CS4201 Practical 1

160001055

October 2019

1 Introduction

This practical involved the implementation of a lexer and a parser for the programming language *Oreo*, given an informal specification of the language and various examples. Furthermore, as an extension, I have implemented functions and comments handling, extensive error handling and a primitive type checker.

2 Design

The design for a parser is always a challenging one, as factors such verifiability, correct error handling and similar often conflict. As such, I decided to work on a multi-layer design stage. The lowest level, the scanner, simply strips white-space, and isolate strings and comments. On top of it the lexer recognizes the tokens and passes them to the parser, whose job to actually properly parse the grammar is simplified. Once the parse tree is constructed, the validator runs on it, ensuring that each rule is well formed. Each stage is independently tested, to ensure correctness, and to make sure that changes at each level will not effect the others.

2.1 Iteration

The entire application is built as a on demand stream of characters. Starting from the original character streams, each layers asks the underlying one for the next element, and waits until the computation is completed for the result. This has the added flexibility that, when the input is of considerable size, it is not needed to allocate for the scanning and lexing phase, which could considerably impact memory usage. It also guarantees that the grammar to be parsed is **LL(1)**, as backtracking is impossible. In exchange, computationally this can be slightly less efficient, as operation are interleaved hurting branch prediction. However, since in a full compiler the lexing and the parsing are almost never the bottleneck, I felt the benefits outweighed the cons.

2.2 Scanning

The scanning operation essentially isolates strings and comments in the code, Other than this it strips white spaces and chunks together characters that are not separated by white-space. Further than this, it also collects the line numbers of the scanned item, to be passed to the next stages for error reporting. The scanning is designed to be strictly $O(n)$ in the size of the input, never actually ever needing to backtrack. Also it is nice to note that comments are actually propagated to next stages, and not simply removed. In theory this could later allow for comment-based syntax extensions[1].

2.3 Lexing

Lexing is substantially more complex than scanning, as it cannot rely on white-space for token separation, and as such needs to have a more complex solution. The lexer we have queries the lexer, and operates on each resulting item in order to generate a corresponding stream of tokens. All of these streams of tokens are then flattened in a contiguous stream that then the parser operates on. As in scanning, the lexer is made so to only ever need to read each character in the input once, which, as the implementation shows, is harder than it looks.

2.4 Grammar

In order to allow for the parser to never need to backtrack, I needed to convert the grammar to **LL(1)**. This involved eliminating left recursion, specifying order of operations, removing ambiguity and computing start and follow sets. This was not particularly challenging for most of the grammar, except for the expression part. In particular, preserving the typing in case of bool expression proved complicated. In order to solve this, I decided to move the type checking to a later stage (as it is done in most compilers) and to work with a simpler fully unambiguos grammar. In particular the grammar we resulted with follows. Note that cursive terms are non terminal, bold ones are tokens and typed ones are tokens that might take multiple values (typically literals). Also, not that comments are not included here, as the parser filters them out by default.

$$P \rightarrow \textbf{program } \texttt{id} \ \textit{compound}$$

$$\textit{compound} \rightarrow \textbf{begin } \textit{statement}\, \textit{statements } \textbf{end}$$

$$\textit{statements} \rightarrow \textit{statement}\, \textit{statements} \mid \epsilon$$

$$\textit{statement} \rightarrow \textit{decl} | \textit{print} | \textit{assign\_or\_fun} | \textit{if} | \textit{while} | \textit{fun}$$

$$\textit{decl} \rightarrow \textbf{var } \texttt{id } \textit{opt\_expr};$$

$$\textit{opt\_expr} \rightarrow := \textit{expr} | \epsilon$$

$$\textit{print} \rightarrow \textbf{print } \textit{expr}; \mid \textbf{println } \textit{expr}; \mid \textbf{get } \texttt{id};$$

