Using Forth in a Concept-Oriented Computer Language Course

Angel Robert Lynas and Bill Stoddart
University of Teesside

September 8, 2008

Abstract

We describe a way of teaching fundamentals of Language Systems (for second-year Computing students), without having to compromise the use of a simple grammar owing to hardware limitations which need no longer apply in this setting.We adopt a top-down approach, reading from right-to-left and splitting the input string on the rightmost operator appropriate to that level. The target platform is Reversible Virtual Machine (RVM) Forth, so a postfix translation is the aim. We introduce a basic arithmetic grammar and expand it during the course to allow unary minus, floating point and function application; this shows how type information can be generated in one pass and resolved in a second, via an internal intermediate code. Each version of the grammar has its productions mapped onto a system of equations which serve as the specification for the implementation functions.

1 Introduction

In teaching Language Systems for Computer Science, the conceptual simplicities can often be obscured by the compromises made to accommodate purely historical restrictions on software writing for far more limited computers than those available today. These compromises take the form, for instance, of extra complications in the simple grammars, and trying to do several jobs in one pass of the compiler.

Accepting that we can now directly implement, at least in a pedagogical setting, a more straightforward approach less hampered by traditional constraints, can lead to a clarification of the conceptual issues involved in top-down parsing.

We used a basic grammar for infix arithmetical expressions, and used the conversion to postfix as a convenient route for exploring grammar analysis. An expansion to include floating point numbers and other facilities proved to be a fruitful method for introducing the usage of type information and two-pass compiling with intermediate code to hide the use of meta-information.

Following some definitions, we describe in Section 2 the overall approach taken, then in Section 3 the basic grammar presented to the students. In Section 4 we examine the extensions to this grammar, the actions of the new compiler's first pass and those of its second. We conclude with a brief look at a possible future application of these ideas for subsequent refinements of the course.

1.1 Some Definitions

In the following, we use some fundamental terminology from the study of grammars:

Terminals These are the strings which are the tokens of the language and appear in generated expressions.

Non-Terminals These are symbols which do not appear in the output, but stand for classes of terminals, or combinations of those classes.

Productions Specifies one way in which a non-terminal can be expanded to a sequence of other non-terminals and/or terminals, eventually generating strings composed entirely of terminals.

2 Higher-Level approach

The usual approach in teaching about language systems has been to adopt a relentlessly left-to-right method, compatible with historical restraints on memory, storage and processing capacity. Thus in the interests of efficiency, perhaps one token is available for "look-ahead" while the current token is being processed (look-ahead by default referring to the token on the right). See for instance the popular compiler texts [1] and [2].

Using these techniques, everything is done in a single pass, including storage and checking of type information (if applicable) and resolution of conditional structure — though we don't consider the latter here.

Certain things must be sacrificed to this methodology, the first casualty being simplicity of grammar. A grammar for basic arithmetic could contain a production (we define <expression> and <term> more fully later)

<expression>::=<expression> + <term>

meaning that one way of constructing an expression is by combining an expression and a term with a "+" sign in between. This we generally abbreviate to

$$E ::= E + T$$

This, however, is not really suitable for left-right parsing owing to the left recursion; that is, an E could expand to $E + T + T + \cdots + T$, so reliably telling where the first <expression> ends is problematical. The order of E and T specifies left-associativity for the + operator and others, ensuring that (for instance) $a - b - c$ is parsed as $(a - b) - c$, rather than $a - (b - c)$ which gives a different result.

In order to remove this left-recursion, such constructs are usually redefined using dashed letters to denote "the rest of the expression", now looking like this

$$E ::= TE'$$
$$E' ::= +TE' \mid \epsilon$$

The symbol ϵ or "*null*" stands for the empty string. The decomposition of the expression E can now be approached unambiguously from the left, as the term (processed by a similarly expanded rule) is recognised and the "rest of the expression" E' is passed downward for further processing. But this represents a considerable loss in simplicity compared to the first grammar.

