Extended Abstract: Nano-parsing

A Data-parallel Architecture for Perverse Parsing Environments

Aaron W. Hsu
aaron@dyalog.com
Dyalog, Ltd.
Bloomington, IN, United States

Brandon Wilson
TBD
Dyalog, Ltd.
TBD, Japan

ACM Reference Format:
Aaron W. Hsu and Brandon Wilson. 2024. Extended Abstract: Nano-parsing: A Data-parallel Architecture for Perverse Parsing Environments. In *Proceedings of The 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '24).* ACM, New York, NY, USA, 3 pages.

1 Introduction

Traditional parsing literature and design tends to focus on LL or LR grammars [1, 4, 7, 8] or have contexts that require minimal computation that can be reasonably expressed in formal notations not designed for general purpose computation [5, 6, 9]. The architecture and tooling support that emerges from this research and the general community usually comes in the form of parser generators, parser combinators, or, usually, a fallback to recursive descent implemented directly in the host language. Context sensitive languages often feel like second class citizens in the software engineering parsing ecosystem [5, 6]. In our work on static, offline parsing of APL code, the complexity of the task has driven us to coin the term "perverse parsing environments" to describe the situation.

2 Challenges

2.1 Defining the Language

Modern APL syntax is not formally specified. Industrial requirements prevent the use of a reduced language. The parser must support multiple variations to facilitate migration from one APL implementation to another. There is no reference implementation either! Extant parsers do not parse modern APL, and none have meaningful APIs for black box introspection. Existing commercial implementations do not parse statically, but operate on token streams. Even these implementations cannot serve as reference implementations because of "bugs" that make finding ground truth impossible via mechanical means alone. Incremental evolution of the parser is the only path forward, demanding frequent and potentially breaking changes; an architecturally resilient and easily modified parser is required.

2.2 Language Ambiguity/Context Sensitivity

The language itself exhibits challenging features such as the famous ambiguous phrase `A B C`, which has over 10 different parses depending on the types of each name. Furthermore:

- It is dynamically typed, but parsing depends on the types of variables
- It mixes dynamic and lexical scope, allowing intermingled function calls
- Code makes extensive use of an eval that can and does introduce new bindings into scope (dynamic/lexical)
- Code is often quite flat, and many functions reference many other functions leading to highly connected call graphs
- Syntax errors may signal at runtime or parse time
- The context of an expression can amount to the whole program quite easily since it may depend on distantly defined values
- Inherent generality and ambiguity can introduce combinatorial factors to the size of the parse tree
- Forward references in the source can alter or affect the parse tree

These and other more mundane context sensitive challenges make any traditional parsing architecture difficult and cumbersome to use.

2.3 Industrial Requirements

As an industrial, commercial parser, we must handle real world source, and it must integrate with the rest of the Co-dfns compiler [2]. This means adequate debugging and source information with excellent and apropos error messages at the right time. The Co-dfns compiler is meant to self-host on the GPU [3], so parsing must execute efficiently on the GPU and CPU, as well as under interpretation and compilation, since it may be embedded into interpreted code. The high connectedness of the AST means we may need the entire source available to us at once to parse it correctly, which presents performance challenges when APL codebases may have millions of lines of code

ARRAY '24, June 25, 2023, Copenhagen, Denmark

```
Input Source:
:Namespace ⋄ f←{(2=+⌿0=X∘.|X)⌿X←⍳ω} ⋄ :EndNamespace
```

