.

`eqtype temp`
  The abstract type of an IRL pseudo-register.

`val newTemp : unit -> temp`
  Returns a fresh pseudo-register.

`val namedNewTemp : string -> temp`
  Returns a fresh pseudo-register whose name extends the given string with a unique integer.

`val arg : int -> temp`
  Returns a temp denoting an abstract argument register indexed by the given number, which should be 1 or greater.

`val isArg : temp -> bool`
  Returns `true` if temp is an argument temp, and `false` otherwise.

`val argIndex : temp -> int`
  If `temp` is an argument temp, returns its index (>= 1). Otherwise, raises a `Fail` exception.

`val reg : string -> temp`
  Returns a temp denoting the MIPS register with the given name, which must begin with a `'$'`. Different invocations of the `reg` function with the same string return the same temp.

`val isReg : temp -> bool`
  Returns `true` if temp is a register temp, and `false` otherwise.

`val newSpill : unit -> temp`
  Returns a new spill temp, whose index is the current value of the spill counter. The counter is incremented as a side effect of invoking this function.

`val isSpill : temp -> bool`
  Returns `true` if temp is a spill temp, and `false` otherwise.

`val spillIndex : temp -> int`
  If temp is a spill temp, returns its index (>= 1). Otherwise, raises a `Fail` exception.

`val resetSpillCounter : unit -> unit`
  Resets the spill counter to have value 1.

`val tempToString: temp -> string`
  Returns the string name of a pseudo-register.

`val tempCompare : temp * temp -> order`
  Returns the order (`LESS`,`EQUAL`,`GREATER`) of two pseudo-registers.

`structure TempSet : ORD_SET`
  Structure for sets whose elements have type `temp`.

`structure TempTable : TABLE`
  Structure for tables whose keys have type `temp`.

APPENDIX D: The IRL Printer

IRL programs, statements, and expressions can be pretty-printed by the following functions in the `IRLPrinter` structure, which can be found in the CVS-controlled file `cs301/irl/IRLPrinter.sml`.

`val printPgm : IRL.pgm -> unit`
Pretty-prints the given IRL program.

`val printStm : IRL.stm -> unit`
Pretty-prints the given IRL statement.

`val printStms : IRL.stm list -> unit`
Pretty-prints the given IRL statment list.

`val printExp : IRL.exp -> unit`
Pretty-prints the given IRL expression.

`val binopToString : IRL.binop -> string`
Returns a string representation of the binop.

`val relopToString : IRL.relop -> string`
Returns a string representation of the relop.

APPENDIX E: The IRL Evaluator

IRL programs can be executed via the IRL evaluator found in the `IRLEval` structure within the CVS-controlled file `cs301/irl/IRLEval`.

`val debug : bool ref`
When `debug` is true, the current statement and machine state are displayed for every statement executed. When `debug` is false, no such information is displayed.

`val eval : IRL.pgm -> unit`
`(eval pgm)` evaluates the given IRL program AST `pgm` using the console as standard input and output

`val evalIn : IRL.pgm -> string -> unit`
`(evalIn pgm infile)` is like `(eval pgm)` except that it uses `infile` as standard input.

`val evalOut : IRL.pgm -> string -> unit`
`(evalOut pgm outfile)` is like `(eval pgm)` except that it uses `outfile` as standard output.

`val evalInOut : IRL.pgm -> string -> string -> unit`
`(evalInOut pgm infile outfile)` is like `(eval pgm)` except that it uses `infile` as standard input and `outfile` as standard output.

`val withStandardHandler : 'a -> (unit -> 'a) -> 'a`
`(withStandardHandler default thunk)` evaluates `thunk` in the context of a standard exception handler. It returns the value of the `thunk` if no exception is raised, and default if an exception is raised.

`val binopToML : IRL.binop -> (Int32.int * Int32.int -> Int32.int)`
Returns an ML function corresponding to the IRL binop

`val relopToML : IRL.relop -> (Int32.int * Int32.int -> bool)`
Returns an ML function corresponding to the IRL relop.

APPENDIX F: The IRL Canonicalizer

The CVS-controled file `cs301/irl/Canon.sml` defines a `Canon` structure that supplies the canonicalization functions below. These functions are essentially those discussed in Chapter 8 of Appel, extended to handle the additional features of IRL. We introduce the following terminology:

- An IRL statement list is in **linear form** if the following two conditions are satisfied:
  - No `SEQ` statements or `ESEQ` expressions appear in the statement list.
  - The parent of every `CALL` expression is an `EXP` statement or a `MOVE(TEMP(…),…)` statement.

