Code Generation

Lecture 13-15

Lecture Outline

- Stack machines
- Low-level Java (LLJ): a target language for code generation
- Code generation for TYPEDLET
- Code generation for TYPEDREFS

Stack Machines

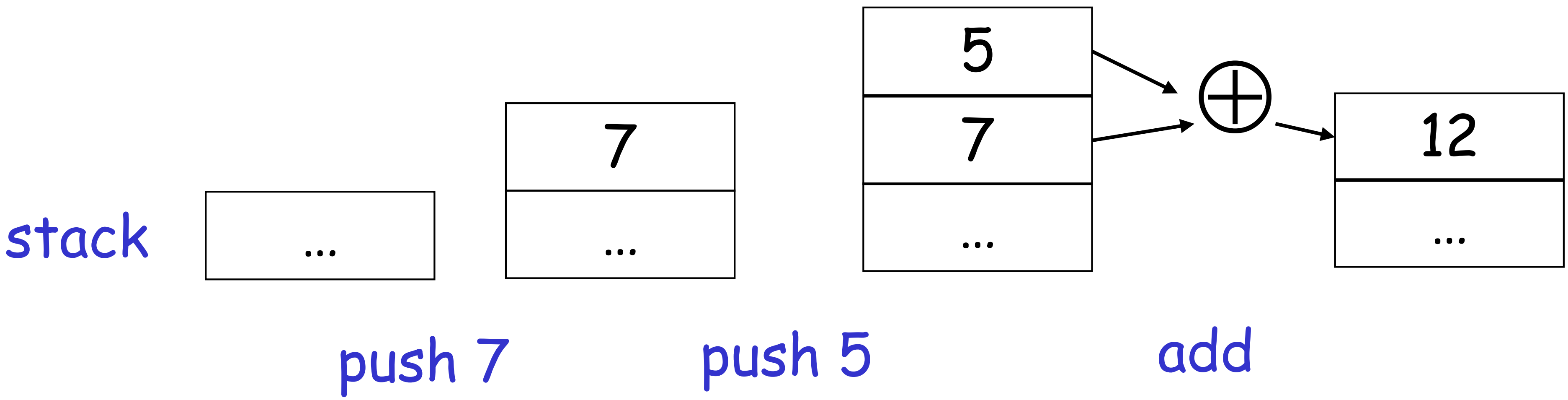
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

Example of a Stack Machine Program

- Consider two instructions
  - push i - place the integer i on top of the stack
  - add - pop two elements, add them and put the result back on the stack
- A program to compute 7 + 5:

push 7
push 5
add

Stack Machine. Example



- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Why Use a Stack Machine ?

- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme
- And therefore a simpler compiler
  - This is what you have to do for PA5

Why Use a Stack Machine ?

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction "add" as opposed to "add r_1, r_2"

  ⇒ Smaller encoding of instructions

  ⇒ More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model

Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register/local variable (called accumulator)
  - Register accesses are faster
- The "add" instruction is now

  acc ← acc + top_of_stack

  - Only one memory operation!

