April 1, 1986

Dr. Landauer:

Enclosed is a description of the time-reversible language JANUS, along with some sample programs. I showed these to you briefly when you spoke here last week on physical limits of computation.

JANUS was written by myself and Howard Derby for a class at Caltech in 1982 (or thereabouts). The class had nothing to do directly with this project. We did it out of curiosity over whether such an odd animal as this was possible, and because we were interested in knowing where we put information when we programmed. JANUS forced us to pay attention to where our bits went since none could be thrown away.

JANUS was fully implemented as described as is bug-free (to our knowledge) despite the disclaimer in the document that it is a “throw-away piece of code.” If it still exists, it is on backup tapes somewhere at Caltech. Hope you find it interesting.

sincerely,
Chris Lutz
IBM Almaden Research Center
K33/801
(LUTZ@ALMVMA)

JANUS: A TIME-REVERSIBLE LANGUAGE

By Christopher Lutz and Howard Derby, circa 1982

JANUS is a compiler and interpreter for the time-reversible language JANUS. The JANUS compiler is written in SLIMEULA, and compiles the code into an internal SLIMEULA Class structure which can be interpreted directly. ‘SLIMEULA’ means SIMULA running on a DECSYSTEM-20.

JANUS is considered to be a throw-away piece of code. It will not be maintained and is not purported to be robust.

The compiler consists of four major parts: A lexical analyzer which tokenizes the input stream and generates a symbol table; a recursive descent parser made of the Init Code of SLIMEULA Classes; an interpreter which consists of the procedure ’exec’ common to all of the Classes created in the parsing; and the runtime command scanner.

A. Lexical Analyzer

The lexical analyzer tokenizes the input stream and sets the global variables token, `token_value`, and `token_type` in accordance with the current token. The following terminal classes are recognized:

`ident`:
An identifier, which is any sequence of letters which is not a keyword. If it has not been encountered before, it is inserted into the symbol table. An identifier may name a procedure, a variable, or both.

`num`:
A number, which is any sequence of decimal digits.

`binop`:
A binary operator, which is any of `+`, `-`, `!` (exclusive OR), `<`, `>`, `&` (logical AND), `|` (logical OR), `=`, `#` (not equals), `<=`, `>=`, `*` (multiply), `/` (divide), and `\` (remainder).

A semicolon in a program line indicates that the rest of the line is a comment.

Lower case letters are converted to upper case on input.

B. Parser, and JANUS Language Syntax

The recursive descent parser makes use of a SLIMEULA Class for every node in the parse tree, except for the root node. The root node is a subroutine which corresponds to the nonterminal Program in the grammar below. Every other nonterminal corresponds exactly to a Class of the same name. This correspondence is the reason for the slightly atypical presentation of the grammar.

The grammar of the JANUS language is as follows:

```
   [Terminals in quotes or lower case.
    { }* indicates zero or more repetitions.
    [ ]  indicates zero or one repetitions.]

Program ::=           {  ident [ ’[’ num ’]’ ]  }*
                      { ’PROCEDURE’ ident Statements }*

Statements ::=         Ifstmt      Statements
                    |  Dostmt      Statements
                    |  Callstmt    Statements
                    |  Readstmt    Statements
                    |  Writestmt   Statements
                    |  Lvalstmt    Statements
                    |   null

Ifstmt ::=            ’IF’    Expression
                    [ ’THEN’  Statements ]
                    [ ’ELSE’  Statements ]
                      ’FI’    Expression

Dostmt ::=            ’FROM’  Expression
                    [ ’DO’    Statements ]
                    [ ’LOOP’  Statements ]
                      ’UNTIL’ Expression

Callstmt ::=          ’CALL’   ident
                    | ’UNCALL’ ident

Readstmt ::=          ’READ’ ident

Writestmt ::=         ’WRITE’ ident

Lvalstmt ::=          Lvalue Modstmt
                    | Lvalue Swapstmt

Modstmt ::=           ’+=’ Expression
                    | ’-=’ Expression
                    | ’!=’ Expression

Swapstmt ::=          ’:’ Lvalue

Expression ::=        Minexp
                    | Minexp binop Expression

Minexp ::=            ’(’ Expression ’)’
                    | ’-’ Expression
                    | ’~’ Expression
                    | Lvalue
                    | Constant

Lvalue ::=            ident
                    | ident ’[’ Expression ’]’

Constant ::=          num
```

For every (`ident [ ’[’ num ’]’ ]`) encountered, the parser’s root node (procedure parseprog) creates an instance of Class `dotaarray`, which contains an array for the storage corre-

This article is a part of the personal letter from Christopher Lutz to Rolf William Landauer in 1986. Under his permission, this article is placed at `http://tetsuo.jp/ref/janus.html` as of today (Aug 13, 2010). All rights belong to Christopher Lutz.

sponding to the variable `ident`. The array is of size `num`, and defaults to size 1 when `num` is not included. A pointer to this class is placed in the variable column of the symbol table entry for `ident`.

