MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper No. 312 November 1988

Support for Obviously Synchronizable Series Expressions in Pascal

by

Jonathan L. Orwant

Abstract

Obviously synchronizable series expressions enable programmers to write algorithms as straightforward compositions of functions rather than as less comprehensible loops while retaining the significantly higher efficiency of loops. A macro package supporting these expressions in Lisp has been in use since December of 1987.

However, the theory behind obviously synchronizable series expressions is not restricted to Lisp; in fact, it is applicable to any programming language. Because many people view packages designed in Lisp to be dependent on the qualities which make Lisp different from other languages, it was decided to support the macro package in the all-purpose language Pascal. This paper discusses its implementation.

Copyright © Massachusetts Institute of Technology, 1988

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

Introduction

Programming normally entails a compromise between legibility of code and speed of execution. If algorithms are expressed as compositions of functions operating on collections of data elements rather than as loops, they are generally easier to understand and modify but run significantly slower than equivalent algorithms expressed as loops. Conversely, algorithms expressed as loops are difficult to comprehend but compile into more efficient code.

To avoid having to make this compromise, the Obviously Synchronizable Series (OSS) macro package was developed [4, 5]. The OSS macro package allows programmers to write code as compositions of functions instead of loops without the disadvantage of inefficient execution inherent in non-loop methods. This inefficiency is avoided by using macro package supported OSS expressions in the construction of code and then passing the completed code to the OSS preprocessor, which substitutes loops for the OSS expressions where appropriate. This system has been in use since December 1987.

Often, packages designed in Lisp are perceived as being dependent upon the qualities that make Lisp different from other languages. However, the theory behind OSS expression support is not limited to Common Lisp; in fact, it is applicable to all programming languages. Lisp was chosen as the original language for implementation because of the ease with which extensions to the language can be made. To demonstrate the feasibility of the OSS package in other languages, it was supported in the all-purpose language Pascal.

Since it was only necessary to demonstrate the feasibility of OSS expressions in Pascal and not to recreate the entire macro package, a parser was built to convert Pascal code into a Lisp-like syntax for use by the OSS preprocessor in Lisp. The resulting code is searched for fragments that involve OSS expressions. These code fragments are grouped with type and variable declaration information and are then passed to the preprocessor, which returns code in loop format that avoids the creation of intermediate series objects. Then, the code is substituted back into the pseudo-Lisp and, using specialized rules developed for a Lisp pretty printer [3], printed out in standard Pascal suitable for compilation. It is important to note that the preprocessor removes all traces of OSS expressions, leaving algorithms in loop form suitable for compilation by a standard compiler.

Background

Loops vs. Compositions of Functions

The inefficiency of compositions of functions operating on collections of data elements is caused primarily by the creation of *intermediate series*. Consider a function that computes the sum of the square roots of a series:

```
(defun sum-sqrts (vect)
  (reduce #'+ (map 'vector #'sqrt (remove-if-not #'plusp vect))))

(sum-sqrts #(4 -4 -16 9))  ⇒  5
```

To compute the **sum-sqrts** of a vector **vect**, two intermediate series are created and stored: the first containing the positive elements of **vect**, and the second containing the square roots of the elements of the first. Only when both of these aggregate data objects have been computed can the answer be returned. In contrast, the same algorithm expressed as a loop runs significantly faster (an order of magnitude on the Symbolics Lisp Machine), but is much more opaque to the human reader:

```
(defun sum-sqrts-loop (v)
  (prog (element last index sum)
        (setq index 0)
        (setq last (length v))
        (setq sum 0)
     L  (if (not (< index last)) (return sum))
        (setq element (aref v index))
        (if (plusp element) (setq sum (+ (sqrt element) sum)))
        (incf index)
        (go L)))
```

Such opacity has led to the encouragement of concise, legible programming styles at the expense of run-time efficiency. To avoid the time and space overhead necessitated by the creation of intermediate series expressions in an algorithm, it is necessary to parallelize the algorithm as much as possible. Instead of computing the functions within an algorithm successively over the entire series and creating a new series expression for each function mapped, the functions are applied to each element individually. In this manner, each element is transferred directly from the function that computes it to any functions which use it without the need for intermediate storage. At the symbolic level, the same number of operations are computed, but intermediate series objects are never created.