  Note that each occurrence of the pattern `RETURN(`*`n`*`,(CALL(`*`e`*`,`*`es`*`))` can be optimized to `TAILCALL(`*`n`*`,`*`e`*`,`*`es`*`)`, removing the need to consider `CALL`s that have `RETURN` as their parent.
- An IRL program is **linear** if the program body and all function bodies are statement lists in linear form.
- A list of IRL statements is a **basic block** if it satisfies the following conditions:

1. It is in linear form.
2. It begins with a `LABEL` statement and contains no other occurrences of a `LABEL` statement.
3. It ends with a `JUMP`, `CJUMP`, `RETURN`, `TAILCALL`, or `HALT` statement, and contains no other occurrences of these statements.

- The parent of every `CALL` expression is an `EXP` statement or a `MOVE(TEMP(…),…)` An IRL program is in **canonical form** if the following two conditions are satisfied:

· The program is in linear form.

· Every `CJUMP` statement is immediately followed by its false label.

- An IRL program is **canonical** if the program body and all function bodies are statement lists in canonical form.

Here are the functions supplied by the Canon structure:

`val linearizeStm : IRL.stm -> IRL.stm list`
  Returns a linear form statement list with the same meaning as the given one.

`val linearizeStms : IRL.stm list -> IRL.stm list`
  Returns a list of statements in linear form with the same meaning as the given list.

`val linearizePgm : IRL.pgm -> IRL.pgm`
  Returns a linear program with the same meaning as the given program.

`val basicBlocks : IRL.stm list -> IRL.stm list list`
  Assume the input statement list is in linear form. Returns a partitioning of the given statement list into basic blocks. The partitioning may add new `LABEL` and `JUMP` statements.

`val traceSchedule : IRL.stm list list -> IRL.stm list`
  Reorder and concatenate a list of basic blocks to yield a statement list in canonical form.

`val canonStm : IRL.stm -> IRL.stm list`
  Returns a canonical form statement list with the same meaning as the given statement.

`val canonStms : IRL.stm list -> IRL.stm list`
  Returns a canonical form statement list with the same meaning as the given statement list.

`val canonPgm : IRL.pgm -> IRL.pgm`
  Returns a canonical program with the same meaning as the give program

APPENDIX G: MIPS Normal Form

We will say that an IRL program is in **MIPS normal form** (or just **normal form**) if it satisfies the following three conditions:

1. It is in canonical form.
2. It does not contain any `EXP` statements.
3. Every IRL statement (or in some cases, sequence of statements) can be straightforwardly translated to a sequence of MIPS assembly instructions without requiring any extra temporary registers. This excludes register `$at` reserved for the assembly process, as well as dedicated machine registers (e.g., `$a0`, `$v0`, `$sp`) needed for certain IRL instructions (`CALL`, `TAILCALL`, `RETURN`, `HALT`).

In particular, MIPS normal form is defined by the left-hand column of the table in Figure 1, which shows a correspondence between IRL statements and MIPS instructions. The table uses the following notation.

- *`bop`* ranges over elemens of `IRL.binop`.
- *`bopi`* ranges over `ADD`, `AND`, `OR`, `XOR`.
- *`sop`* ranges over `LSHIFT`, `ARSHIFT`, `RSHIFT`.
- *`rop`* ranges over elements of `IRL.relop`.
- *`rop0`* ranges over `GE`, `GT`, `EQ`, `NE`, `LE`, `LT`.
- *`lab`*, *`labt`*, *`labf`* range over labels.
- *`labs`* ranges over lists, each of whose elements is a *`lab`*.
- *`r,rd`*, *`rs`*, *`rt`* range over pseudo-registers.
- *`temp`* ranges over expressions of the form `TEMP(`*`r`*`)`.
- *`temps`* ranges over lists, each of whose elements is a *`temp`*.
- *`addr`* ranges over expressions of the form `CONST(`*`n`*`)`, `NAME(`*`lab`*`)`, or `TEMP(`*`r`*`)`.
- *`loc`* ranges over expressions of the form `CONST(`*`n`*`)`, `NAME(`*`lab`*`)`, `TEMP(`*`r`*`)`, or `BINOP(PLUS, TEMP(p), CONST(c)`).