For every (`'PROCEDURE' ident Statements`) encountered, the root node creates an instance of Class `Statements`, which parses `Statements`. A pointer to this class is placed in the procedure column of the symbol table entry for `ident`.

C. Semantics

C.1 Variables

Every variable is contained in an array of integers. All arrays are named by an ident and must be declared in the (`ident [ '[' num ']' ]`) section of the program. The size of the array is `num` unless `'[' num ']'` is omitted, in which case the size defaults to 1. Array indices go from zero to arraysize−1. All variables are global. In the second part of the program, a reference to `ident` without a subscript is equivalent to `ident[0]`. All variables are initialized to zero.

C.2 Expressions

Expressions are evaluated with signed integer arithmetic. The result of the operators =, *, <, >, <=, and >= is either `0` (for FALSE) or `-1` (for TRUE). All operators have the same precedence. Expressions are evaluated from left to right, except for parenthetization. The unary operators `-` (negation) and `~` (logical NOT) bind tightly.

C.3 Modification Operators and the Swap Operator

The modification operators (`+=`, `-=`, and `!=`) and the swap operator (`:`) are the only means for changing the value of variables.

The modification operators evaluate the expression on the right, and modify the variable on the left according to the operator. `+=` adds the expression into the variable. `-=` subtracts it out. `!=` exclusive-or's it in. The expression may not contain the `ident` of the variable on the left, since that event could potentially specify a singular operation.

The swap operator swaps the values of the variables on its left and right. The `ident`'s of the variables may appear in the expression in the subscript of neither variable, since that event could potentially specify a singular operation.

C.4 Control Structures

`Ifstmt` is the analog of the conventional `IF` statement. On entry, the Expression after `'IF'` is evaluated. If it is true (non-zero) then the `Statements` following `'THEN'` is executed, if there is a `'THEN'` clause. Otherwise the `Statements` after `'ELSE'` is executed, if there is an `'ELSE'` clause. The expression after `'FI'` is then evaluated an verified to have the same truth as that after the `'IF'`. If this is not the case, an error condition has resulted, and the program halts, returning control to the command interpreter.

`Dostmt` is the analog of the conventional `DO` statement. On entry, the Expression after `'FROM'` is evaluated, and verified to be true (non-zero). If this is not the case, an error condition has resulted. The `Statements` after `'DO'` is then executed, followed by evaluating the Expression after `'UNTIL'`. If the expression is true, the `'DO'` structure is exited. If it is false, the `Statements` after `'LOOP'` is executed. The Expression after `'FROM'` is then reevaluated, and verified to be false (equal to zero) on penalty of an error conditions. The `Statements` after `'DO'` is executed again, and control loops in this manner until the `'UNTIL' expression` is found to be true.

C.5 Error Conditions and Time-Reversibility

When an attempt is made to perform an operation that is singular (destroys information and so cannot be reversed) an "error condition" results. An error condition could be resolved by reversing the direction of execution of the program at the point of the attempted singular operation, but since error conditions are considered abnormal and do not normally happen in "working" programs, the run-time system responds by simply halting execution of the program. This allows examination of the state at the time of the error condition.

Subscripts out of range are considered an error condition.

C.6 Procedures

All executable statements are contained in exactly one procedure, which is named by the `ident` following `'PROCEDURE'`. A procedure can be executed in the foreward direction by the `'CALL' ident` statement, or in reverse by the `'UNCALL' ident` statement. Note that the direction of execution is toggled each time `'UNCALL'` is used. For example, a procedure which is `'UNCALL'`ed from a procedure which is itself `'UNCALL'`ed from the top level is executed from top to bottom.

Procedures are infinitely recursive, and foreward references are allowed.

