Formulating Type Tagged Parse Trees as Forth Programs.

Dr Campbell Ritchie and Dr Bill Stoddart
Formal Methods and Programming Research Group
Teesside University, UK

August 24, 2009

Abstract

We describe a two pass compilation technique for converting infix expressions to postfix, in which a first pass produces a type tagged tree and a second pass provides type checking and generates Forth code.

The expression language we consider includes sets and sequences, ordered pairs, relations, functions, lambda expressions, strings and arithmetic expressions. The type theory we employ in our infix expression language is an extension of that of the B Method, which is based on the use of power set and product operations as type constructors.

The main interest as a Forth research topic is the ease with which the compiler can be implemented in Forth. The first pass compiler is a set of mutually recursive functions that produce a type tagged tree. The second pass is implemented by providing a Forth definition corresponding to operations found at the non-terminal nodes of the parse tree, and over which is distributed the responsibility of performing type checking and compilation of the final executable code. Due to the close correspondence between parse trees and postfix expressions, the parse tree can be identified with a Forth program and the execution of the second pass of the compiler with the execution of this program.

1 Introduction

We consider the translation to postfix (Forth) of an expression language whose terms include sets, ordered pairs, higher order functions, lambda expressions, strings and arithmetic expressions. We use strong typing on set expressions, so that all elements of a set must be of the same type. Set and sequence expressions can be nested. Some forms may be represented in multiple ways, e.g. small sets of integers can be represented as bitsets, as well as in a standard format. Functions may be represented as executable code, or as sets of ordered pairs with element lookup.

The translation to Forth ensures the source expression is correctly typed, and deals with any conversions required to cope with differing data representations.

This work is part of an attempt to create a reversible high level language with a formal semantics and with a compiler that targets a reversible version of Forth, the "Reversible Virtual Machine" or RVM.

We use a two pass compiler in which the first pass produces a type tagged parse tree. Since there is a close correspondence between parse trees and postfix expressions, we are able to think of the tree as a Forth program.

These ideas, in a basic form, were presented at last years Euro-Forth, but using an expression language limited to integer and floating point arithmetic expressions. When compiling the expression `1+2.5`, the output from the first pass is “ `1`” `INT` “ `2.5`” `FLOAT +_`, and this is a Forth program which produces the output from the second pass, and its type. `INT` and `FLOAT` are constants denoting types. The operation `+_` expects four arguments, these being two expression strings and their types, which can be either `INT` or `FLOAT`. The operation `+_` generates two stack outputs, the postfix expression, “`1 FLOAT 2.5 F+`” and its type, `FLOAT`. The implementation technique provides the possibility of operators which are polymorphic, providing different functions according to the type of their arguments.

Analogously to `+_` we have operations `-_`, `*_` and so on. Each operation in the language has a corresponding operation definition used in the second pass compiler, whose name is formed by appending an underscore to the original operation name. This is used in place of the original name in the output generated by the first pass of the compiler, and is executed during the second pass of the compiler.

This convention is maintained in the present paper. However, we now have two problems that were not present in the previous discussion. Firstly, since strings are now a part of our expression language, we need to represent string expressions which contain string expressions. We do this by using the facility, provided by Unicode, of using opening and closing quotes, as in:

“ “`jim`” $\mapsto$ `1234,` “`fred`” $\mapsto$ `2345` ”

A second problem is that as well as the atomic types `INT` and `FLOAT` we now have type expressions of arbitrary complexity. We represent these types by the strings that will denote them in the final Forth code.

The remainder of the paper is structured as follows. In Section 2 we discuss aspects of our expression language, including sets, sequences, ordered pairs, and types. In section 3 we discuss the formal syntax of the expression language, using the syntax definition to derive the functions used to implement the first pass of the compiler. In Section 4 we consider lexical analysis and the first pass of the compilation process that generates type tagged trees. In Section 5 we discuss the second pass, showing how type information is passed up the tree and how type checking and nested set and sequence structures are handled. In Section 6 we conclude and discuss future work.