Figure 1 effectively defines the instruction selection for IRL to MIPS translation. (See Chapter 9 of Appel for a discussion of instruction selection.) In the Bobcat compiler, instruction selection is performed by a combination of the `CanonToNormal` and `NormalToMIPS` stages. The `CanonToNormal` stage puts IRL code into a normal form that exposes temporary registers so that register allocation can be performed, while the `NormalToMIPS` stage performs the actual translation to MIPS instructions.

Some notes about Figure 1:

- It is assumed that `CONST(0)` and `TEMP(Temp.reg("$zero"))` should be interchangeable. That is, it should always possible to use one in place of the other.
- The five statement IRL sequences corresponding to `seq`, `sne`, …, `slti` are not necessary for defining MIPS normal form, but they do play a part in instruction selection.
- MIPS normal form does *not* make the details of the function calling convention explicit. It is assumed that these details will be determined by the `NormalToMIPS` stage. However, this implies that the register allocator must know which registers are used by the `CALL`, `TAILCALL`, `RETURN`, and `HALT` instructions. See Appendix H for details.

IRL instruction(s)	MIPS instruction(s)
LABEL(*lab*)	*lab*:
HALT(*string*)	print_string and exit syscalls
RETURN(*n*, *temp*)	function return protocol
TAILCALL(*n*, *addr*, *temps*)	tail call protocol
MOVE(TEMP(rd), CALL(*addr*,*temps*))	function call protocol (includes jal or jalr)
MOVE(TEMP(*rd*), *addr*)	li *rd*, *a* if *addr* = CONST(*a*) la *rd*, *lab* if *addr* = NAME(*lab*) move *rd*, rs if *addr* = TEMP(rs)
MOVE(TEMP(*rd*), BINOP(*bop*, TEMP(*rs*), TEMP(*rt*)))	*bop' rd*, *rs*, *rt* where *bop/bop'* is one of ADD/add, AND/and, DIV/div, MOD/rem, MINUS/sub, MUL/mul, OR/or, XOR/xor. LSHIFT/sllv, ARSHIFT/srav, RSHIFT,srlv
MOVE(TEMP(*rt*), BINOP(*bopi*, TEMP(*rs*),CONST(*n*)))	*bopi' rt*, *rs*, *n* where *bopi/bopi'* is one of ADD/addi AND/andi, OR/ori, XOR/xori
MOVE(TEMP(*rd*), BINOP(*sop*,TEMP(*rt*),CONST(*shamt*)))	*sop' rd*,*rt*,*shamt* where *sop/bop'* is one of LSHIFT/sll, ARSHIFT/sra, RSHIFT,srl
MOVE(TEMP(*rt*), MEM(*loc*))	lw *rt*, *a*, if *loc* = CONST(*a*) lw *rt*, *lab*, if *loc* = NAME(*lab*) lw *rt*, *rs*, if *loc* = TEMP(*rs*) lw *rt*, *n*(*rs*), if *loc* = BINOP(PLUS,TEMP(*rs*),CONST(*n*)
MOVE(MEM(*loc*), TEMP(*rt*))	sw *rt*, *a*, if *loc* = CONST(*a*) sw rt, *lab*, if loc = NAME(*lab*) sw *rt*, *rs*, if *loc* = TEMP(*rs*) sw rt, *n*(*rs*), if *loc* = BINOP(PLUS,TEMP(*rs*),CONST(*n*)
MOVE(TEMP(*rd*), BINOP(MINUS, CONST(0), TEMP(*rs*)))	neg *rd*, *rs*
MOVE(TEMP(*rd*), BINOP(XOR, CONST(~1),TEMP(*rs*))	not *rd*, *rs*
MOVE(TEMP(*rd*), BINOP(XOR, CONST(~1), BINOP(OR,TEMP(*rs*),TEMP(*rt*))))	nor *rd*, *rs*, *rt*
JUMP(*addr*, *labs*)	j *addr*, if *addr* = CONST() or NAME(l) jr rs, if *addr* = TEMP(rs)

CJUMP(*rop*, *temp*, *temp*, *labt*, *labf*)
LABEL(*labf*)
	rop' where *rop/rop'* is one of
	EQ/beq, NE/bne,
	GE/bge, UGE/bgeu, GT/bgt ,UGT/bgtu,
	LE/ble, ULE/bleu, LT/blt, ULT/bltu