[1] I am not advocating this here, it is almost certainly a terrible idea, but you *could* do it

$$assign\_or\_fun \rightarrow \texttt{id}\, tuple\_or\_assign;$$

$$tuple\_or\_assign \rightarrow := expr \mid (c\_args)$$

$$if \rightarrow \textbf{if}\ (expr)\ \textbf{then}\ compound\ else;$$

$$else \rightarrow \textbf{else}\ compound \mid \epsilon$$

$$while \rightarrow \textbf{while}\ (expr)\ compound;$$

$$fun \rightarrow \textbf{procedure}\ \texttt{id}\ \ (d\_args) compound;$$

$$d\_args \rightarrow \textbf{var}\ \texttt{id}\ \ d\_args' \mid \epsilon$$

$$d\_args' \rightarrow , \textbf{var}\ \texttt{id}\ \ d\_args' \mid \epsilon$$

$$expr \rightarrow term\ expr'$$

$$expr' \rightarrow \wedge\ term\ expr' \mid \epsilon$$

$$term \rightarrow factor\ term'$$

$$term' \rightarrow \leq_R\ factor\ term' \mid \epsilon$$

$$factor \rightarrow product\ factor'$$

$$factor' \rightarrow \oplus\ product\ factor' \mid \epsilon$$

$$product \rightarrow atom\ product'$$

$$product' \rightarrow \otimes\ atom\ product' \mid \epsilon$$

$$atom \rightarrow unit \mid \textbf{not}\ atom$$

$$unit \rightarrow (expr) \mid \texttt{id}\ \ func\_args \mid \texttt{str} \mid \texttt{int} \mid \texttt{bool}$$

$$func\_args \rightarrow (c\_args) \mid \epsilon$$

$$c\_args \rightarrow expr\ c\_args' \mid \epsilon$$

$$c\_args' \rightarrow , expr\ c\_args' \mid \epsilon$$

$$\wedge \rightarrow \textbf{and} \mid \textbf{or}$$

$$\leq_R \rightarrow < \mid \leq \mid > \mid \geq \mid ==$$

$$\oplus \rightarrow\ + \mid -$$

$$\otimes \rightarrow\ * \mid /$$

This grammar, despite being really verbose, does do operator precedence correctly, grouping operators of the same level of precedence on the right. The operator grouping is as follows: $\wedge, \leq_R, \oplus, \otimes, \textbf{not}$ so the following expression: "$\textbf{not} 1 * 2 + 3 < 4\ \textbf{and}\ 5$" should parse as $(((\textbf{not}(1) * 2) + 3) < 4)\ \textbf{and}\ 5$ and the following "$a - b - c$" will parse as $(a - (b - c))$.

2.5 Parsing

The parser is a simple recursive descent one, which progressively builds a typed abstract syntax tree. It is mostly a direct translation of the grammar above, operating on the stream of tokens produced by the lexer. Each nonterminal f corresponds to a function of the form $f : \texttt{TokenStream} \to Ok\ \texttt{F}|Err\ \texttt{Error}$ where F is a typed node that represent the terminal. Composing this functions quite naturally allows for both good error reporting and flexible parsing.

2.6 Validation

In the original grammar, there were different production rules for boolean and expressions, such that every boolean was an expression and not vice-versa. Despite much trying, I wasn't able to construct an equivalent grammar that was **LL(1)** and still was able to uphold this condition. In fact I speculate that such a grammar does not exist, but this is just speculation. In order to still produce errors when invalid expressions are used as booleans, I added an extra validation step, which in a way mimics the way a type-checker in a complete compiler works. The validator walks the abstract syntax trees and checks the following conditions holds:

$$\texttt{if}(e)\cdots \implies \mathbf{B}(e)$$
$$\texttt{while}\ (e)\cdots \implies \mathbf{B}(e)$$
$$expr \implies \mathbf{V}(expr)$$

Where we define:

$$recV(f, tail) = \begin{cases} tail = \epsilon & \texttt{true} \\ \text{otherwise} & f(tail) \end{cases}$$

And then we recursively (using pattern matching) define $\mathbf{V}, \mathbf{B}$ as follows:

$$\mathbf{V}(expr) \text{ where } epxr \to term\ expr' = \begin{cases} expr' = \epsilon & \mathbf{V}(term) \\ \text{otherwise} & \mathbf{B}(term) \text{ and } \mathbf{B}(expr') \end{cases}$$

$$\mathbf{B}(expr') \text{ where } epxr' \to \wedge term\ expr'_1 = \mathbf{B}(term) \text{ and } rec(\mathbf{B}, expr'_1)$$

$$\mathbf{V}(term) \text{ where } term \to factor\ term' = \mathbf{V}(factor) \text{ and } rec(\mathbf{V}, term')$$

