EDOC???

EUROGAM PROJECT

NSF DATA ACQUISITION SYSTEM

Prototyping a Sort Language

Edition 1.0
November 1990

NSF Software Systems Group
UK Science and Engineering Research Council
Daresbury Laboratory

Prototyping a Sorting Language

David Brightly

November 1990

1 Introduction

This document shows, by way of an example, how the **T** system [1] can be used to prototype a small language and its implementation. **T** is a Lisp dialect deriving from the Scheme [2] language. The chosen example is the core part of a language designed for processing nuclear structure event data. The paper describes a grammar for the language together with a parser and evaluator written in **T**. The parser produces a parse tree for a program in the form of an s-expression. The evaluator animates the execution of the program by walking over the parse tree and recursively evaluating its sub-expressions.

2 A grammar for a simple sorting language

The grammar has been cast in a form suitable for recursive descent parsing. This is the grammar for expressions:

expr ::= [`not`] *relexpr* {(`and` | `or`) *relexpr*}
relexpr ::= *simpexpr* [(`<` | `<=` | `==` | `!=` | `>=` | `>`) *simpexpr*]
| *simpexpr* `in` *window*
simpexpr ::= [−] *term* {(`+` | `-`) *term* }
term ::= *factor* {(`*` | `/`) *factor* }
factor ::= *xfactor* | *id* `!` *indexpr*
indexpr ::= *xfactor* {`!` *xfactor*}
xfactor ::= *id* | (*expr*)

Note that the logical operators have equal precedence, as do the relational operators, and all binary operators associate to the left. Array indexing is a little out of the ordinary, using the BCPL-inspired `!` operator rather than the conventional brackets which are needed elsewhere.

Physicists like to talk about 'windows' and 'gates' so we have some syntax for some data structures representing these:

window ::= `[` *gate* {`,` *gate*} `]`
gate ::= *expr* `..` *expr*

This is the grammar for statements:

program ::= *block*
block ::= `begin` *statement* {; *statement*} `end`
statement ::= *simplestatement* | *block*
simplestatement ::= *assignstatement* | *incstatement* | *filterstatement* | *expression*
assignstatement ::= `set` *id* `=` *expr*
incstatement ::= `inc` *expr* `of` *specref*
specref ::= *id* | *id* `!` *indexpr*
filterstatement ::= `filter` *expr*

The semantics of these statement forms is explained below.

3 Parsing a program

The following recursive descent parsing routines were written directly from the grammar above. The tokens forming the source program to be parsed are symbols in the list `prog`. We use the **T** reader to do the tokenising (see the appendix). Each call to `scan` puts the next token of the program into the variable `sym` and decapitates `prog`.

```
(define openparen '\()
(define closeparen '\))
(define openbracket '\[)
(define closebracket '\])
(define semicolon '\;)
(define equals '\=)
(define bang '\!)
(define comma '\,)

(define (make-node s l r) (list s l r))
(define (make-unary-node s l) (list s l))
(define (make-leaf a) a)
(define (append-node l n) (append l (list n)))

(define (scan)
  (cond ((null? prog) (set sym eof))
        (else (set sym (car prog)) (set prog (cdr prog)))))
(define (lookahead) (if (null? prog) eof (car prog)))
(define (id? i) true)       ; temporary measure

(define (parse-error s) (print s (standard-output)) (ret (false)))
```

