Building an LR parser for Pascal using Forth

Ian van Breda

Hailsham, East Sussex, UK

Abstract

The construction of a parser for Pascal is described. The parser is built by using Forth to include files that define the tokens and grammar of the language and may be thought of as a 'virtual machine' comprising a grammar code with lookup tables. The resulting parser can be run from within Forth or any other environment using simple drivers and is fully LR(1) capable. It is also well-suited for use with languages other than Pascal. An account is given of the author's early experiences with Forth at the University of St. Andrews and the Royal Greenwich Observatory.

1 Historical perspective

University of St. Andrews

I first came into contact with Forth when a colleague at the University Observatory in St. Andrews, Phillip Hill, came back from a visit to Kitt Peak National Observatory in Tucson, Arizona, excited about a newly invented computer language called Forth. A short time later, I used Forth on a spectrum scanner at the Cerro Tololo Observatory in Chile and on an infrared photometer at Kitt Peak, being very impressed with its ease of use and robustness.

At the time, much publicity had been given to the 'software problem', which alluded to the fact that, while there had been spectacular advances in hardware, there had been no equivalent in software.

Although Charles Moore had invented Forth originally for carpet manufacture, it was at Kitt Peak where it made its reputation on the 11-metre radio telescope that was responsible for the discovery of the great majority of molecules in interstellar space found in the early days of that subject. Using just a 16-bit minicomputer, it was possible to control the telescope, acquire observational data and perform data reductions for scientific analysis. This led to Forth's being dubbed 'the language of astronomy'.

We were fortunate to be able to purchase a system from Forth Inc. for a Data General Nova computer, the first Forth system outside the USA. This had the added attraction in St. Andrews that we were close to the Firth of Forth in Scotland: Charles Moore had originally intended to call it 'Fourth' for 'fourth generation language', but the original implementation could only take five-letter names.

This solved the 'software problem' for us instantly. We could have three students at a time working on data analysis using just one 16-bit minicomputer, helped greatly by the compactness of the Forth code. The very short learning curve for Forth meant that they were up and running very quickly in this new environment.

Around this time, there had been a move to use Camac, the interface standard used in nuclear physics, as a standard in astronomy: it was easy to access in software and had a number of existing cards for nuclear physics, of which one of the most interesting was a high-speed pulse counter for photomultipliers. This was connected to the Nova computer and allowed photon-counting photometry to be carried out in a remote dome, around 50 metres way, using a fibre-optic link.

The combination of Nova computer and Camac was also used in a scanning spectrometer. The Nova was later replaced by a PDP-11 computer that doubled as a built-in Camac crate controller.

Royal Greenwich Observatory

Subsequently I moved to the Royal Greenwich Observatory in Sussex (RGO), now sadly shut down after a spectacularly unsuccessful and ultimately fatal move to Cambridge.

Microprocessors were relatively new innovations at that point and we obtained a development system for the 8-bit Motorola MC6800 microprocessor to run a microFORTH system to control a high-speed photometer for use at an observing site in Spain.

Around the same time, a major ImageForth system was obtained from Forth Inc. to run on a PDP-11 computer attached to a scanning microdensitometer, which was run as a national facility for astronomers to scan photographs, both images of the sky and spectrograms. It was used to record scans on 7- and 9-track tapes for university users to take back with them for scientific analysis. It was also used for features like computer-assisted alignment of spectra.

The image processing system was ported to an MC6800 development system for work in the laboratory on an intensified photon counting detector, which was built for the South African Astronomical Observatory for use on their 1.9-metre telescope [1].

The image processing system was also ported to LSI-11 computers for use in the laboratory for development of liquid-nitrogen cooled solid-state detectors, particularly charge-coupled devices (CCDs), with a complete CCD camera being supplied to the Anglo-Australian Telescope. The controllers used to generate the waveforms for reading out CCD chips were based on bit-slice processors, also programmed in Forth. A summary of early developments in microprocessors using Forth at the RGO is given in [2].

Subsequently, detector development in the laboratory was switched to polyFORTH on MC68000 series processors running on VME bus. This was converted in-house from address threading to direct execution / subroutine threading on full 32-bit MC68020 and MC68030 processors [3]. For generating the wave-forms needed to read out CCD chips, MC68008 processors were also used. These were able to adopt the same software system with some minor conditional compilation to distinguish the two systems, mainly for arithmetic and the handling of CPU exception frames.

At the 4.2-metre William Herschel Telescope (WHT) on La Palma, VME systems running Forth were used for acquiring and processing CCD images, both for scientific astronomical detectors and the autoguiders at the various foci of the telescope. They were also used for the integrating intensified TV cameras needed to find and check that the desired faint object was being observed.

Individual fibre-optic links to specially designed VME cards, again programmed in Forth, were used to transfer detector data from the telescope, particularly to provide isolation against lightning strikes.

Ethernet, acting as a 'utility network' was used for the control systems on the telescope, as well as for individual instruments and sensors. Individual nodes on the network employed MC6809 processors programmed in Forth and were connected to the network by means of RS-232 links.

Image processing

The concept of the *ImageForth* image handling and processing system provided by Forth Inc. and installed by Charles Moore and Elizabeth Rather, is shown in Figure 1.

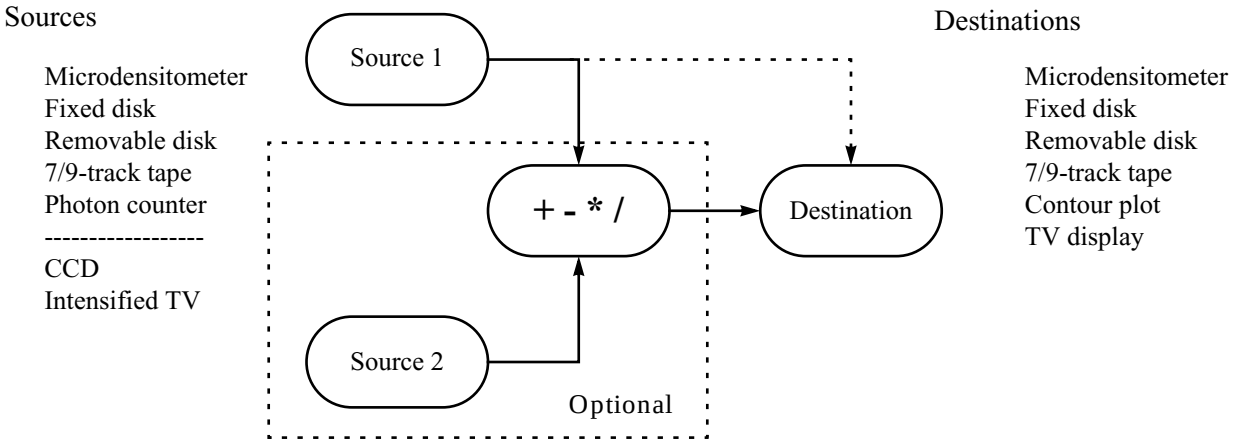

Figure 1. Schematic for the ImageForth image handling system.

In this very elegant scheme, various devices were able to act as image sources or destinations. Some could act only in one capacity (a contour plot on a storage tube could only be a destination, a CCD could only be a source). Others, such as a disk, could act in both roles. Sources simply had to be able to provide an image by reading one row at a time, destinations had to be able to write one row at a time.

An image 'pass' could be performed by transferring it directly to the destination. Optionally, it was possible to select two sources and perform arithmetic on them, such as dividing an image from a CCD by its 'flat-field' to allow for variations in sensitivity between the pixels.