C.7 Read and Write

The `'READ'` statement can be considered a swap between a variable and the outside world. It prints the current value of the variable, and replaces it with the `num` typed by the user to the run-time system. If it is to be time-reversible, when executed backwards, the user must retype its previous value when it was executed forwards.

The `'WRITE'` statement simply types the value of the variable.

C.8 Run Time System

The JANUS run time system is provided to allow the user to initialize and inspect variables, call or uncall procedures, and otherwise control the interpretation of his code. When JANUS is run, it will respond with the query "`file?`", asking for the name of the file to be parsed. After parsing, JANUS will prompt with ">". Commands may be typed to JANUS in either upper or lower case. The commands accepted by JANUS are:

`var[index]`
  Prints out the value of `var[index]`

`var`
  Prints out all elements of array `var`

`var=n`
  Sets `var[0]` to `n`.

`var[index]=n`
  Sets `var[index]` to `n`.

`CALL name`
  Calls procedure name.

`UNCALL name`
  Uncalls procedure name.

`SYMBOLS`
  Types table of all symbols and their attributes.

`TRACE`
  Turns on the trace feature. Lists statements as they are executed, along with the values of any variables that are modified.

`UNTRACE`
  Turns off the trace feature.

`RESET`
  Resets all variables to zero.

`RESET var`
  Resets all elements of array var to zero.

```
; Factorization program in the time reversible language Janus

num     ;Number to factor.  Ends up zero
try     ;Attempted factor.  Starts and ends zero
z       ;Temporary.  Starts and ends zero
i       ;Pointer to last factor in factor table.  Starts zero
fact[20];Factor table.  Starts zero. Ends with factors in ascending order

procedure factor                    ;factor num into table in fact[]
      from  (try=0) & (num>1)
      loop  call nexttry
            from  fact[i]#try                ;Divide out all occurrences of this
            loop  i += 1                     ; factor
                  fact[i] += try
                  z += num/try
                  z : num
                  z -= num*try
            until (num\try)#0
      until (try*try)>num                    ;Exit early if possible

      if    num # 1
      then  i += 1                           ;Put last prime away, if not done
            fact[i] : num                    ; and zero num
      else  num -= 1
      fi    fact[i] # fact[i-1]

      if    (fact[i-1]*fact[i-1]) < fact[i]  ;Zero try
      then  from  (try*try) > fact[i]
            loop  uncall nexttry
            until try=0
      else  try -= fact[i-1]
      fi    (fact[i-1]*fact[i-1]) < fact[i]

      call  zeroi                            ;Zero i

procedure zeroi
      from  fact[i+1] = 0
      loop  i -= 1
      until i = 0

procedure nexttry
      try += 2
      if    try=4
      then  try -= 1
      fi    try=3

procedure readf           ;Load table of factors (to be multiplied by
      read  i             ;  uncalling procedure factor)
      from  z=0
      loop  z += 1
            read fact[z]
      until i=z
      z -= i     ;zero z
      call zeroi
```

```
list[12]    ;List to sort
perm[12]    ;Permutation done during sort.  Initially the identity permulation
n           ;Number of numbers
i  j        ;Loop counters

procedure sort     ;Bubble sort list, permuting perm.
     from  i=0
     loop  j += n-2
           from  j=n-2
           loop  if    list[j] > list[j+1]
                 then  list[j] : list[j+1]
                       perm[j] : perm[j+1]
                 fi    perm[j] > perm[j+1]
                 j -= 1
           until j=i-1
           j -= i-1
           i += 1
     until i=n-1
     i -= n-1

procedure makeidperm   ;Add identity permutation to perm. Use to initialize perm
     from  i=0
     loop  perm[i] += i
           i += 1
     until i=n
     i -= n

procedure readlist     ;Use to initialize list by reading each entry from terminal
     from  j=0
     loop  read list[i]
           i += 1
     until i=n
     i -= n
```

```
num root   z bit

procedure root        ;root := floor (sqrt(num))
     bit += 1
     from  bit=1     ;find exponential ball park
     loop  call doublebit
     until (bit*bit)>num
     from  (bit*bit)>num
     do    uncall doublebit
           if    ((root+bit)*(root+bit))<=num
           then  root += bit
           fi    ((root/bit)\2) # 0
     until bit=1
     bit -= 1
     num -= root*root

procedure doublebit
     z += bit
     bit += z
     z -= bit/2
```