The next definitions form a mutually recursive set of parsers for sub-expressions. Each procedure returns an s-expression representing the parse tree for the sub-expression recognised. For example, the *simpexpr* `a + b + c` is represented by the s-expression `(+ (+ a b) c)`.

```
(define (expr) (logexpr))

(define (logexpr)
  (let ((l (cond ((eq? sym 'not) (scan) (make-unary-node 'not (relexpr)))
                 (else (relexpr)))))
    (labels (((relexpr* t)
              (cond ((eq? sym 'and) (scan) (relexpr* (make-node 'and t (relexpr))))
                    ((eq? sym 'or) (scan) (relexpr* (make-node 'or t (relexpr))))
                    (else t))))
            (relexpr* l))))

(define (relexpr)
  (let ((t (simpexpr)))
    (cond ((eq? sym '<) (scan) (make-node '< t (simpexpr)))
          ((eq? sym '<=) (scan) (make-node '<= t (simpexpr)))
          ((eq? sym '==) (scan) (make-node '== t (simpexpr)))
          ((eq? sym '!=) (scan) (make-node '!= t (simpexpr)))
          ((eq? sym '>=) (scan) (make-node '>= t (simpexpr)))
          ((eq? sym '>) (scan) (make-node '> t (simpexpr)))
          ((eq? sym 'in) (scan) (make-node 'in t (window)))
          (else t))))

(define (simpexpr)
  (let ((l (cond ((eq? sym '-) (scan) (make-unary-node '- (term)))
```

```
                 (else (term)))))
    (labels (((term* t)
              (cond ((eq? sym '+) (scan) (term* (make-node '+ t (term))))
                    ((eq? sym '-) (scan) (term* (make-node '- t (term))))
                    (else t))))
            (term* l))))

(define (term)
  (let ((l (factor)))
    (labels (((factor* t)
              (cond ((eq? sym '*) (scan) (factor* (make-node '* t (factor))))
                    ((eq? sym '/) (scan) (factor* (make-node '/ t (factor))))
                    (else t))))
            (factor* l))))

(define (factor)
  (cond ((and (id? sym) (eq? (lookahead) bang))
         (let ((l (id))) (scan) (make-node 'index l (indexpr))))
        (else (xfactor))))

(define (xfactor)
  (cond ((eq? sym openparen)
         (scan)
         (block0 (expr)
                 (cond ((eq? sym closeparen) (scan))
                       (else (parse-error "expecting ')'")))))
        (else (id))))

(define (id)
  (block0 (make-leaf sym) (scan)))
```

A reference to an array element such as `A!2!3`, more conventionally written as `A[2,3]` perhaps, is represented by the form `(INDEX A (2 3))`, i.e., an array name followed by a list of indexes.

```
(define (indexpr)
  (let ((l (xfactor)))
    (labels (((index* t)
              (cond ((eq? sym bang) (scan) (index* (append-node t (xfactor))))
                    (else t))))
            (index* (list l)))))
```

The language has a notation for one dimensional 'windows', such as `[100..200, 400..500, 800..900]` commonly required in processing event data. These are represented as a list of 2-element lists (or 'gates') such as `(WINDOW (100 200) (400 500) (800 900))`.

```
(define (window)
  (cond ((eq? sym openbracket) (scan))
        (else (parse-error "expecting '['")))
  (let ((l (list 'window (gate))))
    (labels (((gate* t)
              (cond ((eq? sym comma) (scan) (gate* (append-node t (gate))))
                           (else t))))
            (block0 (gate* l)
                    (cond ((eq? sym closebracket) (scan))
                          (else (parse-error "expecting ']'")))))))

(define (gate)
  (let ((l (expr)))
    (cond ((eq? sym '..) (scan) (list l (expr)))
          (else (parse-error "expecting '..'")))))
```

Multi-dimensional arrays of spectra provide a compact notation for expressing complicated selection criteria. We use the indexing notation to refer to an element in an array of spectra.

```
(define (specref)
  (cond ((and (id? sym) (eq? (lookahead) bang))
          (let ((l (id))) (scan) (make-node 'index l (indexpr))))
         (else (id))))
```