2 Sets and Types

We write mathematical expressions in maths font and fragments of Forth code in teletype font. The set extension $\{1, 2, 3\}$ is written in RVM_Forth as
`INT { 1 , 2 , 3 , }`. As is usual in Forth, we adopt a programming style in which each lexical item is a Forth operation. `INT` provides the type of the set elements. The operation `{` opens a new set construction. The commas within the set construction represent an operation that takes an element from the stack and compiles it into the current set, and the operation `}` closes the set construction and leaves a reference to the set on the stack.

The set $\{\{1,2\},\{4\}\}$ has elements which are sets of integers. It may be represented in RVM_Forth as

```
INT SET { INT { 1 , 2 , } , INT { 4 , } , }
```

The following set of string and integer pairs:

$\{$"*Bill*" $\mapsto$ 2673, "*Campbell*" $\mapsto$ 2680, "*Frank*" $\mapsto$ 2680$\}$ may be rendered in Forth as:

`STRING INT PAIR { " Bill" 2673 ↦ , " Campbell" 2680 ↦ , " Frank" 2680 ↦ , }`.

We refer to sets of pairs as "relations". The set of left hand elements (in this case $\{$"*Bill*", "*Campbell*", "*Frank*"$\}$ is the relation's domain, and the set of right hand elements is its range. We can apply relations as functions. If the relation just given is applied to the argument "`Bill`" the result will be `2673`. If the relation is called R this would be represented in the infix expression language as R("*Bill*") and in Forth as `" Bill" R APPLY`.

If the relation is inverted and applied to the value `2680` there is a choice of results: "`Frank`" or "`Campbell`". Such choices may be made non-deterministically and be revised on backtracking.

Our types consist of basic sets, such as `INT` and `STRING`, together with the constructors `SET` and `PAIR`. In this paper we are only concerned with the postfix representation of types. If `T` is a type, `T SET` is the type whose elements are sets of elements of type `T`. If `U` is also a type, `T U PAIR` is the type whose elements are ordered pairs, with the first element of each pair belonging to `T` and the second element to `U`.

We use sets as a general way of representing data. As well as the set operations of union and intersection, we provide operations that are more specifically related to data updates and data queries. If R is a relation and U a relation of the same type, $R \oplus U$ is the relation R updated by entries from U. This expression is represented in Forth as `R U OVERRIDE`.

Type checking for an override operation consists of checking that both arguments are relations of the same type. An operation that requires a slightly more complex type analysis is "domain restriction", denoted by $\lhd$. If R is a relation of elements between of type T and elements of type U, and if S is a set of elements of type T, then $S \lhd R$ is the relation from T to U consisting of the pairs in R whose first elements are in S. In terms of type checking with our postfix type language, we need to check that the postfix representation of the set S, which we wrote as `S`, is of type `T SET` for some postfix type `T` and that the postfix type of `R` is `T U PAIR SET`.

Sequences, in our canonical set representations, are just sets of ordered pairs where the domain elements run from 1 to n. The type of a sequence of elements from `T` is `INT T PAIR SET`. Not all data of this type are sequences of course. Since sequences are just sets, is is permissible to take the union or intersection of two sequences, though the result will only be a sequence under certain special conditions. As an example of where taking the union of two sequences can be useful, suppose we are given two sequences s and t and we want to test whether s is a prefix of t. A suitable test is $s \subseteq t \ \wedge \ (s \cup t) = t$.

3 Expression Grammar and Compilation Functions

Our grammar is written in Hehner's Bunch Theory. Appendix A describes this notation. Appendix B gives the full grammar. We provide sufficient comments in the text for a reader to follow our general approach without consulting the appendices.

