Parsing from Scratch

(Tunight Talk)

Paul Zhu

May 25, 2019

Parsing is Everywhere

- Compilers
- Regular expressions
- Text processing
- Deserialization
- etc.

Theory & Practice Combined

In my last Tunight talk:

> *Programming Languages (PL) is one of the most* theoretical *topic in computer science, and it is also one of the most* practical *field targeting at software and system engineering.*

In parsing, theory and practice are combined – things get worked and many of us know why.

Parsing Techniques in Mainstream

"I give you a grammar, you give me a parser." – Parser generator

"Why not express parsers in terms of a couple of our favorite combinators, and better be monadic?" – Parser combinators

Today

1. Parser Generator
2. Parser Combinators
3. Parsing, the Future
   - Typed Parsing
   - Automated Parsing

Contents

Off-the-shelf Tools

- `lex`, `jflex`, `ocamllex`, `alex`
- `yacc`, `byacc`, `ocamlyacc`, `happy`
- `Antlr`

What does YACC mean?

- Yet another Cartesian Cubical? [1]
- Yet another compiler compiler!

[1] `https://github.com/mortberg/yacctt/`

When Building your own Language ...

What's the dragon book? The dragon book is all about parsing... It turns out almost everything about this is about syntax.

What does Antrl do? Antrl is a parser. It's an awesome compiler compiler, *what can be better than that? That's a* fancy name for parser*!*

Building a language is basically about defining the grammar and adding production rules to this grammar... This is exactly the kind of nonsense *that leads to* badly defined languages*...*

When Building your own Language ...

So at least for the duration of this course I'm not gonna allow you to do this kind of nonsense. I'm offering you a recipe thinking about language design. Think about what features you have, what features you want, what features you will end up being forced to have by your user bases if you are ever successful.

Write a little interpreter, play with the core of the language, and understand the consequences ... Now you go to Stackoverflow and say I've already got my language, and how do I build the language part of my language.

– Shriram Krishnamurthi ("Though my head is often in security, networking, formal methods, and HCI, my heart is in programming languages"). In video `https://www.youtube.com/watch?v=3N__tvmZrzc`*, 38:50 – 41:40.*

Foundation of Parser Generators

Formal language and automata theory, which every CS undergraduate student learns!

In detail:

- LL(1)
- LL(k)
- LR(1), SLR(1), LALR(1), etc.

Demonstration

- LL(1) Parser Generator:
  `https://github.com/paulzfm/LL1-Parser-Gen`

Contents

Why Parser Combinators?

- I don't want to learn any new tools.
- My grammar is changing all the time, à la agile.
- I hope to dynamically construct parsers in runtime.

Yacc is dead?

`http://matt.might.net/articles/parsing-with-derivatives/`

What is Indeed a Parser?

I give you a source (e.g. a char stream), you give me back a parsing result (e.g. a parsing tree), which may fail, together with the remaining source that is not consumed.

Parsec

- 1 parsec $\approx$ 3.26 light-years [2]
- A parser combinator library!

[2] `https://en.wikipedia.org/wiki/Parsec`

Demonstration

- `scala-combinator-library`
- Haskell `Parsec`

Combinators Summary

```
eof      :: Parser ()
char     :: Char -> Parser Char
(<|>)    :: Parser a -> Parser a -> Parser a
choice   :: [Parser a] -> Parser a
try      :: Parser a -> Parser a
many     :: Parser a -> Parser [a]
many1    :: Parser a -> Parser [a]
sepBy    :: Parser a -> Parser sep -> Parser [a]
between :: Parser open -> Parser close -> Parser a -> Parser a
optionMaybe :: Parser a -> Parser (Maybe a)
```

Monadic Operations

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
(>>)   :: Monad m => m a -> m b -> m b
```

A parser is a monad!

Contents

Contents

Motivation: "Bizarre" Behaviors of Parsec

