Chapter 6: Activation Records
Activation Records

Because of recursion, several invocations of a subroutine may exist simultaneously.

Each invocation of a subroutine has its own instances of the local variables lasting from subroutine entry to return.

A subroutine invocation requires the runtime system maintain information about the invocation including the values of local variables (and much more: return address, saved state of the machine, temporaries values, access to the non-local environment, …).

This collection of information is called an activation record.
A subroutine returns only after all the function it has called have returned. Thus subroutine invocations behave in a LIFO manner. So keep activation records in a stack.
Activation Records

Appel, 2nd edition, page 118. “For historical reasons, the run-time stack usually starts at a high memory address…”

Runtime Organization of a Program in Memory

- High memory
  - The stack (local variables)
- Low memory
  - The heap (dynamically allocated variables)
  - Globals variables, program code (read only)
Static link
Dynamic link
Return address
Parameters
Local variables
Register save area
Usual Layout of Activation Record for SPARC architecture
SPARC calling sequence

1. Caller puts arguments in %o0, %o1, ..., %o5. For example:
   mov 1,%o0  ! pass argument 0 in the register %o0
   set LABEL,%o1  ! pass argument 1 in the register %o1
   mov %l4,%o2  ! pass argument 2 in the register %o2

2. If more than six arguments, then the caller puts them in the (caller’s) argument build area. For example:
   mov 6,[%sp+4*(16+6)]  ! pass argument 6 on the stack
   set LABEL,[%sp+4*(16+7)]  ! pass argument 7 on the stack

3. Caller then executes call instruction, which saves the pc in %o7 and jumps to a label. For example:
   call f
   nop
4. Callee executes **save** instruction, which shifts the register window set and reserves space for new stack frame. For example:

```assembly
save %sp, -4*(16+1+t+x)&-8, %sp
```

where \( t \) is the number of locals and temporaries, and \( x \) is the maximum arguments needed by any subroutines of the callee. Stack pointers must be double word aligned.

5. If needed, the callee saves the incoming arguments (0 through 5) into the argument build area of the caller’s frame. For example:

```assembly
st %i0, [%fp+4*(16+0)]  ! store arg 0 into stack
st %i1, [%fp+4*(16+1)]  ! store arg 1 into stack
```

Local variables and temporaries of the current frame are accessed with negative offsets from the \%fp. For example:

```assembly
ld [%fp-4*(1+0)], %l0  ! load local/temp 0
ld [%fp-4*(1+1)], %l1  ! load local/temp 1
```
SPARC return sequence

1. Callee puts (integer) return value in %i0. For example:
   \[
   \text{mov} \ 1, %i0 \quad \text{! put return value in} \ %i0
   \]

2. Callee executes \texttt{restore} instruction, which resets the register window set and pops frame.

3. Callee then executes \texttt{ret} instruction, which jumps to %i7+8 (just past the call and its delay slot in the caller). For example:
   \[
   \text{ret} \quad \text{! "jmp1} \ %i7+8, \ %g0" \text{ after next instruction}
   \]
   \[
   \text{restore} \quad \text{! restore register window set}
   \]
Nonlocal variable access

1. Whenever a function $f$ is called, it can be passed a pointer to the frame of the function statically enclosing $f$; this pointer is the static link.

2. A global array can be maintained, containing – in position $i$ – a pointer to the frame of the most recently entered procedure whose static nesting depth is $i$. This array is called a display.

3. When $g$ calls $f$, each variable of $g$ that is actually accessed by $f$ (or by any function nested inside $f$) is passed to $f$ as an extra argument. This is called lambda lifting.
Nonlocal Variable Access

A local variable is accessed as follows:

\[ r := \text{fp} \quad \text{! relative to frame pointer} \]
\[ r := \text{M}[r+offset] \quad \text{! access local variable} \]

A non-local variable at static distance two is accessed as follows:

\[ r := \text{fp} \quad \text{! relative to frame pointer} \]
\[ r := \text{M}[r+sl] \quad \text{! follow first static link} \]
\[ r := \text{M}[r+sl] \quad \text{! follow second static link} \]
\[ r := \text{M}[r+offset] \quad \text{! access the variable at sd=2} \]
Nonlocal Variable Access

\[ M[fp+k] \]

\[ \text{MEM (BINOP (PLUS,}
      \text{ new TEMP (frame.FP()), CONST (k))}) \]
line 21. prettyprint calls show
   (show’s SL is set to prettyprint’s own frame pointer)

line 15. show calls indent
   (indent’s SL is set to show’s own frame pointer)

line 13. indent calls write

line 17. show calls itself recursively

line 15. show calls indent

line 13. indent calls write
line 14. indent accesses the variable output
An abstract structure to represent frame information

package frame;

