Assignment 3

Solutions

Compiler Design I (Kompilatorteknik I) 2011

DISCLAIMER: In the code listings of the following exercises we use a syntax that closely resembles C. However the C language allows neither nested function definitions (which appear in Exercise 2) nor true by-reference parameter passing (which some parts of Exercise 1 assume).

1 Parameter passing

Suppose we have the following program:

```
1   int foo(int a) {
2     int b = a++;
3     return b * a
4   }
5
6   int bar(int b) {
7     return foo(b);
8   }
9
10  int main() {
11    int c;
12    c = 6;
13    return bar(c);
14  }
```

Assume that the compiler allocates all the variables in the stack.

The single arguments of `foo` and `bar` can be passed either *by-value* or *by-reference*. **For each of the four possible combinations** (i.e. both arguments *by-value*, `foo`'s argument *by-value* & `bar`'s argument *by-reference*, etc.) describe:

1. What kind of data should the compiler put in the argument slots of the activation records for the calls on lines 7 and 13?
2. What kind of assembly code will be necessary to retrieve the value of variable `a` on line 2?

Answer

The aim of this exercise is to show how compilers handle calling-by-value and calling-by-reference.

First of all it is important to understand that the code for the called function is generated once and should be able to retrieve the value of the passed argument regardless of what has happened before. Other calls to each function might also be present and should work with the same code.

Therefore, if the argument is passed by value, it is always the value that should be in the activation record. Similarly, if the argument is passed by reference, its value should be retrievable with a single "dereference". [1]

With the previous in mind, to retrieve the value (question 1.2):

- If `foo` uses call-by-value:

  ```
  lw $a0, OFFSET($FP)
  ```

- If `foo` uses call-by-reference:

  ```
  lw $t0, OFFSET($FP)
  lw $a0, 0($t0)
  ```

(Here `OFFSET` is the specific offset for the slot of the argument.)

With these in mind, for question 1.1, the contents of the argument slots of the activation records will be:

- `foo` uses call-by-value & `bar` uses call-by-value:

  Both slots will contain actual values (i.e. the numeric representation of 6).

- `foo` uses call-by-value & `bar` uses call-by-reference:

  The slot for the call on line 13 contains the address of the variable `c`, whereas the slot for the call on line 7 contains the actual value retrieved from that address before the call (i.e 6).

- `foo` uses call-by-reference & `bar` uses call-by-value:

  The slot for the call on line 13 contains the value of the variable `c` at the time of the call (i.e. 6), whereas the slot for the call on line 7 contains the address of the previous slot.

- `foo` uses call-by-reference & `bar` uses call-by-reference:

  The slot for the call on line 13 contains the address of the variable `c`, and the same address is copied in the slot for the call on line 7. The compiler knows that the function received an argument by reference and has to pass it on by reference, therefore it copies the original reference.

[1] If we allow the first argument to be passed directly in a register (e.g. `$a0`), then the value or reference is what is stored in the register. The code in this case is:

- If `foo` uses call-by-value:
  No code needed, the value is already in the register.

- If `foo` uses call-by-reference:

  ```
  lw $a0, 0($a0)
  ```

2 Activation records and scoping

(This is not the exercise from 2011, but the one solved in the lesson instead.)

In the following listing we show an excerpt of a program in a language that allows nested definition of functions, is *statically scoped* and uses *calls-by-value*:

```
1 int spam(int a, int b) {
2  int egg(int x) {
3    return x*a mod b;
4  }
5  return egg(3);
6 }
```

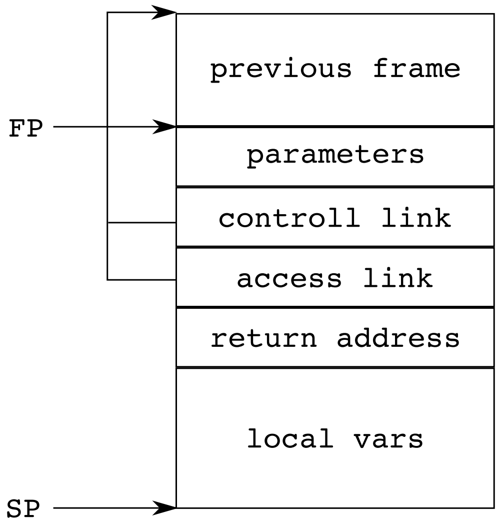
Suppose we have in our `main()` function a call to `spam(7,13);`. Assume that the compiler allocates all the variables (named and temporary) on the stack.

1. What will be the return value of this call?
2. Give a diagram of the state of the stack right before the return of the call to function `egg` on line 3. Your diagram should contain at least the following information regarding the calls to `spam` and `egg` functions (including the initial call):
   (a) boundaries of the activation records
   (b) location of the actual parameters for the calls to these functions
   (c) location of the return values
   (d) location of the return addresses
   (e) location and contents of the control links and access links
   (f) *Optional:* Location of temporary variables
   (g) *Optional:* Contents of actual parameters
3. Show where the values of the following variables are located in the stack and describe the code that should be generated by the compiler to access these values in the specified locations (including any points of reference, like the stack pointer). You may use as many registers as you like to calculate any addresses that are needed:
   (a) Variable `x` on line 3
   (b) Variable `a` on line 3

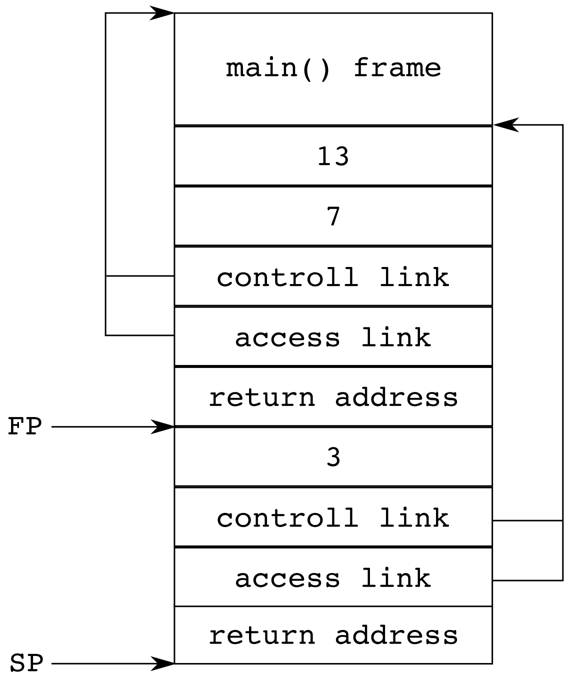
Answers

What will be the return value of this call? $spam(7, 13) = 3 * 7 \ mod \ 13 = 8$

Stack The general idea for a stack layout that supports nested function definitions would look like the following diagram:



Following this idea, the stack right before the return of the `call` to egg looks like the following diagram:



Code x is retrieved directly by accessing the function's parameter inside the function frame:

```
lw $t0, -4($fp)
```

a is retireved using the access link in the function's frame, as it is defined in a higher scope.

```
lw $t0, -c($fp)
lw $t0, -8($t0)
```

This code assumes that values have the size of a word.

3 Code generation

Suppose that we want to generate code for the expression:

```
cond
   <p1> => <e1>;
   <p2> => <e2>;
   ...
   <pn> => <en>;
      1 => <e{n+1}>
dnoc
```

The evaluation of a `cond` expression begins with the evaluation of the predicate `<p1>` (if it exists, n can also be zero). If `<p1>` evaluates to a non-zero value, then `<e1>` is evaluated, and the evaluation of the `cond` expression is complete. If `<p1>` evaluates to zero, then `<p2>` is evaluated, and this process is repeated until one of the predicates evaluates to a non-zero value. The value of the cond expression is the value of the expression `<ei>` corresponding to the first predicate `<pi>` that evaluates to a non-zero value. If all the predicates evaluate to zero, then the value of the `cond` expression is `e{n+1}`.

Write a code generation function: `cgen(cond <p1> => <e1>; ...; <pn> => <en>; 1 => <e{n+1}> dnoc)` for this conditional expression.

Answers

```
cgen(cond <pi> => <ei>; <p{i+1}> => e{i+1}; ... <p2> => e2; <p2> => e2;) {
 cgen(<p1>); // assumes that the result is stored in a0
 emit("li $t0 0");
 emit("beq $a0 %t0 next");
 cgen(<e1>);
 emit("b exiti");
 emit("nexti:");
 cgen(cond <p{i+1}> => <e{i+1}>; ... <pn> => <en>; 1 => e{n+1};);
 emit("exiti:");
}
```

4 Local optimizations

Consider the following basic block, in which all variables are integers and ** denotes exponentiation:

```
a := b + c
z := a ** 2
x := 0 * b
y := b + c
w := y * y
u := x + 3
v := u + w
```

Assume that the only variables that are live at the exit of this block are v and z. In order, apply the following optimizations to this basic block. Show the result of each transformation.

