A Compiler which Creates Tagged Parse Trees and Executes them as FORTH Programs

Campbell Ritchie and Bill Stoddart

Formal Methods and Programming Research Group, Teesside University, Middlesbrough, England

Abstract

Most compilers use separate scanning and parsing, and scan their input from left to right. Lynas and Stoddart (2008) demonstrated an alternative technique where an input string is split into multiple sub-expressions, divided at each operator.

We demonstrate an expression parser which extends this work. It runs in two passes. The first pass scans code left-to-right or right-to-left depending on the associativity of the operator sought, creating pointers to strings which together represent a parse tree. At the end of the first pass, it uses lexical analysers to infer the type of each token, or find the type in a lookup table.

The types follow the theory of the B language, building up more complex types as if with the powerset ($\mathbb{P}$) and Cartesian product ($\times$) operators. The "parser" operations, whose titles all begin with "P" assemble the elements into a postfix model of the parse tree, with the operators tagged with a _ character.

The second pass can be run together with the first, or later, allowing the intermediate representation of the code, as a tagged parse tree, to be inspected. The output can be executed as FORTH code; each operator tagged with a _ is matched by a corresponding operation which tests the types supplied, and leaves the type and postfix representation on the stack; the latter is identical to the FORTH representation of the original expression.

We discuss possible uses of such a compiler, and possible problems about its efficiency, since it runs in quadratic time.

Keywords

FORTH, compiler, recursion, parse tree, postfix notation, type tagging.

Address for Correspondence

C Ritchie, Formal Methods and Programming Research Group, Rm P1.10, Teesside University, Borough Rd, Middlesbrough, England TS1 3BA.

work@critchie.co.uk and *bill@tees.ac.uk*

Introduction

Many programming languages are written in infix notation (e.g. "1 + 2"), but most computers execute the contents of a stack, which is most easily expressed in postfix notation (e.g. "1 2 +"). Both languages used for training only (e.g. "VSL" = very simple language (Bennett (1996)) and commercial production languages (e.g. Java™ (Gosling *et al* (2005)) are compiled by conversion to an intermediate form in postfix notation. Since FORTH programs are written in postfix notation, a compiler which produces its intermediate representation in a form similar to FORTH syntax can use a FORTH virtual machine as its back-end. This allows one to write a compiler consisting only of its front-end.

Lynas and Stoddart (2008) introduced such a compiler, originally for teaching undergraduates. This parses an expression differently from most compilers. Instead of scanning the code and dividing it into tokens, their compiler scans the code for operators and connectives, splitting the expression into sub-expressions at each connective. After the code has been subdivided at every operator, it is held in pointers to strings, which together represent a parse tree. Later, a second pass of operations is used to rejoin the strings along with the operators. This resultant string is identical to FORTH syntax for that expression, and can be executed as FORTH code.

Lynas and Stoddart (2008) introduced operations called by an operator followed by an underscore character; for example, the +_ -_ *_ and /_ operations handle addition, subtraction, multiplication, and division respectively. The first pass of compilation produces a parse tree, which can be tagged for types; Lynas and Stoddart's example shows "1 + 2.5" being converted to

```
" 1" INT " 2.5" FLOAT +_
```

by the first pass of compilation.

The values INT and FLOAT are constants representing integer and floating-point numbers, and the other two values are Strings containing the operands. The second pass executes the values on the stack as a FORTH program. The +_ operation requires four values on the stack representing a parse tree. It compares the types, and places two values on the stack: a pointer to the output String

```
" 1 S>F 2.5 F+"
```

which is itself executable as FORTH code, and the constant FLOAT denoting the type of the result. This implementation technique permits one to use polymorphic operators, providing different operations for different types of operand.

Rather than passing through the code from left to right, separating it into tokens, this parsing technique seeks operators in the code and splits an expression into sub-expressions. It scans the code from left to right for a right-associative operator, and *vice versa*.

We call programs such as +_ tagged operations. We showed plans to use tagged operations, which can be executed as FORTH code last year (Ritchie and Stoddart, 2009). At the end of the first stage of parsing, the operator, along with an underscore "_" is added to the intermediate representation. This produces an output which can be executed as the second stage of parsing. Rather than using integers to represent types, we use strings, allowing types of arbitrary complexity to be declared. This same grammar as we used before, expanded by the addition of Boolean expressions, is shown as an appendix to this paper.

