MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 74

A Contour Model Evaluator for λ-Calculus Expressions

by

S. Nimal Amerasinghe

and

D. Austin Henderson, Jr.

Work reported herein was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N0014-70-A-0362-0006 and National Science Foundation Contract GJ004327. Reproduction in whole or in part is permitted for any purpose of the United States Government.

F February 1973

For some time, the semantics of certain programming languages [1, 2] have been defined in terms of the λ-calculus (λC) [3]. Programs are translated into λ-calculus expressions and expressions are 'evaluated' to yield a resulting "value of the program."

Recently, a new model for explicating language semantics has been receiving attention; the Contour Model (CM) has certain intuitive appeal in the way it handles the scope of variables.

This note demonstrates that corresponding to any λ-calculus expression there is a CM algorithm whose evaluation in the CM mimics the SECD evaluation of the λC expression.

λC Expressions

For an exposition of this subject see [3]. It explains that an expression in the λ-calculus is:

1. an identifier
2. a constant
3. some primitive function of two expressions; we will use only addition and ≤
4. the application of one expression to another
5. a λ-expression, consisting of
   a. an identifier
   b. an expression.

Although there are many ways to write λC-expressions, we will use a prefix-polish fully-parenthesized notation. Thus, for example:

$$(\gamma(\lambda x \cdot (x+1))\ 3)$$

is: the application (γ)
of the λ-expression (λ), having
the identifier x, and
the expression,
the sum of
the identifier, x, and
the constant, 1
to the constant, 3.

Another example is:

$$[\gamma(\lambda f \cdot [(\gamma f3) + (\gamma f4)])\ (\lambda x \cdot (x + 1))]$$

Different kinds of parentheses are for clarity only, and have no semantic content. We will also usually omit the γ's.

CM Algorithms

For an exposition of this subject see [4] or [5].

The significant points for our purposes are :

1. contours contain local variables, code to be evaluated, and "nested" contours.
2. entry into a contour causes "new" local variables to be created: copies of those in the algorithm.
3. on entry into a contour, all label variables are closed in the created contour.

Relation to Previous Work

In [6], McGowan discusses the relationship between these two models. He introduces an intermediate language, which he calls "IL." He gives an algorithm for translating λC-expressions into IL programs. He then defines a modified CM evaluator which uses an IL program as its algorithm in place of a CM algorithm demanded by the regular CM.

This note gives an algorithm for translating a λC-expression into a CM-algorithm; this algorithm when evaluated by the regular CM evalator, will mimic the actions of the SECD-machine evaluator on the original λC-expression.

Order of Evaluation

The Church-Rosser theorem of the λ-calculus states that a particular order of evaluation, the Normal Order, is most successful at yielding values for λC-expressions and also that this order is 'canonical' in the sense that any other order of evaluation, if successful, yields the same values as that yielded by Normal Order evaluation.

The SECD machine defines a particular, non-Normal Order of evaluation of λC-expressions. Consequently there are λC-expressions which yield a value when evaluated in Normal Order but which cause the SECD machine to execute a non-terminating sequence of actions. An example is:

$$[[(\lambda x \cdot (\lambda y \cdot y))\ ((\lambda t \cdot tt)(\lambda t \cdot t\,t))]3]$$

See [2] for exposition of this point and further examples.

The effect of translating one of these 'pathological' expressions into a CM algorithm and then evaluating it is, of course, identical to the SECD machine's activities: an infinite loop (See Appendix 1).

The Translation Algorithm

Step 1: Identify subexpression.

Discussion: Start with a fully-parenthesized λC-expression. We will now identify the subexpressions which will correspond to contours.

Algorithm: Label with numbers the following sets of parentheses:

a. The outermost set

b. Each set immediately enclosing a λ-expression.

Example:

$$1[2(\lambda f \cdot ((f4) + (f3)))\ 2\ 3(\lambda x \cdot (x + 1))3]1$$

Step 2: Depict nesting.

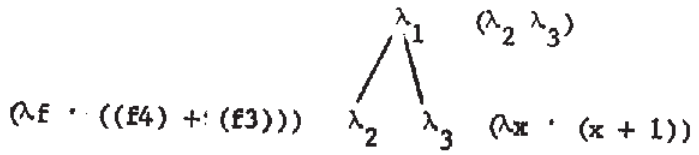
Discussion: This is a formal step which can usually be omitted once familiarity with the algorithm has been achieved. We explicitly depict the nesting of contours and the code for each.

Algorithm:

a. Create a "nesting" tree, labelling each node with the name λ_i, where i is the number of the parenthesis set matched.

b. Beside each node, write the expression contained within the parenthesis set matched, substituting λ_j for the immediately contained labelled subexpressions.

Example:



Step 3: Create the algorithm.

Discussion: This step does all the work. For each node in the nesting tree we create a contour; the contours are nested as indicated in the tree.

Algorithm: Create a contour for each node in the nesting tree. Neat them as indicated by the tree.

