Trieste: A C++ DSL for Flexible Tree Rewriting

Sylvan Clebsch
Microsoft Azure Research
Austin, USA
sylvan.clebsch@microsoft.com

Matilda Blomqvist
Uppsala University
Uppsala, Sweden
matilda.blomqvist@it.uu.se

Elias Castegren
Uppsala University
Uppsala, Sweden
elias.castegren@it.uu.se

Matthew A. Johnson
Microsoft Azure Research
Cambridge, United Kingdom
matjoh@microsoft.com

Matthew J. Parkinson
Microsoft Azure Research
Cambridge, United Kingdom
mattpark@microsoft.com

Abstract

Compilation is all about tree rewriting. In functional languages where all data is tree-shaped, tree rewriting is facilitated by pattern matching, but data immutability leads to copying for each update. In object-oriented languages like Java or C++, a standard approach is to use the visitor pattern, which increases modularization but also adds indirection and introduces boilerplate code. In this paper, we introduce Trieste – a novel tree-rewriting DSL, combining the power of C++ with the expressivity of pattern matching.

In Trieste, sequences of rewrite passes can be used to read a file to produce an abstract syntax tree (AST), convert from one AST to another, or write an AST to disk. Each pass rewrites an AST in place using subtree pattern matching, where the result is dynamically checked for well-formedness. Checking the well-formedness of trees dynamically enables flexibly changing the tree structure without having to define new data types for each intermediate representation. The well-formedness specification can also be used for scoped name binding and generating random well-formed trees for fuzz testing in addition to checking the shape of trees.

Trieste has been used to build fully compliant parsers for YAML and JSON, a transpiler from YAML to JSON, and a compiler and interpreter for the policy language Rego.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Parsers*; *Translator writing systems and compiler generators.*

Keywords: Rewriting, Compilers, Domain-Specific Languages

ACM Reference Format:
Sylvan Clebsch, Matilda Blomqvist, Elias Castegren, Matthew A. Johnson, and Matthew J. Parkinson. 2024. Trieste: A C++ DSL for Flexible Tree Rewriting. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3687997.3695647

SLE '24, October 20–21, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695647

1 Introduction

A compiler translates one language (*e.g.* source code, intermediate language) to another (*e.g.* assembler, bytecode) mainly by reading, rewriting, and writing trees. In functional languages, tree rewriting is facilitated by pattern matching but data immutability means each rewrite requires a copy. In object-oriented languages like Java and C++ rewrites can be performed in-place, but a naive implementation spreads functionality across classes. This is alleviated by techniques like the visitor pattern or specialized tools like Clang's AST Matcher [15]. Other tools facilitate language development by generating lexers [10, 12], parsers [8, 13] or even data types for syntax trees [7, 11], given some specification. At the end of the spectrum, language workbenches like Spoofax [9] can generate full language implementations from declarative specifications. Although it offers more specialized support, it restricts development to the tools given by the workbench.

A tool like ANTLR [11] takes a grammar and generates a full parser as well as other helpful tools like visitors and listeners for further processing of the syntax tree. However, as soon as you move away from the original syntax tree (for example to some intermediate representation), you must define new data types, including boilerplate methods for traversal, listeners and visitors.

This paper presents our ongoing work on Trieste, a DSL for tree rewriting embedded in C++. It is designed to incorporate useful features from these different approaches. The novelty of Trieste is its flexibility: the expected structure of the tree can be changed in every rewrite pass without having to define new data types or visitors. The shape of a tree is instead checked dynamically against a *well-formedness (WF) specification* connected to each rewrite pass. Not having to define new data types for every change to the tree encourages writing applications with many small passes operating on an evolving tree shape. The same specification also enables the generation of random trees for fuzz testing.