The Structure of this Paper

We first look at sets, and how our typing theory uses sets. Then we describe how this compiler has grown from earlier work, and the three operations used in the parser, the “P” operations which splits the text around an operator, and two tagged operations, ⇔_ and +_. Then follows a section about creating and manipulating sets, and the results of using this parser. Finally, we discuss its potential use, and planned future work.

Sets And Types

Following the type theory of the formal specification and development languages Z and B (Abrial, 1996), we regard each value as having the type of the set to which it belongs. A whole number is a member of the integer set $\mathbb{Z}$ expressed in FORTH as “INT”, and a decimal fraction a member of the set $\mathbb{R}$ for real numbers called “FLOAT” in FORTH. It is possible to use the constructor $\mathbb{P}$ for power-set to produce a set whose members are themselves sets; this can be expressed as “INT POW”. Similarly the Cartesian Product operator $\times$ can be used to produce a set or ordered pairs; for example the ordered pair (1, 2) or $1 \mapsto 2$ is a member of the set $\mathbb{Z}\times\mathbb{Z}$, called “INT INT PROD”. Using these constructors, one can create set types of arbitrary complexity, which the set package introduced to RVM FORTH by Stoddart and Zeyda (2002) handles.

Methods

We maintain the convention of Lynas and Stoddart (2008) of appending an underscore to the operator, to give the name of an operation which tests the appropriateness of the typing for that operator. There is however a new problem; we are using arbitrarily complex types, and compound types, in addition to the built-in types “INT” “FLOAT” and “STRING”. These cannot be readily expressed as constants, but can be incorporated in Strings which denote those types in the final FORTH code, and which can be compared and manipulated with simple String operations.

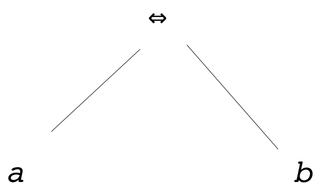
Compiling the Expression Grammar

The grammar is shown as an appendix. It consists of a series of recursive equations, starting with the lowest-precedence operator $\Leftrightarrow$ and gradually increasing precedences. The lowest-precedence operator appears in this line:

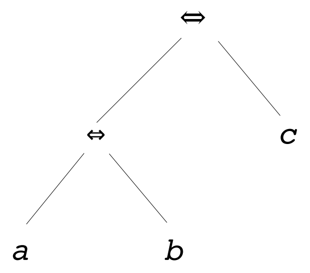
$$E = E \Leftrightarrow E_1,\ E_1$$

. . . meaning that the Strings forming E (for expression) consist of strings from E followed by the $\Leftrightarrow$ symbol and a string from E_1, as well as any strings in E_1. Note this grammar permits left recursion.

The equivalence (“iff”) symbol $\Leftrightarrow$ takes two values, to test for equivalence. It is a binary, infix, left-associative operator. The expression $a \Leftrightarrow b$ can be parsed to form this parse tree:-

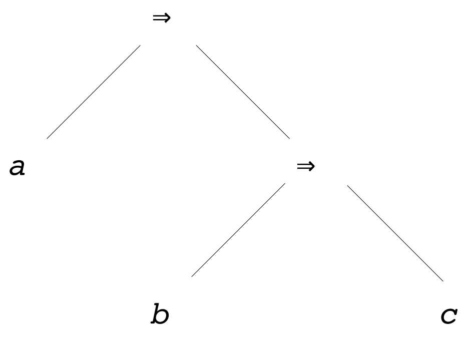


. . . and the expression $a \Leftrightarrow b \Leftrightarrow c$ gives this tree:



In the case where several operators are included in an expression, one splits the expression on the lower precedence operators first. In the case of a left-associative symbol, where there is more than one operator with the same low precedence, the rightmost operator is used to split first (for right-associative operators, one splits first on the leftmost symbol). The values in *a b* and *c* will themselves be expressions from lower lines in the grammar, which are passed on to parsers for expressions from the next line of the grammar.

