yz709-compiler-sup2

Question 1

What is the difference between a register machine and a stack machine as used in programming language run-times? In your discussion include at least, code density, performance, memory allocation, and implementation complexity.

	Register machine vs Stack machine
Code density	- The register machine requires fewer instructions to execute the same program - But the length of a single instruction in a register machine is longer than a stack machine because we need to specify the operand addresses (i.e., specific registers which store the values)
Performance	- The register machine can optimise compiled code much better because it is closer to the actual hardware implementation - The register machine can store a frequently used value in one register to boost execution performance, which a stack machine is not able to achieve - The stack machine has an overhead of pushing and popping values
Memory allocation	- For a register machine, the values are read from and allocated to specified registers, so we have a limitation on the number of registers - For a stack machine, we only have a temporary accumulator and a much larger stack space where we can store values, so we would have less concern about memory allocation
Implementation complexity	- The register machine is much more complex because we need to deal with register allocations - The stack machine is easier to implement because there are only two main operations: push and pop, and several computation operations

💡 Comments:

Question 2

How is the environment implemented in Slang interpreters 0 and 2? Pay special attention to the handling of functions.

- Similarity:
    - A store includes mappings from integer locations to values stored in those locations
    - The `value` data types for both are the same, except the function data type
    - The environment in interpreter 0 is a mapping from variables to values, but in interpreter 2, the environment is a list containing variables and their values

    ```
    type address = int
    type store = address -> value
    and value =
      | REF of addresses
      | INT of int
      | BOOL of bool
      | UNIT
      | PAIR of value * value
      | INL of value
      | INR of value
      ...
    (* interpreter 0 *)
    env = var -> value
    (* interpreter 2 *)
    env = (var * value) list
    ```

- Differences:
    - The function data type in interpreter 0 is a mapping which shows the changes on the value-store pair; however, in interpreter 2, we used a closure to represent a function, which is a pair of two lists: `code` (an instruction stack) and an `env` (a list storing variables and their values). A closure includes the function body and the environment to interpret free variables in that function.

        ```
        (* interpreter 0 *)
        | FUN of ((value * store) -> (value * store))
        (* interpreter 2 *)
        | CLOSURE of bool * closure
        (* for reference,
        code = instruction list
        closure = code * env,
        env = (var * value) list *)
        ```

💡 Comments:

Question 3

What is the main difference between interpreter 0 and interpreter 2? What are the purposes of this change? What are the additions to interpreter 2 in version 3 and how do they improve the compiler?

- In interpreter 0, we would use the implicit OCaml stack when evaluating recursive functions. So in order to prevent using the OCaml stack, we transformed it into a tail-recursive compile function and an explicit stack machine in interpreter 2, in that way, all recursive functions would carry their own stack explicitly.
- The main reason is that the high-level interpreter 0 is inefficient because it needs to check the types before carrying out the operations each time by inspecting the AST. It would greatly boost the efficiency if we do the type checking once, convert the AST into an intermediate code program, then focus on performing the operations.
- However, some complex recursive functions cannot be transformed into tail-recursive functions easily, so we first used continuation-passing-style transformation (CPS) to transform them into tail-recursive functions with an extra argument representing the rest of the computation. Then we used defunctionalisation (DFC) (in version 3) to replace the higher-order functions with data structures. We could later convert continuations into a list that can be used as an explicit stack.

💡 Comments:

Question 4

As you know Java virtual machine is a bytecode interpreter. Here are the instructions of a function `f`. What is the high-level code that produces these instructions?

Annotate fragments of the code. This will require you to look up what these instructions do online. Please do not use a decompiler.