Elimination of intermediate series

Unfortunately, elimination of intermediate series objects in an algorithm is not always possible. Functions such as **sort**, which require all elements of the input series before any elements of the output series can be generated, cannot avoid the generation of intermediate series.

Furthermore, it is often difficult to determine whether or not it is possible to eliminate all intermediate series expressions. Making the optimal choice of which intermediate series to eliminate is NP-hard [2]. This is another perceived obstacle to the use of series expressions–unless detailed knowledge of the compiler is known, the programmer cannot be assured of maximal efficiency. The problem is compounded by the fact that inefficiency is not directly proportional to the number of intermediate series expressions created; the creation of just one intermediate series expression typically leads to a significant loss of efficiency [5].

To guarantee elimination of intermediate series objects, restrictions must be placed on the functions allowed. A suitable restriction is that all expressions allowed must be obviously synchronizable [5]. All predefined OSS functions in the macro package are obviously synchronizable, and built-in checking prevents the user from constructing code that violates this restriction.

The OSS macro package

To support use of obviously synchronizable series expressions, an OSS data type and a library of predefined functions operating on OSS expressions were added to Common Lisp, and an OSS preprocessor was implemented to ensure that the restrictions described above were obeyed and to transform the OSS expressions into loop format. The example above that computes the square roots of the positive elements of the vector `#(4 -4 -16 -9)` would look like this using OSS expressions:

```
(Rsum (TmapF #'sqrt (TselectF #'plusp (Evector #(4 -4 -16 9)))))
⇒ 5
```

`Evector` converts a vector to an OSS series, `TselectF` returns a series that contains only the elements that satisfy the predicate `plusp`, and `Rsum` computes the sum of its elements. `Sqrt` is implicitly mapped using `TmapF`.

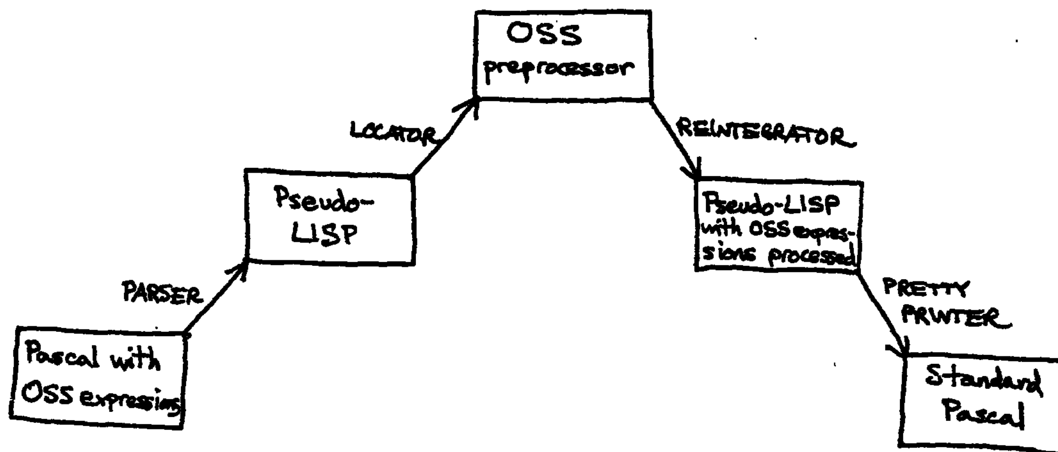
These three functions exemplify the three classes of OSS functions: enumerators, transducers, and reducers. The first letter of each function, pronounced as a separate syllable, designates the class to which it belongs. Enumerators generate series outputs from non-series inputs, transducers generate series outputs from series inputs, and reducers generate a non-series output from a series input. Higher-order functions, currently designated by a final `F`, accept functions as arguments.

