Top Down PEG Scaffold

Natarajan Shankar and Zephyr Lucas

Dartmouth College

May 23, 2024

Outline

1 Introduction and Motivation

2 Prototype Verification System (PVS)

3 Modifications to the PEG Grammar

4 Future Work and Improvements

5 Conclusions, Acknowledgments, References

Common Data Formats

- Text: ASCII, UTF-8, CSV, JSON, XML, YAML
- Code: ELF, JAR
- Document: PDF, Postscript
- Images: JPEG, GIF, BMP
- Audio: WAV, MP3
- Video: MPEG, AVI, Flash
- Compression: DEFLATE

Verification/PEG History

- Mid-1950s: (Chomsky [HMU07]) Formal theory of Parsing
- 1970: (Birman and Ullman [BU70]) Let-Then-Else construct allowing backtracking
- 2004: (Ford [For04]) Parsing Expression Grammars
- 2008: (Warth, Douglass, and Millstein [WDM08]) Supporting Left Recursion in Packrat Parsing
- 2009: (Barthwal and Norrish [BN09]) Proof of Parse Verifier
- 2011: (Koprowski and Binsztok [KB10]) Verified PEG parser
- 2020: (Blaudeau and Shankar [BS20]) Verified Packrat parsing in PVS.

Parsing Expression Grammars

- Replaces CFG ordered choice with a prioritized choice operation
- Results in a nice deterministic parsing algorithm with linear runtime
- Contains non-context free languages
- Picking up in popularity:

	LR	LL	LALR	GLR	Earley	other	**PEG**
Wikipedia	27	34	62	22	7	25	**46**
GitHub	54	67	65	9	7		**105**

[LMR20]

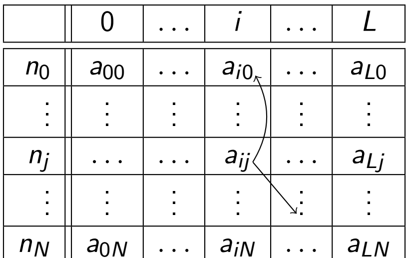
PEG Definition

```
1 len, max: VAR index        % The length of the input
2 byte: TYPE = below(256)
3 strings(len): TYPE = ARRAY[below(len) -> byte]
4 num_non_terminals: byte = 255 % Grammar dependent
5 non_terminal: TYPE = below(num_non_terminals)
6
7 peg : DATATYPE
8  BEGIN
9   epsilon : epsilon?
10   failure : failure?
11   any(p : [byte -> bool]) : any?
12   terminal(a: byte) : terminal?
13   concat(e1, e2: non_terminal) : concat?
14   choice(e1, e2: non_terminal) : or?
15   check(e: non_terminal) : and?
16   neg(e: non_terminal) : not?
17  END peg
```

The Scaffold

- The scaffold is a two-dimensional array mapping each position/nonterminal to an entry.

```
1 scaffold(len) : TYPE =
2   ARRAY[pos: upto(len) ->
3     ARRAY[non_terminal -> (nice_entry?(len, pos))]]
```



	0	$\dots$	i	$\dots$	L
n_0	a_{00}	$\dots$	a_{i0}	$\dots$	a_{L0}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n_j	$\dots$	$\dots$	a_{ij}	$\dots$	a_{Lj}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n_N	a_{0N}	$\dots$	a_{iN}	$\dots$	a_{LN}

Scaffold Entries

- Each entry within the previously defined scaffold is an algebraic datatype with constructors, accessors, and recognizer.

```
1 ent: DATATYPE
2  BEGIN
3   fail(dep: uint64): fail?
4   pending: pending?
5   loop: loop?
6   good(dep: uint64 , span:
7   push(pos: uint32 , nt: uint8): push?
8  END ent
```

Loop Detection

- Non-trivial left recursion is provably impossible to detect within a grammar.
- Previous results either:
  - Modified PEG semantics to define behavior for left recursion (e.g. treat any instance of left recursion as failure)
  - Introduced necessarily conservative checks of well-formedness and only parsed grammars passing these checks.
- We implement a runtime loop detection within the scaffold to determine if there is a dependency loop within the grammar for the given input.