- The high-level code will return 1 when $x = 0$ or $x = 1$, but will fall into an infinite loop for all other values

  ```
  def f(x):
    if (x == 0):
      return 1
    else:
      y = f(1-x)
      return x * y
  ```

```
public static int f(int);
Code:
   0: iload_0             // push 0th argument onto the stack
   1: ifne          6     // pop stack_top, if stack_top != 0 then goto L6
   4: iconst_1            // else push 1 onto the stack
   5: ireturn             // return to the function invoker
   6: iload_0             // push 0th argument onto the stack
   7: iload_0             // push 0th argument onto the stack
   8: iconst_1            // push 1 onto the stack
   9: isub                // pop two values (v1 & v2),
                          // push result v1 - v2 onto the stack
  10: invokestatic  #4    // invoke method f
  13: imul                // pop two values (v1 & v2)
                          // push result v1 * v2 onto the stack
  14: ireturn             // return to the function invoker
```

💡 Comments:

Question 5(y2011p3q5)

Consider a simple grammar for arithmetic expressions:

$$E ::= n \mid x \mid -E \mid E + E \mid (E)$$

with n ranging over integer constants and x ranging over variables. We want to compile expressions to code for a simple stack-based machine with the following instruction set.

instruction	meaning
pushvar k	push the value of the k-th variable on top of stack
push n	push n on top of stack
add	replace the top two stack items with their sum
neg	replace the top stack item with its negation

For this problem, we will not worry about how variables are bound to values nor how abstract syntax trees are produced.

(a) How will your compiler generate code from expressions of the form $-E$? [4 marks]

- Generate code for E, say $complie(E)$, then append the instruction `neg` at the end, i.e., $compile(E)@[neg]$

(b) How will your compiler generate code from expressions of the form $E_1 + E_2$? [4 marks]

- Generate code for E_1, say $compile(E_1)$, then generate code for E_2, say $compile(E_2)$, then append the instruction `add` at the end, i.e., $compile(E_1)@compile(E_2)@[add]$

(c) What code will your compiler generate for the expression `-(-x + (17 + y))`? [4 marks]

```
pushvar x
neg
push 17
pushvar y
add
add
neg
```

(d) Suppose we now want to extend the language of integer expressions with multiplication

$$E ::= \cdots \mid E \; * \; E$$

but we cannot extend the machine with an instruction for multiplication.

Can you implement this extended language directly with the machine instruction set presented above? If not, suggest a *minimal* extension to the instruction set that allows for the implementation of multiplication using the addition from the instruction set. Explain the semantics of your extensions and how you would use them to implement multiplication. [8 marks]

- If our target is $E_1 * E_2$, we first generate code for E_1, say $compile(E_1)$, then generate code for E_2, say $compile(E_2)$
- Use a for loop and an accumulator to iteratively adding $compile(E_2)$ for $compile(E_1)$ times.
- Define a `TEST` instruction to test whether the $compile(E_1)$ is 0, if so, output the accumulator, otherwise decrement $compile(E_1)$ and add $compile(E_2)$ to the accumulator.
- However, we need to add additional instructions at the beginning to test the two values (i.e., $compile(E_1)$ and $compile(E_2)$). If one of them is negative, we should negate the negative one and add a `neg` instruction for the final result; If both are negative, we should negate both and do the above procedure, no change for the final result.

Comments:

Question 6(y2014p3q4)

This question concerns the run-time call stack.

(a) What is a *run-time stack* and why is it important to a compiler writer? [3 marks]

- A run-time stack contains multiple frames, each is associated with an invocation of one function, so each frame contains the function's arguments, location variables and the return address.

- Because for a stack-oriented compiler, all instructions should be executed using the basic operations based on the stack. The static source program text should be translated into dynamic actions at run-time, so the compiler writer needs to know how to translate between them.

(b) The implementation of a run-time call stack typically uses a *stack pointer* and a *frame pointer*. What are their roles and why do we need two pointers? [3 marks]

- A frame pointer contains the base address of the function's frame, the variables can be accessed by an offset from the frame pointer. The frame pointer always points to the current activation record (stack frame), it will not change during the execution of a function.
- The stack pointer always points to the top of the stack, and it may change during the execution as values will be pushed or popped off the stack (e.g., pushing parameters to call another function)
- We need both because the new function invocation needs to use the stack pointer to push new parameters and form new stack frames, the frame pointer is used to execute the function call.