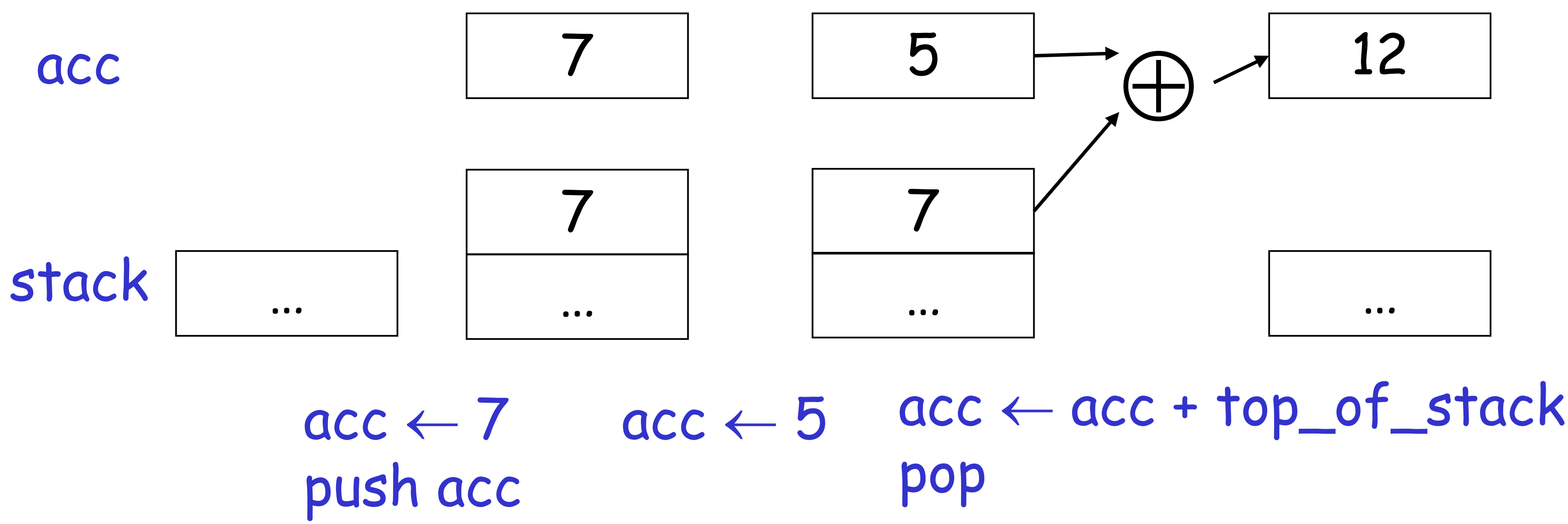
Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
- For an operation $op(e_1,\ldots,e_n)$ push the accumulator on the stack after computing each of $e_1,\ldots,e_{n-1}$
  - The result of e_n is in the accumulator before op
  - After the operation pop n-1 values
- After computing an expression the stack is as before

Stack Machine with Accumulator. Example

- Compute 7 + 5 using an accumulator



A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Notes

- It is **very important** that the stack is preserved across the evaluation of a subexpression
  - Stack before the evaluation of 7 + 5 is 3, <init>
  - Stack after the evaluation of 7 + 5 is 3, <init>
  - The first operand is on top of the stack

From Stack Machines to LLJ

- The compiler generates code for a stack machine
- We want to run the resulting code for a processor
  - one can generate for MIPS, x86, ARM, JVM, LLVM
  - we generate code in low-level Java (LLJ), a subset of Java
- LLJ prevents usage of arbitrary classes, closures, arbitrary local variables, method arguments, method returns, etc.
  - closely resembles machine code
  - uses arrays to simulate stack, activation records, and objects
  - uses global static variables to simulate registers
  - illustrates key concepts of code generation: easy to debug
  - same techniques can generate code for other target languages

Simulating a Stack Machine in LLJ

- The accumulator is kept in the global variable a0
- Four additional global variables: t1, t2, t3, t4
- The stack is simulated using a global array stack
- The index of the next location on the stack is kept in the global variable sp
  - A new value pushed to the stack is stored at index sp
  - The top of the stack is at index sp – 1
- A stack frame is used to store variable bindings
  - the index of the next location on the stack frame is kept in the variable fp

Value class in LLJ: to denote values of any type

```
class Value {
   public int i;
   public boolean b;
   public Lambda l;
   public String s;
}
```

If *C/C++* is the target language, one can use *C*'s union.

A Sample of LLJ statements

- a0.i = t1.i + a0.i;
- a0.b = t1.i > a0.i;
- a0.i = 8;
- stack[sp++].i = a0.i;
  - push an integer to the top of the stack
- t1.i=stack[sp-1].i;
  - load an integer in t1 from the top of the stack
- sp--;
  - pop stack
- frame[fp++] = new Value(); frame[fp-1].i = a0.i;
  - create a new memory location and bind it to the variable with index 0
- a0.i=frame[fp-1-j].i;
  - load an integer in a0 from the variable with index j
- frame[--fp] = null;
  - remove binding of the variable at index 0

A Comprehensive set of all valid instructions in LLJ

In the following, $f could be i, s, b, l and $r or $r<i> could one of a0, t1, t2, t3, t4, $l could be any valid label, $fun is the name of any class that extends Lambda, $c is some integer literal, $lit is some integer, string, or boolean literal

Assigning literals

$r.$f=$lit; // load some literal in a register

$r.l=new $fun(frame,fp); // a lambda literal

Function call

$r.l.apply(); // call a lambda

Binary intrsuctions

$r1.s=$r2.$2+$r3.$3; // where $2 and $3 could be i or s, but not i simultaneously

