

MidiTrans - A MIDI File Transform Language

Roger T. Hartley
Computer Science Department
New Mexico State University
rth@cs.nmsu.edu

Abstract

MidiTrans is an application built on a visual language that supports general processing of standard MIDI files. Unlike most software sequencer programs that have a fixed repertoire of transforms, MidiTrans allows any transform to be implemented using the elements of a visual data-flow language.

MidiTrans allows conditional transforming, and thus has much more of the flavor of a programming language. It allows transforms like: "In this interval in these tracks, if the top note is less than a fifth away from the next highest note, then transpose it up an octave, unless this results too high a note".

A MIDI file is processed by MidiTrans as a structured stream of MIDI commands using the generator/collector metaphor. Arrays and lists are supported as polymorphic structures, and the basic data types include MIDI events and timestamps, both important in MIDI files. A comparison with OpCode's MAX, and Cakewalk's CAL is made to illustrate the power and ease of use of MidiTrans, which runs on Windows 95 or NT platforms.

Introduction

Sequencer Transforms

A typical software sequencer is basically an editor for MIDI (or MIDI-related) events. As well as basic operations like inserting a note, deleting notes, and various kinds of data moves, a sequencer usually allows for global editing operations called transforms. Typical operations are:

- quantize - move note events in time so that they "snap" to a chosen time interval.
- transpose - alter pitches of note events by a constant amount, either up or down.
- slide - move the start time of all events forwards or backwards.
- modify velocity - alter the velocity of all notes (basically how hard the note is played) by a constant amount, either less or more.

Usually a transform can be made to operate on some portion of the file, and not the whole of it, for instance in Cakewalk [1].

The MIDI Data Stream

MIDI (Musical Instrument Digital Interface) data is a stream of bytes interpreted as a sequence of commands. Each command tells a MIDI-capable instrument, such as a synthesizer or sound module, to produce its sounds in particular ways. The commands are of two kinds:

- channel commands - the basic stuff of MIDI. Channel commands include NoteOn, which tells the sound module

to start a note of a particular pitch, and ProgramChange which selects an instrument type to produce the sound.

- system commands - these concern the timing of channel commands and the control and monitoring of the sound module.

For more detail see the MIDI 1.0 specification [2] which includes a description of the necessary electrical connections as well as the MIDI protocol itself.

Standard MIDI Files

The specification of MIDI originally covered only the real-time connection between musical instruments. Later it was clear that the computer could control the instrument through a MIDI interface through programs called sequencers (they produce sequences of MIDI commands). The next step was a method of storing and exchanging MIDI sequences. The Standard MIDI File format [3] was developed to encapsulate MIDI sequences, and to include all the information concerning the performance of the sequence that was previously only implicit in the real-time nature of the MIDI stream.

The SMF format includes the following components:

- The track - similar to a track on an analog tape. There can be several tracks in the sequence that are intended to be played simultaneously.
- MIDI events - any command from the MIDI specification can be included in a standard MIDI file, along with a timestamp that indicates the time at which the command is to be issued.
- Meta-events, which include:
 - Tempo, time and key signature changes.
 - Lyrics and arbitrary text such as instrument names.
 - Marker and cue points for rehearsal, and synchronization to other media, such as video.

Abstract Syntax for Standard MIDI Files

MidiTrans processes a standard MIDI file as an abstract data structure, not as a byte stream. This ensures that the format of the MIDI file is adhered to, so that all files produced by MidiTrans are valid, and can be performed by an appropriate MIDI sound module. To illustrate the structure of a MIDI file, we shall give an abstract syntax. The syntactic categories are:

F = File

FH = File Header

T = Track

TH = Track Header

TT = Track Trailer

S = Timestamp

C = MIDI event

N = number of tracks

F = format (0, 1 or 2)

R = resolution - the smallest subdivision of a quarter note, usually in the range 96 - 2048.

FL = length of the file in bytes

TL = length of track in bytes

TN = track number

M = MIDI event

X = meta-event

The abstract syntax rules are then:

```
F ::= FH { T }1
T ::= TH { S C }0 TT
FH ::= FL N F D
TH ::= TL TN
C ::= M | X
M ::= NoteOn | NoteOff | ProgramChange | Controller | PitchBend | Pressure
X ::= Tempo | KeySignature | TimeSignature | Text | SystemExclusive | Marker | TimeCode
S ::= a number of ticks since the last event (see R)
```

The MidiTrans Virtual Machine

Overview

The highest level abstraction in the virtual machine is that of a dataflow graph. A MidiTrans program is structured as a collection of such graphs each having many roots, which can be exit or entry nodes. Cycles are allowed, but each graph must be fully connected, i.e. every node must be connected to at least one other. Each graph in the collection represents a subprogram, and contains a distinguished node which serves as the main entry point. One graph is considered special and is the “main” program, much as the main function in C. This function must have the same name as the file in which the program is stored. The nodes in a graph represent operations, either MIDI operations, such as noteon and noteoff, or commands which handle the computing chores. The links between the operation nodes carry data. Data are values taken from a pre-defined set of types. Type groups include numbers, strings and symbols, but also the specialized MIDI types midi-event, time, and pitch, among others.

Program execution is a breadth-first traversal of the dataflow graph, where the nodes are held in a priority queue before execution. A node is *firable* when all of its inputs are present and it is a descendent of a node that has just fired, otherwise it has to wait for more inputs. Inputs may be *latched*, in which case their values are retained for a subsequent firing of that node, otherwise the node consumes the values and they are forgotten. When a node fires, it is removed from the queue, unless it is a *state* operation (i.e it stores internal values), such as a generator, which we will describe later in detail. If a node representing a subprogram is encountered, the queue is suspended while the subgraph is traversed. A normal stack mechanism handles subprograms, unless it contains a state operation, when its stack record must be preserved when the subprogram exits. Again, we will describe this mechanism later.