CJUMP(*rop0*,*temp*,CONST(0), *labt*, *labf*)
LABEL(*labf*)
	rop0' where *rop0/rop0'* is one of
	GE/bgez, GT/bgt, LE/blez,
	LT/bltz, EQ/beqz, NE/bnez

MOVE(TEMP(*rd*),CONST(1))
CJUMP(rop,TEMP(*rs*),TEMP(*rt*),*labt*,*labf*)
LABEL(*labf*)
MOVE(TEMP(rd),CONST(0))
LABEL(*labt*)
	rop' rd, rs, rt where *rop/rop'* one of
	EQ/seq, NE/sne,
	GT/sgt, GE/sge, UGT/sgtu, UGE/sgeu,
	LE/sle, LT/slt, ULT/sltu, ULE/sleu

MOVE(TEMP(rt),CONST(1))
CJUMP(LT,TEMP(rs),CONST(*imm*),labt,labf)
LABEL(falseLabel)
MOVE(TEMP(rd),CONST(0))
LABEL(trueLabel)
	slti *rt*, *rs*, *imm*

Figure 1: Correspondence between IRL statements and MIPS instructions

APPENDIX H: The IRL Register Allocator

The IRL register allocator translates an IRL program in MIPS normal form to one in **register reduced MIPS normal form**. This form is one that maps straightforwardly to MIPS machine code.

An IRL program is in register-reduced MIPS normal form if it satisfies the following seven conditions:

1. It is in MIPS normal form.

2. It does not use any pseudo-register temporaries. Rather, all temporaries are either register temporaries, argument temporaries, or spill temporaries. These three kinds of temporaries are all introduced by the register allocation phase. (See Appendix C for a discussion of the four types of temporaries.)

3. All formal parameters for IRL function declarations are argument temporaries.

4. Argument temporaries are only referenced in the following context:

   ```
   MOVE(TEMP(reg), TEMP(arg))
   ```

   where `reg` is a register temporary and `arg` is an argument temporary.

5. Spill temporaries are only used in the following contexts:

   ```
   MOVE(TEMP(reg), TEMP(spill))
   MOVE(TEMP(spill), TEMP(reg))
   ```

   where `reg` is a register temporary and `spill` is a spill temporary. These correspond to load and store instructions in MIPS code.

6. Caller-saves temporary registers (conventionally, those named `$tn`) that are live across a function call are saved to and restored from spill temporaries across that call.

7. Callee-saves temporary registers (conventionally, those named `$sn`) that used within a function body are saved to and restored from spill temporaries across the function body.

The interface to the IRL register allocator is the following function in the `RegAlloc` structure:

```
val alloc : IRL.pgm -> IRL.pgm
```

Given an IRL program in MIPS normal form, returns an IRL program in register-reduced MIPS normal form.

As discussed in Chapter 11 of Appel, there are many strategies that could be used to implement register allocation. The IRL register allocator is designed to support multiple strategies. At the current writing, only one strategy is implemented:

- *naïve*: this strategy assumes that *every* pseudo-register should be spilled. It uses real registers only for reading and writing values from spilled locations on an instruction by instruction basis. This is simple to implement, but extremely inefficient.

The naïve strategy is good enough to allow you to develop and test a `NormalToMIPS` code generator, but it will not yield very good MIPS code. It is hoped that more sophisticated register allocation strategies will be available by the time the contest is run.

While MIPS normal form makes *explicit* most aspects of compiled code, it leaves certain details *implicit*, especially those involving function calls. In order to perform its job, the register allocator must make certain assumptions about the real machine registers used by the following statements:

`MOVE(`*`et`*`,CALL(`*`e`*`,`*`es`*`))` uses the following registers in addition to those used in *`et`* and *`es`*:

- `$ra`,
- `$v0`,
- the first *n* of registers `$a0`, `$a1`, `$a2`, and `$a3`, where *n* is the smaller of 4 and length(*`es`*).
- All caller saves temporary registers (conventionally `$t0` -- `$t9`, though the register allocator is free to choose a different convention).

`TAILCALL(n,`*`e`*`,`*`es`*`)` uses the following registers in addition to those used in *`et`* and `es`:

- `$ra`,
- `$v0`,
- the first *n* of registers `$a0`, `$a1`, `$a2`, and `$a3`, where *n* is the smaller of 4 and length(*`es`*).