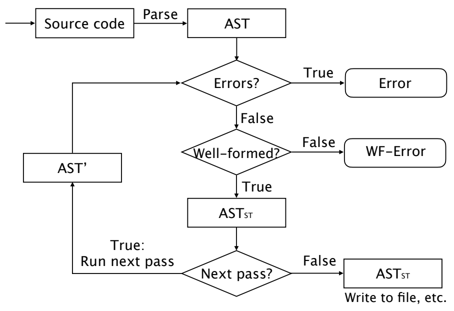

Figure 1. Schematic overview of a Trieste program. The resulting AST of a rewrite pass is checked for errors and well-formedness before continuing.

In this paper, we give an overview of Trieste (Section 2) and present two larger case studies (Sections 3 and 4) before discussing insights and future work (Section 5). Trieste is available as open source [17].

2 Trieste by Example

In this section, we introduce Trieste using (simplified) examples from a compiler frontend for MiniML [1]. The examples are shown in context in Appendix A, and the full code can be accessed online [4].

Figure 1 gives an overview of a general Trieste program. An initial parsing pass translates one or more source files into a first abstract syntax tree (AST) (Section 2.1). The tree data type in Trieste has no prescribed shape and is the same for all Trieste programs. Instead, each pass has an associated WF specification that describes the expected tree structure after applying that pass (Section 2.2). After a pass has run successfully, the tree is checked to comply with its WF specification. The rest of the program is a sequence of *rewrite passes* (Section 2.3). A rewrite pass contains a sequence of rewrite rules that are run once or until fixpoint. A rewrite rule uses pattern matching to select subtrees that should be rewritten. Trieste also provides support for scoped name binding by allowing nodes to hold symbol tables which can be queried for bound names (Section 2.4).

2.1 Parsing

The parsing pass translates a source program into an initial tree. Nodes in the tree have a *token* (the type of the node), a value (typically the corresponding string in the source file) and a vector of children. Except for a few built-in tokens, tokens are language-specific and defined by the language developer as `TokenDef` objects given a unique name and optional flags. The following example shows the token definitions for integers, the addition operator and parentheses:

```
inline const TokenDef Int = TokenDef("int", flag::print);
inline const TokenDef Add = TokenDef("+");
inline const TokenDef Paren = TokenDef("()");
```

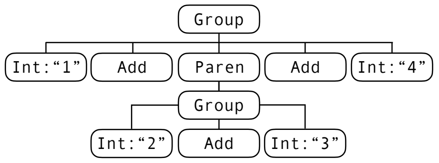

Figure 2. The parse tree for `1+(2+3)+4`. The `Int` token has the `print` flag which means that it is printed with its value.

The `Int` token is defined with the flag `print` meaning that the value of an `Int` node is included when printing the tree.

The parser is implemented as a sequence of parse rules which are tried in order. A parse rule takes the general form `regex >> parse effect`, where the parse effect is a function (here written as a C++ lambda function) that inserts nodes in the parse tree. Trieste implicitly groups parsed tokens under explicit `Group` nodes, allowing the language developer to defer some nesting to later rewrite passes, similar to the parsing of shrubbery notation in Rhombus [6]. As an example, consider the parse rules for the above token definitions:

```
"[[:digit:]]+\\b" >> [](Make& m) {m.add(Int); } // Integers
"+" >> [](Make& m) {m.add(Add); } // '+'
"\\(" >> [](Make& m) {m.push(Paren); } // Enter Paren node
"\\)" >> [](Make& m) {m.term(); m.pop(Paren); } // Exit Paren node
```

Figure 2 shows the result of parsing "`1+(2+3)+4`" with these rules. The "1" matches the first rule, which adds an `Int` node with value "1". Whenever a node is added, a `Group` node is implicitly created as its parent unless it already exists. The "+" matches the second rule that adds an `Add` node as a sibling to the `Int` node. The left parenthesis matches the third rule that *pushes* a `Paren` node to the tree, which adds a `Paren` node and causes subsequent nodes be added as children. When the "2" is added a new `Group` node is created, which becomes the parent of the "2" and the subsequent "+" and "3". The right parenthesis is parsed with the fourth rule that first *terminates* the current `Group` (`m.term()`) and then terminates the `Paren` node (`m.pop(Paren)`) such that the subsequent nodes for "+" and "4" are added as siblings instead of children. The difference between `term()` and `pop()` is that `pop()` reports an error if we are not terminating the expected node.