/* Appel, 2nd edition, Chapter 6, page 127 */

public abstract class Frame {
    public final Label instructionPointer;
    public final List<Access> formals; // including implicit static

    public Frame (Label i, List<? extends Access> f) {
        instructionPointer = i; formals = f;
    }

    public int getNumberOfFormals () { return formals.size(); }

    abstract public Frame newFrame (Label name, List formals);

    abstract public Access allocateLocal (boolean escape);

    public Access allocateLocal () { return allocateLocal(true); }

    public abstract temp.Temp FP(); /* Chapter 7, page 143 */
}

package frame;

import tree.Exp;
public abstract class Access {
    public abstract Exp access (Exp framePtr);
}
The class Frame holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function $f$ with $k$ formal parameters, call newFrame($f, l$), where $l$ is a list of $k$ booleans: true for each parameter that escapes and false for each parameter that does not. (In MiniJava, no parameters ever escape.) The result will be a Frame object.


Calling newFrame() and getting a new frame might change the caller’s frame. For example, the maximum size of the argument build area may increase.
package temp;
public class Temp {}
public class Label {}

new temp.Label ("ClassName"+$"+"methodName")
package translate;
import frame.Frame;
import translate.Access;
import java.util.LinkedList;

public class Level {
    public final Level parent;
    public final Frame frame;
    public Level (Level p, String name, LinkedList formals);
    public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }
}

// Obsolete?
// Appel, 1st edition, page 147

package translate;
import frame.Frame;
import translate.Access;
import java.util.LinkedList;

public class Level {
    public final Level parent;
    public final Frame frame;
    public Level (Level p, String name, LinkedList formals);
    public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }
}
Chapter 7: Translation to Intermediate Code
Intermediate Representation (IR)

- Easy to convert from the abstract syntax; easy to convert to assembly. Should support machine-independent optimizations.
- Often compilers use several IRs.

\[ \text{syntax} \Rightarrow \text{IR}_1 \Rightarrow \text{IR}_2 \ldots \Rightarrow \text{IR}_k \Rightarrow \text{assembly code} \]

- Tiger compiler uses only one IR: `tree.Exp`
abstract class Exp

  class CONST (int value)
  class NAME (temp.Label label)
  class TEMP (temp.Temp temp)
  class BINOP (int binop, Exp left, Exp right)
  class MEM (Exp exp)
  class CALL (Exp fun, ExpList args)
  class ESEQ (Stm stm, Exp exp)

abstract class Stm

  class MOVE (Exp dst, Exp src)
  class EVAL (Exp exp)
  class JUMP (Exp exp, temp.LabelList targets)
  class CJUMP (int rel,Exp l,Exp r,Label t,Label f)
  class SEQ (Stm left, Stm right)
  class LABEL (Label label)
Intermediate Representation

Also \texttt{ExpList} and \texttt{LabelList}. These are used by canonicalization in phase 8. So we can’t change them.
Intermediate Representation

Expressions which stand for the computation of some value (possibly with side effects):

**CONST(i)** The integer constant i.

**NAME(n)** The symbolic constant n corresponding to an assembly language label.

**TEMP(t)** Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

**BINOP()** The application of binary operator o to operands e1 and e2. Subexpression e1 is evaluated before e2.
Intermediate Representation

The remaining expressions which stand for the computation of some value (possibly with side effects):

MEM(e) The contents of a word of memory at address e.

CALL() Procedure call.

ESEQ(s,e) The statement s is evaluated for side effects, then e is evaluated for a result.
Intermediate Representation

The statements of the intermediate representation which perform side effects and control flow:

MOVE(TEMP t,e) Store the results of evaluating e into the temporary t.

MOVE(MEM e1,e2) Store the results of evaluating e2 at the address e1.

EVAL(e) Evaluate e for it side effects and discard the result.

*Called EXP by Appel.)

JUMP(e,l) Transfer control to address e
Intermediate Representation

The remaining statements which perform side effects and control flow:

**CJUMP** Evaluate $e_1$ and $e_2$ in that order; compare using relational operator $o$. If the result is true, jump to $t$, otherwise jump to $f$.