Although not put forward as such, this represented an early application of object-oriented programming, with each device having methods for reading and writing rows.

Transferring to the Apple Mac

The MC680xx Forth was later ported to the Apple Mac, acquiring the name, Forth/68, where it proved to be possible to use pre-emptive multi-tasking, despite that fact that the native Mac OS has poor and complex task-switching and no in-built ability to run with command-line input. The parser described here has been developed on a PowerPC Mac Mini running in Classic mode. Although this is an emulation of the 680xx instruction set, it is easily fast enough to develop the parser, for which it takes less than one second to build all the necessary tables. The 680xx, being CISC, has the advantage of having an assembler that is very easy to use, unlike RISC processors. However, assembler is not needed for building a parser and the code used is ANS-compatible, although a front-end is required to make word lists compatible with Forth/68.

2 Early problems with Forth

Despite its considerable success, it was difficult to get Forth accepted by universities who used the facilities provided by the RGO. Some of the difficulties experienced with it were:

- Restricting names to three characters with counts led to some weird names with the inclusion non-alphanumeric characters. This led some to describe Forth as a 'write-only' language.
- Early systems crashed easily due to lack of syntax checking and complete system visibility.
- The use of Forth blocks for source code meant that the structure of definitions could be difficult to read.
- Initially there was no documentation embedded in the source code but later use of 'shadow blocks' that accompanied each source code block helped considerably.
- Stack operations could become very elaborate, again making reading of source code difficult.
- There was no support for type-ahead of the next command line during a long operation.
- Sometimes users were determined not to accept Forth, come what may.

The last point can be illustrated in the context of the William Herschel Telescope. This telescope was brought into operation immediately after primary commissioning, without the usual two-year bedding in period for telescopes of that size. This was made possible by lending one of our laboratory Forth systems to the La Palma observatory, as the telescope computers were well behind in their software development. It was on this Forth system that the telescope made its reputation by measuring the 'red shifts' (radial velocities) of infrared galaxies that had recently been identified by the IRAS infrared satellite.

However, in committee and without warning to us, one of the users of this facility described Forth as a 'pig'. It turned out that he had been given a list of definitions to use at the terminal that included hyphens, as is usual in Forth. However, he had tried to type these in as underscores and had not even bothered to ask before drawing his erroneous conclusion!

2.1 Advantages

Nevertheless there were many advantages:

- The user command-line interface is much better than either Unix or MS-DOS, which had given command-line input an unjustifiably bad name.
- While system crashes caused problems when some users were developing programs while others were trying to run fully operational programs, the simple solution was not to develop new programs while others were running operational programs.
- It was easy to develop programs for new instruments at the telescope, as these could be run by typing in primitive definitions then combining them into higher-level definitions later.
- Forth provides direct access to hardware, which makes testing much easier.
- Forth responds very rapidly to input commands.
- Pre-emptive multi-tasking made it possible both to run background tasks, for example to drive a cathode-ray display, while undertaking observations. It also meant that observations could be aborted in a controlled fashion when needed.
- Fully tested Forth applications are as robust as the very best programs written in other languages and are generally better.
- No bugs were ever reported on the WHT imaging systems on La Palma.

2.2 More recent developments

Since the early days of Forth there has been a number of improvements that answer most of the criticisms:

- There is plenty of memory space on 32-bit computers to allow full names to be used without having to resort to unusual characters.
- Use of word-processor text files for source code allows much improved documentation, including a brief description of the function of each definition, comment-per-line documentation and the use of indenting and phrasing within each definition.
- System crashes are greatly reduced through syntax checking and disappear almost completely if memory protection is also used.
- 'See-flow' debugging, which I believe was invented by Chris Stephens of Comsol Ltd, eliminates the vast majority of bugs, leaving only possible conceptual errors. In this scheme, source text is displayed alongside the current state of the stacks or registers, with a cursor marking execution progress. Execution advances by single steps by striking a key at the keyboard. The source code to debug is selected by enclosing it in a trace segment, while individual breakpoints can also be inserted. It is also possible to skip a specified number of breakpoints when there is a problem that only occurs after many definitions have been executed. This is much easier to use than the debuggers that come with the IDE development systems common with other languages. Surprisingly, it is not in as common use as might be expected.
- Type-ahead was added for the command line, which can also be edited, an essential for long names.
- LOCALS| greatly simplifies parameter stack operations, making definitions much easier to read.
- LOCAL definitions allow definition headers not needed in a running system to be discarded to avoid clogging up the dictionary with unused names (‘dictionary entropy’).

3 Why use other languages?

If Forth is so superior, why should other languages be used at all?

Firstly, although using a stack on a personal calculator is easier than the bracket notation common on other calculators, many users require a language that is more suited to the notation used in algebra, particularly when it comes to data structures. While algebraic notation can be used within the Forth context, it is usually circumscribed in its scope.

Another important factor is that there are many numerical algorithms written, particularly in C, that are very useful in image processing. For the RGO, there were many more users willing to program in C than in Forth, even though C sits uncomfortably as a high-level language with its terseness and quaintly eccentric choice of operators making it no easier to read than Forth, especially when it does not use comment-per-line documentation. Indeed its header files can be more difficult to read, its use of pointers is much less clear and often its logical expressions are more difficult to read than their Forth counterparts.

3.1 Why use Forth with these languages?

Forth provides a command-line interface that allows direct access to functions and procedures, making it both easier to use and to diagnose any bugs. It also adapts well to use with menus. Where other languages place excessive restrictions on data-typing, Forth can provide import routines to bypass such restrictions.

An important feature of Forth is that it can be used to generate the tables needed for parsing the strict grammars of other languages probably more easily than any of these languages themselves. This may seem surprising, given the free-wheeling nature of Forth. However, it is precisely the ability of Forth definitions to have names consisting of any group of characters that makes it possible uniquely to generate the tables needed for parsing by simply using `INCLUDE` to load the files that define the various aspects of the grammar and executing them directly.

An added bonus is that it is possible to make a parser fully LR(1) capable without the number of states getting out of hand. Normally, the number of these states is so large that special techniques are needed for reducing the number of states (and consequently also reducing the power of the parsing scheme).

3.2 Why Pascal?

The parser described here has been written to make it work easily with different languages, with the Forth code used to build the parser separate from the files that specify the syntax of the language.

Although C and its variants are used widely, Pascal was chosen because of its simplicity and because it wins hands-down over C on readability. This meant that problems of implementation could be solved within a simpler language set-up. C can, of course, be supported by implementing the necessary files that are used to define its tokens and syntax.

3.3 Problems with cross compatibility

A frequently heard criticism is that other languages cannot call Forth routines, although it is possible to call functions and procedures in other languages from Forth, with the appropriate stack adjustments.

This problem lies neither with Forth nor the other languages but in the implementation of those other languages using a single stack to house function / procedure arguments, local variables and the subroutine return address. Disadvantages of doing this include: building of the stack frame is more complex, and hence slower, than when using a separate return stack; the scheme does not match well to assembler subroutine calls, which usually transfer their arguments in registers but can also do so on a parameter stack.

The simple solution is to follow the Forth practice of using separate parameter and return stacks, making it possible for routines to be called both ways.

4 The function of a purser

The schematic workings of a compiler, of which the parser is a key part, is shown in Figure 2.