The parser also has a *mode* feature for grouping parse rules and switching between them (similar modes exist in ANTLR [11]). The parse rule for opening comments in the default mode (not shown here) switches to a separate `"comment"` mode, with three parse rules:

```
p("comment", { // Comment mode
  "\\(\\*" >> [depth](Make&){ ++(*depth); }, // Increase depth
  "\\*\\)" >> [depth](Make& m){
      // Decrease depth, switch back to default mode if depth is zero
      if (--(*depth) == 0) {
        m.mode("start"); } },
  ".|\n" >> [](Make&){} }); // Ignore other tokens
```

A variable `depth` is used to ensure correct nesting, which is checked in the rule for `*)` to determine whether to escape the comment mode and return to the default mode `"start"`.

While introducing simple nesting with push and pop is useful, it is not required, nor does the implementer need to define more than a single mode. Using add() as the only parse effect will produce a flat tree with a single group of tokens, essentially giving the same result as a lexer.

2.2 Well-Formedness Checking

Each pass has a WF specification serving as its postcondition. The main purpose is checking that a rewrite pass results in a tree of the expected shape. A failed check signals a bug in that pass. The WF specification is quite general after the initial parsing pass since most nodes are either a child of a Group node or have a sequence of Group nodes as children. It then becomes increasingly specific with each rewrite pass.

A WF specification consists of a list of *shapes*. A shape takes the form parent_token <<= child_tokens and specifies the shape of a subtree with the parent token as root. As an example, consider the WF specification for a pass rewriting multiplicative expressions:

```
inline const wf::Wellformed wf_mul = wf_funapp // Previous WF
| (Expr <<= (Mul | Add | /*...*/ | Int)++[1]) // Expr has ≥ 1 children
| (Mul <<= (Lhs >>= Expr) * (Rhs >>= Expr)); // Lhs and Rhs are labels
```

The above specification starts by copying the WF specification from the previous pass (wf_funapp) and then adds new shapes. The inner | denotes choice: we specify that Expr nodes can only have children that are Mul, Add or Int *etc.* Furthermore, they must have at least one child (++[1]). The * denotes sequence: Mul nodes must have exactly two children of type Expr. These are labelled Lhs and Rhs respectively, using the >>= operator. Labels are optional and can be used as selectors: given a Mul node m, the left child can be accessed by m/Lhs. The labels themselves are TokenDef objects, allowing fast comparisons. During pass execution, the labels used for selection is dictated by the previous and current WF specifications. Arbitrary specifications can also be used outside of a pass sequence to label and access nodes.

Just as the WF specification acts as the postcondition of one pass, it is also the precondition of the following pass. The WF specifications can therefore be used to generate random well-formed trees for any pass. Using this, Trieste provides automated fuzz-testing of any sequence of rewrite passes, giving developers a simple way to find valid inputs that trigger errors in their passes.

2.3 Rewriting

A Trieste program contains a sequence of rewrite passes. A rewrite pass, in turn, consists of a sequence of *rewrite rules*. Each pass has a specified direction of traversal, bottom-up or top-down, and is run once or until fixpoint. Trieste tries to apply all rules at the current node before continuing to the next node in traversal order. The sequence of rewrite rules is analogous to a *rewrite strategy* as defined in languages like Stratego [19] and ELAN [2], but with an implicit sequence operator between the rewrite rules.