The second lowest-precedence connective is the implication symbol ⇒, which is a binary infix, right-associative operator. The expression $a \Rightarrow b \Rightarrow c$ gives this parse tree:



The final output from a parser (P_E) for a string from “E” depends whether the string contains the operator from that line or not. If it does, the right sub-expression is parsed as an E_1, and the left sub-expression as an E; otherwise the whole expression is regarded as an E_0 and passed unchanged to the E_0 parser. The final output from an string *e*, a member of E can be expressed as

$P_E(e ⁀ “⇔” ⁀ e_1) = P_E(e) ⁀ P_{E1}(e_1) ⁀ “⇔”$ or $PE(e) = P_{E1}(e)$

. . . where P_E and P_{E1} are parsing operations for strings *e* and e_1 which are members of E and E_1. This maintains the form shown by Lynas and Stoddart (2008). This can be expressed as an intermediate representation from the first pass of parsing; for example, $P_E(e ⁀ “⇔” ⁀ e_1)$ gives

“" P_E(e)" " T" " P_{E1}(e_1)" " U" ⇔_”

This is the result of parsing the left string *e* followed by its type “T” followed by the the result of parsing the right string e_0 and its type “U” followed by the tagged operator ⇔_. The output is a string including the "straight" quote marks, which can be executed as FORTH code. “T” and “U” must both be Boolean type for the ⇔ operator.

The maplet operator ↦ creates ordered pairs and accepts operands of any type. As an example, “f ↦ b”, where the types of f and b are foo and bar respectively would be parsed to give this intermediate representation: “" f" " foo" " b" " bar" ↦_”. This can itself be executed as FORTH code to leave “f b ↦” as the postfix form of the expression, and its type, “foo bar PROD”, on the stack.

The “lsplit” and “rsplit” Operations and the First Stage of Lexical Analysis.

The first stage parsing is to analyse the input string representing the expression into its operator and two arguments or operands, left and right. This is done with operations called lsplit and rsplit; lsplit explores the expression from right to left, looking for left-associative operators, and rsplit explores from left to right, to find a right-associative operator. Each requires two values on the stack, the text to be analysed and a sequence containing the operators sought at the present level of precedence.

The lsplit operation is shown here. It uses the following values internally as local variables: the cardinality of the sequence, the end of the string, a loop count index, and a value as a place-holder for the operator string. These are in addition to the two arguments. It also uses the prefix? function, which tests whether a string is a prefix of another, endaz which finds the end of a 0-terminated string, myazlength which determines its length, and the bracket-avoider-for-lsplit function, which skips backwards over any text in brackets.

The essence of the operation is two nested loops. The outer loop counts backwards from the end of the text (skipping any text in brackets) and the inner loop compares each value in the sequence in turn to see whether the operator sought has been found; starting from the current value of “end” this would appear to be a prefix to the string.

The original value of “op” is 0 (*null*); whenever a matching prefix is found, “op” is replaced by that value, and both loops terminate. The left sub-expression can be terminated simply by placing a null (\0) character in the location of the “end” pointer with the C! instruction, and the right sub-expression by adding the length of the “op” string to the current “end” pointer.

```
: lsplit ( s seq -- s1 s2 op )
    ( s is a string in the form "a + b" and seq is a sequence of strings e.g. )
    ( ["+", "-"], s1 is the string as far as the operator, s2 after it, and op )
    ( is the operator. If op is null = 0, the value below it must be regarded as a )
    ( nonsense value and deleted from the stack. )
    ( This function skips over any text in “”, (), [] and {} )
( string seq already on stack ) 0 0 0 0 ( 6 values now on stack )
(: string seq end op count size :)
string endaz to end ( One of the 0s gone )
seq CARD to size    ( Second 0 gone      )
BEGIN
    end string >= op 0= AND
WHILE
    0 to count
    end bracket-avoider-for-lsplit to end ( Skip text in brackets etc. )
    BEGIN
        size count > op 0= AND ( Not reached start of string, nor found op )
    WHILE
        count 1+ to count      ( Go through potential operators )
        seq count APPLY end prefix?
        IF
            seq count APPLY to op
        THEN
    REPEAT
    end 1- to end    ( Count backwards to start of string )
REPEAT
op
IF
    end 1+ to end    ( Terminate string at op )
    0 end C!
    end op myazlength + to end ( Move forward length of op )
THEN
string end op
;
```

Note that if no “op” string is found, a 0 will be left on the stack. In that case, the “left” value will be the original text unchanged, and the “right” value must be discarded as a “nonsense” value. The rsplit operation, which is used for right-associative operators, is very similar but slightly simpler, because there is no need to seek the end of the string before starting the two loops.

In this example, passing “*f* ⇔ *b*” and the sequence STRING [" ⇔" ,] to lsplit places the following three string values on the stack:

f   *b*   ⇔

. . . whereas passing “*f* * *b*” and the sequence STRING [" +" , " -" ,] would produce the result

*f * b*   *???*   null

. . . i.e. the original input unchanged, an undefined or nonsense value, and null (= \0).

Passing “1+2-3-4” and the sequence STRING [" +" , " -" ,] however, splits the input at the second minus, leaving this result on the stack:

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

In all cases where there are several left-associative operators which can be found at the same precedence, the input is split at the rightmost symbol, meaning the left string is parsed recursively; similarly the right string (second value on the stack) is parsed recursively if there are several right-associative operators found with rsplit.

The Parser Operations

The parser operations are similar to one another; an example is shown.

```
: Pequiv ( s -- s1 )
(: text :) text equivalence lsplit
VALUE left VALUE right VALUE op
op
IF
    left RECURSE right Pimplies op bar-line AZ^ AZ^ AZ^
ELSE
    left Pimplies
THEN
;
```

. . . where “equivalence” is defined as STRING [" <=>" , " ⇔" ,] (allowing “<=>” as a synonym for “⇔”) and bar-line a string containing “_\n”[1]. The parser splits the “text” into three parts, “left” “right” and “op”. Passing “f ↦ b” as input results in “f” being parsed recursively, “b” being passed to a P_{implies} parser, and the results being catenated (using the AZ^ operation) with the operators and “_\n”. If no operator is found, only the “left” sub-expression represents a real value, which is passed to the next parser in the sequence. In the case of a right-associative operator, rsplit is used, and the recursion is applied to the “right” sub-expression Eventually the recursion reaches a stage where the expression can be subdivided no more; then the parsers return the text and its type with the necessary quotes and spaces included. For example “f” would be parsed to “" f" " foo"” and the type is sought in a lookup table. This has the corollary that variables must be declared in advance, so their type can be entered into the lookup table. The parser above requires one string as input; for “f ⇔ b”, it returns “" f" " foo" " b" " bar" ⇔_” onto the stack.

It is possible to create versions of the parsers which call the operations directly, e.g. with the ⇔_ instruction, and these will call both phases of compilation together.

The Tagged Operators as Operations

The ↦_ operation as a Simple Example

The output from the parsing operations can be passed to a FORTH virtual machine; the output “" f" " foo" " b" " bar" ↦_” puts the four values f foo b bar onto the stack and calls the ↦_ operation. The ↦_ operation has the simplest typing of any; it accepts any type except *null*, and is shown below:

1 Here, \n is the line-feed character 0x10.

```
: ↦_ ( s1 s2 s3 s4 -- ss1 ss2 )
    (: l-value l-type r-value r-type :)
    " ↦" l-type r-type check-types-not-null
    l-value sspace AZ^ r-value AZ^ "  ↦" AZ^ l-type sspace AZ^ r-type AZ^
           "  PROD" AZ^
;
```

The check-types-not-null operation simply checks that null has not been passed as a type because ↦ has no restriction about its types of operand. Then, ↦_ takes the two operand values “l-value” and “r-value”, catenating them with spaces and ↦. Then is catenates the two type values “l-type” and “r-type” with “PROD” and the appropriate spaces, leaving those two values on the stack. The equivalence operator ⇔ causes the ⇔_ operation to be invoked; this uses the check-types-for-booleans operation, which tests that both operands have a Boolean type. Almost every tagged operation is similar to ↦_, except those for unary operators which only take one value and one type, and those where the output may differ with the type of input.

Every “check” operation emits an error message and then calls ABORT; this means only one error message is displayed, even if there are several errors.

The +_ Operation as a More Complex Example

As described by Lynas and Stoddart (2008), arithmetical operations may take different input types and produce different output. In a simpler version which accepts only integer numbers, the +_ operation is very similar to ↦_. In the version which accepts floating-point numbers as well, it may be necessary to change the type of the argument with the S>F operator, and prefix the + with an F. The complex version follows:

```
: +_ ( s1 s2 s3 s4 -- ss1 ss2 )
    (: l-value l-type r-value r-type :)
    "  +" VALUE op op l-type r-type check-types-for-arithmetic
    l-type " FLOAT" string-eq r-type " FLOAT" string-eq OR
    ( Either or both is float )
    IF
        "  F+" to op
        l-type " INT" string-eq
        IF  ( Add S>F as appropriate )
            l-value "  S>F" AZ^ to l-value
        ELSE
            r-type " INT" string-eq
            IF
                r-value "  S>F" AZ^ to r-value
            THEN
        THEN
        " FLOAT" to l-type
    THEN
    l-value sspace AZ^ r-value AZ^ op AZ^ l-type
;
```

The check-types-for-arithmetic operation confirms that both types are “INT” or “FLOAT”. On checking whether either operand is a “FLOAT”, the operator is changed to “ F+”, and whichever of the operands is an INT has “ S>F” appended. Also the type to return is changed to “FLOAT”. This operation will take " 1" " INT" " 1.34" " FLOAT" +_ and return “ 1 S>F 1.34 F+” and the type “FLOAT”.

Sets and Types

A set expression such as {1, 2, 3} can be implemented in FORTH by executing the code

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

where each token is a FORTH operation. The `INT` provides the type of the set, by placing a pointer to an empty set of `INT`s on the stack, which is opened by the { operation. Each number is placed on the stack in turn, and added to the current set by the comma operator, and the } operation completes the set construction, leaving a new reference to the whole set on the stack.

Sets may contain individual values, or pairs, or sets. {{1, 2}, {3}} is an example of a set of sets, which is represented in FORTH as

```
INT SET { INT { 1 , 2 , } , INT { 3 , } , }
```

and the following is an example of a set of pairs (called a "relation") from Strings to integers:

```
{"Bill" ↦ 2673, "Campbell" ↦ 2680, "Dave" ↦ 2680}
```

The set of its left-hand elements ({"Bill", "Campbell", "Dave"} is called its Domain, and the set of its right-hand elements ({2673, 2680}) is its Range. It can be translated into FORTH as

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

It is possible to retrieve a value from the range of that relation, which we are calling "r", in infix notation by writing r("Bill") which translates to FORTH postfix notation as

```
r " Bill" APPLY
```

If that relation is inverted and the value 2680 applied, there are two possible results, "Campbell" and "Dave"; the choice can be made non-deterministically and on a reversible FORTH implementation the choice can be altered on backtracking.

Sequences can be represented similarly, but using square brackets [...] instead of curly braces {...}; in terms of sets, a sequence is regarded as a relation from integers to another type, T (INT T PROD). In the case of a sequence, its domain is equal to the set of consecutive integers up to its cardinality *c*, expressed as 1 … *c*[2]. It is possible to have relations from integers to T whose domain does not consist of consecutive numbers, and which do not represent sequences. All the set operations can be applied to sequences. As an example where this might be useful, one can compare two sequences of the same type, *s* and *t*; *s* is a prefix of *t*, if *s* is a subset of *t* and the union of *s* and *t* equals *t* ($s \subseteq t \wedge (s \cup t = t)$).

Operations to Create a Set

The operations to create a set are used in the following order:

`{_`	Puts "{" on the stack (twice) and 0 (= null) because the type is not yet known.
`1`	Puts the value 1 on the stack, and the string "INT" being its type.
`,_`	Catenates the type { 1 and , to leave " INT { 1 , ", and changes the type on the stack to "INT".
`2`	Puts the value 2 and its type "INT" onto the stack.
`,_`	Checks the "INT" is the correct type and catenates the previous value with 2 and , to produce " { 1 , 2 , ".
`3`	Puts the value 3 and its type " INT" onto the stack.
`}_`	Checks the remaining "{" matches "}", and that the type "INT" is the same as before, and catenates 3 comma and } to leave " INT { 1 , 2 , 3 , }" and the type "INT SET" on the stack. This resultant code can be executed as a FORTH instruction.

Since the type of variable is checked, we restrict sets to homogeneous sets, i.e. those which only contain one kind of element.

2 We are using 1, not 0, for the number of the first element in the sequence.

Other Set Operations and Typing

In the case of set union and intersection and difference, the types must be checked that the two operands are the same sort of set, i.e. each shows its type as “T POW”. Relational overriding, using the ⊕ operator replaces values in a relation on the left by values in the same domain in the right operand; overriding of s by t is written as $s \oplus t$ and can be implemented in FORTH as “s t OVERRIDE”. So each operand must be a relation of the same type, e.g. “S T PROD”.

The typing for other operations can be more difficult. For example the domain restriction operation, using the ◁ operator requires the left operand be a set of type “T” and the right operand be a relation from “T” to a type “U”. So R ◁ S returns a relation from S of all those elements whose domain is included in the set R, and if the type of R is “T POW”, the type of S must be “T U PROD POW”.

Results

We demonstrate a compiler for expressions which can be simply constructed with a recursive architecture. Each component is relatively simple, the most complicated one being a version of rl-lex which can distinguish “-” as a binary or infix operator, for subtraction, from “-” as a unary prefix operator, which occupies 41 lines when comments are excluded. The operation of the compiler can easily be seen by running a FORTH virtual machine. The expression can be fed onto the stack, followed by the name of the operation to compile it, and the output (highlighted in pale grey) can be seen with the .AZ command; feeding this output back to FORTH e.g. with “copy-and-paste” initiates the second pass of compilation, which produces the type “INT” and the postfix expression 1 2 3 * + 4 /, which evaluates to 2.

```
" 1 + 2 * 3 / 4" Pexpression .AZ " 1" " INT" " 2" " INT" " 3" " INT" *_
" 4" " INT" /_
+_
ok
" 1" " INT" " 2" " INT" " 3" " INT" *_ ok....
+_ ok..
" 4" " INT" /_ ok..
.AZ INTok.
.AZ 1 2 3 * + 4 /ok
1 2 3 * + 4 / . 2 ok
```

More complicated expressions can also be analysed. This set expression produces the intermediate result highlighted in grey, and the second output “INT POW” and “INT { 1 , 2 , 3 , }”:

```
" {1, 2, 3}" Pexpression .AZ {_
" 1" " INT" ,_
" 2" " INT" ,_
" 3" " INT" }_
{_ ok...
" 1" " INT" ,_ ok...
" 2" " INT" ,_ ok...
" 3" " INT" }_ ok..
.AZ INT POWok.
.AZ INT { 1 , 2 , 3 , }ok
INT { 1 , 2 , 3 , } .SET {1,2,3}ok
```

Similarly, nested and bracketed expressions can be compiled, for example:

```
" (1 + 2) * 3 / 4" Pexpression .AZ " 1" " INT" " 2" " INT" +_
" 3" " INT" *_
" 4" " INT" /_
ok
" 1" " INT" " 2" " INT" +_ ok..
" 3" " INT" *_ ok..
" 4" " INT" /_ ok..
.AZ INTok.
.AZ 1 2 + 3 * 4 /ok
1 2 + 3 * 4 / . 2 ok
```

This expression also evaluates to 2.

Discussion

"The primary criterion for a parsing algorithm is that it must be efficient." (Bennett 1996, page 80).

Unfortunately, for each stage, it is necessary to traverse the String representing the expression at each of these stages. As described humorously by Spolsky (2001), traversing a null-terminated (or ASCIIZ) String takes a time proportional to the length of the String. Also, the number of traversals is roughly proportional to the number of operators, which again depends on the String's length. Our compiler must therefore run in quadratic time (O(n^2) complexity). So its utility for compiling long programs must be limited, but performance will be better if a large program can be divided into small functions or operations. It may be possible to enhance the FORTH RVM by adding persistent memory, allowing the output from the first pass of compilation to be retained for use by the second pass.

We have, however, demonstrated a compiler for expressions which the writer and reader can simply understand, and which can easily be expanded to complicated expressions. This compiler has the unusual feature that it recursively seeks operators or connectives, rather than going through the text from left to right. It is quite easy to examine and interpret the code, which makes this technique a potential teaching and research tool. Compilers created with automated tools, e.g. yacc (Johnson, 1975) and lex (Lesk 1975) create much code which is difficult to understand at first reading, and does not lend itself to didactic use.

This compiler is suitable for expansion. For example, it would be easy to add Boolean expressions, including conjunction disjunction and implications. It would also be possible to add more operators of different precedences to the grammar, and intersperse parsers to accommodate those operators.

The grammar, as we have written it, easily permits arbitrary recursion both to left and right. This can be seen in the parse trees, and can be seen where the keyword RECURSE appears in the parsers.

Further Work

We plan to add control structures, to implement assignments, loops and selection (if-then-else blocks). These will necessitate Boolean values to control their flow. For assignment, it will be necessary to declare variables before use, so a technique to add variables and their types to a lookup table is needed. Since Böhm and Jacopini (1966) demonstrated that programs of arbitrary complexity can be assembled from elements of sequence, selection and iteration, these control structures are sufficient to build a language capable of any operations. As well as these, additional control structures, including choice and guards, can take advantage of the reversible virtual machine described by Stoddart Lynas and Zeyda (2010).

We shall need a parsing method for Strings, using the opportunity for nesting "smart" quotes provided by Unicode support. It will be necessary for such quotes to be nested in pairs; this following example, which quotes Milne (1926) shows such nesting:

" "In Which Pooh Goes Visiting and gets Stuck in a Tight Place" by A A Milne
includes the following:
"Pooh . . . said that he must be going on. "Must you?" said Rabbit politely.
"Well," said Pooh, "I could stay a little longer if . . . " " "

It is easy to count from end to end of such a String, until both opening and closing quotes have been identified.

We hope to implement higher order functions, including λ expressions; these may require both a definition of the function and insertion of its input and output types into a lookup table. Some functions may be implementable as sets of ordered pairs from input to output.

It may also be possible, after a full language is written, to bootstrap the compiler by rewriting it in the new language.

Conclusion

We have demonstrated a two-pass compiler for a rich expression language; one can execute the two passes separately or together. It supports strongly-typed sets and sequences, following the conventions of B. This compiler is made up of small, mostly simple modules which are assembled to form a parse tree, and uses operations called after the operators, tagged with a _ character, to complete the compilation.

Since parse trees have a structure very similar to the postfix notation used in FORTH, it is simple to convert a parse tree to FORTH code which can be executed directly.

References

Abrial J-R 1996. *the B Book* Cambridge: Cambridge University Press.

Bennett J P 1996. *Introduction to Compiling Techniques. A First course using ANSI C, lex and yacc* 2/e (the McGraw-Hill International Series in Software Engineering) Maidenhead: McGraw-Hill

Hehner Eric C R 1981. *Bunch Theory: a Simple Set Theory for Computer Science.* Information Processing Letters **12**(1): 26-30

Böhm C, Jacopini G. *Flow diagrams, Turing machines and languages with only two formation rules*, Communications of the Association for Computing Machinery **9(5)**: 366-371

Gosling J, Joy B, Steele G and Bracha G 2005, *The Java Language Specification* 3/e *(Java Series)* Upper Saddle River NJ: Prentice-Hall, also available at http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html Accessed 15th September 2010

Hehner Eric C R 1993. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Berlin: Springer-Verlag. A more recent edition is available at http://www.cs.toronto.edu/~hehner/aPToP/ accessed 17th June 2010.

Johnson S C 1975. *Yacc—Yet Another Compiler-Compiler.* Comp. Sci. Tech. Rep. No 32, Murray Hill NJ: AT&T Bell Laboratories, July 1975.

Lesk M E 1975. *Lex—A Lexical Analyzer Generator.* Comp. Sci. Tech. Rep. No 39, Murray Hill NJ: AT&T Bell Laboratories, October 1975.

Levine J R, Mason T, Brown D 1992. *lex & yacc* 2/e. Sebastopol CA: O'Reilly & Associates, Inc.

Lynas A R, Stoddart W J 2008. Using Forth in a Concept-Oriented Computer Language Course, in ed. *A* Ertl, Proceedings of the 25th EuroForth Conference, Wien, *pages* 7-19. Available at http://www.complang.tuwien.ac.at/anton/euroforth/ef08/papers/proceedings.pdf, accessed 15th June 2010.

Milne A A, 1926. *Winnie the Pooh* London: Methuen & Co Ltd., and other publishers.

Ritchie C, Stoddart W J 2009. *Formulating Type Tagged Parse Trees as Forth Programs*. in *ed.* A Ertl, Proceedings of the 25th EuroForth Conference, Exeter, *pages* 11-22. Available at http://www.complang.tuwien.ac.at/anton/euroforth/ef09/papers/proceedings.pdf, accessed 16th June 2010.

Spolsky J 2001, *Back to Basics* http://www.joelonsoftware.com/articles/fog0000000319.html accessed 15th June 2010, also available as *Back to Basics, in* Spolsky J 2004, *Joel on Software.* Berkeley CA: Apress, *pages* 5-15

Stoddart W J, Lynas A R, Zeyda F 2010. *A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages.* Electronic Notes in Theoretical Computer Science **253(6)**: 33–56, also available at http://www.www.elsevier.com/locate/entcs accessed 17th June 2010

W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In *ed.* A Ertl, 18th EuroForth Conference Proceedings, 2002. On-line proceedings, available at http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/ accessed 21st June 2010.

Appendix

The expression grammar is given here, in bunch notation (Hehner, 1981 and Hehner, 1993). The comma is an operator representing bunch union, including all members of both bunches which are its operands. As an example, if “A” means the bunch of all strings representing arithmetic expressions and “A_1” means the bunch of all strings representing terms in arithmetic, and “+” means joining or catenating two strings around a + sign (with or without spaces), etc., we can regard the top line in an arithmetic grammar as

A = A “+” A_1, A “–” A_1, A_1

. . . i.e. the bunch of strings constituting arithmetic expressions followed by + followed by an arithmetic term, AND arithmetic expressions followed by – followed by a term, AND arithmetic terms. There may be white-space between the sub-expressions and the operator. Another definition of “arithmetic term” or A_1 is an arithmetical expression which does not contain + or – as its lowest-precedence operator.

The following abbreviations are used:

- L A comma separated list of expressions. In this case, the underlined comma $\underline{,}$ is used to represent a literal comma rather than bunch union. L = L$\underline{,}$ E, L
- E An expression
- B A boolean expression
- P An expression representing a pair
- S An expression representing a set
- W An expression representing a string or a set
- A An arithmetic expression (expression representing a number)
- N Numeric literal
- $ String literal
- I An identifier

Details of the grammar of some non-terminals, e.g. string and numeric literals, are omitted below.

E	=	B “⇔” B_1 ,	E_1			
E_1	=	B_2 “⇒” B_1 ,	E_2			
E_2	=	B_2 “∧” B_3 ,	B_2 “ ∨” B_3 ,	E_3		
E_3	=	“¬” B_3 ,	E_4			
E_4	=	E “∈” S ,	E “∉ ” S ,	E_4 “=” E_5 ,	E_4 “≠” E_5	
E_5	=	A “<” A ,	A “≤” A ,	A “>” A ,	A “≥” A ,	E_6
E_6	=	S “⊆” S ,	S “⊈” S ,	S “⊂” S ,	S “⊄” S ,	E_7
E_7	=	E_7 “↦” E_8 ,	E_8			
E_8	=	E_8 “\” E_9 ,	E_8 “∪” E_9 ,	E_8 “∩” E_9 ,	E_8 “⊕” E_9 ,	E_9
E_9	=	S_3 “◁” S_2 ,	S_3 “◁-” S_2 ,	E_{10}		
E_{10}	=	S_2 “ ← ” E ,	W “⁀” W_1 ,	S2 “▷” S_3 ,	S_2 “-▷” S_3 ,	S_2 “↓” A ,
			S_2 “↑” A ,	E_{11}		
E_{11}	=	A “+” A_1 ,	A “-” A_1 ,	E_{12}		
E_{12}	=	A_1 “*” A_2 ,	A_1 “/” A_2 ,	E_{13}		
E_{13}	=	“-” A_3 ,	E_{14}			
E_{14}	=	N ,	$,	I ,	F ,	“{” L “}” ,
			“[” L “]” ,	“(” E “)” ,	λ	

B	=	B “⇔” B_1 ,	B_1				
B_1	=	B_2 “⇒” B_1 ,	B_2				
B_2	=	B_2 “∧” B_3 ,	B_2 “ ∨” B_3 ,	B_3			
B_3	=	“¬” B_3 ,	B_4				
B_4	=	E “∈” S ,	E “∉ ” S ,	E_4 “=” E_4 ,	E_4 “≠” E_4		
B_5	=	A “<” A ,	A “>” A ,	A “≥” A ,	A “≤” A,		B_6
B_6	=	S “⊆” S ,	S “⊈” S ,	S “⊂” S ,	S “⊄” S ,		B_7
B_7	=	“true” ,	“false” ,	I ,	I “(” L “)” ,		“(” B “)”
S	=	S “\” S_1 ,	S “∪” S_1 ,	S “∩” S_1 ,	S “⊕” S_1 ,	S_1	
S_1	=	S_2 “⊲” S_1 ,	S_2 “⊲-” S_1 ,	S_2			
S_2	=	S_2 “ ← ” E ,	S_2 “⌢” S_3 ,	S_2 “⊳” S_3 ,	S_2 “⊳-” S_3 ,	S_1 “↑” A ,	
			S_2 “↓” A ,	S_3			
S_3	=	I ,	F ,	“{” L “}” ,	“[” L “]” ,	“(” S “)” ,	λ
W	=	S_1 “ ← ” E ,	W “⌢”m W_1 ,	S_2 “⊳” S_3 ,	S_2 “⊳-” S_3 ,	S_1 “↑” A ,	
			S_2 “↓” A ,	W_1			
W_1	=	I ,	F ,	“{” L “}” ,	“[” L “]” ,	“(” W “)” ,	λ
A	=	A “+” A_1 ,	A “-” A_1 ,	A_1			
A_1	=	A_1 “*” A_2 ,	A_1 “/” A_2 ,	A_2			
A_2	=	“-” A_2 ,	A_3				
A_3	=	I ,	F ,	N ,	“(” A “)”		
λ	=	$\underline{\lambda}$[3] I • E					

3 Here the underlined $\underline{\lambda}$ is used to represent a literal λ in the text, rather than a λ expression.