Compiler Tools in APL *

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
+1 416 203-0854
bernecky@acm.org

Gert Osterburg
Technoma GmbH
Kurhessenstr. 14
Frankfurt/M
Germany
email: (IPSHARP) technoma@ipsaint

Keywords

Compilers, parsers, finite state automata, FSM, performance.

Abstract

We present the design and implementation of APL Intrinsic Functions for a Finite State Machine (also known as a Finite State Automaton) which recognizes regular languages, and a Parser which recognizes a subset of context free languages, including SLR(1), LALR(1), and LR(1). These are currently being used on a commercial APL mainframe system as part of a large real-time financial trading system, where they are part of a compiler which translates dealer specification statements into APL functions. Use of these Intrinsic Functions more than doubled the performance of the compiler.

In addition to making a significant performance improvement for a large production system, we show these functions to have value in effectively solving many common programming problems, especially those which are inherently or apparently iterative.

1 Introduction

APL is a powerful interactive language which offers primitives to assist in creation of new programs. Among these are *execute* (⍎), which executes APL sentences, and *fix* (⎕fx), which creates APL functions from character tables. Given such primitives, why would an APL programmer want compiler tools?

Several reasons for wanting additional tools are:

- Many applications allow users to specify information in the form of statements expressed not in APL syntax, but in a form closer to a natural language, or in a form closer to unnatural languages such as the more common programming languages. Those statements must be analyzed for syntactic correctness and semantic content according to their rules, which differ from the rules of APL. Such analysis typically requires iteration, an area in which APL interpreters have traditionally performed poorly.

*This paper originally appeared in the APL92 Conference Proceedings. [BO92]

- Many problems which are inherently iterative, or which are not obviously non-iterative, are amenable to easy and fast solutions using compiler tool assists.
- Manually constructed syntax analyzers and parsers are less likely to be fault-free than those constructed with the aid of already debugged compiler tools. Hence, code reliability is improved by the use of extant tools.

Although a complete calculus of compiler tools is beyond the scope of this paper, we have designed and implemented a subset of such tools, which have done an excellent job in practice on a number of applications.

2 A Calculus of Compiler Tools

Syntax-directed program development is an important area of computer science: Considerable research has resulted in the availability of a number of compiler-writing aids such as LEX and YACC [Joh86, LS86]. These tools have turned compiler writing from an art into a science, making creation of simple compilers an undergraduate term project.

The problems faced by APL application programmers are no different from those faced by programmers working in other languages. Having efficient compiler tools available in APL expands the power of APL in quite unexpected ways, as will be shown below.

What sort of tools are required for a calculus of compiler tools? Consider the following set of functions which deal with syntax-directed program development:

- `FSMgenerator` A Finite State Machine generator which takes definitions expressed as regular expressions, and generates an FSM table.
- `PTgenerator` A Parser Table generator, which takes definitions of a grammar and grammar type, and generates a Parser Table.
- `FSM` A Finite State Machine (or Finite State Automaton) which takes an FSM table and source text, and produces a list of machine states and partitions, which can be used to extract Tokens from the text, and to perform certain classes of validation on the text.
- `PARSER` A Parsing function, which takes a Parser Table and Tokens, and produces a Parse Tree.
- `Interpreter` An interpreter or compiler function, which takes APL code fragments and a Parse Tree, and produces APL programs, or directly interprets the desired expressions.

These tools provide a compiler or interpreter generating device which is comparable to Lex and Yacc:

- `FSMgenerator` and `FSM` could be useful in regular pattern matching. SHARP APL's LOGOS, [AGDH86] for example, uses the terminology, but has implemented only a very simple subset of regular expressions. With these functions, full regular expressions could be expressed quite efficiently.
- `FSMgenerator` and `PTgenerator` would make up a parser generator. That is, they generate Tables which, together with the `FSM` and `PARSER`, constitute a proper parser.

Detailed discussion of the entire set of compiler tools is beyond the scope of this paper, so we restrict discussion to the `FSMgenerator`, `PTgenerator`, and the `FSM`. A later section deals with the definitions of these tools.

Token	Example	Regular Expression
nat	123	DIG *
integer	123 -123	(!+ !-) nat
real	1.3 -1.3	(!+ !-) ((nat *) !. nat + nat !. (nat *))
expreal	1.3e3	real !e integer
names	abc a123	ALPH (ALPH + DIG) *
currency	USD DM	(!U !S !D) + (!D !M)

Figure 1: Regular expressions for data validation

Regular Expression	Pattern recognized (p,q are instances of patterns)
!x	the character x
!	the empty string
p q	p followed by q
p + q	p or q
p *	zero or more p

Figure 2: Regular expression definition

3 Regular Expressions

Regular expressions are commonly used in compiler design as a way to compactly describe the legal constructs of a grammar. If we wish to construct a grammar which accepts numbers, characters, and currencies, we might construct regular expressions to define such a grammar. These allow precise definitions in a general way. See Figure 1 for an example of regular expressions for defining tokens.