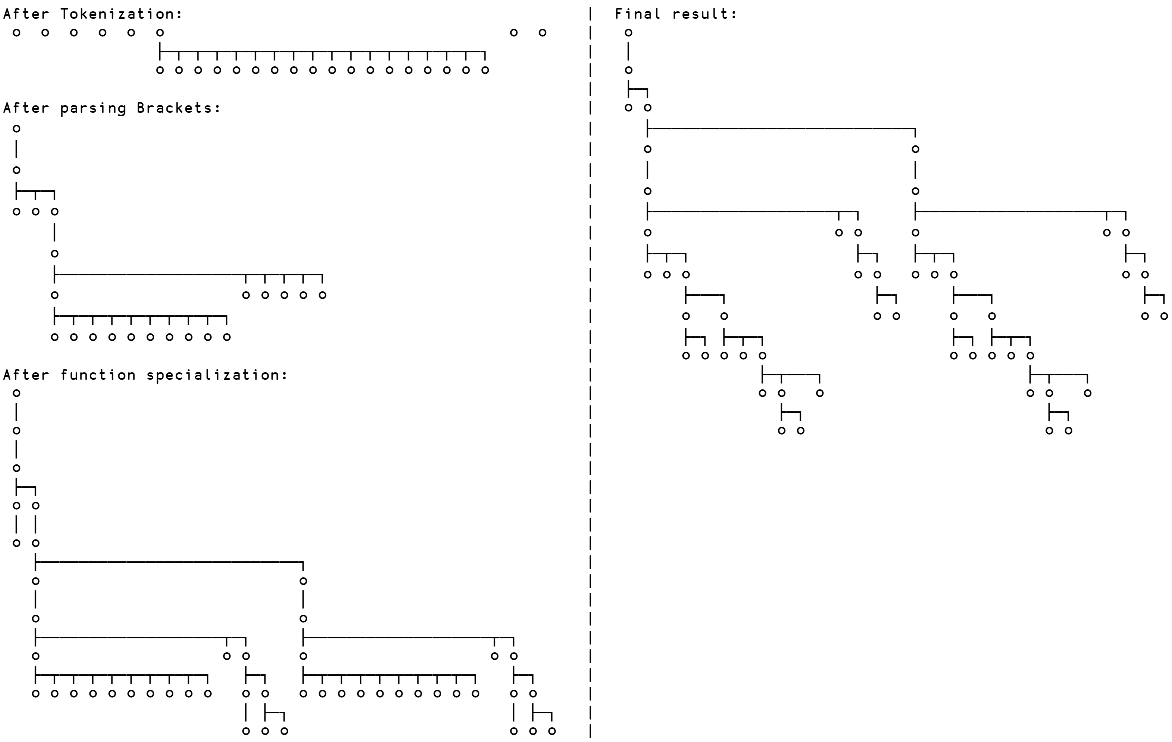

Figure 1. Various stages of the Co-dfns Parser

Finally, the architecture must be maintainable and scalable by a single developer, not a team who can farm out engineering labor such as maintaining separate CPU and GPU implementations that are manually synchronized.

3 Nano-parsing to the Rescue

To address these concerns, we present a parser architecture called Nano-parsing, written in APL, and designed to meet the above needs. The core strategy of nano-parsing is to transpose the act of parsing from a top down, depth-first vision of the tree to a series of data-parallel passes over the input that incrementally refines the AST into its final form. This "bottom up" approach results in many small, independent blocks of code that represent a chain of conceptually functional transformations whose dependencies form a loose semi-lattice. These passes are implemented in a GPU compatible way in a data parallel style of APL.

Fundamentally, this means that at any point, anywhere in the parser, we have full visibility of the global parsing state to use at will. Additionally, since the entire parser is a series of extremely simple and small passes, we no longer must plan ahead carefully the phases of parsing (cut points) that would be needed to handle APL, since these cut points fall out naturally from the design. Previous attempts at parsing APL using more traditional designs demonstrate the difficulty in choosing these cut points and the resulting rigidity and inflexibility in design that results.

The nano-parsing architecture possesses the following desirable qualities:

- It is GPU compatible
- It decouples error reporting from the act of parsing, allowing errors to be dealt with independently
- It allows arbitrary computation at any point throughout the parsing process, such as type inference
- It is well suited for interpreters that may suffer interpretation overhead with traditional methods
- Its semantics is much easier to understand at scale using code inspection because of a linear control flow
- Since control flows directly from pass to pass, it is easier to see what code will be impacted by a change, since all data flows in a single direction
- The global state and independent passes makes it relatively easy to modify a pass or insert new passes without impacting the rest of the code

We present details of the implementation and the architecture as a whole as well as discuss the results of migrating to this new design from a traditional PEG grammar. We will highlight some of the ways this design has enabled us to fix issues in our pasrsing of APL.

References

[1] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 111–122.

[2] Aaron Wen-yao Hsu. 2014. Co-dfns: Ancient language, modern compiler. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 62–67.

[3] Aaron Wen-yao Hsu. 2019. *A data parallel compiler hosted on the gpu*. Ph. D. Dissertation. Indiana University.

[4] Stephen C Johnson et al. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[5] Jan Kurs, Mircea Lungu, and Oscar Nierstrasz. 2014. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*.

[6] Nicolas Laurent and Kim Mens. 2016. Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 15–27.

[7] Francisco Ortin, Jose Quiroga, Oscar Rodriguez-Prieto, and Miguel Garcia. 2022. Evaluation of the Use of Different Parser Generators in a Compiler Construction Course. In *Information Systems and Technologies*, Alvaro Rocha, Hojjat Adeli, Gintautas Dzemyda, and Fernando Moreira (Eds.). Springer International Publishing, Cham, 338–346.

[8] Terence Parr and Kathleen Fisher. 2011. LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices* 46, 6 (2011), 425–436.

[9] William A Woods. 1970. Context-sensitive parsing. *Commun. ACM* 13, 7 (1970), 437–445.