Two-Way Stack Automata and the Ginsburg, Greibach, and Harrison Compiler

Dick Grune
dick@dickgrune.com

May 1, 2012

1 Introduction

Basically, a compiler consists of a lexical part, a syntactic part, and a semantic part. Each has a theoretical foundation and a practical representation:

component	foundation	representation
lexical	finite state automaton	regular expressions
syntactic	push-down automaton	grammars
semantic	?	attribute grammars, ad-hoc programming

The hegemony of the finite state automaton and the push-down automaton is unchallenged, but there is no obvious candidate for the automaton in the box with the question mark, although there are several contenders. In their paper *Stack Automata and Compiling*[1] (abbreviated here *SAaC*), Ginsburg, Greibach, and Harrison propose for that role the "stack automaton", more in particular the "two-way deterministic stack automaton", abbreviated here as *2SA*.

In *SAaC* the authors claim that "many of the most sophisticated sets that have been intensively studied with respect to programming language theory are covered by the new model". The purpose of this text is to test that claim. If true, the 2SA has the advantage of probably being the simplest automaton capable of the job.

SAaC is a mathematically very complex paper, and contributes little to the practical understanding of the 2SA. The paper shows, however, how a compiler running on the 2SA could conceivably consult the symbol table of an Algol program. So a subgoal of this paper will be to learn to program the 2SA, and to try to obtain an understandable program for the 2SA which produces the example of Figure 1 in *SAaC*.

We first discuss the automaton itself and show that it can recognize recursive sets. Next we consider the commands we can give to the machine, and design a very simple but convenient programming language for it, embedded in C. Finally we survey the practical possibilities of the 2SA. In this survey we sketch algorithms to recognize some of the well-known non-CF sets; present three complete programs for the 2SA, one of which produces the example of Figure 1 in *SAaC*; sketch a program to do *sort* or *sort-unique* of strings of arbitrary length; and show how a calling graph can be build.

1.1 The 2SA

The 2SA is a push-down automaton with three extensions:

- the automaton can read the input forward and backward in steps of one position, and change direction at any moment;

[1] J. ACM, **14**(1967)1, 172-201

- the stack pointer, which in a push-down automaton always points to the top of the stack, can move to lower regions of the stack, where it can read, but not write;
- at each cycle the 2SA can output one or more symbols: it can perform a "transduction" from input to output (this feature is also sometimes available in push-down automata).

Since *SAaC* may not be easily accessible to everybody, we list here the most important particulars of the 2SA proposed there:

- Both the stack pointer and the input pointer point *between* symbols; the stack pointer addresses the symbol on its left, the input pointer addresses the symbol on its right.
- The input pointer can be positioned before the first input symbol and after the last; it cannot access the empty place after the last symbol.
- Modifying the stack is done with a single instruction, which replaces a single symbol Z on the top of the stack by a word of symbols w of length 0 or more. It requires the stack pointer to point to the top of the stack, and leaves it pointing to the new top of the stack. There are no separate push or pop instructions.
- Successful termination requires the input pointer to be positioned after the last input symbol.

The inspectable stack allows the 2SA to gather and store information about the input on the stack and then compare it to other segments further on in the input; this allows context checking.

The 2-way movable head allows the context checking to be performed random-access over the input; it also allows information to be discarded from the stack and to be reconstructed later. Fig. 1 in*SAaC* suggests that code generation should be possible as well.

1.2 How to use a 2SA

The two paragraphs above suggest some ways of using a 2SA to implement the semantics and context checking of a compiler, but when one tries to make these ideas more concrete, the view becomes murky soon.

SAaC gives an example of consulting a symbol table on the stack, but upon scrutiny it raises more questions than it answers:

- Where does the R_0 in line t_{37} suddenly come from? The type symbol R_0 was seen in line t_{20}, disappears in lines t_{21} to t_{36} (partly not shown), and then reappears in line t_{37}.
- How does one handle more than one variable of the same type? In the example each variable is identified by a specific symbol, optimistically numbered 0, but if the input program contains N **real** variables, we cannot have N different symbols $R_0 ... R_{N-1}$, for unbounded N. So each has to get a number, and that number cannot be put in the state, since it is unbounded.
- How does one handle block structure and other syntactic structure? No doubt the compiler running on the 2SA machine has all kinds of subroutines, for handling declarations, expressions, etc. Although much of this can be hidden in the state (see Section 3.3), at least the parsing of expressions will need to put subroutine return information on the stack. None of this is visible in the example, which allows assignments of the form `<idf> := <idf>` only.

In other words, the example is simplified below the level of being convincing, and other means are needed to convince ourselves of the suitability of the 2SA.

More in particular, checking context requires writable memory at least proportional in size to the input, to store information for later reference. But the 2SA is really cramped for such memory: it has an input pointer, of limited size; and a stack, unbounded indeed, but writable only at one end. So the main question will be "Where do we put the info?".

The text of *SAaC* is of some help in this respect: it proves that the 2SA is capable of recognizing recursive sets, and since the Algol programs (or programs in any programming languages, for that matter) form a recursive set, the 2SA is, in principle, capable of recognizing Algol programs. (The set of syntactically and contextually correct Algol programs is a recursive set, since there is a decision procedure to see if a string is in that set: the Algol compiler. The set of *dynamically* correct Algol programs is not a recursive set.)

We will first look at the proof, then see how the 2SA can be programmed more or less conveniently, and finally show some programs of increasing complexity to run on the 2SA, one of which produces the sample run from *SAaC*.

2 Proof of Power

SAaC proves that a 2SA can recognize recursive sets by implementing a "linear bounded automaton" in it (Theorem 3.1 on page 186), and assumes recognition by a linear bounded automaton as a definition of recursive sets. Since we are examining the suitability of the 2SAs for compiler construction, we will take recognition by a context-sensitive (CS) grammar as the definition of recursive sets. We will first discuss a simple deterministic recognizer for CS languages, and then show how to implement it on a 2SA.

Needless to say, this will not result in a practical compiler for even the simplest programming language, for three reasons: a CS grammar is a clumsy and unwieldy object; ac ompiler is much more than a recognizer; and the implementation of it on a 2SA is extremely cumbersome and inefficient. But for a proof of power it will (have to) suffice.

2.1 A simple deterministic recognizer for CS languages

CS languages are produced by CS grammars, and a CS grammar is a formal grammar with rewriting rules of the form $P \rightarrow Q$ where P cannot be longer than Q. So `aB` $\rightarrow$ `Ba` is OK, but `aB` $\rightarrow$ `B` is not.

This restriction that no production rule can make the intermediate form shorter gives us immediately a trivial recognizer for CS languages. Given a CS grammar and an input w of length k, produce all intermediate forms of length k; if at least one equals w, w is in the language. If none does, w is not in the language: continuing with longer intermediate forms is pointless, since they will never get shorter again.

2.1.1 The basic recognizer

We will use a slightly more intelligent recognizer. This recognizer starts with the word w, and scans it from left to right until it finds the right-hand side of a rule from the grammar. It then replaces this right-hand side by the left-hand side of the rule. If the left-hand side is smaller than the right-hand side, the recognizer pulls up the rest of the input word; if it is the same size it does nothing; and it cannot be larger. Finally it moves to the left far enough not to miss any matching possibilities that might have arisen through the replacement, and continues its recognition process until no more matches are found. If by doing so it reduces the input word to the start symbol of the grammar, w was in the language; otherwise it was not.

This rough explanation glosses over several questions: how to find a match; what to do if there are multiple matches; and what to do if more than one rule has the same right-hand side. And of course we want answers to these questions that are compliant with the austere memory supply of the 2SA.

Matching all right-hand sides of all grammar rules can be done in parallel by keeping a counter for each right-hand side indicating how much of it has been matched up to here; since there are a finite number of rules and a finite number of positions in their right-hand sides, this requires a finite amount of memory, which can be stored in a single state. This is in fact a finite-state recognizer, as used in the compiler construction programs *lex* and *flex*, in virus scanners, etc.

This information is, however, invalidated by a replacement action. There are two options here: go back to the beginning of the input and reconstruct the information; or attach the information to each position in the input.

2.1.2 Making it deterministic

The multiple matches and rules with the same right-hand sides are actually the same problem. Again there are two solutions: non-deterministically, by copying a recognizer for each possibility; and deterministically, by backtracking search. Since a non-deterministic compiler is very unsatisfactory, we have to go for the deterministic solution.

When multiple courses of action present themselves – and these do include the choice to not recognize anything in the present position and search on elsewhere – the choices are put in a list attached at the input position; the first choice is marked as done; its replacement is performed; and the search continues. Now if the recognizer reaches the end of the input without having reduced it to the start symbol, it moves to the left to find the first symbol marked with an unfinished list of choices. The latest replacement is undone, the next choice in the list is marked done, its replacement is performed, etc. This implements full backtracking search.

Again a problem has been glossed over: how to do the undoing. This is a problem since unlike a replacement, undoing a replacement can require more room in the input than is available. This is the case if the right-hand side is longer than the left-hand side.

Two solutions are possible: a difficult one, and a clumsy one. In the difficult one, we push the rest of the input to the right to make room. In the clumsy one, when replacing a longer string by a smaller, we mark the positions left empty by a special symbol, "void". Then when undoing the replacement the room will still be there. Such void symbols will have to be skipped by all other operations. It is clear that both can be implemented with at most $O(|w|)$ memory.

2.1.3 An example

The recognition of the word *aaabbbcc* can serve as an example. It is generated by the CS grammar $G1$:

1.	S	$\rightarrow$	abc	
2.	S	$\rightarrow$	aSQ	(generate $a^n abcQ^n$ for $n \geq 0$)
3.	cQ	$\rightarrow$	Qc	(move Qs to the left towards the last b)
4.	bQc	$\rightarrow$	$bbcc$	(change Q into bc upon arrival)

which generates the language $a^n b^n c^n$ for $n \geq 1$ (when all Qs are gone the result is $a^n abb^n c^n c, n \geq 0$, which is equal to $a^n b^n c^n, n \geq 1$). The word *aaabbbccc* is recognized in the following steps:

$aaab[bbcc]c$	(rule 4)	$\rightarrow$
$aaabb[Qc]c$	(rule 3)	$\rightarrow$
$aaabbc[Qc]$	(rule 3)	$\rightarrow$
$aaa[bbcc]Q$	(rule 4)	$\rightarrow$
$aaab[Qc]Q$	(rule 3)	$\rightarrow$
$aa[abc]QQ$	(rule 1)	$\rightarrow$
$a[aSQ]Q$	(rule 2)	$\rightarrow$
$[aSQ]$	(rule 2)	$\rightarrow$
S		

where the segment between the brackets shows the matching right-hand side. So, using **B** as a separator, the word

$$\mathbf{B}aaabbbccc\mathbf{B}aaabbQcc\mathbf{B}aaabbcQc\mathbf{B}aaabbccQ\mathbf{B}aaabQcQ\mathbf{B}aaabcQQ\mathbf{B}aaSQQ\mathbf{B}aSQ\mathbf{B}S\mathbf{B}$$

is proof that *aaabbbccc* is a word in the language generated by $G1$.

2.2 Copying a stack segment on the 2SA

Implementing the above recognizer is based on an algorithm to duplicate a segment of the entries on the top of the stack to the top of the stack, provided that this segment is not longer than the input; the remarkable thing is that the actual input is immaterial in the algorithm, just its length matters. We follow the text of *SAaC*, page 188, here.

Suppose we have a word $u = u_1...u_k$ on the top of the stack, demarcated by **B**s, and an input $a_1...a_m$ with $k \leq m$, demarcated by ¢ and \$:

$$\mathbf{B}u_1...u_k\mathbf{B}\rceil \qquad\qquad \lceil ¢a_1...a_m\$$$

Here $\rceil$ is the stack pointer, pointing to the symbol on its left, and $\lceil$ is the input pointer, pointing to the symbol on its right, as in *SAaC*.

First we mark the top of the stack by a special symbol, $\mathbf{B}'$, by pushing it on the stack:

$$\mathbf{B}u_1u_2...u_k\mathbf{B}\mathbf{B}'\rceil \qquad\qquad \lceil ¢a_1...a_m\$$$

(Officially we have to replace **B** by $\mathbf{B}\mathbf{B}'$, since the 2SA as described in*SAaC* allows only the replacement of a symbol Z on top of the stack by a possibly empty word w, but we will allow also simply pushing and popping.)

Now we move the stack pointer to the left, to the second **B**, and one step to the right. This positions it pointing to u_1:

$$\mathbf{B}u_1\rceil u_2...u_k\mathbf{B}\mathbf{B}' \qquad\qquad \lceil ¢a_1...a_m\$$$

and we are ready for the first round of the main loop of our copying operation.

Start of main loop

Suppose we have already copied $u_1...u_{i-1}$, and the stack pointer points at u_i (if the stack pointer points at the separating **B** instead we have copied all of u and we are (almost) done; see below):

$$\mathbf{B}u_1u_2...u_{i-1}u_i\rceil u_{i+1}...u_k\mathbf{B}u_1...u_{i-1}\mathbf{B}' \qquad\qquad \lceil ¢a_1...a_m\$$$

(for $i = 1$ we have the situation above). We absorb the symbol u_i in the state of the 2SA, move to the marker $\mathbf{B}'$, deposit the symbol absorbed in the state (u_i) to the left of it by replacing $\mathbf{B}'$ by $u_i\mathbf{B}'$, and move left over the $\mathbf{B}'$:[2]

$$\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\rceil\mathbf{B}' \qquad\qquad \lceil ¢a_1...a_m\$$$

Now we have copied u_i, but we have also created a problem: how do we find u_{i+1}? This is where the input pointer comes in: we move simultaneously the stack pointer left and the input pointer right until the stack pointer points at the **B**:

$$\begin{array}{ll}
\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\rceil\mathbf{B}' & \lceil ¢a_1...a_{i-1}a_ia_{i+1}...a_m\$ \\
\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}\rceil u_i\mathbf{B}' & ¢\lceil a_1...a_{i-1}a_ia_{i+1}...a_m\$ \\
... & ... \\
\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1\rceil...u_{i-1}u_i\mathbf{B}' & ¢a_1...\lceil a_{i-1}a_ia_{i+1}...a_m\$ \\
\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}\rceil u_1...u_{i-1}u_i\mathbf{B}' & ¢a_1...a_{i-1}\lceil a_ia_{i+1}...a_m\$
\end{array}$$

Now both the stack pointer and the input pointer have moved i positions, and i has been recorded in the position of the input pointer. This can be done because the length of the stack segment u is not larger than the length of the input string w. (Note that there is a one-off error in the explanation on page 188 of *SAaC*: if one goes from $t\mathbf{B}u\mathbf{B}V_1...V_i\mathbf{B}'\rceil$ to $t\mathbf{B}u\mathbf{B}\rceil V_1...V_i\mathbf{B}'$, the stack pointer has moved $i+1$ positions, not i.)

Now the road ahead is clear: first we move the stack pointer to the left to the second **B** and then right one position,[3] all the time leaving the input pointer where it is. The stack pointer now points to u_1:

[2] see Note at end of section

[3] see Note at end of section

$$\mathbf{B}u_1\rceil u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\mathbf{B}' \qquad ¢a_1...a_{i-1}\lceil a_ia_{i+1}...a_m\$$$

The last step is to move the input pointer left and the stack pointer right in tandem, until the input pointer is back at the beginning:

$$\begin{array}{ll}
\mathbf{B}u_1\rceil u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\mathbf{B}' & ¢a_1...a_{i-1}\lceil a_ia_{i+1}...a_m\$\\
\mathbf{B}u_1u_2\rceil ...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\mathbf{B}' & ¢a_1...\lceil a_{i-1}a_ia_{i+1}...a_m\$\\
... & ...\\
\mathbf{B}u_1u_2...u_{i-1}u_i\rceil u_{i+1}...u_k\mathbf{B}u_1...u_{i-1}u_i\mathbf{B}' & ¢\lceil a_1...a_{i-1}a_ia_{i+1}...a_m\$\\
\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}\rceil ...u_k\mathbf{B}u_1...u_{i-1}u_i\mathbf{B}' & \lceil ¢a_1...a_{i-1}a_ia_{i+1}...a_m\$
\end{array}$$

Since the input pointer has moved i positions, the stack pointer has too, and since it started at u_1 it now points at u_{i+1}. We now return to the start of the copy loop, marked in the margin above.

We continue to run this loop until we are about to copy the separating $\mathbf{B}$:

$$\mathbf{B}u_1u_2...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}\rceil u_1...u_{i-1}u_iu_{i+1}...u_k\mathbf{B}' \qquad \lceil ¢a_1...a_{i-1}a_ia_{i+1}...a_m\$$$

Then we change the auxiliary marker $\mathbf{B}'$ to $\mathbf{B}$:

$$\mathbf{B}u_1...u_k\mathbf{B}\rceil u_1...u_k\mathbf{B} \qquad \lceil ¢a_1...a_m\$$$

and the job is done! The whole process can be viewed in detail in Appendix B in the Appendix file.

Note: A slightly more elegant algorithm is obtained by omitting the left move of the stack pointer over $\mathbf{B}'$ just after the replacement of $\mathbf{B}'$ by $u_i\mathbf{B}'$; both stack and input pointers then move $i+1$ positions. This allows omitting the corresponding right move of the stack pointer to position it at u_1 above; it stays at $\mathbf{B}$ (virtually u_0) and the subsequent move over $i+1$ positions brings it to u_{i+1}.

The algorithm is more elegant for several reasons:

- it avoids useless moves;
- the transition from i to $i+1$ occurs automatically and naturally by stacking the copied u_i;
- the algorithm works correctly as long as $|u| \leq |a|$ (or $k \leq m$) rather than as long as $|u| \leq |a|+1$.

2.3 Implementing the simple deterministic recognizer on the 2SA

Recognition occurs by rewriting the input string. Since the 2SA cannot modify the input, we first transfer the input $a_1...a_m$ to the stack, properly demarcated:

$$\mathbf{B}a_1...a_m\mathbf{B}\rceil \qquad \lceil ¢a_1...a_m\$$$

Using the algorithm from Section 2.2 we copy $a_1...a_i$ until we find a right-hand side of a grammar rule, say $A_1...A_p \to B_1...B_q$, with $p \leq q$. We now back up q places, remove $a_{i-q+1}...a_i = B_1...B_q$ from the copy, copy in $A_1...A_p$ instead:

$$\mathbf{B}a_1...a_m\mathbf{B}a_1...a_{i-q}A_1...A_pa_{i+1}...a_m\mathbf{B}'\rceil \qquad \lceil ¢a_1...a_m\$$$

and continue our algorithm. This possible because the length of the segment to be copied cannot be increased by the replacement.

Continuing this way, we produce on the stack a proof that $a_1...a_m$ belongs to the language, by reducing it to the start symbol S in the same way as shown at the end of Section 2.1:

$$\mathbf{B}a_1...a_m\mathbf{B}a_1...a_{i-q}A_1...A_pa_{i+1}...a_m\mathbf{B} \quad ... \quad \mathbf{B}a_1Q_1a_m\mathbf{B}S\mathbf{B}\rceil$$

All this is a lot easier said than done: recognizing the right-hand side requires a finite-state automaton embedded in the state space of the 2SA; the backing up and subsequent insertion of a different string confuses both the copy and the recognition mechanism; and multiple matches must still be handled by full backtracking search. It is true that the authors avoid the complication of the changing length in the replacement of $a_{i-q+1}...a_i$ by $A_1...A_p$ when $p < q$, by using a "linear bounded automaton" rather than a CS grammar to characterize the recursive sets, but enough trouble remains.

The accumulation of complications leads to what *SAaC* calls: "quite formidable mathematical constructions" (page 173), and the bulk of the paper is dedicated to mastering them. The amount of detail is overwhelming, and one can only feel considerable admiration for the authors for having been able to pull it off.

The above describes a parser and context checker, but a compiler is more than that: it usually maintains complicated data structures, and it is far from clear how to manage them in the confines of a 2SA. So for the moment it is still to be seen if the authors of*SAaC* were justified in using "Stack Automata" and "Compiling" in the same title.

3 A Control Language for the 2SA

As a computer, the 2SA leaves much to be desired. It has two variables, the input pointer and the stack pointer, and two memory units: the input, readable everywhere and unwritable; and the stack, readable everywhere and writable at one end. Its only instruction is the "transition", described by an 8-parameter δ function

$$\delta(q, a, Z) = (d, q', e, w, \Omega),$$

where q is the old state; a is the symbol to the right of the input pointer; Z is the symbol to the left of the stack pointer; d is the input pointer movement (left, right, or none); q' is the new state; e is the stack pointer movement (left, right, or none); w is the replacement for the top element of the stack; and Ω is the output. We shall call the part before the = sign the "condition"; the part after it the "action"; and the whole a "rule".

There is quite a distance between this model and a reasonably usable coding language:

- the machine lacks all structure: no grouping mechanism is available, or even suggests itself;
- the algorithm in Section 2.2 shows that the state will play a complex role, but the 2SA's state space is just a set of unstructured symbols, with one operation only: testing for equality;
- there are no subroutines and no parameters;
- many conditions may lead to the same action; as it stand, they all have to be specified separately;
- the condition-action paradigm provides an **if** ... **then** but there is no **else**, often requiring many rules to specify a default;
- the definition of the 2SA requires the specification of a rewrite of the top of the stack on each and every transition, which is unnatural and very inconvenient.

Based on the requirements of the 2SA and considerations of ease of coding and ease of implementing, the following design was chosen.

3.1 The coding language

Programs for the 2SA are written in restricted C and a macro-supported subset of C. For example, a rule

$$\delta(q, a, Z) = (0, q', -1, w, X)$$

is coded as

```
State (q) {
    Rule(IP_at(a) && SP_at(Z),
        (IP_left(), replace(Z, w), output(X), go_to(q')));
}
```

where `State`, `Rule`, `IP_at`, `SP_at`, `IP_left`, `replace`, `output`, and `go_to` are predefined C functions or macros. The commands in the third line may be given in any order; the order presented here seems the most natural one. Duplicate or conflicting commands are detected by the system at run time, and result in an error message and termination of the program run.

2SA program commands

The 2SA is initialized by a call of `init_stack_automaton()`.

The input is specified by calls of `add_to_input(`s`)`, where s is the symbol to be added. The standard input demarcators ¢ and $ are added by the system.

The stack normally starts with only its bottom symbol Z_0 in place. For demonstration purposes a specific stack can be specified by calls of `add_to_stack(`s`)`; if this happens, the input demarcators ¢ and $ are no longer supplied by the system.

The 2SA is started by a call of `start_stack_automaton(`q_0`)`, where q_0 is the start state; for further details see the example files. It is terminated by a call of `go_to(`q_F`)`, where q_F is one of the built-in final states `OK` (for successful termination) and `KO` (for failure). When terminating in state `KO`, the output of the transition is interpreted as the error message. The requirement that the input pointer be positioned to the right of the last input symbol (*SAaC*, p. 179) for successful termination is not enforced.

2SA program structure

A program for the 2SA consists of a set of declarations of user symbols and variables (see Sections 3.2 and 3.7), the separator `BEGIN_CODE`, a sequence of state definitions like the one above, and the terminator `END_CODE`.

A minimal 2SA program might be

```
#include    "stack_aut.h"

static Symbol Smiley = "\\smiley";

int main(void) {
    init_stack_automaton();
    add_to_input(Smiley);
    start_stack_automaton(hello_1);
    return 0;
}

/* there is no user state */
int user_state_has_changed(void) {return 0;}
void print_user_state(void) {}

BEGIN_CODE

State(hello_1) {
    Rule(otherwise,
         (IP_right(), output("Hello world%s", Smiley), go_to(OK)));
}

END_CODE
```

It gives the output

t_0 1 Z_0⌉ ⌈¢☺$
t_1 OK Z_0⌉ ¢⌈☺$ Hello world☺

See the example files `*.c` and the `Makefile` for more detail.

3.2 State

In principle the state of an automaton corresponds to the instruction pointer of a CPU, but the example in Section 2.2 shows that symbols are absorbed in the state and can be retrieved from it. The nature and structure of these "user states" are different for each program.

For these reasons the state of the 2SA has been split in two parts: the "control state" of the 2SA, implemented as the address of the code segment to be performed for that state; and the "user state", implemented as zero or more C variables.

User state variables must be of types with finite domains, for example Booleans, enumeration types and `Symbol`s (see Section 3.7). But finite-range integers would also be acceptable, since there are only a finite number of operations on a finite number of operands with a finite number of results, so everything can be lawfully encoded in a finite set of states.

The user states can be used in the condition of the `Rule` command for testing, just like the input and stack pointers. And they can be set in the action part. All user variable handling is through C expressions.

The user has to supply two routines in connection with user states, `int user_state_has_changed(void)`, which should return `1` if any user state was changed in the latest transition, and `0` otherwise; and `void print_user_state(void)`, which should print the user state(s) without surrounding layout.

All condition-action rules with the same control state must be collected in a single `State` command in the code. An example is

```
State (q) {
    Rule(IP_at(a) && SP_at(Z),
        (IP_left(), replace(Z, w), output(X), go_to(q')));
    Rule(IP_at(Real_type) && type == 0,
        (IP_right(), type = Real_type, go_to(type_found_1)));
}
```

where `type` is a user state variable of type `Symbol` and `Real_type` is a user-defined constant of that type.

3.3 Subroutines

Subroutines are used in programming for two purposes: as a structuring device, and for recursion. A structured compiler algorithm for the assignment in Fig. 1 in *SAaC* could be

```
void assignment(void) {
    identifier();
    look_up_idf();
    becomes_symbol();
    identifier();
    look_up_idf();
    code_assignment();
    return;
}
```

which is greatly preferable to writing out all the specific instructions.

When used for structuring without recursion, the number of possible sequences of calls is bounded by $\Sigma_{i=1}^{P} i!$, where P is the number of subroutines in the program. This is a finite number, so all static stacks can be incorporated in the state of the 2SA.

Static stacks are made available with the commands `call_sub` and `end_sub`. A call of `call_sub`(q, q_1) stacks the state q_1 for later use and sets the new state to q; a call of `end_sub()` retrieves the most recently stacked state and makes it the new state. So `call_sub`(q, q_1) can be read as "first go to the subroutine that starts with state q, and when that subroutine is done continue in state q_1".

With these routines the above C procedure is coded as

```
State(assignment_1) {
    call_sub(identifier_1, assignment_2);
}

State(assignment_2) {
    call_sub(look_up_idf_1, assignment_3);
}
...
State(assignment_6) {
    call_sub(code_assignment_1, assignment_7);
}

State(assignment_7) {
    end_sub();
}
```

where the trailing numbers are positions in the subroutine.

The maximum static depth in controlled by the setting of `MAX_STATIC_STACK_SIZE` in the file `stack_aut.h`.

Subroutines used for *recursion* are a different matter altogether. Recursion can have unlimited depth, so it cannot be accommodated in the state, but must be stored on the stack. This requires the conversion from state to stack symbol(s) and vice versa. Since it is not required for producing Fig. 1 of *SAaC*, this feature has not been implemented.

As an aside, it can be noted that in producing the initial stack shown in Fig. 1, the compiler must have made several recursive calls, for example due to the recursive nature of blocks in Algol, but there is no trace of them in the figure.

3.4 Replication

There is no other way of accessing the symbols in the input and on the stack than by supplying them in the condition part of a rule, where they will be compared by the system. So picking up the type in line t_{20} in Fig. 1 requires a rule for each possible type:

```
State(look_up_idf_6) {
    /* identifier found */
    Rule(SP_at(Real_type),
         /* add type to state */
         (type = Real_type, go_to(look_up_idf_7)));
    Rule(SP_at(Integer_type),
         /* add type to state */
         (type = Integer_type, go_to(look_up_idf_7)));
    Rule(SP_at(Boolean_type),
         /* add type to state */
         (type = Boolean_type, go_to(look_up_idf_7)));
}
```

and the situation is (much) worse when comparing letters.

Such symbols come in sets, and could be declared as follows:

```
static Symbol types[] = {"R_0", "I_0", "B_0", 0};
```

A set declared this way can then be used in a `For_All` command:

```
State(look_up_idf_6) {
    /* identifier found */
    Symbol t;
    For_All (t, types) {
        Rule(SP_at(t),
             /* add type to state */
             (type = t, go_to(look_up_idf_7)));
    }
}
```

3.5 Specifying defaults

A convenient feature of imperative programming is that in a sequence

if A **then** X **else if** B **then** Y **else if** ...

subsequent conditions are only considered if all previous conditions failed. Guarded commands, in which the conditions are evaluated concurrently and an error or non-determinism occurs if more than one condition applies, have never become popular, Dijkstra notwithstanding, since they often require kludges like

if A **then** X ; **if not** A **and** B **then** Y ; ...

The 2SA presents a guarded command model, but in principle overlapping guards cannot occur: there are only three things to condition on: state, input symbol, and stack symbol, and each rule checks all three. But in coding we meet situations in which we want to do one thing if there is a certain symbol under the stack pointer, regardless of the input, but another, dependent on the input, otherwise. The traditional if-then-else_if construction does this naturally, but the guarded command construction does not.

For convenience of coding the rules are applied in textual order: the first one that matches is taken, and the remainder is ignored. So the above example can be coded as

```
Rule(SP_at(X),
    (....));
Rule(IP_at(Y),
    (....));
```

The condition `otherwise` can be used to catch all conditions not covered by the preceding rules. If no rule applies at run time, the 2SA gives an error message, and stops.

3.6 Pointless rewriting of the stack

The official definition requires a stack rewrite to be specified on each and every move: $\delta(q, a, Z) = (d, q', e, w, \Omega)$, where the symbol Z is on the top of the stack, and is rewritten to the string w. When $w = Z$ the write operation is not performed (*SAaC*, bottom of page 177), but Z must still be specified. This is very inconvenient.

The requirement for a stack overwrite rather than a stack write also makes it impossible to just push a string w: one has to tell the machine to erase Z and to push Zw, where Z is the symbol on top of the stack. This is very inconvenient when Z is immaterial or when the entry is applicable with several different symbols on the top of the stack.

Rewriting the stack is specified with the command `replace(`Z`,`w`)`. If Z is a symbol, it has to be equal to the top of the stack; it is then popped from the stack, and the string w is pushed on the stack. If Z is 0, the symbol string w is just pushed on the stack. Routines are available for making strings out of lists of symbols.

3.7 Symbols

Symbols in an automaton are in principle nominal values, since the only operations defined on them are copying and comparison for equality. For coding we also want to print them conveniently, and for this purpose the type `Symbol` has been defined as pointer to an read-only string: `const char *`; the string should be amenable to LaTeX math mode processing.

The symbols used by the 2SA are defined in the first section of the program as named `Symbol`s, for example

```
static Symbol End_of_decls = "Z_1";
static Symbol Aux_marker = "Z_2";
static Symbol Begin_idf_decl = "\\#";
static Symbol End_idf_decl = "\\beta";
```

It is convenient to specify the string only once and for the rest use the symbolic name only; a symbol is then identified by the string's machine address, and can be compared simply with `==`. Named elements of symbol strings require some trickery:

```
static Symbol types[] = {"R_0", "I_0", "B_0", 0};
#define Real_type       types[0]
#define Integer_type    types[1]
#define Boolean_type    types[2]
```

3.8 Output

Output from the 2SA is specified with the command `output(`fmt, s`)`, where fmt is a `printf` format, and s is a 2SA symbol.

Output from the program is a file of LaTeX table entries, with one line for each 2SA transition, in the following format:

$$t_{95} \quad 7,20 \quad I_0 \quad Z_0R_0\#XY\beta I_0\#JK\beta B_0\#IK\beta Z_1R_0 := \uparrow \quad ...;XY := JK\uparrow;... \quad \text{LOAD } I_0$$

The columns show the time, the static state stack, the user state(s), the stack with pointer, the input with pointer, and the output of the transition. The file can be typeset in the environment

```
\begin{longtable}{l l l l l l}
...
\end{longtable}
```

3.9 The implementation

The syntax of the coding language is implemented with one typedef and four macros:

```
typedef const char *Symbol;

#define State(st)             static void st(void)
#define Rule(cond, action)    {if (cond) {action; return;}}
#define For_All(s,ss)         for (s = ss[0]; s; s = _next_For(s, ss))
#define otherwise             (1)
```

It is the shortest compiler I have ever written.

The semantics of the coding language is implemented in the file `stack_aut.c`. The only noteworthy item is the routine `_next_For(`s, ss`)`, used in the for-loop in the `For_All` macro. Whereas normal for-loops over a list manipulate a pointer in the list, this one manipulates the elements in the list: it looks up the symbol s in the list ss and returns the next element. This can be done reliably because the lists are actually sets and cannot contain duplicates (and if they contain duplicates anyway, their addresses will differ).

3.10 A meta-implementation

The above trivial compiler generates code for a C-machine from a 2SA program, but purists might demand code for the 2SA proper, i.e., a complete specification for the transition function δ. Producing it seems possible, but would require considerable analysis.

- The state must be condensed into a single item (`int` or `enum`). The calling graph for the states is actually a calling tree, so all paths can be enumerated; they form the new states. The user states are represented by n `Symbol`s. Each can be present in or absent from the state; this gives 2^n combinations, each of which is combined with each of the states enumerated above.
- Conditions not stated explicitly in the rules but covered by the sequential application of the rules must be determined, and pertinent rules generated for them. This can be done since all symbols are known from the declaration section of the program.

When all this is done, probably some weeding out of inaccessible transitions will be possible.

3.11 Further developments

The following features would ease the programming of the 2SA considerably. They can, however, not be implemented by a few simple macro definitions and require a (small) compiler.

3.11.1 Composite input and stack symbols

In all the above examples stack symbols are of the same data type as input symbols, but that need not be so, nor does either need to be restricted to simple strings. The input symbols as supplied by a lexical analyser are usually pairs of a string representation and a class representation. Stack symbols could be very complex, as long as their set is finite; this may allow storing a lot of (bounded) information in a few stack entries.

The system should differentiate between input symbols and stack symbols, and allow the user to define both their types.

3.11.2 Compilation of static procedures

The transformation from

```
void assignment(void) {
    identifier();
    ...
    code_assignment();
    return;
}
```

to

```
State(assignment_1) {
    call_sub(identifier_1, assignment_2);
}
...
State(assignment_6) {
    call_sub(code_assignment_1, assignment_7);
}

State(assignment_7) {
    end_sub();
}
```

should be performed automatically.

With this feature present, the sort routine from Section 4.5 can be written almost completely as a C program, with only the most primitive operations expressed in `Rule`s and `State`s.

3.11.3 Features for recursion

This requires routines in the following vein. One might be `void call_proc(State s, State p)`, which converts the control state `p` into one or more stack symbols, pushes them on the stack with proper demarcation, and then `go_to`s to state `s`.

Another might be `void end_proc(void)`, which retrieves the stacked control state from the stack, and then `go_to`s to it.

Users will have to stack and retrieve user states where appropiate.

3.11.4 Better checking

Although it is very convenient for all symbols to be identified by their addresses, it is also error-prone: `"A" == "A"` yields *false*. Checking this property is possible and would be worth-while.

4 Examples of 2SA programs

First we show how to recognize some well-known non-CF sets with a 2SA. This is followed by three implemented 2SA programs, with their code. The code is in the present directory; the results are in appendices in the file `Appendix.pdf`. The section is closed by sketches of a string sort program and code to produce a calling graph in a compiler.

4.1 The usual suspects

The set $a^n b^n c^n$ is trivial to recognize: push a^n on the stack, scan b^n while counting off as on the stack, and repeat for c^n.

The set ww is a bit more challenging, as is any matching in which the middle has to be found. Shift through he input and for each pair of input symbols put one marker on the stack; from the end back up, deleting a stack marker for each input symbol; shift right, copying the input symbols to the stack; back up to the beginning, and check if the input consists of twice the stack contents.

Recognizing a^{2^n} is easy again: copy the input to the stack, demarcated by **B**s, and using the algorithm from Section 2.2 copy the stack to the top of the stack, skipping every second a; repeat until failure. If this reduces the stack contents to $\mathbf{B}a\mathbf{B}$, the input was OK.

4.2 Checking integer multiplication

The first 2SA program checks the correctness of inputs of the form $1^a \times 1^b = 1^{a \times b}$.

First the multiplier (1^a) is put on the stack as $*^a$, and the input pointer is moved to the right of the =-sign. We consider the invariant $(a - S) \times b = R$, where S is the number of $*$s on the stack, and R is the distance between the input pointer and the =-sign. At this point the invariant holds: $(a - a) \times b = 0$.

We now enter a loop in which we first unstack one $*$. This invalidates the invariant, and to restore it we have to increase R by b, which we do as follows.

1. We move the input pointer left to the right of the =-sign, and for each 1 we move over, we push one s on the stack, R in total.

2. We move the input pointer left to the right of the $\times$-sign, and for each 1 we move over, we push one s on the stack, b in total. Now there are $R + b$ s-s on the stack.

3. We move the input pointer right to the right of the =-sign.

4. We pop all the s-s, and for each s popped we move the input pointer one position to the right.

Now the invariant has been restored, and there is one $*$ less on the stack.

When all $*$s are gone, $S = 0$, and the invariant reads $a \times b = R$. If the input pointer is now at the \$, the input is OK; otherwise it is not.

The code is in the file `mult.c`; a sample output is in Appendix A.

The program does not use the full power of the 2SA: the interior of the stack is never examined. This shows that the problem can also be solved on a 2-way push-down automaton.

4.3 Copying a stack segment

The stack segment copy operation described in Section 2.2. The code is in the file `copy.c`; a sample output is in Appendix B.

4.4 Consulting the symbol table of an Algol program

The point of the entire exercise: reproducing Fig. 1 from *SAaC*. The code is in the file `ggh.c`; the output is in Appendix C. It has 20% more transitions than Fig. 1; these are due to the state changes from static routine calls and returns.

4.5 Sorting

One operation that is often used in programming is finding the next element in a list. Normally this is implemented with a simple index or pointer, of which any programming language has an unlimited supply. The 2SA, however, has only two pointers, each with restrictions. Processing an element usually requires both these pointers, so when the processing of that element is over, its position is lost. And without that position we cannot find its successor.

The remedy is to process the elements in some testable order, keeping the element being processed in an identifiable place on the stack. When the next element is required, we stack a "sentinel", an element larger than any in the list; now the stack has $...e_1...e_2$, where e_1 is the old element and e_2 is the sentinel. We scan the list, which is assumed to reside in the input, for pertinent elements, and each time when we find an element e_3 such that $e_1 < e_3 < e_2$, we unstack the top element e_2 and stack e_3 instead. When we reach the end of the list in the input, the top element will be the next element after e_1, or, if the sentinel is still there, we have exhausted the list.

As an application of this technique and as an example of a "real" application, we will now sketch a *sort* program, which accepts a list of words separated by spaces (_) as input, and outputs the words in lexicographical order. So for the input `who_laughs_last_laughs_best`[4] it produces the output `b e s t _ l a s t _ l a u g h s _ l a u g h s _ w h o` (each letter being output separately).

The approach is the following:

1. Push the empty string, demarcated by **B**s, on the stack, and move the input pointer to the first letter of the first word.

2. Push a sentinel, a word larger than any word, on the stack, which now has the form $\mathbf{B}w_1\mathbf{B}w_2\mathbf{B}$. Consider the two-part invariant "(1) w_1 is the last word processed; (2) w_2 is the smallest word larger than w_1 that we have seen between the beginning of the input and the input pointer". It holds at this point.

3. While the input pointer is not at the end of the input, repeat the following step.

   (a) Read the next word w_3 in the input, all the while comparing it to w_1 and w_2. If it is lexicographically between w_1 and w_2, unstack w_2 and stack w_3 instead.

   (b) Move the input pointer to the beginning of the next word if present, or to the end of the input otherwise. This updates part (2) of the invariant.

[4] Dutch proverb

If the sentinel is still on the top of the stack, there are no more words to be output, and we are done. Otherwise, the word on the top of the stack w_2 is the next lexicographically larger word after w_1; continue with Step 4.

4. Position the input pointer to beginning of the first word.
5. While the input pointer is not at the end of the input, repeat the following steps.
   (a) Read the next word in the input, and if it is equal to w_2, output it.
   (b) Move the input pointer to the beginning of the next word if present, or to the end of the input otherwise.

   This updates part (1) of the invariant.
6. Go to step 2. (Stacking the sentinel will make the w_2 just processed the new w_1).

It is clear that each operation used in the above description can be implemented fairly simply on the 2SA.

At the end of the algorithm, the stack contains the sort-unique of the input.

A small variation of the algorithm can recognize input consisting of unsorted words followed by the same words in sorted order:

```
who_laughs_last_laughs_best=best_last_laughs_laughs_who
```

or unsorted words followed by single copies of the words sorted (sort-unique):

```
who_laughs_last_laughs_best=best_last_laughs_who
```

4.6 Constructing a calling graph

Next to the symbol table, the calling graph is an important component in a compiler. We will sketch here, in even less detail than for the sorting program, how to create one; consulting it is then trivial.

We assume the input consists of (or can be viewed as consisting of) a sequence of procedure names, each with the names of the procedures it calls directly:[5]

Input: ⌈¢$horse(cat, dog)pig(horse)cat(pig, cow)dog()cow()$\$

where *horse* calls *cat* and *dog* directly, and *dog* calls no other procedures.

The calling graph will be a sequence of procedure names, each followed by the names of all procedures it calls directly or indirectly. We will call the procedure the "caller" and the called procedures the "callees".

The algorithm goes as follows:

1. Stack an empty caller name, with an empty callee list, demarcated.

   Input: ⌈¢$horse(cat, dog)pig(horse)cat(pig, cow)dog()cow()$\$
   Stack: **B**()**B**#

2. Stack the sentinel caller name, and find the smallest caller name between the one on the top of the stack, and the one next below, using the technique from Section 4.5. If a caller name is found, copy it to stack together with all its callees; otherwise we have processed all callers and are done.

   Input: ¢⌈$horse(cat, dog)pig(horse)cat(pig, cow⌈)dog()cow()$\$
   Stack: **B**()**B**$cat(pig, cow⌉$#

3. Move the stack pointer to the first callee:

[5] This example represents the program from Figure 1.28 in "Modern Compiler Design", by Dick Grune et al.

Input: ¢↾$horse(cat, dog)pig(horse)cat(pig, cow)dog()cow()$$
Stack: $\mathbf{B}()\mathbf{B}cat(pig↿, cow\#$

4. Find in the input the caller with the name the stack pointer is pointing to:

   Input: ¢$horse(cat, dog)pig↾(horse)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig↿, cow\#$

5. For each of its callees in the input, try to find it in the callees already on the stack, and if it is not there, add it:

   Input: ¢$horse(cat, dog)pig(horse↾)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig, cow, horse↿\#$

   This is can be done because the input pointer remains confined to the callee name, and the stack pointer to the segment (.....#

6. Now we need to find the name of the next callee to be processed. Move the input pointer back to the name of the caller:

   Input: ¢$horse(cat, dog)↾pig(horse)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig, cow, horse↿\#$

7. Search the callees on the stack for the caller name the input pointer is pointing to:

   Input: ¢$horse(cat, dog)↾pig(horse)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig↿, cow, horse\#$

   This brings the stack pointer back to where it was in Step 4.

8. Move the stack pointer one name to the right, if possible:

   Input: ¢$horse(cat, dog)↾pig(horse)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig, cow↿, horse\#$

   If that was possible we have found the next callee, and go to Step 4; otherwise we continue with Step 9.

9. We have now processed all callees of the present caller; all direct and indirect callees of it have been assembled on the stack:

   Input: ¢$horse(cat, dog)↾pig(horse)cat(pig, cow)dog()cow()$$
   Stack: $\mathbf{B}()\mathbf{B}cat(pig, cow, horse, cat, dog↿\#$

10. Close the callee list with)**B**:

    Input: ¢$horse(cat, dog)↾pig(horse)cat(pig, cow)dog()cow()$$
    Stack: $\mathbf{B}()\mathbf{B}cat↿(pig, cow, horse, cat, dog)\mathbf{B}\#$

    Note that this shows that *cat* is recursive.

11. We are now ready for the next caller, and go to step 2.

5 Conclusions

The art of programming the 2SA consists of two parts: finding an algorithm that can work in the confines of the 2SA memory; and coding the components of the algorithm in 2SA transitions. The first requires brain power, and the use of invariants; the second is easy, using the facilities described in Section 3. Still, programming the 2SA feels somewhat like building a model of the Notre Dame from match sticks.

There is no formal proof that 2SAs are suitable for compiler writing, short of actually writing one, which would require a ridiculous amount of work. However, given the fact that a sorting routine and a routine for building a calling graph could be designed with a few hours thought, the possibility does not look bad. Still, the 2SA is very far from having the same relationship to semantics as the pushdown automaton has to syntax or the finite state automaton to lexical analysis. For semantics, attribute grammars do a far better job, but there seems to be no automaton related to it, and no pertinent theory.

The 2SA is not a Turing machine, so a compiler from C to 2SA is not possible.

Fig. 1 in *SAaC*, which serves there as a demonstration of the suitability of the 2SA as a compiler machine, can be reproduced fairly easily by a fairly structured program for the 2SA (see in particular Section 3.3).

In summary: Myth Plausible.