1)

1) [whileC (c : Expr) (e : Expr)]

2) [(s-exp-match? `{while ANY ANY} s)
 (let ([sl (s-exp->list s)])
 (whileC (parse (second sl)) (parse (third sl))))]

3) [(whileC c e) (while c e env sto)]
 (define (while c e env sto)
 (with [(v-cond sto-cond) (interp c env sto)]

 (if (equal? v-cond (numV 0))
 (v*s (numV 0) sto-cond)
 (with [(v-body sto-body) (interp e env sto-cond)]
 (while c e env sto-body)))]))

4) a) Les variables libres sont cnd, if-true et if-false. (Surement celle qui ne sont pas definie par des let)

 b) L'expression est équivalente à {if cnd if-true if-false}.

 c) Il suffit de remplacer l'expression {if cnd if-true if-false} par le code précédent en s'assurant que les variables res, v-cnd et not-cnd sont fraîches et n'apparaissent pas dans cnd, if-true ou if-false.

5) (if e1
 (if e2
 #t
 #f)
 #f)

6) (define (andn e)
 (and (first e) (andn (rest e))))

7) On retire le sucre syntaxique par couches successives. On commence par remplacer un and par une suite de and binaires. On remplace ensuite chaque and binaire par un if.

2)

8) (define (extend-env b env) (begin (hash-set! (first env) (bind-name b) (bind-location b)) env))

9) (define (lookup [n : symbol] [env : Env]) : Location
 (cond

```
[(empty? env) (error ' lookup " free identifier ")]
[(! (equal? none (hash-ref (first env) n)) (hash-ref (first env) n))]
[ else (lookup n (rest env))]))
```

10) (extend-env (bind par l) (cons (make-hash empty) c-env))
C'est une formule générale qu'on devrai ajouter à chaque cas dans le interp.
On rajoute une case au env qu'on set a (bind par l).

11) (define (get-global-dic [env : Env]) : (hashof symbol Location) (if (empty? (rest env)) (first env) (get-global-dic (rest env))))

12) a) Lors d'un appel par référence sur une variable, on ne créé pas une nouvelle variable pour le paramètre. Celui-ci constitue un alias pour l'argument.

b)
```
(define (interp [e : ExprC] [env : Env] [sto : Store ]) : Result
    (type-case ExprC e
        ...
        [ globalC (id body)
            (let ([ g-dic (get-global-dic env)])
                (cond
                 (hash-ref g-dic id)
                    [ (equal? none (hash-ref g-dic id)) (error '
                    lookup " free identifier ")]
                    [ else (interp
                                body
                                (extend-env (bind id loc) env)
                                sto)]))]
```

13) a) En portée lexicale, il est possible de déterminer à la lecture du code si une variable est définie ou pas. En portée dynamique, la définition des variables dépend du flot d'exécution.

b)La modification de l'environnement fait passer l'interpréteur en portée dynamique. Si on considère le code
{ let {[f { lambda {x} { begin { if x { let {[i 42]} 0} 0} i} }]} ... }
Alors {f 0} produit une erreur "free identifier" alors que {f 1} est correct.
L'expression global est similaire à l'expression let. Elle ne change pas elle-même le type de portée de l'interpréteur.