In fact, any pattern which can be derived from the recursive definition in Figure 2 can be recognized by a Finite State Machine.

As noted earlier, the reliability of grammars based on regular expressions and compiled by a generator are more likely to be correct than those constructed manually in an ad hoc manner.

Regular expressions are thus seen to be valuable both in scanning for tokens in a traditional programming language, or in the validation of statements in a custom-designed end-user oriented command language.

4 Compiler Tool Definitions

4.1 The FSM Generator Definition

The Finite State Machine (also known as a Finite State Automaton (FSA) or FSM) generator function is written in ISO standard APL, except for the use of `⎕signal` to signal errors during argument validation. Performance of the generator itself is usually not critical, because a generator is typically used only once, while its results are used many times.

The duty of `FSMgenerator` is to take grammar definitions expressed as regular expressions, and produce a table which can be used as the left argument to the `FSM` Intrinsic Function. In working with typical grammars, the effort involved to create a finite state machine definition by hand can be excessive. The `FSMgenerator` automates this process.

Proving that a finite state machine definition is correct is also difficult. The use of a function to create these definitions ensures that they are correct if the grammar is correct.

4.2 The PTgenerator Definition

The PTgenerator function takes as argument a set of grammar rules as shown in the second part of Figure 7. Rules are given by:

Nonterminal $\rightarrow$
List-of-Terminals-and-Nonterminals

Terminals are identified by a leading '$' and are assumed to be defined via regular expressions. After performing some plausibility checks such as:

- does each Nonterminal occuring right of '→' also occur left of at least one rule?

- does each Nonterminal on the left occur also on the right side (except for the first one, which is considered to be the starting Nonterminal).

The parser starts constructing the parsing tables according to the type (slr in Figure 7) of the grammar. For algorithmic details we refer to Aho, et al [ASU86].

The result of PTgenerator is either the GOTO and Action tables (as shown in Figure 8) or an error message indicating the reason why the construction of the parsing table failed.

In the SWAP application discussed later, an SLR(1) grammar is used. A LALR(1) generator also exists, but it runs fairly slowly. LALR(1) is also the technique used by Yacc.

4.3 The FSM Definition

A Finite State Machine derives its name from the fact that its state after examining a new argument item is dependent upon the current state of the machine and the value of the new item, based on elements indexed from a state table, denoted here as the FSM table.

Figure 11 is an APL model of the FSM primitive as implemented in SHARP APL, a non-stacking finite state automaton.

The `FSM` Intrinsic Function takes two arguments – an FSM table and a list of text. Its result is an integer list of the same shape as the text argument, whose elements are the state the FSM entered after scanning each character in the text. Negative elements indicate the ends of partitions.

The FSM table is really two arguments rolled into one: The first row is the one-origin `⎕av` indices of characters which will be used to select a column from the remaining rows of the table. Characters which are not explicitly indicated in this row are assumed to appear in the last column of the row. The remaining rows of the FSM table are next-state values, selected by the current state of the FSM (initially 1), and the column indicated by the next character in the argument. The rows are one-origin indices into the rows of the FSM table excluding the first row (or, zero-origin indices into the rows including the first row).

Legal values in these rows are `¯1` or legal one-origin indices into the FSM table after the first row has been dropped. Zeros and values greater than `1↑⍴FSMtable` cause domain error. Other positive values indicate the row of the `FSMtable` to be used as the next FSM state.

Negative one indicates an illegal transition. This causes the previously generated result element to be negated, marking the end of a partition of legal characters. The FSM resets to the initial state, and rescans the offending character from that state. A failure in this state causes negative one to be stored in the current result element, and the machine restarts on the next character, starting a new partition.

It was the choice of using negative result elements to mark partition ends which led to the choice of origin one in the FSM definition. Otherwise, origin zero would have been used, in keeping with the philosophy of newer APL dialects. Other definitions for the FSM would make things a bit simpler in this area. For example, one might return a two-element boxed list of lists in which the first is a partitioning, and the second is merely the FSM states.

4.4 Benefits of the FSM

A Finite State Machine primitive offers several benefits to the APL programmer: It is often easier to create an FSM to solve a problem, than to understand the series of APL primitives required to solve the same problem efficiently and without resource

problems, such as `WS FULL`. The Unix tab expansion problem noted below is a typical example of this sort.

FSM arguments can be generated automatically, based on regular expression definitions. These are more reliable than hand-written programs, a bonus for those who care about correct results as well as good performance.

The excellent performance of FSMs is due to the fact that the scanning process is a linear algorithm in terms of the length of the input, and is more likely more efficient than hand-written programs.

4.5 The PARSER Definition

`PARSER` is a model for a Finite State (stack) Machine. PARSER, as shown in Figure 12, recognizes a subset of context free languages among which are SLR(1), LALR(1) and LR(1). The speciality of the subset consists of the bottom up technique and the fact that the parser only looks ahead 1 token. The generality comes from the device generating the parsing tables.

Although automatic generation of code generators may result in poor performance, our experience in the applications arena shows the contrary: "handmade" systems are quite often poor compared to those generated by compiler tools.

4.6 Inclusion of Semantic Functions

It is desirable to be able to include functions which implement the semantics of the language into the parser. One way to associate semantics is an algebraic one which fits nicely into APL. Consider the rules of the grammar definition in Figure 3, which is a trivial language for adding up ones.

Associating a function with each rule, as in Figure 4 allows one to translate the parse tree into an expression over this set of functions.

```
Expr =: Expr + Expr
Expr =: ( Expr )
Expr =: 1
```

Figure 3: A simple grammar for adding ones.

```
F1 : Expr Expr =: Expr    Add 2 numbers
F2 : Expr =: Expr         Identity
F3 : nil =: Expr          Constant 1
```

Figure 4: Associating functions with grammar rules.

5 Case Studies

The following case studies offer several examples of the use of `FSM` and `PARSER` in actual production applications as well as concocted situations.

5.1 Unix Tab Expansion

UNIX[1] character strings often contain embedded tab characters. These are understood to represent typewriter-like tabs, in which tabstops are set in the first column, and every eight columns thereafter. Expansion of these character strings, replacing tabs by an appropriate number of spaces, is a typical example of APL in which the obvious solution involves an iteration for each tab character or newline character. This is computationally expensive, and ugly enough that we will not show it here.

An advanced APL programmer will eventually come up with an algorithm based on partitioning, such as that in Figure 5, by recognizing that a tab or newline forces the eight residue of the cursor position to zero, and that other characters increase the residue by one. Such an algorithm outperforms the naive iterative algorithm by a factor of 25 or more.

However, time constraints and programmer skills often limit our ability to see efficient non-iterative

[1] UNIX is a trademark of AT&T.

approaches, and performance of the application suffers as a result of this.

The use of an FSM to perform the tabbing, as shown in Figure 6, produces a function which outperforms the best partitioned APL solution by a factor of two. Furthermore, it may be easier to comprehend by a relatively inexperienced programmer.

Here is how the FSM-based tabber works. The purpose of a tabber is to replace tab characters by an appropriate number of blanks. The number of blanks depends on the eight residue of the column number in which the tab occurs. Assume that the first column is column 0. A tab there will take us to column 8, implying 8 blanks. A tab in column 6, however, will only produce 2 blanks. The 8 residue of the column number of the first tab is merely the number of characters preceding the tab. Therefore, we know how to process the first tab.

But what about later tabs? That's easy, it turns out, because the eight residue at any tab character will always be zero, so we can start counting from zero again. This means that a partitioning of the array on tabs, and counting characters in each partition, gets us close to the solution.

But what about newline? That's easy as well, because the only effect of newline is to set us counting from zero again.

Recognition of these facts leads us to the partitioned tabber function in Figure 5. Now, let's do it using the FSM. Here is the FSM matrix used by the code in Figure 6:

```
    ∇ r←tabber2 ⍵;i;pv;pv2
[1] ⍝ UNIX Partitioning Tabber
[2] pv←⍵=tab
[3] pv2←⍵=cr
[4] i←,8-1+8|(pv∨pv2) ¯2⌽⍤⍴pv
[5] r←1+(~pv2)×(pv2∨pv)\i
[6] ⍵[pv/⍳⍴pv]←' '
[7] r←r/⍵
    ∇
```

Figure 5: Unix tabber using partitioning.

```
    ∇ r←tabber ⍵;i;rm
[1] ⍝ UNIX FSM Tabber
[1] r←tabfsm fsm ⍵
[2] r←(8⍴1), 8 7 6 5 4 3 2 1[|r]
[3] i←⍵∈tab
[4] ⍵[i/⍳⍴i]←' '
[5] r←r/⍵
    ∇
```