$$\mathbf{B}(term) \text{ where } term \to factor\ term' = \begin{cases} term' = \epsilon & \mathbf{B}(factor) \\ \text{otherwise} & \mathbf{V}(factor) \text{ and } \mathbf{V}(term') \end{cases}$$

$$\mathbf{V}(term') \text{ where } term' \rightarrow \leq_R factor\ term'_1 = \mathbf{V}(factor) \text{ and } rec(\mathbf{V}, term'_1)$$

$$\mathbf{V}(factor) \text{ where } factor \rightarrow product\ factor' = \mathbf{V}(product) \text{ and } rec(\mathbf{V}, factor')$$

$$\mathbf{B}(factor) \text{ where } factor \rightarrow product\ factor' = \mathbf{B}(product) \text{ and } factor' = \epsilon$$

$$\mathbf{V}(factor') \text{ where } factor' \rightarrow \oplus product\ factor'_1 = \mathbf{V}(product) \text{ and } rec(\mathbf{V}, factor'_1)$$

$$\mathbf{V}(product) \text{ where } product \rightarrow atom\ product' = \mathbf{V}(atom) \text{ and } rec(\mathbf{V}, product')$$

$$\mathbf{B}(product) \text{ where } product \rightarrow atom\ product' = \mathbf{B}(atom) \text{ and } product' = \epsilon$$

$$\mathbf{V}(product') \text{ where } product' \rightarrow \otimes atom\ product'_1 = \mathbf{V}(atom) \text{ and } rec(\mathbf{V}, product'_1)$$

$$\mathbf{V}(atom) \text{ where } atom \rightarrow unit = \mathbf{V}(unit)$$

$$\mathbf{V}(atom) \text{ where } atom \rightarrow \textbf{not}\ atom_1 = \mathbf{B}(atom_1)$$

$$\mathbf{B}(atom) \text{ where } atom \rightarrow unit = \mathbf{B}(unit)$$

$$\mathbf{B}(atom) \text{ where } atom \rightarrow \textbf{not}\ atom_1 = \mathbf{B}(atom_1)$$

$$\mathbf{V}(unit) \text{ where } unit \rightarrow \texttt{id}\ func\_args | \texttt{str} \mid \texttt{int} \mid \texttt{bool} = \texttt{true}$$

$$\mathbf{V}(unit) \text{ where } unit \rightarrow (expr) = \mathbf{V}(expr)$$

$$\mathbf{B}(unit) \text{ where } unit \rightarrow (expr) = \mathbf{B}(expr)$$

$$\mathbf{B}(unit) \text{ where } unit \rightarrow \texttt{id}\ func\_args \mid \texttt{bool} = \texttt{true}$$

$$\mathbf{B}(unit) \text{ where } unit \rightarrow \texttt{str} \mid \texttt{int} = \texttt{false}$$

Using this simple (yet verbose) rules, we can ensure that every expression is valid, the operand of boolean operators are indeed booleans and that the conditions in control flows are of the right type.

2.7 Error Handling

Error handling is done independently in each stage of the parser. The scanner is able to report unclosed strings and comments. The lexer is able to report when some characters cannot be recognized as tokens (i.e. `=` when not in `==`, `<=`, `=>`, `:=`). Similarly the parser reports most of the error in the grammar and generally it is quite insightful. Up to this point, the error information contains the line number and a succint description of what the compiler was expecting and what it was found. The further validation, which was added later after the grammar changes, is able to validate typing very effectively, but since the abstract syntax tree does not track line information it is not able to point out where the error is. In order to accommodate that the best solution would be a two layer parse tree, one strongly typed as the one we have now and one typeless one which would track line information. That would require a lot of changes in the parsing, which I did not have time to implement. While the scanner and the lexer can lazily built the stream and report every error at the end, the parser works eagerly reporting errors as soon as they come. While it would be possible to parse as much as possible before reporting errors, it would have involved a lot of change as:

- Introducing an `Error` node in the typed tree
- Allowing the `Error` to be a children of any other node

Which would have made any further operation on the syntax tree more cumbersome. Furthermore, this way integrated neatly with Rust's syntax and try operator, which significantly cut down the lines of code.

