Abstract
Antoni Diller's book *Compiling Functional Languages* contains a complete compiler and reducer for a simple language. The compiler is a Pascal program of 1379 lines. A version of that program written in the Turing programming language is presented. The new version makes use of Turing language features such as modules and variant records, supports tracing of each pass of the compiler as a new feature, and takes advantage of a number of different minor design and program formatting choices. The Turing version is about 20% smaller than the original version, provides more features and is easier to read.

Contents
1 Introduction 2
2 The Lispkit Language 3
3 The Turing Program 3
   3.1 Main 3
   3.2 Scanning 6
   3.3 Parsing 8
   3.4 Translating 11
   3.5 Reducing 17
4 Examples of Output 22
5 Observations 26

*This note was written during a sabbatical at INRIA (Institut National de Recherche en Informatique et Automatique) in Rocquencourt, France.*
1 Introduction

Antoni Diller’s textbook\(^1\) describes the implementation of functional languages using combinator-based graph reducers. The book was written for senior undergraduate or beginning graduate students. I read it when it first appeared.

Diller’s book contains as an appendix a complete compiler and reducer for a simple Lisp-like language. This program is written in Pascal and comprises 1379 lines. The motivation for its inclusion is given in the introductory paragraphs of the appendix [p.200]:

The program included here is—I believe—a good Pascal program, but—as already mentioned—it is not particularly efficient. By studying it—in connection with Chapters 2 to 5—I hope that the reader will grasp the main ideas involved in the compilation and evaluation of a program written in an applicative language. A highly efficient compiler would not serve this purpose, because efficient programs are difficult to understand. This is usually because they include non-intuitive auxiliary data structures for the sole purpose of improving their performance. Having understood the implementation technique involved, then the reader will be better prepared to refine and develop it. Therefore, the main reason for including the program here is to help the reader understand the main ideas involved in implementing a functional language by means of combinators.

Furthermore, I believe it to be generally true that the ability to intelligently criticize any idea shows that you thoroughly understand it. Therefore, I hope that readers will study critically the program included here, all the time looking out for ways in which the techniques that it contains can be improved and enhanced.

In connection with other work I was doing, I was interested not only in understanding Diller’s program, but also in using the Turing programming language.\(^2\) I translated the program into Turing, taking advantage of several of the language features that were not available to Diller in Pascal.

I took Diller at his word, and studied his program enough to make some changes to it. But I agree with his assessment: his program is a good one. Mine is different because I used different tools. I think my program is better, but this is a credit to the tools not a criticism of Diller. At the same time, there are many relatively small things that I would do differently than Diller (e.g., the choosing of names for variables and routines) if starting from scratch, but I have not made those systematic changes here.

The following changes were made to the program:

- Modules were used as a structuring device. There are modules named Scan, Parse, Translate and Reduce encapsulating the compiler passes, as well as Make and Test to encapsulate routines for manipulating the program tree representation shared by the translator and the reducer.

- Variant records were used to represent nodes in trees and graphs, rather than using records with the union of all the required fields. I found this one of the most difficult aspects of Diller’s program to cope with when first studying it—trying to understand how the records were used for each case.

- Diller’s program uses functions in many places where there are side effects. In particular, his functions often allocate memory for nodes. These side effects are “benign”, in that they do not violate the mathematical definition of a function save in the case of storage exhaustion, but they are not in the spirit of the Turing constraints. My program uses procedures for all of these.

- My scanner is simpler because it always returns a token. It may return an end of file token, which will cause a parse error.

- My graph reducer introduces some procedures not in Diller’s version. For example, I use one routine to handle the manipulation of all binary operators.

---

\(^1\)Antoni Diller, *Compiling Functional Languages*, (Chichester/New York/Brisbane/Toronto/Singapore: John Wiley & Sons Ltd., 1988).

To really know what was going on, I needed to see intermediate stages of the manipulation of each program. I added routines to generate output—to "trace" the action of the system. All of the following can be optionally printed:

- the source program,
- the token stream generated by the scanner,
- the parse tree generated by the parser,
- the program graph generated by the translator, and
- the sequence of combinators applied during reduction.

2 The Lispkit Language

Diller describes the language Lispkit, which is a Lisp-like functional language. Readers familiar with such languages can get the main ideas of the language from the syntax summary given in Figure 1.

The language has lambda abstraction, simple (non-recursive) definitions of named functions, and recursive definitions of named functions. A small set of operators on integers and strings is provided.

The language has very little syntactic sugar. For example, the form of the recursive definition clause is

(letrec
  (context in which to use function)
  (recursive function to use)
)

This simple program written in the language uses a recursive function definition to compute the length of the list \([a\ b\ c\ d\ e]\), given as a constant in the example.

(letrec
  (length (quote (a b c d e)))
  (length lambda (x)
    (if (eq x (quote nil))
      (quote 0)
      (add (quote 1) (length(tail x))))))

This program is one of the examples given in section 4 (the second last), where all of the intermediate output from the compiler is shown.

3 The Turing Program

3.1 Main

This section primarily presents my code, with as little commentary as is possible. I assume readers are familiar with the theory of combinators and also with Diller’s program.

The structure of the program is shown in Figure 2. The first line of each file shown here is a comment containing the name of the file. These file names are used in include statements.

The main program provides definitions for variables shared among modules, including

- the boolean variable controlling intermediate output,
- the variant records used in the parse tree (created by the parser and read by the translator), and
- the variant records used in the program graph (created by the translator and manipulated by the reducer).
\<program\> ::= \<application-clause\>
  | \<letrec-clause\>
  | \<let-clause\>
  | \<quote-clause\>

\<application-clause\> ::= (quote NIL)
  | (\<one-place-op\> \<clause\>)
  | (\<two-place-op\> \<clause\> \<clause\>)
  | (if \<clause\> \<clause\> \<clause\>)
  | (\<non-empty-clause-seq\>)

\<non-empty-clause-seq\> ::= \<clause\>
  | \<clause\> \<non-empty-clause-seq\>

\<one-place-op\> ::= sq | odd | even | head | tail | atom | null | not | chr

\<two-place-op\> ::= add | sub | mul | div | rem | leq | eq | and | or | not

\<clause\> ::= (letrec \<clause\> . \<declaration-list\>)
  | (let \<clause\> . \<declaration-list\>)
  | (lambda \<argument-list\> \<clause\>)
  | (quote \<S-expression\>)
  | \<application-clause\>
  | \<name\>

\<declaration-list\> ::= (quote NIL)
  | (\<non-empty-declaration-seq\>)

\<non-empty-declaration-seq\> ::= ((\<name\> . \<clause\>)
  | (\<name\> . \<clause\>) \<non-empty-declaration-seq\>)

\<argument-list\> ::= (quote NIL)
  | (\<non-empty-argument-seq\>)

\<non-empty-argument-seq\> ::= \<name\>
  | \<name\> \<non-empty-argument-seq\>

\<S-expression\> ::= \<atom\>
  | (\<S-expression-seq\>)

\<S-expression-seq\> ::= \<S-expression\>
  | \<S-expression\> . \<S-expression\>
  | \<S-expression\> \<S-expression-seq\>

\<atom\> ::= \<name\> | \<numeral\>
\<name\> ::= \<letter\> | \<digit\> \<name\> | \<letter\> \<name\>
\<numeral\> ::= \<digit\> | \<digit\> \<numeral\>
\<letter\> ::= a | b | ... | z | A | B | ... | Z
\<digit\> ::= 0 | 1 | ... | 9

Figure 1: The Lispkit Language
The body of the program is a loop that reads a file name and then translates and interprets the program found in the file. The name of a file can be optionally followed by a string of output flags. In the absence of flags, the only output produced is a listing of the source program. Rather than complicating the scanner by having it echo the input, this program reads the input file an extra time to produce the output listing.

```plaintext
% Simple Functional Language Compiler
% David T. Barnard 2 October 1994
%
% This compiler is based on a program in
% Compiling Functional Languages
% Antoni Diller
% John Wiley & Sons Ltd. (1988)
%
% output control
var listing, tracingScan, tracingParse, tracingTranslate, tracingReduce: boolean
%
% stream number for the source file
var source: int
%
% nodes in the parse tree
type tKind: enum(integer, symbol, cons)
var trees: collection of
  union kind: tKind of
    label tKind.integer: value: int
    label tKind.symbol: symbol: string
    label tKind.cons: kar, kdr: pointer to trees
end union
type tree: pointer to trees
var root: tree
```
3.2 Scanning

Each call to the scanner produces a single token for the parser. Since the parser produces no trace output until it has completely constructed the parse tree, the sequence of values produced as trace output from the scanner appears as one block of output even if the parse trace is on. The trace output from this pass is a stream of token values, together with the contents of those that represent classes.
% [w2scan.ml]
% Scanner module

module scan

import {source, token, tracingScan, var numericValue, var symbol, var currentToken, var tokensOnLine}

export {initialize, getToken}

var c: string(1)

function isNumerals: boolean
    result not eof(source) and ("0" <= c and c <= "9")
end isNumerals

function isLetter: boolean
    result not eof(source) and (["A" <= c and c <= "Z"] or ["a" <= c and c <= "z"])
end isLetter

function isWhiteSpace: boolean
    const tab := chr(9); const newline := chr(10); const space := chr(32)
    result not eof(source) and (c = space or c = tab or c = newline)
end isWhiteSpace

procedure getChar
    if not eof(source) then
        get(source, c:1)
    end if
end getChar

procedure getLiteral
    currentToken := token.literal
    symbol := c
    getChar
    loop
        exit when not (isLetter or isNumerals)
        symbol += c
        getChar
    end loop
end getLiteral

procedure getNumeric
    var i := 1
    if c = "+" then
        getChar
    elsif c = "-" then
        i := -1
        getChar
    end if
    if not eof(source) then
        currentToken := token.endFile
    elsif isNumerals then
        currentToken := token.numeric
        numericValue := 0
        loop
            numericValue := 10 * numericValue + (ord(c) - ord("0"))
            getChar
        end loop
        exit when not isNumerals
    end loop
    numericValue *= i
    else
        currentToken := token.error
    end if
end getNumeric

procedure getPunctuationMark {t: token}
    currentToken := t
    getChar
procedure getToken
  loop
    exit when not isWhiteSpace
    getChar
  end loop
  if isLetter then
    getLiteral
  elsif c = '+' or c = '-' or isNumerical then
    getNumeric
  elsif c = '(' then
    getPunctuationMark(token.leftParen)
  elsif c = ')' then
    getPunctuationMark(token.rightParen)
  elsif eof(source) then
    getPunctuationMark(token.endFile)
  else
    getPunctuationMark(token.error)
  end if
  if tracingScan then
    case currentToken of
      label token.numeric: put " [numeric: , numericValue:9 , "] ..
      label token.literal: put " [literal: , symbol:9 , "] ..
      label token.leftParen: put " [leftParen ] ..
      label token.period: put " [period ] ..
      label token.rightParen: put " [rightParen ] ..
      label token.endFile: put " [endFile ] ..
      label token.error: put " [error ] ..
    end case
    tokensOnLine += 1
    if tokensOnLine % 4 = 0 then
      put ""
    end if
  end if
end getToken

procedure initialize
  tokensOnLine := 0
  getChar
  getToken
end initialize

3.3 Parsing

The parser accepts a larger (less constrained) language than that given in Figure 1. It looks for S-expressions which, to provide interesting computations when reduced, may contain lambda abstractions, let clauses, etc. The approach of Diller's program, followed here, is to accept a lot and to reduce whatever can be reduced. Some things will be accepted (parsed, translated, partially reduced) that do not result in much happening.

The trace output from this pass is an indented listing of the parse tree. The generality of the parser means the internal data structure is simple, and thus requires considerable interpretation when the listing is read.
include "s2scan.m"

procedure error(msg: string)
    put "===>", msg
end error

procedure makeSymbolNode (s: string, var t: tree)
    new trees, t
    tag trees(t), tKind, symbol
    trees(t).symbol := s
end makeSymbolNode

procedure makeNumericNode (i: int, var t: tree)
    new trees, t
    tag trees(t), tKind, integer
    trees(t).value := i
end makeNumericNode

procedure cons (x, y: tree, var t: tree)
    new trees, t
    tag trees(t), tKind, cons
    trees(t).kar := x
    trees(t).kdr := y
end cons

procedure atom (var t: tree)
    if currentToken = token.literaI then
        makeSymbolNode(symbol, t)
    elseif currentToken = token.leftParen then
        scan getToken
        sExpressionListPlus(t)
    elseif currentToken = token.period then
        sExpression(t)
    elseif currentToken = token.literal or currentToken = token.numeric then
        atom(t)
    else
        error("Expected s expression")
        t := nil(trees)
    end if
end atom

forward procedure sExpressionListPlus (var t: tree)
import (tree, forward sExpression)

procedure sExpression (var t: tree)
    if currentToken = token.literal or currentToken = token.numeric then
        atom(t)
    elseif currentToken = token.leftParen then
        scan getToken
        sExpressionListPlus(t)
    else
        error("Expected s expression")
        t := nil(trees)
    end if
end sExpression

body procedure sExpressionListPlus
    var newT: tree
    sExpression [new T]
    var aNode: tree
    if currentToken = token.rightParen then
        makeSymbolNode("nil", aNode)
        cons [new T, a Node, t]
    elseif currentToken = token.period then
        scan getToken
        sExpression(t)
        cons [new T, t, aNode]
        t := aNode
        if currentToken not = token.rightParen then
            error("Expected right paren")
        end if
    elseif currentToken = token.literal or currentToken = token.numeric
or currentToken = token.leftParen then
  sExpressionListPlus(t)
  cons(new t, t, aNode)
  t := aNode
else
  error("Expected s expression completion")
  t := nil(trees)
end if
end sExpressionListPlus

\% routines for printing the parse tree

function show(t: tree): string
  if trees(t).kind = tKind.integer then
    result = "int/" + intstr(trees(t).value)
  else \% trees(t).kind = tKind.symbol
    result = "sym/" + trees(t).symbol
  end if
end show

procedure printTree(t: tree, level: int, lineNeeded: boolean)
  if lineNeeded then
    put level:2, repeat(" ", level) ..
  end if
  if t = nil(trees) then
    put "<nil>";
  elsif trees(t).kind = tKind.cons then
    if trees(t).kar.kind not = tKind.cons then
      put "<", show(trees(t).kar), ">" ..
    else
      put "<" ..
      printTree(trees(t).kar, level + 1, false)
      put level:2, repeat(" ", level), "<>
    end if
    if trees(t).kdr.kind not = tKind.cons then
      put show(trees(t).kdr), ">
    else
      put "", printTree(trees(t).kdr, level + 1, true)
    end if
  else
    put "<", show(t), ">"
  end if
end printTree

procedure makeTree [var t: tree]
  if tracingScan then
    put "[Token list begins]"
  end if
  scan.initialize
  sExpression(t)
  if tracingScan then
    if tokensOnLine mod 4 not = 0 then
      put ""
    end if
    put ""
  end if
  if tracingParse then
    put "<<Parse tree begins>>"
    put "<: begins cons node"
    printTree(t, 0, true)
    put ""
  end if
end makeTree
end parse
3.4 Translating

I separated out the routines that make nodes into a module, because when I was reading Diller’s program this notion of “making” a node seemed an abstraction worth representing. Since nodes are made by the translator and also at intermediate stages in reduction, this module is shared by those two.

Similarly, I put the routines that ask questions about the structure and contents of the program graph into a shared module.

---

```plaintext
module make
import {gKind, graph, var graphs}
export {combinator, stringConst, variable, integerConst, applicative}

procedure makeNodeWithSymbol (s: string, k: gKind, var g: graph)
  new graphs, g
  tag graphs(g), k
  graphs(g).symbol := s
end makeNodeWithSymbol

procedure combinator (s: string, var g: graph)
  makeNodeWithSymbol(s, gKind, combinator, g)
end combinator

procedure stringConst (s: string, var g: graph)
  makeNodeWithSymbol(s, gKind, symbol, g)
end stringConst

procedure variable (s: string, var g: graph)
  makeNodeWithSymbol(s, gKind, variable, g)
end variable

procedure integerConst (i: int, var g: graph)
  new graphs, g
  tag graphs(g), gKind.integer
  graphs(g).value := i
end integerConst

procedure applicative (x, y: graph, var g: graph)
  new graphs, g
  tag graphs(g), gKind.application
  graphs(g).rator := x
  graphs(g).rand := y
end applicative
end make

module test
import {gKind, graph, graphs}
export {same, atom, graphNil}

function same (p, q: graph): boolean
  if p = nil(graphs) then
    result q = nil(graphs)
  elseif q = nil(graphs) or graphs(p).kind = graphs(q).kind then
    result true
  elseif graphs(p).kind = gKind.application then
    result same(graphs(p).rator, graphs(q).rator)
    and same(graphs(p).rand, graphs(q).rand)
  elseif graphs(p).kind = gKind.integer then
    result graphs(p).value = graphs(q).value
  elseif % gKind.symbol, gKind.combinator, gKind.variable:
    result graphs(p).symbol = graphs(q).symbol
  end if
end same

function atom (p: graph): boolean

```

---
The translator itself is very similar to Diller’s. The only real change here is the use of Turing’s forward definition feature. Some recursive patterns in Diller’s program are made simpler here by separating the definition of a procedure’s interface from the body of the procedure. This allows the procedure to be called by other procedure that are presented after the header but before the body, which can themselves be called from the body of the forward procedure. Some minor restructuring results from this change.

The trace output from this phase is an indented listing of the program graph. As is the case with the parse tree, this requires considerable background to be appreciated!

```plaintext
% [h2trans.ml]
% Translator module
module translate

import (tracingTranslate, var make, var test,
        tree, trees, iKind, graph, gKind, var graphs)

export (makeProgram, printGraph)

procedure applicativeCons (a, b: graph, var g: graph)
    var alpha, beta: graph
    make.combinator ("conn", alpha)
    make.applicative(alpha, a, beta)
    make.applicative(beta, b, g)
end applicativeCons

function isTreeNil (p: tree): boolean
    result trees(p).kind = tKind.symbol and trees(p).symbol = "nil"
end isTreeNil

function intLength (p: tree): int
    if isTreeNil(p) then
        result 0
    else
        result 1 + intLength(trees(p), kdr)
    end if
end intLength

function car(p: graph): graph
    result graphs(graphs(p).root).rand
end car

function cdr(p: graph): graph
    result graphs(graphs(p).rand)
end cdr

procedure reduce(procedure p(x, y: graph, var z: graph), a, xs: graph, var g: graph)
    if test, graphNil(xs) then
        g := a
    else
        var alpha: graph
        p[a, car(xs), alpha]
        reduce(p, alpha, cdr(xs), g)
    end if
end reduce

procedure accumulate(procedure p(x, y: graph, var z: graph),
                      k, xs: graph, var g: graph)
    if test, graphNil(xs) then
        g := k
    else
```
var alpha: graph
accumulate(p, b, cdr(xs), alpha)
y[car(xs), alpha, g]
end if
end accumulate

forward procedure makeGraph { t: tree, var g: graph }
import(tree, trees, tKind, var make, graphs, forward translateLetRec,
    forward translateLet, forward translateLambda,
    forward translateQuote, forward translateApplicative)
procedure getListFromTree{ expNmS: string, t: tree, var g: graph }
if isTreeNil(t) then
    make.stringConst("nil", g)
else
    var alpha, beta: graph
    if expNmS = "exp" then
        makeGraph(trees(t).kar, kdr, alpha)
    elsif expNmS = "name" then
        makeGraph(trees(t).kar, kar, alpha)
    else
        makeGraph(trees(t), kar, alpha)
    end if
    getListFromTree{ expNmS, trees(t).kar, beta } applicativeCons(alpha, beta, g)
end if

end if
end getListFromTree

function occurs(x, y: graph): boolean
if test.atom(y) then
    result test.same(x, y)
else
    result occurs(x, graphs(y).rator) or occurs(x, graph(y).rand)
end if
end occurs

function isThree(x, e: graph): boolean
result not test.atom(e) and not test.atom(graphs(e).rator)
    and not occurs(x, graphs(graphs(e).rator).rator)
end isThree

function third(y: graph): graph
result graphs(y).rand
end third

function second(y: graph): graph
result graphs(graphs(y).rator).rand
end second

function first(y: graph): graph
result graphs(graphs(y).rator).rator
end first

function oneth(y: graph): graph
result graphs(y).rator
end oneth

function twoth(y: graph): graph
result graphs(y).rand
end twoth

procedure bracketAbstraction{ x, e: graph, var g: graph }
var alpha, beta, gamma, delta, epsilon: graph
if not occurs(x, e) then
    make.combinator("K", alpha)
    make.applicative(alpha, e, g)
elsif isThree(x, e) then
    if not occurs(x, second(e)) then
        make.combinator("B1", alpha)
        make.applicative(alpha, first(e), beta)
    else
        make.combinator("B2", alpha)
        make.applicative(alpha, first(e), beta)
    end if
end if
end bracketAbstraction
make_applicative(beta, second(c), gamma)
bracketAbstraction[x, third(c), delta]
make_applicative(gamma, delta, g)
else if not occur(x, third(c)) then
make_combinator("C1", alpha)
make_applicative(alpha, first(c), beta)
bracketAbstraction[x, second(c), gamma]
make_applicative(beta, gamma, delta)
make_applicative(delta, third(c), g)
else
make_combinator("C1", alpha)
make_applicative(alpha, first(c), beta)
bracketAbstraction[x, second(c), gamma]
make_applicative(beta, gamma, delta)
make_applicative(gamma, twoth(c), g)
end if
else if not testatom(c) then % isTwo
if not occur(x, twoth(c)) then
make_combinator("C", alpha)
bracketAbstraction[x, oneth(c), beta]
make_applicative(alpha, beta, gamma)
make_applicative(gamma, twoth(c), g)
else if not occur(x, oneth(c)) then
g := oneth(c)
else
make_combinator("B", alpha)
make_applicative(alpha, oneth(c), beta)
bracketAbstraction[x, twoth(c), gamma]
make_applicative(beta, gamma, g)
end if
else
make_combinator("S", alpha)
bracketAbstraction[x, oneth(c), beta]
make_applicative(alpha, beta, gamma)
bracketAbstraction[x, twoth(c), delta]
make_applicative(gamma, delta, g)
end if
else
make_combinator("T", g)
end if
end bracketAbstraction

procedure letRecG(x, y: graph, var g: graph)
var alpha, beta: graph
make_combinator("W", alpha)
bracketAbstraction[x, y, beta]
make_applicative(alpha, beta, g)
end letRecG

procedure translateApplicative(t: tree, var g: graph)
var alpha: graph
getListFromTree("a", t, alpha)
reduce(make_applicative, graphs[graphs[alpha].rator], and, graphs[alpha].rand, g)
end translateApplicative

procedure translateLet(t: tree, var g: graph)
var does: tree := trees(trees(t).kdr).kdr
var alpha, beta, gamma, delta: graph
makeGraph(trees(trees(t).kdr).kdr).kdr, beta, alpha)
gListFromTree("name", does, beta)
accumulate(bracketAbstraction, alpha, beta, gamma)
gListFromTree("exp", does, delta)
reduce(make_applicative, gamma, delta, g)
end translateLet
procedure translateLambda(t: tree, var g: graph)
  var alpha, beta: graph
  makeGraph(trees(t).kdr, kdr, alpha)
  getTreeFromTree(quote(t), trees(t).kdr, kdr, beta)
  accumulate(bracketAbstraction, alpha, beta, g)
end translateLambda

procedure miniTranslate(t: tree, var g: graph)
  var alpha, beta: graph
  if trees(t).kind = tKind.con then
    miniTranslate(trees(t).kdr, kdr, alpha)
    miniTranslate(trees(t).kdr, kdr, beta)
    applicativeCons(alpha, beta, g)
  elseif isTreeNil(t) then
    makeStringConst("nil", g)
  elseif trees(t).kind = tKind.integer then
    makeIntegerConst(trees(t).value, g)
  else
    var intLength = trees(t).kind = tKind.symbol
    makeStringConst(trees(t).symbol, g)
end if
end miniTranslate

procedure translateQuote(t: tree, var g: graph)
  miniTranslate(trees(t).kdr, kdr, g)
end translateQuote

procedure translateLetRec(t: tree, var g: graph)
  var e: tree := trees(t).kdr, kdr
  var decs: tree := trees(t).kdr, kdr
  if intLength(decs) = 0 then
    makeGraph(e, g)
  else
    var nameList, expList, graph
    var alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota: graph
    getTreeFromTree("name", decs, nameList)
    getTreeFromTree("exp", decs, expList)
    if intLength(decs) = 1 then
      makeGraph(e, alpha)
      bracketAbstraction(car(nameList), alpha, beta)
      make,combinator("Y", gamma)
      bracketAbstraction(car(nameList), car(expList), delta)
      make,applicative(gamma, delta, epsilon)
      make,applicative(beta, epsilon, g)
    else
      make,combinator("K", alpha)
      makeGraph(e, beta)
      make,applicative(alpha, beta, gamma)
      accumulate(letRecG, gamma, nameList, delta)
      make,combinator("Y", epsilon)
      make,applicative(epsilon, expList, zeta)
      accumulate(letRecG, zeta, nameList, eta)
      make,combinator("Y", theta)
      make,applicative(theta, eta, iota)
      make,applicative(iota, delta, g)
    end if
  end if
end translateLetRec

body procedure makeGraph
  var s: string := trees(t).symbol
  if s = "eq" or s = "sub" or s = "mul" or s = "div" or s = "rem" or s = "add"
    or s = "odd" or s = "even" or s = "head" or s = "eq" or s = "leq"
    or s = "tail" or s = "atn" or s = "null" or s = "if" or s = "not"
    or s = "and" or s = "or" or s = "cons" or s = "chr" then
    make,combinator(s, g)
  end if
end makeGraph
elsif $s$="null" or $s$="true" or $s$="false" then
  makeStringConst($s, g)
else
  makeVariable($s, g)
end if
elsif $trees(t).kind = tKind.cons$ then
  var $y$: tree := $trees(t).car$
  if $trees(y).kind = tKind.symbol$ then
    if $trees(y).symbol = "letrec" then translateLetRec($t, g)$
    elsif $trees(y).symbol = "let" then translateLet($t, g)$
    elseif $trees(y).symbol = "lambda" then translateLambda($t, g)$
    elseif $trees(y).symbol = "quote" then translateQuote($t, g)$
    else translateApplicative($t, g)$
  end if
else
  translateApplicative($t, g)$
end if
end makeGraph

function show($g$: graph): string
  case $g$.kind of
    label $g$.Kind.combinator: result "comb:" + $g$.symbol
    label $g$.Kind.symbol: result "str:" + $g$.symbol
    label $g$.Kind.variable: result "var:" + $g$.symbol
    label $g$.Kind.integer: result "int:" + intstr($g$.value)
  end case
end show

procedure printGraph($g$: graph, level: int, lineNeeded: boolean)
  if lineNeeded then
    put level:2, repeat(" ", level) ..
  end if
  if $g$ = nil($g$s) then
    put "(nil)"
  else
    if $g$.kind = $g$.Kind.application then
      if $g$s{$g$.rator}.kind not = $g$.Kind.application then
        put "{" ..
        printGraph($g$s{rator}, level + 1, false)
        put level:2, repeat(" ", level), "}" ..
      else
        put "{" ..
        printGraph($g$s{rator}, level + 1, false)
        put level:2, repeat(" ", level), "}"
      end if
    else
      put "\n"
      show($g$s{rand}, "}")
    end if
  end if
end printGraph

procedure makeProgram($t$: tree, $g$: graph)
  makeGraph($t$, $g$)
  if tracingTranslate then
    put "{Program graph begins)"
    put " {: begins appl node"
    printGraph($g$, 0, true)
    put ""
  end if
end makeProgram

end translate
3.5 Reducing

The major change made here is to consolidate the handling of unary operators and of binary operators. There is also some minor restructuring resulting from the use of forward procedures, as was the case with the translator.

The trace output from program interpretation is a stream of the names of the combinators that are applied. I do not attempt to print the values of the operands.

```plaintext
module reduce
import[Kind, graph, var graphs, var make, var test, tracingReduce]
export[eval]

var reductionsApplied: int
const blanks := " 

function isApplicativeList (o: graph): boolean
if test.atom(o) or test.atom(graphs(o).rator) then
result false
else
const y: graph := graphs(graphs(o).rator).rator
result graphs(y).kind=gKind.combinator and graphs(y).symbol="con"
end if
end isApplicativeList

procedure printString(s: string)
put s, blanks(length(s).5) ..
end printString

procedure printProgramGraph(p: graph)
if graphs(p).kind = gKind.integer then
printString(intstr(graphs(p).value))
elsif graphs(p).kind = gKind.application then
printString("(")
printProgramGraph(graphs(p).rator)
printProgramGraph(graphs(p).rator)
printString(")")
else % combinator or symbol or variable
printString(graphs(p).symbol)
end if
end printProgramGraph

forward procedure printProgramGraphAll(q: graph)
import(isApplicativeList, forward printProgramGraphList, var symbolsOnLine)

% left ancestors stack
const aStackMax := 100
type aStackRange: 1..aStackMax
type aStack: array aStackRange of graph

procedure growAStack (var p: aStack, var top: aStackRange)
loop
exit when graphs[p[top]].kind not = gKind.application
p[top + 1] := graphs[p[top]].rator
top += 1
end loop
end growAStack

procedure makeAStack(o: graph, var p: aStack, var top: aStackRange)
top := 1
p[top] := o
```
growAStack(p, top)
end makeAStack

forward procedure evalFun(g: graph, var gPrime: graph)
import(graph, graphs, gKind, aStack, aStackRange, makeAStack, oneReduction)
procedure reduceS (var s: aStack, var top: aStackRange)
  var alpha, beta: graph
  make.applicative(graphs[s[top-1]], rand, graphs[s[top-3]], rand, alpha)
  graphs[s[top-3]], vator := alpha
  make.applicative(graphs[s[top-2]], rand, graphs[s[top-3]], rand, beta)
  graphs[s[top-3]], rand := beta
  top := 3
end reduceS

procedure reduceK (var s: aStack, var top: aStackRange)
  graphs[s[top-2]] := graphs[graphs[s[top-1]], rand]
  top := 2
end reduceK

procedure reduceI (var s: aStack, var top: aStackRange)
  graphs[s[top-1]] := graphs[graphs[s[top-1]], rand]
  top := 1
end reduceI

procedure reduceB (var s: aStack, var top: aStackRange)
  var alpha: graph
  graphs[s[top-3]], vator := graphs[s[top-1]], rand
  make.applicative(graphs[s[top-2]], rand, graphs[s[top-3]], rand, alpha)
  graphs[s[top-3]], rand := alpha
  top := 3
end reduceB

procedure reduceC (var s: aStack, var top: aStackRange)
  var alpha: graph
  make.applicative(graphs[s[top-1]], rand, graphs[s[top-3]], rand, alpha)
  graphs[s[top-3]], vator := alpha
  graphs[s[top-3]], rand := graphs[s[top-2]], rand
  top := 3
end reduceC

procedure reduceS1 (var s: aStack, var top: aStackRange)
  var alpha, beta, gamma: graph
  make.applicative(graphs[s[top-2]], rand, graphs[s[top-4]], rand, alpha)
  make.applicative(graphs[s[top-1]], rand, graphs[s[top-3]], rand, alpha)
  graphs[s[top-4]], vator := beta
  make.applicative(graphs[s[top-3]], rand, graphs[s[top-4]], rand, gamma)
  graphs[s[top-4]], rand := gamma
  top := 4
end reduceS1

procedure reduceB1 (var s: aStack, var top: aStackRange)
  var alpha, beta: graph
  make.applicative(graphs[s[top-1]], rand, graphs[s[top-2]], rand, alpha)
  graphs[s[top-4]], vator := alpha
  make.applicative(graphs[s[top-3]], rand, graphs[s[top-4]], rand, beta)
  graphs[s[top-4]], rand := beta
  top := 4
end reduceB1

procedure reduceC1 (var s: aStack, var top: aStackRange)
  var alpha, beta: graph
  make.applicative(graphs[s[top-2]], rand, graphs[s[top-4]], rand, alpha)
  make.applicative(graphs[s[top-1]], rand, graphs[s[top-4]], rand, beta)
  graphs[s[top-4]], vator := beta
  graphs[s[top-4]], rand := graphs[s[top-3]], rand

end procedure
top <- 4
end reduceCI

procedure reduceY (var s: aStack, var top: aStackRange)
  graphs[s[top-1]].rand := graphs[s[top-1]].rand
  top := s[top-1]
end reduceY

procedure reduceU (var s: aStack, var top: aStackRange)
  var alpha, beta, gamma, delta, epsilon: graph
  make.combinator("head", alpha)
  make.combinator("tail", delta)
  make.combinator(gamma)
  graphs[s[top-2]].rand := epsilon
  top <- 2
end reduceU

procedure reduceBinOp (op: string[1], var s:aStack, var top: aStackRange)
  var alpha, beta, gamma: graph
  evalFun(graphs[s[top-1]].rand, alpha)
  evalFun(graphs[s[top-2]].rand, beta)
  const a := graphs[alpha].value
  const b := graphs[beta].value
  if op = "+" then make.integerConst(a + b, gamma)
  elseif op = "-" then make.integerConst(a - b, gamma)
  elseif op = "*" then make.integerConst(a * b, gamma)
  elseif op = "/" then make.integerConst(a div b, gamma)
  else make.integerConst(a mod b, gamma)
end if
graphs[s[top-2]] := graphs[gamma]

procedure boolToStr(b: boolean, var g: graph)
  if b then
    make.stringConst("true", g)
  else
    make.stringConst("false", g)
  end if
end boolToStr

procedure reduceParity(parity: int, var s:aStack, var top: aStackRange)
  var alpha, beta: graph
  evalFun(graphs[s[top-1]].rand, alpha)
  if parity = 0 then
    boolToStr(graphs[alpha].value mod 2 = 0, beta)
  else
    boolToStr(graphs[alpha].value mod 2 not = 0, beta)
  end if
  graphs[s[top-1]] := graphs[beta]
  top := 1
end reduceParity

procedure reduceComp (leqOrEq: string, var s:aStack, var top: aStackRange)
  var alpha, beta, gamma: graph
  evalFun(graphs[s[top-1]].rand, alpha)
  evalFun(graphs[s[top-2]].rand, beta)
  if leqOrEq = "leq" then
    boolToStr(graphs[alpha].value <= graphs[beta].value, gamma)
  else
    boolToStr(test.same(alpha, beta), gamma)
  end if
  graphs[s[top-2]] := graphs[gamma]
procedure reduceHead(var s: aStack, var top: aStackRange)
  var alpha: graph
  evalFun(graphs[s[top-1]], rand, alpha)
  graphs[s[top-1]] := graphs[graphs[alpha], rand], rand)
  top := 1
end reduceHead

procedure reduceTail(var s: aStack, var top: aStackRange)
  var alpha: graph
  evalFun(graphs[s[top-1]], rand, alpha)
  graphs[s[top-1]] := graphs[alpha], rand]
  top := 1
end reduceTail

procedure reduceUnary(op:string, var s: aStack, var top: aStackRange)
  var a, b: graph
  evalFun(graphs[s[top-3]], rand, a)
  if op="sq" then make.integerConst(graphs[a], value=graphs[a], value, b)
  elsif op="ch" then make.stringConst(char(graphs[a], value), b)
  elsif op="at" then boolToStr(testation[a], b)
  elsif op="al" then boolToStr(graphs[a], symbol="nil", b)
  end if
  graphs[s[top-1]] := graphs(b)
  top := 1
end reduceUnary

procedure reduceIf(var s: aStack, var top: aStackRange)
  var alpha: graph
  evalFun(graphs[s[top-3]], rand, alpha)
  if graphs[alpha], symbol = "true" then
    graphs[s[top-3]] := graphs[graphs[s[top-2]], rand]
  else
    graphs[s[top-3]] := graphs[graphs[s[top-3]], rand]
  end if
  top := 3
end reduceIf

procedure reduceBool(first, second: string,
  var s: aStack, var top: aStackRange)
  var alpha, beta, gamma: graph
  evalFun(graphs[s[top-1]], rand, alpha)
  evalFun(graphs[s[top-2]], rand, beta)
  if graphs[alpha], symbol = first then
    boolToStr(graphs[beta], symbol="true", gamma)
  else
    booleanConst(second, gamma)
  end if
  graphs[s[top-2]] := graphs(gamma)
  top := 2
end reduceBool

procedure oneReduction(var spine: aStack, var tspi: aStackRange)
  var s := graphs(spine[tspi]), symbol
  if s = "Y" then reduceK (spine, tspi)
  elsif s = "I" then reduceI (spine, tspi)
  elsif s = "B" then reduceB (spine, tspi)
  elsif s = "C" then reduceC (spine, tspi)
  elsif s = "S1" then reduceSl (spine, tspi)
  elsif s = "S3" then reduceB1 (spine, tspi)
  elsif s = "C1" then reduceCl (spine, tspi)
  elsif s = "Y" then reduceY (spine, tspi)
  elsif s = "Ψ" then reduceU (spine, tspi)
```plaintext
evalFun

body procedure evalFun

var stack: AStack

var stackPtr: AStackRange

makeAStack(g, stack, stackPtr)

var n: graph := stack(stackPtr)

loop

exit when graphs[n].kind not = gKind.combinator or

graphs[n].symbol = "cons"

if tracingReduce then

reductionsApplied += 1

put "/", reductionsAppliedA, ":" ..

printProgramGraphAll(n)

put "/" ..

if reductionsApplied mod 5 = 0 then

put "\n"

end if

end if

oneReduction(stack, stackPtr)

n := stack(stackPtr)

end loop

gPrime := g

evalFun

procedure printProgramGraphList(v: graph)

var p: AStack

var top, len: AStackRange

makeAStack(v, p, top)

printString("(")

var alpha: graph
evalFun(graphs[p[top-1]].rand, alpha)

printProgramGraphAll(alpha)

len := top

loop

exit when len = 3

evalFun(graphs[p[len-2]].rand, alpha)

printProgramGraphAll(alpha)

len -= 1

end loop

if test.graphNil(graphs[p[len-2]].rand) then

printString("\n")

else

printString("\n")

evalFun(graphs[p[len-2]].rand, alpha)

printProgramGraphAll(alpha)
```

4 Examples of Output

Here is the full output for a few programs. Since the option to produce an output listing is turned on, the text of the programs is included here.

The first is a simple program to demonstrate the kind of output from the various tracing routines.

```
===Give filename of program, possibly followed by output flags,===
==== (list, scan, parse, translate, reduce), exit with ".":====
==== s1p02.l===
(add(add(quote 2)(quote 253))(quote 1)).
```

```
[Token list begins]
[<leftParen>][<literal: add>][<leftParen>][<literal: add>][<leftParen>]
[<leftParen>][<literal: add>] [numeric: 2] [<rightParen>]
[<leftParen>][<literal: add>] [numeric: 253] [<rightParen>]
[<rightParen>][<leftParen>][<literal: add>] [numeric: 1]
[<rightParen>][<rightParen>][<rightParen>]

<<Parse tree begins>>
<: begins cons node
4 <:<sym: quote> 6 <:<int: 253> <sym: nil> 4 <sym: nil> 1 <>
2 <:<sym: quote> 4 <:<int: 1> <sym: nil>
```
The second program is also a simple one. It illustrates the point made earlier, that the reducer does only what it can and then it stops. In this case it can do nothing.

The final program is more complex; it was given as an example earlier in the paper. It contains a recursive definition of a function to compute the length of a list.
(letrec
  (length (quote (a b c d e)))
  (length lambda (x))
  (if (eq x (quote nil))
      (quote 0)
      (add (quote 1) (length(tail x)))))
).
{{Program graph begins}}
{{: begins appl node
0: {{::comb:C1}{comb:I1}}
  1 {}  
  2 {{::comb:cons}{str:a}}  
  3 {}  
  4 {{::comb:cons}{str:b}}  
  5 {}  
  6 {{::comb:cons}{str:c}}  
  7 {}  
  8 {{::comb:cons}{str:d}}  
  9 {}  
 10 {{::comb:cons}{str:e}}  
 11 {}  
0}  
1 {{::comb:Y}}  
2 {{::comb:B1}{comb:S}}  
3 {}  
4 {{::comb:C1}{comb:I1}}  
5 {}  
6 {{::comb:C1}{comb:eq}}  
7 {}  
8 {{::comb:cons}{str:nil}}  
9 {}  
10 {{::comb:cons}{str:nil}}  
11 {}  
12 {{::comb:B1}}  
13 {{::comb:B1}{comb:add}}  
14 {}  
15 {{::comb:C1}{comb:B}}  
16 {}  
17 {{::comb:tail}}
//Execution begins//
//  6:01 //  7:if  //  8:01 //  9:eq  // 10:I  //
// 66:tail // 67:tail /
//Reductions applied: 67/
//Execution ends with result://
5 Observations

These are, of course, my own opinions. Readers are invited to form their own conclusions from comparing the two versions.

1. The use of modules makes the program easier to read because it clearly delimits the scope of procedures. The Turing program is patently less flat than the Pascal program.

2. The use of variant records certainly does not reduce the amount of text in the program, since the Turing \texttt{case} statement (or sometime the \texttt{if} statement) is used to separate the different instances, but it makes more clear what data are being constructed or manipulated in a node.

3. The use of procedures rather than functions with (benign) side effects makes some of the code less easy to read. The explicit allocation of temporary nodes keeps getting in the way of what is happening mathematically. This decision should perhaps be reconsidered.

4. The various straightforward changes in structure (no errors handled in the scanner, grouping similar operations in the reducer, etc.) all make the program more readable.

5. The various forms of intermediate output make the program more valuable as a pedagogical device since the output allows students to see precisely what is constructed for various inputs.