Figure 6: Unix tabber using FSM.

```
 3  157  1
 9    1  2
10    1  3
11    1  4
12    1  5
13    1  6
14    1  7
15    1  8
 1    1  1
 9    1  2
 9    1  2
 9    1  2
 9    1  2
 9    1  2
 9    1  2
 9    1  2
 9    1  2
```

How do we interpret this? First, we recognize that there are three classes of characters of interest. These are newline (`cr`), tab (`tab`), and all others. `tab` and `cr` have the property that they force us to a column with an eight residue of zero. Other characters merely increment the column number (This simple example precludes backspace, linefeed, and other control characters in the input) modulo eight. The `⎕av` index of `tab` is 3, and that of `cr` is 157, in origin one. These appear in the first row of the FSM table, indicating which column we will select a new state from when the corresponding character is the next one in the argument. The third column contains a 1, merely as a placeholder, which will collect all the non-interesting characters (C).

Note that, because there are no `¯1`'s in the FSM table, that the result will consist of a single partition only.

The FSM starts off in row 1 of the FSM table (the row containing `9 1 2`). If we see a non-interesting character C, the FSM picks the 2 as its new state. Another C takes it to state 3, and so on, until it is in state 8. Another C at that time takes it back to state 1. Therefore, the FSM implements a modulo-8 counter for C.

If the FSM encounters a `cr`, it always goes to state 1, resetting the modulo-8 counter.

If the FSM encounters a `tab`, it goes to a state from 9 to 15, or state 1, depending on the state of the FSM at the time.

The interesting part of all this is that the result of the FSM is an integer list giving the state of the FSM after processing the corresponding character of the right argument. These states are then used by the `tabber` function to index an integer list which is then used with replicate to expand the tabs to an appropriate number of blanks, after the tabs have been replaced by blanks.

If we didn't need the varying states of the result for the index operation, the trailing rows of the FSM table, which are otherwise identical, could be merged into a single row.

Here is the FSM output of states for a simple text list, in which tabs are represented by `∘`, followed by the result of indexing to obtain the replicate argument:

```
      t←'Tx∘is∘∘here',nl','∘too'
      ⎕←f←tabfsm fsm t
2 3 11 2 3 11 9 2 3 4 5 1 9 2 3 ¯4
      ⎕←e←((8⍴1),⍳8)[⎕f]
1 1 6 1 1 6 8 1 1 1 1 1 8 1 1 1
      ⎕←e/t
Tx∘∘∘∘∘∘is∘∘∘∘∘∘∘∘∘∘∘∘∘here
∘∘∘∘∘∘∘∘too
```

5.2 Input Validation

One design problem with primitives in APL and other languages [Ber91] is that they often do *more* than we want them to do. The `⎕vi` verb in SHARP APL was designed to assist in the validation of numeric lists expressed as characters. The verb returned a Boolean list marking valid and invalid nu-

meric entries in its argument. However, for many applications, the generality of ⎕vi is excessive – it permits the inclusion of floating point numbers expressed in exponential format, such as `123.45e6`, as well as complex numbers such as `0j1`. Such numbers, although perfectly valid in APL arrays, are often invalid in applications where they might represent part numbers or salaries. Of course, some of us have imaginary salaries, but that's not the point. Specific code is needed to validate such input, and ⎕vi is inadequate to the task. What can be done?

Consider the following concocted specification of input validation: We wish to validate a character list of purported numbers. The numbers are allowed to be integers or decimal numbers, but complex numbers are forbidden, as are numbers expressed in exponential format. Leading plus (+) or minus (−) signs are permitted, but the APL high minus (¯) is forbidden.

This can be done effectively in APL with use of the `FSM` Intrinsic Function. The `FSMgenerator` could be used to generate the FSM table, but doing it by hand will help to clarify how the FSM actually operates.

We start with a list of valid characters with a question mark at the end to indicate the invalid ones:

```
' +-.0123456789?'
```

We use these to start forming the table, then add letters corresponding to states of the FSM. In the following, we have elided the columns corresponding to the characters 3-9, to reduce the space required to display the table.

State			Signal characters							
Name	Id	#	''	+	-	.	0	1	2	?
Initial	i	1	i	s	s	f	d	d	d	e
Digits	d	2	i	e	e	f	d	d	d	e
Sign	s	3	e	e	e	f	d	d	d	e
Fraction	f	4	i	e	e	e	f	f	f	e

This table can be read as follows: If we are in the initial state i, a blank will leave us in the initial state i. Thus, leading blanks are skipped. A sign will take us to state s, indicating a signed digit. A digit will take us to state d, indicating digits. A period will take us to state f, indicating fractional numbers.