(c) For some compilers the activation records (stack frames) contain *static links*. What problem are static links used to solve and how do they solve this problem? [3 marks]

- e.g., suppose we have a function f, it nested f_1 which nested f_2, which further nested f_3, then f_1, f_2, and f_3 can all access the local variables of f, but they cannot find them directly, the static links are designed to help find those variables.
- When nesting first-order functions, some argument (free variable) values are out-of-scope for one function but in a parent scope and are stored on the stack, the static links are designed to find these argument values, they are added to every stack frame and point to the stack frame of the parent function. The parent function should store and provide all the argument values.

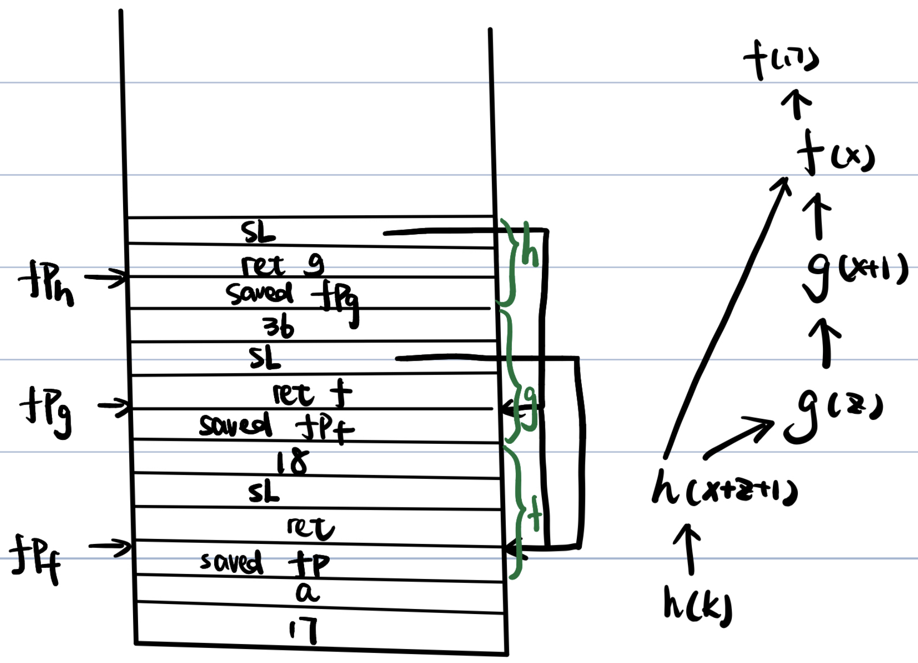
(*d*) (*i*) Consider a programming language that does not allow functions to be returned as results, but does allow the nesting of function declarations. Using ML-like syntax, we have the following code in this language.

```
let fun f(x) =
    let
        fun h(k) = k * x

        fun g(z) = h(x + z + 1)
    in
       g(x + 1)
    end
in
    f(17)
end
```

Draw a diagram illustrating the call stack from the call of `f` up to and including the call of function `h`. Make sure all function arguments are included in the diagram and clearly indicate static links. [5 marks]

- We have static links at the top inside each frame, followed by the return address, the saved frame pointer, arguments, and other free variables.



(*ii*) Using your diagram, explain how the code generated from the body of function `h` can access the values associated with the variables `k` and `x`. In each case make it clear what information is known at compile-time and what information is computed at run-time. [6 marks]

- k is a formal variable that can be found at stack location $fp_h - 2$ (assuming we have a heap pointer a at $fp_h - 1$), since fp_h cannot be known at run-time, we can find the information of k at run-time.
- x is a variable with its closest definition in f, it follows the static link of h to the frame f and gets the value of argument x, which we could find at location $fp_f - 2$ at run-time (assuming we have a heap pointer a at $fp_h - 1$).

💡 Comments: