

**Simultaneous Evolution of Programs and their Control Structures**  
by Lee Spector

**Full citation:**

Spector, L. 1996. Simultaneous Evolution of Programs and their Control Structures. In *Advances in Genetic Programming 2*, edited by P. Angeline and K. Kinnear. Cambridge, MA: MIT Press.

## Lee Spector

This chapter shows how a genetic programming system can be used to simultaneously evolve programs and their control structures. Koza has previously shown that the performance of a genetic programming system can sometimes be improved by allowing for the simultaneous evolution of a main program and a collection of automatically defined functions (ADFs). This chapter shows how related techniques can be used to simultaneously evolve a main program and a collection of automatically defined macros (ADMs). Examples are provided to show how the use of ADMs can lead to the production of useful new control structures during evolution, and data is presented to show that ADMs sometimes provide a greater benefit than do ADFs. The chapter includes a discussion of characteristics of problems that may benefit most from the use of ADMs or from architectures that include both ADFs and ADMs. It is suggested that ADMs are likely to be useful for evolving intelligent action systems for complex environments, and data is presented to show that this is the case for one such application. The chapter concludes with a discussion of several directions for further research.

### 7.1 Introduction

Modern programming languages support the production of structured, modular programs through several mechanisms including subroutines, coroutines, and macros. Koza has previously shown that the power of a genetic programming system can often be enhanced by allowing for the simultaneous evolution of a main program and a collection of subroutines [Koza 1994a]. Additional studies have investigated factors underlying the performance of Koza's automatically defined functions (ADFs) along with alternative techniques for the automatic generation of subroutines. For example, Angeline and Pollack developed an alternative technique called Module Acquisition (MA) [Angeline and Pollack 1992, Angeline 1994] and Kinnear compared the utility of ADFs with that of MA [Kinnear 1994]. Additional alternatives have also been proposed [Iba et al. 1993, Rosca and Ballard 1995].

While subroutines promote program modularity and code reuse, they do not normally provide programmers with the tools needed to produce new *control structures* or to otherwise enhance the structure of their programming languages. Many languages provide an alternative mechanism, macros, to support this need [Kernighan and Ritchie 1988, Steele 1990].

This chapter shows how a genetic programming system can simultaneously evolve a program and its control structures; that is, it shows how a genetic programming system can simultaneously evolve a main program and a collection of automatically defined macros (ADMs). Experiments on a variant of Koza's obstacle-avoiding robot problem are presented to show that ADMs are in some cases more useful than ADFs. For other problems, however, ADFs are more useful than ADMs. ADFs and ADMs serve different purposes — as do functions and macros in common programming languages — and we can therefore expect architectures that include both ADFs and ADMs to provide the best support for the evolution of programs in certain complex domains.

Sections 7.2 and 7.3 briefly describe the genetic programming framework and the use of ADFs in genetic programming. Sections 7.4 and 7.5 discuss the use of macros to define new control structures and show how a genetic programming system can simultaneously evolve a main program and a set of ADMs that are used by the main program. The remainder of the chapter presents data from case studies, a discussion of the results, and directions for future research.

## 7.2 Genetic Programming

Genetic programming is a technique for the automatic generation of computer programs by means of natural selection [Koza 1992]. The genetic programming process starts by creating a large initial population of programs that are random combinations of elements from problem-specific function and terminal sets. Each program in the initial population is then assessed for fitness. This is usually accomplished by running each program on a collection of inputs called fitness cases, and by assigning numerical fitness values to the output of each of these runs; the resulting values are then combined to produce a single fitness value for the program.

The fitness values are used in producing the next generation of programs via a variety of genetic operations including reproduction, crossover, and mutation. Individuals are randomly selected for participation in these operations, but the selection function is biased toward highly fit programs. The reproduction operator simply selects an individual and copies it into the next generation. The crossover operation introduces variation by selecting two parents and by generating from them two offspring; the offspring are produced by swapping random fragments of the parents. The mutation operator produces one offspring from a single parent by replacing a randomly selected program fragment with a newly generated random fragment.

Over many generations of fitness assessment, reproduction, crossover, and mutation, the average fitness of the population may tend to improve, as may the fitness of each best-of-generation individual. After a preestablished number of generations, or after the fitness improves to some preestablished level, the best-of-run individual is designated as the result and is produced as the output from the genetic programming system.

The impact of alternative approaches to genetic programming can only be assessed by measuring performance over a large number of runs. This is because the algorithm includes random choices at several steps; in any particular run the effects of the random choices may easily obscure the effects of the alternative approaches.

To analyze the performance of a genetic programming system over a large number of runs one can first calculate  $P(M,i)$ , the cumulative probability of success by generation  $i$  using a population of size  $M$ . For each generation  $i$  this is simply the total number of runs that succeeded on or before the  $i$ th generation, divided by the total number of runs conducted. The more steeply the graph of  $P(M,i)$  rises, the better the system is performing.

Given  $P(M,i)$  one can calculate  $I(M,i,z)$ , the number of individuals that must be processed to produce a solution by generation  $i$  with probability greater than  $z$ .<sup>1</sup>  $I(M,i,z)$  can be calculated using the following formula:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The more steeply the graph of  $I(M,i,z)$  falls, and the lower its minimum, the better the system is performing. Koza defines the minimum of  $I(M,i,z)$  as the “computational effort” required to solve the problem with the given system.<sup>2</sup>

### 7.3 Automatically Defined Functions

Koza has shown that the performance of a genetic programming system, as measured by the number of individuals that must be processed to produce a solution with a probability of 99%, can often be improved by allowing for the simultaneous evolution of a main program and a collection of subroutines [Koza 1994a]. He implements the evolution of subroutines by considering one part of an evolved program to be a main program (or “result producing branch”) while other parts are treated as definitions for automatically defined functions (ADFs). Each ADF may have its own function and terminal set, and hierarchical references between ADFs are allowed. Koza showed that the use of ADFs allows a genetic programming system to exploit regularities of problem domains, improving system performance. As mentioned in the Introduction above, alternative techniques have also been proposed for the automatic generation of subroutines.

### 7.4 Macros

The term “macro” is used in this chapter to refer to operators that perform source code transformations. Many programming languages provide macro definition facilities, although the power of such facilities varies widely. For example, C provides substitution macros by means of a preprocessor [Kernighan and Ritchie 1988], while Common Lisp allows the full power of the programming language to be used in the specification of macros [Steele 1990]. A macro “call” is textually transformed into new source code prior to compilation or interpretation; this process is often called *macro expansion*.

Macros, like subroutines, can assist in the modularization of complex programs and in the exploitation of domain regularities. In certain circumstances macros can be more useful than subroutines.<sup>3</sup> In particular, one can implement new *control structures* with macros.

<sup>1</sup>For the analyses in this chapter a value of  $z=99\%$  is always used.

<sup>2</sup>The  $P(M,i)$  and  $I(M,i,z)$  measures were developed by Koza and are discussed on pages 99 through 103 of [Koza 1994a].

<sup>3</sup>A good discussion of related issues can be found in [Graham 1994].

One does this by writing macros that expand into code fragments that include the arguments to the macro call, unevaluated, in one or more places. If the bodies of code that appear as arguments to the macro work by side effect or are sensitive to their calling contexts, then the macro call can produce an effect not obtainable with subroutines. For example, consider the following Common Lisp definition for a macro called `do-twice`<sup>4</sup>:

```
(defmacro do-twice (code)
  `(progn ,code ,code))
```

Consider the case in which we have a global variable `accumulator` and a function `accumulate` that takes one numeric argument and has the side effect of adding the value of the argument to `accumulator`. The call `(do-twice (accumulate 23))` will expand into `(progn (accumulate 23) (accumulate 23))` and will cause the side effect to be produced twice; 46 will be added to `accumulator`. `Do-twice` could not have been implemented as a Common Lisp *function* because the arguments in a function call are evaluated before being passed. A `do-twice` function would, in this context, receive only the *result* of `(accumulate 23)`, not the code; it therefore could not produce the effect of two calls to `(accumulate 23)`.

More generally, the utility of macros stems in part from the fact that they control the evaluation of their own arguments. This allows one to use macros to implement control structures that perform multiple evaluation or conditional evaluation of bodies of code.

One often builds new macros that leverage the utility of pre-existing macros or built-in control structure syntax. For example, one could use an existing `if` structure to build an `arithmetic-if` control structure that branches to one of many bodies of code on the basis of the result of a specified numerical calculation. Similarly, one could use an existing `while` structure, along with problem-specific operators for a robot control domain, to build a `while-no-obstacles` control structure that causes a body of code to be repeatedly evaluated until an obstacle is sensed.

There are many domains in which problem-specific control structures are useful. In a robot control domain one might want to use a control structure that causes an action to be repeated until a condition in the world becomes true. For example, the following macro causes the robot to turn until the given `sense-expression` returns non-nil, and returns the value of the given `value-expression` in its final orientation:

```
(defmacro where-sensed (sense-expression value-expression)
  `(progn (while (not ,sense-expression)
            (turn))
         ,value-expression))
```

---

<sup>4</sup>The “backquote” syntax used in this definition is documented in [Steele 1990]. Briefly, a backquote specifies that the following expression is a template; the expression is returned literally, except that sub-expressions preceded by a comma are evaluated.

This macro would be most useful when the bodies of code specified for `sense-expression` and `value-expression` depend on the orientation of the robot. If the domain provides many opportunities for the construction of such expressions then `where-sensed` may provide a useful behavioral abstraction that could not be obtained with ordinary subroutines.

## 7.5 Automatically Defined Macros

It is possible to simultaneously evolve a main program and a set of automatically defined macros (ADMs) that may be used by the main program. In the experiments described below, only the simplest sort of *substitution* macros are used; each macro specifies a template for the code into which macro calls will expand. Each ADM definition is treated as if it was defined with `defmacro`, with its body preceded by a backquote, and with an implicit comma before each occurrence of a parameter. Conceptually, macro expansion is performed by replacing the call with the template, and by then replacing occurrences of the macro's parameters with the bodies of code that appear in the call. While languages such as Common Lisp allow one to write macros that specify more general code transformations, substitution macros can nonetheless be quite useful.<sup>5</sup>

In practice one can avoid full expansion of substitution macros by expanding the ADMs incrementally during evaluation. Incrementally expanded substitution ADMs are easy to implement through minor modifications to Koza's publicly available ADF Lisp code [Koza 1994a]. In Koza's code an ADF call is evaluated by binding the ADF's parameter symbols to the (evaluated) arguments passed to the ADF, and by then calling `fast-eval` on the evolved ADF code tree. For ADMs one can change `fast-eval` so that it treats ADMs like pseudo-macros, passing them unevaluated code trees as arguments. One can then evaluate an ADM call by substituting the unevaluated argument code trees for the parameter symbols in the evolved ADM code tree, and by then calling `fast-eval` on the result:

```
(defun adm0 (a0)
  (fast-eval (subst a0 'arg0 *adm0*)))
```

Zongker's C-based `lil-gp` system provides an even simpler way to implement substitution ADMs [Zongker 1995]. One simply specifies the automatically defined modules to be of type `EVAL_EXPR` (rather than `EVAL_DATA`, which is used for normal ADFs), and ADM semantics result. This is achieved by changing the runtime interpretation of each parameter symbol to branch to the code tree passed as an argument; actual expansion of the macro call is thereby avoided altogether.

More complex implementation strategies are required for ADMs that perform non-substitutional transformations of their arguments. In some cases it may be necessary to fully

---

<sup>5</sup>Examples of macros that perform more exotic code transformations can be found in [Graham 1994].

expand the ADMs prior to evaluation; this may lead to long macro-expansion delays and to very large expansions. All of the experiments described in this chapter used substitution ADMs and therefore took advantage of one of the simple implementation techniques described above.

Koza has previously made limited use of macro expansion in genetic programming; he used it as a means for deleting elements of programs during the simultaneous evolution of programs and their architectures [Koza 1994b]. The present work argues for the more general use of macro semantics through the evolution of ADMs.

While ADMs and ADFs are in fact compatible and may be used together, the present study highlights the differences between ADFs and ADMs by using each to the exclusion of the other, and by contrasting the results. The simultaneous use of ADFs and ADMs is suggested as an area for future research.

## 7.6 The Dirt-Sensing, Obstacle-Avoiding Robot

Among the domains in which we expect ADMs to have utility are those that include operators that work by producing side effects, operators that are sensitive to their calling contexts, or pre-existing macros. Koza's obstacle-avoiding robot problem (hereafter "OAR") has all of these elements. The problem as expressed by Koza is quite difficult, and he was only able to solve it by using an unusually large population (4,000). A somewhat simpler version of OAR can be produced by adding an additional sensor function (IF-DIRTY); with this change the problem can easily be solved with a population as small as 500.

The goal in the dirt-sensing, obstacle-avoiding robot problem (hereafter "DSOAR") is to find a program for controlling the movement of an autonomous floor-mopping robot in a room containing harmless but time-wasting obstacles. The problem is an extension of OAR, which Koza describes as follows:

In this problem, an autonomous mobile robot attempts to mop the floor in a room containing harmless but time-wasting obstacles (posts). The obstacles do not harm the robot, but every failed move or jump counts toward the overall limitation on the number of operations available for the task.

...the state of the robot consists of its location in the room and the direction in which it is facing. Each square in the room is uniquely identified by a vector of integers modulo 8 of the form  $(i, j)$ , where  $0 \leq i, j \leq 7$ . The robot starts at location (4,4), facing north. The room is toroidal, so that whenever the robot moves off the edge of the room it reappears on the opposite side.

Six non-touching obstacles are randomly positioned in a room laid out on an 8-by-8 grid. ...

The robot is capable of turning left, of moving forward one square in the direction in which it is currently facing, and of jumping by a specified displacement

in the vertical and horizontal directions. Whenever the robot succeeds in moving onto a new square (by means of either a single move or a jump), it mops the location of the floor onto which it moves. [Koza 1994a, p. 365]

OAR uses terminal sets consisting of the 0-argument function (`MOP`), the 0-argument function (`LEFT`), random vector constants modulo 8 ( $\mathcal{R}_{v,8}$ ), and the names of arguments for ADFs. The same terminal sets are used in DSOAR.

(`MOP`) moves the robot in the direction it is currently facing, mops the floor at the new location, and returns the vector value (0,0). If the destination contains an obstacle then the robot does not move, but the operation still counts toward the limit on the number of movement operations that may be performed in a run. (`LEFT`) turns the robot  $90^\circ$  to the left and returns the vector value (0,0).

The OAR function set consists of the operators `IF-OBSTACLE`, `V8A`, `FROG`, `PROGN`, and the names of the ADFs. DSOAR uses the same function set (substituting the names of ADMs when appropriate) and adds one new operator, `IF-DIRTY`.

`PROGN` is a 2-argument sequencing operator that returns the value of its second argument. `IF-OBSTACLE` is a 2-argument conditional branching operator that evaluates its first argument if an obstacle is immediately in front of the robot; it evaluates its second argument otherwise. `IF-OBSTACLE` is implemented as a pseudo-macro. `V8A` is a 2-argument vector addition function that adds vector components modulo 8. `FROG` is a 1-argument movement operator that jumps the robot to the coordinate produced by adding (modulo 8) its vector argument to the robot's current location. `FROG` acts as the identity operator on its argument, and fails in the same way as `MOP` when the destination contains an obstacle. `IF-DIRTY` is a 2-argument conditional branching operator that evaluates its first argument if the square immediately in front of the robot is dirty; it evaluates its second argument if the square has been mopped or if it contains an obstacle. `IF-DIRTY` is implemented as a pseudo-macro.

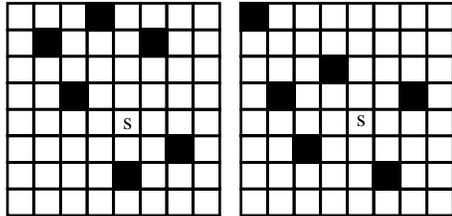
Two fitness cases, shown in Figure 7.1, are used for the DSOAR problem. Each program is evaluated once for each fitness case, and each evaluation is terminated prematurely if the robot executes either 100 (`LEFT`) operations or 100 movement operations (`MOP` and `FROG` operations combined). The raw fitness of each program is the sum of the squares mopped over the two fitness cases. A program is considered to have solved the problem if, over the two fitness cases, it successfully mops 112 of the total of 116 squares that do not contain obstacles.

### 7.6.1 Results

200 runs of a genetic programming system were performed on this problem, half with ADFs and half with ADMs. Each program in each population had an architecture consisting of a result-producing branch and two modules (either `ADF0` and `ADF1` or `ADM0` and `ADM1`). `ADF0` and `ADM0` each took 1 argument and `ADF1` and `ADM1` each took 2 arguments.<sup>6</sup> `ADF1`

---

<sup>6</sup>Koza used a 0-argument `ADF0` and a 1-argument `ADF1` for OAR.



**Figure 7.1**  
The fitness cases used for the DSOAR problem.

and ADM1 could call ADF0 and ADM0, respectively, and result producing branches could call both of the automatically defined modules. The population size was 500 and the maximum number of generations per run was 51. Tournament selection was used with a tournament size of 7.

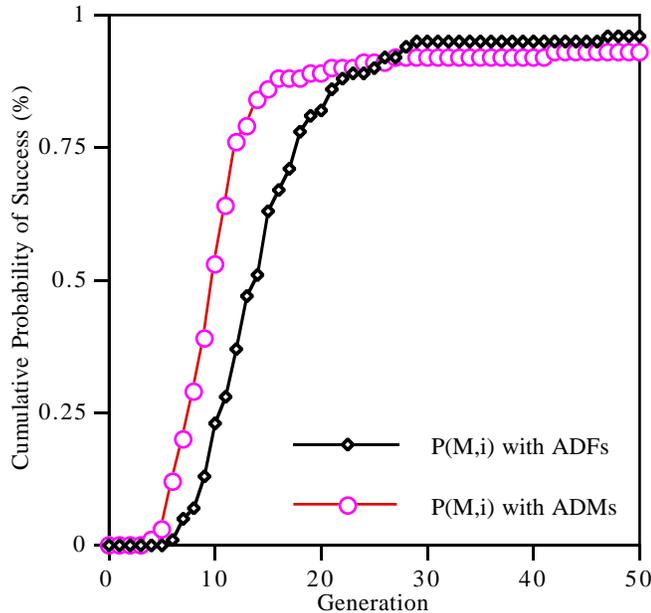
Figure 7.2 shows a summary of the results as a graph of  $P(M,i)$ , the cumulative probability of success by generation. The probability of producing a solution to this problem using a population size  $M=500$  by any given generation is generally greater with ADMs than with ADFs; this can be seen by noting that the ADM line rises faster than the ADF line. Figure 7.3 shows a summary of the results as a graph of  $I(M,i,z)$ , the number of individuals that must be processed to produce a solution with probability greater than  $z=99\%$ . The number of individuals that must be processed is lower for ADMs than for ADFs; this can be seen by noting that the ADM line falls faster than the ADM line, and that it reaches a lower minimum. The minimum, defined by Koza as the “computational effort” required to solve the problem, is 26,000 when ADFs are used; the computational effort is 21,000 when ADMs are used instead.

The results show that for DSOAR, with the given parameters, ADMs are somewhat more useful than ADFs; the number of individuals that must be processed to produce a solution using ADMs is lower than the number needed to produce a solution using ADFs.

## 7.7 The Lawnmower Problem

A related but negative result was obtained for Koza’s 64-square Lawnmower problem. The Lawnmower problem is identical to OAR (described above) except that there are no obstacles and the IF-OBSTACLE operator is not used. Note that this domain includes no pre-existing macros and no operators that are sensitive to the robot’s environment. It does, however, include operators that work by side effect, so one might expect ADMs to be more useful than ADFs.

200 runs of a genetic programming system were performed on the 64-square Lawnmower problem, half with ADFs and half with ADMs. Aside from the switch to ADMs for half of

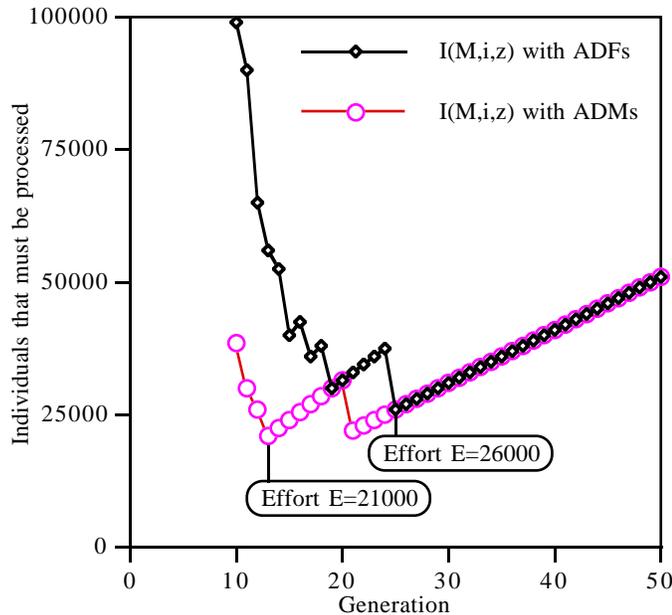


**Figure 7.2**  
 $P(M,i)$  for ADFs and ADMs on the DSOAR problem. Population size  $M=500$ , number of runs  $N=100$ .

the runs, these runs used the same parameters as did Koza's [Koza 1994a]. The results are shown in Figures 7.4 and 7.5. Figure 7.4 shows  $P(M,i)$  for ADFs and for ADMs on this problem; it is not obvious from casual inspection of this graph that either type of module provides greater benefit. Figure 7.5 shows  $I(M,i,z)$ , from which it is clear that ADMs are more of a hindrance than a help on this problem; the computational effort (minimum for  $I(M,i,z)$ ) is 18,000 for ADMs, but only 12,000 for ADFs.

### 7.8 When Are ADMs Useful?

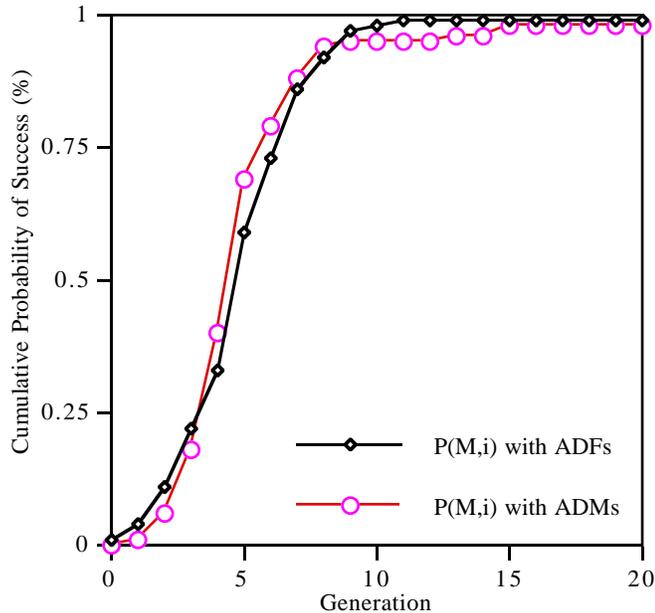
The semantics of ADFs and of substitution ADMs are equivalent when all operators in the domain are purely functional. The only difference between an ADF and an ADM in such a case is that the ADM may take more runtime to evaluate, since the code trees appearing as arguments to the ADM may have to be evaluated multiple times. It is therefore clear that substitution ADMs can only be more useful than ADFs in environments that include operators that are not purely functional — that is, operators that work by side effect or are sensitive to their calling environments.



**Figure 7.3**  
 $I(M,i,z)$  for ADFs and ADMs on the DSOAR problem. Population size  $M=500$ , number of runs  $N=100$ , desired probability of success  $z=99\%$ .

Even when a domain includes operators that work by side effect, there is no guarantee that the use of ADMs will result in less individuals being processed than will the use of ADFs. The negative result on the Lawnmower problem is a testament to this fact. Informally we may speculate that the “less functional” a domain is, the more likely that ADMs will be useful in that domain. If context-sensitive and side-effecting operators play an important role in a given domain, then it is likely that new and/or problem-specific control structures will be useful; we can therefore expect a genetic programming system to take advantage of automatically defined macros to produce control structures that help in evolving a solution to the problem. DSOAR includes two critical context-sensitive operators that are not present in the Lawnmower problem (`IF-OBSTACLE` and `IF-DIRTY`). It is therefore not surprising that ADMs are more useful in DSOAR than in the Lawnmower problem.

Although ADFs and ADMs have been contrasted above for expository purposes, they are in fact completely compatible with one another. Just as a human programmer may wish to define both new functions and new control structures while solving a difficult programming problem, it may be advantageous for genetic programming to define a collection of ADFs

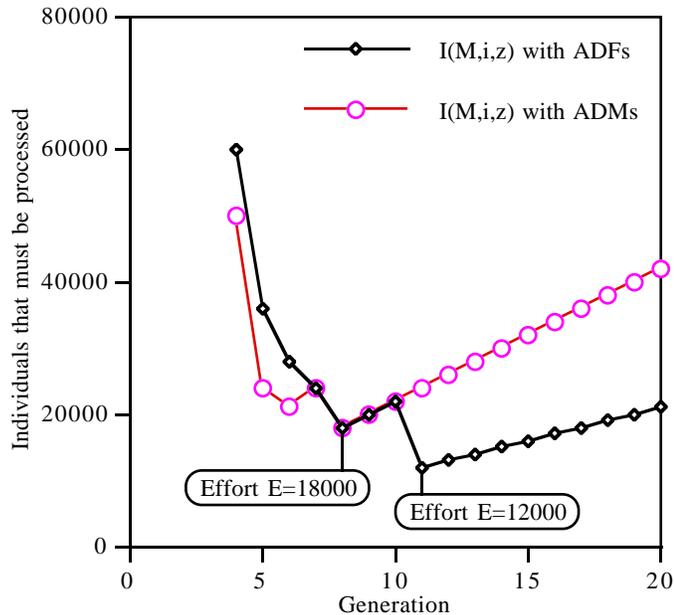


**Figure 7.4**  
 $P(M,i)$  for ADFs and ADMs on the Lawnmower problem. Population size  $M=1,000$ , number of runs  $N=100$ .

and a collection of ADMs. Functions may be most helpful for some aspects of a problem domain, while macros may be most helpful for others. Since the optimal number of ADFs and ADMs may not be clear from the outset, it may also be advantageous to simultaneously evolve programs and their macro-extended architectures, in the style of [Koza 1994b].

A reasonable speculation is that architectures that include ADMs, or some combination of ADFs and ADMs, will be particularly useful in application areas that have traditionally made use of exotic control structures. One such application area is “intelligent action systems,” in which systems are built to reason and act in complex environments.<sup>7</sup> Computational architectures for intelligent action systems tend to use multiple control “levels,” blackboard-based opportunistic control structures, “monitor” processes, and other complex and unusual control structures [Dean and Wellman 1991]. A genetic programming system with ADMs should be capable of evolving and refining such control structures to suit particular problem environments. The next section describes work on the “wumpus world” environment [Russell and Norvig 1995] from which it is clear that ADMs can indeed be useful for the evolution of intelligent action systems.

<sup>7</sup>The names “reactive planning” and “dynamic-world planning” are also sometimes used for this research area.



**Figure 7.5**  
 $I(M,i,z)$  for ADFs and ADMs on the Lawnmower problem. Population size  $M=1,000$ , number of runs  $N=100$ , desired probability of success  $z=99\%$ .

## 7.9 Wumpus World

Russell and Norvig have written an Artificial Intelligence textbook in which the concept of an “intelligent agent” serves as a unifying theme [Russell and Norvig 1995]. In order to motivate their discussions of knowledge representation and reasoning procedures they present an environment called “wumpus world” that “provides plenty of motivation for logical reasoning” (p. 153). Although wumpus world is much simpler than most real-world environments, it is nonetheless more complex than the problem environments discussed above.

Wumpus world is represented as a grid of squares (6 by 6 in the experiments described below) surrounded by walls. The agent’s task is to start in a particular square, to move through the world to find and to pick up the piece of gold, to return to the start square, and to climb out of the cave. The cave is also inhabited by a “wumpus” — a beast that will eat anyone who enters its square. The wumpus produces a stench that can be perceived by the agent from adjacent (but not diagonal) squares. The agent has a single arrow that can be used to kill the wumpus. When hit by the arrow the wumpus screams; this can be heard

anywhere in the cave. The wumpus still produces a stench when dead, but it is harmless. The cave also contains bottomless pits that will trap unwary agents. Pits produce breezes that can be felt in adjacent (but not diagonal) squares. The agent perceives a bump when it walks into a wall, and a glitter when it is in the same square as the gold.

The wumpus world agent can perform only the following actions in the world: go forward one square; turn left  $90^\circ$ ; turn right  $90^\circ$ ; grab an object (e.g., the gold) if it is in the same square as the agent; release a grabbed object; shoot the arrow in the direction in which the agent is facing; climb out of the cave if the agent is in the start square.

The agent's program is invoked by the simulator to produce a single action for each time-step of the simulation. The program itself has no side effects on the world — it simply returns the name of one of the valid actions and the simulator then causes that action, and any secondary effects, to happen in the world. The agent's program may nonetheless contain side-effect-producing operators. This is because the agent may have persistent state and because its program may contain operators that access and modify that state. Since Russell and Norvig present wumpus world as an example of an environment that demands logical reasoning about acquired world knowledge, we can assume that side effects on the agent's state are important for success. One might expect a successful wumpus world agent to use its persistent state to maintain knowledge about visited squares, and to support the deduction of hidden properties of the world (e.g., the locations of pits from breezes).

The agent's program has a single parameter, a "percept" that encodes all of the sensory information available to the agent. The agent's program can refer to the components of the percept arbitrarily many times during its execution; that is, sensing is free.

A variety of function and terminal sets might be used for wumpus world agents. For the present experiments the seven valid actions were simply mapped onto the integers from zero to six (inclusive) and the function set consists of arithmetic operators that manipulate and return numbers in this range.<sup>8</sup> The agent's persistent state is implemented as a two-dimensional (7 by 7) indexed memory [Teller 1994], each element of which holds a single number.

The terminal set consists of the integers from zero to six, a zero-argument operator that returns a random number in the same range (`rand7`), the names of arguments for any ADFs or ADMs, and five sensor variables that are set from the agent's percept. Each of these variables (`*stench*`, `*breeze*`, `*glitter*`, `*bump*`, and `*sound*`) is set to one if the percept contains the corresponding feature, or to zero otherwise.

The function set includes the names of ADFs and ADMs, when appropriate, and the operators listed in Table 7.1.

An agent is assessed on the basis of its performance in four randomly generated worlds. In each world the agent is allowed to perform a maximum of 50 actions. Since the arrangement of gold, pits, and wumpus in each world can vary considerably, success in four randomly generated worlds requires a robust agent. In fact, since the gold may sometimes be

---

<sup>8</sup>The actual mapping from integers to actions is as follows: 0 = forward, 1 = turn right, 2 = turn left, 3 = shoot, 4 = grab, 5 = release, 6 = climb.

**Table 7.1**  
Wumpus World Operators

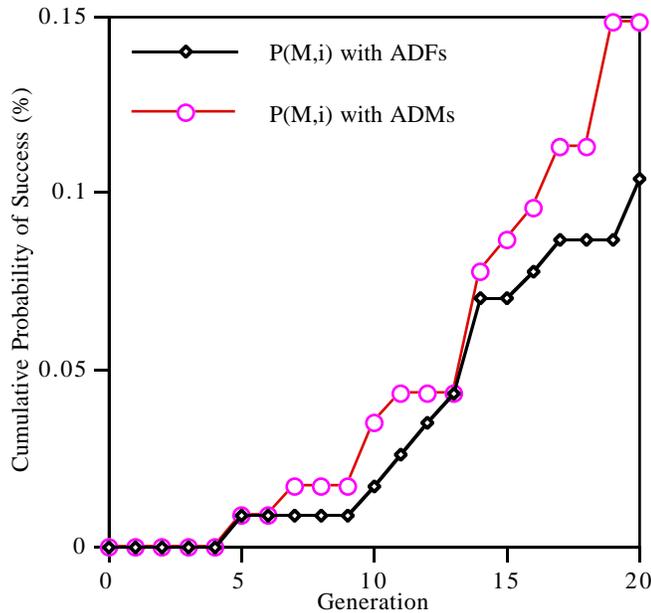
Name	# of arguments	description
and	2	A conjunction operator; returns 1 if both arguments are non-zero, or zero otherwise.
or	2	A disjunction operator; returns 1 if either or both arguments are non-zero, or zero otherwise.
not	1	A negation operator; returns 1 if the argument is zero, or zero otherwise.
progn2	2	A sequencing operator; returns the value of the second argument.
ifz	3	“If zero,” A conditional pseudo-macro; returns the result of evaluating the second argument if the first argument evaluates to zero; returns the result of evaluating the third argument otherwise.
iflte	4	“If less than or equal,” a conditional pseudo-macro; returns the result of evaluating the third argument if the first argument evaluates to a value less than or equal to the second argument; returns the result of evaluating the fourth argument otherwise.
-	2	A subtraction operator; returns the difference of the arguments modulo seven.
+	2	An addition operator; returns the sum of the arguments modulo seven.
*	2	A multiplication operator; returns the product of the arguments modulo seven.
read	2	A memory accessor for a two dimensional indexed memory; returns the contents of the memory location indexed by the arguments. Memory locations contain zero if they have not yet been written to.
write	3	A memory modifier for a two dimensional indexed memory; returns the <i>previous</i> contents of the memory location indexed by the first two arguments, and then fills the memory location with the third argument.

placed in a pit or in a square surrounded by pits, success may sometimes be impossible.

Russell and Norvig score wumpus world agents as follows: 1,000 points are awarded for obtaining the gold, there is a 1-point penalty for each action taken,<sup>9</sup> and there is a 10,000-point penalty for getting killed. While it is possible to use such scores as fitness values for genetic programming, they are not sufficiently informative to reliably drive the genetic programming system to promising regions of the search space. For this reason Russell and Norvig’s scoring system was modified for the present experiments as follows: 100 points are awarded for obtaining the gold, there is a 1-point penalty for each action taken, there is a 100-point penalty for getting killed, and there is a 100-point penalty for each unit of distance between the agent and the gold at the end of the run.<sup>10</sup> Standardized fitness is calculated as the average score in four newly-generated random worlds, subtracted from 100. An agent is considered to have solved the problem when its average score in four random worlds is greater than zero. To have obtained such a score an agent must have grabbed the gold in at least one and usually two or more of the four random worlds, and it can have died in at

<sup>9</sup>The simulation ends when the agent climbs out of the cave or dies, or when 50 actions have been performed.

<sup>10</sup>Note that neither of these scoring systems explicitly rewards an agent for climbing out of the cave, although less action penalties will be accumulated if an agent climbs out and thereby ends the simulation.



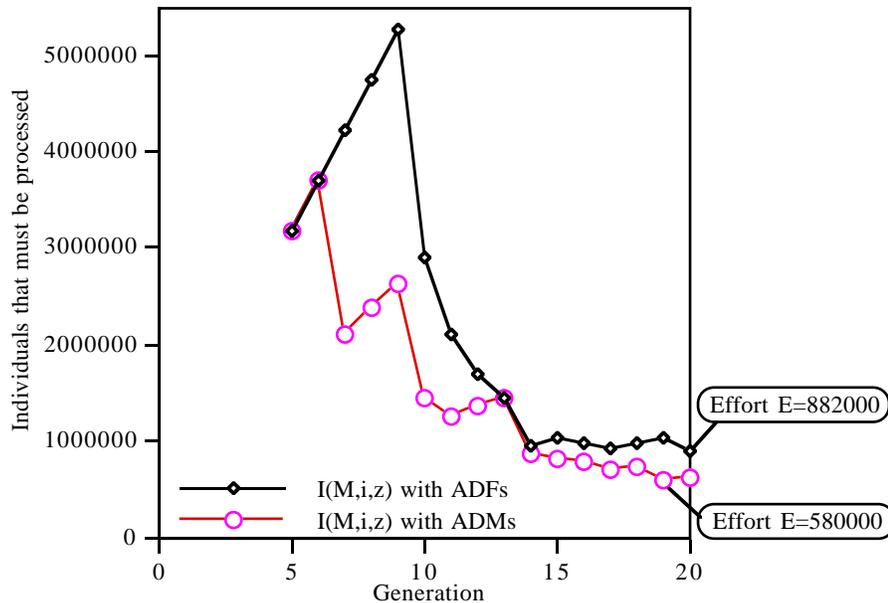
**Figure 7.6**  
 $P(M,i)$  for ADFs and ADMs on the wumpus world problem. Population size  $M=1,000$ , number of runs  $N=115$ .

most one of the four random worlds.

230 runs of a genetic programming system were performed on the wumpus world problem, half with ADFs and half with ADMs. The wumpus world simulator code provided by Russell and Norvig was used for fitness evaluation, with all parameters except for those described above left at their default values. As with DSOAR above, each program in each population had an architecture consisting of a result-producing branch and two modules (either ADF0 and ADF1 or ADM0 and ADM1). ADF0 and ADM0 each took 1 argument, and ADF1 and ADM1 each took 2 arguments. ADF1 and ADM1 could call ADF0 and ADM0, respectively, and result producing branches could call both of the automatically defined modules. The population size was 1,000 and the maximum number of generations per run was 21.<sup>11</sup> Tournament selection was used with a tournament size of 7.

The results are shown in Figures 7.6 and 7.7. Figure 7.6 shows  $P(M,i)$  for ADFs and for ADMs on this problem, while Figure 7.7 shows  $I(M,i,z)$ . It is evident from both of these graphs that ADMs provide a greater benefit. The computational effort is 882,000 for ADFs

<sup>11</sup>The lower number of generations was used because the wumpus world simulator is quite slow, and because the difficulty of the problem suggested that a large number of runs should be performed.



**Figure 7.7**  
 $I(M,i,z)$  for ADFs and ADMs on the wumpus world problem. Population size  $M=1,000$ , number of runs  $N=115$ , desired probability of success  $z=99\%$ .

and 580,000 for ADMs.

The evolved agent programs appear to be reasonably robust. It should be noted that a large number of deaths are to be expected even for the most intelligent agents in this environment. For example, suppose that an agent is born into a square with a breeze. The agent does not yet have any other information about neighboring squares, and therefore could not possibly infer the direction of the pit. The first move in such situations will be deadly at least 25% of the time regardless of the agent's strategy. Even at later points in a simulation, when the agent has more knowledge, it may be necessary to risk death in order to navigate to the gold. And as noted above, the gold may in some cases be unreachable.

Russell and Norvig provide the code for a single wumpus world agent in their code. Although they call it "stupid-wumpus-agent" it is considerably better than random. It will consistently grab the gold when it sees it, turn around when it bumps into something, and attempt (poorly) to avoid pits and the wumpus. In 1,000 test runs on random worlds stupid-wumpus-agent managed to stay alive 158 times and to grab the gold 22 times. By comparison, one of the agents evolved with ADMs, chosen arbitrarily from the set of successful best-of-run programs, managed to stay alive 479 times and to grab the gold 64 times.

## 7.10 Future Work

With respect to wumpus world, it should be noted that while many of the successful programs appeared to use complex, memory-based strategies, others used simple, reactive, memory-free strategies. This might indicate that the success criterion was too weak; perhaps the successful reactive programs were “lucky” in the selection of random worlds, and would show their weaknesses if subjected to more fitness cases. On the other hand, it may be the case that the “knowledge and reasoning” demands of the wumpus world are less than expected. Further experiments and analysis will be required to sort this out.

Several avenues for future work remain in applying ADMs to additional problems, and in refining our understanding of the types of problems for which ADMs will be useful. Experiments with combinations of ADFs and ADMs, and with the simultaneous evolution of programs and their macro-extended architectures (the number of ADF/Ms and the number of arguments that each takes), should also help to shed light on the utility of ADMs.

Even when ADMs decrease the number of individuals that must be processed to solve a problem, the runtime costs of ADMs may cancel any savings in problem solving time. Such costs include the time spent on redundant re-evaluation of purely functional code fragments, and, depending on the techniques used to implement ADMs, macro-expansion costs. Further studies must be conducted on the trade-offs involved.

The ADMs considered in this chapter are all substitution macros, but it should also be possible to evolve more powerful code transforming operators. In addition, macros are useful in more ways than were sketched above; for example, they can be used to establish variable bindings or, like `setf` in Common Lisp, to implement “generalized variables.” These uses of macros, and the benefits of richer macro definition facilities, should also be explored.

## 7.11 Conclusions

The human programmer’s toolkit includes several module-building tools, each of which can be useful in certain circumstances. Genetic programming systems should have access to a similar toolkit. In particular, they should have access to macro-definition facilities so that they can evolve control structures appropriate to particular problems. Automatically defined macros (ADMs) can improve the performance of a genetic programming system, but more work must be done to refine our understanding of the conditions under which ADMs, or combinations of ADFs and ADMs, are likely to be helpful. One heuristic, consistent with the experiments reported in this chapter, is that ADMs are likely to be useful in environments within which context-sensitive or side-effect-producing operators play important roles. Another heuristic, also consistent with the experiments reported in this chapter, is that ADMs are likely to be useful in domains that have traditionally made use of exotic control structures; for example, intelligent action systems. Perhaps the best strategy,

given sufficient computational resources, is to simultaneously evolve programs and their macro-extended architectures (the number of ADF/Ms and the number of arguments that each takes). This strategy is currently under investigation.

## Acknowledgments

The idea for this work emerged from a conversation with William E. Doane in which he suggested that we simultaneously evolve chromosomes and protein expression mechanisms. The comments of anonymous reviewers, Peter Angeline, Frederic Gruau, Thomas Haynes, and John Koza lead to several significant improvements in this work.

## Bibliography

- Angeline, P.J. (1994) "Genetic Programming and Emergent Intelligence." In K.E. Kinnear Jr., Ed., *Advances in Genetic Programming*, pp. 75–97. Cambridge, MA: The MIT Press.
- Angeline, P.J., and J.B. Pollack (1992) "The evolutionary induction of subroutines." In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum.
- Dean, T.L., and M.P. Wellman (1991) *Planning and Control*. San Mateo, CA: Morgan Kaufmann Publishers.
- Graham, P. (1994) *On Lisp: Advanced Techniques for Common Lisp*. Englewood Cliffs, NJ: Prentice Hall.
- Iba, H., T. Karita, H. de Garis, and T. Sato (1993) "System Identification Using Structured Genetic Algorithms." In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, edited by S. Forrest, pp. 279–286. San Francisco: Morgan Kaufmann Publishers, Inc.
- Kernighan, B.W., and D.M. Ritchie (1988) *The C Programming Language*. Second Edition. Englewood Cliffs, NJ: Prentice Hall.
- Kinnear, K.E. Jr. (1994) "Alternatives in Automatic Function Definition: A Comparison of Performance." In K.E. Kinnear Jr., Ed., *Advances in Genetic Programming*, pp. 119–141. Cambridge, MA: The MIT Press.
- Koza, J.R. (1992) *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.
- Koza, J.R. (1994a) *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.
- Koza, J.R. (1994b) "Architecture-altering Operations for Evolving the Architecture of a Multi-part Program in Genetic Programming." Computer Science Department, Stanford University. CS-TR-94-1528.
- Rosca, J.P., and D.H. Ballard (1995) "Causality in Genetic Programming." In *Proceedings of the Sixth International Conference on Genetic Algorithms*, edited by L.J. Eshelman, pp. 256–263. San Francisco: Morgan Kaufmann Publishers, Inc.
- Russell, S.J., and P. Norvig (1995) *Artificial Intelligence, A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.
- Steele, G.L. Jr. (1990) *Common Lisp*. Second Edition. Digital Press.
- Teller, A. (1994) "The Evolution of Mental Models." In K.E. Kinnear Jr., Ed., *Advances in Genetic Programming*, pp. 199–219. Cambridge, MA: The MIT Press.
- Zongker, D., and B. Punch (1995) *lil-gp 1.0 User's Manual*. Available on the World-Wide Web at <http://isl.cps.msu.edu/GA/software/lil-gp>, or via anonymous FTP to [isl.cps.msu.edu](ftp://isl.cps.msu.edu), in the directory "/pub/GA/lilgp".