Question 1: Theoretical Questions

Q1.1 Why are special forms required in programming languages? Why can’t we simply define them as primitive operators? Give an example

Special forms are required in case the semantics of the evaluation does not follow the default ‘procedure application’ semantics, i.e., evaluation of the operator and the operands + application. The ‘if’ special form, for example, does not required pre-evaluation of the ‘then’ and the ‘else’ sub- expressions. In case ‘if’ was a primitive operator rather than a special form, the evaluation of the following expression would cause an error: (if (> 3 4) 5 (\ 6 0))

Q1.2 Let us define the L0 language as L1 excluding the special form ‘define’. Is there a program in L1 which cannot be transformed to an equivalent program in L0? Explain or give a contradictory example

No, Since there are no recursive forms in L1 (e.g., user procedures), any var reference can be replaced by its defined value expression.

Q1.3 Let us define the L20 language as L2 excluding the special form ‘define’. Is there a program in L2 which cannot be transformed to an equivalent program in L20? Explain or give a contradictory example

Yes. L2 supports user procedures which may contain recursion calls (a variable which is defined as a lambda may appear in the body of the lambda). In this case, var references cannot be replaced by their defined value.

Q1.4 In practical session 5, we deal with two representations of primitive operations: PrimOp and Closure. List an advantage for each of the two methods

*Advantages of PrimOp:

Efficiency: PrimOp represents primitive operations as machine-level instructions, which can be executed efficiently by the computer's processor. This makes PrimOp a good choice for implementing frequently used operations, such as arithmetic operations, on large data sets.

*Advantages of Closure:

Flexibility: Closure represents operations as functions, which can be composed and combined in flexible ways. This makes Closure a good choice for implementing complex algorithms, where the behavior of the program is not easily expressed as a sequence of machine-level instructions.

Q1.5 For the following high-order functions in L3, which gets a function and a list as parameters, indicate (and explain) whether the order of the procedure application on the list items should be sequential or can be applied in parallel:

L3 is pure functional, i.e., there are no side-effects (beside the special define form, which is not CExp)

• map

Can be applied in parallel. The evaluation of the function application on one item has no effect on the evaluation application of the function on another item.

• Reduce

The reducer application is based on the reduce of the previous items. Can be applied in parallel just in case the reducer procedure is commutative and associative (e.g., +)

• Filter

Can be applied in parallel. The evaluation of the predicate application on one item has no effect on the evaluation application of the function on another item.

• all (returns #t is the application of the given boolean function on each of the given list items returns #t)

Can be applied in parallel. The evaluation of the predicate application on one item has no effect on the evaluation application of the function on another item. In addition, and operator is commutative and associative.

• compose (compose a given procedure with a given list of procedures)

Can be applied in parallel just in case the procedure composition is associative.

Q1.6 What is lexical address? Give an example which demonstrates this concept [2 points]

Lexical addressing is a method of determining the memory address of a variable based on its position relative to the current execution context. In other words, the address of a variable is determined by its position in the program's source code.

Q1.7 Let us define L31 as the L3 language with the addition of ‘cond’ special form (as described in practical session 4)

```
(cond <cond-clauses>*) / cond-exp(cond-clauses:List(cond-clause))
<cond-clause> ::= (<cexp> <cexp>+)  // cond-clause(test:cexp, then:List(cexp))
```