Programming Assignment 3: A Translator and Interpreter

COMS W4115

Prof. Stephen A. Edwards

Spring 2002

Columbia University
Department of Computer Science

Programming Assignment 3

I’m giving you classes for an intermediate representation (IR) and an interpreter

Your job is to complete the partially-written translator skeleton

Result: interpreter able to execute complete Tiger programs

The Interpreter

Source program

↓ Lexer/Parser

AST

↓ Translate

IR

↓ Interpret

Output

The Intermediate Representation

I designed it to be

- easy to execute
- easy to translate into actual (MIPS) assembly
- easily generated from Tiger

An idealized low-level assembly language supporting

- accessing a stack with static links
- arrays and records
- the standard library

Assembly language

Most assembly languages characterized by

- Opcodes: instructions such as add, mov, jmp

  The Tiger IR is standard with special instructions for records and arrays.

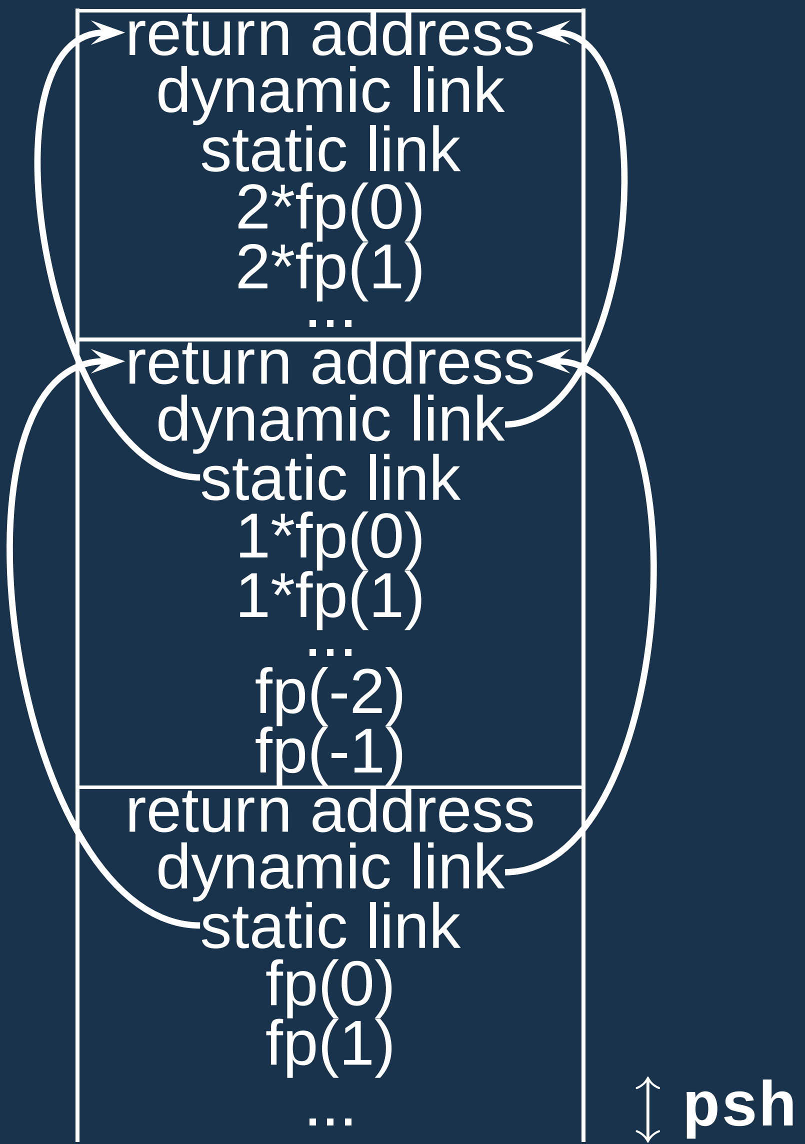
- Operands/Addressing modes: How to route data to and from these commands

  Our addressing modes are stack-relative with knowledge of static links to simplify Tiger variables.

- Programmer's Model: What things (registers, memory, etc.) the instructions can access

  Just a stack. No registers, memory is implicit.

Programmer's Model



Top activation record:

fp(0)

fp(1) ...

Last activation record:

fp(-1)

fp(-2) ...