Events and Commands

Following the Postscript model, MidiTrans has a common representation for both output events and program commands. For example, the postscript command to draw a line, a display event, has the same syntax as the command to add two numbers:

```
%% draw a line
100 200 lineto
%% add two numbers
100 200 add
```

As a comparison, here is a play event from MidiTrans, and an add command:

```
48 85 noteon
48 85 add
```

Although it appears that the syntax follows the postfix model of Postscript, in fact, the concrete syntax of MidiTrans has a model that mixes standard infix format, and a dataflow overlay, as we shall see later. For a preview, a simple two note sequence can be computed and played by attaching the MIDI event, with timestamp to the end of a list and then sending the list to a MIDI output device with the operation play:

```

createlist | *1001 attach | *1002 attach | play ]
85 64 120 1 note | 0 timestamp *1001 ]
120 30 add *1011 ]
85 64 *1011 1 note | 96 timestamp *1002 ]

```

The *note* operation takes three inputs, in the order pitch, velocity, channel and duration. The notation *n, where n is an integer, denotes a value shared by more than one operation. A MidiTrans program has the structure of a dataflow graph, with many roots. In the example, the operations createlist, the first noteon and noteoff, and add are roots of the graph, and play is a terminal node.

Implementation Data Types

Here we give an implementation syntax for the data types used in MidiTrans. The simple types are:

- *short integer*
- *long integer*
- *pitch*
- *timestamp*
- *pitch interval*
- *time interval*
- *string*
- *symbol*
- *boolean*

The structured types are:

- *array*, written as [*T*], where *T* is the type of the array's elements.
- *list*, written as (*T*), where *T* is the type of the list's elements (it may be a generic type for polymorphic lists, or another structured type).
- *tuple*, written as < *T1*, *T2*, ... >, where *T1*, *T2*, etc. are the (possibly different) types of the tuple's elements.
- *pair*, written as *symbol* . *T*, where *T* is the type of the value associated with the symbol.
- *property list*, written as a tuple of pairs: < *pair1*, *pair2*, ... >
- *sequence*, which is essentially a lazy list, whose values can only be used one at a time
- *file*, which is the type associated with any MIDI file

Since the type *file* is so important, we will explain it in more detail. We will use the symbol *D* to denote a function mapping abstract syntax to data structure. The data structure for *file* is implemented in MidiTrans as follows:

- $D(F) = (D(T))$, i.e. a file is a list of tracks
- $D(T) = (D(TH), event, D(TT))$, i.e. a track is a list of events, each having a timestamp
- $event = < D(S), D(C) >$

Note that measure and event are not part of the abstract syntax; they are only implementation structures. The types array, list and tuple in MidiTrans are dynamic and polymorphic; they not only support the virtual machine, but are also available as programming constructs.

A Visual Dataflow Language

The MidiTrans virtual machine is modeled after Postscript [4] in that it incorporates the basic command set (the categories M and X above) into a general computational framework. However, there the similarity ends, because, whereas Postscript is a postfix stack-based language, MidiTrans is a visual data-flow language. Each operation is shown as a labeled rectangle with one or more inputs, called sockets, and one or more outputs called plugs. Data flows between operations along wires that connect plugs to sockets. Below (Figure 1) is a generic operation showing three plugs and three sockets. One plug and one socket is designated as the primary connector in order to distinguish main data flows from subsidiary data flows, or control inputs and outputs. There can, however, be more than one primary input and/or output in special cases. Inputs may be latched, indicated by a solid rectangle on the connection.

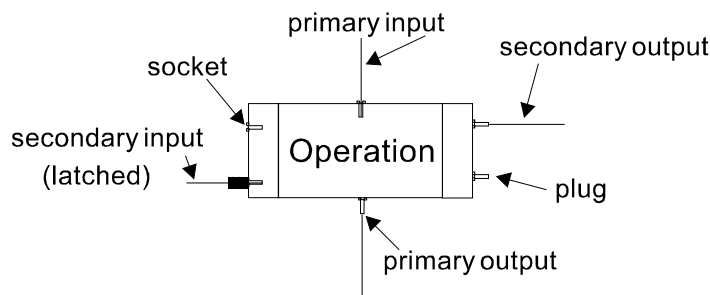


Figure 1. A Generic Miditrans Operation

User Data Types

MidiTrans has a full range of built-in types: integer, boolean, symbol, string for general computation and pitch, time, time-interval, and pitch-interval for MIDI computation and representation. Lists and arrays, the structured types, are polymorphic and dynamic. i.e. they can change their size at run-time, and store any object of built-in type, as well as objects of other structured type.

Lists are doubly linked and have the operations createlist, push, pop, attach, detach, head and tail associated with them. Push, pop and head operate at the front of the list, while attach, detach and tail operate at the rear.

Arrays are zero-based variable length structures with the operations createarray, set, and get.

A property list is attached to an object, and can store any number of key/value pairs for arbitrary use. The operations are createobj, setvalue and getvalue. Keys are of type *symbol*.

A file is an array of tracks, each of which is an array of measures, each of which in turn is a list of MIDI events. The operations filein, fileout, usefile, createfile, addtrack, and gettrack operate on these special structures.

A sequence is an abstract data type used specifically for iteration (see generate/collect below). Its operations are

The Operation Set

The set of operations splits into two main groups: *MIDI* operations and *dataflow* operations. The MIDI operations are:

aftertouch	channel pressure	controller
fileheader	keysignature	note
noteoff	noteon	pitchbend

progchange	sequence	sysex
tempo	text	timesignature
trackheader	tracktrailer	

All of these operations produce a MIDI event (without timestamp) from the appropriate inputs. For instance, a *noteon* operation with inputs of channel number, pitch, and velocity produces the event with these parameters at its output. *note* has both *noteon* and *noteoff* parameters since it has a start time and a duration. A timestamp may be added by adding a `TIMESTAMP` property to the event with `setvalue` (see generic object operations, below).

The other set of operations comprises the dataflow operations:

Arithmetic:

add	sub	div
mult	mod	

Arithmetic is done on either short or long integers, or a mixture of both. There is no floating point type.

List operations:

createlist	attach	detach
pop	push	head
tail		

A list is a double-ended queue, which can be accessed from either end, but not at any intermediate element. Pop, push and head access the left-most element; attach, detach and tail access the right-most element. List elements can contain any type of value.

Array operations:

createarray	get	set
-------------	-----	-----

Arrays are dynamic, polymorphic structures. An array must be created with some number of elements initially, but it will automatically grow if necessary. Elements can be any type and are accessed in the usual fashion with an integer index value.