Proof of Parse

- Definition: A structure which can be used to verify a correct parse.
  - For CFG this is provably easier computationally [CS63]
  - Open question whether or not this is easier for PEGs
- For our work:
  - This naturally falls out of our proof invariants in PVS
  - Proof of parse can be independently verified
  - Parse tree can be easily extracted from the proof of parse
- Negligible overhead to build and check the proof of parse.

Proof of Parse in PVS

- Defined a proof-of-parse format for success, failure, loop outcomes and extracted valid proofs from parser output.

```
1  buildproof(len, G, (s: strings(len)),
2          (rootpos: upto(len)), (rootnt: non_terminal))
3          (st: endstate(len, G, s, rootpos, rootnt),
4          n, (i | i <= len)):
5     {P | good_parsetree?(len, G, s)(qempty, n, i, P)
6        AND entry(P) = st`scaf(i)(n) }
7   = (IF loop?(st`scaf(i)(n))
8     THEN (LET
9            pendfun = (LAMBDA (n: non_terminal): pending),
10           A = (LAMBDA (i: upto(len)): pendfun)
11         IN buildloop(len, G, s, rootpos, rootnt)
12                     (st, A, qempty, n, i))
13    ELSE buildtree(len, G, s, rootpos, rootnt)
14                  (st, qempty, n, i)
15    ENDIF)
```

Motivation for Modifications

- Even with memoization (packrat parsing), the PEG grammar can be slow because it is not backtracking efficiently from failure.
- One source of slowness is that the base case handles only one character at a time.
- As grammars and data languages evolve, the classes of parsers and verifiiers will need to adapt to match.

Let Then Else

- A variant `let x = A then B else C` can be used more efficiently and replaces several PEG constructors and reduces backtracking.

```
1 prepeg: DATATYPE
2 BEGIN
3   epsilon : epsilon?
4   failure : failure?
5   any(p : [byte -> bool]) : any?
6   terminal(a: byte) : terminal?
7   lte(fst: nterm, lthen : nterm, lelse: nterm): lte?
8 END prepeg
```

Let Then Else (cont)

- LTE does not gain any power, as:

$$\texttt{lte(A, B, C)} \equiv \texttt{AB / (!A)C}$$

- LTE does not lose any power either:

$$\begin{aligned}
\texttt{AB} &\equiv \texttt{lte(A, B, f)} \\
\texttt{A/B} &\equiv \texttt{lte(A, }\epsilon\texttt{, B)} \\
\texttt{\&A} &\equiv \texttt{lte(lte(A, f, }\epsilon\texttt{), f, }\epsilon\texttt{)} \\
\texttt{!A} &\equiv \texttt{lte(A, f, }\epsilon\texttt{)}
\end{aligned}$$

Adding DFA Tokenization

- Change the base case from a single terminal to instead cover any DFA. (This is tricky DFA and PEG semantics are not compatible.)

```
1 prepeg: DATATYPE
2 BEGIN
3   epsilon : epsilon?
4   failure : failure?
5   any(dfa : dfa) : any?
6   terminal(a: byte) : terminal?
7   lte(fst: nterm, lthen : nterm, lelse: nterm) : lte?
8 END prepeg
```

Evaluation

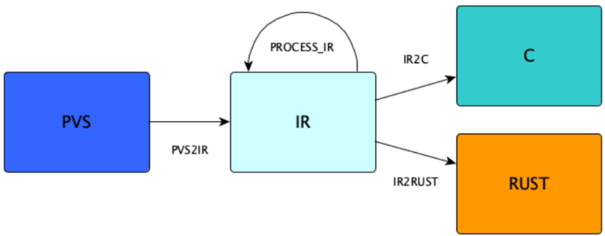
- The original proofs were hard: took around four to six weeks with over a 150 nontrivial proof obligations.
- Proofs replayed robustly through these modifications even though the change was significant.
- Run time speedup (Currently non-rigorous testing)
  - Let-Then-Else speed up was around 30%
  - With DFA construct, the parsing time was halved.

Proof Robustness

- Change is the only constant, so proofs do need to be robust.
- The pre-packaged automation (built-ins and strategies) made the proofs reusable across similar proof obligations and across the evolution of the specification/algorithm.
- There are many sources of non-robustness:
  1. Proof commands that mention formula numbers are sensitive to reordering/shifting of formulas in a sequent
  2. Proof commands that include expressions are sensitive to changes in these expressions, e.g., Skolem names.
  3. If the order of cases changes, then proofs can lose synchrony.
  4. If definitions change, the proof obligations can get detached from their proofs.
- A lot of post-processing is needed to define proof strategies and refactor proofs to make them robust.

PVS2C

- PVS2C generates safe, efficient, standalone C code for a full functional fragment of PVS.
- The translation is factored through an intermediate language that represents PVS expressions in A-normal form and performs a light static analysis to identify the *release points* for references.



Other Future Work

- Extensions to the parser-interpreter formalizations:
  - Lazy scaffold column allocations in sparse DFA based PEG parsing
  - Adding attribute grammars and views to PEG semantics
- PEG proof of parse provably easier than parsing? (open theoretical question)
- Defining an abstract machine which can formalize this work as well as other parsing algorithms / formalisms.
- Rigorous profiling and performance studies of the PVSC to gain insights on both grammar optimizations as well as code generation optimizations.
- Verification of code extractor

Conclusions

- Scaffolding automata approach to parsing PEGs within a verification system
  - Dynamic loop detection
  - Chart parsing
  - Ease of generating proof of parse
- Simple modifications to PEGs can improve efficiency
- Proofs are robust to changes in the grammar
- Demonstration of verifying parsing algorithms with
  - Proof robustness
  - Not inherently inefficient extracted code

Acknowledgments

- This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075 and in part under Contract No. HR001119C0121. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).
- Special thanks to Sean Smith, Ben Kallus and the rest of Dartmouth's trustlab for their insight and support on this project.
- Special thanks to Sam Owre and Prashant Mundkur from SRI for their help and feedback.

Thank you!

Questions?

References I

Aditi Barthwal and Michael Norrish, *Verified, executable parsing*, Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009 (Giuseppe Castagna, ed.), Lecture Notes in Computer Science, vol. 5502, Springer, 2009, pp. 160–174.

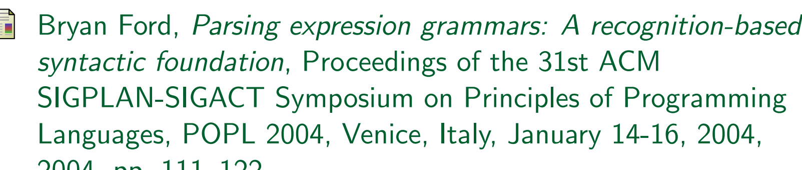
Clement Blaudeau and Natarajan Shankar, *A verified packrat parser interpreter for parsing expression grammars*, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (New York, NY, USA), CPP 2020, Association for Computing Machinery, 2020, p. 3–17.

References II

Alexander Birman and Jeffrey D Ullman, *Parsing algorithms with backtrack*, 11th Annual Symposium on Switching and Automata Theory (SWAT 1970), IEEE, 1970, pp. 153–174.

N. Chomsky and M.P. Schützenberger, *The algebraic theory of context-free languages**, Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.), Studies in Logic and the Foundations of Mathematics, vol. 35, Elsevier, 1963, pp. 118–161.

References III



- Bryan Ford, *Parsing expression grammars: A recognition-based syntactic foundation*, Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004, 2004, pp. 111–122.
- J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to automata theory, languages, and computation*, Pearson/Addison Wesley, 2007.
- Adam Koprowski and Henri Binsztok, *Trx: A formally verified parser interpreter*, Log. Methods Comput. Sci., 2010.

References IV

Bruno Loff, Nelma Moreira, and Rogério Reis, *The computational power of parsing expression grammars*, Journal of Computer and System Sciences **111** (2020), 1–21.

Alessandro Warth, James R. Douglass, and Todd Millstein, *Packrat parsers can support left recursion*, Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (New York, NY, USA), PEPM '08, Association for Computing Machinery, 2008, p. 103–110.