COMS W4115
Programming Languages and Translators
Programming Assignment 3: Tiger Interpreter

Prof. Stephen A. Edwards
Columbia University

Assigned March 25th, 2002
Due 11:59 PM on April 8th, 2002

> For this assignment, you will be adding a pass that translates semantically correct Tiger code into a pseudo assembly code that can be interpreted using supplied classes and easily translated into MIPS assembly code, the task for the fourth assignment. The goal of this assignment is to produce an interpreter for the complete Tiger language.

Compilation is a process of refinement: transforming a stream of characters into first tokens, then an abstract syntax tree, then into pseudo-assembly code, and finally into true assembly code for a real processor.

I have written a collection of classes (in prog3/Interp) that implements a simple intermediate language designed for implementing Tiger. These classes describe assembly language instructions, a stack, classes for representing objects in memory such as records, and methods that execute these instructions.

Your goal in this assignment is to translate the AST you generated in assignment 1 and checked in assignment 2 into a sequence of instructions in this intermediate code. Feel free to use, modify, or ignore any of the code I give you; you will be graded on whether the the test programs we run through your interpreter produce the correct result.

1 The Intermediate Code

I designed this intermediate code to be simple, contain instructions that work nicely for implementing the Tiger language, and have instructions that can be easily translated into actual assembly code. This intermediate code is similar to what is found in many compilers, a "three-address" code, so named because the arithmetic operators take three addresses: two sources and one destination.

Figure 1 shows the inheritance trees of the supplied classes. The two on the left (Rooted at Statement and Operand) describe the statements in the intermediate code and their operands (arguments) listed in Figure 2. Each instruction has its own class except for the binary operators (add, sub, etc.), which are implemented by the Binop class.

Every statement has a "next" field that links them together in a sequence. Calling Statement.append() on a statement appends a statement or a sequence to the sequence starting at that statement. It returns the appended statement, so statement sequences can be assembled easily as shown in the example in Figure 4.

The constructor for each class derived from Statement takes

```
mov dest, src
neg dest, src
add dest, src1, src2
sub dest, src1, src2
mul dest, src1, src2
div dest, src1, src2
equ dest, src1, src2
neq dest, src1, src2
lt  dest, src1, src2
leq dest, src1, src2
gt  dest, src1, src2
geq dest, src1, src2

jmp target
jsr target, depth
rts
bnz target, src
bz  target, src
Label:

sys index
psh offset
rec dest, size
arr dest, count, src
```

Figure 2: Pseudo-instructions for the simple assembly language. The first group manipulates data, the second group handles branching and subroutines, and the third group contains miscellaneous instructions.

Notation	**Addressing Mode**
10	Integer constant
`"hello"`	String constant
nil	nil constant
`Local1`	Label
fp(5)	Frame pointer relative
3*fp(4)	Static link relative
op[op]	Block relative

Figure 3: Addressing Modes

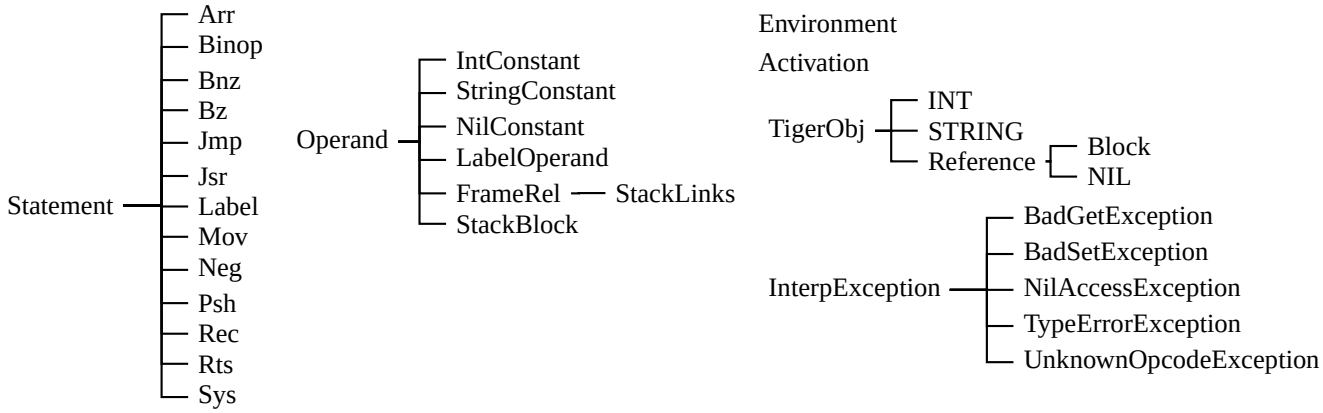

Figure 1: Supplied classes.

zero or more arguments that correspond to the instruction's operands. Sometimes these are simply constants (e.g., the argument of the psh instruction), but most of the time they are an Operand-derived object.