File operations:

createfile	addtrack	gettrack
settrack	filein	fileout
usefile		

Since a file is actually represented by a list of tracks, *addtrack* and *gettrack* are special operations that only handle files and tracks. *filein* and *fileout* are the read and write operations respectively and *usefile* is a read operation when the file has already been loaded.

Relational:

equal	gte	lt
-------	-----	----

equal, *gte* and *lt* (greater than or equal and less than) only handle numeric input, producing a boolean output

Logical:

not and or

These are the usual boolean operations.

Generators/collectors:

generate collect createseq

We will save description of these special operations until later.

Generic object operations:

createobj getvalue setvalue

An *object* is really just a property list (objects cannot have values as they do in Common Lisp), where properties can be any symbol and values can be any value of any type. A MIDI event may be computed by creating an object, putting appropriate values on its property list, and then inserting (for instance) into a measure list. Properties like CHANNEL, VELOCITY, PROGRAMNUMBER, PRESSURE and TIMESTAMP are built-in constants and can be used to add and retrieve values to and from objects. There is an implicit conversion between such objects and the values produced by the MIDI operations (see above), so that *getvalue* and *setvalue* will work with these as well.

Pitch operations:

interval pitchname transpose

Interval takes two pitches and produces a numeric interval equal to the number of semitones between the pitches. It can be positive or negative according to the ordering of the inputs. *Pitchname* produces a conventional note/octave/accidental string for the input pitch value. *Transpose* takes a pitch and an interval and produces a new pitch which is the input pitch transposed by the interval.

Generic dataflow operations:

gate delay join

Gate is the generic conditional operation; it only passes its primary input through to the output when its control input is true. *Delay*, delays its input by storing its value until the operation fires again, when it stores the new value and passes on the old one. *Join* effectively merges its two inputs by firing when only one of its inputs is present

Output:

print show

Print simply displays its argument in a results window and passes it through unchanged. Any value may be printed, but structured values such as arrays only their type. *Show* displays its value by replacing the label in the operation's box, and also passes it through unchanged. It is mainly used for tracing a program's execution.

In addition to these operations, user procedures, called *functions*, may be defined by naming a dataflow graph.

Executing a MidiTrans program

As outlined above, a MidiTrans program is executed by a depth-first traversal of the dataflow diagram, starting with a user-defined operation that has the same name as the file in which the program is stored. For instance, the function *gennotes* is stored in a file called *gennotes.mt*. An operation will only fire when all its inputs are present. It then produces an output (or not, as appropriate) and this output becomes the input for another operation. Since operations connected to its output are descendents in a traversal sense, then the operations immediately affected by the firing are considered next. If an operation fails to fire, it cannot have any descendents, thus stopping the dataflow. Whether it fires or not, it is removed from the queue. When the queue empties, then the program halts.

Function calls are implemented by suspending the depth-first traversal in one dataflow graph and traversing the one in the called function (recursion is allowed). Apart from function call, control is implemented through the special operation *gate* which basically operates like a dataflow switch. MidiTrans also implements a form of iteration. Since a dataflow graph contains no explicit control structures, iteration is implemented in a functional way, use generators [5] (or iterators as they are called in C++ [6]).

Control

The operation *gate* operates like a dataflow version of the if-then-else form in procedural languages. It has two inputs, a primary, which is the value to be switched, and a boolean control which is either true or false. If the control is true, the the

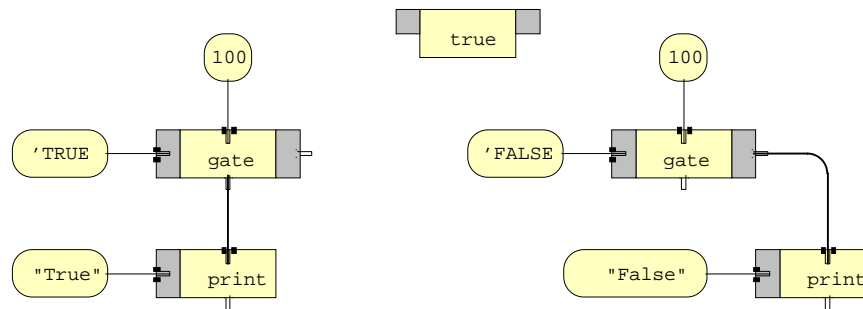


Figure 2. The gate operation: the left one prints "True" and the right one prints "False"

primary input is copied to the primary output. If it is false, then the input is switched to the secondary output. In both cases, the other output remains unbound, i.e. there can be no descendents along that branch. Figure 2 shows simple examples of each case.

Generators

A generator is an operation that accepts any object of structured type as its primary input, and, on each new firing, produces a single element of the structure at its output. For an array or a list, this is the object stored in each successive element. For a property list it is the successive key/value pairs (called properties). For a file it is the tracks that make up the file, and for a sequence, created with *createseq*, it is the succession of integers from lower bound to upper bound set when the sequence was created. A generator can fail to fire if its control input is false, or if the input is absent. The secondary output is false whenever the generator succeeds and turns true when the generation is over. A generator can only be restarted by feeding it a new structured object. Figure 3 is a simple sequence generator that prints the elements of an array from 1 to 10. The inputs to *createseq* are lower bound (1), upper bound (10) and step value (1). Its output is a sequence value which feeds the generator. When the generator fires for the first time it produces a value of 1, the lower bound of the sequence. This value is printed. Normally this would terminate execution, since *print* has no descendents, but since the generator is a state operation, it fires again. In fact it is kept on the execution queue. This time it produces 2 (stored value plus the step value), which is again printed. This iteration proceeds, printing 3 to 10. Now the generator

fires, but produces no output and switches itself off, thus ending the execution; exhausted generators are removed from the queue. It also produces a secondary output value of true, indicating that its last value has been produced, but in this tiny example the value is not used. Better examples of generation, and also collection are coming up.

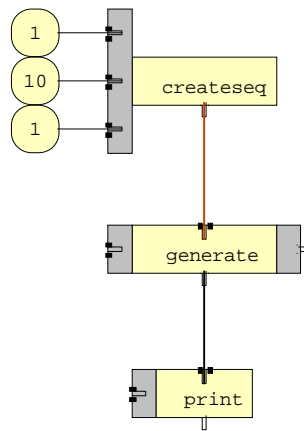


Figure 3. An example of a generator

Collectors

The collect operation can work hand-in-hand with a generator to accumulate values in various useful ways. Figure 4 shows a collector summing the numbers given it by a generator. In this example, the collector switches off when its internal value (monitored with every collection through the secondary output) reaches a specific amount. The generator also switches off early in this case. Other variations on collecting include counting, minimum value, maximum value and remembering the last value received. These are indicated by a different symbol in the top-most secondary input.

The MidiTrans program syntax

Although MidiTrans programs are typically created and edited in the graphical environment described in the next section, MidiTrans also has an ASCII representation that can be useful in some cases. All MidiTrans programs are stored in disk files in this format, so it is essential to know how this works. The comparison to Postscript is again apparent. Postscript can be used purely as a page description language and only handled internally by graphical applications. Typically a drawing program will translate mouse or stylus gestures into Postscript commands, and store these in ASCII form in file so that the drawing produced by the gestures may be reproduced later by replaying the commands in the file. However, Postscript is also a general-purpose computing environment, and many extraordinary effects may be programmed by writing Postscript code directly.

MidiTrans is similar in that commands (MIDI operations) may be stored in ASCII form and played back later, producing a musical performance. However, MidiTrans is not a sequencer, and cannot translate keyboard gestures into commands; it assumes this has already been done. What it does is to allow manipulations of these translated MIDI gestures into other forms through a general-purpose computing environment. It is the ASCII representation of these manipulations, the stored form of the dataflow graph that we will describe now.

Linearization of a graph

A graph is a two dimensional description, whereas text only really has one dimension- left to right. Since a MidiTrans dataflow graph can have many entry points, each entry point must correspond to a separate piece of text. It can also contain cycles, and these cycles must be broken in order to write them out in a linear form. The trick is to turn the graph

into a *forest* of trees, each of which has only one root. Where a leaf of a tree joins another tree (or itself), the link must be cut, and a marker must be placed at both ends of the cut. Thus the graph below (Figure 5) on the left (drawn in brief summary form with a node being a dot, not a labeled box) can be linearized as the *set* of graphs on the right.

Note how each successive cut point separates the graph into two. It only remains to indicate the cuts a textual form.

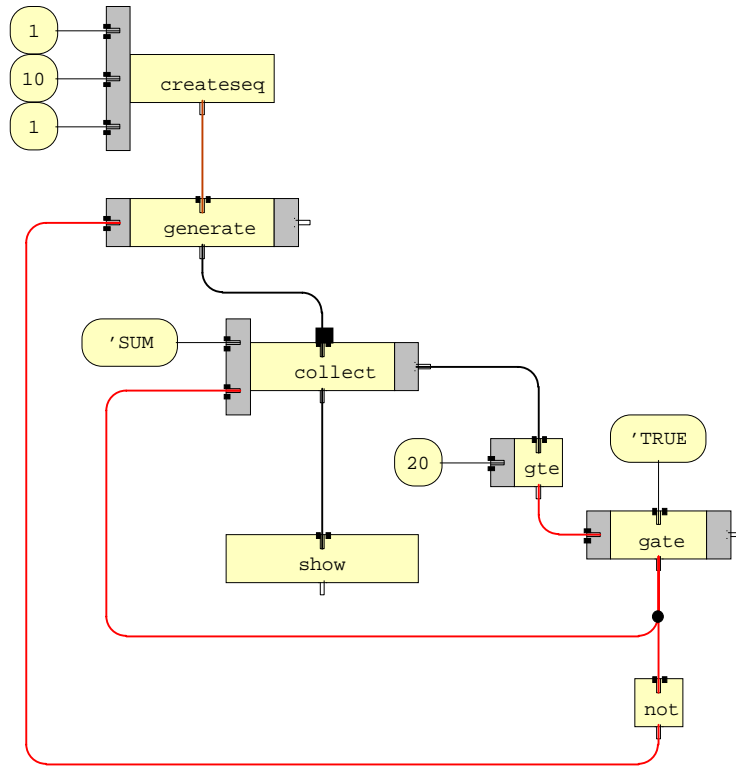


Figure 4. A short-circuited collector

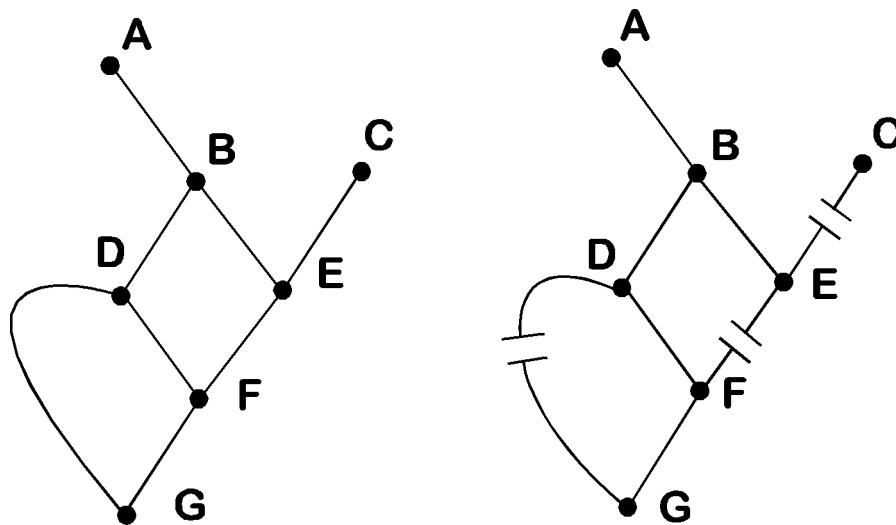


Figure 5. Linearization of a graph

The MidiTrans linear form

Each operation is represented by its label with its inputs before the label and its outputs after it. Inputs are ordered with the primary input first and the secondaries, if any, taken in counter-clockwise order from the visual graph. Outputs are ordered with the primary last, and the secondaries before it, taken in counter-clockwise order from the visual graph. If a primary output feeds directly into a primary input of another operation, then the vertical bar is used (the pipe symbol is appropriate for dataflow). Constant inputs are written as themselves. All other inputs are written as *markers*, with an asterisk followed by a unique positive integer. Multiple outputs and split outputs, where an operation's output goes to more than one input are handled specially. As an example, Figure 6 shows two arithmetic operations feeding a third. The end of a linear flow is indicated with a *cap*,].

