"Change-making" Case Study
M.J. Clancy and M.C. Linn

Problem

Abelson and Sussman, in the textbook *Structure and Interpretation of Computer Programs*, describe a program that counts the number of ways to make a given amount of change, *assuming that an unlimited number of each coin are available*. Determine how the program works, and modify the program so that it counts the number of ways to make a given amount of change using *a limited number of each coin*.

The Abelson and Sussman program

Here is Abelson and Sussman's description of the program.

> This problem has a simple solution as a recursive procedure. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:
>
> Number of ways to change amount *a* using *n* kinds of coins =
>
> Number of ways to change amount *a* using all but the first kind of coin
>
> \+ Number of ways to change amount *a–d* using all *n* kinds of coins, where *d* is the denomination of the first kind of coin.
>
> To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.
>
> Thus we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully, and convince yourself that we can use it to describe an algorithm if we specify the following degenerate cases:
>
> - If *a* is exactly 0, we should count that as 1 way to make change.
> - If *a* is less than 0, we should count that as 0 ways to make change.
> - If *n* is 0, we should count that as 0 ways to make change.

We can easily translate this description into a recursive procedure:

```
(define (count-change amount)
   (cc amount 5))

(define (cc amount kinds-of-coins)
   (cond
      ((= amount 0) 1)
      ((or (< amount 0) (= kinds-of-coins 0)) 0)
      (else (+ (cc (- amount
                      (first-denomination kinds-of-coins))
                   kinds-of-coins)
               (cc amount (- kinds-of-coins 1)) )) ))
```

```
(define (first-denomination kinds-of-coins)
   (cond
      ((= kinds-of-coins 1) 1)
      ((= kinds-of-coins 2) 5)
      ((= kinds-of-coins 3) 10)
      ((= kinds-of-coins 4) 25)
      ((= kinds-of-coins 5) 50) ))
```

(The first-denomination procedure takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we are thinking of the coins as arranged in order from smallest to largest, but any order would do as well.)

Two examples

The Abelson and Sussman program counts the ways to make a given amount of change using an unlimited number of each coin. Thus, there are thirteen ways to make 25 cents using an unlimited number of quarters, dimes, nickels, and pennies:

0 pennies + 0 nickels + 0 dimes + 1 quarter
0 pennies + 1 nickel + 2 dimes + 0 quarters
0 pennies + 3 nickels + 1 dime + 0 quarters
0 pennies + 5 nickels + 0 dimes + 0 quarters
5 pennies + 0 nickels + 2 dimes + 0 quarters
5 pennies + 2 nickels + 1 dime + 0 quarters
5 pennies + 4 nickels + 0 dimes + 0 quarters
10 pennies + 1 nickel + 1 dime + 0 quarters
10 pennies + 3 nickels + 0 dimes + 0 quarters
15 pennies + 0 nickels + 1 dime + 0 quarters
15 pennies + 2 nickels + 0 dimes + 0 quarters
20 pennies + 1 nickel + 0 dimes + 0 quarters
25 pennies + 0 nickels + 0 dimes + 0 quarters

The modified program will count the number of ways to to make a given amount of change using a limited number of each coin. Thus, there are four ways to make change for 25 cents using at most 5 pennies, 3 nickels, and 2 dimes:

0 pennies + 1 nickel + 2 dimes
0 pennies + 3 nickels + 1 dime
5 pennies + 0 nickels + 2 dimes
5 pennies + 2 nickels + 1 dime

Study questions

1. How many ways are there to make 35 cents in change using an unlimited number of coins?
2. How many ways are there to make 35 cents in change using at most 2 quarters, 2 dimes, 8 nickels, and 5 pennies?
3. Why are there no ways to make 25 cents in change using no dimes, at most 5 pennies, and at most 3 nickels?

4. Why are there no ways to make 25 cents in change using dimes, nickels, and exactly 1, 2, 3, or 4 pennies?
5. Invent a situation where someone would want to know the information this program generates.
6. Describe a problem that is similar to this problem and might be solved in a parallel way.

Understanding the unlimited-coins program

What is a good way to start trying to understand the program?

To modify the unlimited-coins program, we have to understand how it works. One way is to trace through it, trying to infer what all the parts are doing. Another way is to try to solve the problem ourselves, then to compare our solution with the program. Another approach is to try to understand the algorithm in the problem description and then see how the code represents the algorithm.

1. (stop and think) Why not try to just write a new program instead of modifying an existing one?

What are the key differences between the limited-coins and the unlimited-coins solutions?

Just reading the descriptions of the two versions of the problem suggests that limiting the number of coins shouldn't be too large a modification. The program should just stop counting the options for a given coin when it reaches some limit for the number of that coin. The program needs to know how many of each coin it has and then, in the general case, to do the same thing it did when the number was unlimited. Having decided that, we now turn to the Abelson and Sussman code.

How do programmers figure out how some unfamiliar code works?

To understand a program it helps to understand the problem the program is solving. To understand this program, we try solving a simple example by hand and then trace the solution in the code. Our hope is that an idea of how we would solve the problem by hand will provide some structure for understanding the code.

Consider the problem of generating all the ways to make 11 cents using an unlimited number of nickels and pennies. (Here we have simplified the number of kinds of coins being used.)