`RETURN(`*`n`*`, TEMP(`*`t`*`))` uses registers `$ra and $v0` in addition to *`t`*:

`HALT(`*`s`*`)` uses registers `$a0` and `$v0`.

APPENDIX I: The IRL Enumerator

Within the back end --- especially within the NormalToMIPS code generator --- it is sometimes necessary to compute global properties of a program or function definition. For instance, we might want to know what is the maximal index of a spill temporary within a function body, or a list of all labels used for global variables within a program.

This sort of information could be collected in a standard recursive traversal of the IRL program. For instance, to determine the maximal spill index of a list of statements, we could write a set of mutually recursive functions that operate on statements, statement lists, expressions, and expression lists. But every time we want to compute a different property, we need to write a different set of mutually recursive definitions, which is tedious and error-prone.

An alternative that works for many properties is to decompose the property determination into two processes: one that enumerates all the expressions and/or statements within a tree, and another that walks over the resulting list determining the property. It is often much easier to write a single function that walks over a list of IRL expressions than a set of mutually recursive functions that process IRL statements and expressions.

As an example of the power of this approach, consider the problem of collecting a set of all the temps used in a list of statements. First, let's assume that there is a function `IRLEnum.expsStms` that returns a list of all expressions found in a left-to-right preorder depth-first traversal of a list of IRL statements. As a concrete example of `expsStms`, suppose that SS is the following statement list:

```
[MOVE(TEMP(spill.3),TEMP($t2)),
 MOVE(TEMP($t5), BINOP(PLUS, TEMP($t3),TEMP($t1))),
 MOVE(TEMP($t4), TEMP(arg.2))]
```

Then `expsStms(SS)` would be the following list of expressions:

```
[TEMP(spill.3),
 TEMP($t2),
 TEMP($t5),
 BINOP(PLUS, TEMP($t3),TEMP($t1))
 TEMP($t3),
 TEMP($t1))
 TEMP($t4),
 TEMP(arg.2)]
```

It is now an easy matter to collect a set of all temporaries used within a list of statements as follows:

```
(* Local abbreviation *)
structure TS = Temp.TempSet

fun allTemps stms =
      let fun addTemp(TEMP(t),temps) = TS.add(temps,t)
            | addTemp(_,temps) = temps
      in List.foldr addTemp TS.empty (IRLEnum.expsStms stms)
      end
```

This sure beats writing a big set of mutually recursive functions to accomplish the same task. And the same expression enumerator can be reused to determine many such properties.

The `IRLEnum` structure provides a set of such enumerators, both for the expressions within a program phrase and for the statements within a program phrase. All of the following enumerators lists the expressions (or statements) of the given phrase in the order that they would be discovered in a left-to-right pre-order traversal of the phrase:

`val expsPgm : IRL.pgm -> IRL.exp list`
: Returns a list of all expression occurrences within a program, including those found in the function declarations as well as those found in the main body of the program.

`val expsStms : IRL.stm list -> IRL.exp list`
: Returns a list of all expression occurrences within a statement list.

`val expsStm : IRL.stm -> IRL.exp list`
: Returns a list of all expression occurrences within a statement.

`val expsExps : IRL.exp list -> IRL.exp list`
: Returns a list of all expression occurrences within an expression list (including subexpressions of the list components).

`val expsExp : IRL.exp -> IRL.exp list`
: Returns a list of all expression occurrences within an expression (including the expression itself and all immediate and non-immediate subexpressions of the expresion).

`val stmsPgm : IRL.pgm -> IRL.stm list`
: Returns a list of all statement occurrences within the program, including those found in the function declarations as well as those found in the main body of the program.

`val stmsStms : IRL.stm list -> IRL.stm list`
: Returns a list of all statement occurrences within the statement list (including substatements of the list components).

`val stmsStm : IRL.stm -> IRL.stm list`
: Returns a list of all statement occurrences within the statement (including the statement itself and all immediate and non-immediate substatements of the statement).

`val stmsExps : IRL.exp list -> IRL.stm list`
: Returns a list of all statement occurrences within an expression list.

`val stmsExp : IRL.exp -> IRL.stm list`
: Returns a list of all statement occurrences within an expression.

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

CS301 Problem Set 7

Due Friday, December 8, 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			
Problem 1 [80]			
Problem 2 [90]			
Problem 3 [80]			
Total [200]			