1.1 Operands

The intermediate code supports six different types of operands (addressing modes), listed in Figure 3. Integer and string constants are self-explanatory: they simply produce their value. Label operands are used in targets for control-flow instructions such as jmp.

The intermediate code expects all data to be stored on the stack (not the most efficient, but conceptually easy), and provides three ways to get at data on the stack. The simplest is frame pointer relative (FrameRel). Something like fp(n) refers to the nth value in the current activation record, which you probably want to use for temporaries.

Negative offsets (e.g., fp(-1)) access the activation record of the calling routine to access function call arguments and a return value. fp(-1) refers to the topmost element of the caller's stack, fp(-2) refers to the second-to-topmost element, and so forth. For functions that return a value, the suggested calling convention is to store the result at fp(-1), put the first argument at fp(-2), the second at fp(-3), and so forth. For functions that do not return a value, put the first argument at fp(-1), the second at fp(-2), etc.

Static link relative (StackLinks) such as k*fp(n) follows k static links before referring to the nth value. This allows direct access to variables in the local or any parent scopes.

Block relative (BlockRel) consists of two operands: a base operand that refers to a block (a record or array in a variable on the stack), and an offset that refers to an integer (either an integer constant for accessing a field of a record or an integer value for subscripting an array). The syntax for this mode is k*fp(n)[j] for accessing field j of the nth variable in the kth parent scope or k*fp(n)[j*fp(m)] for accessing the element indexed by j*fp(m)th of the array at k*fp(n).

1.2 Data-manipulation Instructions

The binary operator instructions (add, sub, ..., geq) have three arguments: a destination and two sources. Each reads the values of its two operands, performs the calculation, and stores the result in the destination. The mov instruction simply copies its single source to its destination. The neg instruction is like mov, but expects an integer source and negates its value before moving it to its destination.

1.3 Control-flow Instructions

The jmp (jump) instruction simply sends control to a label instruction. The bnz (branch non-zero) instruction checks the value of its source operand and sends control to a label if the value is non-zero. The bz (branch on zero) branches only if the source value is zero.

The jsr (jump to subroutine) does three things: it creates a new activation record, sets the return address to the next instruction, and follows the given constant number of static links to create the static link for the new activation record. The rts (return from subroutine) reverses this process, discarding the activation record and sends control to the return address stored in it.

The label instruction does nothing; it is simply a target for the control-flow instructions.

1.4 Miscellaneous Instructions

The sys (system call) function calls one of the built-in functions, such as print, given by its constant index. The psh (push) instruction adds or removes a given (constant) number objects from the current activation record, adjusting the size of the stack.

The rec (record) instruction creates a new Block of the given (constant) size and stores it at the destination. The arr (array) instruction creates a new Block of the given (variable) size, fills it with copies of the object given by the src, and stores it at the destination.

2 The Run-Time Environment

The run-time environment consists of a stack of activation records that contain stacks of TigerObj objects such as integers, strings, and references to memory blocks that also store arrays of TigerObjs to implement records or arrays.

The INT and STRING classes are little more than wrappers for Java int and String classes. One difference: the INT class is the only one with copy semantics; STRINGS and all others,

```
Label l = new Label("print");
Statement printFunc = l;
printFunc.append(new Sys(Sys.PRINT))
         .append(new Rts());

Statement s = new Psh(1);
s.append( new Mov(new FrameRel(0),
                  new StringConstant("Hello world\n")))
 .append( new Mov(new FrameRel(0),
                  new StringConstant("This works\n")))
 .append( new Jsr(new LabelOperand(l), 0) );

System.out.println("#### The Program:");
printFunc.printAll();
System.out.println();
s.printAll();

System.out.println("#### The Output:");
s.executeAll(false);
```

Figure 4: A code fragment that creates the "hello world" program, prints it, and runs it."

when copied, just make aliases in accordance to the Tiger semantics.

The rec and arr statements create new blocks, whose size remains fixed once created.

The Activation class represents a stacked activation record. It consists of a dynamic link (points to the activation record of its caller), a static link (points to the activation record of its lexically-scoped parent, set up by the jsr instruction), a return address (points to the statement just after the jsr statement that created the activation record), and a variable-sized stack of TigerObj objects.

The psh instruction grows and shrinks the size of the topmost activation record, which corresponds to moving the stack pointer. Use this to allocate local variables and temporaries.

3 Putting It All Together

Figure 4 shows a fragment of code that creates the "hello world" program to illustrate how these classes are intended to be used. The Statement.printAll method prints out a sequence of instructions starting at the given statement. Similarly, the Statement.executeAll method executes a sequence of instructions, and takes a single argument that indicates whether each instruction should be printed as it is executed for debugging purposes.