2.8 Testing

Since abstract syntax trees can grow to considerable sizes, especially when a grammar is so granularly defined as ours, and since parsing can present many pitfalls I wanted a testing solution that allowed for easily adding and verifying new cases, without having to manually validate each one. To this avail, I took a page out of the rust analyzer[2] project, and used a snapshot based solution. Each test generates a snapshot of the output of that test, and on each change the user gets a chance to review said snapshot to verify that the behaviour is as expected. In particular, adding each of the examples as a snapshot test allowed me to be sure I was compliant to the specification. Also, since the testing independently verified each layer of the parse, I could verify that changes in one section would not negatively effect the others. Also, without running the program, the snapshots are version controlled and serve as documentation for what the parsing will result in.

[2] `https://github.com/rust-analyzer/rust-analyzer`

2.9 Outputs

For the output requirements of the program, I decided to aim for a machine readable format. Since my artistic skills are not in a great place, this seemed like the best solution, and also this felt like a decent way to allow for other programs to use the output in a proper way. As for machine readable format, I allow the user to choose between YML, JSON and RON (Rusty Object Notation). While neither of them is particularly pretty, I personally quite like the YML output. Also, since the serialization is powered by the `serde` [3] framework, it would be trivial to add any more of the supported data types.

3 Implementation

3.1 Rust

The choice of Rust for this project was dictated by various factors:

- I already had experience with Rust in parsers, thanks to some contributions I did on Rust analyzer
- The snapshot testing library works very well for ASTs
- A strong type system allows for the ASTs to be completely typechecked
- Performance
- Static memory safety
- Pattern Matching
- Try error handling

Some of these, like performance and testing, are simply nice to have, and don't effect the program too much. Others, like the strong type checking and the pattern matching, are simply invaluable, as they prevent you from shooting yourself in the foot too often. In particular, without the Try operator, the size of the parser might have doubled our tripled, and with it the complexity overhead.

3.2 Zero Copy

One of the most interesting features of this program is that I actively aimed to reduce allocations and copying to a minimum. In fact, the only allocations in the entire core are those that are needed to avoid infinitely recursive types, and for the statement list. Even the identifier token never allocates, but instead keeps a reference to the data in the original input string. As for copying, I can guarantee that for complex types it never happens unless it is in an error path.

[3] `https://serde.rs/#data-formats`

3.3 Iteration

The parser operates on a `TokenStream`, which is simply a trait wrapper over the `Peekable<Iterator<Item = Token>>` trait. What does this imply, together with the zero allocation condition, is that the grammar must be **LL(1)** and that the parser never backtrack. In fact, this condition can be statically checked by the compiler, which I think is awesome. The above follows from the fact that a we can at most only peek one token ahead in the stream, and from the fact that a general iterator cannot be rewinded. Furthermore, the no allocation property implies we cannot store a dynamic amount of tokens to further use as rewind stack, and as such we must be satisfying the desired properties.

3.4 Benchmarking

While it was not required for the practical, I found it interesting to write a couple of benchmarks for the parser. I have three benches for parsing, one which test a multitude of linear assignments, one which tests some very deeply nested ifs and one which tests a single extremely complex expression. Further, I have some benchmarks that test performance for lexing and scanning. My findings are as follows, first of all I can comfortably handle 32 level of nesting before a stack overflow, which is nice[4]. Secondly, scanning and lexing account for remarkably big slice of the time the program spends on parsing. Third, troughput seems to decrease with the increase of input size, which seems to be caused by the decreased data locality (at least so I speculate). The result are as follows (troughput in MiB/s).

Statements	Scanner	Lexer	Parsing
128	4.3	28.7	13.2
256	2.2	29.0	13.0
512	1.1	27.8	11.8
1024	0.587	28.6	7.3
2048	0.288	28.3	6.4

Table 1: Troughput on Linear Program

Nesting	If	Expr
2	37.8	27.9
4	33.8	21.2
8	31.1	19.0
16	33.3	18.0
32	30.8	17.8

Table 2: Troughput on Non Linear Program

[4]Using an explicit stack would allow for an almost limitless number of levels, but anyone using more than 32 nested statement is a madman already

A couple of notes, first of all the result of the benchmark are a bit unfair, as the scanner is the only one which effectively iterates trough each of the input bytes over which the troughput is calculated. The lexer already (for example) skips string, comments and whitespace. Finally the parser operates only on prepackaged tokens, which speeds it up quite a bit. Also, I suggest to take the benchmarks in the nested cases with a grain of salt, as since the input size is smaller they are more prone to performance changes. In order to isolate the results of each stage, we collect the result of the previous stage, so not to have the on demand computation we used to have before.