A rewrite rule has the general form pattern >> effect. A pattern describes the structure of a subtree, and an effect is a function from the matched subtree to a modified subtree that should replace the matched one. As an example, consider the following rule from the pass that rewrites additive expressions in the same way multiplications were structured in the WF specification in the previous section:

```
In(Expr) * T(Expr)[Lhs] * T(Add)[Op] * T(Expr)[Rhs] >>
  [](Match& m){ return Expr << (m(Op) << m[Lhs] << m[Rhs]); }
```

The * denotes a sequence: the pattern in the rule above matches three nodes: an Expr node followed by an Add node and another Expr node, all of which are immediate children of an Expr node (In(Expr)). The resulting node is wrapped in an Expr node since it might be a subexpression in a larger expression. Patterns may be labelled using the [] operator which makes it possible to access specific parts of the matched subtree m within the effect. Note that this rule will match a subtree as long as the correct sequence is found inside an Expr node – what comes before and after these nodes is not specified. In the effect, << is used to denote a parent/child relationship: after this rewrite rule is applied the Add node will be the parent of its previous siblings. The full pass definition and an example of an input-output tree are given in Appendix A.

The pattern can be extended to also handle subtraction by replacing T(Add) with T(Add,Sub) (T lists a disjunction of tokens) instead of adding an almost identical rule for subtraction. Addition and subtraction can be rewritten by the same rule within the same pass since they have the same precedence. On the other hand, multiplication has higher precedence than addition and is, therefore, processed in a prior (albeit very similar) pass. This rewrite rule changes the structure of additive expressions and the WF specification is updated accordingly:

```
inline const wf::Wellformed wf_add = wf_mul // Previous WF
| (Add <<= (Lhs >>= Expr) * (Rhs >>= Expr))
| (Sub <<= (Lhs >>= Expr) * (Rhs >>= Expr))
```

Rewrite passes can be further modularized into *readers*, *rewriters* and *writers*. The reader contains a parsing pass and possibly further rewrite passes. A rewriter contains passes to transform one tree to another (see Section 3.2). Finally, the writer starts from a tree, performs some rewrites, and then writes the final tree to a file. This modularization allows reusing code and building pipelines of rewrite passes.

2.4 Name Binding with Symbol Tables

Most programming languages support some form of scoped name binding. In Trieste, nodes with certain tokens can be specified to hold *symbol tables* which map node values (*e.g.* variable names) to nodes in the tree (*e.g.* the definition where the variable is bound). The scope of the symbol table covers every descendant of its node. Scopes may be nested.

In MiniML, each function introduces a new scope in addition to the global program scope. Therefore, every node of type `Fun` and `Program` will hold a symbol table:

```
inline const TokenDef Fun = TokenDef("fun", flag::symtab);
inline const TokenDef Program = TokenDef("program",
     flag::symtab | flag::defbeforeuse);
```

The symbol table in a `Fun` node maps parameters and function names to their respective bind sites. The symbol table in a `Program` node maps let-bound names to their definitions (let is the only top-level definition in MiniML). The `Program` node also has the `defbeforeuse` flag meaning that a lookup in this symbol table will not return anything if the symbol being looked up occurs before its definition in the source file.

Symbol tables are used in our MiniML compiler frontend for accessing types of bound identifiers when checking types of (identifier) expressions. Symbol table bindings are defined in the WF specification. Consider the following WF specification for a first type inference pass that adds fresh type variables to bound names (shapes irrelevant to symbol table bindings are omitted):

```
inline const wf::WellFormed wf_fresh = reader_::wf // Previous WF
| (Fun <<= FunDef) // Holds symbol table for Fun name and Param
| (FunDef <<= Ident * Type * Param * Expr)[Ident] // Bind functions
| (Param <<= Ident * Type)[Ident] // Bind parameters
| (Let <<= Ident * Type * Expr)[Ident]; // Bind let definitions
```

The `[Ident]` after the `FunDef`, `Param` and `Let` shape tells us that each node of these types should be added to their closest nesting symbol table with their `Ident` value as lookup key. For this to work, their `TokenDefs` must have the `lookup` flag:

```
inline const TokenDef Param =
  TokenDef("param", flag::lookup | flag::shadowing);
inline const TokenDef FunDef =
  TokenDef("fundef", flag::lookup | flag::shadowing);
inline const TokenDef Let = TokenDef("let", flag::lookup);
```

The `Param` and `FunDef` tokens also have the flag `shadowing`: storing a node with any of these tokens in a symbol table shadows any other nodes with the same key in nesting scopes and prohibits nodes with the same key from being bound in the same scope.