We now come to parsing procedures for program language statements. Statements are recognised by a distinguishing leading token and return special s-expressions. For example, the assignment statement `set a = b + c` is represented by the s-expression `(SET A (+ B C))`.

```
(define (statement)
  (case sym
    ((begin) (blk))
    ((inc) (inc-statement))
    ((set) (assign-statement))
    ((filter) (filter-statement))
    (else (expr-statement))))

(define (blk)
  (cond ((eq? sym 'begin) (scan))
        (else (parse-error "expecting 'begin'")))
  (let ((l (list 'block (statement))))
    (labels (((state* t)
              (cond ((eq? sym semicolon)
                      (scan) (state* (append-node t (statement))))
                     (else t))))
            (block0 (state* l)
                    (cond ((eq? sym 'end) (scan))
                          (else (parse-error "expecting 'end'")))))))

(define (inc-statement)
  (scan)
  (let ((l (list 'inc (expr))))
    (append-node l (cond ((eq? sym 'of) (scan) (specref))
                         (else (parse-error "expecting 'of'"))))))

(define (assign-statement)
  (scan)
  (let ((l (list 'set (id))))
    (append-node l (cond ((eq? sym equals) (scan) (expr))
                         (else (parse-error "expecting '='"))))))

(define (filter-statement)
  (scan)
  (list 'filter (expr)))

(define (expr-statement)
  (list 'expr (expr)))
```

Finally, a program as a whole is just a *block* so we can parse it with

```
(define (parse-program) (scan) (blk))
```

which returns an s-expression representing the program.

4 An example program

Our language has no facilities for reading in an event to be sorted, so we simulate this by an initial sub-block of assignments of arbitrary values to simple variables that can be taken as event components.

```
begin
  begin
    set a = 17 ;
    set b = 3 * 4 ;
    set c = 10 ;
    set d = 15 ;
    set e = 7 ;
    set g = 11 ;
    set h = 13
  end ;

  begin
    set h = 14 ;
    h < g ;
    inc h of s
  end ;

  begin
    filter b + 300 in [ 100 .. 200 , 300 .. 400 ] ;
    inc a + 255 of s ;
    begin
      a - b > 4 and d - c > 0 ;
      inc  a + b * c of t ! ( c * d  in [ 100 .. 200 , 300 .. 400 ] )
    end ;
    inc e * g + h of u ! ( b + c )
  end
end
```

5 An example parse tree

This is the list structure returned by `parse-program` applied to the above example:

```
(BLOCK (BLOCK (SET A 17)
              (SET B (* 3 4))
              (SET C 10)
              (SET D 15)
              (SET E 7)
              (SET G 11)
              (SET H 13))
       (BLOCK (SET H 14) (EXPR (< H G)) (INC H S))
       (BLOCK (FILTER (IN (+ B 300) (WINDOW (100 200) (300 400))))
              (INC (+ A 255) S)
              (BLOCK (EXPR (AND (> (- A B) 4) (> (- D C) 0)))
                     (INC (+ A (* B C))
                          (INDEX T ((IN (* C D) (WINDOW (100 200) (300 400)))))))
              (INC (+ (* E G) H) (INDEX U ((+ B C))))))
```

6 Evaluating the parsed program

In this section we will give some semantics for our little language by describing an evaluator for the s-expression returned by `parse-program`. First we need a means of recording the values held in program variables. Rather than invent new names for the program variables we can keep them separate from names used in the evaluator itself by using a **T** concept called a *locale*. For example, if we need to record that program variable `A` has value 23 we will bind the value 23 to the symbol `A` in the locale `symbols`.

```
(define symbols (make-empty-locale '*symbols*))
```

The heart of the evaluator is the procedure `eval-form` which takes an s-expression and recursively applies itself where necessary to its sub-expressions. When it reaches numeric or symbolic sub-expressions it either returns these directly or looks up their current value in the `symbols` locale.

```
(define (first l) (car l))
(define (second l) (cadr l))
(define (third l) (caddr l))

(define (eval-error s) (print s (standard-output)) (ret (false)))

(define (eval-form f)
  (cond ((symbol? f) (*value symbols f))
        ((number? f) f)
        ((list? f) (eval-list f))
        (else (eval-error "not a recognised form"))))
```