In addition to predefined operations on OSS expressions, the macro package supports user construction of OSS functions using `defunS`, and permits local binding of OSS variables with `letS`. Similar in syntax to the Common Lisp `defun` and `let`, these functions employ checking that prevents the programmer from building OSS expressions that violate any of the restrictions necessary to ensure conversion to loops [4].

Once code has been written, the OSS preprocessor converts the OSS expressions in the code to loop format consisting of a *prolog* containing statements to be executed before the actual loop; the *body* of the loop; and an *epilog* containing statements to be executed after the loop. The resulting code is suitable for normal compilation and retains no traces of the original OSS expressions. Thus, no change needs to be made to the actual structure of Lisp.

Lisp was chosen as the language for the original implementation of the OSS macro package because of the ease with which new language constructs can be developed in it. However, the theory behind the development of the macro package applies to all programming languages. Since Lisp is commonly viewed as an idiosyncratic language, and any developments to it to be intrinsically specific to Lisp, a project was undertaken to support the OSS macro package in standard Pascal [1]. Pascal was chosen because of its all-purpose nature; demonstrating the feasibility of OSS expression support in Pascal provides a compelling argument for its adoption across all programming languages.

Implementation



The above diagram illustrates the architecture of the implementation. Pascal code containing OSS expressions is parsed into a Lisp-like notation suitable for the OSS preprocessor. Then, the OSS locator searches for fragments that contain OSS expressions and OSS functions, and passes these to the preprocessor. The preprocessor converts the OSS expressions into loop format optimized for speed. An integrator then incorporates the code back into the pseudo-Lisp, making some minor modifications to handle the intricacies of the preprocessor output. Finally, a pretty printer converts the pseudo-Lisp back into standard Pascal, suitable for compilation by a standard Pascal compiler.

The OSS data Type

The most visible change to Pascal to support obviously synchronizable series expressions is the introduction of the OSS data type. This provides the capacity to construct OSS variables, expressions, procedures, and functions following the same syntax as regular Pascal. OSS declarations resemble vector declarations:

```
type Integers = oss of Integer;
var Inputstream : oss of Char;
function Outputstream(x: integer) : oss of Real;
```

Also supported in Pascal is most of the library of predefined OSS functions developed for Lisp [4]. However, not all of the predefined OSS functions are applicable to Pascal. Functions such as defunS need no analog in Pascal; an OSS function is defined in the normal manner but is given OSS inputs and/or an OSS output. Other functions, such as those operating on lists, are excluded from the Pascal library.

The predefined library of functions violates the Pascal convention of strong typing. This could be avoided by expanding the set of functions to include identical functions

that operate upon different types, but this is not necessary as all traces of OSS expressions are removed once the OSS preprocessor has completed its task and converted the code to loop format. Normal type checking can then be enforced by the Pascal compiler.

An Example

As an example of how OSS expressions might be used in Pascal, a job queue data abstraction which could serve as part of an operating system was implemented. The type definitions are shown below. A `JobQ` is a pointer to a chain of entries which point to records describing jobs. These records have a number of fields, the first of which is a numerical priority.

```
type JobQ = ^JobQentry;
type JobQentry = record job: JobInfo; rest: JobQ end;
type JobInfo = ^JobRecord;
type JobRecord = record priority: real; ... end;
```

Several functions are defined which operate on job queues. `AddToJobQ` allocates space for a new queue entry and then adds the entry to the front of the queue:

```
procedure AddToJobQ (J: JobInfo; var Q: JobQ);
  var E: ^JobQentry;
begin
  new(E);
  E^.job := J;
  E^.rest := Q;
  Q := E
end
```

