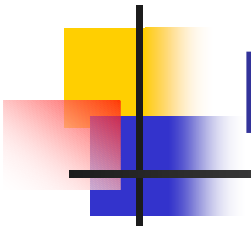
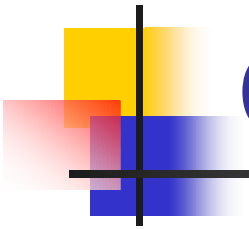
A New Parser

Neil Mitchell



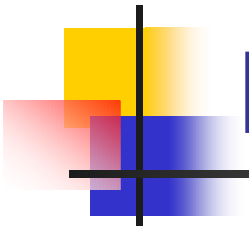
Disclaimer

- This is not something I have done as part of my PhD
- I have done it on my own
- I haven’t researched other systems
- Its not finished
- Any claims may turn out to be wrong



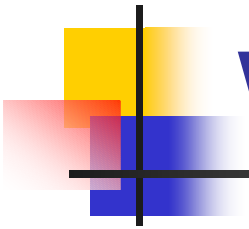
Overview

- What is parsing?
- What systems exist?
- What do you want to parse?
- What is my system?
- Why is mine better (or not)?
- How do you interact with a parser?



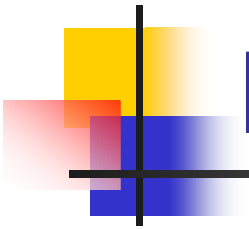
Parsing is a function

- From Text to Abstract Syntax Tree
- Parse :: String -> Tree(Token)
- How?
  - Hand coded
  - Using Lex/Yacc (or Flex/Bison)
  - Parser combinatiors
  - Etc...



What Systems Exist?

- Lex/Yacc – the classic
- Created to write parsers for C
- Steps
  - Write a grammar file, including C code
  - Generate a C file
  - Compile C file
  - Link to your code

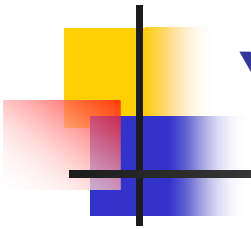


Lex

- Lex :: String -> List(Token)

Uses Regular Expressions to split up a string into various lexemes.

Runs in *O(n)*, using Finite State Automata.



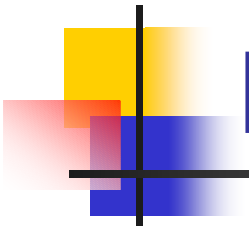
Yacc

- Yacc :: List(Token) -> Tree(Token)

Based on a BNF grammar.

Runs in just over *O(n)*, using an LALR(1) stack automaton.

Often fails unpredictably...

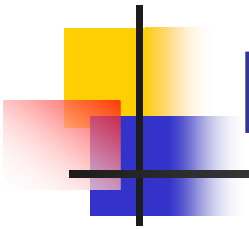


Early

- Early :: List(Token) -> Tree(Token)

Almost identical to Yacc, but removes the unpredictable failures, requiring less knowledge of LALR(1)

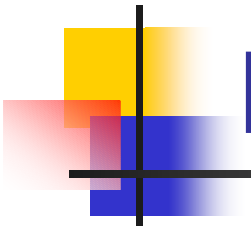
A fair bit slower, worst case of $O(n^3)$ or $O(n^{2.6})$ depending on implementation.



Parser

- ParserC = Yacc . Lex
- ParserHaskell = Happy . Alex
- ParserJava = ...

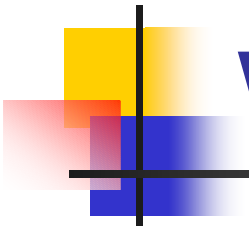
Very language dependant, Yacc/Lex both tied to C



Bad points

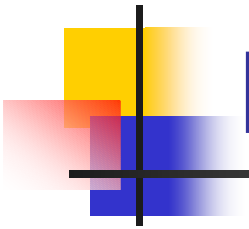
- Language dependant
- Yacc – shift/reduce conflict
- Not CFG
- Not very intuitive to write Yacc

- Summary: Lex good, Yacc bad



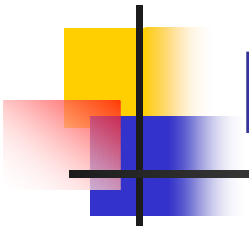
What do you want to parse?

- Languages: Haskell, C#, Java
- Configurations: INI, XML
- Grammar files for this tool
- NOT: Perl, Latex, HTML, C++
  - Insane syntax
  - Horrid history
  - Twisted parody of languages



Brackets, Strings, Escapes

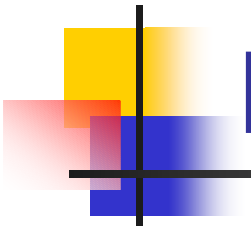
- Brackets () [] {} <> - Yacc
- Strings “” ‘’ – Lex
- Are strings not brackets, just which disallow nesting?
- What about escape characters?
  - Parse them in Lex: “((\.)|.)*”
  - Re-parse them later



My System

- Bracket :: String -> Tree(Token)
- Lex :: String -> List(Token)
- Group :: Tree(Token) -> Tree(Token)

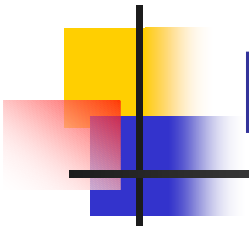
- Parser :: String -> Tree(Token)
- Parser = Group . map Lex . Bracket



Bracket

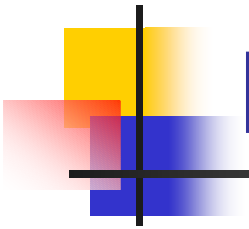
- Match brackets, strings, escape chars
- Define nesting

```
main = 0 * all [lexer]
all : round string
round = "(" ")" all
string = "\"" "\"" escape [raw]
escape = "\\" 1
```



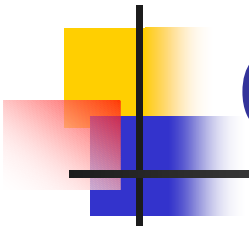
Lex

- Same as traditional Lex, but...
- Easier – no need to do string escaping
- Can be different for different parts
  - In comments use [none]
  - In strings use [raw]
  - Can have many lexers for different parts



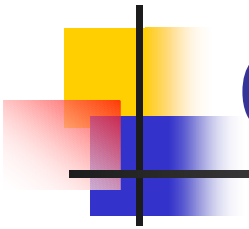
Lex (2)

```
keyword = `[a-zA-Z][a-zA-Z0-9_]*`
number = `[0-9]+`
white = `[ \t]`
star = "*"
eq = "="
for.
while.
```



Group

- Group :: Tree(Token) -> Tree(Token)
- Id :: a -> a
  - Therefore "Group = Id" works
- Sometimes you need a higher level of structure, what the brackets mean
- The most complex element (unfortunately)

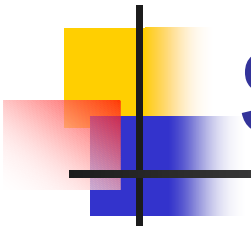


Group (2)

root = main[*{rule literal}]

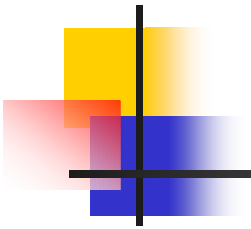
rule = line[keyword eq {regexp string}]

literal = line[keyword dot]



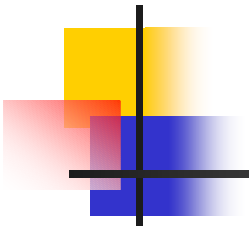
Summary of BLG

- Complete lack of embedded C/Haskell
- Data format defined generically
  - Can be Haskell linked list
  - Can be C array
  - There is an XML format defined
- Similar in style to each other
- All "simple" langauges



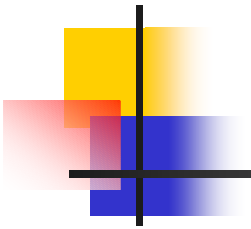
Implementation

- Bracket
  - Deterministic Push down stack automata
- Lex
  - Steal existing lex, FSA
- Group
  - FSA? Maybe...
  - Have a sketched automaton



Implementation (2)

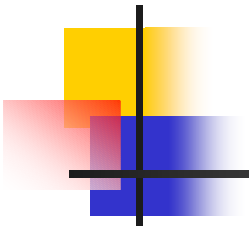
- I have implemented most of it in C#
- Slow, but very useable
- Bracket seems pretty perfect
- Lex uses Regex objects, but works
- Group is less complete, uses backtracking, doesn't have maximal munch semantics, NP, etc.



Implementation (3)

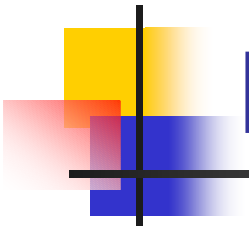
- BLG is self-parsing ☺
- 1 Lex file for all 3
- 1 Bracket file for all 3
- 3 Group files, one each

- Reuse is good



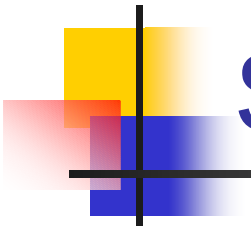
Interaction

- How do you interact with a parser?
- Yacc/Lex
  - Translate, Compile, Link, Execute
- BLG
  - Translate, Compile, Link, Execute
  - Compile into resource file
  - Load at runtime (Text Editors)



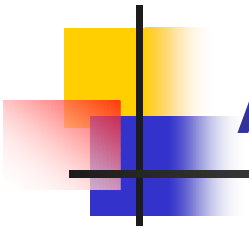
Exclamations!

- BLG defines a complete set of exclamations which allow for code hoisting and deleting
- Remove tokens from the output (white space/comments)
- Promote tokens, i.e. line![x] returns x
- Simple, but ignored here



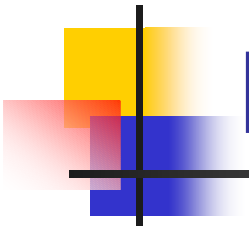
$Directives

- In the Bracket, before any processing
- Stream processing directives
- $text (remove '\r', append '\n')
- $tab-indent (for Haskell/Python)
- $upper-case
- Easy, simple, generic, reusable



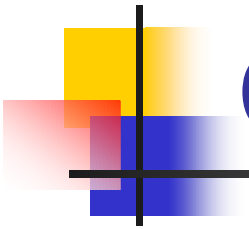
Advantages

- Language neutral
- Haskell parsing
  - GHC in Haskell
  - Hugs in C
  - Could now use the same grammar
- Can reuse elements, i.e. Lex and Bracket are almost identical for C#/Java



But best of all

- The grammars are really easy to specify
  - A bit of a leap
  - Would need years of hypothesis testing
  - And maybe even a working implementation
- Faster
  - Almost irrelevant, thanks to faster computers



Questions?

What did I explain badly?

I would really appreciate any feedback!

Should I ditch the entire idea?

Should I implement it?

Should I give up my PhD to sell this system?