The top level rule for our grammar is:

$$E \;=\; E \text{ “}\mapsto\text{”} \; E_0,\; E_0$$

E is the bunch of strings in our infix expression language. Terminal symbols are shown in quotes. The maplet symbol $\mapsto$ is an infix symbol which yields an ordered pair; it is the lowest precedence operator in the grammar, and is left associative. E_0 is the bunch of strings from our expression language which do not contain $\mapsto$ at the top level. In parsing an expression we first look for the lowest precedence symbol, and since it is a left associative symbol we scan from the right to find the rightmost occurrence of such a symbol in the expression. Let PE be the function that takes a string which is a valid infix expressions, and returns the string that would be generated by the first pass of the compiler, then if e is any string from E and e_0 any string from E_0 we have the following properties:

$$PE(e \text{ “}\mapsto\text{”} \; e_0) \;=\; PE(e)\; PE(e_0) \text{ “}\mapsto\_\text{”}$$
$$PE(e_0) \;=\; PE_0(e_0)$$

These cover the two cases, where the expression to be parsed contains a maplet symbol at the top level, and where it does not. Note that we compile the tagged maplet symbol $\mapsto\_$, which will be used in the second pass to process type information and produce a maplet operator for the final code.

The next level of precedence contains the symbols $\setminus, \cup, \cap, \oplus$. Again these are left associative. The associated grammar rule is:

$$E_0 \;=\; S \text{“}\setminus\text{”} S_0,\; S \text{“}\cup\text{”} S_0,\; S \text{“}\cap\text{”} S_0 \; S \text{“}\oplus\text{”} S_0, E_1$$

Here, S is the bunch of all set expression, S_1 the bunch of strings from S without any of $\setminus, \cup, \cap, \oplus$ at the top level, and E_1 the bunch of expressions from E_0 without any of $\setminus, \cup, \cap, \oplus$ at the top level.

The rule tells us that any string from E_0 is either a string from S followed by one of $\setminus, \cup, \cap, \oplus$ followed by a string from S_0, or else it is a string from E_1.

Let PS, PS_0 and PE_1 be functions that parse strings from S, S_0 and E_1 respectively. Let op be one of $\cup, \cap, \oplus$, e_0 a string from E_0, s a string from S, s_0 a string from S_0, and e_1 a string from E_1. Let *space* be a sting containing just a space character. Then we can characterise PE_0, the function to compile code for strings from the bunch E_0, with the following equations.

$$PE_0(s \; op \; s_0) \;=\; PS(s)\; PS_0(s_0)\; space\; op \text{“}\_\text{”}$$
$$PE_0(e_1) \;=\; PE_1(e_1)$$

Moving to the next level of precedence we have the symbols for domain restriction and domain subtraction, which are of necessity right associative:

$s \lhd r$ has the same type as r and hence $s_1 \lhd s_2 \lhd r$ must parse as $s_1 \lhd (s_2 \lhd r)$. The grammar rule for this level is:

$$E_1 \ = \ S_2 \lhd S_1, \ S_2 \lhd\!\!- S_1, E_2$$

Now if e_1 is a string from E_1, s_1 a string from S_1, s_2 a string from S_2, *op* a string from "$\lhd$","$\lhd\!\!-$" and e_2 a string from E_2 we can characterise the compiling function PE_1 as follows:

$$PE_1(s_2 \ op \ s_1) = PS_2(s_2) \ PS_1(s_1) \ space \ op \ \text{“\_”}$$
$$PE_1(e_2) \ = PE_2(e_2)$$

Other grammar rules involving binary operators lead to compiling functions in the same way.

Now let us see how a set extension is compiled. The grammatical form of a set extension is:

“{” L “}”

Where L is a list of expressions, with grammatical description:

$L \ = \ E, \ L$ “ ,” E

The set extension $\{1, 0, x + 1\}$ will compile to:

{_ “ 1” “ INT” ,_ “ 0” “ INT” ,_ “ x” “ INT” “ 1” “ INT” +_ ,_ }_

Let the function that performs parsing of set extensions be *PSE* and the function that parses a list be *PL*. Let *list* be a string from L, and e a string from e. Then we can characterise *PSE* and *PL* with these equations.