`EJobQtails` and `EJobQ` are OSS functions with series outputs. Once routed through the OSS preprocessor, they will disappear, and any expressions that reference them will have the reference replaced by the appropriate code in loop format. `EJobQtails` uses the predefined higher-order OSS function `EnumerateF` to enumerate tails of a queue: `<Q, Q^.rest, Q^.rest^.rest, ...>`, and `EJobQ` enumerates the jobs in a queue by calling `EJobQtails` and implicitly mapping the operation of following a pointer and selecting the `job` field over the pointers returned by `EjobQtails`.

```
function EJobQtails (Q: JobQ): oss of JobQ;
  function JobQrest (Q: JobQ): JobQ;
    begin JobQrest := Q^.rest end;
  function JobQnull (Q: JobQ): Boolean;
    begin JobQnull := Q=nil end;
begin EJobQtails := EnumerateF(Q, JobQrest, JobQnull) end

function EJobQ (Q: JobQ): oss of JobInfo;
  var Qs: oss of JobQ;
begin
  Qs := EJobQtails(Q);
  EJobQ := Qs^.job
end
```

The function RemoveFromJobQ removes a job from the end of a queue. RemoveFromJobQ first enumerates the tails of the queue using EJobQTails, and then uses the predefined OSS functions Rlast and Tprevious to obtain a pointer to the penultimate queue entry. The "rest" pointer in this entry is then set to nil to remove the last entry from the queue. If there is no next to last entry, then the queue variable itself is set to nil. RemoveFromJobQ then locates the last entry in the queue and frees the storage associated with it via dispose, returning the contents of its job field. It is assumed that there is at least one job in the queue. Since there is only one instance of EJobQtails in RemoveFromJobQ, the OSS preprocessor will create code which only traverses the queue once.

```
function RemoveFromJobQ (var Q: JobQ): JobInfo;
  var Qs: oss of JobQ;
      NextToLast, Last: JobQ;
begin
  Qs := EJobQtails(Q);
  NextToLast := Rlast(Tprevious(Qs), nil);
  if NextToLast=nil then Q := nil else NextToLast.rest := nil;
  Last := Rlast(Qs, nil);
  RemoveFromJobQ := Last^.job;
  dispose(Last)
end
```

SuperJob, below, inspects a job queue and returns the last (i.e., longest queued) job in the queue whose priority is more than two standard deviations larger than the average priority of all of the jobs in the queue. If there is no such job, nil is returned. The first four statements in the function compute the mean and deviation of the priorities. The fifth statement selects the jobs which have sufficiently large priorities. The last line selects the last of these jobs, if any.

```
function SuperJob (Q: JobQ): JobInfo;
  var Jobs, SuperJobs: oss of JobInfo;
      Count: Integer;
      Mean, Deviation: Real;
begin
  Jobs := EJobQ(Q);
  Count := Rlength(Jobs);
  Mean := Rsum(Jobs.priority)/Count;
  Deviation := sqrt(Rsum(TmapF(sqr,Jobs.priority)/Count - sqr(Mean));
  SuperJobs := Tselect(Jobs.priority>Mean+2*Deviation, Jobs);
  SuperJob := Rlast(SuperJobs, nil)
end
```

This code is a good example of the way OSS expressions are intended to be used. It is important to realize from the above example that OSS expressions are used to convert relatively straightforward programs into simple programs rather than to convert truly complex programs into less complex programs. Most applications involve very simple OSS expressions.

The Parser

The first step in implementation is to convert the Pascal code into a Lisp-like syntax for use by the OSS preprocessor. The parser used for this task utilizes standard methods, creating a recursive-descent parse tree.

When parsed into Lisp, `EJobQTails` and `RemoveFromJobQ`, taken as samples for brevity, are converted into:

```
(DEFUN-FUNC EJOBQTAILS ((Q JOBQ)) (OSS JOBQ)
 (DEFUN-FUNC JOBQREST ((Q JOBQ)) JOBQ
   (PROGN (SETQ JOBQREST (FIELD (FOLLOW^ Q) 'REST))))
 (DEFUN-FUNC JOBQNULL ((Q JOBQ)) BOOLEAN
   (PROGN (SETQ JOBQNULL (= Q NIL))))
 (PROGN (SETQ EJOBQTAILS (ENUMERATEF Q JOBQREST JOBQNULL))))
```