```
p1 = (char 'a' >> return 1) <|> (char 'a' >> return 2)
parse p1 "" "a"

p2 = (optionMaybe (char 'a')) >>= \x ->
        (optionMaybe (char 'a')) >>= \y ->
            return (x, y)
parse p2 "" "a"

p3  =  (eof >> return [])
   <|> (p3 >>= \xs -> char 'a' >>= \x -> return $ xs ++ [x])
parse p3 "" ""
parse p3 "" "aa"

p4  =  (char 'a' >> char 'b' >> return 1)
   <|> (char 'a' >> char 'c' >> return 2)
parse p4 "" "ab"
parse p4 "" "ac"
```

Combinators as a Formal Language

Grammar/Expression $g ::= \bot \mid \varepsilon \mid c \mid x \mid g \vee g' \mid g \cdot g' \mid \mu x.g$

where

- $c \in \Sigma$, a finite set of characters
- $v \in V$, a countably infinite set of variables
- Kleene star is definable by means of a fixed point:

$$g^* = \mu x.(\varepsilon \vee g \cdot x)$$

This part is based upon: Neelakantan R. Krishnaswami, Jeremy Yallop. A Typed, Algebraic Approach to Parsing. PLDI'19.

Ambiguity in Predictive Parsing

- (Disjunctive non-determinism) When parsing a string w against a grammar of the form $g_1 \vee g_2$, then we have to decide whether w belongs to g_1 or g_2.
- (Sequential non-determinism) When parsing a string w against a grammar of the form $g_1 \cdot g_2$, we have to split it into two pieces w_1 and w_1 so that $w = w_1 \cdot w_2$ where w_1 belongs to g_1 and w_2 belongs to g_2.

Q: How can we solve these problems?

Eliminating Ambiguities

$$
\begin{aligned}
\mathit{null}(L) &= \varepsilon \in L \\
\mathit{first}(L) &= \{c \mid \exists w \in \Sigma^*. c \cdot w \in L\} \\
\mathit{flast}(L) &= \{c \mid \exists w \in L \setminus \{\varepsilon\}, w' \in \Sigma^*. w \cdot c \cdot w' \in L\}
\end{aligned}
$$

Eliminating disjunctive non-determinism: their first character need be disjoint.

Lemma 1

If $\mathit{first}(L) \cap \mathit{first}(M) = \varnothing$ *and* $\neg(\mathit{null}(L) \wedge \mathit{null}(M))$, *then* $L \cap M = \varnothing$.

Eliminating sequential non-determinism: the split need be unique.

Lemma 2

If $\mathit{flast}(L) \cap \mathit{first}(M) = \varnothing$ *and* $\neg\mathit{null}(L)$ *and* $w \in L \cdot M$, *then there are unique* $w_L \in L$ *and* $w_M \in M$ *such that* $w_L \cdot w_M = w$.

Do it the PL Way!

- A grammar/expression can be regarded as programs.
- We only expect a part of the programs which have "good" properties.
- Thus, we specify a type system that is sound, i.e., every well-typed program do have the "good" properties.

Nowadays, PL is almost about types!

Types: Motivation

$$\text{Type } \tau :: \{\text{null} :: \text{Bool}, \text{first} :: \mathcal{P}(\Sigma), \text{flast} :: \mathcal{P}(\Sigma)\}$$

We expect the types to overapproximate the properties of the language, and we define the satisfaction judgment $L \models \tau$ as

$$L \models \tau \iff \mathit{null}(L) = \tau.\text{null} \wedge \mathit{first}(L) \subseteq \tau.\text{first} \wedge \mathit{flast}(L) \subseteq \tau.\text{flast}$$

Types Definition

$$\tau_\perp = \{\text{null} = \text{false}, \text{first} = \varnothing, \text{flast} = \varnothing\}$$

$$\tau_\varepsilon = \{\text{null} = \text{true}, \text{first} = \varnothing, \text{flast} = \varnothing\}$$

$$\tau_c = \{\text{null} = \text{false}, \text{first} = \{c\}, \text{flast} = \varnothing\}$$

$$\begin{aligned}\tau_1 \vee \tau_2 = \{&\text{null} = \tau_1.\text{null} \vee \tau_2.\text{null}, \text{first} = \tau_1.\text{first} \cup \tau_2.\text{first},\\ &\text{flast} = \tau_1.\text{flast} \cup \tau_2.\text{flast}\}\end{aligned}$$

$$\tau_1 \oplus \tau_2 \triangleq (\tau_1.\text{first} \cap \tau_2.\text{first} = \varnothing) \wedge \neg(\tau_1.\text{null} \vee \tau_2.\text{null})$$

$$\tau_\perp \cdot \tau = \tau_\perp = \tau \cdot \tau_\perp$$

$$\begin{aligned}\tau_1 \cdot \tau_2 = \{&\text{null} = \tau_1.\text{null} \vee \tau_2.\text{null},\\ &\text{first} = \tau_1.\text{first} \cup \text{if } \tau_1.\text{null} \text{ then } \tau_2.\text{first} \text{ else } \varnothing,\\ &\text{flast} = \tau_2.\text{flast} \cup \text{if } \tau_2.\text{null} \text{ then } \tau_2.\text{first} \cup \tau_1.\text{flast} \text{ else } \varnothing\}\end{aligned}$$

$$\tau_1 \circledast \tau_2 \triangleq (\tau_1.\text{flast} \cap \tau_2.\text{first} = \varnothing) \wedge \neg\tau_1.\text{null}$$

$$\tau^* = \{\text{null} = \text{true}, \text{first} = \tau.\text{first}, \text{flast} = \tau.\text{flast} \cup \tau.\text{first}\}$$

Typing Judgment

$$\text{Context } \Gamma, \Delta ::= \bullet \mid \Gamma, x : \tau$$

The main typing judgement has the form

$$\Gamma; \Delta \vdash g : \tau$$

is read as "under ordinary hypotheses Γ and guarded hypotheses Δ, the grammar g has the type τ", where the guarded variables are those which cannot occur at the head of a string.

Example 1

If $x : \tau$ is guarded, the grammar x is considered ill-typed, but $c \cdot x$ is permitted.

Typing Rules

$$\boxed{\Gamma; \Delta \vdash g : \tau}$$

$$\text{T-Bot}\ \frac{}{\Gamma; \Delta \vdash \bot : \tau_\bot} \qquad \text{T-Eps}\ \frac{}{\Gamma; \Delta \vdash \varepsilon : \tau_\varepsilon}$$

$$\text{T-Char}\ \frac{}{\Gamma; \Delta \vdash c : \tau_c} \qquad \text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma; \Delta \vdash x : \tau}$$

$$\text{T-Union}\ \frac{\Gamma; \Delta \vdash g_1 : \tau_1 \qquad \Gamma; \Delta \vdash g_2 : \tau_2 \qquad \tau_1 \oplus \tau_2}{\Gamma; \Delta \vdash g_1 \vee g_2 : \tau_1 \vee \tau_2}$$

$$\text{T-Concat}\ \frac{\Gamma; \Delta \vdash g_1 : \tau_1 \qquad (\Gamma, \Delta); \bullet \vdash g_2 : \tau_2 \qquad \tau_1 \circledast \tau_2}{\Gamma; \Delta \vdash g_1 \cdot g_2 : \tau_1 \cdot \tau_2}$$

$$\text{T-Fix}\ \frac{\Gamma; \Delta, x : \tau \vdash g : \tau}{\Gamma; \Delta \vdash (\mu x : \tau . g) : \tau}$$

Remark: In T-Fix, the guarded x ensures that no left-recursive definitions are typeable.

Contents

Log Parser

```
/*  A logging code snippet extracted from:
    hadoop/hdfs/server/datanode/BlockReceiver.java */

LOG.info("Received block " + block + " of size "
    + block.getNumBytes() + " from " + inAddr);
```

Log Message

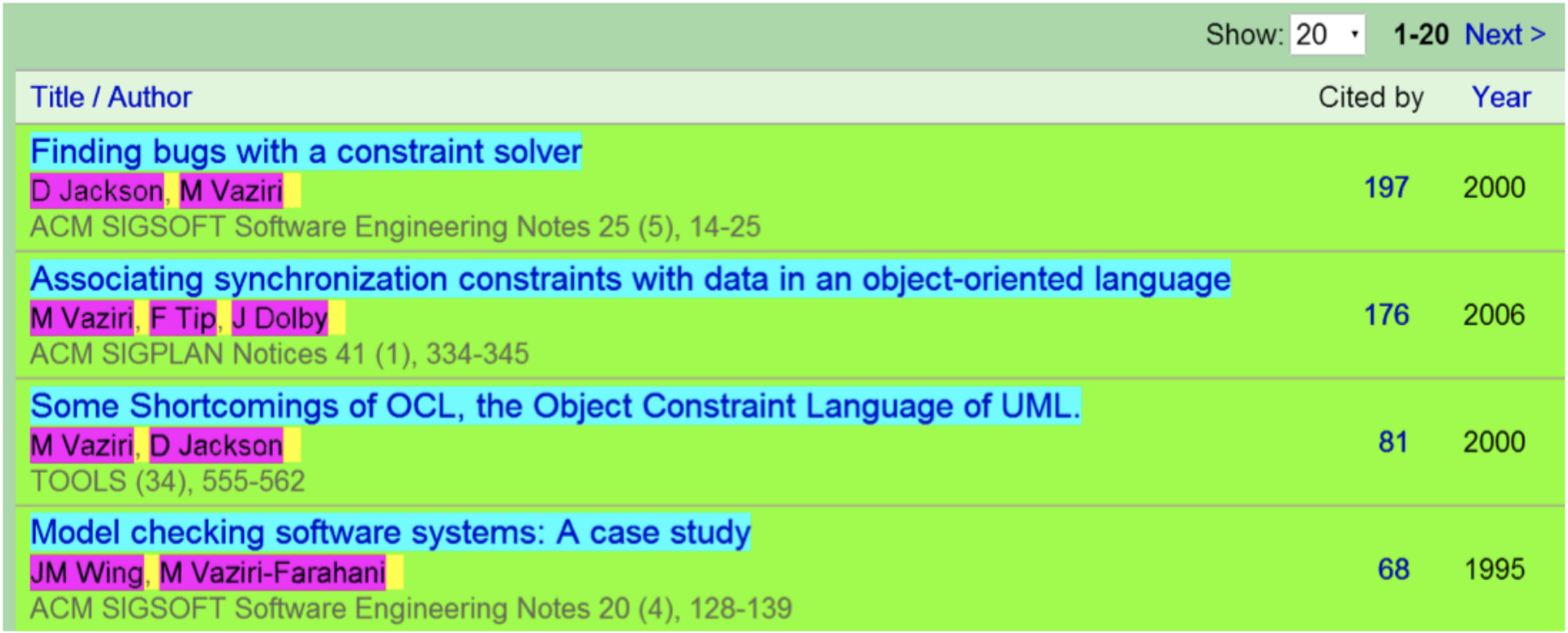
```
2015-10-18 18:05:29,570 INFO dfs.DataNode$PacketResponder: Received
block blk_-5627252808530876 85 of size 67108864 from /10.251.91.84
```

Structured Log

TIMESTAMP	2015-10-18 18:05:29,570
LEVEL	INFO
COMPONENT	dfs.DataNode$PacketResponder
EVENT TEMPLATE	Received block <*> of size <*> from /<*>
PARAMETERS	["blk_-5627252808530876 85", "67108864", "10.251.91.84"]

Jieming Zhu, Shilin He, Jinyang Liu, et al. Tools and Benchmarks for Automated Log Parsing. ICSE'19. https://github.com/logpai/logparser

FlashExtract



Show: 20 1-20 Next >

Title / Author	Cited by	Year
Finding bugs with a constraint solver D Jackson, M Vaziri ACM SIGSOFT Software Engineering Notes 25 (5), 14-25	197	2000
Associating synchronization constraints with data in an object-oriented language M Vaziri, F Tip, J Dolby ACM SIGPLAN Notices 41 (1), 334-345	176	2006
Some Shortcomings of OCL, the Object Constraint Language of UML. M Vaziri, D Jackson TOOLS (34), 555-562	81	2000
Model checking software systems: A case study JM Wing, M Vaziri-Farahani ACM SIGSOFT Software Engineering Notes 20 (4), 128-139	68	1995

Vu Le, Sumit Gulwani. FlashExtract: A Framework for Data Extraction by Examples. PLDI'14.

AlphaRegex

Description	
Strings have at most one pair of consecutive 1s	
Examples	
Positive	Negative
0 11 101010 00011000 00100110001	111 110011 0110110 00011001100 011100011110
Answers	
Students	A variety of uncertain answers
ALPHAREGEX	(1?0)*1?1?(01?)*

Mina Lee, Sunbeom So, Hakjoo Oh. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. GPCE'16.

Parsify

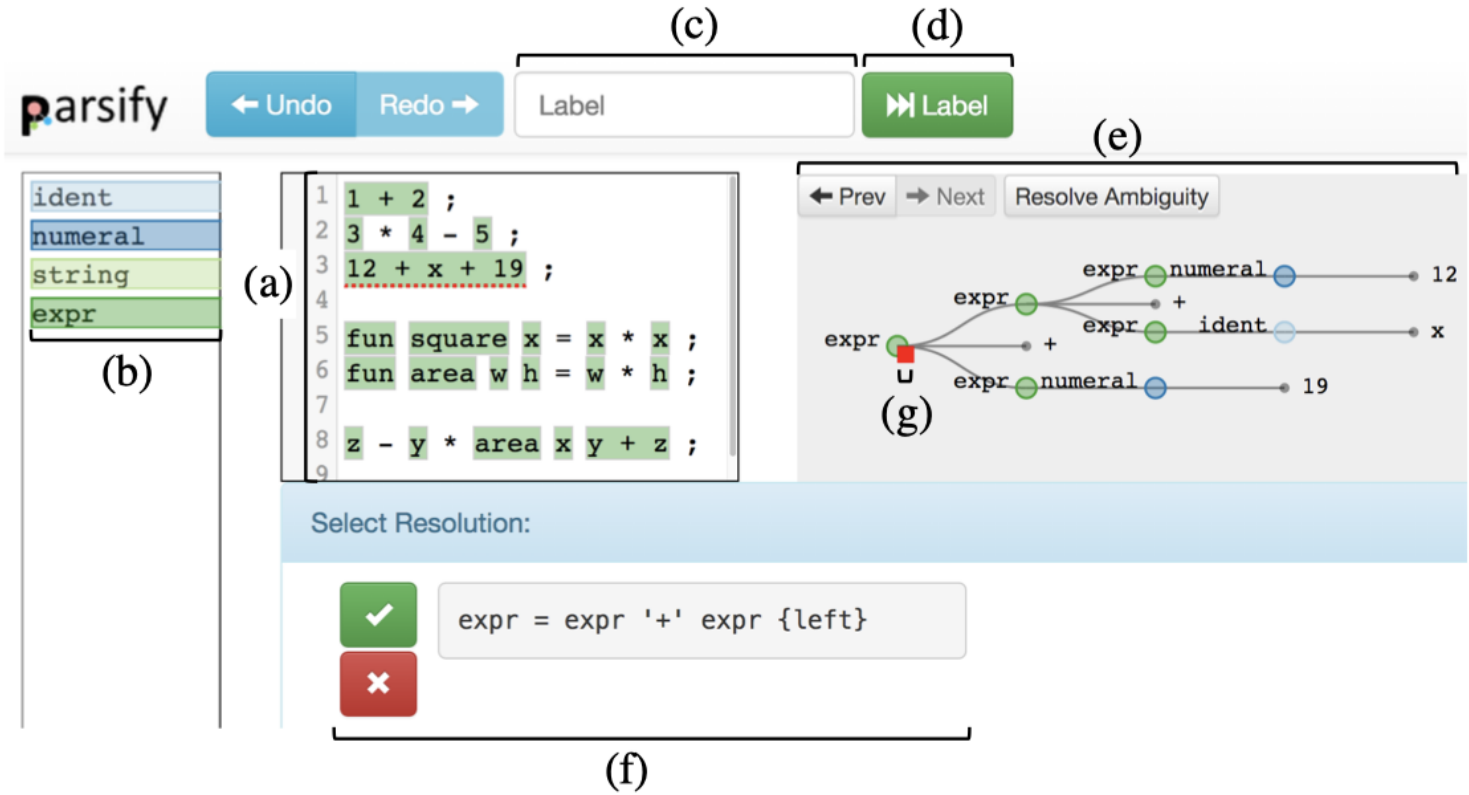


Figure 1. The Parsify user interface: (a) File View, (b) Legend, (c) Label Box, (d) Label Button, (e) Parse Tree Pane, (f) Resolution Pane, and (g) Negative Label

Alan Leung, John Sarracino, Sorin Lerner. Interactive Parser Synthesis by Example. PLDI'15.

Glade

- Target language $\mathcal{L}(C_{\text{XML}})$, where the context-free grammar C_{XML} has terminals $\Sigma_{\text{XML}} = \{\texttt{a}, \ldots, \texttt{z}, \texttt{<}, \texttt{>}, \texttt{/}\}$, start symbol A_{XML}, and production

$$A_{\text{XML}} \rightarrow (\texttt{a} + \ldots + \texttt{z} + \texttt{<a>} A_{\text{XML}} \texttt{})^*$$

- Oracle $\mathcal{O}_{\text{XML}}(\alpha) = \mathbb{I}[\alpha \in \mathcal{L}(C_{\text{XML}})]$
- Seed inputs $E_{\text{XML}} = \{\alpha_{\text{XML}}\}$, where $\alpha_{\text{XML}} =$ `<a>hi</a>`

Osbert Bastani, Rahul Sharma, Alex Aiken, Percy Liang. Synthesizing Program Input Grammars. PLDI'17.

`https://github.com/obastani/glade`

Parsing is an Active Field

Although the topic, the theory, and the techniques of parsing are almost well-studied in academia and widely-used in industry, research has never been stopped and novel progress are made year by year.