$r1.i=$r2.i $op $r3.i; // where $op could be +, -, *, /

$r1.b=$r2.i $op $r3.i; // where $op could be >, <, <=, >=, ==, !=

Jump related instructions

case $l: // a label

{label=$l;break;} // unconditional jump

if($r.b){label=$l;break;} // conditional jump

if(!$r.b){label=$l;break;} // conditional jump

Print a resister

System.out.println($r.$x) // $x is i, b, or s

Modify and read from stack

stack[sp++].$f=$r.$f; // push the value of a register

$r.$f=stack[sp-1].$f; // load top of stack to a register

sp--; // pop

stack[$c].$f=$r.$f;

$r.$f=stack[$c].$f;

Modify and read from frame

frame[fp-1-$c].$f=$r.$f; // store a register in a variable

$r.$f=frame[fp-1-$c].$f; // load a variable in a register

frame[fp++].$f=new Value(); //allocate space for a local variable

frame[--fp]=null;//deallocate space which was allocated for a variable

frame[$c].$f=$r.$f;

$r.$f=frame[$c].$f;

frame[$c].$f=new Value();

frame[$c]=null;

LLJ Statements. Example.

- The stack-machine code for 7 + 5 in LLJ:

acc ← 7	a0.i = 7;
push acc	stack[sp++].i = a0.i;
acc ← 5	a0.i = 5;
acc ← acc + top_of_stack	a0.i = a0.i + stack[sp-1].i;
pop	sp--;

Formal definition of TYPEDLET: grammar

```
prog: expr;

expr: num
    | iden
    | '(' expr ')'
    | expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | 'let' iden ':' type '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr;

type: 'int'
    | 'boolean'
```

```
num: INT;
iden: ID;

ID  :  ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..
'Z'|'0'..'9'|'_')* ;
INT :  '0'..'9'+ ;
WS  :  (' '|' \r' | ' \t' | ' \n')+ -> skip ;
```

Code Generation Strategy

- For each expression e we generate LLJ code that:
  - Computes the value of e in a0
  - Preserves sp and the contents of the stack

- We define a code generation function cgen(e) whose result is the code generated for e

Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:

```
cgen(j) =
        a0.i = j;
```

- Note that this also preserves the stack, as required

Code Generation for Add

```
cgen(e1 + e2) =
        cgen(e1)
        stack[sp++].i = a0.i;
        cgen(e2)
        t1.i = stack[sp-1].i;
        a0.i = t1.i + a0.i;
        sp--;
```

- Code in blue is what we generate
- Possible optimization: Put the result of e_1 directly in t1 ?

Code Generation for Add. Wrong!

- Optimization: Put the result of e_1 directly in t1?

$$
\begin{aligned}
&\text{cgen}(e_1 + e_2) = \\
&\qquad \text{cgen}(e_1) \\
&\qquad \text{t1.i = a0.i;} \\
&\qquad \text{cgen}(e_2) \\
&\qquad \text{a0.i = t1.i + a0.i;}
\end{aligned}
$$

- Try to generate code for : 3 + (7 + 5)

Code Generation Notes

- The code for + is a template with "holes" for code for evaluating e_1 and e_2
- Stack-machine code generation is recursive
- Code for $e_1 + e_2$ consists of code for e_1 and e_2 glued together
- Code generation can be written as a recursive-descent of the AST
  - At least for expressions

Code Generation for Comparison

$cgen(e_1 > e_2) =$

```
        cgen(e1)
        stack[sp++].i = a0.i;
        cgen(e2)
        t1.i = stack[sp-1].i;
        a0.b = t1.i > a0.i;
        sp--;
```

Code Generation for if-then-else

```
cgen(if e1 then e2 else e3) =
        cgen(e1)
        if (a0.b) goto true_branch;
    false_branch:
        cgen(e3)
        goto end_if;
    true_branch:
        cgen(e2)
    end_if:
```

Code Generation for if-then-else

```
cgen(if e1 then e2 else e3) =
        cgen(e1)
        if (a0.b) goto true_branch;
    false_branch:
        cgen(e3)
        goto end_if;
    true_branch:
        cgen(e2)
    end_if:
```