If we are in state s, we have already seen a sign, and are looking for the rest of the number. A blank takes us to the error state e, indicating a lone sign with no number following it. Another sign also leads to state e, indicating adjacent signs, such as +−. Further digits take us to state d, to skip over remaining digits.

If we are in state d, we have seen part of a valid number, which may or may not be signed. A blank takes us back to the initial state i, where we restart the scan on the next character. A sign or illegal character takes us to the error state e, indicating a sign in the middle or at the end of a number, or an illegal character.

The reader is left to comprehend state f.

If we replace the state-marking characters with numeric indices, we are left with the following FSM table (again with columns for 3-9 elided):

```
 1  3  3  4  2  2  2 -1
 1 -1 -1  4  2  2  2 -1
-1 -1 -1  4  2  2  2 -1
 1 -1 -1 -1  4  4  4 -1
```

Catenating the ⎕av indices of the signal characters onto the top of this table gives us `fsmv`, the left argument to the FSM. Here are two examples of its use on legal and illegal arguments:

```
      fsmv fsm '45 0 ¯1 +134 .5'
2 2 1 2 1 3 2 1 3 2 2 2 1 4 ¯4
      fsmv fsm '3j5 1e5 45+9'
¯2 ¯1 2 1 ¯2 ¯1 2 1 2 ¯2 3 ¯2
```

The last item in the result is negative, indicating the end of a partition. Valid partitions are positive

(except for the last one), and are separated by `1`s. Catenating a blank onto the end of the argument, and discarding the last result element from `fsm`, simplifies result analysis.

In practice, a validator such as this might examine the result to ensure its validity (`∧/1↓(fsmv fsm ⍵)>0`), replace + and − in ω by `' '` and `'¯'`, respectively, and pass the result to `⎕fi` to do the remaining work. If `⎕fi` is unable to do the job, the `fsm` result is often useful in partitioning ω to do the work in APL in a more efficient manner.

5.3 SWAP Compiler

Reuters AG, of Frankfurt, Germany, operates SWAP, a large APL-based real-time financial application used in securities trading. This application has a very high concurrent user load, and consumes significant mainframe processor time, of which a large portion is spent in analyzing and compiling user requests and conditions, stated in a formal language.

The use of the compilation tools described here, in particular, the `FSM` and `PARSER`, more than halved the CPU time required for the compilations.

5.4 A Simple Calculator

Consider the example of a grammar to describe a simple calculator. Let's look at it, to see how the compiler tools would work to construct it.

The input to the compiler generator, which are token definitions, syntactic definitions, and a set of semantic functions, are shown in Figure 7.

The semantic model associates with each syntactic rule a function which has as many arguments as there are symbols on the right side of the rule and produces a result. Therefore, the calculator grammar leads to the functions which appear in Figure 8. The functions which interpret the semantics are as follows:

```
compile sc (,)

/* 1. Token Definitions */
lex
$lparen = !(
$rparen = !)
$one = !1
$plus = !+
end

/* 2. Syntactical definitions */

/* 3. Semantics */
/* E is Expression */
/* T is a term */
/* R is rule number*/
/* APL-function */
syntax slr
/*R1*/ E → E $plus T { E + T }
/*R2*/ || T { T } ;
/*R3*/ T → $lparen E $rparen { E }
/*R4*/ || $one { ⍎ $one } ;
end
```

Figure 7: Compiler generator input

Fn	Definition	Res	Description
f1	E plus T	E	Add E and T
f2	T	E	Return T
f3	(E)	T	Return E
f4	one	T	Convert to numeric

Figure 8: Calculator grammar functions.

State	GOTO table		Action table				
	E	T	+	1	(	)	end
B 1	3	2	0	5	0	4	0
E 2	0	0	6	0	0	0	0
3	0	0	¯2	0	¯2	0	¯2
4	3	7	0	5	0	4	0
5	0	0	¯4	0	¯4	0	¯4
6	8	0	0	5	0	4	0
7	0	0	6	0	9	0	0
8	0	0	¯1	0	¯1	0	¯1
9	0	0	¯3	0	¯3	0	¯3
B = Begin State E = End State							

Figure 9: Parser table generated for calculator

```
f1:⍵[1]+⍵[3]
f2:⍵
f3:⍵[2]
f4:⍎⍵
```

The Parser Table generator, `PTgenerator`, in turn generates the parsing table of Figure 9.

