ARRAY Submission:

Extended abstract:

The Nano-parsing architecture ~~for~~: Data Parallel Parsing for Perverse Environments.

Figures:

- Diagram of Parsing Passes

Outline:

Problem Statement
Solution
Implementation

Problem Statement:

Traditional parsing literature and design tends to focus on well defined grammars over languages that are mostly context free or have contexts that require minimal computation that can be reasonably

4/6/2024

expressed in formal notations not designed for general purpose computation. The architecture and tooling support that emerges from this research and the general community usually comes in the form of parser generators, which compile grammars written in a restricted ~~do~~ language to some implementation such as a parse table or recursive descent parser; parser combinators, which allow the more concise expression of and underlying model; or, usually, a recursive descent parser implemented directly in the host language, since this often is the most flexible and scallable option as complexity increases. Managing a context sensitive language in these systems can feel like a second class citizen.

On our work on static,

offline parsing of APL code, the situation is much more complex. The complexity of the task and the myriad challenges has caused us to coin the term "perverse parsing environments" to describe the situation, since non-trivial context sensitivity is only one issue.

The first issue comes from defining just what APL is as a syntax. Since the parser must meet industrial requirements, we cannot simply use a reduced language. There is no formal specification of the language. Worse, there are multiple linguistic variations that must be supported, since code migration from one APL implementation to another is a major use case for a static parser. But it does not stop there, because there is no reference implementation of

4/6/2024

an APL parser either! ~~The~~ Extant parsers are woefully incomplete with respect to modern APL, and none of them have a meaningful API to permit even basic kinds of black box introspection. Existing commercial implementations ~~often~~ do not implement a static parser, but operate on token streams. Unfortunately, even ~~most~~ not the major implementations cannot serve as black box reference implementations b/c of a preponderance of "bugs" that do not affect everyday users, but that do make finding ground truth impossible via mechanical means alone. This makes the process of writing a parser a fundamentally incremental one with frequent introduction of potentially breaking new information demanding a resilient parser that can be modified

easily at will ~~or~~.

The second issue comes from the language itself, which exhibits many challenging features, such as the fambul ambiguous $f phrase:

A B C

which has well over 10 different parses depending on the types of each name. But there are more:

- It is dynamically typed, but parsing depends on types. [type inference implied.]
- It mixes dynamic and lexical scope, and they may intermingle.
- Source code makes extensive use of execute in some places, and execute can and often does introduce new bindings into various scopes and must account for lexical and dynamic scope.

- Code bases are often quite flat, with no call graphs of very high out and in degree.
- Some syntax errors must be reported at runtime and not parse time.
- Symbols can be overloaded to mean different syntactic things in different contexts.
- ~~Certain~~ There are many "traditional" context sensitivities.
- The context of an expression can amount to the whole program because a parse of an expression may depend on distant, non-local information
- Computed gotos can form ~~a~~ no a major means of control flow and often involve execute (eval).
- Inherent generality and ambiguity can introduce combinatorial factors to the size of the tree.

Foward references in the source can alter or affect the parse tree.

These and other more mundane challenges makes any traditional parser architecture difficult and cumbersome to use.

The third major class of issues comes from the context of the parser itself. As an industrial parser, it must address and deal with source in the field, and it must integrate with the rest of the Co-dfns compiler. This means it must deliver adequate debugging and source data in the AST and deliver excellent and appropos error messages at the right stage of compilation. Since the whole Co-dfns compiler is meant to run & self-host on the GPU, the parser must execute

efficiently on the GPU and CPU. It must also perform well when integrated with the APL interpreter, since it may embed into such products. Since real APL codebases may ~~dom-~~ ~~ant~~ have millions of lines of code in them, we must handle this. Given the high connectedness and combinatorial nature of the AST, ~~we must have a sufficiently efficient~~ ~~a~~ this means we may need, in effect, the entire source available to us at once to parse it correctly, which presents immediate performance challenges.

The final nail is the need for an architecture maintainable and scalable ~~for~~ by a single developer, not a team who can farm out engineering labor such as maintaining a GPU

implementation that differs from the GPU version and that must be synchronized.

To address these concerns, we present a parser architecture we ~~ca~~ deemed Nano-parsing, written in APL and designed to meet the above needs. ~~Because of the~~ The core strategy of nano-parsing is ~~th~~ 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 ~~parsen~~ 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 "cut points" of a series of parsers that would be needed to handle APL, since these cut points are inherent in the design. Previous attempts at ~~th~~ passing APL using more traditional designs ~~has~~ de have demonstrated the difficulty in ~~d~~ choosing the cut points and the

fragility in the result.

This structure has the following advantages:

- It is GPU compatible
- It decouples error reporting from the ~~e~~ ~~struct~~ act of parsing, allowing errors to be ~~report~~ dealt with independently.
- It allows any arbitrary computation at any point throughout the parsing process, such as type inference
- It is well-suited for interpreters that may suffer interpretation overhead of traditional methods.
- Its semantics is much easier to understand from code inspection because of a linear control flow in complex languages
- 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 P&G parser generator. We will highlight some of the ways this design has enabled us to fix issues in our parsing of APL.