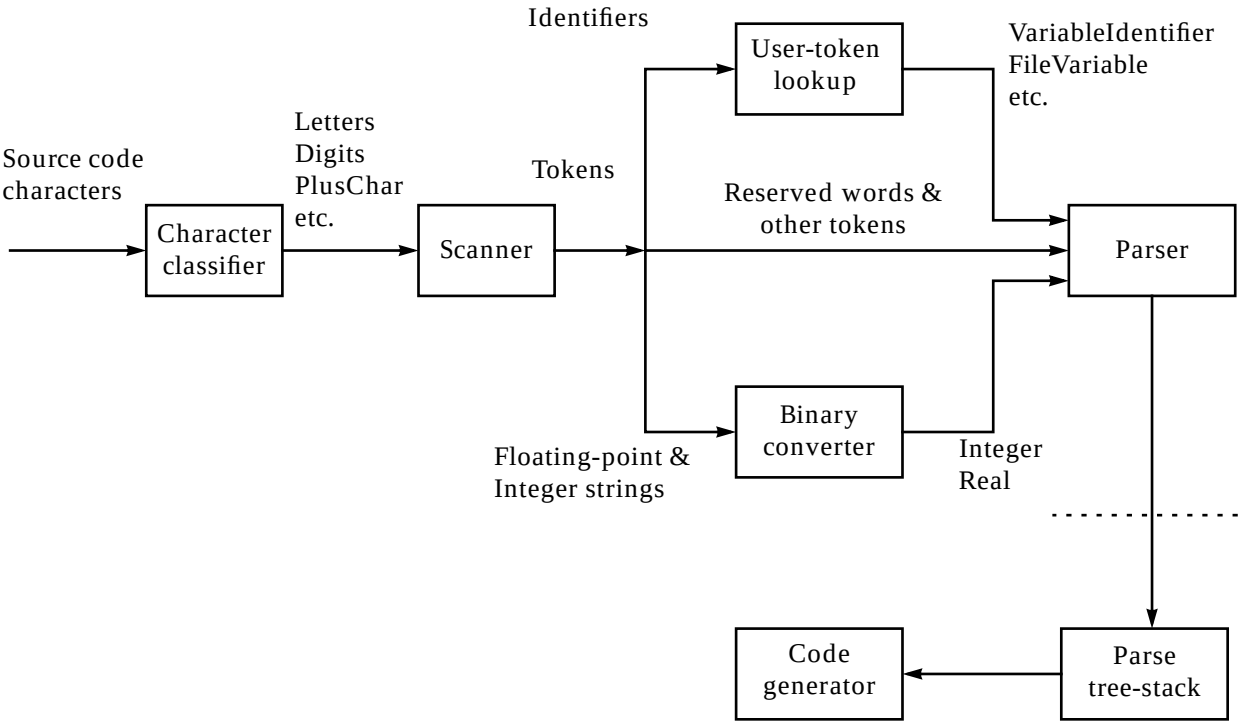

Figure 2. Outline operation of a compiler

Source code in the form of individual characters is passed to a classifier. This identifies generic letters and digits to simplify the process of scanning for tokens. It is implemented in the form of a simple lookup table.

The scanner extracts tokens from the input stream. This is different from the equivalent process in Forth where the tokens are simply obtained by looking for character strings that are separated by white

space (spaces and tabs) and ends of lines. For example, it is possible to write a formula in Pascal without spaces between the tokens. Thus

```
x:=y+z;
```

contains six tokens which need to be identified separately.

Some tokens, such as reserved words, can be passed directly to the parser for checking that the grammar for the language is being followed correctly. However, other tokens, particularly identifiers and numbers, need to be grouped in some way, as a grammar cannot possibly be defined based on the actual contents of such tokens.

Instead, user-defined tokens are classified into named groups, i.e. token types. Thus, when defining a new field in a Pascal record, we need a new identifier (`NewIdentifier`) that names the field. However, when used later in the source code, the parser must recognise it as the name of a data item that has already been defined, `FieldIdentifier`. This process is handled by calls to the semantic routines that are used to 'decorate' the basic grammar and which join the parser to the real world of computing.

Tokens and token types are passed to the parser, which checks that they satisfy the grammar of the language. The parser also builds an intermediate representation (IR) of the program being compiled, with the help of the semantic routines. The IR can take a number of different forms. In Figure 2 it is shown as a 'tree-stack'. In such a scheme, *semantic records* are added to a stack as parsing proceeds and are then incorporated into a tree structure as each grammar production is recognised.

The tree can later be converted into code that can run on a computer either as assembler code or can even take the form of Forth.

The parser itself extends as far as the dotted line in Figure 2. This is a natural break point, since building of the parser requires considerable dictionary space both for Forth code and for the support tables needed. It is therefore better, in a running compiler, for the parser builder to generate the lookup tables needed by the compiler and save these to disk, where they can easily be picked up later.

This results in a considerable saving in the memory needed, even though memory on modern computers is very large by comparison: the parse builder uses around 125K bytes for 32-bit Forth but needs only 15K bytes for the parse tables when using 16-bit table-cell sizes.

So far the development has proceeded as far as the dotted line in Figure 2, i.e. the parser is complete for Pascal, with some elements of tree-stack building added for testing out on sample programs.

Using Forth

The tokens and grammar are defined using files specific to the language for which a parser is being created. These files are simply passed through the Forth interpreter, sometimes in multiple passes.

A key issue is that it is possible to define the symbols in these files as Forth words that can be executed by the interpreter. Apart from meta symbols which control the building process, the symbols normally carry a reference index: there is no need to decorate the files with explicit numerical values, often used in other environments.

It is important to bear in mind that there are three stages to consider here:

(i) Building the tables needed for parsing by interpreting the language-specific files 'on-the-fly';

(ii) Using lookup tables and grammar execution code when compiling a program;

(iii) Actually running a program after it has been compiled.

We are concerned mostly with (i) here and how it relates to (ii).

The parser tables are built by including the Forth source code files interspersed with the files that define the syntax of the target language; the parser builder does not exist as a separate program.

5 Scanner

The scanner consists of two parts, the character classifier and token extraction. The notation used here follows closely that given by Fischer & LeBlanc in the original edition of *Crafting a Compiler* [4].

5.1 Classifying characters

In languages like Pascal, we need to classify source-code characters for building tokens. A partial listing of the file used to build the lookup table for performing this classification is shown in Listing 1; the numerical codes are ASCII-specific. This follows loosely the notation given by Fischer & LeBlanc for input to the

ScanGen scanner generator program.

The difference here is that all the words are Forth words that either define new names or help to build the lookup table when executed. The words must also be separated by white space or end-of-lines.

```
CharacterClasses
   CharClass Tab          =  09
   CharClass EndLine      =  10 , 13
   CharClass Blank        =  32
   CharClass LetterE      =  E , e
   CharClass OtherLetter  =  A .. D , F .. Z , a .. d , f .. z
   CharClass Digit        =  0 .. 9
   CharClass Star         =  *
   CharClass PlusChar     =  +
   ........................
   CharClass UpArrow      =  ^
   CharClass SingleQuote  =  '
EndCharacterClasses
```

Listing 1. Part of the file used to classify characters in Pascal

Before the Pascal-specific file can be included, we need definitions for the Forth words that define the named categories and also those that build the character-class lookup table. For this purpose, two vocabularies (i.e. named word lists [5]) are used, one for the words that do the classifying (`CLASSIFY`), one for the vocabulary that contains the class definitions (`CHARACTER-CLASSES`). Each such definition has a numerical-index in its parameter field, assigned in chronological order as it appears in the file.

The Forth source code for achieving this is shown in the partial listing given in Listing 2. Ada line-comments, prefixed by `--` are used for explanatory comments; they are also used for titles where related definitions are grouped together.

