Documentation for *Parsing* videos

This .zip archive contains source files and reference documents in support of David Brailsford's *Computerphile* videos on parsing. These videos are built around various exercises using the *yacc* parser-generator and *lex* lexical-analyzer generator. The following sections describe the various files that will be found when the archive is extracted.

1. Reference Documents

The reference documents **yacc.pdf** (by Steve Johnson) and **lex.pdf** (by Mike Lesk and Eric Schmidt) are those that accompanied the mid-1970s Bell Labs' release of UNIX Version 7. These are the ultimate 'Reference Manuals' in this collection.

The document **`cphile-parsing.pdf`** attempts to survey the deterministic subset of Chomsky Type 2 grammars and in particular LL(k) and LR(k) parsable grammars. The various simple examples are presented in yacc-compatible syntax, with single-letter alphabetic tokens such as 'a' 'b' and 'c' being sufficient to illustrate grammar differences, stack behaviour and parser actions.

The extra document **`cphile-lexnotes.pdf`** illustrates that **`lex`**'s input format for tokens is based on Regular Expressions (these are equivalent in expressive power to Chomsky Type 3 grammar rules). The **`lex`** example shown in this document creates a recogniser for four possible patterns of **`a`**s and **`b`**s. This recogniser is left in a file named **`lex.yy.c`** which is then compiled with **`gcc`** using the flag **`-ll`** (or **`-lfl`**) to include the correct libraries

The document, **`cphile-yacclex.pdf`** sets up a program to parse requests to add together two integers in roman-number format . The **`lex`** and **`yacc`** scripts for these program are in files **`roman1.l`** and **`roman1.y`** respectively and the command sequence used to get an executable binary was:

```
lex roman1.l
yacc roman1.y
gcc -w y.tab.c -lfl -o roman1
```

2. Running the example programs

The various parsers in this collection, developed from **`lex/yacc`** inputs are: **varnames**, **test**, **roman1**, **roman4**, **furry**, **yoda** and **palxprog** .

We now deal with each of these parsers in turn. The actual executable x86 64-bit binaries (prepared under Open SuSe 13.1) are included. These may well execute correctly, e.g. by typing **./furry**, in other 64-bit Linux environments (but this is not guaranteed). In each case the command sequence that created these binaries is given. Note carefully that for **`lex`**-only binaries the **`lex`**-produced file that gets compiled is **`lex.yy.c`**. In cases where **`yacc`** is also involved, the C program part of the **`yacc`** file will be seen to contain **`#include "lex.yy.c"`** to make sure that the tokenizer front-end is compiled-in with the translated **`yacc`** grammar rules. The ultimate C source code for each parser is written into **`y.tab.c`** and it is this file that gets compiled to produce the parser executable. Note carefully that each use of **`lex`** and **`yacc`** will over-write, respectively, **`lex.yy.c`** and **`y.tab.c`**. It is the user's responsibility to file away 'old' versions of these files if needed.

In the Open SuSe Linux environment used to prepare the distributed files all calls of **`yacc`** get resolved into calls to GNU's **`bison`**, whereas calls to **`lex`** are resolved into calls of Carnegie-Mellon's **`flex`**. When the generated C code for the parser gets compiled the necessary **`flex`** library material is included via a **`-lfl`** flag to the call of **`gcc`**. The **`-w`** flag to

gcc is to suppress warnings about the non-standard use of the function **error** within the C part of **lex.yy.c** or **y.tab.c.**

The first two examples that follow use **lex** only, with no involvement from **yacc**,

2.1. varnames

This example uses **lex**'s Regular Expression capabilities to write recognisers for unsigned integer numbers and for classic program-language identifiers — which must start with a letter, followed by zero or more letters or digits in any combination. Each input is terminated with a newline, **\n**. Use **<ctrl>C** to quit the parser. The **varnames** executable was produced by

```
lex varnames.l
gcc -w lex.yy.c -lfl -o varnames
```

After invoking **varnames** just type in integer numbers or text strings that correspond to REs, in order to receive verification that a match has taken place.

2.2. test

This is another straightforward example using only **lex**. It recognises 4 varieties of RE involving different patterns of the letters **a** and **b**, For each input string **test** will outputs a message saying which of the 4 rules produced a match (or an error message if no match was possible). Compilation took place as follows:

```
lex test.l
gcc -w lex.yy.c -lfl -o test
```

2.3. roman1

This example adds together two Roman numerals but gives the answer as a conventional Arabic number. The addition process is done by a very minimal **yacc** grammar. The recognition of valid Roman numerals is done in **lex** using Regular Expressions. Thus if we execute **roman1** the program will await input such as **mmii + ix** and will produce:

```
mmii    converts to 2002
ix      converts to 9
Answer is 2011
```

Notice that the '+' sign must have spaces on each side of it. The RE recogniser for roman numerals will also accept uppercase letters e.g. MMII Unfortunately valid Roman numerals are supposed to be written in a left-to-right context-sensitive manner such that thousands (**m**) precede hundreds (**c**) precede tens (**x**) and finally units (**i**). Certain allowed abbreviations must also be coped with e.g. **xc** (90), **xl** (40), **ix** (9) and **iv** (4)..

All of which means that **mmii + mcmxliv** will yield the answer 3946. But disappointingly so also will **miim + cmmxliv**, which shows that the tokenizer is capable of letting 'illegal' Roman numbers slip through the net. It is left as an exercise for the interested reader to improve the tokenizer!

Notice that the grammar in the **yacc** input file **roman1.y** relies on being supplied with two tokens, of type **NUMBER**, from the tokenizer, to be added together..

The sequence used to build **roman1** is:

```
lex roman1.l
yacc roman1.y
gcc -w y.tab.c -lfl -o roman1
```

2.4. `roman4`

This example uses the same Roman numbers tokenizer that was used in **`roman1`**. This time the yacc grammar allows the operand between the two roman numbers to be **`+ - *`** or **`/`**. the grammar irself uses left-recursive rules to ensure that left-associativity is enforced and a descending hierarchy (stat, term, primary) of nested definitions for left-hand sides of rules to ensure that **`*`** and **`/`** take precedence over **`+`** and **`-`**.

At the deepest grammar level there is a rule which allows self-embedding recursion **`'(' stat ')'`** to provide for a parenthesised statement to become a low-level ‘primary’. This type of recursion means that this grammar is Chomsky Type 2, rather than lex’s REs (Type 3). Thus the parser that **`yacc`** produces *must* contain a run-time stack.

You will probably want to try out the classic 8 / 4 / 2 associativity test with this grammar, (but remember that your input must be in Roman numerals: you will need to type in **`viii / iv / ii`** which should deliver back the (Arabic numerals) answer of 1. Remember also that *all* operator symbols and parentheses in the input string *must* have spaces on either side of them. Bearing that in mind, you may then want to try (for example)

```
lxiv /  ( vi + ii ) * iv
```

which should yield 32, and

```
mmxix / dclxxiii *  lxiv / ( vi + ii )
```

which should yield 24.

2.5. `furry`

This simple grammar was introduced in the *Computerphile* video entitled *Bottom Up Parsing*. The file **`furry.l`** introduces the 12 fixed strings that are the words and phrases in this grammar giving each word an index number (returned as **`yylval`** to the **`yacc`** parser) and a second number denoting what type of object is being returned (NOUN, VERB, PHRASE etc.). These categories feature prominently in the **`yacc`** parser which reveals their true role as types of object that occur as interior (i.e. non-terminal) nodes in the constructed parse tree.

Notice that the grammar rules, specified inside **`yacc`**, use a syntax that is similar to Backus-Naur form (BNF) but not identical to it. BNF uses angle brackets for non-terminal symbols aand the doesn’t need to quote literal text. By contrast, all entities appearing in **`yacc`** grammars are by default non-terminal symbols which, in turn, then requires that anything that is a literal character, or text string, must be quoted. A PDF of the external (BNF angle bracket) form of this grammar can be found in the file **`furry-grammar.pdf`**

The compilation sequence for this parser was

```
lex furry.l
yacc furry.y
gcc -w y.tab.c -lfl -o furry
```

You can now execute the **`furry`** binary and try the classic input sentence of: **`the robot stroked two furry dice`** Notice that if you try the sentence: **`the robot stroked the robot`** you will see that the robot parses in two different ways depending on whether it is in a subject or object position.

You can keep feeding new test sentences (terminated by a newline) into the parser for as long as you want and, if they match the grammar, you will be told which grammar rules were used. The parser can be terminated with **`&`** on a line on its own, or via CTRL-C.

2.6. **yoda**

This example takes as a starting point the parsed subject-verb-object (SVO) input sentence using the **furry** grammar. It then outputs that same sentence in OSV order (which simulates one of the speech patterns of Yoda the Jedi Master from the *Star Wars* movie :-)

Thus an input of **the robot stroked two furry dice**, once validated, will be transformed on output to **two furry dice the robot stroked**.

The **lex** file for this parser is identical to that for **furry** and indeed the yacc part of the parser begins by verifying that the input is furry-compatible i.e. uses only the the fixed **furry** vocabulary and is in SVO order. The further actions taken are then to store the parsed substrings in the subject, verb and object positions into the respective arrays **subjstring**, **verbstring** and **objstring**. Close inspection of the C actions in this new grammar shows how careful positioning of **strcat** and **strcpy** procedure calls within the subject, verb and object sub-trees can bring about the desired effect.

The compilation sequence for **yoda** was:

```
lex yoda.l
yacc yoda.y
gcc -w y.tab.c -lfl -o yoda
```

Note: the files **lex.yy.c** and **y.tab.c**, in the .zip file, are those that result from the above sequence.

2.7. A final curiosity

This final example is an implementation of a grammar for strings of form $ab^{2n+1}c$ i.e. the letter *a* followed by an odd number of *b*s followed by a *c*. The grammar is analysed in some detail in Appendix A of the file **cphile-parsing.pdf** and its key feature is that it tries to parse the inner substring of *b*s using middle-out, self-embedding, recursion. This strategy is perfectly feasible in principle, because it treats the inner substring as a palindromic substring consisting entirely of *b*s. But successful parsing involves identifying the mid point of the string and how is this to be done when all the *b*s look the same ?!

The compilation sequence is

```
lex paltest.l
yacc paltest.y
gcc -w y.tab.c -lfl -o palxprog
```

When processing this grammar with **yacc**, as above, it emits a warning about a Shift-Reduce conflict and in such cases it takes the Shift option to try and build up a longer handle. Hence the parser built by **yacc** keeps shifting well beyond the mid-point of the *b*s and then crashes. An alternative rule is provided in the grammar which replaces this mid-point by a letter **X**.

Try this parser with the input **abbXbbc** and it parses correctly because the X is a signal to stop stacking *b*s and to match the them with *b*s in the input string, by popping them off the stack. But if the *X* is removed the mid-point is not visible and the parser keeps shifting until it crashes.

For all the above reasons this grammar is of Type 2 (non-deterministic) (see Appendix B of **cphile-parsing.pdf**). In principle a stack suffices as the sole memory resource, but in this particular case one cannot decide, without arbitrary lookahead, when to stop shifting and start reducing.