and

```
(DEFUN-FUNC REMOVEFROMJOBQ ((VAR Q JOBQ)) JOBINFO
 (V-LET ((QS (OSS JOBQ)) (NEXTTOLAST JOBQ) (LAST JOBQ))
  (PROGN (SETQ QS (EJOBQTAILS Q))
         (SETQ NEXTTOLAST (RLAST (TPREVIOUS QS) NIL))
         (IF (= NEXTTOLAST NIL)
             (SETQ Q NIL)
             (SETQ (FIELD NEXTTOLAST 'REST) NIL))
         (SETQ LAST (RLAST QS NIL))
         (SETQ REMOVEFROMJOBQ (FIELD (FOLLOW^ LAST) 'JOB))
         (DISPOSE LAST))))
```

Functions such as `DEFUN-PROGRAM`, `DEFUN-FUNC`, `DEFUN-PROC`, `FIELD`, and `FOLLOW^` need not be valid Lisp functions as the code is not required to actually run in Lisp; it only needs to resemble Lisp to the extent that the OSS preprocessor is able to deal with the fragments received from it.

The most important consideration when parsing the Pascal code into Lisp is to make sure that all of the information needed to reconstruct the Pascal code is retained in Lisp. The best example of this concerns typing. Since Pascal has strong typing and Lisp has weak typing, some means had to be developed of retaining the type information in Lisp. This is done using variations on the Lisp `LET` statement: type, variable, constant, and label declarations are retained using `T-LET`, `V-LET`, `C-LET`, and `L-LET` macros, each of which contains a list of declarations and the body of statements lexically enclosed by the declaration.

Most high-level constructs in Pascal, such as `while...do` and `if...then` statements are never seen by the OSS compiler extension. Since only the immediate context of OSS expressions are necessary to convert them to loops, the high-level functions need not be valid Lisp. Most often, just the OSS expression itself will be passed along.

Locating OSS expressions

After the Pascal code is converted to this pseudo-Lisp, OSS expressions are located and passed off to the Lisp OSS preprocessor. Some minor modifications must be made to put the code into the appropriate format for the preprocessor, such as prefacing functional arguments with `#'` and specifying the type of some of the variables. The above example yields:

```
(DEFUNS EJOBQTAILS (Q)
  (ENUMERATEF Q #'JOBQREST #'JOBQNULL :TYPE JOBQ))
```

For RemoveFromJobQ, only the body of the function is used, with the V-LET parameters turned into a form more palatable by the OSS preprocessor:

```
(LETS ((QS (EJOBQTAILS Q)))
  (LETS ((NEXTTOLAST (RLAST (TPREVIOUS QS) NIL)))
    (IF (= NEXTTOLAST NIL)
        (SETQ Q NIL)
        (SETF (FIELD NEXTTOLAST 'REST) NIL))
    (LETS ((LAST (RLAST QS NIL)))
      (SETQ REMOVEFROMJOBQ (FIELD (FOLLOW^ LAST) 'JOB))
      (DISPOSE LAST))))
```

OSS functions must be passed in their entirety so that the preprocessor can replace references to them in other segments of the code with the appropriate code in loop format. To simplify the identification of OSS functions in other parts of the code, the OSS functions are passed to the preprocessor first.

If OSS variables are declared, all lexically enclosed statements are passed to the preprocessor. Hence, the entire bodies of RemoveFromJobQ and SuperJob are collected, including statements in the code that do not involve OSS expressions. Had no OSS variables been declared, then only statements using OSS functions, either predeclared or defined elsewhere in the program, would have been returned. Although the code given here does not present any such examples, they are common in simpler applications of OSS series.

The OSS expression locator compiles a list of triggers as it searches through the declaration statements of a program. These triggers act as "cues" which signal that the code containing them should be sent to the OSS preprocessor. Any OSS variable, procedure, or function is appended to the list of triggers when declared.