Symbol tables are populated between two rewrite passes, meaning that it is the WF specification of the previous pass that defines how nodes are bound. Thus, after the first inference pass it is possible to look up names in the symbol tables. For example, consider a rule for inferring types of identifier expressions in a later pass:

```
T(Expr) << T(Ident)[Ident] >>
[](Match& m) {
  Nodes defs = m(Ident)->lookup(); // Symbol table lookup
  if (defs.size() != 0){
    Node def = (defs.back())->clone(); //Latest definition
    return Expr << def/Type << m(Ident); }
  else {
    return err(m(Expr),"unbound name" + m(Ident)); } },
```

The rule matches an `Expr` node whose child is an `Ident`. The `lookup()` method returns a vector of `Fun`,`Param` or `Let` nodes whose `Ident` node has the same value as the matched one. As MiniML allows multiple let bindings with the same name in the same scope we take the last element of the vector, *i.e.* the latest binding. The `/` operator for field accesses comes in handy here – all nodes that can be looked up have a child implicitly labelled `Type` so `def/Type` gives the `Type` node.

If the lookup returns an empty vector, no identifier with this value was bound to a symbol table in scope and the matched `Expr` node is replaced by an `Error` node (abstracted into the `err()` function). The `Error` tokens are pre-defined and are used to signal any kind of user-facing errors during parsing or rewriting.

3 Case Study: YAML and JSON

The dynamic nature of Trieste trees eases the manipulation of data in multiple formats within the same codebase. To demonstrate this, we provide fully compliant parsers for YAML [14] and JSON [3], as well as a rewriter between the two. They are available in the Trieste repository [17].

3.1 JSON

The JSON implementation in Trieste exposes this API:

```
inline const wf::Wellformed wf =
  (Top <<= wf_value_tokens++[1])
  | (Object <<= Member++)
  | (Member <<= Key * (Value >>= wf_value_tokens))[Key]
  | (Array <<= wf_value_tokens++);
Reader reader(bool allow_multiple);
Writer writer(const std::filesystem::path& path,
              bool prettyprint, bool sort_keys,
              const std::string& indent);
```

The reader comprises three passes:

1. **Parse**: Tokenizes input with nested braces/brackets
2. **Groups**: Specializes the `Group` nodes from the parser to `ArrayGroup` and `ObjectGroup` nodes.
3. **Structure**: Constructs `Array` and `Object` structures.

The writer can be used to output a well-formed AST as a JSON file (by default, without whitespace). Readers and writers can be chained together (for example, to create a minifier):

```
json::reader().file("in.json") >> json::writer("in.min.json");
```

The `json::writer` function takes a file path as an argument and creates a new `Writer` object with a single rewrite pass, that traverses the tree once and consists of a single rule:

```
In(Top) * ValueToken++[Value] >> // Match a sequence of ValueTokens
  [path](Match& m) {
    return File << (Path ^ path.string()) // Add path to tree
             << (Contents << m[Value]); }
```

The ++ pattern greedily matches a sequence of zero or more `ValueTokens` under the `Top` node. It is replaced by a `File` node with a new `Path` node carrying the given path, and a `Contents` node grouping the sequence of `ValueTokens`. This is the structure that a `Writer` object expects. Note that `m[Value]` gives the matching *sequence* of `ValueTokens` – `m(Value)` used in previous examples gives a single node.