**SEQ(s1, s2)** The statement $s_1$ followed by $s_2$.

**LABEL(n)** Define name to be the current machine code address.
Example

new SEQ (new MOVE (temp,1), // temp := 1;
new SEQ (new CJUMP(<,x,5,T,F), // if x<5 goto T else F
new SEQ (new LABEL(F), // F:
new SEQ (new MOVE (temp,0), // temp := 0;
    new LABEL(T) // T:
))))
Example 1

Abstract Syntax

\[ \text{OpExp(PLUS,}\]
\[ \text{IntExp(3),}\]
\[ \text{IntExp(4))} \]

Intermediate Trees

\[ \text{BINOP(PLUS,}\]
\[ \text{CONST 3,}\]
\[ \text{CONST 4)} \]
package translate;
class GenericIRTree {
    abstract Exp asExp();
    abstract Stm asStm();
    Stm asCond (Label t, Label f) { throw new UnsupportedOperationException();
    public String toString () {
        return String.format ("IR: %s", asStm().toString());
    }
}

Three views of the code:

1. as an expression
2. as a statement
3. as a conditional
“The whole point of the Cx representation is that conditional expressions can be combined easily with the MiniJava operator &&.” Appel, 2nd, Section 7.2, page 149.
Generic IR Trees

class ExpIRT and extends GenericIRT

class StmIRT and extends GenericIRT

abstract class Cx extends GenericIRT

class RelCx extends Cx /* Page 149. */
Generic IR Trees

class ExpIRTree extends GenericIRTree {
    private final tree.Exp exp;
    ExpIRTree (tree.Exp e) { exp = e; }
    tree.Exp asExp() { return exp; }
    tree.Stm asStm() { return new tree.EVAL(exp); }
    // unCx ....
}

class StmIRTree extends GenericIRTree {
    private final tree.Stm stm;
    StmIRTree (tree.Stm s) { stm = s; }
    tree.Stm asStm() { return stm; }
    // asExp, asCond not implemented
}
Generic IR Trees

abstract class Cx extends GenericIRTree {
    tree.Exp unEx() { /* Program 7.2, page 161. */ } 
    abstract tree.Stm unCx (Label t, Label f);
    // unNx "left as exercise"
}

“Making ‘simple’ Cx expressions from Absyn comparison operators is easy with the CJUMP operator.” Appel, 2nd, Section 7.2, page 149.

class RelCx extends Cx { /* Page 149. */
    final private int relop;
    final private tree.Exp left, right;

    RelCx (int op, tree.Exp l, tree.Exp r) {
        relop = op; left = l; right = r;
    }

    public tree.Stm asCond (Label t, Label f) {
        // new tree.CJUMP
    }
}
If \( i = \text{Translate}("x<5")\), then \( \text{asCond}(i,t,f) \) should be \( \text{CJUMP}(\text{LT},x,t,f) \).
This sketch of the class appears in Appel, 2nd, Section 7.2, page 150.

class IfThenElseExp extends GenericIRTree {
    private final GenericIRTree cond, e2, e3;
    final Label t = new temp.Label("if","then");
    final Label f = new temp.Label("if","else");
    final Label join = new temp.Label("if","end");
    IfThenElseExp (GenericIRTree c, GenericIRTree thenClause, GenericIRTree elseClause) {
        assert c!=null; assert thenClause!=null;
        cond = c; e2 = thenClause; e3 = elseClause;
    }
    public Exp asExp() { /* ... */ }
    public Stm asStm() { /* ... */ }
    public Stm asCond (Label tt, Label ff) { /* ... */ }
}

A Simpler Translate

“To simplify the implementation of Translate, you may do without the Ex, Nx, Cx constructors. The entire Translate module can be done with ordinary value-expressions.” Appel, page 178.
Fragments
7.1 Draw a picture of the IR tree that results from each of the following expressions. Assume all variables escape.

b.

```plaintext
let
  var i := 8
  type intArray = array of int
  var b := intArray[10] of 0
in
  b[i+1]
end
```
Chapter 8: Basic Blocks and Traces
The constructs of the intermediate representation trees of package tree are crafted to match the capabilities of real machines. Yet some aspects do not correspond exactly in order to facilitate the translation from the abstract syntax.

- **CJUMP** can jump to one of two labels, but real machine instructions fall through to the next instruction if the condition is false.

- **ESEQ** within expressions require attention to the order of evaluating the subtrees $2*(x:=3)+x$ BINOP(+, BINOP(*,2,ESEQ(x:=3,3)), x)

- **CALL** within expressions

- **CALL** within **CALL** complicates passing parameters in a fixed set of registers
We can take any tree and rewrite it to an equivalent tree without the troublesome cases. The SEQ constructs will all move to the top of the tree and become unnecessary. The tree can be replaced by a list of the other constructs.

1. The tree is flattened by removing SEQ and ESEQ
2. The list is broken into basic blocks containing no internal jumps
3. The basic blocks are ordered into a set of traces in which every CJUMP is followed by its false label.

The package **canon** does all of this.