What method, then, could be used to parse productions like $E \rightarrow E + T$ as is? Clearly a right to left approach would be more apropriate here, splitting the expression at the rightmost top-level "+" encountered, where the remaining expression on the left is dealt with by a recursive call to the expression parser, and the term is dealt with by the term parser. Terms are split on "*" or "/" if any occur, again the rightmost, for example $T = T * F$, where F is a factor with no top-level operations. We can now fill in the rest of the grammar and formalise the operation of a recursive parser.

3 A Simple Arithmetic Grammar

The initial grammar developed for the students, which allows for unsigned integers, identifiers, and bracketed expressions, is as follows. The order of the non-terminal expansions reflects the precedence order of the operators; lower precedence operators are scanned for first, and the non-terminal split if applicable. The higher precedence operators appear closest to the terminals in the postfix output and are thus executed first. The vertical bars on the left-hand side indicate alternative productions for the same non-terminal.

Non-Terminals:

- E is an Expression; these can contain a plus or minus sign at the top level (i.e. not within any brackets).
- T is a Term, which contains no pluses or minuses at its top level, but can contain "*" or "/" for multiply or divide.
- F is a Factor, containing no top-level operations, but can expand to an unsigned number U, or an identifier I, or a bracketed expression (E), the contents of which are recursively expanded as an expression. We do not define U or I further at present; in any case U will be later replaced by a non-terminal which accommodates both integer and floating point numbers.

$$E ::= E + T \mid E - T \mid T$$
$$T ::= T * F \mid T/F \mid F$$
$$F ::= U \mid I \mid (E)$$

The compilation to postfix is described by a set of equations involving mutually recursive functions which act on appropriate strings of each non-terminal class. The right-hand sides of these equations show the output from a given string; at the top levels, this will involve invoking other functions on part of the string — possibly including the function itself recursively. In the following, lower-case e, t, f, u and i represent strings belonging to the nonterminals E, T, F, U and I respectively; that is, e is a particular expression, t is a particular term, and so on. We have the functions

P_E takes a string e and returns the translation of that string in postfix.

P_T takes a string t and returns the translation of that string in postfix.

P_F takes a string f and returns the translation of that string in postfix.

If no operators are found by P_E or P_T, they pass the entire string down to the next function. The symbol "$\frown$" is used for string concatenation. The mutually recursive equations which define the compilation are

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"\ +"} \tag{3.1}$$

$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"\ -"}$$

$$P_E(t) = P_T(t)$$

$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"\ *"} \tag{3.2}$$

$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"\ /"}$$

$$P_T(f) = P_F(f)$$

$$P_F(u) = u$$

$$P_F(i) = i$$

$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

At the top two levels — expression and term — the parsing is right to left; specifically, a Forth operation LSPLIT is used to search for a symbol in $\{+, -\}$ or $\{*, /\}$ at the top level. The stack parameters are the string to be searched and a sequence of strings which are the tokens to search for. An example call follows, but a brief explanation is needed first.

A string in RVM-Forth can be an "ASCII-Zero" or AZ string, which is terminated by an ASCII null (in other words, a 0 after the final character, like strings in C and some other languages). They are created with the $\boxed{\texttt{"}}$ word, terminated by another quote. Also, sequences are created with the syntax

```
<type> [ <element1> , <element2> , ]  ( and so on)
```

where the comma word allocates storage for each element. The type can be simple (integer, string) or composite, involving pairs and nested sets or other sequences.

An example call to LSPLIT using a string EXPR1, then, is

```
EXPR1 STRING [ " +" , " -" , ] LSPLIT
```

This would be an call to process EXPR1 using the symbols $\{+, -\}$.

Should LSPLIT encounter a right bracket ")", this means that a lower-level expression is included in the top-level one, which expression will be dealt with by a later recursive call to P_E. So LSPLIT will not continue its search for operators until it finds the matching left bracket; it will keep track of the level, incrementing for any right brackets and decrementing for matching left ones, until its own level zero is reached again.