`CharacterClasses` is defined in the `FORTH` vocabulary, but the words that do the actual classifying are defined in the `CLASSIFY` vocabulary, The Pascal-specific file, as given in Listing 1, is then loaded by including the file whose name string is `$CharacterClasses`.

On subsequent execution of `CharacterClasses`, when the Pascal file is included, the compilation vocabulary is set to `CHARACTER-CLASSES`. The search order is set to the `CLASSIFY` vocabulary, searched first, followed by `FORTH-HOOKS` which is a vocabulary containing some very basic Forth words, including `\` to allow comments to be included in the text, and a few 'get-out-of-jail' words, such as `EMPTY`, which allow basic recovery if things go wrong during loading.

`CharClass` defines a named classification using `CREATE`, with a running chronological index (`CharacterClassCount`) in the parameter field of the defined word; the vector code is not executed at this stage.

When building the classification lookup table, whose address is set in `CharacterClassTable`, the equal and comma definitions use `GET-CODE` to find the code for the next character in the input stream if a single character. If the input item is a double character, the numerical code is returned (ASCII here). This code is used as an index into the table by `SET-CLASS` to insert the current class index into `CharacterClassTable`.

The double-dot not only sets the class code in the table for the next character / number-pair but also fills in the gaps between it and the previous table entry. Thus `A .. D` fills in all of the table elements for A, B, C, D with the current character class.

Several classifications, such as `Plus`, contain only a single character.

5.2 Building tokens

Before considering how the Pascal tokens are extracted from the input stream, a file is included giving names to all the Pascal tokens and token types. A partial listing is given in Listing 3, as the file is too big to include in its entirety here. Once again the file is executed in the normal way, as for any Forth source-code file, by using `INCLUDE` to load it.

```
   \ Initialising character class look-up table
: CharacterClasses   CharacterCount       \ Element count/size-in-bytes
   CHAR_CELL                              \ Size of each element in bytes
   0                                      \ No auxiliary parameters
   1 NEW-TABLE  DUP TO CharacterClassTable
                                          \ New dictionary-aligned 1-D table
   ERASE-TABLE                            \ Fill with default zeros
   CHARACTER-CLASSES DEFINITIONS          \ Compile to CHARACTER-CLASSES
   VIA  FORTH-HOOKS CLASSIFY  ENDVIA ;    \ Search CLASSIFY/emergency exits


CLASSIFY DEFINITIONS  VIA  FORTH  ENDVIA
   \ Place definitions below in CLASSIFY but search FORTH on its own.
   \  This means that the CLASSIFY definitions below cannot call each other

   \ Making a new class category
      \ Expects: previous class number; class name in input stream
      \ Returns: new class number
: CharClass ( n $ -- n|)
   CharacterClassCount                    \ Count is index code for class
   DUP CREATE,                            \ Define character class as a
      \  constant offset in scan actions table: 0=illegal
   DUP TO CurrentCharacterClass           \ Save as class being processed
   1+ TO CharacterClassCount              \ Advance current class total
   DOES> ( -- addr)                       \ Tail code expects parameter addr
      CharacterClassVector EXECUTE ;      \ Execute table-building code

   \ Starting specification of a class
      \ Expects: single ASCII character or numerical code in input stream
: = ( $ -- char/u)
   GET-CODE                               \ Next character/numerical code
   SET-CLASS ;                            \ Enter class number into table

   \ Continuing class specification
      \ Expects: character/numerical code in input stream
: , ( char $ -- char/u)                   \ Enter class for next item
      -- This = is the CLASSIFY version
   GET-CODE                               \ Next character/numerical code
   SET-CLASS ;                            \ Enter class number into table

   \ Making multiple insertions in classification table for a range
      \ Expects: ASCII character/numerical code in input, last item in range
: .. ( $ -- char2/u2)   CharacterCode 1+  \ Bump first to avoid double entry
   GET-CODE                               \ ASCII/numerical code, last char
   1+ SWAP  2DUP > NOT  ABORT" Characters in wrong order"
                                          \ Abort if not in ascending order
   DO  I SET-CLASS  LOOP ;                \ Fill in table entries for range
      -- This omits first item in range, else SET-CLASS would abort

   \ Terminating character class defining
: EndCharacterClasses
   FORTH DEFINITIONS ;                    \ Restore basic compile/search

FORTH DEFINITIONS                         \ Restore basic compile/search

$CharacterClasses INCLUDED                \ Build character-class defns.
```

Listing 2. Building the character-class lookup table

```
-- Single-character tokens --
     Token '+'         Token '-'         Token '*'         Token '/'

-- Double-character tokens --
     Token '<='        Token '>='        Token '<>'        Token ':='

-- Reserved words --
     Reserved and         Reserved array         Reserved begin

-- User-defined token types --
     UserToken DigitSequence             UserToken UnsignedReal
     UserToken Identifier                UserToken NewIdentifier
```

Listing 3. Partial listing of the Token List file for Pascal

`Token` is a predefined word that gives a name to a token for use when interpreting the formal grammar for Pascal; these definitions are placed in a `TOKEN-LIST` vocabulary. The names are enclosed in quotes to differentiate them from the meta characters used in the formal definition of the grammar.

`Reserved` places definitions for the reserved words in their own vocabulary, `RESERVED-WORDS`. The reason for this is that, when parsing, the reserved words need to be searched first before any other definitions, by placing `RESERVED-WORDS` at the top of the search order.

`UserToken` defines *token types* for tokens defined by the user and are also held in `TOKEN-LIST`. These are identifiers and numbers which need to be categorised by kind for processing in the grammar.

As with character classes, the defined words are given an ordered index and use vectored code for execution later when processing the Pascal grammar file.

It will be seen that, for example, the plus character appears in three guises: as `+` in the source code of a program; as `Plus` for building tokens; and as `'+'` in the formal grammar. This might seem overkill but it does make the various files easier to read and, in the formal grammar, is essential to differentiate tokens form the meta symbols.

In the literature, token building generally uses so-called *regular expressions* to build what is somewhat grandiloquently called a *finite automaton*. In such an implementation, each token / token-type is defined by a regular expression which is then used to build a lookup table. The automaton simply goes from state to state according to the class of each character in the input stream; there is no nesting.

The disadvantage of using regular expressions is that some token definitions overlap, such as integers and floating-point numbers, so these need to be spliced together creating some complication for the code used to build the lookup table.

Instead, we use a scheme based on the common EBNF (Extended Backus-Naur Form) developed originally for processing Algol grammars. In this, each token-building definer must start with a character or characters that are unique to it.

The extended EBNF meta characters have the following meaning:

```
|       -- alternatives
( ... ) -- grouping
[ ... ] -- single option
{ ... } -- multiple option
.       -- continuation
```

Extra definitions are added for processing the tokens

```
{Toss}  -- discard character
{Scrub} -- discard complete token
#Token  -- specifies a token type
```

A partial listing of the file used to define how tokens are built is shown in Listing 4.

As before, the required lookup table is built with the help of Forth words, defined specifically for the purpose, by including the file. For example, opening brackets of various types start a new line in the lookup table; some extra processing is needed, however, to removed duplicate and redundant rows.

All `Token:` and `Gluon:` definitions start a new row in the scanner lookup table. `#Token` expects the name of a token or token type following in the input stream and inserts a marker for that token / type.

This generates a lookup table for which a partial listing is shown in Listing 5.

```
-- Macros --
     : Letter   ( LetterE | OtherLetter ) ;

-- Anonymous white space --
     Token:   { WhiteSpace } {Toss}

-- Simple non-alpha tokens --
     Gluon:   PlusChar '+'
     Gluon:   Less '<'  .  [ Greater '<>' | Equal '<=' ]

-- Compound tokens --
     Token:   Letter #Token Identifier .
     { Letter | Digit } #Token Identifier
```

Listing 4. Partial listing of file used to define how tokens are built

```
 Action codes
    REUSE-CHAR = 0                          APPEND-CHAR = 1
    NO-APPEND = 2                           SCRUB-TOKEN = 3
    Class         Do  Token         Next    Class         Do  Token         Next
 Row 0
    Default       2   ErrorToken       0    Unprintable   2   ErrorToken       0
    EndFile       2   EndOfFile        0    Tab           2   xx               0
    EndLine       2   xx               0    Blank         2   xx               0
    LetterE       1   Identifier       5    OtherLetter   1   Identifier       5
    Digit         1   DigitSequence    8    Star          1   '*'              0
    PlusChar      1   '+'              0    MinusChar     1   '-'              0
    Slash         1   '/'              0    Equal         1   '='              0
    Less          1   '<'              1    Greater       1   '>'              2
    DotChar       1   '.'              3    CommaChar     1   ','              0
    ColonChar     1   ':'              4    Semi          1   ';'              0
    LParenChar    1   '('             13    RParenChar    1   ')'              0
    RBracket      1   ']'              0    LBracket      1   '['              0
    LBrace        2   xx              14    RBrace        2   ErrorToken       0
    UpArrow       1   '^'              0    SingleQuote   2   xx               6
 Row 1
    Equal         1   '<='             0    Greater       1   '<>'             0
 Row 2
    Equal         1   '>='             0

 Row 5
    LetterE       1   Identifier       5    OtherLetter   1   Identifier       5
    Digit         1   Identifier       5
```

Listing 5. Partial listing for the lookup table used to build tokens and token types

The table is two-dimensional with rows corresponding to states of the automaton and columns corresponding to the input character-code classification. Each cell in the table contains three fields: what to do with the character, token index code if any; and the next row to go to. Only non-default table cells are shown.

As each character is taken from the input stream and classified, the character itself is normally added to a string which will ultimately form the required token string.

The action to take is looked up in the table row that corresponds to the current state of the automaton. It may be to re-use the character without adding it to the token string, extract the string from the input stream and add it to the token string, extract it from the input but discard it (as in character strings delimited by quotes that don't form part of the string as such) and erase the current token (as happens with comments which, in this implementation, are not passed to the parser).

The default table cell is to re-use the character without extracting it from the input stream, with no token set and destination row zero. Moving to row zero also indicates that no further characters may be added to a token.

Processing of a token always starts at row zero with an empty token string. Thus, if an unprintable character is encountered, it is not added to the token string, but generates a special `ErrorToken` and returns to row zero indicating that the token is complete. In this case the token string is empty.

If a `<` symbol, which has the classification `Less`, is encountered at the start of a token, it is added to the token string, and control moves to row 1. If followed by a character classified as `Equal` or `Greater`, the character is added to the token string and control is transferred to row zero, indicating that the token is complete. Any other character causes a return to row zero, without adding to the token string.

We thus end up with possible strings `<`, `<=` or `<>` for the tokens themselves with the scanner returning index values for the named tokens `'<'`, `'<='` or `'<>'`, as applicable.

When a character `LetterE` or `OtherLetter` is encountered, the token is set as being an `Identifier` and control is transferred to row 5. If a letter or digit comes next, the character is added to the token string and the token again noted as an `Identifier`, with the token data updated. Processing continues at row 5. For any other character class, control is returned to row zero, indicating that the token is complete. In this case, both the token-type, `Identifier`, and the token string itself are returned, as the former is needed for checking the grammar while the latter is used to identify the actual identifier name.

The above scheme means that the longest possible token is always taken. Possible ambiguity is avoided by not always marking a string as a possible token. If we take the example of the string 100..200, this represents a range in Pascal. At the point where 100 is input, the token can be a `DigitSequence` (integer). However, when the dot following is found, the string could be part of a floating-point (real) number, 100.9, say, so the dot is added to the token string, which now has four characters, but the separate token code, character count (three) and saved `>IN` pointer are not updated.

For a real number, the dot must be followed by a `Digit` character. However, in this case it is followed by another dot, which is not allowed in a floating-point number. Control is therefore returned to row zero, indicating that the token is complete. The token-string count is backed off to three, `>IN` is backed off to before the first dot and a `DigitSequence` is returned with a token string containing just the 100.

6 Parsing

The function of a parser is to make sure that the source code for a program satisfies the syntax of the language. It also has the task of building an intermediate representation for the program.

6.1 Grammar notation

Grammars are often illustrated in the form of syntax diagrams, which can be very helpful in understanding the way in which the grammar works. However, this is not too much help in writing a parser for which we need a formal textual method of describing the grammar. This is done with a list of *productions* expressed in EBNF notation. The start of the list for Pascal is shown in Listing 6.

```
Productions

<start>                         -> <program> ( EndOfFile | endPascal ) ;

<program>                       -> <program_heading> ';' <block> '.' ;

<program_heading>               -> program <new_identifier> #ProcedureId
                                   [ <program_parameter_list> ] ;
```

Listing 6. Start of productions for Pascal

The left-hand side of each production takes the form of a named *nonterminal*, which specifies some construct in the language. The nonterminal is said to *produce* the right-hand side, which consists of a mixture of other nonterminals, *terminals*, which are the tokens and token types for the language, and meta characters which specify the way in which the production works in the context of the language.

The grammar is described as context-free, since, whenever a nonterminal occurs on the right-hand side of a production, it always has the same meaning.

The notation used follows closely that used by Fischer & Leblanc [4], adapted so that the file can be included in Forth, with all words delimited by white space and end-of-lines. Nonterminals are included in hairpin brackets for visibility in plain-text files. In order to enable nonterminals to be defined in the Forth dictionary, compound names use underscores as separators.

For the terminals, reserved words appear as they are, while special symbols appear in quotes and user tokens are identified by their token types. Semantic-routine references appear by name with a hash symbol in front. The hash has no special significance, except that it catches the eye in a plain-text listing.

The EBNF meta symbols have similar meanings to those used for defining how tokens are constructed (Section 5.2) but there is no dot for continuation. The symbol `->` starts the body of a production and a semicolon indicates its end.

Items in parentheses invariably represent a list of options separated by alternation symbols (vertical bars). Items in square brackets represent a single option, which may or may not be present in the source code being parsed. Thus a program parameter list may not be present in a program header (Listing 6).

Multiple options are included in braces. In the production

```
<label_declaration_part>      -> [ label DigitSequence
                                    { ',' DigitSequence } ';' ] ;
```

there may be nothing in the label declaration part (because the entire production is enclosed in square brackets). However, when present, it must start with the reserved word, `label`, followed by a `DigitSequence`. This, in turn, may be followed by as many `DigitSeqence` terminals representing other labels as we like, each preceded by a comma. A semicolon in the source code terminates the list.

6.2 The parsing process

Parsing of a program proceeds by looking up the next token/type in the input stream and using it as a terminal *lookahead*, starting at the `<start>` production, Listing 6. A Pascal program must start with the reserved word `program`. As implemented here, this is accepted as a valid lookahead by the first production, which nests a return point past `<program>` and goes on to the start of the second production. Again `program` is a valid lookahead, so control is passed to the start of the third production.

This can be represented by a series of *configurations*:

```
<start>                     -> <program> ● ( EndOfFile | endPascal ) ;

<program>                   -> <program_heading> ● ';' <block> '.' ;

<program_heading>           -> ● program <new_identifier> #ProcedureId
                               [ <program_parameter_list> ] ;
```

Listing 7. Initial configurations in the Pascal grammar

where the various points are marked with dots (blobs), with the first two being nested on a parse return stack.

At this point we encounter the `program` token at the start of the production. This is checked against the current lookahead and control is passed to the point after `program`. A new terminal is extracted from the input stream which, in this case, must be a `NewIdentifier`:

```
<program_heading>           -> program ● <new_identifier>
                               [ <program_parameter_list> ] ;
```

The production for `<new-identifier>` is now processed and parsing proceeds as before.

After processing any arguments in the program parameter list, if present, the end-of-production unnests the parse return stack and control is transferred back to the `<program>` production, as in Listing 7. Here, a semicolon is expected in the input stream and a check is made against the lookahead terminal; if they don't match, a syntax error is declared. If they do, the parser goes on to process `<block>` which does most of the work of parsing the program. It must be followed by a full-stop (period).

Once the `<program>` nonterminal completes, the parse return stack is unnested and control is returned back to the `<start>` production, where the next terminal must either be an end-of-file or `endPascal`, which is an extra reserved word added for returning to Forth processing.

At the end of the parse, some form of intermediate representation is returned. Here it is envisaged that it will be an *abstract syntax tree*, which is probably the simplest form to build and is also good for optimisation. The ultimate goal, though, is to produce assembler code for a target processor.

6.3 LL vs. LR parsing

The Forth programmer will immediately see a close resemblance between grammar productions and colon definitions in the way that they work when executed.

However, there are a couple of important reasons why we can't simply convert a grammar production to some form of colon-style definition. Firstly, a given nonterminal may have more than one production. For example

```
<adding_operator> -> '+' | '-' | or ;
```

is the equivalent of three productions, which can also be written as

```
<adding_operator> -> '+' ;
                  -> '-' ;
                  -> or ;
```

Listing 8. Splitting up alternative productions

Thus `<adding_operator>` can accept any one of three operands as its lookahead token and a selection mechanism is needed for determining which production to follow. Here it is done by means of a *parser lookup table*. The table has rows corresponding to nonterminal index, columns corresponding to terminal index and entries being the starts of productions in the grammar.

So far we have described a parsing technique known as top-down, predictive or recursive descent parsing. It is also known as LL(1) parsing, as the parse processes the source code input from left to right and the derivation also proceeds from left to right in the grammar, with a single terminal lookahead. In principal, it is possible to use more than one terminal as a lookahead: LL(*k*) denotes recursive descent parsing with *k* lookahead terminals but the lookup tables become impractically large. In what follows we shall take LL to mean specifically LL(1).

However, there are cases where choosing which production to select is ambiguous. Thus in a function declaration

```
<function_declaration> -> <function_heading> ';' <block> |
                          <function_heading> ';' <directive> |
                          <function_identification> ';' <block> ;
```

Listing 9. A nonterminal with a prediction conflict between productions

there is no way of predicting which of the of the productions applies in a particular situation, as all three accept the reserved word, `function`, as the lookahead terminal: `function` is the only terminal in the *first set* of each production, i.e. the set of terminals that can be accepted at the start of a production.

The solution adopted in LR(1) parsing (Leftmost parse, Rightmost derivation, single lookahead terminal) is to process productions which conflict in this way in parallel; parsing proceeds until a complete production is recognised. This is also known as bottom-up parsing or shift-reduce parsing from the way that it processes the parse stack often used in other implementations. As with LL parsing, in what follows we shall take LR to mean pecifically LR(1) parsing.

By way of illustration, we consider just the first two productions, where the conflict is most obvious. If these are processed in parallel then we have an LR state represented by

```
<function_declaration> -> ● <function_heading> ';' <block> ;
                       -> ● <function_heading> ';' <directive> ;
```

LR configurations also include the follow set, i.e. the full set of terminals that can follow the left-hand-side nonterminal in the grammar. This is because the productions may need to be processed in parallel until one or more productions reaches its end-of-production, in which case the follow set may be needed to

resolve any conflict. If that is not possible, we have a so-called *shift-reduce* or *reduce-reduce* conflict and the grammar must be modified to avoid this.

In the case above, we do not need to go so far, since, when we eventually get past the semicolon token in the two productions,

```
<function_declaration> -> <function_heading> ';' ● <block> ;
                       -> <function_heading> ';' ● <directive> ;
```

the parser will accept, as lookahead tokens, any of `begin`, `const`, `function`, `label`, `procedure`, `type` and `var` for `<block>` and `Identifier` for `<directive>`. Since the two sets do not overlap, the parser can determine uniquely which production to select.

At this point, the LR state has split into two separate productions, which might be described as *singleton states* and which can be parsed using the simpler LL method.

6.4 Encoding the grammar

Named nonterminals

Before approaching how to represent the grammar, we first need to assign numerical values to the various named nonterminals that appear in the grammar productions.

This is done by making a pass on the grammar productions file using the techniques shown in Listing 2. In this case the meta characters and symbols have empty definitions, while the vector codes for the terminals are also set to do nothing. Any word that cannot be recognised is checked to see that its name is enclosed in hairpin brackets. If so, it is added to the dictionary and given a running index, which is later sorted alphabetically to help with displaying the grammar from within the Forth parser builder program.

Brackets

It is common practice to rewrite productions containing bracket pairings to avoid their use. For example,

```
<statement_sequence> -> <statement> { ';' <statement> } ;
```

can be rewritten by adding a new production as

```
<statement_sequence> -> <statement_sequence> ';' <statement> ;
                     -> <statement> ;
```

However, there are two significant disadvantages to this: firstly, it requires a major rewrite of the Pascal grammar to convert it to this form, as there is a large number of brackets of various types in the Pascal EBNF grammar; secondly, as seen above, the nonterminal is changed from being able to be processed using LL to the more complex LR processing.

The way chosen here is to treat each opening bracket as an anonymous nonterminal in its own right that 'owns' the production(s) enclosed within the bracket pairing. These nonterminals are given ordered index values that follow immediately after those for the named nonterminals (in hairpins).

Grammar coding

We can then construct a version of the grammar that can be executed when parsing a program, rather in the manner of a series of interpretive Forth colon definitions for the individual productions.

In this representation, the productions are laid out end-to-end as a series of *grammar items*, in the order that they are given in the grammar, but without the left-hand nonterminals. Every symbol in the right hand side of each production is represented by a grammar item, consisting of two or more 16-bit cells (they can be compiled as 32-bit, if desired).

The format of these items is shown in Figure 3. Each item starts with an item 'class', which specifies its type. This is followed by one or more parameters. For a terminal it is the token or token-type index, generated when the token list was loaded, as in Listing 3. For (named) nonterminals it is the index generated in the first pass on the grammar. For a semantic item, it is an index into a list of semantic routines that are to be executed as parsing proceeds.

The class specifies the type of action to take in each case. In a sense, we have a 'virtual machine' with each class consisting of a machine instruction, followed by a series of parameters that define the detailed behaviour of each instruction through the medium of lookup tables.

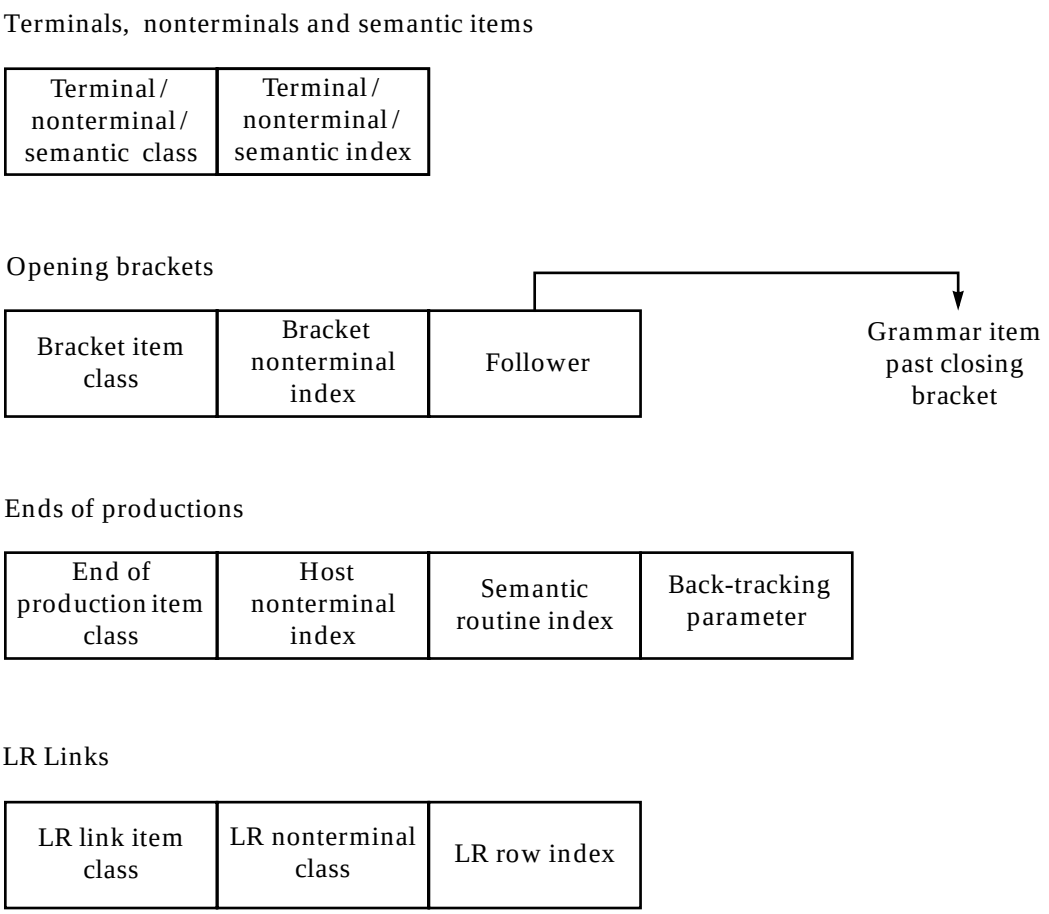

Figure 3. Structure of items in the grammar code

For an opening bracket, the class is specific to the type of bracket: parenthesis, square bracket or brace. The index following is that for the anonymous nonterminal for that bracket, while the follower gives the offset past the matching closing bracket within the grammar coding.

End-of-production items are generated for terminating semicolons, closing brackets and alternation symbols. The index for a semicolon is that for the host named-nonterminal on the left-hand side of the production. For a closing bracket, it is the index for the opening bracket, treated as a nonterminal. For an alternation symbol, it is the either the index for the left-hand named nonterminal, if not inside a bracket pair, or that for the opening bracket, if inside a bracket pairing.

The third parameter is the index for an optional semantic routine when it occurs at the end of a production.

It is assumed that, when parsing, each nonterminal will add a semantic record to the parse tree-stack. However, it is not always possible to guarantee that such a record will be present when an optional nonterminal construct that derives the empty string is not present. By backtracking, it is possible to enter an empty semantic record on the parse tree-stack of the right type, so that any semantic routine executed at the end of a production can always expect a given number of nonterminals on the stack for that production.

LR processing is slightly different in that two classes are used, followed by an index for the LR state.

Sets

In order to be able to build the required parse lookup tables, an additional pass is first made on the grammar productions text file to build a grammar table coded as given in Figure 3 but ignoring semantic references and assuming that none of the nonterminals needs LR processing. We call this the 'LL grammar' and building these tables is then done through one or more passes on this variant of the

grammar in memory, rather than the text file. In all cases the pass is repeated until there are no more changes found.

This allows some important tables to be defined. Firstly, we need to know which nonterminals 'derive the empty string', i.e. are optional and may be absent in the source code for a program. An example of this is:

```
<empty_statement>              -> ;
```

which has an empty production. Or

```
<constant_definition_part>     -> [ const <constant_definition> ';'
                                  { <constant_definition> ';' } ] ;
```

where the construct may or may not be present in a program. On the first pass on the LL grammar code, both of these nonterminals will be marked as deriving the empty string.

Because `<empty_statement>` is in the list of productions for `<simple_statement>`, which can be written:

```
<simple_statement>      -> <empty_statement> ;
```

`<simple_statement>` will only be marked as deriving the empty string on the second pass, since the status of `<empty_statement>` is not known first time round.

Two other sets are essential in building the parse tables. The first set is the set of lookahead terminals that can start a nonterminal. These sets are held as bit patterns in an array indexed by nonterminal parameter value. For example, in Listing 6, it can be seen immediately that the reserved word, `program`, is in the first set for `<program_heading>` and will be added to the list of first sets on the first pass. However, it will only become apparent that `program` is also in the first set for the `<program>` nonterminal on the second pass. Again, passes are repeated until there are no further changes to the first sets.

Once the derives-the-empty-string and first-sets tables have been set up, it is possible to find the follow sets for all the nonterminals, again by multiple passes on the LL grammar code.

All of this then allows those nonterminals to be found that need to be treated by LR parsing because there are conflicts between the first sets (or first and follow sets if the nonterminal can derive the empty string).

Bracket nonterminals may also need to be treated using LR methods. For example, in

```
<case_statement>        -> case <case_index> of <case> { ';'<case> } [ ';' ]
                           end ;
```

the brace nonterminal must be treated using LR, as the production itself starts with `';'` which is also in its follow set, so when `';'` is encountered as the lookahead, the parser will not know beforehand whether this is the beginning of another `<case>` construct or the optional `';'` at the end.

6.5 Building the final grammar execution code

All of this now allows the final grammar code table to be built. This time, semantic routines are included, along with LR links for those nonterminals and brackets needing LR processing. The index field for each of these is set to the index values for an LR state., which are given index values following on from those for named nonterminals and bracket nonterminals. The LR link replaces the normal nonterminal/bracket grammar item.

There is one LR state for each LR link item in the grammar. Each LR state must then be 'closed' by looking at all productions and other LR states that the state can link to, with new LR states being added, as needed. Each of these is given its own index and an LR link that is added to the end of the grammar execution code in the form shown in Figure 3.

In the Pascal grammar, as written in Jensen & Wirth [6], there are only 11 LR links within the main body of the grammar rising to 25 LR states altogether when these are closed.

Two parse tables are built, one has rows that correspond to index values for named nonterminals, bracket nonterminals and LR state index values which are tagged onto the end of the terminals, one per LR

state. The columns are numbered by index for the lookahead terminal. Each entry contains a position in the grammar code.

The second table is only used in the LR parsing process to look up the return point once a production has been recognised in the parsing process. It has rows corresponding the LR state index and columns matching the nonterminal index for the production in question.

As these tables are very sparsely populated, they are implemented in compact form using double-offset indexing as described in [4].

6.6 Grammar item actions

Associated with each grammar item is a 'driver' specific to that item. The different forms for the brackets give rise to three different grammar-item classes, making a total of nine altogether. These drivers are language-independent and so only need to be written once. They also have much code in common.

This gives a very simple scheme for executing a parse. We begin with the `<start>` production and a zero on the parse return stack. For each grammar item, the drivers carry out the following actions:

`Terminal`

When a terminal grammar item is processed, the driver looks to see if there is a match with the current lookahead terminal. If so, control is transferred past the item, otherwise a syntax error is declared.

`Named nonterminal`

For a named nonterminal, control is transferred to the start of the production that matches with the terminal lookahead, by looking it up in the main parse table. The return position past the grammar item is saved on the parse return stack.

If the nonterminal derives the empty string (i.e. is optional in program source code) and the terminal is not in the start set of the nonterminal, control is transferred past the grammar item without any nesting. If the terminal is not in the follow set, a syntax error is declared.

`Opening parenthesis`

This is very similar to that for a named nonterminal, except that, if the bracket-nonterminal can derive the empty string and the terminal is not in the start set but instead is in the follow set, control is transferred past the closing parenthesis.

`Opening (square) bracket`

Again behaviour is very similar to a parenthesis. If the lookahead terminal is in the first set for the bracket nonterminal, control is transferred to the start of the matching production (normally just past the opening bracket) and the location past the closing bracket is nested on the parse stack. Thus control will be passed to that point, once processing of the production has been completed.

Since bracket pairs always indicate a single-option for a program, if the lookahead terminal is not included in the start set of the production but is in the follow set for the bracket nonterminal, control is transferred past the closing bracket without any nesting. Again, a syntax error is declared if the lookahead terminal is in neither the start set nor the follow set for the bracket nonterminal.

`Opening brace`

This is almost identical to the behaviour for an opening bracket. The difference is that, when a match is found with the start set for the brace nonterminal, it is the point before the opening brace that is saved on the parse return stack, so the process will be repeated until a terminal is found that is not in the start set.

`Semantic reference`

The driver for this type of item uses the index in the grammar item to look up a semantic routine whose address or Forth token can be found in a table of references to the semantic routines for the language. This is the case where the semantic call is located within the body of a production, which must be in LL mode at that point for it to work.

`End of production`

If backtracking is needed because there is a nonterminal in the production that can derive the empty string or is an option bracket, the driver works backwards through the parse tree stack and inserts an empty semantic record into the parse tree-stack if the nonterminal is not present. It then executes the semantic routine, if there is one, which can then expect a fixed number of items on the parse tree-stack.

The semantic routine must leave a single nonterminal on the tree-stack. If required, it may need to add a semantic record, e.g. when an operator is encountered, as in Listing 8, there must be a semantic routine, e.g. `#AddOp` (not shown in the listing), that adds a semantic record for the operator using the most recent token extracted from the input stream to specify its type.

At the end, the driver marks the top item on the tree-stack with the index for the enclosing nonterminal, LHS nonterminal or bracket nonterminal, as appropriate.

`LR Link`

Processing of LR links is only slightly more complex than their LL counterparts and two drivers are used basically for looking up two different parser tables.

In the first case, the main parse table is looked up by row (LR state index) and column (terminal lookahead) and control is transferred to the point in the grammar code specified in the table, which can be to another LR-state link, or other grammar item, in which case processing will continue in LL mode.

However, the return destination saved on the parse return stack corresponds to the second class field in the LR link, Figure 3. Ultimately, the return will always occur at the end of a production, after a switch has been made to a singleton state, i.e. LL parsing.

At this point the second driver comes into play and looks up the second parse table by LR index for the row number and nonterminal index from the completed end-of-production to determine the location in the grammar code to which control should now be transferred.

Parse tree-stack

It is envisaged that the semantic routines will be responsible for adding semantic records to the parse tree-stack and for building tree structures by combining these, as required, by unlinking them from the stack and relinking them into a tree structure that constitutes the intermediate representation. From there it is possible to convert the representation to assembler code, which may include optimisation. An alternative would be to convert the tree directly into Forth and load that via an optimising compiler.

It is also envisaged that syntax errors will simply generate an abort, with an appropriate error message. Compilers that try to correct errors generally end up with more consequential errors further on in the parse, which only serve to confuse the issue (as in Java). Correcting such errors is very quick on modern desktop computers so making corrections one-by-one easy to do.

7 Rewriting the grammar

Throughout, the intention has been to minimise the rewriting of the grammar. In particular, it has been found unnecessary to rewrite it in a bracket-free form, sometimes called a ‘standard’ form. As shown above, EBNF works fine and can be executed directly in Forth to produce a very compact set of parse tables.

However, there is one area where some rewriting is necessary. This involves the question of Identifier and is because nonterminals that have ‘identifier’ in their names invariably end up at a production that consists of `Identifier` on its own. This means that we get a large number of reduce conflicts – almost fifty in Pascal, including the notorious ‘dangling’ else which occurs in both Pascal and languages based on the C syntax.

Thus, instead of

```
<field_identifier>     -> Identifier ;
```

we need to write

```
<field_identifier>     -> FieldIdentifier ;
```

This means that, when a field is defined, its entry in the dictionary must be flagged as being a `FieldIdentifier` token type by the appropriate semantic routine.

8 Conclusion

The scheme given here has found to work very successfully in parsing simple Pascal programs, using a subset of the semantic routines needed for a full compiler. The tables and grammar code used for this are very compact, amounting to around 15K bytes in 16-bit form.

The combination of LL and LR techniques has been found to operate very smoothly. Only 25 LR states are needed in total, so there is no need to try to reduce the very large number of states generated by conventional LR-only parsers using the common SLR and LALR techniques, which also reduce the power of the parser. The parser described here is fully LR-capable. Pathological examples of grammars quoted in the literature, that are not able to be handled using SLR or LALR but still satisfy the conditions for LR parsing, are handled with ease by the parser described here.

For those studying compiler writing or defining a new language, the parser may be set to show how the first and follow sets for the different nonterminals are built up, pass by pass. It is also possible to examine the various productions, including LR links in an easy to read form. A report on the language can also be written automatically as a plain-text file to disk giving a variety of information about the language, including details of the LR states and how they link together. This is reasonable for Pascal which has only 25 LR states.

The report can then be examined at leisure. If needed, the building of the start and follow sets, which require several passes on the grammar code, can also be included in the report.

At the same time, it is still possible to treat the parser as a 'black box' by feeding it with the files required to specify the language syntax without having to do any special coding.

The grammar code and lookup tables are such that the parser can easily be run in any computer environment, although it is simpler to do so in Forth.

References

[1] A.R. Jorden, P.D. Read and I. G. van Breda. Photon counting Reticon system-description and performance. *SPIE Conference, Instrumentation in Astronomy IV*, 331, 368-375, 1982.

[2] I. G. van Breda and N. M Parker. Forth and microprocessor applications at the Royal Greenwich Observatory. *Microprocessors and Microsystems*, 7, 203-211, 1983.

[3] I. G. van Breda and D. J. Thorne. The Handling of Large Format CCD Images. *ESO-OHP Workshop on the Optimization of the Use of CCD Detectors in Astronomy, ESO Conference and Workshop Proceedings*, 25, 83-92, 1986.

[4] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler.* Benjamin / Cummings Publishing Co., 1988.

[5] I.G. van Breda. Some comments on the Proposed Standard. *EuroForth* 2012.

[6] K. Jensen and N.Wirth, revised by A. B. Mickel and J. F. Miner. *Pascal User Manual and Report*. Springer Verlag, 1991.