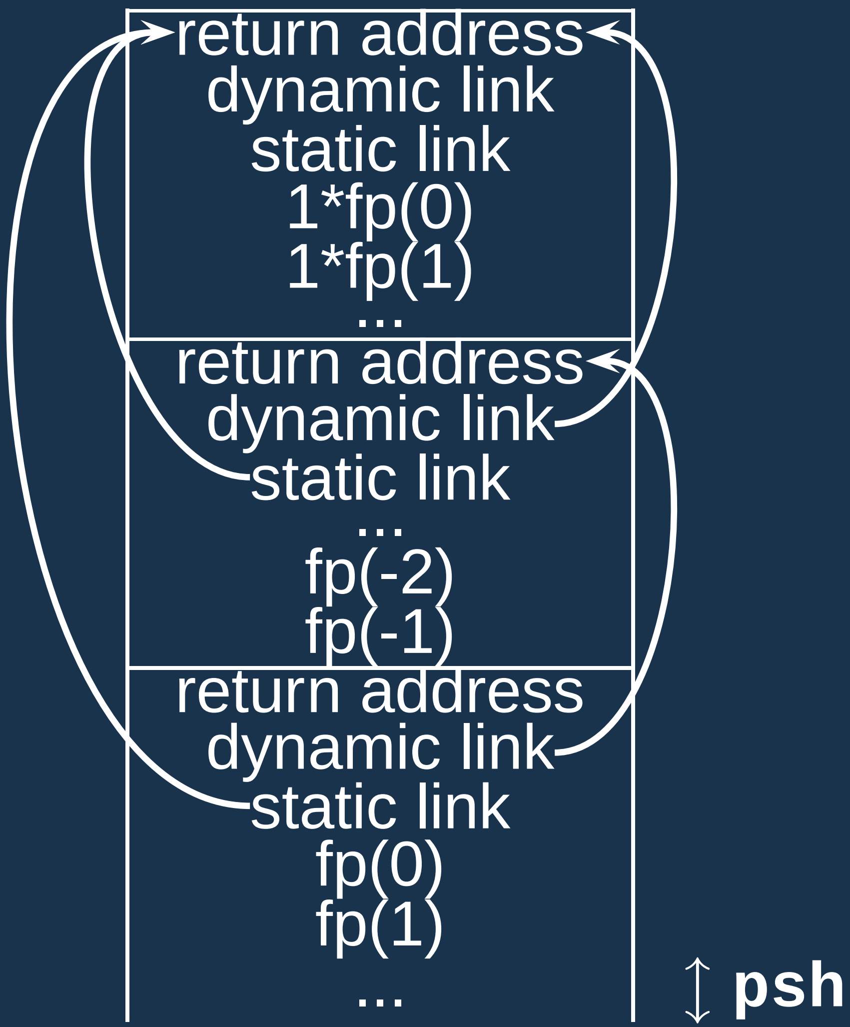
Following static links:

1*fp(0)

1*fp(1) ...

2*fp(0) ...

Programmer's Model



1*fp(0) follows one static link to reach an activation record.

```
let
  function A() =
    let
      function C() = ...
      function B() = C()
    in B() end
in A() end
```

Addressing Modes

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 (index into first using second)

Data Manipulation Statements

`mov dest, src`

Copy the contents of the source to the destination

`neg dest, src`

Read the source, negate it (must be int), and store it in the destination

`add dest, src1, src2`

Binary arithmetic commands: Perform src1 op src2, store result in dest. Also sub, mul, div, equ, neq, lt, leq, gt, geq.

Called “Three Address Code.”

Control-Flow Statements

`Label:`

A branch target. A label is a statement not an attribute in this IR.

`jmp target`

Unconditional branch to the target

`jsr target, depth`

Jump to a subroutine. Creates a new activation record, stores the return address, and creates the new static link by following *depth* existing static links.

`rts`

Return from subroutine. Destroys topmost activation record and branches to the return address.

Conditional Branching Statements

`bnz target, src`

Branch to target if source is non-zero.

`bz target, src`

Branch to target if source is zero.

Miscellaneous Statements

`sys index`

Call system function *index* (e.g., print, printi, flush)

`psh offset`

Allocate or release *offset* fields on the stack in the current activation record