$$PSE(\text{“ \{”} \ list \ \text{“ \}”} \ = \ \text{“ \{\_”} \ PL(list) \ \text{“ \}\_”}$$
$$PL(list, e) \ = \ PL(list) \ \text{“,\_”} \ PE(e)$$
$$PL(e) \ = \ PE(e)$$

4 Lexical Analysis and First Pass Compilation

Whereas classical lexical analysis reads and distinguishes tokens by reading left to right, we have a bidirectional lexical analyser which finds the lowest priority connective at each scan. This is implemented by two Forth functions. `RL-LEX` scans from right to left and is used to search for left associative connectives, and `LR-LEX` scans from left to right and is used to search for right associative connectives. These words take as input parameters a string address and a sequence of tokens to be searched for. Where tokens are multi-character and one token may be a prefix of another, the longer token is placed first in the sequence to prevent spurious detection of the shorter token.[1] If a token is found, the returned values are the part of the expression string that lies before the token, the part that lies after, and the token. If no token is found, the returned values are the expression string and two null values.

Whilst searching we only check for tokens at the “top level”. We are not at the top level if we are currently inside some kind of bracket structure. The four types of bracket defined in the expression language are precedence brackets (. . .), set brackets (. . .), sequence brackets [. . .], and matching string quotes “. . . ”. Whilst performing a lexical scan, if a bracket is detected, we stop looking for the

[1] Where a symbol at a lower order of precedence is a prefix of a symbol with a higher order of precedence there is a potential problem which we have not tried to solve.

given tokens and look instead for the relevant brackets until we return to the top level. For example if scanning right to left and a closing set bracket is found, we then look for opening and closing set brackets until the original bracket is matched; we ignore any sequence or structuring brackets, but we must change mode if a quote bracket is detected, since within quotes any set brackets that occur are part of a string literal rather than part of the expression's structure.

The first pass compiler has a separate Forth function for each syntactic category in the expression grammar. We obtain a collection of mutually recursive functions. We use a uniform naming convention in which, for example, strings from the syntactic category E are translated to postfix as described by the mathematical function PE, which is implemented by the Forth function `PE`. The top level function `PE`, which can parse any expression, is the last function to be defined, but is required by many of the functions that precede it, e.g by functions that handle bracketed expressions. To handle this situation RVM-Forth has a defining work `OP` used in these circumstances as:

```
NULL OP PE ( now we can refer to PE but not execute it)

.... (define all compiler functions)

: P ( define the functionality required of PE) ... ;
' P to PE ( assign the functionality of P to PE)
```

Within the definition of `P` we look for the rightmost maplet symbol at the top level:

```
:  P ( az1 -- az2, parse an expression from E, leaving az2 the
  first pass postfix translation of the expression az1.  )
  (:  VALUE e :)
    e STRING [ "↦" , ] RL-LEX
    VALUE BEFORE VALUE AFTER VALUE OP-STRING
    OP-STRING NULL =
    IF ( e did not contain a ↦ symbol at the top level)
      BEFORE PE0
    ELSE
      BEFORE RECURSE AFTER PE0
      SPACE^ OP-STRING _^ AZ^
    THEN
  1LEAVE ;
```

Within this definition, `AZ^` performs catenation of asciiz strings (the string form used in this project), `SPACE^` appends a space to a string, and `_^` appends an underscore.

As a second example we describe PE1, which looks for the right associative domain restriction and domain subtraction operations.

```
:  PE1 ( az1 -- az2, parse an expression from E1, leaving az2,
the first pass postfix translation of the expression az1.  )
  (:  VALUE e :)
    e STRING [ "◁—,  ◁" , ] LR-LEX
    VALUE BEFORE VALUE AFTER VALUE OP-STRING
    OP-STRING NULL =
    IF ( e did not contain ◁— or ◁ at the top level)
      BEFORE PE2
    ELSE
      BEFORE PS2 AFTER PS1
      SPACE^ OP-STRING _^ AZ^
    THEN
  1LEAVE ;
```

We see that these functions are very similar. They either fail to find any symbols at the current precedence level and fall through into a function which deals with higher precedence symbols, or they find a symbol, apply appropriate functions to parse the text before and after the symbol, and catenate the results followed by the operator with an appended underscore.