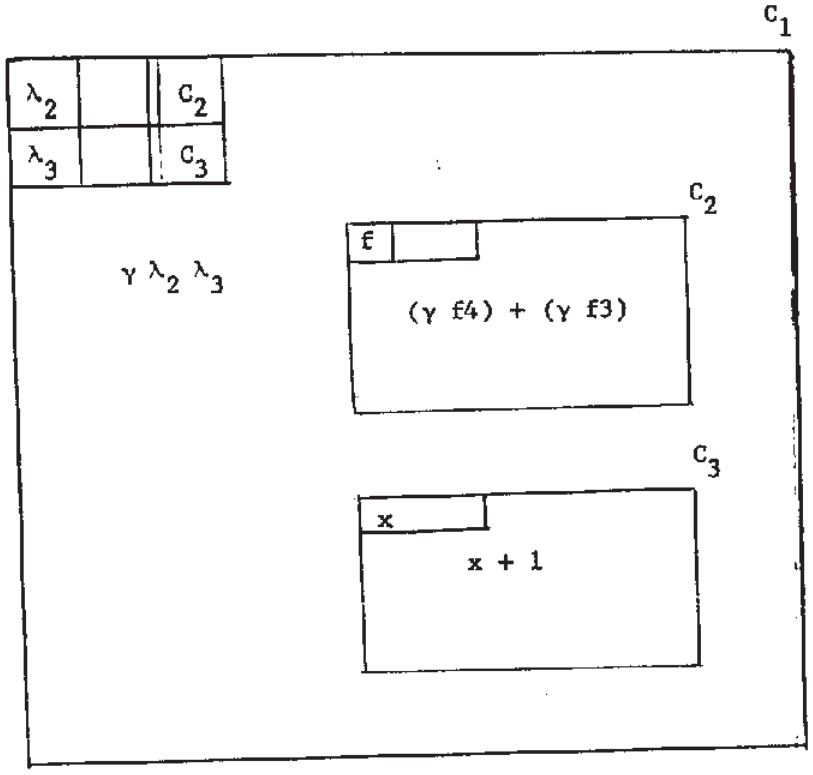
a. Contour name (corresponding to node λ_i): c_i.

b. Contour code: the code written beside the node in the nesting tree, omitting, where relevant; the "λ ⟨identifier⟩." Replace implied γ's with actual ones, and remove unnecessary parenthesis.

c. Contour parameters: if the node represents a λ-expression, use the identifier part of the λ-expression as a parameter variable. Otherwise, (top node) none.

d. Contour "local" variables: One for each immediately enclosed contour (node which is a son in the tree); its name should be λ_k (where the node dominated is λ_k) and its value should be the label value.

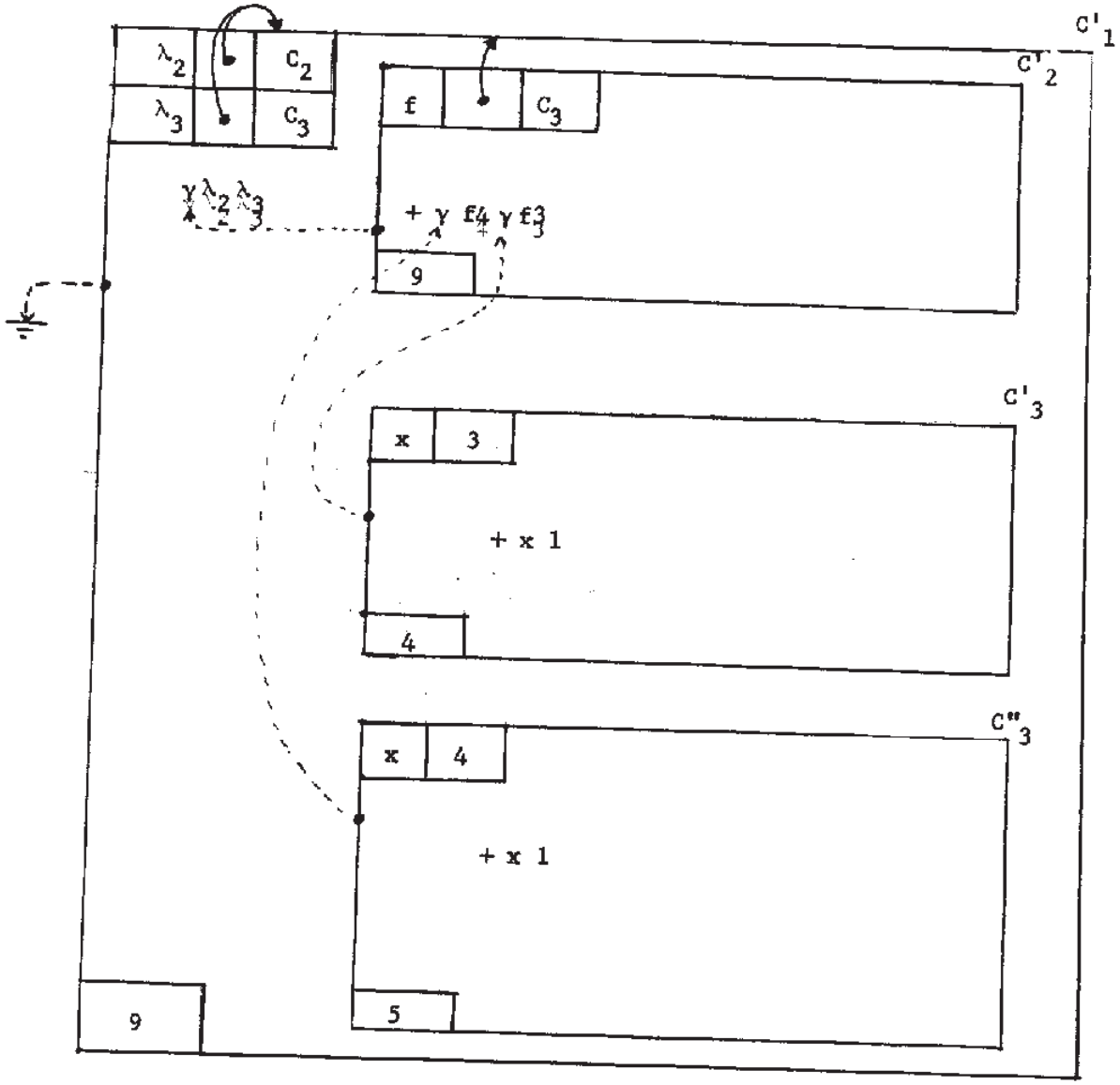
Example:



Notes on Notation: By replacing "x + 1" with "+ x 1" it is possible to have all code execution be purely right-to-left. The laxity with which the code parts of the Contour algorithm are defined reflects the looseness of code specification in the Contour Model.

Execution of the Example

We give here only the final snapshot, with all contours retained. We use the lower left-hand corner of each contour to indicate the value returned. We use the dotted arrows to indicate the call-return point of each contour; the arrow points to the 'γ' in some code which caused the contour in question to be created. A hidden stack mechanism is assumed to implement the right-to-left evaluation of the prefix-Polish code.



A More Complex Example:

λC-expression:

$$([\lambda f.\ (\lambda f.f3)(\lambda x.(fx) + 2)][\lambda x.x+1])$$

Note the bindings of the 'f' identifiers.

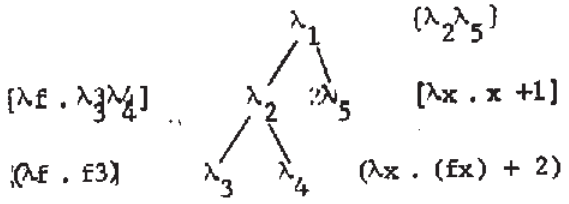
PAL:

```
let f(x) = x + 1
in

Let f(x) = (fx) + 2
in

f3
```

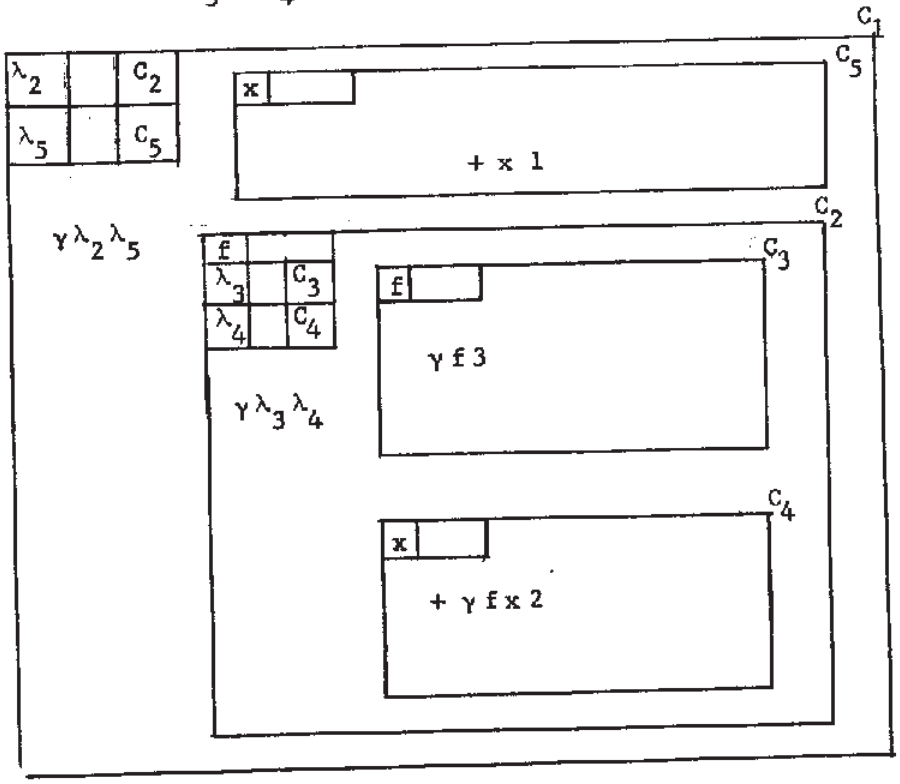
Step 1:

$$\overset{12}{(}[\ \lambda f\ .\ (\overset{3}{\lambda} f\ .\ f3)(\overset{34}{\lambda} x\ .\ (fx) + 2)][\overset{425}{\lambda} x.\ x+1]\overset{51}{]})$$

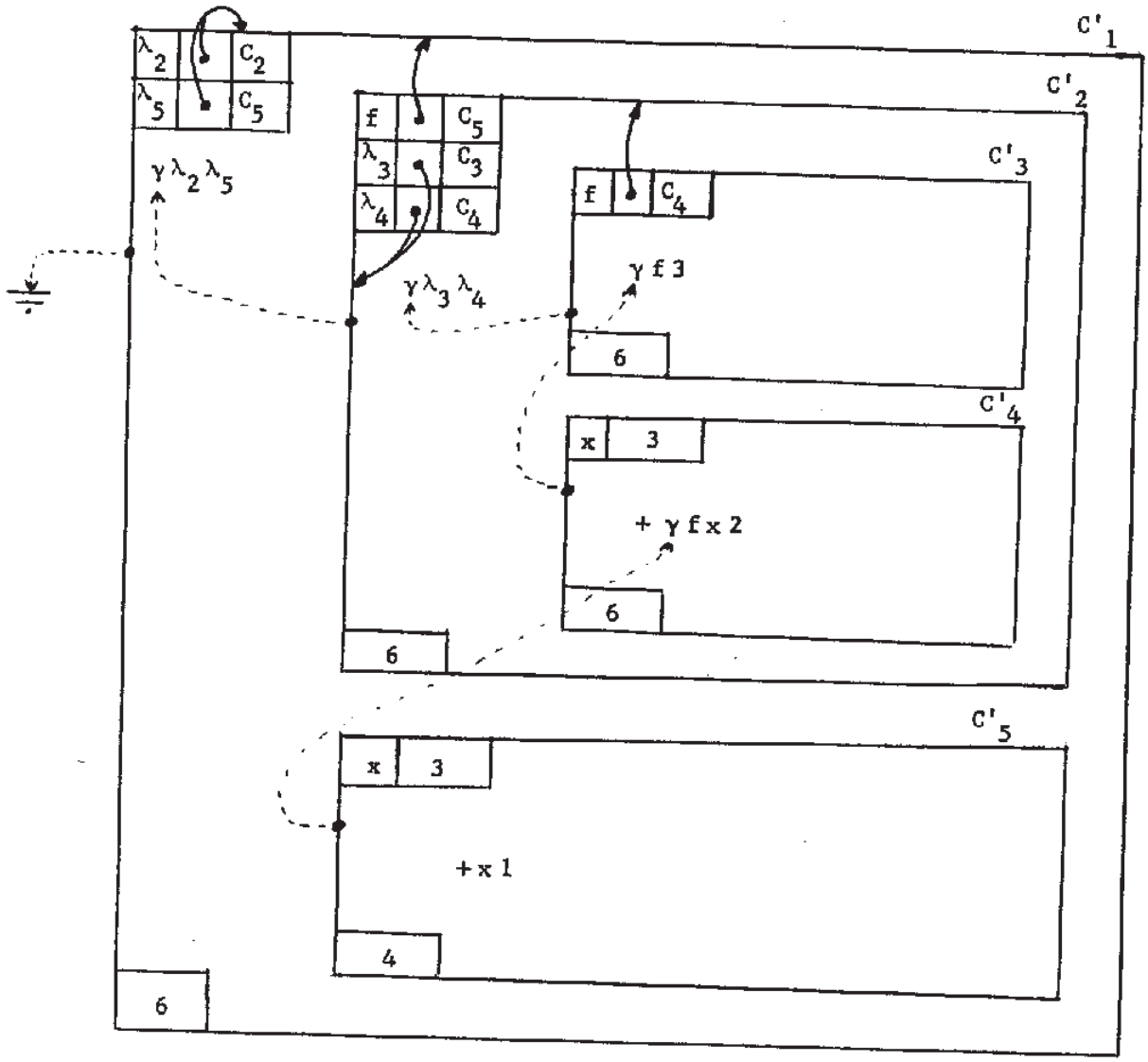
Step 2:



Step 3:



Execution Record



Note that the 'f' in C_4' refers to the variable defined in C_2', while the 'f' in C_3' refers to the f defined there. This mimics the λ-calculus variable scoping rules.

Another Example, with function values

λC-expression:

$$\{[\lambda g\ .\ (\lambda a\ .\ (ga)5)\ 4][\lambda y\ .\ \{\lambda a\ .\ (\lambda g\ .\ g)(\lambda x\ .\ x + a + y)\}3]\}$$

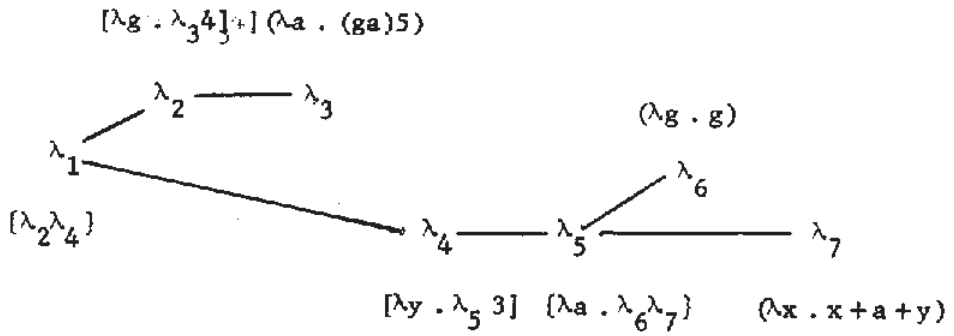
PAL:

```
let g(y) =
    let a = 3
    in
        let g(x) = x + a + y
        in
            g
in
let a = 4
in
    [g(a)](5)
```

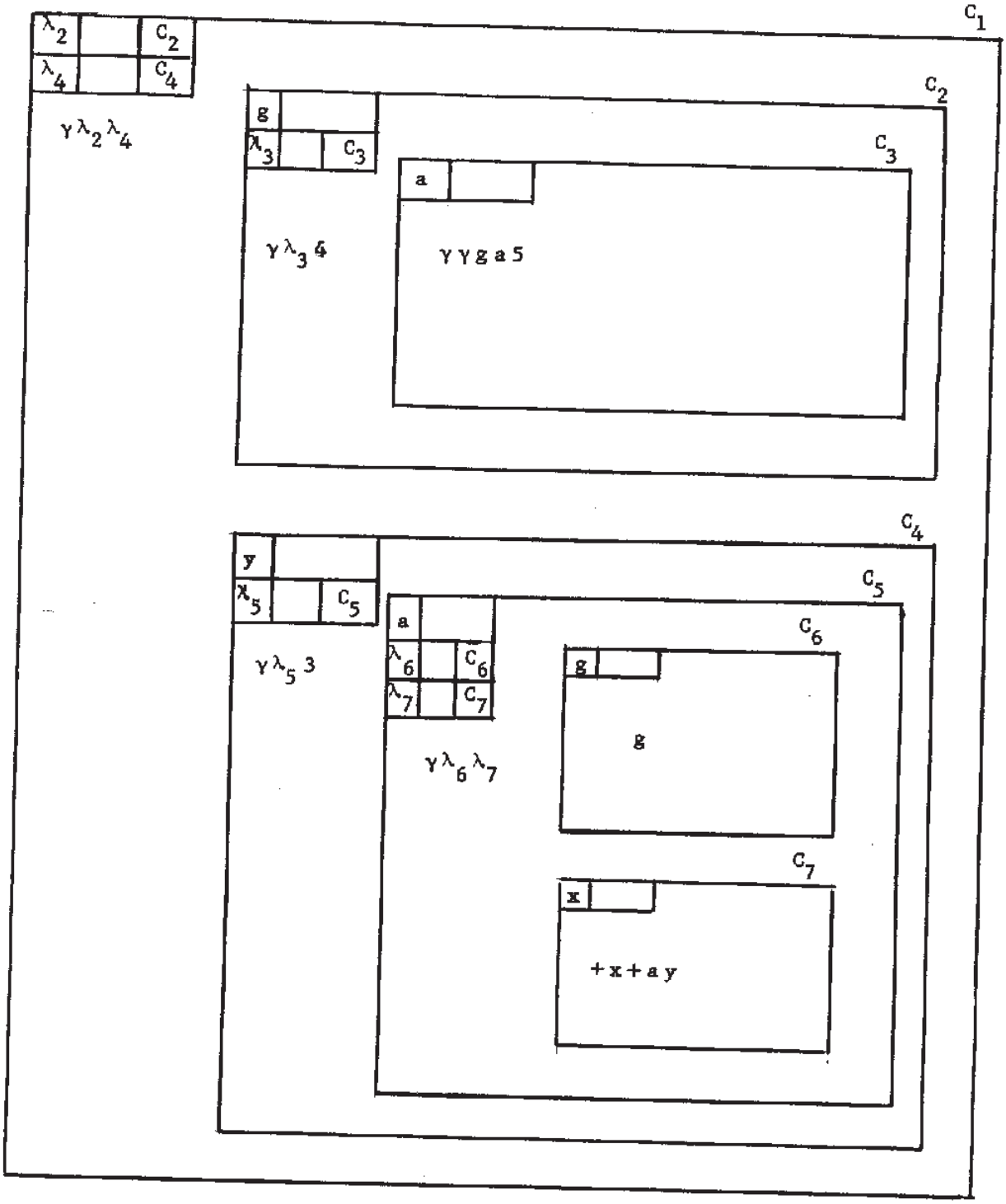
Step 1:

```
1 2    3          3  2 4    5     6      6 7             7 5 4 1
{ [λg . (λa . (ga)5) 4] [λy . {λa . (λg . g) (λx . x+a+y) }3] }
```