But there is no goto in Java

Code Generation for if-then-else

```
cgen(if e1 then e2 else e3) =
        cgen(e1)
        if (a0.b) goto true_branch;
    false_branch:
        cgen(e3)
        goto end_if;
    true_branch:
        cgen(e2)
    end_if:
```

Use switch statement of Java

Code Generation for if-then-else

- Top-level while loop and a switch statement
- To implement goto, set label and break

```
int label = 0;
while(true) {
    switch(label) {
        case 0:
        // generated code goes here
        return;
    }
}
```

Use switch statement of Java

Code Generation for if-then-else

```
labelCounter = 1;
cgen(if e1 then e2 else e3) =
        true_label = labelCounter++;
        false_label = labelCouter++;
        end_label = labelCounter++;
        cgen(e1)
        if (a0.b) {label=true_label; break;}
   case false_label:
        cgen(e3)
        label = end_label; break;
   case true_label:
        cgen(e2)
   case end_label:
```

Use switch statement of Java

Code generation for let-in

- Use frame stack to create a binding for id

```
cgen(let id: int = e1 in e2) =
        cgen(e1)
        frame[fp++] = new Value();
        frame[fp-1].i = a0.i; // use .b if type of id is boolean
        cgen(e2)
        frame[--fp]=null;
```

Code generation for variable access

cgen(id) =

    a0.i = frame[??].i   // assume id is of type int

- frame has no reference to id
- Which index to use to get binding of id from frame?

Code generation for variable access

```
cgen(id) =
        a0.i = frame[??].i   // assume id is of type int
```

- frame has no reference to id
- Which index to use to get binding of id from frame?
  - Keep an environment during code generation
  - environment provides the index of the variable
  - Generated code has no environment

Code Generation with Index Environment

- Define cgen(e, E), where
  - E is a list if identifiers
- id::E is the list with id prepended to E
  - Example: "d"::["a", "b", "c", "a"] = ["d","a", "b", "c", "a"]
- E(id) is the index of the first occurrence of id in E
  - Example: ["d", "a", "b", "c", "a"]("a") = 1
  - Example: ["d", "a", "b", "c", "a"]("d") = 0
  - Example: ["d", "a", "b", "c", "a"]("c") = 3

Code generation for let-in

- Use frame stack to create a binding for id

```
cgen(let id: int = e1 in e2, E) =
        cgen(e1, E)
        frame[fp++] = new Value();
        frame[fp-1].i = a0.i; // use .b if type of id is boolean
        E' = id::E
        cgen(e2, E')
        frame[--fp]=null;
```

Code generation for variable access

```
cgen(id, E) =
        a0.i = frame[??].i   // assume id is of type int
```

- E now has reference to id

Code generation for variable access

```
cgen(id, E) =
        j = E(id)
        a0.i = frame[??].i   // assume id is of type int
```

- E now has reference to id

Code generation for variable access

```
cgen(id, E) =
        j = E(id)
        a0.i = frame[fp-1-j].i   // assume id is of type int
```

- E now has reference to id
- Generated code has no environment
  - no expensive lookup to find the binding of a variable

How to change cgen for remaining expressions?

- Simply add E to cgen

```
cgen(j, E) =
        a0.i = j;
```

How to change cgen for remaining expressions?

```
cgen(e1 + e2, E) =
        cgen(e1,E)
        stack[sp++].i = a0.i;
        cgen(e2,E)
        t1.i = stack[sp-1].i;
        a0.i = t1.i + a0.i;
        sp--;
```

Code generation

- Generated code is efficient
  - no dynamic type checking
  - no expensive lookup for variable bindings
  - no need to visit AST and check the kind of each AST node
- Next how to generated code for lambdas with closures

Formal definition of TYPEDREFS: grammar

```
prog: expr;
expr: num
    | iden
    | '(' expr ')'
    | expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | iden '=' expr
    | 'let' iden ':' type '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr
    | 'while' expr 'do' expr
    | 'letrec' iden '(' iden ':' type ')' ':' type =
expr in expr
    | '(' expr expr ')'
    | '{' exprseq '}'
exprseq: exprseq ';' expr
    | expr
```

```
type: 'int'
    | 'boolean'
    | 'nulltype'
    | '(' type ')' '->' type

num: int;
iden: ID;

ID : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..
'Z'|'0'..'9'|'_')* ;
int : '0'..'9'+ ;
WS : (' '|' \r' | ' \t' | ' \n')+ -> skip ;
```