When the locator encounters a code fragment that might need processing by the OSS preprocessor, it first searches to see if the expression contains a reference to a predefined OSS function. Barring that, it looks to see if there is a reference to a member of the list of triggers. If either of these two cases succeed, the code fragment is passed to the OSS preprocessor. Type information, conveyed by the :TYPE keyword, is also passed to the preprocessor.

Once this information has been found, code is ready to be passed to the OSS preprocessor. The preprocessor converts the code fragment into loops and then returns the new code with loops included where possible.

The code returned for RemoveFromJobQ is as follows:

```
(LET (NEXTTOLAST LAST QS #:OUT-3868 #:SHIFTED-3855
      #:STATE-3846 #:STATE-3856)
  (DECLARE (TYPE JOBQ NEXTTOLAST)
           (TYPE JOBQ LAST)
           (TYPE JOBQ QS)
           (TYPE JOBQ #:SHIFTED-3855)
           (TYPE JOBQ #:STATE-3846)
           (TYPE JOBQ #:STATE-3856))
  (TAGBODY
           (SETQ #:STATE-3846 Q)
           (SETQ #:STATE-3856 NIL)
           (SETQ NEXTTOLAST NIL)
           (SETQ LAST NIL)
  #:L-3870 (COND ((JOBQNULL #:STATE-3846) (GO OSS:END))
                 (T (SETQ QS #:STATE-3846)
                    (SETQ #:STATE-3846 (JOBQREST #:STATE-3846))))
           (SETQ #:SHIFTED-3855 #:STATE-3856)
           (SETQ #:STATE-3856 QS)
           (SETQ NEXTTOLAST #:SHIFTED-3855)
           (SETQ LAST QS)
           (GO #:L-3870)
   OSS:END )
  (SETQ REMOVEFROMJOBQ (FIELD (FOLLOW^ LAST) 'JOB))
  (DISPOSE LAST)
  (IF (= NEXTTOLAST NIL)
      (SETQ Q NIL)
      (SETF (FIELD NEXTTOLAST 'REST) NIL))
  )
```

AddToJobQ is not passed to the preprocessor because it contains no OSS expressions. The OSS functions EJobQTails and EJobQ are not returned, as their processed code is substituted for references to them in RemoveFromJobQ and SuperJob.

Unparsing

Once the substitution is made, the pseudo-Lisp is used to produce a Pascal program ready for compilation. In effect, a "de-parser" is being used. Using a Lisp pretty printer, an early version of which is described in [3], a dispatch table was built containing approximately forty simple rules to reconstruct the modified Pascal program from the pseudo-Lisp. For example, the rule to process `if...then...else` statements is contained in a dispatch entry that looks like this:

```
(set-dispatch-entry (cons-with-car if)
  #'(lambda (xp obj)
    (if (= (length obj) 4)
        (format* xp #"~!if ~W ~_then ~W ~_else ~W~." (cdr obj))
        (format* xp #"~!if ~W ~_then ~W~." (cdr obj)))) 0)
```

This rule triggers on any `cons` that has a `car` equal to `if`. It then tests to see how many elements are in the list; if there are four elements, as in `(if (= foo bar) (writeln foo) (writeln bar))`, it will be parsed into an `if...then...else` statement. Otherwise, the pseudo-Lisp is parsed into an `if...then` statement. The final number

5 indicates the *priority* of the rule relative to other rules in the dispatch table. The `~W` declarative, particular to the PP pretty printer, causes a dispatch of its argument. Thus, the `(= foo bar)` in the above example would be printed out as `foo = bar` after it triggered on a dispatch entry dealing with `=`.

Lower-priority rules handle more general functions, such as the conversion from the Lisp format of functions, `(function param1 param2 param3 ...)` to the Pascal format, `function(param1, param2, param3, ...)`.