Also, just to have some pictures, here are some flamegraphs: As you can

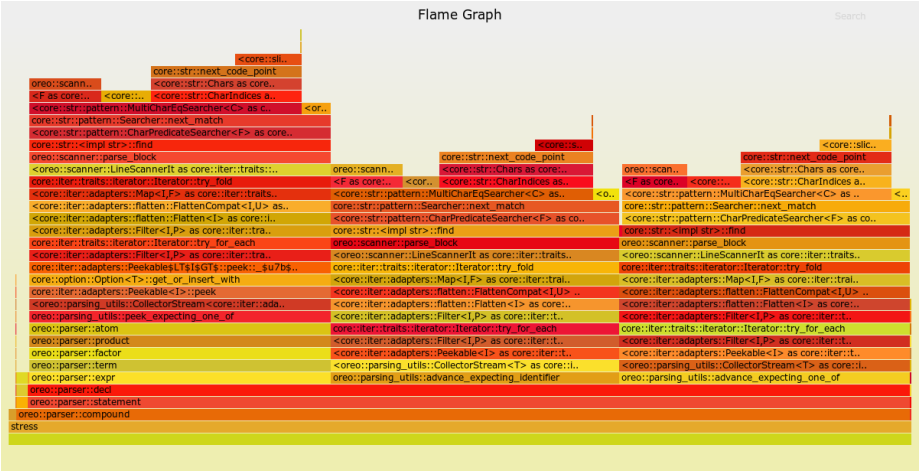

Figure 1: Flamegraph on demand parsing

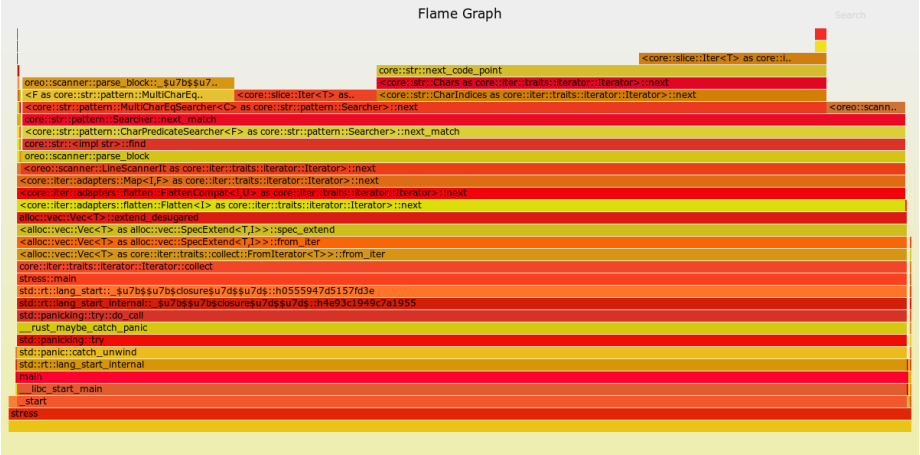

Figure 2: Flamegraph parsing after collecting tokens

see, in the on demand model each section takes approximately the same time,

put the actual impact of parsing is much more evident in the model where we precompute everything ahead of parsing. As you can see, (or actually as you can't see), the parsing in the second graph accounts for approximately 0.5% of the running time, and as such is definitely not the bottleneck. Most of the time is spent looking for characters in strings, which is what happens in both the scanner and the lexer.

4 Conclusion

I thoroughly enjoyed this practical, as it finally allowed me to practically develop a parser for a programming language which, despite not being terribly complex, had many real life problems to be addressed. The development of a grammar from the specification presented many challenges, and finding a workaround for some of the issues that it had was also a stimulating exercise. Having an occasion to use a solid and efficient testing framework also made this extremely nice. Finally, while performance was never a focus for this practical, I still found it worthwhile to analyze, as it showed the discrepancy between expectations and code.