2. (stop and think) One approach to generating the ways of making change is random guessing, i.e. writing down each way we think of until we can't think of any more. Why is this a bad idea?

3. (stop and think) Describe an algorithm that generates all the possibilities for making eleven cents using unlimited nickels and pennies.

4. (stop and think) Will your algorithm work for any combination of coins?

We write down all the possibilities in a table:

possibilities for nickels	*possibilities for pennies*
0 nickels	11 pennies
1 nickel	6 pennies
2 nickels	1 penny

5. (stop and predict) Create the table of ways to make 20 cents using dimes, nickels, and pennies.

Suppose we now include dimes in the coins to use, and make change for 20 cents to give more possibilities. A table for this situation is the following:

possibilities for dimes	*possibilities for nickels*	*possibilities for pennies*
0 dimes	0 nickels	20 pennies
	1 nickel	15 pennies
	2 nickels	10 pennies
	3 nickels	5 pennies
	4 nickels	0 pennies
1 dime	0 nickels	10 pennies
	1 nickel	5 pennies
	2 nickels	0 pennies
2 dimes	0 nickels	0 pennies

6. (application) Solve some other examples, e.g., generate the ways to make 72 cents using an unlimited number of dimes, nickels, and pennies.

7. (analysis) Consider the ways of making 50 cents, using dimes, nickels, and pennies. Without generating the table, determine a reasonable value for the number of rows it will have.

8. (reflection) Could the number of ways to make 30 cents with dimes, nickels, and pennies be found by making a straightforward computation instead of by generating a table?

9. (reflection) Does the problem ask for the number of ways or the table of ways? Why is this important?

What algorithm does the table suggest?

The process for creating the table leads to the following procedure for counting ways to make a given amount *amt* using dimes, nickels, and pennies:

Determine the maximum number of dimes that can be used, and write down possibilities in the first column of the table that indicate the use of 0 dimes, 1 dime, and so on up to that maximum.
For the "0 dimes" possibility, write down all the ways to make change for *amt* cents using only nickels and pennies.
For the "1 dime" possibility, write down all the ways to make change for *amt*–10 cents using only nickels and pennies.
For the "2 dimes" possibility, write down all the ways to make change for *amt*–20 cents using only nickels and pennies.
…
For the possibility corresponding to the maximum number of dimes—call it *m*—write down all the ways to make change for

amt–10*m* cents using only nickels and pennies.

Let's try this with 30 cents. At most three dimes can be used, so there are four possibilities for dimes. We write them down:

3 dimes + 0 more cents
2 dimes + 10 more cents
1 dime + 20 more cents
0 dimes + 30 more cents

For each possibility, we then enumerate the ways of making the remaining amount with nickels and pennies.

This algorithm can be stated in terms of any collection of coins, and any amount *amt*:

Write down all possibilities for the first coin.
The total number of ways to make *amt* is equal to the sum of the number of ways for each of the possibilities, using the coins that remain.

This is called a *recurrence relation*, since it expresses the number of ways to make change in terms of itself ("recurrence" has the same etymology as "recursion"). Note that the two expressions on the right hand side of the equation are *smaller* versions of the problem; one uses fewer coins, and the other uses a smaller amount. Any recurrence must have base cases. The base cases here represent an amount as small as possible, and a number of coins as small as possible.

10. (reflection) How are recurrence relations related to recursive procedures?

How can this relation be simplified?

Re-examining this algorithm suggests a way to express it more concisely. What it's doing is computing

the number of ways using 0 of the first coin
+ the number of ways using 1 of the first coin
+ …
+ the number of ways using as many as possible of the first coin

But, just as a simple sum of the integers from 0 to *n* can be expressed recursively as 0 + the sum of the integers from 1 to *n*, so also can the sum of the ways of making change be expressed recursively:

the number of ways using 0 of the first coin
+ the number of ways using *1 or more* of the first coin

More formally, we have the following:

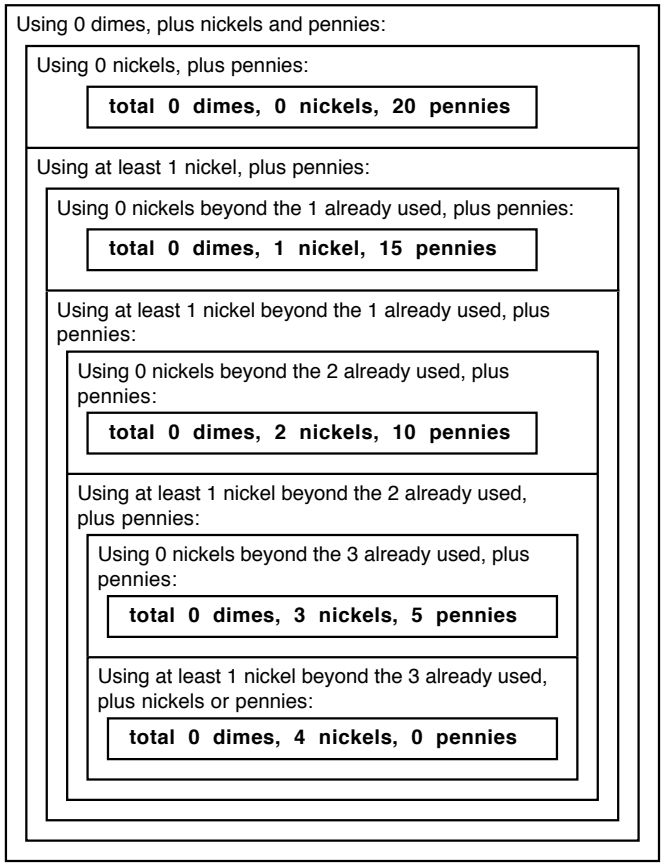
Number of ways to change amount *a* using *n* kinds of coins =

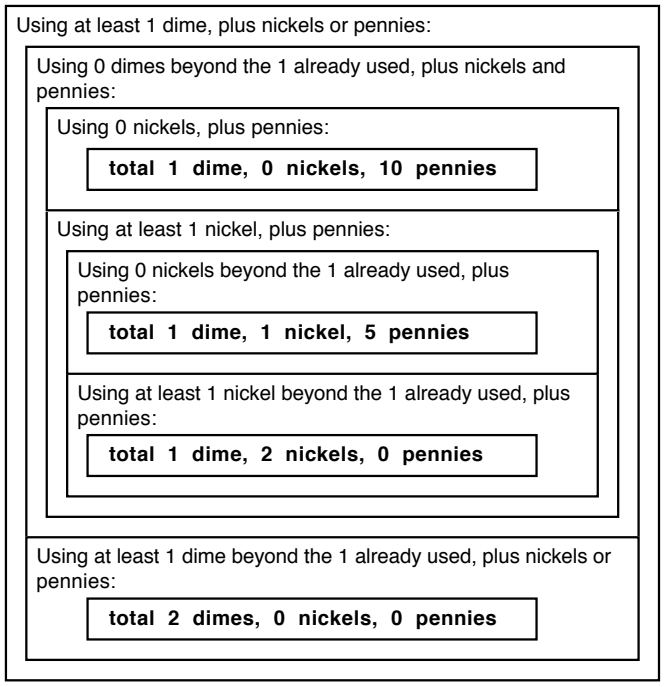
Number of ways to change amount *a* without using the first kind of coin

\+ Number of ways to change amount *a* using *at least one* of the first kind of coin.

The first term in the sum is a straightforward translation of the first term in Abelson and Sussman's recurrence relation. To see how the second term above corresponds to the second term in Abelson and Sussman's sum, consider how the first coin is used. If we know that at least one of the first kind of coin is used, we can separate that coin from the amount. What's left is an amount reduced by the first coin value, which is to be made using any of the specified kinds of coins.

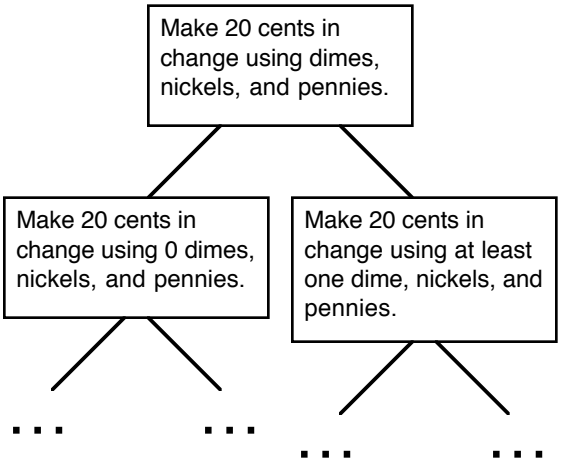
We can solidify our understanding by using the recurrence relation to generate the ways to make 20 cents using dimes, nickels, and pennies. The diagram below does this, with inner boxes elaborating the possibilities for the enclosing boxes.




Using at least 1 dime, plus nickels or pennies:
Using 0 dimes beyond the 1 already used, plus nickels and pennies:
Using 0 nickels, plus pennies:
total 1 dime, 0 nickels, 10 pennies
Using at least 1 nickel, plus pennies:
Using 0 nickels beyond the 1 already used, plus pennies:
total 1 dime, 1 nickel, 5 pennies
Using at least 1 nickel beyond the 1 already used, plus pennies:
total 1 dime, 2 nickels, 0 pennies
Using at least 1 dime beyond the 1 already used, plus nickels or pennies:
total 2 dimes, 0 nickels, 0 pennies

11. (analysis) Label each box in the diagram with the amount of change made by coins already chosen, and the amount of change remaining to make.

12. (reflection) Another way to represent the diagram is as a tree, with the choices available at a given point represented by subtrees. Here's an example:



Which representation do you prefer?

What pattern for solving counting problems is illustrated by these recurrence relations?

All the recurrence relations just described are examples of splitting a counting problem into nonoverlapping groups that together cover all the possibilities. Our first algorithm split the number of ways to make change into the ways that used 0 of the first kind of coin, those that used 1 of the first kind of coin, and so on. Obviously each way to make change falls into exactly one of these categories. Similarly, the Abelson and Sussman algorithm counts the ways that use 0 of the first kind of coin, and those that use ≥1 of the first kind of coin. Again, each way of making change falls into exactly one of these groups.

13. (application) Consider the following way to count the number of integers between 1 and 100, inclusive:

the number of integers between 1 and 100
= the number of integers between 1 and 100 that are divisible by neither 3 nor 5
\+ the number of integers between 1 and 100 that are divisible by 3
\+ the number of integers between 1 and 100 that are divisible by 5.

Check this "equation"; if it is incorrect, fix it.

14. (application) The *partitions* of a positive integer are the different ways to break the integer into pieces. The number 5 has seven partitions:

5	(one piece)
4, 1	(two pieces)
3, 2	(two pieces)
3, 1, 1	(three pieces)
2, 2, 1	(three pieces)
2, 1, 1, 1	(four pieces)
1, 1, 1, 1, 1	(five pieces)

The order of the pieces doesn't matter, so the partition 2, 3 is

the same as the partition 3, 2 and thus isn't counted twice. 0 has one partition.

Suppose we wanted to compute the number of partitions of 5. Someone proposes to do it by computing the following sum:

the number of partitions of 0
+ the number of partitions of 1
+ the number of partitions of 2
+ the number of partitions of 3
+ the number of partitions of 4

i.e., the number of partitions of 5 that start with 5, plus the number of partitions that start with 4, plus the number that start with 3, plus the number that start with 2, plus the number that start with 1.

What's wrong with this idea?

How do Abelson and Sussman implement the recurrence relation?

Referring to the Scheme code, we see that the recurrence relation is coded in the cc procedure (in the third condition of the cond expression). The expression

```
(cc (- amount (first-denomination kinds-of-coins))
    kinds-of-coins)
```

represents the number of ways to make change for the given amount using at least one of the first kind of coin. Each such way consists of one of the first kind of coin, plus change for the amount that remains. The expression

```
(cc amount (- kinds-of-coins 1))
```

represents the number of ways to make change without using the first kind of coin. We infer from this and from the first-denomination procedure that the values of kinds-of-coins correspond to coin use as follows.

kinds-of-coins	*potential coins to use*
5	half-dollars, quarters, dimes, nickels, and pennies
4	quarters, dimes, nickels, and pennies
3	dimes, nickels, and pennies
2	nickels and pennies
1	pennies only

Kinds-of-coins thus represents the number of kinds of coins left to consider, and is used by first-denomination to select a particular coin to use.

15. (stop and analyze) What would the call

```
(cc 100 4)
```

represent?

16. (stop and analyze) What changes to the code would be necessary to compute the number of ways to make change for a given amount using only dimes, nickels, and pennies?

17. (stop and analyze) What changes to the code would be necessary to compute the number of ways to make change for a given amount using only half-dollars, dimes, and pennies?

18. (stop and analyze) What changes to the code would be necessary to compute the number of ways to make change for a given dollar amount, using bills of denominations $50, $20, $10, $5, $2, and $1?

What recursive patterns are used here?

Note that this solution corresponds to a common pattern of recursion with integers. Any recursive procedure on integers will have base cases that represent the integer values for which the procedure can immediately return a value. It also has one or more recursive calls that solve smaller versions of the same problem, using values that are closer to the base cases.

A typical situation is that the base cases represent values as small as possible. Then the arguments to the recursive call should be smaller values than those that were passed to the procedure. Thus the pattern may be represented as follows:

```
if the value is as small as possible, return the
    corresponding result;
        ...
call the procedure with a smaller argument.
```

The cc procedure has two integer arguments, amount and kinds-of-coins. There are base cases for each, representing their smallest possible values. The recursive calls solve smaller versions of the same problem, either with a decremented amount and the same collection of coins, or with the same amount and a smaller collection of coins.

19. (application) List some other examples of this recursive pattern.

20. (application) Write a procedure sum-facts that, given an integer *n*, returns the sum of the factorials of values from 1 to *n*. Thus

```
(sum-facts 4)
```

should return 33, i.e. 1!+2!+3!+4! = 1+2+6+24.

How were the values for the base cases derived?

Normally, a good way to determine that a recursive routine is correct is to make a table, building it up from small values to large values. One problem in this case is that we have not determined why each base case returns 0 or 1. Why should (cc amount 0) be 0 rather than 1? Does (cc 0 1) mean there is 1 way to make change for 0 cents rather than 0 ways?

To clarify the base cases we will trace the code on some sample values and then go back to see what would have happened if the base values had been chosen differently. We start with arguments as small as possible and then build up from them.

test case	*how the answer is computed*	*returned value*
`(cc 0 1)`	Amount = 0, so 1 is returned.	1
`(cc 1 1)`	Neither of the base cases is true, so the expression `(+ (cc 0 1) (cc 1 0))` is evaluated. (cc 0 1) returns 1, as just noted. (cc 1 0) returns 0 via the second base case.	1
`(cc 2 1)`	Neither of the base cases is true, so the expression `(+ (cc 1 1) (cc 2 0))` is evaluated. (cc 1 1) returns 1, as just noted. (cc 2 0) returns 0 via the second base case.	1
`(cc 1 2)`	Neither of the base cases is true, so the expression `(+ (cc -4 2) (cc 1 1))` is evaluated. (cc –4 2) returns 0, via the second base case. (cc 1 1) returns 1 as just noted.	1
`(cc 2 2)`	Neither of the base cases is true, so the expression `(+ (cc -3 2) (cc 2 1))` is evaluated. (cc –3 2) returns 0, via the second base case. (cc 2 1) returns 1 as just noted.	1

So far all the results make sense. We notice a couple of patterns: (cc amount 1), if amount is greater than 0, should always return the result of evaluating (+ (cc (– amount 1) 1) (cc amount 0)), i.e. 1. In English, this means that there is 1 way to make change for a given amount using only pennies. Also, (cc 3 2) and (cc 4 2) will return (+ (cc –2 2) (cc 3 1)) and (+ (cc –1 2) (cc 4 1)) respectively, and both of these are 1. In English, this means that here is no way to use nickels to make change for amounts of 4 cents or less.

Let's now trace some situations where there are at least two ways to make change.

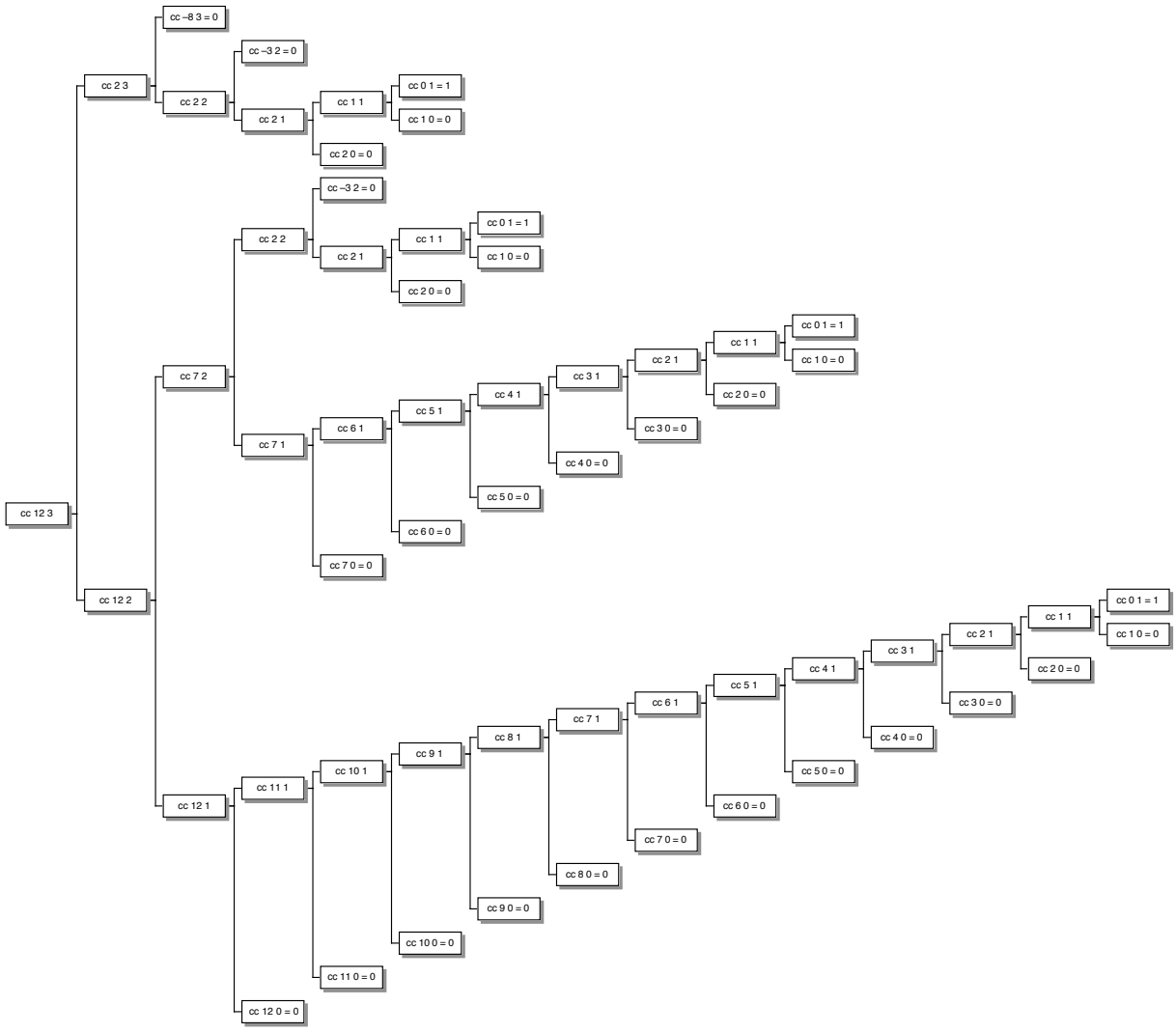
test case	*how the answer is computed*	*returned value*
`(cc 5 2)`	Neither of the base cases is true, so the expression `(+ (cc 0 2) (cc 5 1))` is evaluated. (cc 0 2) returns 1 via the first base case. (cc 5 1) fits the pattern just noticed, so it returns 1.	2
`(cc 6 2)`	Neither of the base cases is true, so the expression `(+ (cc 1 2) (cc 6 1))` is evaluated. (cc 1 2) returns 1 as noted above. (cc 6 1) fits the pattern just noticed, so it returns 1.	2
`(cc 5 3)`	Neither of the base cases is true, so the expression `(+ (cc -5 3) (cc 5 2))` is evaluated. (cc –5 3) returns 0 via the first base case. (cc 5 2) returns 2, as just noted.	2
`(cc 6 3)`	Neither of the base cases is true, so the expression `(+ (cc -4 3) (cc 6 2))` is evaluated. (cc –4 3) returns 0 via the first base case. (cc 6 2) returns 2, as just noted.	2

We now have enough information to explore why the base case values give a correct answer. (cc 1 1) should return 1, since there is one way to make change for 1 cent using pennies; thus either (cc 0 1) is 0 and (cc 1 0) is 1, or vice-versa. Since there is one way to make change for 2 cents using pennies, we want (cc 2 0) to be 0. It makes sense, then, to define (cc amount 0) to be 0 for all amount values and to define (cc 0 k) to be 1 for positive values of k. Since we want (cc 1 2) to be 1, (cc –4 2) should be 0; other cases suggest that (cc amount k) should be 0 for any negative amount.

21. (modification) Rewrite cc to use kinds-of-coins = 1 for its base case. Compare the resulting procedure with Abelson and Sussman's version.
22. (analysis) Does the order in which the base cases are checked matter? Why or why not?
23. (analysis) Is there another way to determine the base cases and get the correct answer?

How does cc really work?

To understand the behavior of cc, it helps to draw a *tree* of evaluations. Each node in the tree represents a call to cc. Diagrammed below is the tree representing the evaluation of (cc 12 2).



24. (analysis) How many calls to cc would be avoided by having the base case for kinds-of-coins be 1 rather than 0?
25. (analysis) Suppose the order in which the coins are tried were reversed. That is, pennies would be tried first, then nickels, then dimes, and so on. Abelson and Sussman claim that this change would not affect the correctness of the computation. To what extent would this affect the number of calls to cc? Hint: limit yourself to nickels and pennies, and compare the trees resulting from (cc 5 2) for each order.
26. (application) A variation on the previous study question about partitions generates a procedure that returns the number of partitions none of whose pieces is bigger than a given size. Here's the procedure that generates all the partitions and a start on the bounded partition problem.

```
(define (num-partitions n)
;; Return the number of partitions of the integer n.
   (cond
      ((< n 0) 0) ; can't partition negative #
      ((= n 0) 1) ; one partition of 0
      (else (num-bounded-parts n n)) ))
```

Here are the first few lines of the procedure num-bounded-parts is the one we just mentioned.

```
(define (num-bounded-parts n max)
;; Return the number of partitions of the integer n,
;; no piece of which is greater than max.
   (cond
      ((= max 0) 0)   ; none with max piece = 0
      ((< n 0) 0)     ; as above
      ((= n 0) 1)     ; as above
      ((< n max) (num-bounded-parts n n))
      (else _______) ))
```

Complete the definition of the procedure num-bounded-parts, and test your pair of procedures with values from 1 to 5.

27. (reflection) When might num-bounded-parts be useful?

28. (application) Compare the procedure number-of-partitions with the count-change procedure by filling in the blank in the following statement:

Counting partitions is like making change, where the coins are _____ .

29. (application) Write and test a procedure number-of-sums to count the ways an integer n can be expressed as the sum of *different* positive integers. Examples:

7 can be expressed as the sum of different positive integers in five ways: 7, 6+1, 5+2, 4+3, and 4+2+1.

9 can be expressed as the sum of different positive integers in eight ways: 9, 8+1, 7+2, 6+3, 6+2+1, 5+4, 5+3+1, and 4+3+2.

30. (application) Compare the procedure you wrote for the preceding problem to the count-change procedure by filling in the blank in the following statement.

Counting ways of expressing *n* as the sum of positive integers is like making change, where the coins are __ .

31. (reflection) How are count-change, number-of-partitions, and number-of-sums similar? What is a good name for this group of procedures?

Critique and improvement of the code

How can the code be improved?

Now that we understand the code, we consider how it might be improved. In general, there are several properties of code to consider:

- its clarity;
- its elegance;

- its efficiency.

32. (stop and think) What parts of the code were difficult to understand?

How can the clarity of the code be improved?

Two things contribute to the clarity of code: the algorithm, and the style in which it is written. In this case, the algorithm is represented by the recurrence relation, which is directly translated in the code. As for style, there are a couple of ways to make this code easier to understand.

(a) Procedure and argument names could be more informative. The name cc is not as good as something like number-of-ways. A comment for the renamed procedure would also be helpful, describing its inputs and output. It would also help to name the initial argument to cc/number-of-ways, say, ALL-COINS (written in upper case to differentiate it from a variable).

(b) Complex procedures could be separated into meaningful parts. It took some study to determine exactly how kinds-of-coins worked. This could have been made more apparent by introducing new procedures called no-more? to check if there were any more coins in the list, and remaining-coins, to remove a coin from consideration.

33. (reflection) Write a principle to govern the clarity of code. Call it the "Literacy Principle".

How can the elegance of the code be improved?

Elegance of code is hard to define. Code is inelegant if it is clumsy, it's overly long, or it makes insufficient use of the features of the programming language in which it's written.