Multiple outputs

Outputs of an operation that possibly has many are bracketed together with parentheses, (). This allows for linear forms to be contained in a linear form. Split is an operation that has two outputs and one input. An example of how to linearize it is in Figure 7. Formatting has been added for clarity, but is not part of the syntax.

Split outputs

In a similar fashion, split outputs are bracketed together with diamond brackets, <>. Any output from any operation may be split and fed into many inputs. (On the other hand, multiple outputs cannot feed into a single input - this violates the dataflow model). Figure 8 shows an example.

A Complete Program

Figure 9 shows a complete program with two functions and the corresponding linear form is in Figure 10. Although it is a meaningless program it does contain at least one example of every aspect of the linearization.

```
define (2) sub1 (1) {
  *1 1 add *1003 ]
  *2 2 mult *1004:2 <
    *1003 equal *1006 ]
    *1006 gate *3 ]
  >
}

define (0) eg1 (0) {
  1 5 1 createseq | 'true generate (
    *1000 ]
    *1001:2 <
      "next" print ]
    >
  )
  1 10 createseq | *1001 sub1 | 'SUM *1000 collect | "sum is" print ]
}
```

Figure 10. The linear form for Figure 9.

Appendix A gives a complete BNF for the linear form.

The Programming Environment

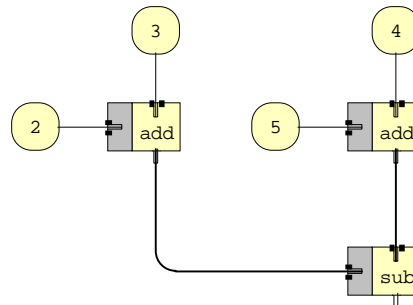
Modular structure

The current implementation of MidiTrans consists of three main parts:

1. A file manager for load, save, print and display functions.
2. An interpreter for programs that also handles playing of MIDI files and/or MIDI fragments.
3. A graphical editor for programs.

Look and feel

Access to these modules is through a conventional menu/submenu system, although keyboard shortcuts (accelerators) are provided for many functions. The multiple document interface system of Windows can show many views of either program files, or MIDI files simultaneously. Menus are sensitive to the type of the active window, so that, for instance, selecting Save for a MIDI file will convert the internal format to a standard MIDI byte stream, whereas selecting Save for a program file will use the linear form generator to save an ASCII version of the program. All the usual operations, such as moving, resizing, and closing on windows are available through standard mouse and/or keyboard operations. The only real addition to the usual operations comes in editing a graphical display of a program function. This needs further explanation.

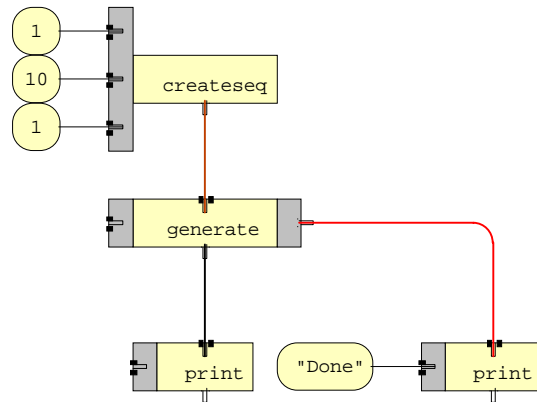


```
3 2 add *1001 ]  
4 5 add | *1001 sub ]
```

Figure 6. A simple datflow graph

Graphical editing of MidiTrans programs

When a program is loaded, only one function, the “main” function, is displayed. Others may be displayed as needed by selecting its name from the Display submenu. In an effort to keep the size of a function’s graph in bounds, a dashed-line rectangle shows the limits of the printed page for the currently selected printer. Although it is not necessary to keep the graph within these bounds, it makes good sense on two grounds. First, anything outside the boundary will not print; in other words, there is no scaling of the graph when printing, in order to preserve readability. Second, graphs that grow too much are unreadable, being a mass of connections between tiny boxes. It is better to keep graphs small and to use the functional abstraction mechanism with meaningful names to make things easy for the reader and the developer.

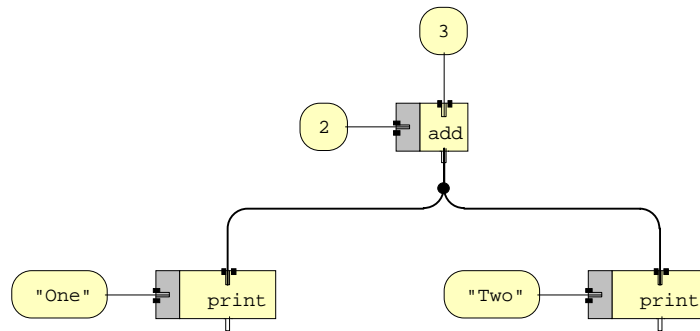


```

1 10 1 createseq | generate (
| print ]
| □ "Done" print ]
)

```

Figure 7. The linear form for multiple outputs



```

3 2 add *1001:2 <
"One" print ]
"Two" print ]
>