Sub-expressions in the form of lists are evaluated by dispatching on the symbol at their head:

```
(define (eval-list l)
  (case (car l)
    ((+) (add (eval-form (second l)) (eval-form (third l))))
    ((-) (if (eq? (length l) 3)
             (subtract (eval-form (second l)) (eval-form (third l)))
             (negate (eval-form (second l)))))
    ((*) (multiply (eval-form (second l)) (eval-form (third l))))
    ((/) (divide (eval-form (second l)) (eval-form (third l))))
    ((<) (if (less? (eval-form (second l)) (eval-form (third l))) 1 0))
    ((<=) (if (<= (eval-form (second l)) (eval-form (third l))) 1 0))
    ((==) (if (equal? (eval-form (second l)) (eval-form (third l))) 1 0))
    ((!=) (if (not-equal? (eval-form (second l)) (eval-form (third l))) 1 0))
    ((>=) (if (not-less? (eval-form (second l)) (eval-form (third l))) 1 0))
    ((>) (if (greater? (eval-form (second l)) (eval-form (third l))) 1 0))
    ((not) (if (zero? (eval-form (second l))) 1 0))
    ((and) (if (or (zero? (eval-form (second l)))
                   (zero? (eval-form (third l)))) 0 1))
    ((or) (if (and (zero? (eval-form (second l)))
                   (zero? (eval-form (third l)))) 0 1))
    ((block) (eval-block l))
    ((set) (eval-set l))
    ((filter) (eval-filter l))
    ((expr) (eval-expr l))
    ((inc) (eval-inc l))
    ((index) (eval-index l))
    ((in) (eval-in l))
    ((window) (eval-window l))
    (else (eval-error "not a known operator"))))
```

Note that the logical operators take non-zero to denote 'true' and zero to denote 'false', and always return 1 or 0 respectively.

The following procedures evaluate statement level forms. All statement level forms return a truth value which is used to determine the semantics of a block of statements.

An assignment form evaluates its third element and binds this value to the symbol given by its second element in the `symbols` locale, issuing a trace report as it does so, and returns TRUE:

```
(define (eval-set f)
  (let ((var (second f))
        (val (eval-form (third f))))
    (*lset symbols var val)
    (format (standard-output) "setting ~A to ~A~%" var val)
    true))
```

An expression form evaluates its second element and returns TRUE if the result is non-zero, else FALSE. A filter form does the same.

```
(define (eval-expr f)
  (not-zero? (eval-form (second f))))

(define (eval-filter f)
  (not-zero? (eval-form (second f))))
```

The simple language defined here has no explicit control constructs. Instead, a program is seen as a nest of statement blocks which themselves consist of a sequence of simple statements or further blocks. A block is evaluated by evaluating the statements or blocks within it in order until one of them returns FALSE. Any remaining statements or blocks within the block are then skipped. Blocks themselves always return TRUE. `eval-block` issues a trace report when statements are skipped in this way.

```
(define (eval-block f)
  (labels
   (((eval-block* l)
     (cond ((null? l) true)
           (else (cond ((eval-form (car l))  (eval-block* (cdr l)))
                       (else (format (standard-output)
                                     "abandoning block~%") true))))))
   (eval-block* (cdr f))))
```

Event sorting programs are evaluated for the side effect of incrementing channels in spectra. An increment form just reports that it is incrementing the channel given by the expression in its second element in the spectrum denoted by its third element. This third element must be either a symbol or an array element reference.

```
(define (eval-inc f)
  (format (standard-output) "incrementing channel ~a of spectrum ~a~%"
          (eval-form (second f)) (eval-spec (third f)))
  true)

(define (eval-spec f)
  (cond ((symbol? f) (format nil "~a" f))
        (else (eval-index f))))

(define (eval-index f)
  (if (not (list? f)) (eval-error "not an index expression"))
  (if (neq? 'index (first f)) (eval-error "not an index form"))
  (format nil "~a:~a" (second f) (map eval-form (third f))))
```

Finally, we give the semantics of the `in` operator which tests the value of an expression against a window and returns the number of the first gate of the window through which the value passes. A value 'passes' a gate if it is no less than the gate's lower bound and no greater than its upper bound.

```
(define (eval-in f)
  (window-test (eval-form (second f)) (third f)))

(define (window-test e w)
  (labels (((window-test* v i l)
            (cond ((null? l) 0)
                  ((and (>= v (first (first l)))  (<= v (second (first l))))  i)
                  (else (window-test* v (add i 1) (cdr l))))))
    (window-test* e 1 (cdr w))))
```