1. algebraic simplification
2. common sub-expression elimination
3. copy propagation
4. constant folding
5. dead code elimination

When you have completed part 5, the resulting program will still not be optimal. What optimizations, in what order, can you apply to optimize the result of 5 further?

Answer

1) Algebraic optimization:

```
a := b + c
z := a * a
x := 0
y := b + c
w := y * y
u := x + 3
v := u + w
```

2) Common sub-expression elimination:

```
a := b + c
z := a * a
x := 0
y := a
w := y * y
u := x + 3
v := u + w
```

3) Copy propagation:

```
a := b + c
z := a * a
x := 0
y := a
w := a * a
u := 0 + 3
v := u + w
```

4) Constant folding:

```
a := b + c
z := a * a
x := 0
y := a
w := a * a
u := 3
v := u + w
```

5) Dead code elimination:

```
a := b + c
z := a * a


w := a * a
u := 3
v := u + w
```

Doing common sub-expression elimination, copy propagation, constant propagation and dead code elimination one more time, we get the final minimal form:

```
a := b + c
z := a * a
v := 3 + z
```

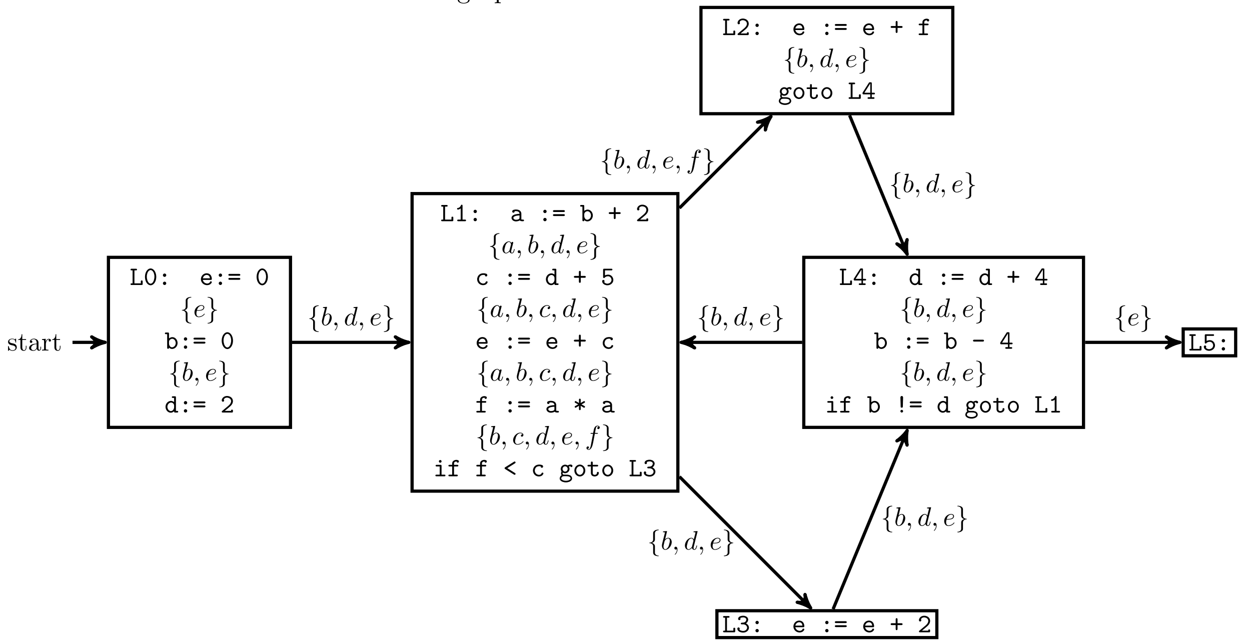
5 Register allocation

Consider the following program.

```
L0: e := 0
    b := 1
    d := 2
L1: a := b + 2
    c := d + 5
    e := e + c
    f := a * a
    if f < c goto L3
L2: e := e + f
    goto L4
L3: e := e + 2
L4: d := d + 4
    b := b - 4
    if b != d goto L1
L5:
```

This program uses six temporaries, `a-f`. Assume that the only variable that is live on exit from this program is `e`. Draw the register interference graph. (Drawing a control-flow graph and computing the sets of live variables at every program point may be helpful.)

Answers First we draw the control flow graph:



The register interference graph is therefore: