Journal of Universal Computer Science, vol. 3, no. 3 (1997), 148-171
submitted: 26/2/96, accepted: 14/10/96, appeared: 28/3/97 © Springer Pub. Co.

# Linear Time Simulation of Invertible Non-Deterministic Stack Algorithms

Nils Andersen
(Department of Computing Science, University of Copenhagen, Denmark
nils@diku.dk)

**Abstract:** It is demonstrated how a program making use of a single stack may be transformed, via memorization, into an equivalent one running in time proportional to the sum of variabilities at certain program points of the original program. This result generalizes Cook's linear time simulation of a deterministic two-way push-down automaton and also provides a lucid explanation of Cook's construction.
Obtaining an efficient transformed program depends on making good use of the stack to reduce variabilities at the critical program points. It is suggested to obtain such a program directly from a source program expressed in a non-deterministic language with invertible operations and annotated with a kind of "cuts" somewhat similar to cuts in a Prolog program.
**Key Words:** 2DPDA, Cook's transformation, memorization, non-deterministic programming, backtracking, program transformation
**Category:** I.2.2, F.3.3

## 1 Introduction

The result, also expounded in [Aho, Hopcroft and Ullman (1974)] (Section 9.4), by Stephen A. Cook (see [Cook (1972)]) that a 2DPDA (a two-way deterministic push-down automaton) could be simulated in time proportional to the length of its input tape is extraordinary in view of the fact that such an automaton, if run according to its definition, might take an exponential number of steps, and his achievement inspired the widely used Knuth–Morris–Pratt string matching algorithm [Knuth, Morris and Pratt (1977)].

[Jones (1977)] showed how the large tables implied by Cook's simulation method could be built online during simulation, thus avoiding the construction of useless entries.

The context of this work was automata theory, describing algorithms with binary output, in the form of acceptance or rejection of a string (and thus defining a corresponding formal language). [Bird (1977)] extended the transformation to algorithms with general program variables, as this paper will also do, but Bird only treated programs of a very special form (built around a loop with a single pop operation), and some postprocessing had to be done by hand.

[Andersen and Jones (1994)] exhibited a general method to *compile* a stack program directly into a semantically equivalent program that ran in linear time if the original program was a 2DPDA in program form. The present paper is a reformulation of that result, using a more straight-forward program notation and a more explicit computation of the quantity $u$ determining the running time of the transformed program. In the source program a set of control flow arcs is selected in such a way that one of the selected ars is contained in each path from a pop to a push, each path from a push to a push and each loop without a pop. The sum of the variabilities (a term introduced by Peter Naur to denote the size of a state space) at these arcs constitute $u$.

The usefulness of the transformation thus crucially depends on obtaining a low value for $u$, and to that end the present paper suggests using a non-deterministic algorithm as a starting point, transforming it into a deterministic version as suggested by [Floyd (1967)], but making the stack required by Floyd's method directly available to the user rather than letting it be a mere auxiliary device. The programmer should assist the transformation by selecting the required set of control flow arcs; due to non-determinism an arc may be selected in its forward or in reverse direction.

[Section 2] introduces the source language and [Section 3] describes an example program. The actual transformation is presented in [Section 4], and [Section 5] proves the linear running time result. The idea of taking outset from a non-deterministic program is presented in [Section 6] (an experimental language constructed to test this idea is described in [Appendix A]), and [Section 7] concludes. .

## 2 Source Language

The computing device (a 2DPDA) for which Cook proved his result has, among others, the following characteristics:

1. It is an imperative model, stepping deterministically from state to state.
2. It has a stack that may be pushed or popped, and the top of the stack participates in determining the next step.
3. There is no dynamic reading operation: the input is copied to the read-only tape of the automaton before computation starts.
4. The only variable, more than the stack, is an input tape head position (assuming values in the interval between 1 and the length of the input).
5. The result of a computation is **accept** or **reject**.

The present paper modifies and generalizes this computational model in several ways:

1. The transformation might, in principle, be applied to any sequential imperative algorithm using a single stack; to expose the idea we describe it for programs written in flow chart form.
2. Programs may push and pop the stack, but in the present model we assume data completely unavailable while stored away on the stack. Before information in the stack top can be used it must be popped to a variable.
3. There are no input operations in our flow charts. Conceptually, input may be conceived of as loaded into constants of the program (*e.g.* into a constant array) before execution.
   This of course begs the question as to the meaning of "linear time execution". We shall deal with that question below.
4. It is customary to employ variables in a very liberal way in flow charts, but for the present purpose it proves convenient to indicate the introduction and abolishment of variables explicitly. Assignments that update a variable via a computation based on its old value are therefore distinguished from

initializing assignments, and a specific flow chart symbol to indicate the deallocation of a variable is introduced.

5. The outcome of a computation may be more than one bit; each result symbol in a flow chart indicates an output value.

### 2.1 Syntax

Let $V$ be a set of (simple) *variables*. We assume that each $v \in V$ is associated with a range $R_v$ of possible values of this variable. The expressions *expr* and conditions *cond* that appear in our flow charts are formulated in some applicative language $L$ by means of constants (which may be simple or subscripted) and variables from $V$, combined with standard arithmetic, relational and Boolean operators and functions (such as $+$, $-$, $\times$, max, $<$, $=$, $\neg$, $\vee$, $\wedge$, *etc.*). Conditions may also test for emptiness of the stack by using a built-in Boolean predicate **empty**.

For an expression or condition $e$ we let $vars(e) \subseteq V$ denote the variables occurring in $e$.

A *flow chart* has a unique *entry point* [Tab. 1](a) and a number of symbols of the eight kinds shown in [Tab. 1](b). (There are no assignments of the forms "$v$ := *expr*" where $v \notin vars(expr)$, or "$v$ := **pop**", since such operations could never make use of the old value of $v$; they might be simulated by the sequences "**exit** $v$; **enter** $v$ := *expr*" and "**exit** $v$; **enter** $v$ := **pop**".)

Formally, a *flow chart over L* is a quintuple $(Q, q_0, live, symbol, successors)$ where $Q$ is the set of *program points*, $q_0 \in Q$ is the unique *entry point*, and *live*, *symbol* and *successors* are three mappings with domain $Q$, obeying the restrictions mentioned below:

For each $q \in Q$ we use $V_q$ to denote $live(q) \subseteq V$, the set of program variables live at entry to $q$, and we require $V_{q_0} = \emptyset$. Each $symbol(q)$ is a specific flow chart symbol, and $successors(q) \in Q^0 \cup Q^1 \cup Q^2$, where the length of $successors(q)$ is related to the kind of $symbol(q)$ as shown in [Tab. 1](b).

[Tab. 1] also shows the restrictions on the ways variables may be employed in $symbol(q)$ and the required connections between $V_q$ and $V_{q'}$ for $q'$ in the tuple $successors(q)$.

### 2.2 Semantics

Consider a fixed flow chart $C$.

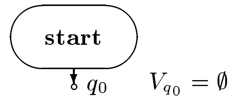
A *(total) configuration* $(q, x, t)$ in $C$ is a triple with a program point $q$, a store $x$ and a stack $t$ of values. If $V_q = \{v_1, \ldots, v_n\}$ then $x$ is a tuple from the Cartesian product $R_{v_1} \times \ldots \times R_{v_n}$. The initial configuration $(q_0, (), [])$ consists of the entry point, the empty store and the empty stack.

The computation with $C$ is the sequence of configurations that starts with the initial one and continues according to the standard interpretation of flow chart symbols.

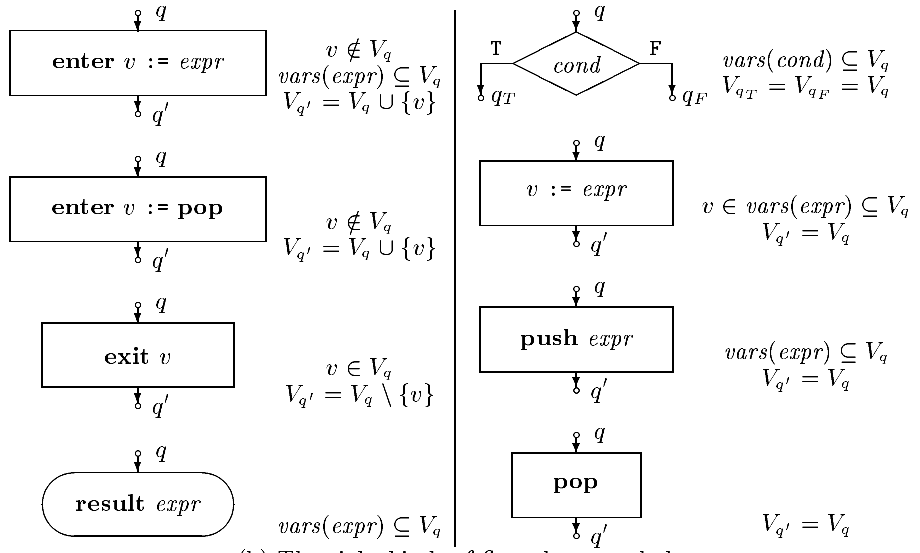
A more formal presentation may be found in [Andersen and Jones (1994)], but a few special features are noted here:

Reaching a **pop**-instruction with an empty stack is an error.

When a value has been pushed onto the stack it becomes completely invisible to the program until (if ever) it is popped from the stack again. Not even the stacktop is available in expressions or conditions (which is in contrast to the usual




(a) Designation of the flow chart entry point.




(b) The eight kinds of flow chart symbols.

**Table 1:** Flow chart symbols and the corresponding requirements on the set of live variables at each program point.

automata-theoretic formulation). We do, however, permit a test on whether the stack is empty or not (designated by "**empty**"), rendering superfluous a specific stack bottom marker.

There is no flow chart element for output. When (if ever) execution stops at a node "**result** *expr*", the outcome of the computation is the value of *expr* in the present store.

### 2.3 Variabilities

There is no flow chart element for input either, so each flow chart has a unique (finite or infinite) computation. What would normally be thought of as a "program" (taking input) is here a *family* of flow charts (where the values of certain constants vary).

For each program point $q \in Q$ with $V_q = \{v_1, \ldots, v_n\}$ we let $S_q$ denote the subset of $R_{v_1} \times \ldots \times R_{v_n}$ consisting of those stores that actually occur at $q$ during computation. The size of $S_q$ plays an important rôle in this research and is called the *variability* of $q$, denoted $v_q$.

In the ordinary model where complexity of computation is measured as a function of the size of the input, large input gives rise to large variations in the values of variables during execution. In our model, variations in the values of variables are mirrored in the variabilities $v_q$. We therefore form $u = \sum_{q \in Q} v_q$, and a preliminary answer to the question about the sense of the term "linear time" is "time proportional to $u$". A more precise answer is given in [Section 5].

Constants don't contribute to $v_q$ or $u$, and [Mogensen (1994)] has explained how one might even permit WORM tapes (Write Once Read Many arrays, a kind of constant arrays with deferred initialization) in the program without affecting the results presented in this paper.

A *surface state* is a triple $(q, x, \mathit{isEmpty})$ where $q \in Q, x \in S_q$ and *isEmpty* is a boolean indicating whether the stack is empty. Stack discipline means that from a particular configuration $(q, x, t)$, the subsequent part of the computation is determined by the surface state $(q, x, t = [])$ only, until (if ever) the stack becomes lower than $t$.

Let $Q_{\text{psh}}$ denote the set of program points of push symbols, and define $u_{\text{psh}} = \sum_{q \in Q_{\text{psh}}} v_q$. It may be of some interest to note how the quantities $u$ and $u_{\text{psh}}$ enter into estimates of the *maximum* running time of a program.

**Proposition 1.** *During the computation of a halting program, the stack height never exceeds* $u_{\text{psh}}$.

*Proof.* Associate with each element of the stack during a computation the $(q, x)$-part of the configuration in which it was pushed. If this set of pairs contained a duplicate, the program would continue in a loop. □

Let $\gamma$ denote the size of the stack alphabet.

**Proposition 2.** *A computation with more than* $u(1 + \gamma + \gamma^2 + \ldots + \gamma^{u_{\text{psh}}})$ *configurations will compute forever.*

*Proof.* According to [Prop. 1] the number of different stacks in a terminating program is at most $1+\gamma+\gamma^2+\ldots+\gamma^{u_{\text{psh}}}$, but repeating a configuration combining a specific surface state with a specific stack would make the computation loop. □

In other words, the worst case running time of a non-looping stack program is $\mathcal{O}(u \cdot (u_{\text{psh}} + 1) \cdot \gamma^{u_{\text{psh}}})$.

### 2.4 Special Case: 2DPDA

A 1-head 2-way deterministic push-down automaton (defined in [Cook (1972)] or [Aho, Hopcroft and Ullman (1974)]) consists of a finite state control attached to a push-down stack and an input tape with a read-only head confined between endmarkers $\vdash$ and $\dashv$. The input string to be recognized is located between the two endmarkers, and operation is begun with the head scanning $\vdash$ and a bottom marker $\perp\!\!\!\perp$ on the stack.

In one move the machine may, depending on the internal state, the symbol scanned by the head, and the top of the push-down store: change internal state, let its input head remain stationary or move it left or right, and push or pop a symbol $\neq \perp\!\!\!\perp$ (or leave the stack unchanged). Alternatively, the machine may either accept or halt without accepting.

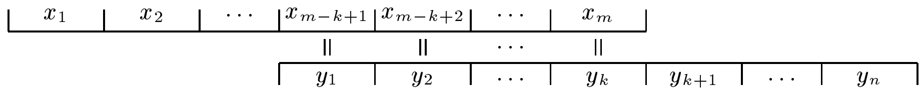
This device is easily modelled by a flow chart of the kind we use here: The input is a constant array $a_1 \ldots a_n$ (let $a_0 = \vdash$, $a_{n+1} = \dashv$), $V$ consists of the input pointer $i$ and the symbol *top* on top of the stack, and moves are imitated using the following ideas:

- Does the head scan a particular symbol $a$? $a_i = a$
- Have we reached the bottom marker of the push-down stack? **empty**
- Is symbol $A \neq \perp\!\!\!\perp$ on top of the push-down stack? $top = A$
- Move the head left/right $i$ := $i \mp 1$
- Pop the stack: **exit** *top* → **enter** *top* := **pop**
- Push symbol $A$ onto the stack: **push** *top* → **exit** *top* → **enter** *top* := $A$

$R_i$ is $\{0, 1, \ldots, n, n+1\}$, and $R_{\text{top}}$ is the set of stack symbols. Each $v_q$ is bounded by $(n+2)\gamma$, and if only the string $a_1 \ldots a_n$ is varied but the flow chart and its symbols are otherwise kept fixed, the quantity $u$ is $\mathcal{O}(n)$.

## 3 Example: The Longest Overlap Problem

To reconstruct a long string from the knowledge (or even sometimes only partial knowledge) of some of its substrings is a practical problem that appears in numerous disciplines including the biochemical determination of gene-sequences and dendrochronology. To illustrate our transformation we shall use an idealization of that problem, called the *Longest Overlap Problem*(LOP): Two strings $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$ of characters from an alphabet $\Sigma$ are given, and we want to determine the largest $k$, $0 \leq k \leq \min\{m, n\}$, such that the $k$-suffix of $x$ equals the $k$-prefix of $y$ [see Fig. 1].

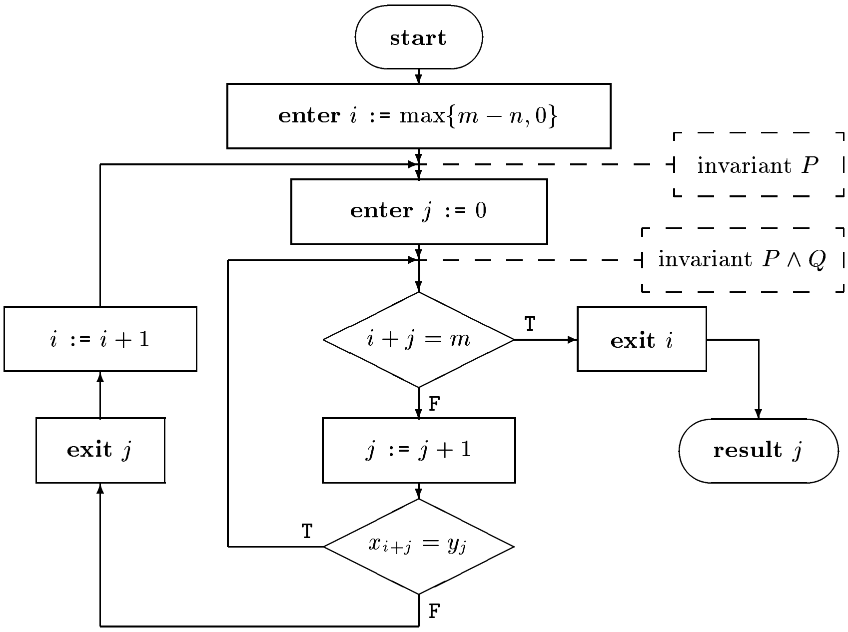



$$\max\{k, 0 \leq k \leq \min\{m,n\} \mid x_{m-k+1} \ldots x_m = y_1 \ldots y_k\}$$

**Figure 1:** The Longest Overlap Problem.

Equivalently, this may be rephrased as a search for $m - i$ where $i$ is the least index ($\max\{0, m-n\} \leq i \leq m$) such that $x_{i+1} \ldots x_m = y_1 \ldots y_{m-i}$.

Let us express $x_{i+1} \ldots x_{i+j} = y_1 \ldots y_j$ by saying that "$i$ anchors a match of length $j$". Since we are looking for the least $i$ that satisfies a certain condition (that of "anchoring a match reaching to the end of the $x$-string"), an obvious procedure would be to try the possible values of $i$ in increasing order and for each anchor $i$ try increasing values of the length $j$.




$P$ : No index $h$ preceding $i$ anchors a match of length $m-h$:
$\forall h, \max\{m-n, 0\} \leq h < i : x_{h+1} \dots x_m \neq y_1 \dots y_{m-h}$
$Q$ : The present index $i$ anchors a match of length $j$:
$x_{i+1} \dots x_{i+j} = y_1 \dots y_j$

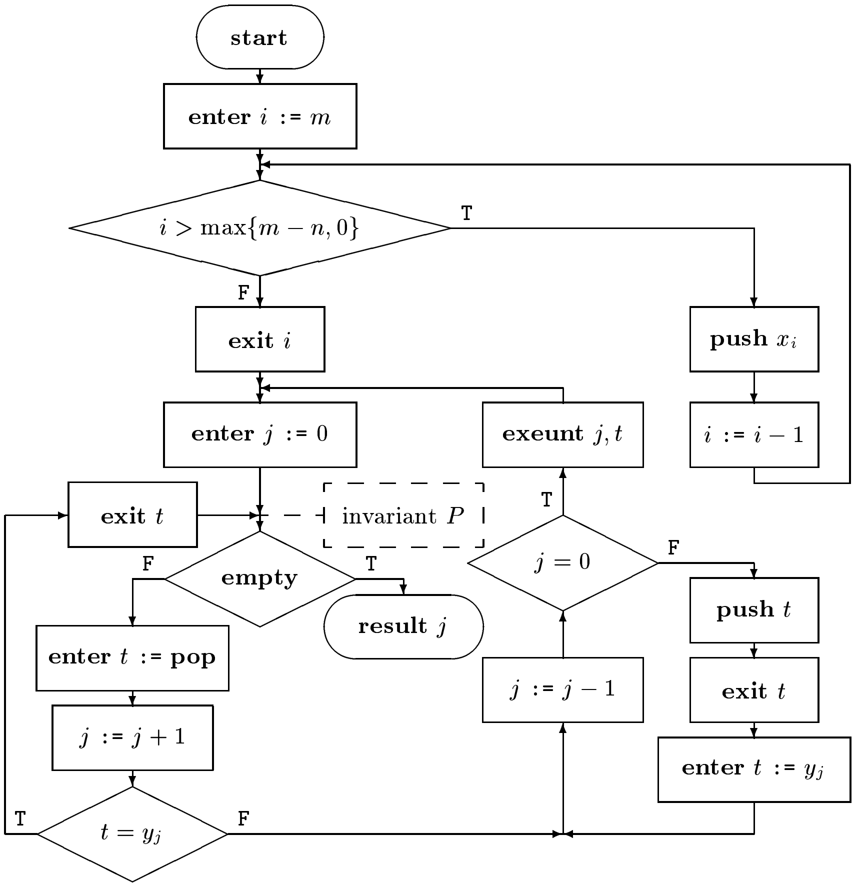
**Figure 2:** A naïve program for LOP.

A program with this behaviour, written in our flow chart language, is shown in [Fig. 2]. Note that $x$, $m$, $y$ and $n$ are constants in the program.

The worst case running time of this program is $\mathcal{O}(\min\{m,n\}^2)$ (try for instance $x = a^{m-1}b, y = a^n$). Unfortunately, although the program is expressed in the right source language, our method is not able to improve it. The ranges of $i$ and $j$ both have length $\min\{m,n\}+1$, and for this program $u = \mathcal{O}(\min\{m,n\}^2)$. One may observe that the program also doesn't use a stack at all; the key to an improvement is to reduce $u$ by using the stack in a clever way.

A fruitful observation is (we shall later [see Section 6] suggest how this idea might be automated): When during the computation $x_{i+j}$ is compared to $y_j$, the preceding elements $x_{i+1}, \dots, x_{i+j-1}$ have been found equal to $y_1, \dots, y_{j-1}$. It would therefore be possible to keep the $x$-elements on a stack and pop each $x_{i+j}$ passing the test $x_{i+j} = y_j$ since the popped elements could be recovered from $y$.

The program in [Fig. 3] is based on this idea. The running time still is $\mathcal{O}(\min\{m,n\}^2)$ but for this program $u = \mathcal{O}(\min\{m,n\} \cdot |\Sigma|)$ (where $|\Sigma|$ denotes the number of characters in the alphabet), and our transformation may now be applied.



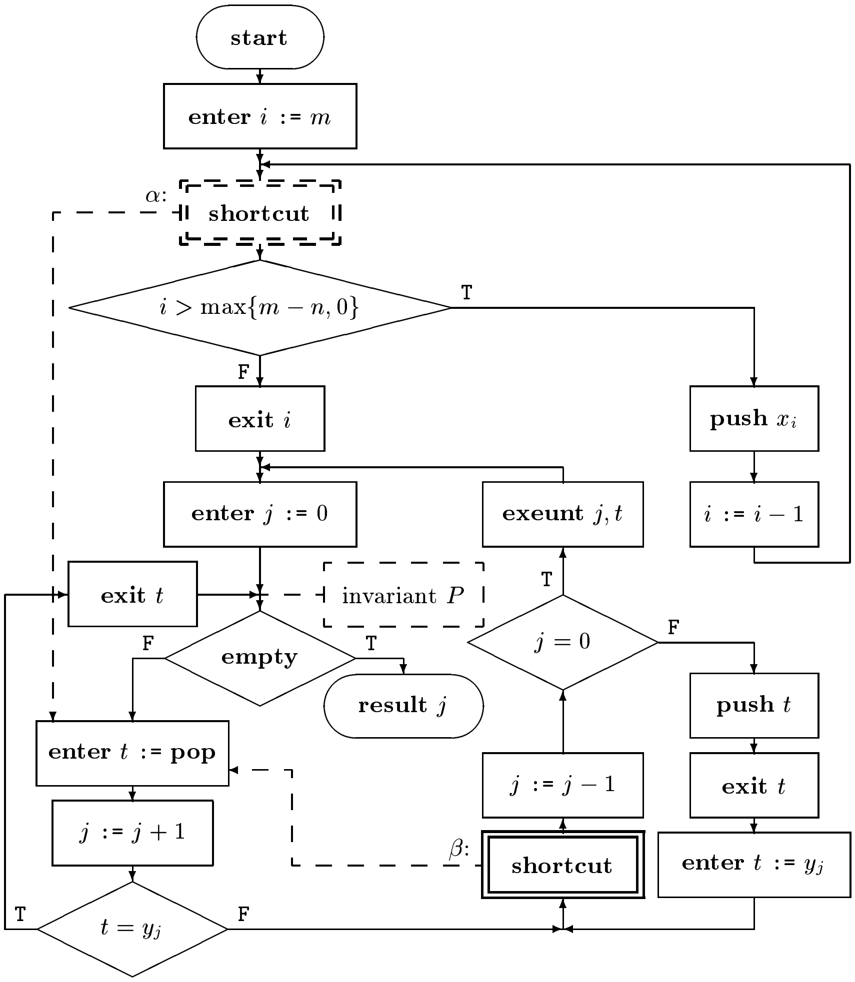

$P$ : The stack (from top to bottom) contains a suffix $x_{i+j+1}, \ldots, x_m$ of $x$, and for the value of $i$ defined in this way it is the case that

– No index $h$ preceding $i$ anchors a match of length $m - h$: $\forall h, \max\{m-n, 0\} \leq h < i : x_{h+1} \ldots x_m \neq y_1 \ldots y_{m-h}$
– The present index $i$ anchors a match of length $j$: $x_{i+1} \ldots x_{i+j} = y_1 \ldots y_j$

**Figure 3:** A stack program for LOP.

## 4 Transformation

Cook's crucial observation to improve the running time is that during a computation the progress from a particular program point is completely determined by the surface state at that point, until (if ever) the top of the stack is popped. For each surface state it is therefore sufficient to generate the ensuing computation, until the stack is popped, in detail once; the transition may then be entered into a table, and if the same surface state is ever met again, a direct jump to the pop symbol may be performed.




$P$ : Same invariant as in [Fig. 3]

**Figure 4:** The transformed program for LOP.

By the following observation it is legal to omit the stack emptiness bit:

If the stack is empty there is no need to memorize the state, waiting for a pop, since a pop operation would be illegal. If the stack is not empty, it remains so until the first pop operation. We may therefore imply for all memorized surface states that the stack is non-empty, and information to that extent need not be stored.

To avoid keeping track of every single program point as computation proceeds we select only some of the arcs in the flow chart, breaking them by inserting a new kind of one-entry one-exit flow chart symbol, a **shortcut**. A shortcut doesn't alter the set of live variables but just invokes the bookkeeping necessary for

Cook's improvement.

[Fig. 4] shows a program for LOP where shortcuts (double-boxed) have been inserted at program points $\alpha$ and $\beta$. (Please, for the moment, disregard the fact that one of the boxes is dashed, and also disregard the dashed arrows.)

Let $Q_{\text{pop}}$ and $Q_{\text{sht}}$ denote the set of program points of pop symbols and of shortcut symbols, respectively, and introduce globally three new variables:

**trace:** to hold a list of pairs $(q, x)$, $q \in Q_{\text{sht}}$, $x \in S_q$
**dest:** a table from pairs $(q, x)$, $q \in Q_{\text{sht}}$, $x \in S_q$ to pairs $(r, y)$, $r \in Q_{\text{pop}}$, $y \in S_r$
**dump:** a stack of lists of the kind contained in *trace*, driven in lock-step with the original program stack

Flow chart symbols changing the stack are encumbered with some bookkeeping, employing the three new variables, and an interpretation is given to the shortcut symbol. The new semantics should be as shown in [Tab. 2].

| | |
|---|---|
| $r$ **enter** $v$ **:= pop** | Function: Pop the stack to initialize $v$.<br>Bookkeeping: Let $y \in S_r$ denote the present set of values; augment the table *dest* with an entry $(q, x) \mapsto (r, y)$ for each pair $(q, x)$ present in the list *trace*.<br>Afterwards assign the previous trace list, popped from *dump*, to *trace*. |
| $r$ **pop** | Function: Pop the stack (and discard the popped value).<br>Bookkeeping: As above. |
| **push** *expression* | Function: Push the value of *expression* on the stack.<br>Bookkeeping: Push the present value of *trace* on *dump*, and continue with an empty list in *trace*. |
| $q$ **shortcut** $q'$ | Bookkeeping: If the stack is empty don't do anything, just continue from $q'$. Let us say, in this case, that "the shortcut symbol skipped".<br>Otherwise, look up $(q, x)$ in *dest* where $x \in S_q$ is the present store. If an entry $(q, x) \mapsto (r, y)$ is found, jump directly to the pop symbol $r$ and change store to $y$. We shall use the terminology that "the shortcut jumped".<br>If there is no such entry, add the pair $(q, x)$ to *trace* and continue from $q'$. We shall say that "the shortcut fell through". |

**Table 2:** Operations involving *trace*, *dump* and *dest*

The new interpretations may mean a drastic change in program performance: Computation is delayed by the bookkeeping operations but may also be shortened by jumping shortcuts. Note that an arrow in the diagram is no longer the only possibility for transfer of control: potentially, a shortcut may jump directly to a pop symbol found via table-lookup.

In [Fig. 4] the added possibilities for flow of control are shown as dashed arrows.

When notation $Q$, $V_q$, $u$, etc. is used below it refers to the values *before* the three new bookkeeping variables were introduced.

The net effect of the program doesn't change: If program $P'$ is obtained from $P$ via the insertion of shortcuts and the new semantics, then since transitions in *dest* are obtained during an imitation of $P$, and $P'$ only deviates from $P$ by sometimes jumping according to the information in *dest*, we have

**Theorem 3.** *The simulation of $P$ by $P'$ is faithful in the sense that*

1. *for each configuration reached during the computation with $P'$ a similar configuration (with global variables trace, dump and dest removed) will be reached during computation with $P$.*
2. *any computation with $P$ may be continued to a configuration similar (in the above sense) to a configuration reached during computation with $P'$.*

A formal proof of this (although for a different program syntax) may be found in [Andersen and Jones (1994)].

**Corollary 4.** *$P'$ loops if and only if $P$ does so.*

## 5 Running Time

Assume, as above, that $P'$ is obtained from the flow chart program $P$ via insertion of shortcuts and the new semantics. We first find some bounds on the number of flow chart symbols executed by $P'$ and afterwards estimate the bookkeeping overhead.

**Lemma 5.** *If $P$ doesn't loop, shortcut operations will, during computation with $P'$, skip at most once and fall through at most once with the same store.*

*Proof.* If some shortcut skipped twice with the same store, then obviously the program would loop. Similarly, if we arrived at shortcut point $q$ with store $x$, and the pair $(q, x)$ was already present in *trace* or *dump*, then the program would loop.

We may thus assume that during computation with $P'$ all the pairs in *trace* and *dump* are different; but when a pair during a pop operation is removed, a corresponding entry is made in *dest*, preventing this pair from being reentered into *trace* later. We therefore conclude that during the whole computation each of the pairs $(q, x)$, $q \in Q_{\text{sht}}$, $x \in S_q$, is inserted in *trace* at most once, so that the shortcut at $q$ may at most once fall through with store $x$. □

To avoid unnecessary recomputation we must break critical loops: Shortcuts should be inserted in such a way that

a. Each path from a pop symbol or a push symbol to a push symbol is broken by a shortcut.
b. Each loop in the flow chart not containing a pop symbol contains a shortcut.

Define $u_{\text{sht}} = \sum_{q \in Q_{\text{sht}}} v_q$, the sum of variabilities at shortcuts.

**Lemma 6.** *If a. has been observed, and if no shortcut during computation with $P'$ skips or falls through twice with the same store, then $P'$ executes at most $2u_{\mathrm{sht}} + 1$ push operations and also at most that many pop operations.*

*Proof.* Consider the push and the shortcut operations met during the computation with $P'$, and in particular associate with each push operation the most recently executed shortcut (if any). Because of a. only the first push may not have a preceding shortcut, and otherwise the associated shortcut is unique and cannot have jumped. Now apply Lemma 5.

By the nature of a stack, the number of pops cannot exceed the number of pushes. □

The main result of this section is that if $P'$ doesn't loop, it executes $\mathcal{O}(u_{\mathrm{sht}})$ flow chart symbols:

**Theorem 7.** *If a. and b. have been observed, the following conditions on the execution of $P'$ are equivalent:*

*i) $P'$ loops*
*ii) Some flow chart symbol is executed more than $4u_{\mathrm{sht}} + 2$ times*
*iii) Some shortcut symbol skips or falls through twice with the same store*

*Proof.* i) $\Rightarrow$ ii) is obvious.

ii) $\Rightarrow$ iii): Let $q$ be a symbol in the flow chart. With every occurrence of $q$ during the computation with $P'$ except the last one associate the first subsequent pop or shortcut. If it is a jumping shortcut, associate this occurrence of $q$ with the pop operation that is the destination of the jump instead. Because of b. the indicated shortcuts or pops exist and are unique. Assume that there were more than $4u_{\mathrm{sht}} + 2$ occurrences of $q$ but that no shortcut skipped or fell through twice with the same store. In that case more than $2u_{\mathrm{sht}} + 1$ pop operations would be executed, contradicting Lemma 6.

iii) $\Rightarrow$ i) by Lemma 5. □

By Theorem 7 iii) it would be easy to extend the bookkeeping with a detection of looping.

Preceding each push operation with a shortcut will assure requirement a., but it is desirable to obey a. and b. with a minimum of shortcut operations, or rather: with a minimum value of $u_{\mathrm{sht}}$.

The two shortcuts in [Fig. 4] observe a. and b.; the values are $V_\alpha = \{i\}$, $\max\{m-n, 0\} \le i \le m$, $v_\alpha = 1 + \min\{m, n\}$, $V_\beta = \{j, t\}$, $1 \le j \le \min\{m, n\}$, $t \in \Sigma$, $v_\beta \le \min\{m, n\} \cdot |\Sigma|$, $u_{\mathrm{sht}} = v_\alpha + v_\beta = \mathcal{O}(\min\{m, n\}|\Sigma|)$.

Inspection of [Fig. 4] reveals that the initializing loop is executed once for each value of $i$ and never reentered. The shortcut at $\alpha$ may therefore never jump and is in fact unnecessary (which is why its box was dashed), but leaving out $v_\alpha$ does not change the order of the bound on $u_{\mathrm{sht}}$.

### 5.1 Bookkeeping Overhead

Using linked lists it is easy to represent *trace* and *dump* in such a way that each of the following operations can be done in constant time:

– Adding a pair $(q, x)$ to *trace*
– Testing whether *trace* is empty
– Removing the front pair from *trace* (if not empty)
– Pushing *trace* onto *dump*
– Popping a trace list from *dump*

If the program doesn't loop, the total number of pairs in *trace* and *dump* will never exceed $u_{\mathrm{sht}}$.

The table *dest* contains at most $u_{\mathrm{sht}}$ entries. We shall assume that this table is implemented with hashing techniques in such a way that the following operations also take constant time:

– Inserting an entry $(q, x) \mapsto (r, y)$ into *dest*
– Looking up a pair $(q, x)$ in *dest* (whether or not an entry $(q, x) \mapsto (r, y)$ exists)

Under these assumptions, if $P'$ doesn't loop it follows from Theorem 7 and Lemma 6 that the time spent in bookkeeping operations [see Tab. 2] is $\mathcal{O}(u_{\mathrm{sht}})$.

### 5.2 LOP May Be Solved in $\mathcal{O}(\min\{m, n\})$ Time

In general, the transformation will improve the running time, but the result may not be optimal.

For LOP we conclude that the transformed program will, in the worst case, run in time proportional to $\min\{m, n\} \cdot |\Sigma|$. Since the overlap may be as long as $\min\{m, n\}$ the dependency of the running time for any algorithm solving LOP on this value seems unavoidable; the dependency on the size of the alphabet is less obvious. The reader may have noted the similarity between LOP and the well-known problem of searching a pattern in a string of characters. Using the so-called KMP algorithm [Knuth, Morris and Pratt (1977)] occurrences of an $n$ character pattern among a string of $m$ characters may be found in time proportional to $m + n$. The device used in the string searching problem may in fact also be employed in LOP: first, based only on $y$, compute its so-called "*next*-table", identifying for each prefix of $y$ the longest overlap (other than the identical mapping) of this prefix with itself. This table can be computed in time proportional to the length of $y$, and afterwards the problem can be solved in time also proportional to the length of $y$.

LOP is not altered, if $x$ and $y$ are reversed and interchanged; we may therefore choose to compute and use the *next*-table for the shorter of the two strings, giving a running time proportional to $\min\{m, n\}$, independent of the size of the alphabet.

Historically (see [Knuth, Morris and Pratt (1977)]), Knuth used Cook's theorem on languages recognizable by 2DPDAs to derive linear algorithms for pattern-matching problems whose running time also depended on the size of the alphabet. Pratt's (and Morris') improvement to an algorithm with time independent of alphabet size doesn't seem available via Cook's construction.

## 6 Inversion

We have seen how any stack program may be transformed so that it doesn't repeat states during execution. A crucial point in obtaining a good execution time, however, was to use the stack in a clever way, giving the original program a low value of $u_{\mathrm{sht}}$. This section contains a suggestion as to how such a stack program might be obtained.

In one of the first descriptions of backtrack programming [Floyd (1967)] introduces an auxiliary stack to ensure correct unravelling from blind alleys in a non-deterministic program. It seems natural to make this stack available right from the beginning, thus permitting the user to construct an *invertible non-deterministic stack program*, as explained below.

We now also permit the extra flow chart symbols shown in [Fig. 5].

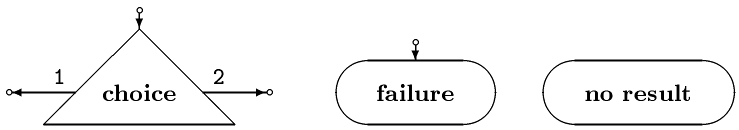



**Figure 5:** New non-deterministic flow chart symbols.

The semantics of the new symbols is as follows: Computation should never reach “failure”. At a choice symbol computation continues along the exit marked “1”, if that is possible without ever reaching “failure”. Otherwise computation continues along the exit marked “2”. This description presupposes that at least one series of choices exists that will not reach “failure”; otherwise the program immediately halts without result.

### 6.1 Invertible Non-Deterministic Flow Charts

The above description of the semantics has been “teleological”, with choices being made depending on future events. A direct deterministic meaning is obtained if we arrange for each symbol in the diagram to be invertible, so that its effect may be undone. The alternative equivalent semantics then is as follows: At a choice branch always take the exit marked “1”. If “failure” is reached, follow the arrows backwards and reverse the computation until the most recent choice branch where exit “2” has not been tried; then take that exit, continuing now in the direction of the arrows. If “start” is reached during backtracking, halt without result.

To permit a computation to be reversed some rewritings of the flow chart might be necessary:

- A plain **pop** operation that just discards the stack top cannot be permitted: a popped value must always be assigned to some variable.
- At an **exit** operation where the use of a variable ceases, the ultimate value of that variable must be recreatable. If it is not deducible from the program, a **push** $v$ could precede the **exit** $v$ operation.
- If the right hand side of an assignment $v$ := ... is not a one-one-function of $v$, the old value of $v$ could be saved on the stack.
- Finally, if some $q$ appears in $\mathit{successors}(q')$ for several program points $q'$, it might not be obvious where to backtrack from $q$; in that case it may be necessary to let each of the $q'$ push a distinguishing mark.

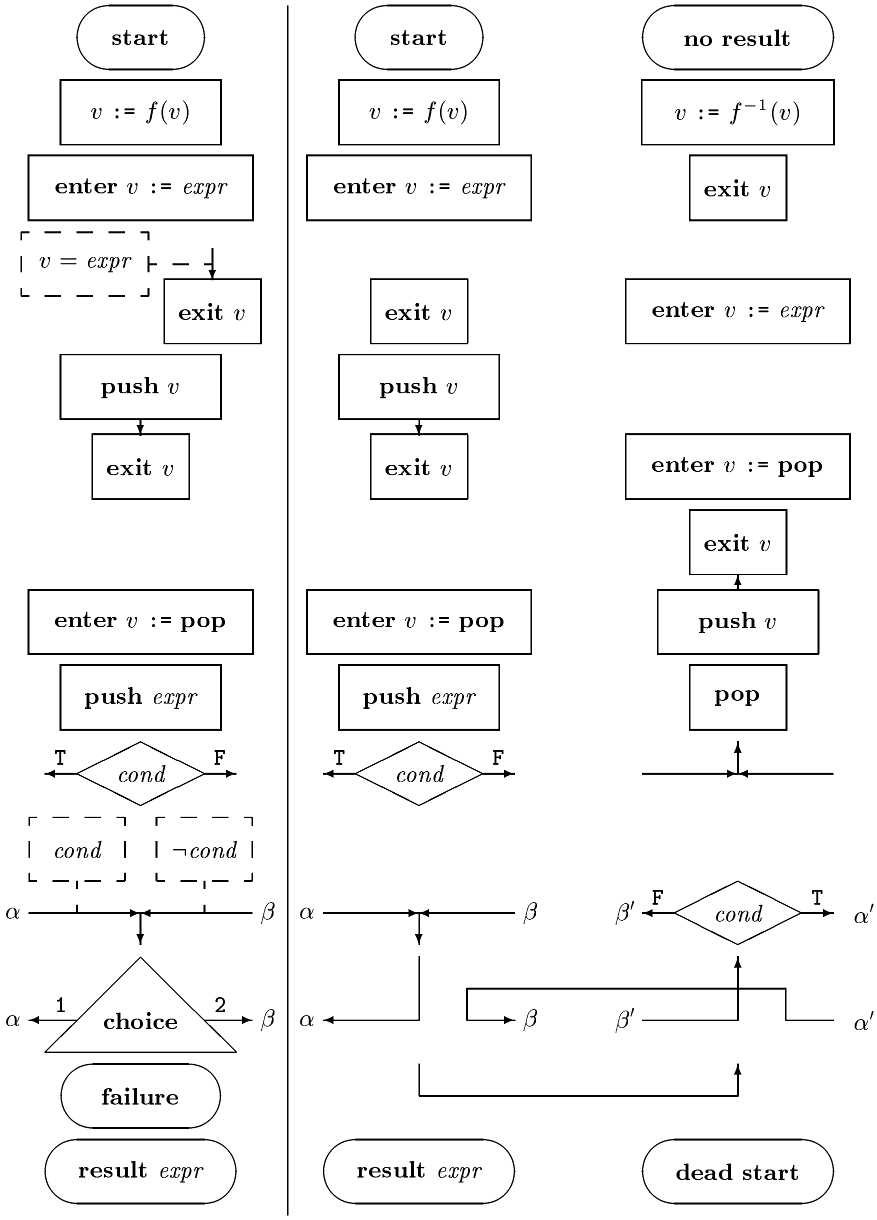



Table 3: Invertible non-deterministic flow chart symbols with their direct and reverse deterministic counterparts.

In the remainder of this paper we shall modify the flow chart symbolism by letting (binary) "join" be a proper flow chart symbol. A join has no computational effect (and does not alter the set of live variables), but formally some new requirements are added for the program points of a flow chart:

- the entry point $q_0$ is not in $successors(q')$ for any program point $q'$, and $symbol(q_0)$ is not join
- if $symbol(q)$ is join, then $q$ is in $successors(q')$ for precisely two program points $q'$
- every remaining $q$ is in $successors(q')$ for precisely one program point $q'$

A join is just depicted as a pair of converging arrows.

### 6.2 Unravelling Non-Determinism

The invertible non-deterministic symbols are shown along with their deterministic counterparts in [Tab. 3].

By means of this table a non-deterministic source program may be transformed into a deterministic one. In the main, this transformation copies the source program and adds a kind of reflected image, with transversals between the copy and the image where the source program had a **failure** or a **choice** symbol. In the reflection each arrow is reversed, and a symbol with $a$ in-going and $b$ out-going arrows is reflected into a symbol with $b$ in-going and $a$ out-going arrows.

A **result** symbol creates an anomaly under his transformation: its mirror image is a symbol with no in-going arrows, denoted **dead start** in [Tab. 3].

*Remark.* Under a different semantics, with no ranking among the two exits of a **choice** symbol so that the goal was to find *all* possible solutions, the deterministic interpretation of a non-deterministic **result** symbol should be to record the solution and then backtrack (as for a **failure** symbol).

The anomaly would then disappear: the deterministic computation would (if at all) stop with a complete list of solutions at the mirror image of the **start** symbol "**no result**" (which ought to be renamed).

Inversion of structured programs (with assignments and compound, conditional and repetitive statements) is also discussed in Chapter 21 in [Gries (1981)].

Solving LOP in the new non-deterministic notation is very simple: just push the desired suffix of $x$ onto the stack, and afterwards check it against the characters of $y$. We obtain the *longest* overlap by marking the choice of push with "1" and check with "2", as in [Fig. 6].

It has been possible to annotate exits and joins as required for inversion, with one exception, marked "$i = ?$" in the figure. Inverting "**push** $x_i$", however, can be done without referring to $i$, so not knowing $i$ does not block backtracking.

[Fig. 7] shows the result of applying the rules of [Tab. 3] to the non-deterministic program [see Fig. 6].

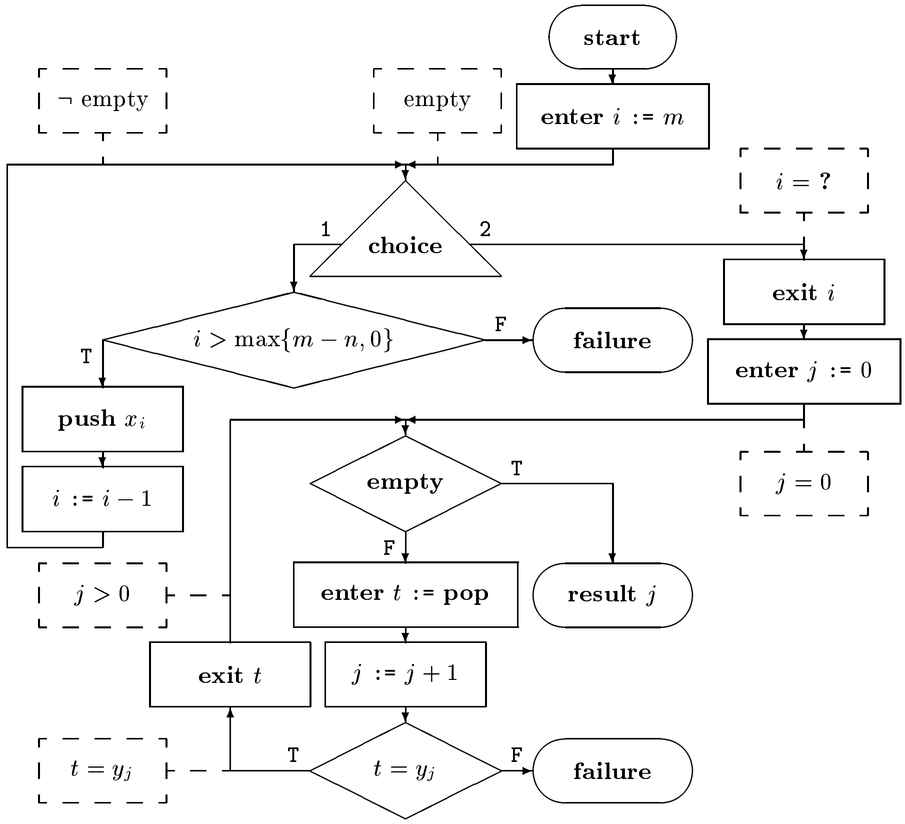



**Figure 6:** Non-deterministic program for LOP.

### 6.3 Linear Notation

A mechanical application of the rules leaves the unfinished statement "**enter** $i$ := **?**" in [Fig. 7], but keeping in mind that $i$ is not needed during backtracking it is possible to complete the program. In fact, a few local transformations will recreate *exactly* the program of [Fig. 3]. The required transformations are:

1. Move "**push** $t$; **exit** $t$" forward through the two arms of the test "$j = 0$"
2. Let the copy of "**push** $t$" going in the "T" direction and "**pop**" cancel each other
3. Move the "**exit** $i$" preceding "**enter** $j$ := 0" back through the preceding join
4. Omit the creation and increment of $i$ during backtracking, also deleting one of the copies of "**exit** $i$" from the previous step and the "**exit** $i$" preceding "**no result**"

Instead of unravelling the non-deterministic program first and then annotate it with shortcuts it is possible to use the non-deterministic formulation directly. To test out the examples of this paper a small experimental non-deterministic linear imperative programming language has been defined and implemented.

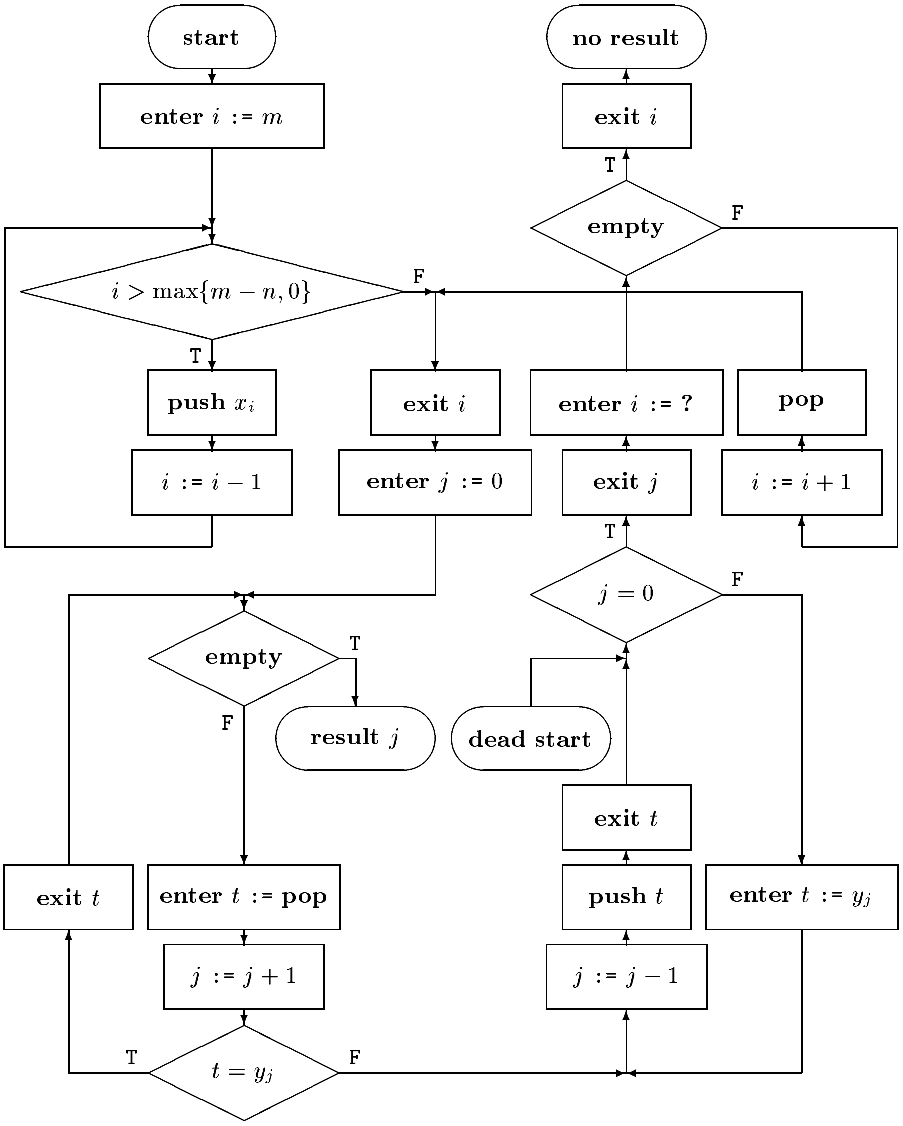



**Figure 7:** Non-deterministic LOP-program unravelled.

Shortcuts now come in two kinds, denoted by *cutfore* (active in the direction of program flow) and *cutback* (active during backtracking) in the language.

[Fig. 8] shows our program for LOP in this syntax ("\/" is the infix maximum operator). Each exit must be annotated with the last value of the variable and each label definition must be annotated with a condition that is true when the label is passed from the statement above it but false when a jump to it is performed. We use double braces $\{\{\ldots\}\}$ for these annotations. Furthermore, there must be exactly one go_to-statement leading to each label. A question mark denotes the undefined value, which has been added to the range of values of variables (thereby permitting the trick necessary to invert our LOP-program). Arithmetic operations on the undefined value continue to be undefined, whereas a test involving "undefined" always yields "true".

The experimental language is completely desribed in [Appendix A].

```
   enter i := M;
{{empty}} augment:
   choose
      begin
         if i = (M-N) \/ 0 then failure;
         cutfore;
         push X[i];
         i -:= 1;
         go_to augment
      end
   or
      begin
         exit {{?}} i;
         enter j := 0;
      {{j = 0}} reduce:
         if empty then result j;
         enterpop t;
         cutback;
         j +:= 1;
         if t <> Y[j] then failure;
         exit {{Y[j]}} t;
         go_to reduce
      end
```

**Figure 8:** Non-deterministic annotated program for LOP in linear notation.

If, as is the case here, every loop in the unravelled program contains a push or a pop operation, a safe way of securing conditions a. and b. is to let a cutfore precede every push and a cutback follow every pop in the non-deterministic program. This has been done in [Fig. 8] (but actually the "cutfore" is superfluous, since the loop hidden in the first alternative of the non-deterministic choice is only executed once).

Experiments confirm that this program finds the longest overlap between $0^{n-1}1$ and $0^n$ in linear time.

Many other fast pattern-matching algorithms (matching a pattern in a string, length of longest palindromic prefix of a string, determining whether one string is a substring of another) may be described in a similar fashion. [Fig. 9] shows, in the same notation, an algorithm that will locate a pattern of length $n$ in a string of length $m$ in time $\mathcal{O}(m + |\Sigma| \cdot n)$ where $|\Sigma|$ is the size of the underlying alphabet.

```
   enter i := M;
{{empty}} guess:
   choose
      begin
         if i = 0 then failure;
         cutfore;
         push X[i];
         i -:= 1;
         go_to guess
      end
   or
      begin
         exit {{?}} i;
         enter j := 0;
      {{j = 0}} verify:
         if j=N then begin
            exit {{N}} j;
            result 1
         end;
         if empty then failure;
         enterpop t;
         cutback;
         j +:= 1;
         if t <> Y[j] then failure;
         exit {{Y[j]}} t;
         go_to verify
      end
```

**Figure 9:** Non-deterministic annotated program for string matching.

## 7 Conclusion

We have exhibited a program notation for stack programs (with explicit allocation and deallocation of variables) and two rules for annotating them with "shortcut-points" in such a way that annotated programs may be given a new semantics under which they will run in time bounded by a constant times the sum of the variabilities at shortcut-points. It is therefore important for the utility of the method to start out with an annotated stack program with a low value of this sum.

The efficient semantics uses certain tables very similar to the tables used by Cook in his linear time 2DPDA simulation, but rather than being precomputed our tables are constructed on line so that only entries corresponding to configurations actually occurring during computation are ever filled in.

As a possible source of stack programs we have suggested an extension of the notation with non-deterministic elements, but restricted in such a way that all elements become invertible.

For many string processing algorithms this extended notation combines a straight-forward description with the improved execution time implied by Cook's transformation.

## References

[Aho, Hopcroft and Ullman (1974)] Aho, A. V., Hopcroft, J. E., Ullman, J. D.: "The design and analysis of computer algorithms"; Addison-Wesley, Reading, MA (1974).

[Andersen and Jones (1994)] Andersen, N., Jones, N. D.: "Generalizing Cook's transformation to imperative stack programs"; Results and trends in theoretical computer science, LNCS (Lecture Notes in Computer Science) 812, Springer, Berlin (1994), 1–18.

[Bird (1977)] Bird, R. S.: "Improving programs by the introduction of recursion"; Comm.ACM (Communications of the ACM), 20 (1977), 856–863.

[Cook (1972)] Cook, S. A.: "Linear-time simulation of deterministic two-way pushdown automata"; Information Processing 71, North-Holland, Amsterdam (1972), 75–80.

[Floyd (1967)] Floyd, R. W.: "Nondeterministic algorithms"; J.ACM (Journal of the ACM), 14 (1967), 636–644.

[Gries (1981)] Gries, D.: "The science of programming"; Springer, New York (1981).

[Jones (1977)] Jones, N. D.: "A note on linear time simulation of deterministic two-way pushdown automata"; Inf.Process.Lett. (Information Processing Letters) 6 (1977), 110–112.

[Knuth, Morris and Pratt (1977)] Knuth, D. E., Morris, J. H., Pratt,V. R.: "Fast pattern matching in strings"; SIAM J.Comput. (SIAM Journal on Computing), 6 (1977), 323–350.

[Mogensen (1994)] Mogensen,T. Æ.: "WORM-2PDPAs: An extension to 2DPDAs that can be simulated in linear time"; Inf.Process.Lett. (Information Processing Letters) 52 (1994), 15–22.

### Acknowledgements

I am indebted to Neil Jones for the idea of extending Cook's construction to programs with variables and for his constant support and to Niels H. Christensen for discussions on the correct treatment of tests for empty stack. I would also like to thank the careful anonymous referee for helpful comments.

## A The Experimental Non-Deterministic Language

Input is presented via constant parameters. This language was specifically designed to deal with the string processing algorithms mentioned in the article; it therefore contains as constants two integral values `M` and `N` and two integer arrays `X[1..M]` and `Y[1..N]`. Furthermore, to accomodate the examples, the value domain has been extended with an explicitly undefined value (denoted `?`).

### A.1 Syntax

#### A.1.1 Context-Free Rules

`varble` = *identifier*
`label` = *identifier*
`const` = *one or more digits (non-negative integer)*

```
expr4   = varble | const | “?” | “M” | “N”
          | “X” “[” expr “]” | “Y” “[” expr “]” | “(” expr “)”
expr3   = expr4 | expr3 “*” expr4
expr2   = expr3 | expr2 “+” expr3 | expr2 “-” expr3
expr1   = expr2 | expr1 “/\” expr2
expr    = expr | expr “\/” expr1
cond1   = “false” | “true” | “empty” | “not” cond1 | “(” cond “)”
cond    = cond1 | expr “=” expr | expr “<” expr | expr “>” expr
          | expr “<>” expr | expr “<=” expr | expr “>=” expr
revbl   = “enter” varble “:=” expr | “exit” “{{” expr “}}” varble
          | “enterpop” varble | “push” expr | “pushexit” varble
          | varble “+:=” expr | varble “-:=” expr
          | “cutfore” | “cutback”
stmnt   = “result” expr | “failure” | “choose” stmnt “or” stmnt
          | “go_to” label
          | “begin” stmnts “end”
stmnts  = stmnt | revbl “;” stmnts
          | “if” cond “then” stmnt “;” stmnts
          | “{{” cond “}}” label “:” stmnts
program = stmnts
```

### A.1.2 Additional Syntactic Requirements

Variables are allocated and deallocated explicitly as indicated by the `enter`-, `exit`-, `enterpop`- and `pushexit`-statements. The set of variables live at each statement entry may therefore be computed statically, and expressions and conditions may, of course, only contain live variables. Similarly, only live variables may be exited, and only new variables may be entered.

In constructions that contain an expression as well as a variable (`enter`- and `exit`-statements and the two accumulating assignment statements) the expression may not contain the variable.

Binary joins in the flow of control are expressed linearly by means of labels. Each label used in the program must therefore be uniquely defined and must have exactly one `go_to`-statement leading to it.

## A.2 Semantics

The meaning of a program is explained via its preprocessing into a list of statements in pseudo-C. Compared to C, our target language has the following peculiarities:

- Input is assumed to be loaded into the arrays `int x[m], y[n];` in advance.
- The function `lookup(`$a$`,`$i$`,`*limit*`)` checks that $i$ is in the range $1 \leq i \leq limit$ and if so has as its value $a$`[`$i$`-1]` (thereby simulating 1-origin indexing).
- The range of variables declared with the type-specifier `int*` is the set of integers extended with an extra “wild” value, denoted `wildcard`.
- Functions `max` (`min`) with two parameters compute the maximum (minimum) of their arguments.

- These functions as well as the binary arithmetic operators `+`, `-` and `*` work for extended integers, producing `wildcard` as a result whenever an operand (or both operands) is `wildcard`.
- The binary relational operators `==`, `!=`, `<`, `<=`, `>` and `>=` produce `1` (true) if an operand (or both operands) is `wildcard`.
- The functions `enter` and `exit` update a list of live variables. (Declarations in our source program are not assumed to be properly nested.)
- This list is used by the function `shortcut` to manage three auxiliary variables `trace`, `dest` and `dump` as described in [Tab. 2].
- The program also manipulates a stack; `push`$(n)$ pushes $n$ on top of this stack, and `pop()` pops the stack and has the popped element as its function value. These functions also manage the necessary additional bookkeeping [see Tab. 2].
- The value of a function call `empty()` is 1 or 0 according to whether the stack is empty or not.
- Each label $\ell$ in the source program is duplicated to labels denoted $\ell$ and $\ell'$ in the target program.
- In addition to ordinary integral results `return` is also permitted to transmit the message `"no result"`.

### A.2.1 Translations

$\mathcal{E}$ : `expr` $\rightarrow$ expressions in pseudo-C

$\mathcal{E}[\![v]\!] = v$ for a variable $v$
$\mathcal{E}[\![n]\!] = n$ for a constant $n$
$\mathcal{E}[\![\texttt{?}]\!] = \texttt{wildcard}$
$\mathcal{E}[\![\texttt{M}]\!] = \texttt{m}$
$\mathcal{E}[\![\texttt{N}]\!] = \texttt{n}$
$\mathcal{E}[\![\texttt{X[}e\texttt{]}]\!] = \texttt{lookup(x,}\mathcal{E}[\![e]\!]\texttt{,m)}$
$\mathcal{E}[\![\texttt{Y[}e\texttt{]}]\!] = \texttt{lookup(y,}\mathcal{E}[\![e]\!]\texttt{,n)}$
$\mathcal{E}[\![\texttt{(}e\texttt{)}]\!] = \mathcal{E}[\![e]\!]$
$\mathcal{E}[\![e_1 \diamond e_2]\!] = \mathcal{E}[\![e_1]\!] \diamond \mathcal{E}[\![e_2]\!]$ where $\diamond$ is one of the operators `+`, `-` or `*`
$\mathcal{E}[\![e_1 \texttt{/\textbackslash} e_2]\!] = \texttt{min(}\mathcal{E}[\![e_1]\!]\texttt{,}\mathcal{E}[\![e_2]\!]\texttt{)}$
$\mathcal{E}[\![e_1 \texttt{\textbackslash/} e_2]\!] = \texttt{max(}\mathcal{E}[\![e_1]\!]\texttt{,}\mathcal{E}[\![e_2]\!]\texttt{)}$

$\mathcal{C}$ : `cond` $\rightarrow$ expressions in pseudo-C (using `0` and `1` as the truth values)

$\mathcal{C}[\![\texttt{false}]\!] = \texttt{0}$
$\mathcal{C}[\![\texttt{true}]\!] = \texttt{1}$
$\mathcal{C}[\![\texttt{empty}]\!] = \texttt{empty()}$
$\mathcal{C}[\![\texttt{not } c]\!] = \texttt{!}\mathcal{C}[\![\texttt{c}]\!]$
$\mathcal{C}[\![\texttt{(}c\texttt{)}]\!] = \mathcal{C}[\![c]\!]$
$\mathcal{C}[\![e_1 \lhd e_2]\!] = \mathcal{E}[\![e_1]\!] \lhd \mathcal{E}[\![e_2]\!]$ where $\lhd$ is `<`, `<=`, `>` or `>=`
$\mathcal{C}[\![e_1\texttt{=}e_2]\!] = \mathcal{E}[\![e_1]\!]\texttt{==}\mathcal{E}[\![e_2]\!]$
$\mathcal{C}[\![e_1\texttt{<>}e_2]\!] = \mathcal{E}[\![e_1]\!]\texttt{!=}\mathcal{E}[\![e_2]\!]$

$\mathcal{R}$ : `revbl` $\rightarrow$ a pair of mutually inverse pieces of program text in pseudo-C

$\mathcal{R}[\![\texttt{enter } v \texttt{ := } e]\!] = (\{\texttt{int}^\star\ v \texttt{ = } \mathcal{E}[\![e]\!]\texttt{; enter(}v\texttt{);}\ ,\ \texttt{exit(}v\texttt{);}\})$
$\mathcal{R}[\![\texttt{exit }\{\{\ e\ \}\}\ v]\!] = (\texttt{exit(}v\texttt{);}\ ,\ v \texttt{ = } \mathcal{E}[\![e]\!]\texttt{; enter(}v\texttt{);})$
$\mathcal{R}[\![\texttt{enterpop } v]\!] = (\{\texttt{int}^\star\ v \texttt{ = pop(); enter(}v\texttt{);}\ ,\ \texttt{push(}v\texttt{); exit(}v\texttt{);}\})$
$\mathcal{R}[\![\texttt{push } e]\!] = (\texttt{push(}\mathcal{E}[\![e]\!]\texttt{);}\ ,\ \texttt{pop();})$

$\mathcal{R}[\![\texttt{pushexit}\ v]\!] = (\texttt{push(}v\texttt{); exit(}v\texttt{);}\ ,\ v\ \texttt{= pop(); enter(}v\texttt{);})$
$\mathcal{R}[\![v\ \texttt{+:=}\ e]\!] = (v\ \texttt{+=}\ \mathcal{E}[\![e]\!]\texttt{;}\ ,\ v\ \texttt{-=}\ \mathcal{E}[\![e]\!]\texttt{;})$
$\mathcal{R}[\![v\ \texttt{-:=}\ e]\!] = (v\ \texttt{-=}\ \mathcal{E}[\![e]\!]\texttt{;}\ ,\ v\ \texttt{+=}\ \mathcal{E}[\![e]\!]\texttt{;})$
$\mathcal{R}[\![\texttt{cutfore}]\!] = (\texttt{shortcut();}\ ,\ \texttt{/* nothing */})$
$\mathcal{R}[\![\texttt{cutback}]\!] = (\texttt{/* nothing */}\ ,\ \texttt{shortcut();})$

$\mathcal{S}$ : `stmnts` $\rightarrow$ a statement-list in pseudo-C
$\mathcal{S}[\![\texttt{result}\ e]\!] = \texttt{return}\ \mathcal{E}[\![e]\!]\texttt{; /* dead start: */}$
$\mathcal{S}[\![\texttt{failure}]\!] = \texttt{/* nothing */}$
$\mathcal{S}[\![\texttt{choose}\ s_1\ \texttt{or}\ s_2]\!] = \mathcal{S}[\![s_1]\!]\ \ \mathcal{S}[\![s_2]\!]$
$\mathcal{S}[\![\texttt{go\_to}\ \ell]\!] = \texttt{goto}\ \ell\texttt{;}\ \ \ell'\texttt{:}$
$\mathcal{S}[\![\texttt{begin}\ ss\ \texttt{end}]\!] = \mathcal{S}[\![ss]\!]$
$\mathcal{S}[\![r\,\texttt{;}\,ss]\!] = \text{let}\ (\rho, \rho')\ \text{be}\ \mathcal{R}[\![r]\!]\ \text{in}\ \rho\ \ \mathcal{S}[\![ss]\!]\ \ \rho'$
$\mathcal{S}[\![\texttt{if}\ c\ \texttt{then}\ s\texttt{;}\ ss]\!] = \texttt{if (}\mathcal{C}[\![c]\!]\texttt{)}\ \{\mathcal{S}[\![s]\!]\}\ \texttt{else}\ \{\mathcal{S}[\![ss]\!]\}$
$\mathcal{S}[\![\{\{\ c\ \}\}\ \ell\texttt{:}\ ss]\!] = \ell\texttt{:}\ \mathcal{S}[\![ss]\!]\ \texttt{if (!(}\mathcal{C}[\![c]\!]\texttt{)) goto}\ \ell'\texttt{;}$

$\mathcal{P}$ : `program` $\rightarrow$ the list of statements in the body of function `main()` in pseudo-C
$\mathcal{P}[\![ss]\!] = \mathcal{S}[\![ss]\!]\ \texttt{return "no result";}$