I've supplied `TigerTranslate.g` as a starting point for your translator. Like `TigerSemant.g` from assignment 2, this is a partially-working version of the translator that is missing some important functionality. You need to complete the code for handling control-flow (if, while, for, and break statements), function declarations and calls, and constructors for records and arrays.

The TI.java file is a simple front-end for the interpreter. You will want to modify it to enable and disable printing.

The RecordInfo class I supplied is designed to manage information about variables and their location in the activation record for a function. It supplies two main features: a mechanism for allocating and deallocating data on the stack in an activation record, and a symbol table mechanism for tracking where each variable is stored on the stack and the label for each function definition.

The stack management mechanism provides the ability to allocate and free stack space. The newTmp method allocates space for a new field (variable, temporary, etc.) on the stack and returns an operand that refers to it. This is used, for example, to create space to store the value of the left-hand-side of a binop as its being calculated. The mark method remembers how much space has been allocated and the release method discards the space allocated since the last free.

The rule for BINOP illustrates how these work together:

```
#( BINOP
   expr[d,r]
   { r.mark(); Operand tmp = r.newTmp(); }
   expr[tmp, r]
   {
     // FIXME: Change this to use the right operator
     r.append( new Binop(Binop.ADD, d, d, tmp));
     r.release();
   }
 )
```

The newTmp call allocates space for where the second expression will store its result. The append operation adds a Binop (add in this incompete example) that adds the result of the first expression to the second expression and stores it in d, the destination operand. Finally, the release method frees the stack space allocated by the newTmp call.

The symbol table mechanism in RecordInfo maintains a stack of scopes (I took the code directly from the second assignment) and provides a way to locate a variable in the stack.

The enterScope and leaveScope methods operate just as they did for static semantic analysis: enterScope begins a fresh scope in the function and leaveScope forgets all the functions and variables that have been defined since the last enterScope.

The newVar method takes the name of a variable and binds it to a newly-allocated position on the stack in the topmost scope using newTmp. The findVar method returns an operand that can access a variable given its name. In addition to the current activation record, FindVar also looks through all lexically-scoped parents to also find variables there. The returned operand correctly follows the right number of static links.

The jsr instruction (Jump-to-subroutine) takes a constant parameter for the number of static links to follow to create the next static link. You will have to write the code that calculates this number (it depends on whether the called function is in the current scope or an outer one).

For the skeleton Tiger interpreter (TI.java), I inserted a psh instruction at the beginning that allocates as much space as the stack will ever use (RecordInfo.size() gives this information), but you might want to use a less brute-force solution. There's a tradeoff between calling psh frequently (i.e., every time stack space is allocated or released) versus calling it exactly once at the beginning of a function.

Function arguments are placed at the end of an activation record so that the caller can access them using operands like fp(-1), fp(-2), etc.

The sys instruction invokes standard library functions (I've written the print and printi functions for you). This instruction behaves like a normal called function in that it expects to find its arguments at fp(-1), fp(-2), etc. For each standard library function, write a stub function that consists of a sys instruction followed by an rts, and translate a call to a library function into a jsr to one of these stubs. For example, have your interpreter create code like

```
Printi:
  sys 1
  rts

  % Print a 1
  mov fp(-1), 1
  jsr Printi, 0
```

3.1 Things You Will Have To Solve

Although instructions are provided for calling and returning from functions (i.e., jsr and rts), you will need to make sure these instructions are used correctly. The challenges here are figuring out how to pass function parameters (make sure a function can use fp(-2) to get the first argument) and calculating the number of static links to follow to get the static link for the new stack (an operand of the jsr instruction).

Record variable creation and access has not been implemented. You will need to use some of the code you created in the second assignment to track the types so you will know the offset into a record for the lvalue rule.

Control-flow instructions (if-then-else, if, while, for, and break) have to be implemented. There are conditional and unconditional branching instructions in the intermediate code for this.

The lazy logical operators need to be translated into conditional branches. The rule for BINOP will have to be changed to do this.

You need to implement the remaining system functions and make sure they can be called.

4 Deliverables

Feel free to use, modify, or ignore any of the files I give you. Ultimately, you will only be graded on whether the simple test programs on which we run your interpreter produce the correct results. Make sure you implement the print and printi standard library functions.

As before, use `~cs4115/bin/submit_code` to submit

- Your customized `TigerTranslate.g` file.
- All other .java files (and .class files for the parser) you use or create, including we gave you.
- A README file describing your interpreter. I want to hear how you dealt with
  - Calculating the number of static links to follow at a jsr.
  - Where you put the arguments for a jsr.
- A subdirectory called "tests" containing test programs.
- A file called `MEMBERS` that contains a space-separated list of the uni IDs of each of the members in your group.

Make sure we can build your semantic analyzer by running ANTLR on `TigerTranslate.g` and then `javac` on `TI.java`. We do not want a Makefile.