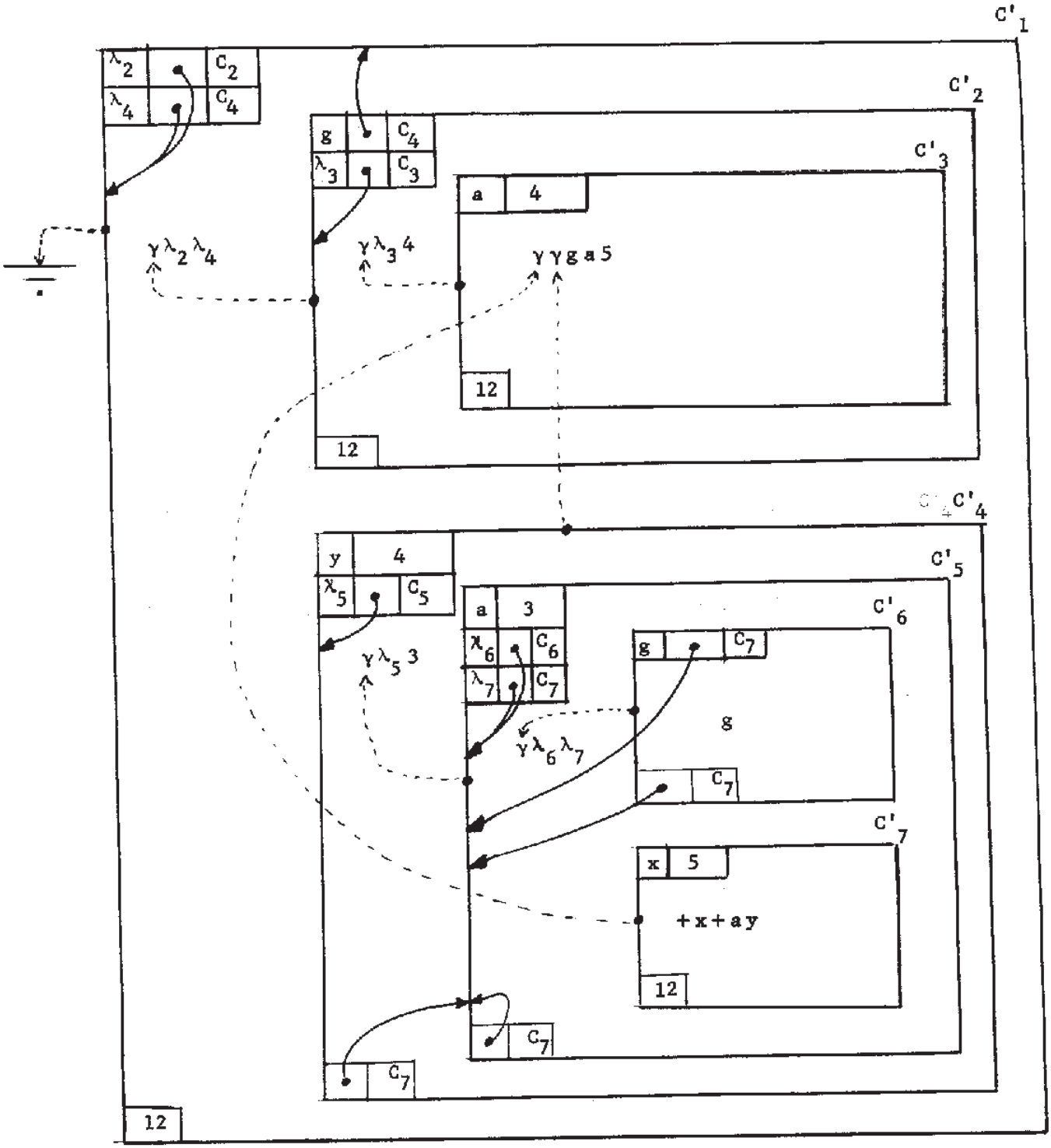
Step 2:



Step 3:


C1
λ2
C2
λ4
C4
γ λ2 λ4
C2
g
λ3
C3
γ λ3 4
C3
a
γ γ g a 5
C4
y
λ5
C5
γ λ5 3
C5
a
λ6
C6
λ7
C7
γ λ6 λ7
C6
g
g
C7
x
+ x + a y

Execution Record



A Comment

The translation algorithm we have presented here is based on the essential similarity of the "scoping" of identifiers in the λ-calculus and the Contour model: namely, "any identifier use refers to the most immediately defining expression/contour defining that identifier."