The body of the first-denomination procedure has a long cond that can be replaced with more elegant code. A good technique for eliminating a long cond is to represent the selection as a sentence instead, used with a lookup procedure. (This technique was also used in exercises accompanying the "Difference Between Dates" and "Roman Numerals" case studies.) Here, the sentence would contain coin values, and would be accessed with the builtin procedure item. This would be coded as follows:

```
(define (first-denomination kinds-of-coins)
   (item
      kinds-of-coins
      '(1 5 10 25 50)))
```

But why keep track of the position in the sentence at all? The sentence itself can tell us what coins to use; first-denomination can merely be (first kinds-of-coins), remaining-coins can be (butfirst kinds-of-coins), and no-more? can just be a call to empty?.

34. (reflection) Augment the Literacy Principle to include elegance of code.

How can the efficiency of the code be improved?

To determine if the code can be made more efficient, one may examine the tree of evaluations. In the tree for the computation of (cc 12 3), we see that (cc 2 2) is computed twice, and (cc 2 1) is computed four times. The code would execute more quickly if these results could be saved somehow, and looked up rather than recomputed.

35. (application) Improve the efficiency of the code. (Abelson and Sussman discuss this in exercise 3.27, p. 218.)

36. (reflection) Write an "Efficiency Principle" to use when writing programs in the future.

Is the code improved with these changes?

The result of applying all these improvements appears below. Note that it is now very easy to switch back and forth between representations for the coins to be used.

```
(define ALL-COINS '(50 25 10 5 1))

(define (count-change amount)
   (number-of-ways amount ALL-COINS))

(define (number-of-ways amount kinds-of-coins)
   (cond
      ((= amount 0) 1)
      ((or (< amount 0) (no-more? kinds-of-coins)) 0)
      (else (+
               (number-of-ways
                  (- amount (first-denomination kinds-of-coins))
                  kinds-of-coins)
               (number-of-ways
                  amount
                  (remaining-coins kinds-of-coins)) ))))

(define (no-more? kinds-of-coins)
   (empty? kinds-of-coins))

(define (remaining-coins kinds-of-coins)
   (butfirst kinds-of-coins))

(define (first-denomination kinds-of-coins)
   (first kinds-of-coins))
```

37. (reflection) How else might the code be improved?

Modification of the code to handle limited numbers of coins

How should the limits on the number of coins be used be represented?

We plan modifications to the change-counting procedures now that we understand them fairly well. We need to decide how to represent the number of each coin that is available and how the procedures should be modified to use the represented information.

38. (stop and predict) What parts of the program will need to be modified to deal with limited numbers of coins?
39. (reflection) Will the program need different base cases to deal with limited numbers of coins?

The first decision involves representing the number of each coin available for making change. This information must be passed somehow to the number-of-ways procedure. There are two options: either incorporate the information into the kinds-of-coins argument, or add a third parameter to number-of-ways to specify the available quantity of each coin.

In general, it is better to combine related information into a single package. The fewer parameters, the fewer details to worry about; it will also be easier to keep the two pieces of information coordinated if they are stored together. (Conversely, the more complicated we make a given structure, the more likely we are to make mistakes in updating it or accessing its components.)

Thus we will represent the coin values and quantities in a single sentence, with coin values alternating with coin quantities. The example given in the problem statement, specifying a coin collection of 5 pennies, 3 nickels, and 2 dimes, might then be represented by the sentence

```
(10 2 5 3 1 5)
```

(For clarity we retain the highest-to-lowest arrangement of coins used in the Abelson and Sussman code.)

40. (reflection) Describe an alternative way to represent the number of coins. Compare this representation to the one implemented.
41. (stop and think) What about coins that are not available?

We'll assume that there is at least 1 of each kind of coin, since this will probably simplify the code. That is, a coin sentence like

```
(10 2 5 0 1 5)
```

will not be legal input to the new number-of-ways procedure.

In addition, since the name kinds-of-coins no longer completely describes the coin collection, we choose a better

name for this parameter. The name coin-sent seems appropriate.

What recursion pattern will be used with the coin sentence?

Recursion with sentences typically involves the following pattern:

a base case that tests for an empty sentence, and
a recursive call whose argument is a smaller sentence.

We keep this pattern in mind as we modify the program.

42. (reflection) Create a principle to use when designing the solution to a new recursion problem. Call it the "Recycling Principle".

How is the new representation of the coin sentence used?

We focus now on the number-of-ways procedure to see how it will use the new representation.

43. (stop and predict) What additional procedures will be needed to implement the new representation?

The base cases should not change. The recursion reflects the division of the count into the number of ways using at least one of the first coin and the number of ways not using the first coin; in each case, the new amount and the updated coin sentence is passed to the recursive call. The following changes will be necessary:

- The first-denomination and remaining-coins procedures will have to work correctly for the new representation of the coin sentence.
- The new coin sentence in the first recursive call must represent the use of one of the first kind of coin. Thus we will define a procedure first-coin-used that takes a coin sentence as an argument and returns the result of removing one of the first kind of coin from the sentence. The new call to number-of-ways would then be

```
(number-of-ways
   (- amount (first-denomination coin-sent))
   (first-coin-used coin-sent))
```

What arguments should be passed to number-of-ways to represent the use of the first kind of coin?