Code generation for variable access

```
cgen(id, E) =
        j = E(id)
        a0.i = frame[fp-1-j].i   // assume id is of type int
```

- E now has reference to id
- Need to define cgen(id, E) for each type of id
- Generated code has no environment
  - no expensive lookup to find the binding of a variable

Code Generation for variable assignment

```
cgen(id = e, E) =
        cgen(e, E)
        j = E(id)
        frame[fp-1-j].i = a0.i; // assuming type of id is int
```

- Again generated code has no reference to the environment

Code generation for while...do...

```
cgen(while e1 do e2, E)
        begin_while = labelCounter++;
        end_while = labelCounter++;
        case begin_while:
                cgen(e1, E)
                if (!a0.b) {label = end_while; break;}
                cgen(e2, E)
                label = begin_while; break;
        case end_while:
```

- Remember we have no goto statements in LLJ

Code generation for {e1; ...; en}

```
cgen({e1; ...; en}, E) =
        cgen(e1, E)
        ...
        cgen(en, E)
```

Code generation for Lambdas

```
cgen( (e1 e2), E) =
        cgen(e1, E)
        stack[sp++]. l = a0.l;
        cgen(e2, E)
        t1.l = stack[sp-1].l;
        sp--;
        t1.l.apply();
```

- a0 contains the value of argument
- when the lambda's body is executed, we first need to assign a0 to the formal parameter of the lambda
- We need to define Lambda.apply (coming next)

Value class in LLJ: to denote values of any type

```
class Value {
   public int i;
   public boolean b;
   public Lambda l;
   public String s;
}
```

If *C/C++* is the target language, one can use *C*'s union.

Code generation for let-in

- Use frame stack to create a binding for id

```
cgen(let id: int = e1 in e2, E) =
        cgen(e1, E)
        frame[fp++] = new Value();
        frame[fp-1].i = a0.i; // use .b if type of id is boolean
        E' = id::E
        cgen(e2, E')
        frame[--fp]=null;
```

Code generation for variable access

```
cgen(id, E) =
        j = E(id)
        a0.i = frame[fp-1-j].i   // assume id is of type int
```

- E now has reference to id
- Need to define cgen(id, E) for each type of id
- Generated code has no environment
  - no expensive lookup to find the binding of a variable

Implementation of Lambdas

- Piggyback on Java's methods
  - most general-purpose languages provide the abstraction of functions
  - Java, C, C++, JavaScript, LLVM, Java bytecode
- Need a way to save the current closure
  - save frame and fp along with a lambda
  - note that elements of frame could get rewritten
  - while (c) do let x: int = e in letrec f(y: int): int = x + y in f(3)
  - Each time we create a lambda in the above, a different x gets captured, but each x occupy the same location in the frame
  - Solution: copy elements of frame to a closure stack and associate the closure with the lambda

Lambda class

```
abstract class Lambda {
   public Value[] closure;

   public Lambda(Value[] frame, int fp) {
      this.closure = new Value[fp];
      System.arraycopy(frame, 0, this.closure, 0, fp);
   }

   public abstract void apply();
}
```

- A new class inheriting Lambda is used to represent the code of each lambda
- closure array saves the frame at the time of the creation of the lambda
- class Lambda can be simulated using struct and functions pointers in C and LLVM

Code generation for letrec ... in ...

```
int funCounter = 1;

cgen(letrec id1(id2: int): int = e1 in e2, E) =
          funName = "Fun"+funCounter++;
          frame[fp++] = new Value();
          frame[fp-1].l = new funName (frame, fp);
          E' = id1::E;
          cgen(e2, E')
          frame[--fp] = null;
```

Code generation for letrec ... in ...

```
int funCounter = 1;

cgen(letrec id1(id2: int): int = e1 in e2, E) =
          funName = "Fun"+funCounter++;
          frame[fp++] = new Value();
          frame[fp-1].l = new funName (frame, fp);
          E' = id1::E;
          cgen(e2, E')
          frame[--fp] = null;
          createLambda(funName,
                    frame[fp++] =new Value();  // bind parameter to first variable
                    frame[fp-1].i = a0.i;
                    cgen(e1, id2::E')
                    frame[--fp] = null;
          );
```

Create a Lambda class

```
createLambda(funName, code) =
        public class funName extends Lambda {
                public funName (Value[] frame, int fp) {super(frame,fp);}
                public void apply() {
                        Value[] frame = new Value[FRAME_SIZE];
                        int fp = closure.length;
                        System.arraycopy(closure, 0, frame, 0, closure.length);
                        int label = 0;
                        while(true) {
                                switch(label) {
                                        case 0:
                                                code
                                        return;
                                }
                        }
                }
        }
```

How to determine FRAME_SIZE statically?