5.4.1 The Calculator Interpreter

Let's walk through an example of the resulting calculator in action:

```
Input          '1+1+1'
                ↑↑↑↑↑
Tokennumbers    01234
```

In the resulting parse list, positive numbers identify the above tokens, and negative numbers identify rules (functions). The parse list for the above example is:

```
0 ¯4 ¯2 1 2 ¯4 ¯1 3 4 ¯4 ¯1
```

This bottom up parse list corresponds to the following derivation, in which the leftmost term is the rule number being applied. Application of the rule generates the next line in the derivation.

```
R4   1+1+1
R2   T+1+1
R4   E+1+1
R1   E+T+1
R4   E+1
R1   E+T
     E
```

This derivation translates to the following postfix expression (assuming a function `N` which gets the next available input token):

```
((((N f4)f2),N,(N f4))f1,N,(N f4))f1
```

The above translates to an APL expression when read from right to left (using strand notation):

```
f1 (f4 N) N (f1 (f4 N) N (f2 f4 N))
```

Note that these operations can be performed automatically. Also, the generator can derive from the semantic definition of `f2` that it in fact is the identity function and therefore can be omitted in the expression to be executed.

6 An FSM table generator

In order to generate FSM tables based on regular expressions, the lexical and syntactic definitions in Figure 10 are required, as well as the following APL functions:

- STAR This function constructs, based on the argument fsm, a fsm1 which recognizes all fsm-patterns an arbitrary number of times.
- CAT Constructs an fsm which recognizes all possible catenations of pattern from the left and right argument fsms.
- PLUS Constructs an fsm which recognizes either a pattern from the left or right hand argument fsm.
- IDENTITY Simply return the argument.
- CHAR Construct the fsm which recognizes the given character.
- EMPTY Construct a fsm which recognizes the empty list.

The algorithms used to implement these functions can be found in the literature, such as Aho, et al. [ASU86] It is possible to implement such functions rather quickly (using the additional concept of non-deterministic FSM's). In practice, one might use slightly more complicated and efficient algorithms.

7 Problem Areas

There are several potential problems with the Finite State Machine primitive:

- There is no need, per se, for an FSM primitive. It is easily written directly as an iterative APL function. The performance of such a function in traditional interpreters, however, argues for a primitive, AP, or associated function. However, is performance an argument for new primitives, or merely an argument for higher performance APL interpreters and compilers? Stronger arguments for a primitive are those of utility and correctness. We sidestepped this argument by

```
compile fsmgenerate (,)

lex /* Scanner def*/
/* Strings of length 1 */
$char = !ANYCHARACTER
/* empty string */
$empty = !' !'
/* left paren */
$lparen = !(
/* right paren */
$rparen = !)
/* or operation */
$plus = !+
/* Kleene's star function */
$star = !*
end

syntax slr
/* Syntax def – semantic def*/
E → E $star {STAR E}
| E T {E CAT T}
| E $plus T {E PLUS T}
| T {IDENTITY T}
;
T → $char {SIMPLE $CHAR}
| $empty {EMPTY}
| $lparen E $rparen {IDENTITY E}
;
```

Figure 10: Sample FSM table generator

using an Intrinsic Function instead of defining a new primitive.

- The FSM table for the FSM primitive can get excessively large. Consider an FSM machine definition which decodes variable-length data records. These records typically consist of a two-character count of the data which follows, followed by the data itself, and more records of the same type. An FSM to decompose these would require more than 65536 states, and would needlessly examine every character of the argument. A straightforward iterative APL function might run considerably faster, and not require the large amounts of FSM definition storage which the FSM definition requires. CRC calculations are even larger, requiring `2*32` states or more.

  An anonymous reviewer of this paper noted that:

  > Programming paradigms based on FSAs are a current area of research and a literature search should turn up other work on language support for FSAs. Very powerful FSA table compression algorithms have been developed that solve some of the problems cited and may significantly extend the domain of problems addressed by FSAs.

  We endorse such a search, although time pressure has precluded inclusion of specific references in this paper.

- Most APL systems running on Unix platforms offer the ability to call Yacc and Lex, as well as other utilities, from APL. Given this, why is Yet Another Function required? One reason was that our platform was IBM's MVS operating system, not Unix. A second reason is that the primitives do not currently offer all of the generality of the Unix tools, and hence may be considerably more efficient. Since we have not made timings of both sets of tools on a Unix platform, we were not able to test this conjecture. Finally, Yacc and Lex make certain types of syntax analysis difficult or impossible. Raul Rockwell, of Carnegie Mellon University, described problems (private communication) in trying to use them to describe the syntax of J[HIMW90].