```

Figure 8. The linear form for split outputs

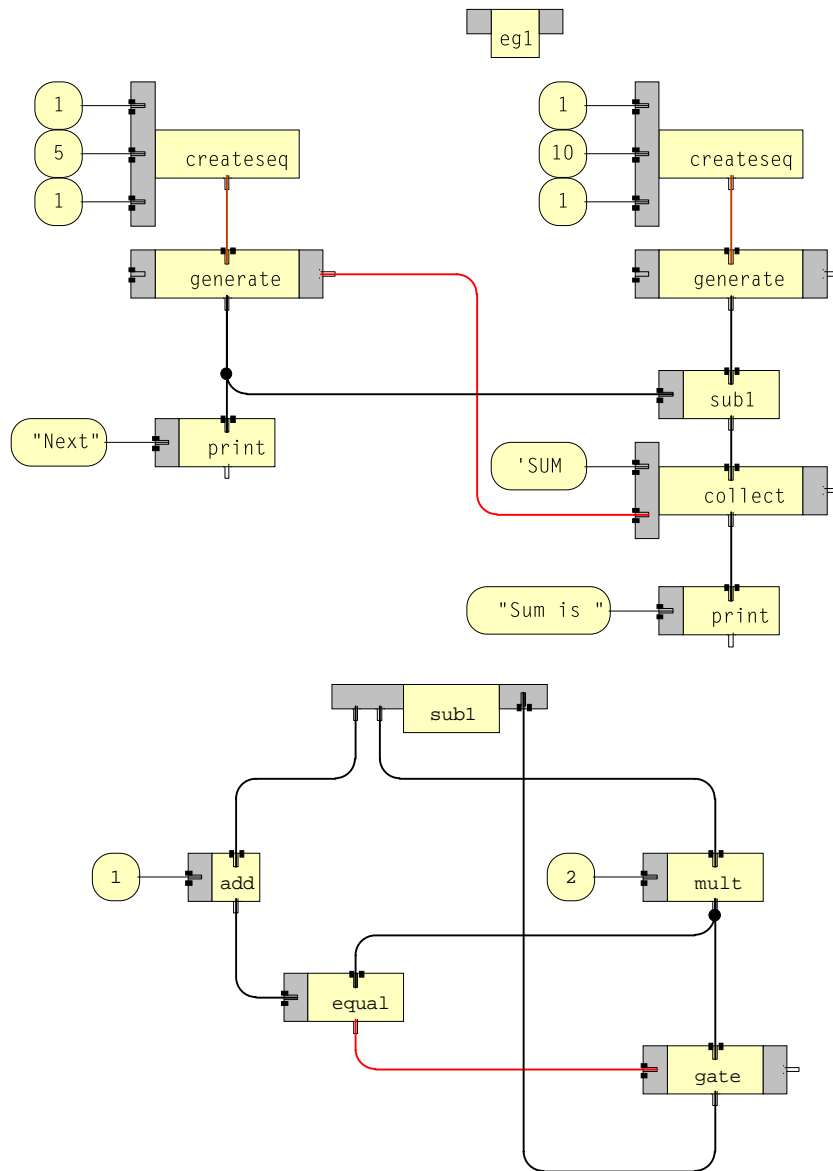


Figure 9. A complete program

Comparison with MAX

The version of MAX described here is Opcode's commercial version, which they describe as an object-oriented MIDI programming toolkit. It is heavily graphical in its look and feel, being somewhat similar to a drawing program with pre-defined visual elements that can be placed on the screen and connected in a variety of ways. Many of its elements can be interacted with since they have user-interface aspects. Some operate in real-time as well.

The basic programming element is an *object* and these are contained in a *patch*, which visually is drawn in a window as a set of interconnected objects. Objects send messages to each other along the connections, sometimes called *wires*, giving MAX a dataflow-like programming model. Objects can be patches themselves, giving MAX a useful level of procedural abstraction. The content of a message can be a value, such as an integer, a floating-point number, or a symbol, and lists of these. A message can also be a control value called *bang*. MAX objects are not pure dataflow processors, however. An object will trigger only when its first or left-most, input contains a message. The other inputs only change the state of the object. This is very similar to the latched input in MidiTrans, except that a MAX object may trigger without all inputs

having messages in them, depending on the object's default values. Figure 11a shows a simple MAX patch which multiplies an input number by 11. Figure 11b does nothing because its left-most input has no message. 11c shows how an input can override the object's default.

There are several other areas where MAX does not display dataflow behavior, notably in its use of global 'variables' through the *send* and *receive* objects. A message sent by an object may be sent to all receive objects with the same name-- a kind of wireless transmission.

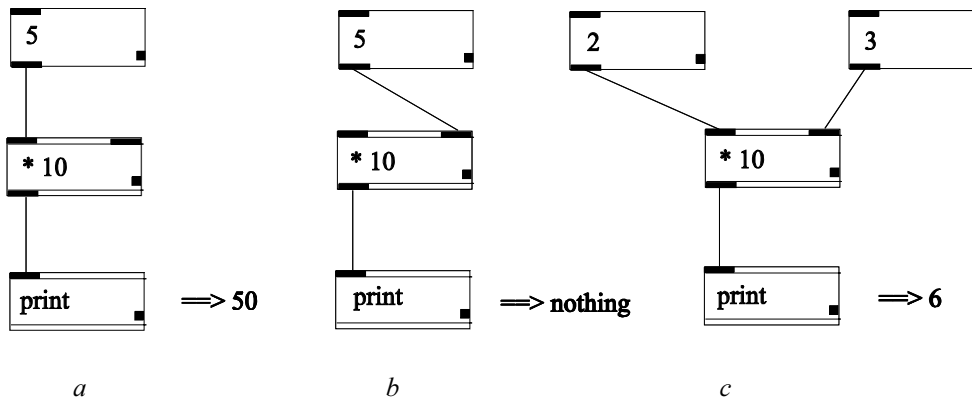


Figure 11. Dataflow in MAX

Comparison with CAL

As an example of how MidiTrans can produce the same results as a CAL program, consider the following:

```

;prolog
(do
  (int GT 0)      ;Gate Threshold
  (int LT 60)    ;Low Threshold
  (int HT 100)   ;High Threshold
  (int PCT 50)   ;Compression Percentage
  (int DV 0)     ;Delta Velocity
  (int Add)      ;Amount to Add
  (int Sub)      ;Amount to Subtract
  (getInt GT "Gate Velocity:" 0 127)
  (getInt LT "Low Threshold:" 0 127)
  (getInt HT "High Threshold:" 0 127)
  (getInt PCT "Compression(%):" 0 100)
)
; body
(if (== Event.Kind NOTE)
  (do
    (if (< Note.Vel GT)
      (delete)
      (if (< Note.Vel LT)
        (do
          (= DV (- LT Note.Vel))
          (= Add (* DV PCT))
          (= Add (/ Add 100))
          (= Note.Vel (+ Note.Vel Add))
        )
      )
    )
  )
)