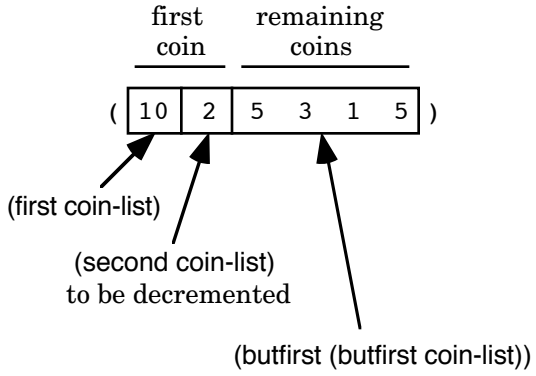
Using one of the first kind of coin involves subtracting 1 from the number of available coins with that value. For example, use of a dime from the coin sentence

```
(10 2 5 3 1 5)
```

should result in the new coin sentence

```
(10 1 5 3 1 5)
```

Essentially, first-coin-used must take the coin sentence apart, then put it back together with the number of the first coin updated. It helps to identify the relevant pieces of the coin sentence, and Scheme code for accessing them, in a diagram:



What is the Scheme code for first-coin-used?

Reading from the diagram, we see that we need to assemble a sentence whose first word is the first word in coin-sent, whose second word is the second word in coin-sent minus 1, and whose remaining words are the third, fourth, etc. words in coin-sent. Here's the code that does this.

```
(sentence
   (first coin-sent)
   (- (second coin-sent) 1)
   (butfirst (butfirst coin-sent)))
```

Coins can run out. How is this handled?

We haven't yet dealt with the situation where the entire quantity of a given kind of coin has been used. Handling this situation merely involves a test: if there is only 1 of the first kind of coin left, first-coin-used should return the result of removing the first coin completely. Here is the completed version of first-coin-used; for clarity, it calls the remaining-coins procedure already used in number-of-ways.

```
; Return the result of using one of
; the first kind of coin in coin-sent.
; Coin-sent is assumed to be nonempty,
; and to include at least one of each
; kind of coin.

(define (first-coin-used coin-sent)
   (if (= (second coin-sent) 1)
      (remaining-coins coin-sent)
      (sentence
         (first coin-sent)
         (- (second coin-sent) 1)
         (remaining-coins coin-sent) ) ) )
```

The complete program for handling a limited number of coins appears in Appendix 1. Comments accompany each procedure, to describe the procedure's purpose along with whatever assumptions are made about its arguments.

44. (analysis) Apply the Literacy Principle and the Efficiency Principle created earlier to this code. What improvements do these principles suggest?
45. (reflection) Improve the Literacy Principle and the Efficiency Principle based on this experience.

How is the program assembled and tested?

It makes sense to test first-coin-used separately from the rest of the code. It involves two cases; we invent an example of each.

46. (stop and help) Test first-coin-used with suitable coin sentences.

Once we're confident that first-coin-used works, we can test the remaining code. Good test data includes extreme values—an empty coin sentence, and a zero amount—as well as "typical" values for which we can easily check the answers.

47. (analysis) What happens if a coin sentence containing a coin of quantity 0 is provided as an argument to number-of-ways?
48. (analysis) What quantity is returned if the check for no more of the first kind of coin in first-coin-used is omitted?
49. (analysis) What happens when a sentence containing an odd number of words is given as an argument to number-of-ways?
50. (analysis) What is returned if a sentence in which the coin values and quantities are out of order is given as an argument to number-of-ways?
51. (reflection) Generate a list of text cases and explain why each is needed to make sure the procedure works properly.
52. (reflection) Create a principle to govern testing of programs. Call it the "Testing Principle".
53. (modification) Modify the program to return a way of making the given amount with the given coins.

Appendix 1 — the complete limited-coins program

```
; Return the number of ways of providing amount cents
; in change using the coins in coin-sent.
; Coin-sent consists of alternating coin denominations
; and nonzero quantities; for instance, (10 2 5 3 1 5)
; is a sentence representing 2 dimes, 3 nickels, and
; 5 pennies.

(define (number-of-ways amount coin-sent)
   (cond
      ((= amount 0) 1)
      ((or (< amount 0) (no-more? coin-sent)) 0)
      (else (+
               (number-of-ways
                  (- amount (first-denomination coin-sent))
                  (first-coin-used coin-sent))
               (number-of-ways
                  amount
                  (remaining-coins coin-sent)) ))))

; Return the result of using one of the first kind of coin
; in coin-sent. Coin-sent is assumed to be nonempty, and to
; include at least one of each kind of coin.

(define (first-coin-used coin-sent)
   (if (= (second coin-sent) 1)
      (remaining-coins coin-sent)
      (sentence
         (first coin-sent)
         (- (second coin-sent) 1)
         (remaining-coins coin-sent) ) ) )


; Return true exactly when there are no more coins
; in coin-sent.

(define (no-more? coin-sent)
   (empty? coin-sent) )

; Return the result of removing all of the first kind of
; coin from coin-sent, which is assumed to be nonempty.

(define (remaining-coins coin-sent)
   (butfirst (butfirst coin-sent)) )

; Return the denomination of the first coin in coin-sent,
; which is assumed to be nonempty.

(define (first-denomination coin-sent)
   (first coin-sent) )

; Return the second word in the sentence s, which is
; assumed to contain at least two words.

(define (second s)
   (first (butfirst s)) )
```