8 Implementation Notes

The FSM and PARSER described above, as well as other performance-critical functions, were implemented on SHARP APL/370 as Intrinsic Functions. Intrinsic Functions appear to the user as locked APL functions, but are, in fact, written as assembled or compiled code. The design and implementation of Intrinsic Functions support in SHARP APL was done by Leigh Clayton, of the APL Software Division of Reuters Information Systems (Canada) Limited.

Intrinsic Functions offer a simple way to provide high-performance functions to specific APL users without being forced to deal with the detailed design issues facing those who design APL primitives for use by the entire APL community. They can be tailored to meet the needs of specific applications, rather than being general tools.

9 What Next?

A rather obvious redefinition of the FSM, which is worthwhile considering as a language primitive,

would take a two-element boxed left argument, instead of the current integer argument. The second item would be the integer table formed from all but the first row of the current argument. The first item would be a list of characters, corresponding to the columns of the second item. A system with heterogeneous arrays might merely allow the first row of the argument to be boxed. The use of heterogeneous arrays in this way has some performance impact, and also increases FSM table storage, but these are not necessarily serious obstacles.

This extension, in itself, is of marginal value, but other definitions allow more general capabilities than those in our FSM. Two possible extensions come to mind:

- Instead of having a list of characters as the first item, it would be possible to have a list of lists of characters. A text item would select an FSM table column based on the presence of the item in one of the lists. This would allow a smaller FSM table in common circumstances, such as placing the digits from zero to nine in a single column, not possible in the definition presented here.

- A more interesting extension is possible in J or dialects of APL which permit tacit definition and gerunds, as described by Bernecky and Hui[BH91]. This is to embed selection verbs directly into the list. This would allow more sophisticated capabilities in recognition. For example, it would be easy to extend the definition to numeric right arguments, and pick a column based on the range of the number. Histograms come to mind as a possible application of this extension.

  The extension of arguments to include tacit verbs[HIMW90] represented as gerunds offers significantly increased power in the language, and should be examined in detail.

One problem with these tools is that it is difficult to associate Action Routines with each state change, and to describe the arguments to such routines. Since J offers the capability to create function arrays as gerunds, J may be better equipped to deal with this problem. An added question is whether J can implement the FSM directly, perhaps as a scan.

Another problem is that Lex, Yacc, and our compiler tools are not really adequate for the development of "real" compilers or interpreters, because one can often construct smaller, better performing parsing tables by hand. Since performance is usually a critical issue in compilers, and since programming language syntax and semantics are usually fairly static, the effort required to manually create better parsing tables often pays off.

However, in the area of applications programming, rapid and significant changes in requirements are quite common. In such cases, the benefits of compiler tools are obvious, in terms of speedy development and application reliability.

Perhaps the most significant contribution of our work is defining the FSM to return the list of intermediate states, rather than the final state only. As was shown in the examples, the intermediate states are of great value in control of later computation.

References

[AGDH86] David B. Allen, Leslie H. Goldsmith, Mark R. Dempsey, and Kevin L. Harrell. LOGOS: An APL programming environment. *ACM SIGAPL Quote Quad*, 16(4):314–325, July 1986.

[ASU86] Alfred V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ber91] Robert Bernecky. Fortran 90 arrays. *APL-CAM Journal*, 13(4), October 1991. Originally appeared in *ACM SIGPLAN Notices*, 26(2), February 1991.

[BH91] Robert Bernecky and R.K.W. Hui. Gerunds and representations. *ACM SIGAPL Quote Quad*, 21(4), July 1991.

[BO92] Robert Bernecky and Gert Osterburg. Compiler tools in APL. *ACM SIGAPL Quote Quad*, 23(1):22–33, July 1992.

[HIMW90] Roger K.W. Hui, Kenneth E. Iverson, E.E. McDonnell, and Arthur T. Whitney. APL\? *ACM SIGAPL Quote Quad*, 20(4):192–200, August 1990.

[Joh86] Steve Johnson. *UNIX Programmer's Manual: Supplementary Documents 1*, chapter Yacc: Yet another Compiler Compiler. University of California, Berkeley, CSRG, 1986.