`rec dest, size`

Allocate a new record with *size* fields and store it at the destination

`arr dest, count, src`

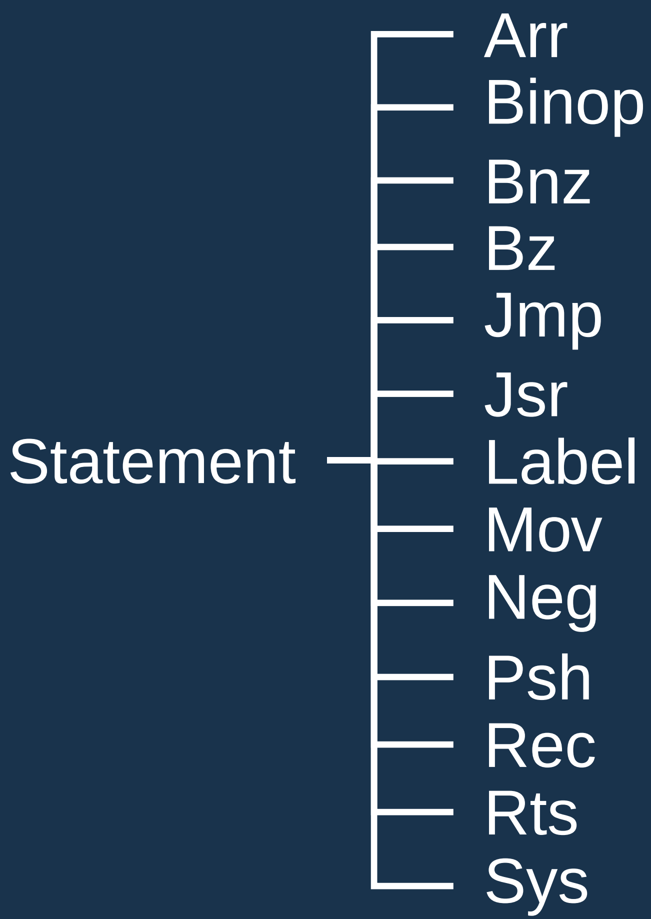
Allocate a new array with *count* fields, fill it with copies of *src*, and store it at the destination

Run-time Tiger Objects

- TigerObj
  - INT
  - STRING
  - Reference
    - Block
    - NIL

```
class TigerObj {
  TigerObj copy()
  String string()
}
```

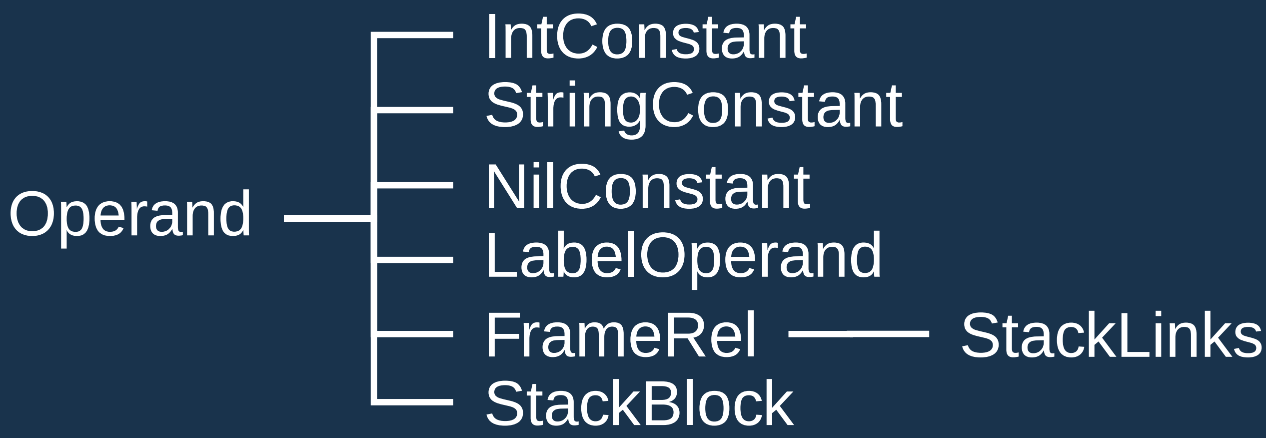
Statement Classes



Statement Classes

```
class Statement {
  Statement next
  public String string()
  public void printAll()
  public Statement execute(Environment e)
  public void executeAll(boolean trace)
  public Statement append(Statement s)
  public Statement insert(Statement s)
}
```

Operand Classes



Operand Classes

```
class Operand {
    public String string()
    public void set(Environment e, TigerObj o)
    public TigerObj get(Environment e)
}
```

Hello World

```
  psh 1  % Make space for the argument
  mov fp(0), "Hello world\n"
  jsr print, 0  % Print the first string
  mov fp(0), "This works\n"
  jsr print, 0  % Print the second string

print:
  sys 0 % Call print()
  rts   % return to call
```

prints

```
Hello world
This works
```

Hello World

```
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 Jsr(new LabelOperand(l), 0) );
 .append( new Mov(new FrameRel(0),
                  new StringConstant("This works\n")))
 .append( new Jsr(new LabelOperand(l), 0) );

s.printAll();          // Print the main program
printFunc.printAll();  // Print the print function
s.executeAll(false);   // Execute the main program
```

```
  psh 1
  mov fp(0), ”Hello world”
  jsr print, 0
  mov fp(0), ”This works”
  jsr print, 0

print:
  sys 0
  rts
```

Translation

Translation

Yet another ANTLR pass.