7 Conclusions

I hope that this paper has shown that it is practical to build a prototype of a problem-oriented language using **T**. The work described here (including preparation of the document itself) has taken about a week of elapsed time, starting with no previous experience of writing in a Lisp-like language, but with a passing acquaintance with [2]. This effort should be contrasted with the likely development time for a similar prototype in, say, C. One advantage of the **T** system is the ease with which a program can be developed incrementally. Both parser and evaluator were developed "bottom-up", by building on top of already checked-out lower-level procedures, virtually all of which were keyed in one at a time and checked out on small program fragments or s-expressions before moving on.

This rapidity of development means that you can afford to experiment with language constructs. Instead of spending weeks and months developing a compiler and run-time system from an abstract specification which intended users of the system cannot understand, you can sit down with users and demonstrate how their sorting requirements can (or cannot!) be expressed in the prototype language. Any doubts about the semantics of the language can be cleared up by watching the evaluator execute a sample program. The benefit from this, of course, is that you have more confidence that you have the right product when you finally come to build a production implementation of the language. And by then you already have a parser for the final form of the language and only need to replace the evaluator with a suitable code generator.

The little sorting language described here is clearly only a starting point for such a language and the implementation of the parser and evaluator takes lots of shortcuts which could not be permitted in a piece of production software. For example, the `scan` procedure returns unclassified tokens and the *id* parser, `id`, accepts any token as a valid identifier. Directions which could be investigated include:

- syntax for declarations, especially for arrays of spectra and simple variables, enabling run-time checking of array references;
- proper treatment of integer and real types in expressions;
- syntax for function application in expressions, and for user-declared functions;
- multi-dimensional spectra and windows (two at least);
- extended control constructs such as `case`;
- language constructs for dealing with permutations—especially valuable for highly symmetric event data.

8 Acknowledgements

I'd like to thank my colleague Dave Love for installing the **T** system at DL and for demonstrating that higher order languages can usefully be put to work in this environment.

References

[1] *The T Manual—Fifth Edition*, Rees, Adams and Meehan, Computer Science Department, Yale University, 1988

[2] *Structure and Interpretation of Computer Programs*, Abelson and Sussman, McGraw-Hill, 1985

A Appendix—Running the example program

Having loaded the parser and evaluator procedures we need to initialise the variable `prog` to a list of the tokens of our source program. To save effort writing a scanner we will use the **T** reader to lexically analyse the source program and produce a list of tokens for the parser. This makes the `scan` procedure trivial but we have to be careful to escape certain **T** reserved characters such as ‘`(`’ and ‘`;`’ and to leave white space around individual tokens.

```
(set prog '(
begin
  begin
    set a = 17 \;
    set b = 3 * 4 \;
    set c = 10 \;
    set d = 15 \;
    set e = 7 \;
    set g = 11 \;
    set h = 13
  end \;

  begin
    set h = 14 \;
    h < g \;
    inc h of s
  end \;

  begin
    filter b + 300 in \[ 100 .. 200 \, 300 .. 400 \] \;
    inc a + 255 of s \;
    begin
      a - b > 4 and d - c > 0 \;
      inc  a + b * c of t ! \( c * d  in \[ 100 .. 200 \, 300 .. 400 \] \)
    end \;
    inc e * g + h of u ! \( b + c \)
  end
end
))
```

We can now compile and execute our program by invoking

```
(eval-block (parse-program))
```

The trace reports we get are as follows:

```
setting A to 17
setting B to 12
setting C to 10
setting D to 15
setting E to 7
setting G to 11
setting H to 13
setting H to 14
abandoning block
incrementing channel 272 of spectrum S
incrementing channel 137 of spectrum T:(1)
incrementing channel 91 of spectrum U:(22)
```

The “abandoning block” report arises because the expression statement `(h < g)` returns FALSE.