Once printed back out again into Pascal, with the OSS expressions converted to loops, the code in the Job queue example looks like this:

```
label 1,2,3,4,5,6,7;

procedure AddtoJobQ (J: JobInfo; var Q: JobQ);
var E: ^JobQentry;
begin
  new(E);
  E^.job := JobInfo;
  E^.rest := Q;
  Q := E
end;

function RemoveFromJobQ (var Q: JobQ): JobInfo;
var NEXTTOLAST: JOBQ;
    LAST: JOBQ;
    QS: JOBQ;
    SHIFTED3812: JOBQ;
    STATE3803: JOBQ;
    STATE3813: JOBQ;
begin
  STATE3803 := Q;
  STATE3813 := nil;
  NEXTTOLAST := nil;
  LAST := nil;
 1:
   if JOBQNULL(STATE3803) then
      goto 2
      else begin QS := STATE3803;
                 STATE3803 := JOBQREST(STATE3803)
           end;
   SHIFTED3812 := STATE3813;
   STATE3813 := QS;
   NEXTTOLAST := SHIFTED3812;
   LAST := QS;
   goto 1;
 2:
  REMOVEFROMJOBQ := LAST^.JOB;
  if NEXTTOLAST = nil then Q := nil else NEXTTOLAST.REST := nil;
  dispose(last)
end;
```

```
function SuperJob (Q: JobQ): JobInfo;
var
   ITEM3869, JOBS, ITEMS3874: JOBINFO;
   ITEMS3829, ITEMS3873, STATE3828, STATE3872: JOBQ;
   ITEMS3838, ITEMS3847, NUM3837, NUM3846, OUT3863: REAL;
   ITEMS3858: BOOLEAN;
   COUNT: INTEGER;
begin
  STATE3828 := Q;
  COUNT := 0;
  NUM3837 := 0;
  NUM3846 := 0;
 3:
  if JOBQNULL(STATE3828) then
     goto 4
     else begin ITEMS3829 := STATE3828;
                STATE3828 := JOBQREST(STATE3828)
          end;
  JOBS := JOB.ITEMS3829;
  COUNT := COUNT + 1;
  ITEMS3838 := JOBS.PRIORITY;
  NUM3837 := NUM3837 + ITEMS3838;
  ITEMS3847 := SQR(JOBS.PRIORITY);
  NUM3846 := NUM3846 + ITEMS3847;
  goto 3;
 4:
  MEAN := NUM3837 / COUNT;
  DEVIATION := SQRT(NUM3846 / COUNT - SQR(MEAN));
  OUT3863 := MEAN + 2 * DEVIATION;
  STATE3872 := Q;
  ITEM3869 := nil;
 5:
   if JOBQNULL(STATE3872) then
      goto 7
      else begin ITEMS3873 := STATE3872;
                 STATE3872 := JOBQREST(STATE3872)
           end;
  ITEMS3874 := JOB.ITEMS3873;
  ITEMS3858 := ITEMS3874.PRIORITY > OUT3863;
  if not(ITEMS3858) then goto 6;
  ITEM3869 := ITEMS3874;
 6:
  goto 5;
 7:
  SUPERJOB := ITEM3869
end;
```

Conclusion

Use of OSS expressions in Pascal allows the programmer to avoid the compromise that must normally be made between readability and efficiency of code. While OSS expressions cannot guarantee that efficient code will automatically be produced, they are a significant step forward in permitting programmers to design code as they wish without having to

concern themselves with efficiency issues. Through the support of OSS expressions, these benefits can be obtained in any programming language.

Acknowledgements

Dr. Richard C. Waters played an instrumental part in nearly every aspect of this project. The theory involved is all his. For a more thorough explanation of the OSS macro package, see [4] and [5].

References

[1] K. Jensen and N. Wirth, "Pascal: User Manual and Report, 3rd ed.", Springer-Verlag, 1985.

[2] A. Goldberg and R. Paige, *Stream Processing*. Rutgers report LCSR-TR-46, Aug. 1983.

[3] R. Waters, "PP: A Lisp Pretty Printing System", MIT/AIM-816, December 1984.

[4] R. Waters, "Synchronizable Series Expressions: Part I: User's Manual for the OSS Macro Package", MIT/AIM-958, November 1987.

[5] R. Waters, "Synchronizable Series Expressions: Part II: Overview of the Theory and Implementation", MIT/AIM-959, November 1987.