The only thing to be noted is that the Contour model requires that all functions used be named. Consequently, function denoting λC expressions must be supplied with names for translation into Contour algorithms. We do this by introducing the identifiers "λ_i", which are distinct both from one another and from any identifiers in the λC expression being translated. We conjecture that a translation is impossible without the introduction of function names. This translation is therefore 'satisfactory' only if such name introduction is considered legitimate. There will be those who feel it is not.

Recursion

As explained thoroughly in [2], recursion is achieved in the λ-calculus by using a so-called "fixed-point operator." Thus the recursive factorial function

$$f(x) = \text{if } x \leq 1, \text{ then } 1 \text{ else } f(x-1) * x$$

is expressed as the fixed point of the λC expression:

$$\lambda f . (\lambda x . (\text{if } x \leq 1 \text{ then } 1 \text{ else } f(x-1) * x))$$

That is:

$$Y(\lambda f . (\lambda x . (\text{if } x \leq 1 \text{ then } 1 \text{ else } f(x-1) * x)))$$

where Y is a fixed point operator.

Such a fixed point operatory Y can itself be expressed as a λC-expression. In fact there are many λC-expressions which implement Y; one is

$$\{\lambda G . [\lambda g . gg][\lambda h . G(hh)]\}$$

Thus we see that in one sense, this paper need say nothing more about recursion; we know how to handle λC-expressions, and that is enough to handle the λC-expressions of Y which cause recursive behavior.

However, knowledge of the Contour Model, in particular of its use in explaining ALGOL semantics, leads us to ask,"In ALGOL we get recursion 'free'; why then do we have to stand on our heads to get recursion in λC-expressions"?

We now demonstrate that we can indeed harness the "inherent recursion" capabilities of the Contour Model for λ-calculus recursion.

Translation of $Y(\lambda f . \lambda x . \text{———})$

Step 1: In the original λC-expression, replace

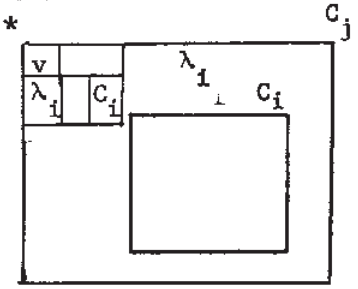
$Y(\lambda f . \lambda x . \text{———})$ with

$Y(\lambda^{*} f . \lambda x . \text{———})$ nil

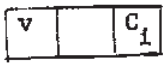
The "*" is a marker which will indicate formation of a special recursive contour in step 3. "nil" is an indicator of "no arguments." Thus, the creation of a recursive function is treated as the application of a special function cell to no arguments.

Step 2: Use the previously-given algorithm to translate the resulting pseudo λC-expression, marking (with '*') those contours generated from marked λ-expressions.

Step 3: Each marked contour will have the form



where v is the identifier which is being recursively defined. Alter the cell for "v" to be



(identical to the cell for λ_i).

Comment on Translation

As contour C_j will be applied to no arguments, the altering of the v cell from an argument form to a local label makes syntactic sense.

As we demonstrate in the following example, the function returned as value of the application of C_j to no arguments is the function cell C_i closed in some contour C_j' generated from algorithm contour C_j. The identifier in C_j' will also have this value. Any application of the returned function value will cause creation of a contour nested within contour C_j'. Hence, uses of the identifier v within such applications will be resolved in C_j', yielding the same function as is being applied. Thus the returned function value is a recursive function.

We therefore have succeeded in harnessing the "intrinsic recursive" capabilities of the Contour Model for λ-calculus recursion.

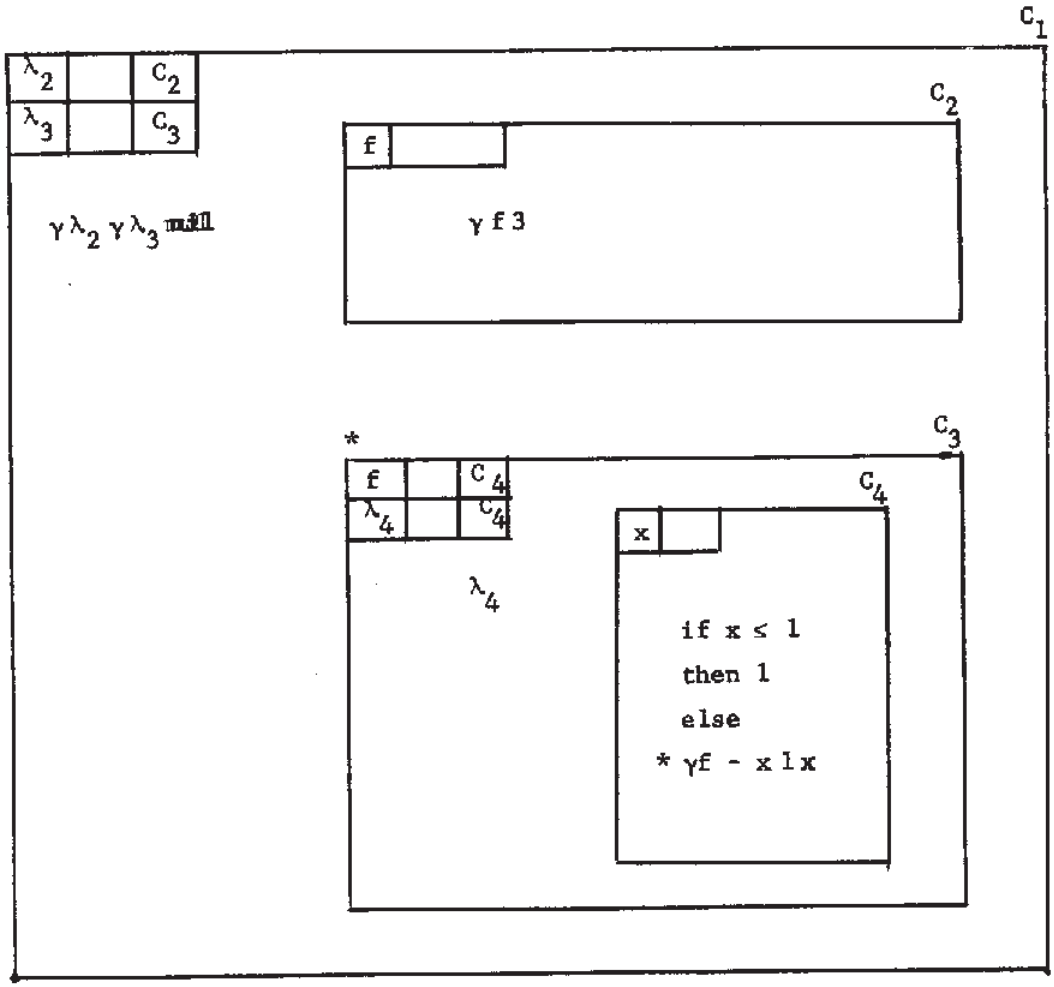
We now demonstrate the translation and execution just described.