Where parsing functions are more complex, it is because the particular symbol found reveals more about the before and after text. For example consider the grammar rule:

$$E_2 = S_1 \text{“}\leftarrow\text{”} E,\ W_1 \text{“}\frown\text{”} W_2,\ S_1 \text{“}\rhd\text{”} S_2,\ S_1 \text{“}\mathrel{-}\rhd\text{”} S_2,\ S_1 \uparrow A,\ S_1 \downarrow A,\ E_3$$

In this case, after the string belonging to E_2 is split into before and after strings by RL-LEX, the functions to be applied to the before and after text depend on the symbol that has been found. For example if the symbol is "$\leftarrow$" we are appending a value to a sequence. The text before the symbol is a set expression from S_1, to be processed by PS1, and following text is an expression from E_3, to be processed by PE3. If, however, the symbol found is the catenation symbol "$\frown$", we are catenating two sequences or two string expressions. In that case the before text is from W_1 and the after text is from W_2, and these are to be processed by PW1 and PW2 respectively. Implementation of PE2 thus requires a case analysis based on the symbol found.

The lexical analysers have a wider use than detecting infix symbols. For example to parse a function application, which has syntactic form given by the equation:

$$F \ = \ S_2 \text{ “(”} L \text{“)”},\ F \text{ “(”} L \text{“)”}$$

We start from the right of the expression, move back one character, and search for the token "(". The before string is then the function expression and the after string the argument list. To parse the argument list with function PL we search right to left for a comma. If none is found the argument list is a single expression to be proceeded with PE, if a comma is found the following text is an expression to be processed with PE and the before text is again an argument list to be recursively processed by PL.

The first pass compiler collects type information from numeric literals and numeric strings and from identifiers, whose type is held in a symbol table (assumed to be already in existence). The result of a first pass compilation is a postfix expression containing a mixture of special operators and type tagged literals and identifiers. The special operators, which will run during the second pass, are named with the names of the corresponding operators in the original expression, with additional underscores: $\mapsto\_$ $\cup\_$ $\cap\_$ and so on. We could have re-used the existing names and written the definition of these second pass operations in a separate word list. However, the present choice of names serves to help the reader (and ourselves) to remember which pass of the compiler is currently being discussed.

5 Second Pass Type Checking

The intermediate postfix code produced by the first pass of the compiler may be viewed as a tree which has the literals and identifiers which appear at its leaves tagged with type information. As the intermediate code is executed, the types of more complex sub-expressions, such as set structures, are derived, along with the postfix code for these sub expressions.

We illustrate the technique by considering the compilation of some set extensions, where, in additional to deriving the types of the sets concerned, the second pass compiler must perform type checks to ensure that all sets are homogeneous in the sense that any set may only contain elements of one particular type.

A simple example would be compilation of $1, 3, 5$. The first pass produces the following text:

```
{_ " 1" " INT" ,_ " 2" " INT" ,_ " 3" " INT" }_
```

In the following trace we see how this is translated into two strings, representing a Forth set expression and the type of the set. In the trace, strings which are on the stack are represented by the form they take in the Forth source code, and the distinction is made by whether they occur in the "Forth Code" column, or the "Stack" column. `NULL`, a constant with value zero representing lack of information about the types of a set element, is shown in the same way.

Forth Code	Stack
	Empty
{_	" {" NULL
" 1" " INT"	" {" NULL " 1" " INT"
,_	" { 1 ," " INT"
" 3" " INT"	" { 1 ," " INT" " 3" " INT"
,_	" { 1 , 3 ," " INT"
" 5" " INT"	" { 1 , 3 ," " INT" " 5" " INT"
}_	" INT { 1 , 3 , 5 , }" " INT SET"

The `{_` word places an initial set expression string (consisting of just an open set brace) onto the stack followed by a `NULL`, signifying lack of information about the type of the set's elements.

