Parser Design

Neil Mitchell

June 25, 2004

1 Introduction

A parser is a tool used to split a text stream, typically in some human readable form, into a representation suitable for understanding by a computer. Usually this representation will be either a stream of tokens, or some tree structure, such as the abstract syntax tree for a programming language. There are many parser tools in existence, but by far the most well known are Lex [**?**] and Yacc [**?**], and their open source alternatives Flex [**?**] and Bison [**?**]. For the purposes of this paper, I will use the initial names to refer to either tool, as the latter ones are reverse compatible with the former.

2 Review of Lex and Yacc

Lex is a lexical analysis tool based on the theory of regular expressions, taking a stream of text and transforming it to a stream of tokens. Yacc is a context free grammar based parser, taking a stream of tokens and producing a tree of tokens. Typically a stream would be parsed once with Lex, and then the lexemes generated would be parsed with Yacc. These tools are designed to fit together in a natural manner, with similar conventions to each other, and an easy mechanism for their combination.

Both Lex and Yacc take as input a syntax file describing the language, and produce as output a C file that can be used to parse the language. A compiler is then needed to turn this C file into executable code, and parse the instructions embedded in the language description which are included in C code.

2.1 Typing

Viewing the output of both Lex and Yacc as a function of the input allows a type to be given to each end. For Lex and Yacc the appropriate types would be:

$$\text{Lex} : \text{String} \mapsto \text{List(Token)}$$
$$\text{Yacc} : \text{List(Token)} \mapsto \text{Tree(Token)}$$

This shows that applying Lex then Yacc will generate a type of Tree(Token) from String – being the aim of a parser.

2.2 Language Dependence

One negative aspect of Lex and Yacc is their dependance of the C language. In response to matching a token arbitrary embedded C commands are executed, which could perform any possible action. This means that the language definition also contains a particular implementation of that language, and hence cannot be reused for another purpose. This also ties the parser to C, requiring a working knowledge of C to write the language. In addition since the language must be compiled with the C compiler, this means any project making use of the compiler must be C compatible – typically meaning either C or C++.

Since the C language is so intertwined with Lex and Yacc, it is quite understandable that alternative implementations exist which tie the parser to a different language. Some examples of an equivalent to C with Yacc include Java with Jay [**?**] and Haskell with Happy [**?**]. Interestingly enough, while many languages have cloned Yacc relatively few have cloned Lex but rely on hand coded versions to split a string into a list of tokens.

2.3 Class of languages

The classes of languages that can be dealt with by bison and flex are those which are in LALR(1). This is a pretty large subset of CFG's. Because additional statements in C can be used, which is a Turing complete language, in fact any recursively enumerable language can be described – although anything more than LALR(1) is quite hard to do.

2.4 Yacc internal details

In order to effectively design a Yacc grammar, it is very often necessary to understand the internal details behind LALR(1) parsers. Often a Yacc file will give errors such as "shift/reduce conflict" while is meaningless to anyone who does not understand parsers to a significant depth. This means that to write a Yacc parser experience in parser implementation is required, which otherwise would be unnecessary.

2.5 Multiple passes

The Lex and Yacc approach involves splitting the parsing up into multiple sections, which can then be combined. The advantage of having multiple stages is that any one can be replaced by a custom coded version, and the stages can be used individually.

With this system, the split is done very much in a Chomsky language hierarchy manner. While this makes sense from a language design point of view, the end user should not need to appreciate context free grammars to implement a parser. It is possible that an alternative split would give a more intuitive language – although any such comparison would be highly subjective.

2.6 Performance

The algorithm used to implement Lex runs in $O(n)$ where n is the length of the input string. The algorithm for implementing Yacc runs in approximately $O(n)$ where n is the number of lexemes generated by Lex. This gives very good performance, and in practice is more than satisfactory for most tasks.

3 Design Discussion

In order to think about the appropriate primitives in a parsing architecture it is necessary to think what things require parsing. Since more and more languages have built in regular expression features, such as Perl [**?**], only for things which fall outside this scope are likely to cause users to turn to a more powerful tool. While some implementations of regular expressions are incredibly powerful, one particular area on which they have problems is bracketing, particularly nested brackets. The other problem is the lack of readability, with too much information encoded as special characters.

When thinking of what to parse, it is useful to give instances of things that do require parsing regularly, and to make these kinds of things easy to write definitions for. For this paper, the thoughts include the programming languages C# [**?**] and Haskell [3], XML [**?**], and finally the language definitions which are created for this tool. These languages are picked to give a wide range of cases, including both human readable and more machine readable languages, and brace and indent delimited structures.

Some particular examples that are *not* included in this list are HTML [**?**], LaTeX[**?**] and C++ [1]. These are deliberately left out because they are older languages and have many quirks that prevent existing parsers from working well on them. This includes highly context sensitive areas, the ability to redefine concepts during the parsing, non-conforming input etc.

3.1 Brackets

The first thing that is hard to define traditionally is bracketing. Often the bracketing is some of the most important structure in a program, however using Yacc this can be split over many lines and is not made explicit. The bracketing phase is also interspersed with determination of the content, making the allowable nesting of brackets a hard thing to determine. In reality this is one of the core concepts underlying any language, and one of the first things a language designer would specify.

3.2 Escape Characters

The next complex task is escape character, although their complexity is largely masked by Lex. In most languages a string is both started and ended with the quote character. However, quote characters may also appear within the string, and in this case the quote needs to be treated specially. Usually Lex is instructed to skip the escape character, however the position and meaning of this escape character is not recorded – meaning the string must be parsed again later. Obviously this duplication is bad, and can lead to different interpretations at different points.

One way to view a string, is as a bracket that does not allow nesting. When viewed this way an escape character could be considered as a special token within a string – not part of a continuous character stream.

3.3 Dangling `else` statements

These generate warnings in Yacc, and warnings should either be able to be suppressed, or fixed. Having them as standard is just bad.

4 Design

The parsing scheme which I propose has three stages comprising of the following types.

$$\text{Bracket} : \text{String} \mapsto \text{Tree(Token)}$$
$$\text{Lex} : \text{String} \mapsto \text{List(Token)}$$
$$\text{Group} : \text{Tree(Token)} \mapsto \text{Tree(Token)}$$

For my scheme, the Bracket stage splits the text into sections, typically delimited by brackets, and then sends each sections raw text to Lex. This means that many different Lex schemes may be used within one single language definition. The Lex operation in my implementation is identical to that in traditional parsing – however while Lex would usually focus mainly on thing such as strings and escape characters, this is now dealt with by Bracket making Lex considerably simpler.

The final operation called Group is one for which there is no parallel in the original parsing architecture – group is about taking a tree of tokens and grouping sections to make more intuitive data structures. Group is not a CFG, but is simply a regular expression engine in the same spirit as Lex but applied to tokens instead of characters. Using this regular expression idea, many of the problems with Yacc evaporate, for example the `else` statement now is greedily eaten as with Lex expressions.

4.1 Output

All stages generate lists of Named elements. A named element has a Name, and extra fields depending on its type. An object generated by Lex would have a Text field. An object generated by Group would have a children property, comprising of a list of Named objects. An object generated by Bracket would have children, along with a start and stop text.

The output is defined as above, but the exact implementation is not. The actual output will vary – when precompiled the names may correspond to types, when loaded dynamically they could correspond to strings. Different languages would represent different elements in different ways – for example C# would use named attributes in a class, while Haskell would use the position of arguments within a list.

The one output that is defined concretely is the XML serialisation. This can then be used as an interchange language when parsers between different languages wish to communicate.

4.2 Bracket

In order to introduce the syntax and semantics of the Bracket stage, I present below a bracket definition for all the parser syntax files.

```
# Main item
main = 0 * blankline commentline line [lexer]

# Individual lines
!blankline = "\n" 0
!commentline = "#" "\n"
line = 0 "\n" comment regexp string round square curly
```

```
# Non-nesting elements
regexp = "‘" "‘" escape [raw]
string = "\"" "\"" escape [raw]
!comment = "#" !"\n"
escape = "\\" 1

# Nesting elements
square = "[" "]" curly square round regexp string
round = "(" ")" curly square round regexp string
curly = "{" "}" curly square round regexp string
```

The Bracket file is structured as a list of rules, one per line. In addition there are comments prefixed with `#` and blank lines. Each rule has the name of the rule, an equals sign, and then a description of the start and end tokens. After this comes a list of the nested elements, and optionally the lexer to use in square brackets. The initial rule invoked is that named main.

The exclamation character `!` at the start of the line, for example on `commentline`, means that this token is deleted from the output. The match element is either a literal string, a length or a star to indicate that the match is at the end of the string.

The lexer specified is used to parse the text matched by this bracket. If no lexer is specified, then the lexer from the enclosing element is used. The predefined lexers raw and none are defined automatically, raw returns the text as a lex object, and none returns no elements.

4.3 Lex

A lex file for parsing the various parser elements is given, which would be named lexer in the Bracket file given above.

```
# Lexical elements
keyword = ‘[a-zA-Z]+‘
number = ‘[0-9]+‘
!white = ‘[ \t]‘

# Individual characters
star! = "*"
ex! = "!"
eq! = "="
colon! = ":"
quest! = "?"
plus! = "+"
```

The lexical rules feature both regular expressions in back quotes ‘, and standard strings in normal quotes ". Names may have either exclamations in front of them or after them, before causes the entire token to be deleted from the output, after keeps the token but deletes its contents – in effect just an optimisation.

4.4 Group

The group present here is for the Lex syntax, and would first have been parsed by the Bracket file described initially.

```
root = main![*{rule range}]

rule = line![ ?ex keyword ?ex !eq {regexp string} ]
range = line![ keyword !colon string ]
```

This definition starts with the root keyword, being the equivalent to main for Bracket. Each production must then follow on from the root, to build a tree where the root element is root. Where square brackets are used, this indicates the child nodes of a Bracket object. Each element may be proceeded by any one of question mark, star and a plus sign, indicating how many of the element follow. If a curly brace is used to group many elements, then this represents choice between the elements – any one may follow. Note that only single elements may be chosen between, this is to create a readable output format with everything uniquely named.

An exclamation character before a keyword deletes that token from the output, and an exclamation character after a token but before a square bracket promotes the inner tokens. For example in the above line named rule, the line token will be removed but all the children of line will become children of rule.

5 Implementation

When implementing this new parser system, each element can be implemented individually, or all 3 systems could be implemented in one unit. The exact implementation is not covered in this initial version of the paper, however future versions are likely to contain fast and efficient algorithms in detail.

5.1 Bracket

The Bracket program can be implemented as a deterministic push down stack finite state automaton. This will give $O(n)$ performance, where n is the length of the input.

5.2 Lex and Group

Both of these operations are regular expression, with the former working on strings and the latter on terminals. As such both can be implemented in $O(n)$.

5.3 Memory Usage

One point to note about this new system is that the only text in the output is text that can be directly taken from the input – if the input is stored in memory, then the output can be directly used from this store, without reallocation.

5.4 Overall

Because each component is $O(n)$, the total time to parse is therefore also $O(n)$, which beats Lex and Yacc as a combination.

6 Interaction

This section discusses the interaction the parser can have with other components in a software program. The traditional model of Lex and Yacc is to generate C source code, and many similar parsers also adopt this approach. However, this clearly derives from very old times, when a command line or Make was used to compile programs. In a modern architecture, this may be appropriate, but options such as resource files may also be useful to consider.

6.1 Token Output

The final advantage of Lex/Yacc is that tokens are passed to the internal language structure as and when they are generated – meaning that a program does not have to build an entire abstract syntax tree in memory – although many still do. The way to overcome this with my parser would be to implement a SAX [**?**] style interface as with XML, passing each token to a procedure as it is generated. This can then be defined in an independent way from the grammar.

6.2 Code Generation

The standard method to embed a parser in a source file is by generating source code.

6.3 Resource File

Possibly a more appropriate mechanism would be to define a custom file representation for a parser, and include this as a resource file. This file could then be loaded by the parser API, and used directly. This allows the parser to be reused without even recompiling it for alternative languages.

6.4 Dynamic Parser Generation

An appropriate API could be provided that would load a Bracket, Lex and Group specification and then parse things using this definition. This would require compiling the parser on the fly, and possibly some method of caching the compiled version. This could then be used in programs such as text editors which provided custom syntax colouring.

7 Extensions

Despite the power of the parser, there are still going to be things that cannot be parsed correctly – particular examples being the indentation delimiting by Haskell, and the C++ `#include` statement. These are traditionally handled in custom code when using Lex and Yacc, and custom code again appears to be appropriate in these situations.

The extension proposed is to allow Bracket to invoke methods in the programming language, however in a restricted manner. Standard methods such as indent and include could be reserved, however an extension method would allow people requiring custom grammars to write new ones.

This would typically be done at a very early stage of the parsing, and in a controlled and functional manner (i.e. no side effects to the operation).

While this extension would damage language operation to some degree, it is a small price to pay for the increased usage. However, by discouraging this practice where not essential, it is hoped that language constructs will be kept to a minimum.

8 Conclusion

Currently, when writing two parsers, generally two entirely different sets of Lex and Yacc need to be written. In some cases, the Lex file may be shared or partially shared between two files, but the Yacc file is almost always customised.

With this new scheme, the Lex and Bracket stages could be written in a generic enough manner to allow reuse between many related languages, for example C# and Java. This would allow a library of parsers to be written. In addition, the removal of language specific features allows the specification of a language such as C# to be used in both C and C# by for example the cooperative open source efforts to clone C#.

References

[1] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Programming Languages - C++ ISO/IEC 14882:1998(E)*, September 01 1998.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, second edition, 2000.

[3] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, first edition, 2003.