```

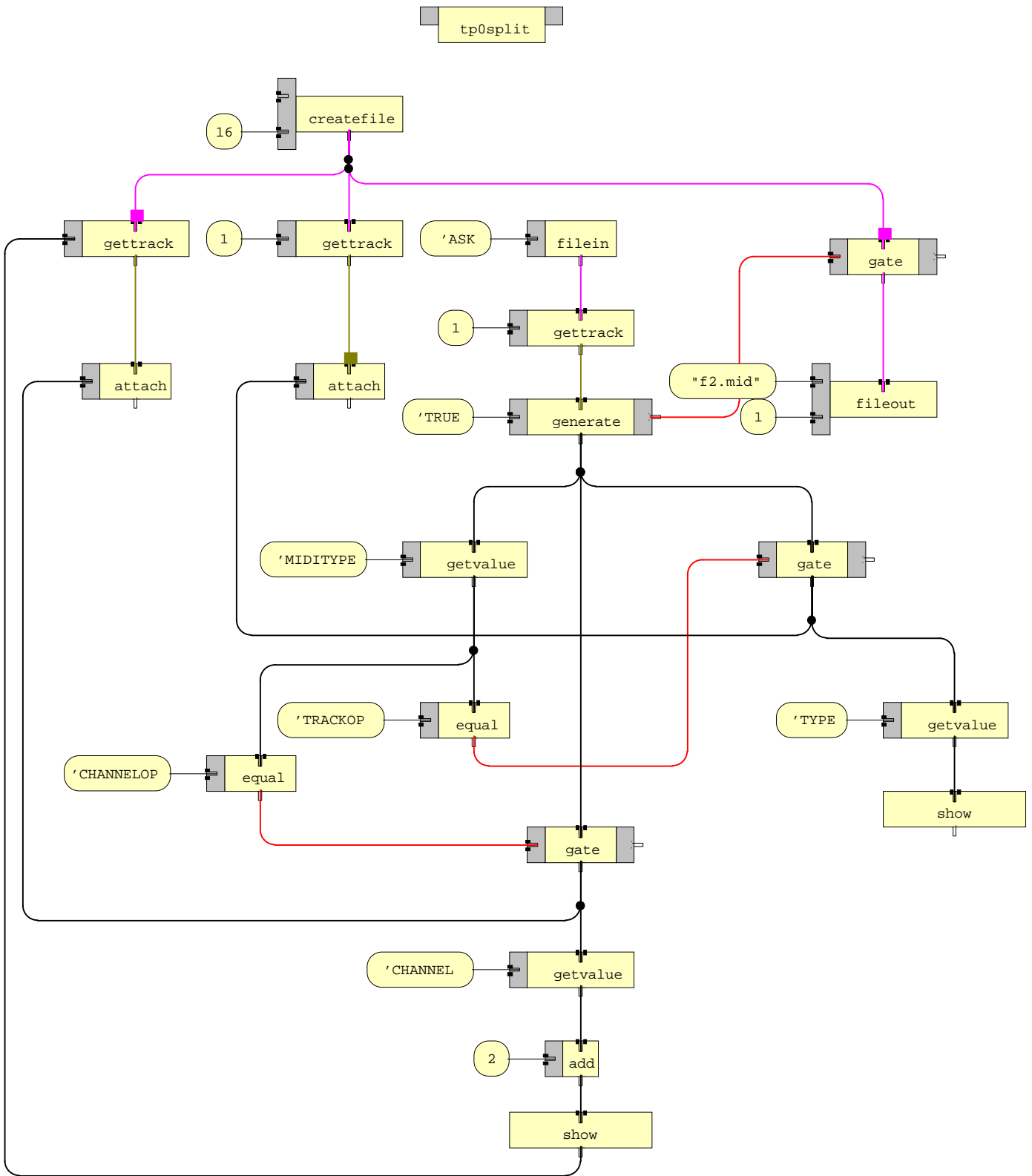


Figure 12. A MidiTrans Program that splits a type 0 file into individual tracks - one per channel

```

)
  (if (> Note.Vel HT)
    (do
      (= DV (- Note.Vel HT))
      (= Sub (* DV PCT))
      (= SUB (/ Sub 100))
      (= Note.Vel (- Note.Vel Sub))
    )
    NIL
  )
)
)
)
NIL
)
; epilog
NIL

```

CAL has a LISP-like syntax, but is essentially a simple procedural language with variables, assignment, loops and conditional forms. The prolog is run before the body, and is used for declaration and initialization of variables. The body is effectively run for every event encountered in the input stream. These events are any of the standard MIDI events, but there is no notion of tracks; only the contents of one track can be presented to a CAL program.

The MidiTrans equivalent is shown in Figure X. The function `gennotes` is not shown but is a generator function that only produces note events when fired, thus making it like the hidden portion of a CAL program, plus the filter conditional given in the first line of the CAL program's body.

Example: Splitting a Type 0 File

In order to illustrate the expressiveness of MidiTrans, we shall present a program of an operation that only some sequencers can do. A type 0 standard MIDI file is completely contained in one track. This makes it very difficult for a sequencer to edit because the single track contains MIDI messages for many channels, whereas a track in a type 1 file usually only contains messages for a single channel. Since a channel is usually reserved for a single instrument (or patch, or program,) it would be good to be able to split a type 0 file into as many tracks as necessary to separate all the channels used.

The algorithm to be implemented by MidiTrans iterates through the track, placing each MIDI event, of whatever type, in a separate track corresponding to its channel number plus 2 (track 1 is reserved for tempo and time signature changes in a type 1 file). The program is shown in Figure 11.

The dataflow graph of the sample program has two roots, a useful variation in MidiTrans graphs. In the example, both `createfile` and `filein` are roots. They are executed in left to right order as displayed in the window, as are the operations connected to a single output with 'blobs'. An example of this multiple output is `createfile`, which is connected to the two `gettrack` operations to the left of the graph, and the `gate` operation at the extreme right. The order of execution is thus:

```

createfile
gettrack (rightmost)
filein
generate (a track)
generate (an event)
getvalue (MIDITYPE)
equal (CHANNELOP) => false

```

gate (pass the event through the secondary)

equal (TRACKOP) => true

gate (pass the event)

attach (to track 1)

...

generate (an event)

getvalue (MIDITYPE)

equal (CHANNELOP) => true

gate (pass the event through the primary)

getvalue (CHANNEL) => 0

add (2)

gettrack (2)

attach (to track 2)

...

generate (an event) => finishes

generate (a track) => finishes

gate (pass the new file)

fileout (save the new file)

Along the way, many operations are rejected because their inputs are not complete. Eventually they will be executed when their inputs are completed and they become descendants of fired operations. The generate operations are special because their finishing depends on the size of the input structure, not simply on the presence of an input. They only finish when their input is exhausted.