The word `,_` takes four string arguments, a partial set expression, a representation of the type of the set elements (or `NULL` if the type is not yet known), an expression giving the current element, and an expression giving its type. It

checks whether the type of the new element is the same as that of previous elements, and extends the partial set expression to include the new element.

Finally, `}_` takes the same four string arguments as `,_`. It checks the type of the final element, extends the set expression to include this element and adds the closing brace, then prepends the type of the set element, returning the resulting string as its first return value. Its second return value is the type of the set, formed by appending " `SET`" to the element type.

The use of the stack during the second pass provides for this approach to cope seamlessly with nested set structures.

We now look at an example of a relation in which some elements are strings. We consider the expression:

$$\{\text{“ } joe\text{”} \mapsto 90, \text{“ } Methuselah\text{”} \mapsto 900\}$$

The first pass compilation yields

`{_ " " joe"" " 90" ↦_ ,_ " " Methuselah"" " 900" ↦_ }`

and here we need nested quotes; we are building a calculus of partial expression strings, and in such a partial expression string we now have a string literal. After the second pass we obtain two stack items, the expression string:
" `STRING INT PAIR { " joe" 90 ↦ , " Methuselah" 900 ↦ , }`"
and its type, which is " `STRING INT PAIR SET`".

The new second pass operation introduced by this example is ↦_ This expects four string arguments: a first expression, its type, a second expression, and its type. It produces two results: a new expression combining the previous expressions as a pair, and the type of the pair. It checks the types are equal, then catenates the expressions and appends a maplet symbol to obtain the new expression. If T and U are the expression types, then the new expression has type T U `PAIR`.

The second pass compiler for operations handling set and sequence descriptions are shown below, omitting `[_` and `]_` which are very similar to `{_` and `}_`.

```
: {_ "  { " NULL ( NULL represents the lack of type information at
the start of a set expression construction ) ;

: ,_ ( set-exp:$ type1:$ element-exp:$ type2:$ -- exp' type2 )
  (: VALUE set-exp VALUE type1  VALUE element-exp  VALUE type2 :)
    type1 NULL <> IF
      type1 type2 STRING= NOT
      ABORT" Non-homogeneous types in set"
    THEN
    set-exp element-exp AZ^  "  , " AZ^ type2
  2LEAVE ;

: \u21a6_ ( exp1:$ type1:$ exp2:$ type2:$ -- pair-exp:$ type:$ )
  ( \u21a6 is the maplet character )
  (: VALUE exp1  VALUE type1  VALUE exp2  VALUE type2 :)
    exp1 exp2 AZ^ "  \u21a6" AZ^  type1 type2 AZ^  "  PAIR" AZ^
  2LEAVE ;

: }_ ( set-exp:$ type1:$ element-exp:$ type2$ -- set-exp' set-type )
```

```
(: VALUE set-exp  VALUE type1  VALUE element-exp  VALUE type2 :)
  type1 NULL <> IF
    type1 type2 STRING= NOT
    ABORT“ mismatched type in final set element”
  THEN
  type2 SPACE^ set-exp AZ^ element-exp AZ^  “  , } ” AZ^
  type2 “  SET” AZ^
2LEAVE ;
```

As a final example of type checking instances of the domain restriction operation, i.e expressions for the form $S \lhd R$. Here, S must be an expression that represents a set, and R an expression whose domain elements are of the same type as S. If the type of R (expressed in postfix) is `T U PAIR SET` then the type of S is `T SET`. The second pass operator $\lhd_\perp$ must check that each type has the correct form, and that the type of the elements of S is equal to the type of the elements of the domain of R. Here arity checks, scanning right to left, provide the key to dismantling postfix type expressions into their component parts.

6 Conclusions and Future Work

We have described a two pass compilation technique applicable to fairly complex expressions. We believe the technique will extend to the compilation of full operations. The first pass of a compilation produces a parse tree, in which literal values and identifiers, which form the leaves of the parse tree, are tagged with type information. The operations at the nodes of the parse tree match the operations of the expression language, but perform type analysis and compilation functions. The first pass of the compiler is provided by a collection of mutually recursive functions, each of which is designed to compile inputs taken from a specific syntactic category, established in the grammar. The second pass of the compiler is provided by definitions corresponding to operations appearing at the non-leaf nodes of the parse tree.

A Bunch Notation and Grammars

Grammars are usually written in terms of "production rules". We want to see a grammar a a set of simultaneous equations on strings. We want, also, to consider both non-terminal symbols, E, E_0 etc. and terminal symbols "$\cup$", "$+$" etc. to be collections of strings, with non terminal symbols generally denoting a plurality of possible strings.

Mathematical descriptions of grammars rely on set theory, but this has some disadvantages. Set theory provides two things simultaneously: collection and packaging: the set $\{1, 3, 5\}$ both collects together the elements 1,3 and 4, and packages them up as a new element. for grammar descriptions we want the power to collect, without being obliged to package, That is we want the ability to deal in plurality as well as elements.

In Eric Hehner's Bunch theory, a bunch is the contents of a set. This $1, 3, 5$ is the bunch that forms the contents of the set $\{1, 3, 5\}$ Collection and packaging become orthogonal concepts. The comma is now an operation rather than syntax. It signifies bunch union. Thus if A and B are bunches then A, B is a bunch consisting of the elements from A and the elements from B.

In our description of grammar, we are dealing with bunches of strings. The main operation is string catenation. We can write is as $s^\frown t$ but we elide the catenation symbol and just write $s\ t$. Where catenation is applied to bunches of more than one element, it is lifted in an obvious way: if $X =$ "*John*, *Tom*" and $Y =$ "*Jones*, *Smith*" then $X\ Y =$ "*JohnJones*", "*JohnSmith*", "*TomJones*", "*TomSmith*"

Relating our use of bunch notation to classical grammar descriptions, comma can be thought of as choice, and juxtaposition as sequencing. Taking an example line from the grammar:

$A\ =\ A\ \text{“}+\text{”}\ A_0,\ A\ \text{“}-\text{”}\ A_0,\ A_0$

this tells us the bunch S is made up of strings from the three bunches A "$+$" A_0, A "$-$" A_0 and A_0. The bunch A "$+$" A_0, consists of strings made up of a string from A followed by "$+$" followed by a string from A_0, and so on.

B Expression Language Syntax

Symbols listed in order of precedence, low precedence symbols first, symbols of equal precedence are enclosed in brackets.

$\mapsto\ (\setminus\ \cup\ \cap\ \oplus)\ (\lhd\ \lhd\!-)\ (\leftarrow\ \frown\ \rhd\ \rhd\!-\ \downarrow\uparrow)\ (+\ -)\ (*\ /)\ \tilde{}$

Most binary connectives are left associative, the exceptions being $\lhd$ and $\lhd\!-$ which are right associative.

Non terminal symbols for the grammar.

L a comma separated list of expressions E an expression λ a lambda expression S an expression representing a set W an expression representing a string or a set F an expression representing a function application A an arithmetic expression N numeric literal \$ string literal I an identifier

E_0 expressions from E without $\mapsto$ at the top level.
E_1 expressions from E_0 without $\setminus\ \cup\ \cap\ \oplus$
E_2 expressions from E_1 without $\lhd\ \lhd\!-$
E_3 expressions from E_2 without $\leftarrow\ \frown\ \rhd\ \rhd\!-\ \uparrow\downarrow$
E_4 expressions from E without $+\ -$
E_5 expressions from E_4 without $*\ /$

E_6 expressions from E_5 without ˜ (unary minus)

S_0 expressions from S without $\setminus\ \cup\ \cap\ \oplus$
S_1 expressions from S_0 without $\lhd\ \lhd\!-$
S_2 expressions from S_1 without $\leftarrow\ \frown\ \rhd\ \rhd\!-\ \uparrow\downarrow$

W_0 expressions from W without $\setminus\ \cup\ \cap\ \oplus$
W_1 expressions from W_0 without $\lhd\ \lhd\!-$
W_2 expressions from W_1 without $\leftarrow\ \frown\ \rhd\ \rhd\!-\ \uparrow\downarrow$

A_0 expressions from A without $+\ -$
A_1 expressions from A_0 without $*\ /$
A_2 expressions from A_1 without ˜ (unary minus)

B.1 Expression grammar equations

$E = E\text{“}\mapsto\text{”}E_0, E_0$
$E_0 = S\text{“}\setminus\text{”}S_0, S\text{“}\cup\text{”}S_0, S\text{“}\cap\text{”}S_0, S\text{“}\oplus\text{”}S_0, E_1$
$E_1 = S_2\text{“}\lhd\text{”}S_1, S_2\text{“}\trianglelefteq\text{”}S_1, E_2$
$E_2 = S_1\text{“}\leftarrow\text{”}E, W_1\text{“}\frown\text{”}W_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\uparrow\text{”}A, S_12\downarrow\text{“}A, E_3$
$E_3 = A\text{“}+\text{”}A_0, A\text{“}-\text{”}A_0, E_4$
$E_4 = A_0\text{“}*\text{”}A_1, A_0\text{“}/\text{”}A_1, E_5$
$E_5 = \text{“˜”}A_1, E_6$
$E_6 = N, \$, I, F\text{“(”}L\text{“)”}, \text{“”}L\text{“”}, \text{“[”}L\text{“]”}, \text{“(”}E\text{“)”}, \text{“(}\lambda\text{”}IE\text{“)”}$

$S = S\text{“}\setminus\text{”}S_0, S\text{“}\cup\text{”}S_0, S\text{“}\cap\text{”}S_0, S\text{“}\oplus\text{”}S_0, S_0$
$S_0 = S_1\text{“}\lhd\text{”}S_0, S_1\text{“}\lhd\!-\text{”}S_0, S_1$
$S_1 = S_1\text{“}\leftarrow\text{”}E, S_1\text{“}\frown\text{”}S_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\uparrow\text{”}A, S_1\text{“}\downarrow\text{”}A, S_2$
$S_2 = I, F, \text{“”}L\text{“”}, \text{“[”}L\text{“]”}, \text{“(”}S\text{“)”}, \lambda$

$W = S\text{“}\setminus\text{”}S_0, S\text{“}\cup\text{”}S_0, S\text{“}\cap\text{”}S_0, S\text{“}\oplus\text{”}S_0, W_0$
$W_0 = S_1\text{“}\lhd\text{”}S_0, S_1\text{“}\lhd\text{”}S_0, W_1$
$W_1 = S_1\text{“}\leftarrow\text{”}E, W_1\text{“}\frown\text{”}W_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\rhd\text{”}S_2, S_1\text{“}\uparrow\text{”}A, S_1\text{“}\downarrow\text{”}A, W_2$
$W_2 = I, F, \text{“”}L\text{“”}, \$, \text{“(”}W\text{“)”}, \lambda$

$A = A\text{“}+\text{”}A_0, A\text{“}-\text{”}A_0, A_0$
$A_0 = A_0\text{“}*\text{”}A_1, A_0\text{“}/\text{”}A_1, A_1$
$A_1 = \text{“˜”}A_1, A_2$
$A_1 = \text{“˜”}A_1, A_2$
$A_2 = I, F, N, \text{“(”}A\text{“)”}$

$\lambda = \text{“}\lambda\text{”}I\text{“}\bullet\text{”}E$
$L = E, L\text{“,”}E$
$F = S_2\text{“(”}L\text{“)”}, F\text{“(”}L\text{“)”}$