On encountering a symbol/token in its search set, the operation splits the input string into a left part, the symbol string, and a right part. As can be seen from equations (3.1) and (3.2) above, the first part is sent recursively to P_E or P_T as appropriate, the second part to P_T or P_F, while the operation token is output last of all, so that it occurs *after* the outputs from parsing the rest of the string. If no search symbol is found, the entire string is passed down to the next parsing level.

When the factor level is reached, subsequent processing of U and I can be done left to right, as no left recursion occurs in these definitions. A factor of the form "(E)" merely has its brackets stripped and the contents sent back to the top-level function P_E. We include the code for P_T (called PT), which parses terms into terms and factors, as figure 1 below. The local variable syntax uses the words (: and :) to delineate the declarations, with values being taken from the stack.

```
: PT ( az1 -- az2, parse a term, leaving az2 the postfix
       translation of the term az1 )
   (: VALUE e :)
    e STRING [ " *" , " /" , ] LSPLIT
    VALUE BEFORE VALUE AFTER VALUE OP-STRING
    OP-STRING NULL =
    IF
      BEFORE PF      ( No op found, so e was a factor )
    ELSE
      BEFORE RECURSE  AFTER PF  ^  OP-STRING  ^
    THEN
  1LEAVE ;
```

Figure 1: RVM-Forth code for P_T.

The general technique is known as Recursive Descent Parsing, and is well-known, though we do not know of its having been implemented in this bidirectional way. The usual categories of LL(k) or LR(k) do not apply, strictly speaking, as they denote solely unidirectional parsing; for instance the Earley recogniser [3], based on Knuth's LR(k) algorithm. Our technique is adaptable insofar as right-associative operators can and will be accommodated by a sort of mirror image of LSPLIT which would read from left to right.

The parsing can be illustrated with a couple of examples, which can either be represented as parts of trees (useful for the students initially), or in a linear fashion which follows the equations closely. Given the input string "$x * x - 1 - (x - 1) * (x + 1)$", for instance, the first parsing step is shown in figure 2.

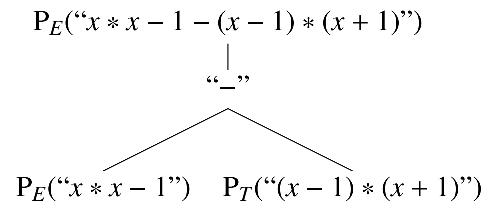
P$_E$("$x * x - 1 - (x - 1) * (x + 1)$")
"–"
P$_E$("$x * x - 1$")
P$_T$("$(x - 1) * (x + 1)$")

Figure 2: Splitting at the top level.

The left-hand side of this is then the input to a recursive call to P_E; the rest of this is shown (as far as the Factor level) in figure 3.

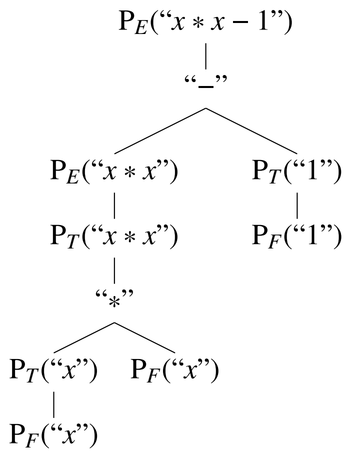

Figure 3: Splitting the left-hand half down to Factor level.

In linear style, the first part of this second tree would appear as

$$P_E("x * x - 1") = P_E("x * x") ^\frown P_T("1") ^\frown " - "$$

which we can see follows the pattern of equation 3.1.

As regards performance, we have not yet undertaken any exhaustive time complexity analyses. Some empirical run-throughs indicate that the basic technique is probably $O(n^2)$ for a suitable grammar; the limitations with respect to which grammars can be handled require further investigation, however.

4 Floating Point Extension

We now consider extending the grammar to include unary minus, function application and floating point capability. The inclusion of mixed-mode arithmetic requires the simple compiler to become a two-pass compiler, whereby the first pass generates intermediate code containing type information, as we now have integer and floating point to deal with. These require different operations for arithmetic and printing, and sometimes conversion is required from integer to floating point.

Unary minus is dealt with by converting the minus sign into a tilde "$\sim$", as the compiler uses this internally; it's converted back for the Forth-readable output, and allows multiple minuses in a row which are converted consistently. The details are not examined further here. Function application is of the form "<identifier>(<arg-list>)", with the arguments comma-separated.

The basic grammar is extended as follows, with the extra non-terminals

- F_0 This is simply an unsigned factor.
- N This replaces U, and is a general number (integer or float), parsed by the floating point state machine to be described later.
- L This is a comma-separated list of arguments to a function.

$$\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T/F \mid F \\
F &::= F_0 \mid -F \\
F_0 &::= N \mid I \mid (E) \mid I(L) \\
L &::= L, E
\end{aligned}$$

The list of arguments L is parsed right to left, in a similar way to E and T, but split by `LSPLIT` on a comma.

4.1 First Pass (intermediate code)

The generating functions for the upper levels are similar for those in the previous section, but instead of including a straightforward plus or multiply sign in the output, they include the string for an internal operation which the second pass of the compiler will use to resolve the types and assign the correct integer or floating point version of the Forth operation in the final output; these internal operations all have an underscore suffix. The first few equations are thus similar to those near (3.1).

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"~+\_"}$$

$$P_E(t) = P_T(t)$$

$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{"~-\_"}$$

$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"~*\_"}$$

$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{"~/\_"}$$

$$P_T(f) = P_F(f)$$

$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

Below this, terminals begin to be output in the intermediate code, however these need to be strings rather than literals; spaces are added where required to ensure the final output is consistently space-separated. In the equations below, the symbol $\boxed{\texttt{\textbackslash"}}$ denotes a literal embedded quote, and ϕ is a function identifier (not a numerical variable). We also have the additional parameters

- n Any integer number string.
- i Any integer identifier (variable) string — note, no longer a general identifier
- l Any list of arguments.
- r Any floating point number string.
- r' Any Forth equivalent[1] to r.
- x Any real identifier string (floating point variable).

The identifiers are stored in symbol tables which specify some predefined families of identifier strings; this obviates any immediate need for typed declarations, however this is an area which will be examined at a later stage. Thus identifiers beginning with the letters `i-n`, in either case, are integer while all others are treated as floating point.

We then introduce the additional parsing functions

- P_{F0} Returns the intermediate code for an unsigned factor.
- P_L Returns the intermediate code for an argument list.

[1]We distinguish these as they may use different symbols for unary minus in exponent notation.

The remaining conversion equations for the intermediate code can now be defined (note L is shorthand for l, e; and l could be a single e).

$$
\begin{aligned}
&P_F(\texttt{" \sim "} \frown f) = P_F(f) \frown \texttt{" NEGATE\_"} \\
&P_F(f) = P_{F0}(f) \qquad\qquad (f \text{ can be } n, i, r, x, \text{ or } \phi) \\
&P_{F0}(\phi(L)) = P_L(L) \frown P_{F0}(\phi) \qquad (P_{F0} \text{ would simply output the string } \phi) \\
&P_{F0}(n) = \backslash\texttt{"} \frown n \frown \backslash\texttt{"} \frown \texttt{" int"} \\
&P_{F0}(i) = \backslash\texttt{"} \frown i \frown \backslash\texttt{"} \frown \texttt{" int"} \\
&P_{F0}(r) = \backslash\texttt{"} \frown r' \frown \backslash\texttt{"} \frown \texttt{" float"} \\
&P_{F0}(x) = \backslash\texttt{"} \frown x \frown \backslash\texttt{"} \frown \texttt{" float"} \\
&P_L(l \frown \texttt{","} \frown e) = P_L(l) \frown P_E(e) \\
&P_L(e) = P_E(e)
\end{aligned}
$$

The final four P_{FO} equations show the type information being included in the output string; for example, the tokens "36" or "3.142" become the strings (with embedded quotes around the numbers)

```
“ " 36" int”
“ " 3.142" float”
```

This is because we are still in the intermediate code, which will be converted to input for Forth by the second-pass operations — those with the appended underscore — e.g. +_ , *_ . We can now examine these in the context of the second pass of the compiler.

4.2 Second Pass

For the simple example infix input 10.5+5∗2.5, the first pass will have produced the intermediate output string with embedded quotes

```
" 10.5" float " 5" int " 2.5" float *_ +_                    (4.1)
```

This is then tokenised and interpreted by the second-pass compiler. At this level, `int` and `float` are interpreted as integer constants, so when the operation *_ is reached, the stack has three string/integer pairs on it. The code for *_ is shown in figure 4.

It takes four arguments consisting of two string/integer pairs, the integers containing type information. The four possible combinations of `int` and `float` are checked (the second case is applicable here). The numerical/ variable values are output as strings, followed by the appropriate ordinary arithmetical operators, here either * or `F*` (the floating point version). In addition, since `F*` requires that both its arguments be floating point, the conversion operator[2] `S>F` is inserted in the output string after any integer values.

[2]Single-precision integer to Floating point.

```
: *_  ( az1 type1 az2 type2 -- az3 type3)
 (: VALUE e1  VALUE type1  VALUE e2  VALUE type2 :)
   CASE
     type1 int =  type2 int =  AND  ?OF
       e1 e2 ^ " *" ^ int
     ENDOF
     type1 int =  type2 float =  AND  ?OF
       e1 " S>F" ^ e2 ^  " F*" ^  float
     ENDOF
     type1 float =  type2 int =  AND  ?OF
       e1  e2 " S>F" ^ ^  " F*" ^  float
     ENDOF
     type1 float =  type2 float =  AND  ?OF
       e1  e2  ^  " F*" ^  float
     ENDOF
     ( otherwise ) " Type error" AZ-ABORT
   ENDCASE
 2LEAVE ;
```

Figure 4: RVM-Forth code for *_ .

Finally the constant corresponding to the value is output, either `int` or `float`. Thus the output is a string and an integer, suitable for input to another operation of this kind.

After *_ has interpreted and dealt with the relevant parts of string (4.1), we come to +_ , which finds the following four arguments on the stack:

```
"10.5" float "5 S>F 2.5 F*" float
```

the last two being the output from *_ . Note these strings are actual strings.

The code for +_ is very similar to the above, and the final output will be

```
"10.5 5 S>F 2.5 F* F+" float
```

The final type indicator is dropped from this (by the second-pass compiler when the end of the expression is reached), and the remaining string can now be compiled by Forth.

For an example with identifiers (subject to the conventions mentioned in 4.1, page 8) we use

$(i + 7) * (j + 1.5)$

The first pass produces the *string* (recall these quotes are embedded)

```
" i" int " 7" int +_ " j" int " 1.5" float +_ *_
```

After both +_ have been processed, the stack will contain the following:

```
"i 7 +" int "j S>F 1.5 F+" float
```

Finally *_ will produce this

```
"i 7 + S>F j S>F 1.5 F+ F*"
```

having dropped the `float` type indicator.

The second pass doesn't have such a neatly defined set of equations to encapsulate it, as the arguments are no longer a single string, but four arguments (stack parameters): string, integer, string, integer. Also, `NEGATE_` and `F->F_`, dealing with negating variables and numbers, and type-checking the arguments for function applications[3], have different signatures again. They also have more conditions to cover with, for instance, four combinations of `int` and `float`. The specifying equations for +_ would be:

$$+\_(n_1, \texttt{int}, n_2, \texttt{int}) = n_1 \frown n_2 \frown \texttt{" +"}$$

$$+\_(n, \texttt{int}, x, \texttt{float}) = n \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

$$+\_(x, \texttt{float}, n, \texttt{int}) = \texttt{" S>F"} \frown x \frown n \frown \texttt{" F+"}$$

$$+\_(x, \texttt{float}, x, \texttt{float}) = \texttt{" S>F"} \frown x \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

4.3 Floating Point State Machine

For the general numeric literals subsystem — which handles floating point and exponent form literals — we adopt a state machine. A feature of the exponent form for infix expressions is that unary minus in an *exponent* must be written as a tilde rather than a normal minus (this does not apply to any other unary minus). Thus expressions such as

`-3.467e~6` or `-2.5e~10`

are admissible. This is to simplify the otherwise unwieldy process of identifying unary minuses for floating-point exponents and replacing them internally with tildes; the outputs will have normal minus signs for the usual Forth input. The machine itself has seven states; N_2, N_4 and N_7 are terminal states, and ϵ denotes an empty string.

[3]Neither are described in detail here.

$N ::= N_1 \mid DN_2 \mid .N_3$
$N_1 ::= DN_2 \mid .N_3$
$N_2 ::= \epsilon \mid DN_2 \mid .N_4 \mid eN_5$
$N_3 ::= DN_4$
$N_4 ::= \epsilon \mid DN_4 \mid eN_5$
$N_5 ::= N_6 \mid DN_7$
$N_6 ::= DN_7$
$N_7 ::= \epsilon \mid DN_7$

5 Conclusions and Further Work

The idea for this course stemmed from a desire to teach Language Systems at a more pure conceptual level, so that the concepts would be less obscured by their almost immediate abandoning for a more complex bottom-up approach, as was previously done.

We therefore used a top-down right-to-left method to parse expressions down to factor level, using recursive descent techniques to generate postfix versions readable by the Forth system — later via intermediate code to hold type information. The course thus began with deceptively simple concepts and could build naturally to a reasonably capable two-pass expression compiler. So the students learn about grammars and programming simple compilers in Forth.

We could go in a number of different directions with this, perhaps expanding in other ways than the floating point facility; for instance, one idea is to adapt this approach to make use of RVM Forth's native support for sets (using the C package contributed by Frank Zeyda [5]). This could be used to develop, for the course, an application to accept sets and basic set operations and generators from a suitably defined grammar, and convert them to valid Forth. Even eschewing floating-point support here, we still have the issue of types, as sets can contain strings and integers, and also pairs (maplets, using the ASCII notation `|->`) and nested sets. Therefore recursion would again be required to tease out the type information at each level and generate the appropriate tags for the first-pass output.

For illustration, a brief example of a simple set enumeration with string-integer pairs and the output which would be generated:

```
{"Dave" |-> 3291, "Li" |-> 3419} becomes
STRING INT PROD { " Dave" 3291 |-> , " Li" 3419 |-> , }
```

The grammar would, among other things, have to ensure that the maplet operator retained left associativity, and had the correct type signature. Some operators have right associativity, for instance domain restriction, and the bidirectional functionality will be needed to parse these. We will give consideration to a generalised and consistent system for type declarations in these language grammars, which would work with the set types of Forth and also with numeric types; also a closer analysis of the time complexity over a greater variety of grammars.

References

[1] A. V. Aho and J. D. et al Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[2] R. C. Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall, 1979.

[3] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM, Volume 13, No 2*, 1970.

[4] W. J. Stoddart and A. R. Lynas. A Reversible Computing Approach to Forth Floating Point. In M. A. Ertl and P. Knaggs, editors, *23rd EuroForth Conference Proceedings*, October 2007. On-line proceedings.

[5] W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In M. A. Ertl, editor, *18th EuroForth Conference Proceedings*, 2007. On-line proceedings.

The RVM-Forth software and manual can be downloaded from

```
http://www.scm.tees.ac.uk/formalmethods/download/rvm0_1.zip
http://www.scm.tees.ac.uk/formalmethods/download/rvmman.pdf
```