Example of Recursive Evaluation

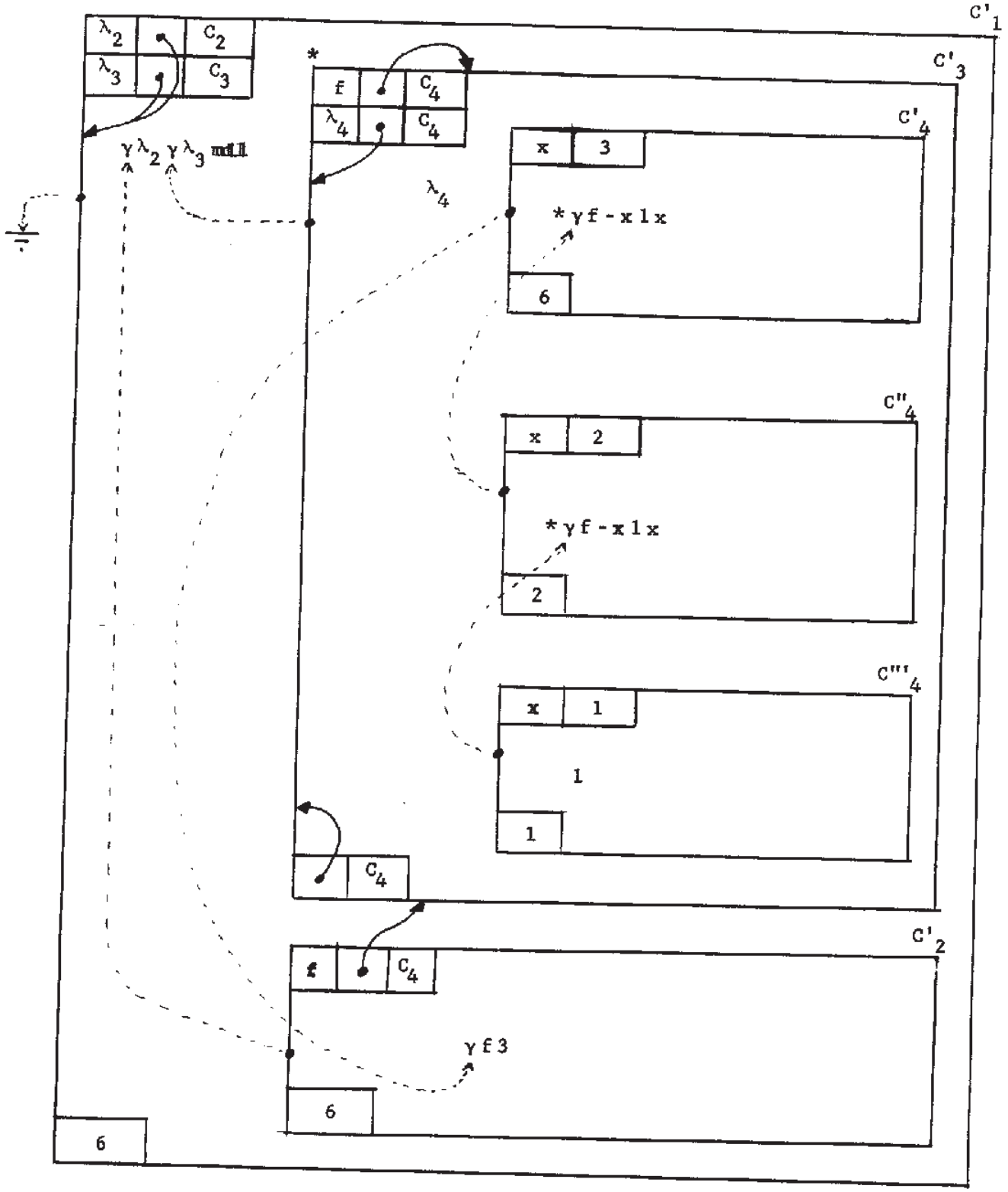
The demonstration uses the factorial function we looked at earlier:

$$\{\ [\lambda f\ .\ f3]\ [Y(\lambda f\ .\ [\lambda x\ .\ (\text{if } x \le 1 \text{ then } 1 \text{ else } f(x-1) * x)]\ )]\ \}$$

Translation:



Evaluation:


C'1
λ2
C2
λ3
C3
*
f
C4
λ4
C4
C'3
C'4
x
3
γλ2 γλ3 mkl
λ4
*γf-x1x
6
C"4
x
2
*γf-x1x
2
C"'4
x
1
1
1
C4
C'2
f
C4
γf3
6
6

Conclusion:

This note gives an algorithm for translating λ-calculus expressions into algorithms for the Contour Model. The algorithm depends on supplying names for the functions in the λ-calculus expression.