- Can be computed statically at compilation time
  - Traversal of the body of a lambda
  - Count the number of variables declared using let nd letrec
  - For PA5, we will set FRAME_SIZE to a reasonably large value
    - Feel free to compute FRAME_SIZE statically and use that in code generation

Code generation for let-in. Why not the following?

- Use frame stack to create a binding for id

```
cgen(let id: int = e1 in e2, E) =
        cgen(e1, E)
        frame[fp++].i = a0.i; // use .b if type of id is boolean
        E' = id::E
        cgen(e2, E')
        frame[--fp]=null;
```

An example that shows the issue if we do not use frame[fp++] = new Value()

```
letrec g(z) = 1
in { let x = 1 in letrec f(y) = x + 1 in g = f; let x = 2 in  x; (g 3)}
```

Two optimizations

- Do we need to copy entire frame in closure while creating a Lambda? no
  - only copy elements that are captured by the Lambda
- Do we need to create a Value object for each declaration of a variable using let...in... or letrec...in...?
  - Only create Value object for variables that are captured by the nested lambdas
- Real-world implementations perform these optimizations
- You can implement these optimizations in PA5, but not required

FreeVars and CapturedVars sets (will use short-forms F and C, respectively)

- f.F is the set of free variables of a lambda f, i.e. variables that has been read/written in the body of the f (and nested lambdas), whose declaration cannot be found in the body of f (and nested lambdas)
- f.C is the set of captured variables of a lambda f: variables that are free in lambdas nested in f, but whose declaration could be found in the body of f

Examples of FreeVars (i.e. F) and CapturedVars (i.e. C) of a lambda

```
let x = 1 in
        letrec f(y) =
                let z = 3 in
                        letrec g(w) = x + z + w in g
        in
                f(x)(4)
```

f.F= {x}
f.C = {z}
g.F = {x, z}
g.C = {}

FreeVars and CapturedVars sets (will use short-forms F and C, respectively)

- f.F is the set of free variables of a lambda f, i.e. variables that has been read/written in the body of the f (and nested lambdas), whose declaration cannot be found in the body of f (and nested lambdas)
- f.C is the set of captured variables of a lambda f: variables that are free in lambdas nested in f, but whose declaration could be found in the body of f
- While creating a lambda, only copy values of variables in F to closure
- While declaring a local variable, create a new Value when accessing a variable in C only
- Vars(e, E, F, C):
  - associates sets F and C with each lambda expression
  - define recursively for each expression type

Further Optimization Opportunities

- Advantage: Very simple code generation

- Disadvantage: Slow code
  - Storing/loading temporaries requires a store/load from the stack and sp adjustment

A Better Way

- Idea: Determine index of temporaries statically

- The code generator must assign a location in the stack for each temporary

Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- make stack a local variable to each Lambda
  - similar to frame
- Add a new argument to code generation: the position of the next available temporary

cgen(e,n,E) : generate code for e and use temporaries whose index is n or higher

Code Generation for Add (original)

```
cgen(e1 + e2, E) =
        cgen(e1,E)
        stack[sp++].i = a0.i;
        cgen(e2,E)
        t1.i = stack[sp-1].i;
        a0.i = t1.i + a0.i;
        sp--;
```

Code Generation for Add

```
cgen(e1 + e2, n, E) =
        cgen(e1,n,E)
        stack[n].i = a0.i;
        cgen(e2,n+1,E)
        t1.i = stack[n].i;
        a0.i = t1.i + a0.i;
```

Notes

- The temporary area is used like a small, fixed-size stack

- Exercise: Write out cgen for other constructs

Optimize access to frame

- Similarly one can get rid of fp and compute index in frame statically
- One could also combine frame and stack in a single array
  - size of both can determined statically