RecordInfo class provides context

- What variables are in scope
- How to access each variable (an Operand)
- Allocation in the current activation record

Translating Expressions

The main operation:

```
expr [ Operand d, RecordInfo r ]
  { Operand o; }
  : n:NUMBER
    { int i = Integer.parseInt(n.getText(),10);
      r.append(new Mov(d, new IntConstant(i))); }
  | o=lvalue[r] { r.append( new Mov(d, o)); }
  | #( BINOP
       expr[d,r]
       { r.mark(); Operand tmp = r.newTmp(); }
       expr[tmp, r]
       { r.append( new Binop(Binop.ADD, d, d, tmp));
         r.release();
       }
     )
```

Translating If-Then-Else

if *expr1* then *expr2* else *expr3*

```
  d = expr1
  bz Else, d
  d = expr2
  goto Exit
Else:
  d = expr3
Exit:
```

Translating While

while *expr1* do *expr2*

```
Again:
  d = expr1
  bz Break, d
  d = expr2
  jmp Again
Break:
```

Translating For

for *I* := *expr1* to *expr2* do *expr3*

```
   I = expr1
Again:
   d = expr2
   lt  d, I, d
   bnz Exit
   d = expr3
   sub I, I, 1
   jmp Again
Exit:
```

Calling Functions

Calling foo(x : int, y: int) : int with foo(3, 4)

```
  mov fp(7), 3
  mov fp(6), 4
  jsr Foo, 0
  mov out, fp(8)

Foo:
  mov x, fp(-2)
  mov y, fp(-3)
  ...
  mov fp(-1), result
  rts
```

Assumes the current activation record has nine fields.

Make sure caller follows assumption made by callee.

Return value is at TOS (fp(-1)), first argument at fp(-2), etc.

Standard Library Functions

Calling print(x : string)

```
  psh 1
  mov fp(0), "Hello"
  jsr Print

Print:
  sys 0  % Assumes argument at fp(-1)
  rts
```

A hack, but a convenient one.

Resembles many processor's trap facility for calling system functions.

Managing the Stack

The psh instruction adjusts the stack pointer.

```
  jsr Foo


Foo:
  mov fp(0), 5    % Error: activation record empty
  psh 2
  mov fp(0), 5    % OK
  mov fp(2), 5    % Error: only space for 2
```

TI currently records the maximum stack space a function consumes, then **psh**s that amount of space on entry.

Not good for function parameters: probably want to psh as space is needed and left unused (mark() and release()).

Records

```
let
  type pt = { x : int, y : int }
  var p := pt { x = 7, y = 9 }
  var z := 11
in z := p.y end

  psh 2                        % space for p, z
  rec fp(0), 2                 % allocate p
  mov fp(0)[0], 7              % p.x = 7
  mov fp(0)[1], 9              % p.y = 9
  mov fp(1), 11                % z := 11
  mov fp(1), fp(0)[1]          % z := p.y
```

Arrays

```
let
  type a = array of int
  var a := a [5] of 0
  var b := 3
in
  b := a[3];
  a[4] := 7;
end
```

```
psh 5

mov fp(0), 5
mov fp(1), 0
arr fp(0), fp(0), fp(1)

mov fp(2), 3

mov fp(3), 3
mov fp(2), fp(0)[fp(3)]

mov fp(4), 4
mov fp(0)[fp(4)], 7
```

Lazy Logical Operators

```
d = a & b | c

  bz L1, fp(0)     % a
  bnz True, fp(1) % a
L1:
  bz False, fp(2) % c
True:
  mov fp(3), 1     % d = 1
  jmp Next
False:
  mov fp(3), 0     % d = 0
Next:
```

Comments

I chose a stack model because it’s the minimum necessary.

All modern processors use registers, which are harder to deal with because they run out. (“register spilling”)

The interpreter is terribly inefficient; a smarter one might use arrays and less object orientation.

The intermediate representation is fairly standard, although it has more high-level constructs than is typical.

A *real* compiler would greatly optimize (simplify) the output.