There could be empty tracks at the end of the program, depending on whether the input file has MIDI events in a particular channel or not. Note that all non-channel events, such as time signature, and tempo are placed in track 1, as is customary with type 1 MIDI files.

Conclusion

MidiTrans is a general-purpose utility following a dataflow graph metaphor in a visual language framework. Its operation set consists of special purpose operations for handling MIDI events from any standard MIDI file, and also general purpose operations for creation and modification of standard MIDI files. This general purpose approach makes possible many modifications, or transforms, that are difficult or impossible to carry out with standard commercial software sequencers.

References

[1] Twelve Tone Systems. Cakewalk Reference Manual. Twelve Tone Systems, 1994.

[2] The MIDI 1.0 Specification. Available from the International MIDI Association, 5316 w. 57th St., Los Angeles, CA 90056. USA.

[3] The Standard MIDI File Specification. Available from IMA (see [2]).

[4] Adobe Systems Inc. Postscript Language Reference Manual, 2nd. Edition. Addison Wesley, 1990.

[5] Sterling, L.S. The Practice of Prolog. Cambridge: MIT Press, 1990.

[6] Eckel, B. C++ Inside and Out. New York: McGraw Hill, 1993.

Appendix A; The linear form grammar

The grammar to be used is a conventional BNF with non-terminal symbols written in upper case only, and terminal symbols written in lower case. Repetition is indicated by enclosing a sequence in braces, with superscript of 0 or 1 for any number, or at least one repetitions, respectively.

The non-terminal symbols are:

1. MT, the class of MT programs
2. FD, the class of function definitions
3. HEADER, the class of function headers
4. FB, the class of function bodies
5. OPLINE, the class of operation pipelines
6. OPNAME, the class of operator names
7. OP, the class of single operations
8. SPLIT, the class of split outputs
9. MULTI, the class of multiple outputs
10. INSEQ, the class of input parameter sequences
11. OUT, the class of output parameter sequences
12. SPAR, the class of parameters
13. DSEQ, the class of labels
14. INCLUDE, the class of includes

The terminal symbols are:

1. pipe ('|'), the data-flow symbol
2. split ('<'), the start of a split output
3. star ('*'), the parameter prefix
4. cap (']'), the termination symbol
5. join ('>') the end of a split output
6. dquotes (""), the string delimiter
7. quote (""), the symbol quoter.

8. multibegin ('('), the start of multiple outputs
9. lparen ('(', left parenthesis
10. multiend (')'), the end of multiple outputs
11. rparen (')'), right parenthesis
12. begin ('{'), the begin body symbol
13. end ('}'), the end body symbol
14. colon (':'), the split parameter designator
15. define ('define'), the definition keyword
16. include ('include'), the include keyword
17. latch ('!'), the input latch symbol
18. eof, the end-of-file symbol

The BNF for programs is:

```
MT ::= FDSEQ eof           // a sequence of function definitions, followed by end-of-file
```

A sequence of function definitions is:

```
FDSEQ ::= {FD | INCLUDE}0 // any number of function definitions, or includes
```

Each function definition is:

```
FD ::= HEADER begin FB end
                                     // a header followed by a function body in braces
```

An include is:

```
INCLUDE ::= include STRING // STRING is a filename, e.g. myfile.mt
```

A function body is:

```
FB ::= {OPLINE}0
                                     // a sequence, possibly empty, of operation pipelines
```

The definition of the function header is:

```
HEADER ::= define lparen DSEQ rparen OPNAME lparen DSEQ rparen
                                     // the keyword define followed by a digit sequence in parentheses, an
                                     // operation name and another digit sequence in parentheses
```

A pipeline of operations is:

```
OPLINE ::= OP { | OP }0 ( cap | SPAR cap | SPLIT | MULTI )
                                     // an operation followed any number of operations preceded by '|',
                                     // terminated with either ']', a parameter and ']', multiple outputs, or a
                                     // split output
```

A split output is:

```
SPLIT ::= SPAR colon DSEQ (cap | SPLITSEQ)
                                     // a split parameter specification, where the digits after the colon give
```

//the degree of splitting, followed by a split sequence, or terminated
// with '['

The split outputs are given by:

```
SPLITSEQ ::= split {OPLINE}1 join  
// '<' followed by at least one operation pipeline and '>'
```

Multiple outputs are:

```
MULTI ::= multibegin {OUT}2 multiend  
// '(', followed by at least two outputs and ')'  
OUT ::= SPAR cap | [pipe [latch] ] OPLINE | SPLIT  
// either a parameter and ']', or an operation pipeline,  
// optionally preceded by '[' (possibly latched), or a split output
```

An operation is:

```
OP ::= INSEQ OPNAME // an input sequence followed by a name
```

An input is:

```
INSEQ ::= [latch] { SPAR | NUM | SYMBOL | STRING }0  
// a possible latch, then any number of parameters, numbers,  
// symbols or strings
```

The rest of the rules give the token syntax:

```
OPNAME ::= ALPHASEQ // a name is a sequence of alphabetic characters  
SPAR ::= star DSEQ // a parameter is '*' followed by a digit sequence  
NUM ::= ( + | - ) DSEQ // a number is '+' or '-' followed by a digit sequence  
SYMBOL := quote ALPHASEQ // a symbol is '"' and an alphabetic sequence  
STRING ::= dquotes CHARSEQ dquotes // a string is """, any character sequence and """  
DSEQ ::= {DIGIT}1  
CHARSEQ ::= {CHAR}1  
ALPHASEQ ::= {ALPHA}1  
CHAR ::= ASCII character set  
ALPHA ::= a - z | A - Z  
DIGIT ::= 0 - 9
```