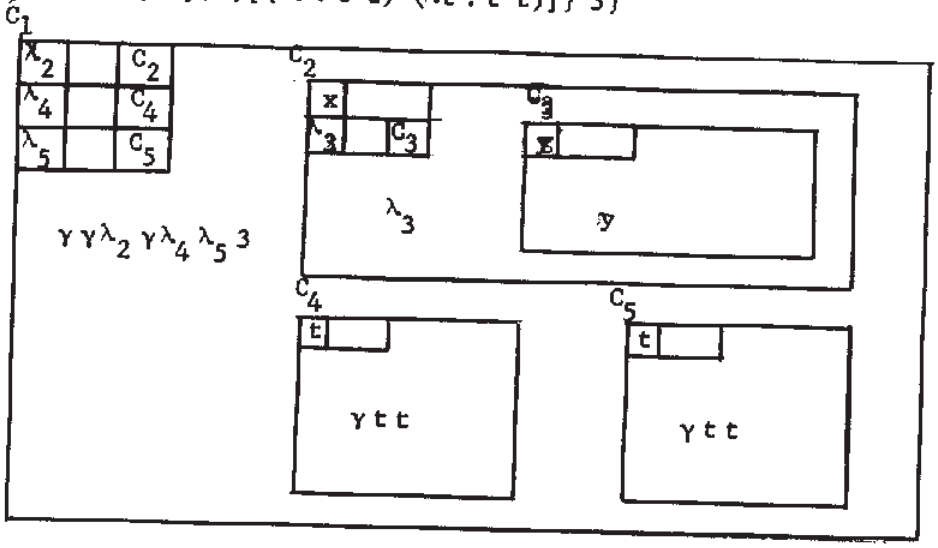
Although recursion is provided for by the λ-calculus itself, it is more appealing to provide for it explicitly in the Contour Model. Consequently, a special rule for translating "Y(λf . (λx:. ———))" is given which yields algorithms causing the Contour Model evaluator to achieve the recursion effect of the λ-calculus Y.

Appendix 1: Loop on Pathological λC-Expressions

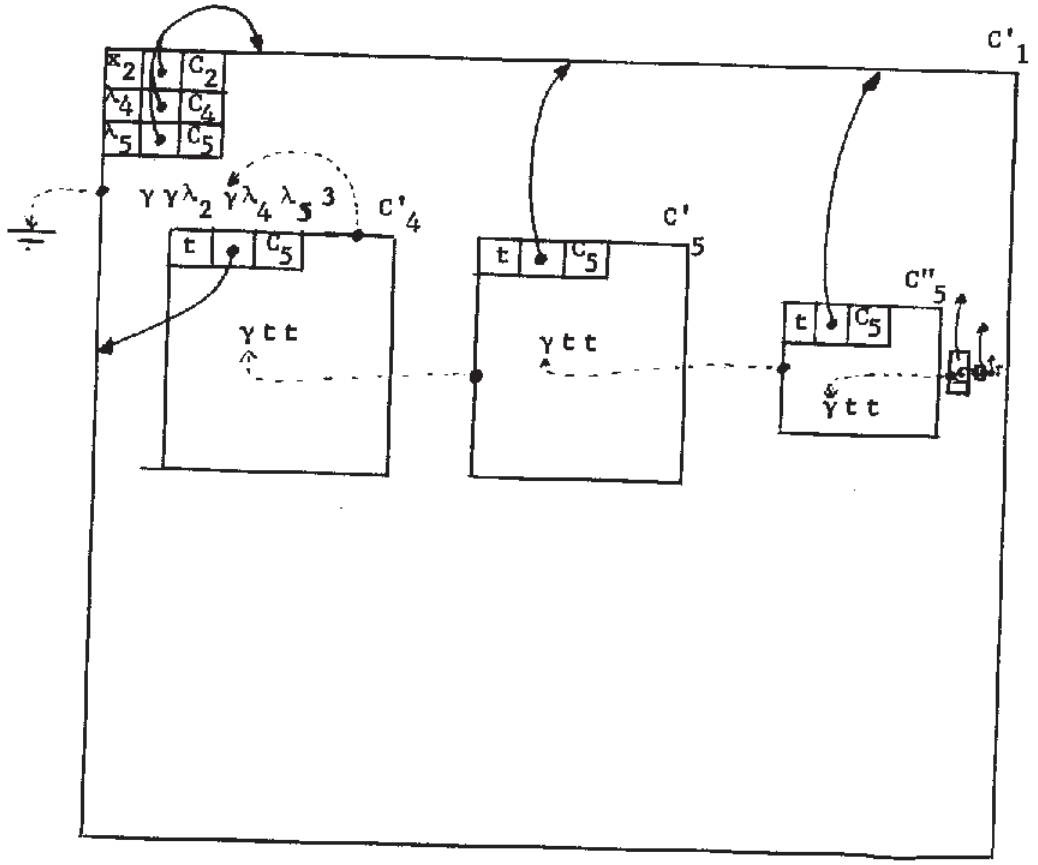
λC-expression

```
11 2    3     3 2 4      4 5      5    1
{{ (λx . (λy . y) )[(λt . t t) (λt . t t)]} 3)
```

Contour Model Algorithm



Incomplete Record of Execution



References

1. Landin, P. J., "The mechanical evaluation of expressions, Comput. J. 6, 4 (January 1964).

2. Wozencraft, J. M. and A. Evans, Jr., "Notes on Programming Linguistics," Department of Electrical Engineering, M.I.T., 1970 (Notes for course 6.231).

3. Church, A., "The calculi of Lambda conversion," Annals of Mathematical Study, No. 6, Princeton, New Jersey, 1941.

4. Johnston, J. B., "The contour model of block structured processes, Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, Vol. 6, No. 2, ACM, February 1971, pp 55-82.

5. Berry, D. M., "Introduction to Oregano," Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, Vol. 6, No. 2, ACM, February 1971, pp 171-190.

6. McGowan, C. L., "The contour model Lambda calculus machine," Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices, Vol. 7, No. 1, ACM, January 1972, pp 110-115.