[LS86] Lesk and Schmidt. *UNIX Programmer's Manual: Supplementary Documents 1*, chapter Lex – A Lexical Analyzer Generator. University of California, Berkeley, CSRG, 1986.

```
    ∇ z←α rbefsm ω;⎕io;i;s;n;fs;k
[1] ⍝ apl model of finite state machine intrinsic function
[39] k←⎕io←s←i←1 ⍝ initial state and index.
[40] z←(⍴ω)⍴666 ⍝ initialize result. (values are unimportant).
[41] ⎕signal(0=⎕nc 'α')/domerr
[42] ⎕signal(1≠⍴⍴ω)/rnkerr ⍝ vector right argument
[43] ⎕signal(2≠⍴⍴α)/rnkerr ⍝ matrix left argument.
[44] ⎕signal(1≥1↑⍴α)/lenerr ⍝ need at least two rows in left.
[45] ⎕signal(0∊≠α[1;])/domerr ⍝ no duplicates in row 0.
[46] ⎕signal(~α[1;]∊⍳256)/domerr
[47] ⍝ prevent any index errors in fsm.
[48] ⎕signal(~(,fs← 1 0 ↓α)∊¯1,⍳1↑⍴α)/domerr
[49] ⎕signal(0∊⍴α)/lenerr ⍝ no funnies with empty left arguments.
[50] ⍝
[51] ω←(¯1↑⍴α)⌊⎕av[α[1;]]⍳ω ⍝ map ω to column numbers in left.
[52] fsml:→(i>⍴ω)⍴fsmlz ⍝ done?
[53] z[i]←n←fs[s;ω[i]]
[54] →(¯1=n)⍴fsmnp
[55] s←n
[56] fsmnxt:i←i+1
[57] →fsml
[58] ⍝
[59] ⍝ If s=1, α[i] is illegal as token starter.
[60] ⍝ If s≠1, char may or may not be OK. Rescan from state 1.
[61] fsmnp:→(s=1)⍴fsmg
[62] s←1
[63] →fsml
[64] ⍝
[65] fsmg:s←1 ⍝ reset to initial state, and continue scanning.
[66] →fsmnxt
[67] fsmlz:→(0∊⍴ω)⍴fsmlzz
[68] ⍝ Mark end of last partition.
[69] z[⍴z]←-|z[⍴z]
[70] fsmlzz:
    ∇
```

Figure 11: APL model of the Finite State Machine.

```
    ∇ r←α mparser ω;ac;accept;nt;nr;ls;EM;i;a;s;na;gt
[31] 'EM accept ls nr ac gt' ⍙assign α ⍝ Separate Argument
[32] EM←1,EM ⍝ EM is used as an internal stack afterwards.
[35] nt←⍴dupout ls
[37] accept←accept,EM[2]
[40] r←⍳i←0 ⍝ Init Result (r) and loop Counter (i)
[41] ⍝ now do for each input symbol (token):
[42] LP:na←ac[s←EM[1];a←ω[i+1]] ⍝ which action to do next?
[43] →(0>na)⍴reduce ⍝ either reduce action
[44] →(0<na)⍴shift ⍝ or shift action
[45] →(accept≡s,a)⍴acc ⍝ or accepting input
[46] err:→0,r←(-⍴r),r ⍝ or syntax error.
[47] shift: ⍝ Shift action:
[48] ⍝ push negative rule number(na) and
[49] ⍝ current token (a) onto stack.
[50] EM←na,a,EM
[51] r←r,i ⍝ append token no i to result
[52] →LP,i←''⍴i+1 ⍝ increment input pointer
[53] reduce: ⍝ Reduce action
[54] r←r,na ⍝ append neg. rule number (na) to result.
[55] EM←(2×nr)[na←|na]↓EM ⍝ pop stack (discard rule used)
[56] ⍝push nextstate, leftsideofruleused
[57] EM←(gt[EM[1];|ls[na]],ls[na]),EM
[58] →LP ⍝ continue with next input symbol
[59] acc:r←(⍴r),r ⍝ Accept input
    ∇

    ∇ ⍙⍙Names ⍙assign ⍙⍙Values
[1] ⍝⍝ Model strand assignment:
[2] ⍝⍝ 'var1 var2 ... ' ⍙assign EnclosedVector
[3] ⍙⍙Names←(vfe 0≠⍤¨>⍴⍤¨>⍙⍙Names)/⍙⍙Names←' ' LTOV ⍙⍙Names
[4] ⎕signal( (⍴⍙⍙Names)≡⍴⍙⍙Values)⍴5
[5] ⍎vfe '⋄',⍤¨>⍙⍙Names,⍤¨>(<'←>⍙⍙Values['),⍤¨>(⍕⍤¨>(⍳⍴⍙⍙Values)),⍤¨>']'
    ∇

    ∇ r←dupout ω
[1] r←((⍳⍴ω)=ω⍳ω)/ω
    ∇
```

Figure 12: APL Parser model.