3.2 YAML

YAML is a more complex data language than JSON, adding meaningful whitespace, default values, aliases/anchors, block literals, and more. The YAML reader comprises 16 passes, including passes for parsing and grouping, nesting by indentation, handling and block and flow-style constructs, and validating tags and anchors. Trieste's design allows for each pass to focus on a single task which makes a small edit to the WF specification. For example, here is the specification for the attributes pass which handles node tags and anchors:

```
const wf::Choice attr_tokens = (col_tokens | AnchorVal | TagVal);
const wf::Choice attrval_tokens =
  attr_tokens - (DocumentStart | DocumentEnd);
const wf::Choice attr_flow_tokens =
  fgroup_tokens | AnchorVal | TagVal;
const wf::Wellformed wf_attr = wf_coll
  | (AnchorVal <<= Anchor * (Val >>= attrval_tokens))
  | (TagVal <<= TagPrefix * TagName * (Val >>= attrval_tokens))
  | (DocumentGroup <<= attr_tokens++)
  | (FlowGroup <<= attrflow_tokens++)
  | (KeyGroup <<= attrvalue_tokens++)
  | (ValueGroup <<= attrvalue_tokens++);
```

Note how the token lists can be expanded with the | operator, or reduced with the - operator.

In addition to a reader, the YAML implementation provides writers for both standard YAML and YAML events, as well as a rewriter for conversion from YAML to JSON. As such, one can write a YAML to JSON converter in a single line:

```
yaml::reader().file("input.yaml") >> yaml::to_json() >>
  json::writer("output.json");
```

4 Case Study: Rego

The rego-cpp compiler and runtime for Rego (the policy language of the Open Policy Agent [5]) is fully implemented in Trieste [16]. Whereas the reader exposed by the rego-cpp implementation is similar to the JSON and YAML readers above in that it parses input files and creates ASTs, the unify rewriter is quite different. It takes as input an AST which merges one or more ASTs (from data files, modules, or input terms) with a query. The passes then resolve the query by performing unification. While the final step happens in pure C++, most of the work is done by Trieste passes which transform the AST into a form which is more easily unified.

These passes go beyond simple parsing to actions like lifting comprehensions as rules, or performing capture analysis. The hybrid nature of Trieste (as a DSL for tree rewriting within a C++ program) is what enables this. For example, here is a rule which uses an *action* in the pattern (the lambda in parentheses) as a further constraint on the pattern, adding a deep analysis (here, whether a term is constant):

```
In(RuleComp, RuleFunc, RuleSet, DefaultRule) *
  T(Term)[Term]([](NodeRange& n) { // Action lambda
    return is_constant(n.front()); }) >>
  [](Match& m) { return DataTerm << *m[Term]; }
```

The prefix * in the effect gets the children of its operand. As another example, this code supplies a pre-condition action for a pass definition:

```
unify.pre(Rego, [builtins](Node node) {
  Node query = node / Query;
  try {
    Nodes results = Resolver::resolve_query(query, builtins);
    node->replace(query, Query << results);
  } catch (const std::exception& e) {
      node->replace(query, err(query, e.what()));
  } return 0; });
```

This code will execute once for each Rego node, replacing the query with the results of the unification.

5 Discussion and Future Work

In Trieste, the shape of the AST evolves with each rewrite pass. The full grammar of the abstract syntax does not need to be specified upfront; we can explore a suitable representation one rewrite pass at a time. We believe this makes Trieste suitable for language prototyping: some kinds of nodes can be processed carefully, while others can be ignored – rewrite passes will still traverse the entire tree and simply ignore subtrees that never match. Adding a new kind of node consists of adding a new token definition. No class hierarchies or visitor classes need to be extended, and existing rewrite passes will continue to work as before.

While selectors (*e.g.* n/Lhs) facilitate interacting with nodes when dropping to C++, the dynamic tree shape means that the compiler will not stop you from using incorrect labels to access node (in contrast to having static types). Since the shape of a tree can change even within a single pass, having pattern matching is crucial for safely updating the tree.

Name bindings, selectors, and specifications never silently go out of sync since the WF specification is used for all three. We have also found that the automatic fuzz testing (also provided using the WF specification) helps uncover mistakes in rewrite passes. Failed fuzz tests implicitly provide hints for appropriate Error-rules and catch corner cases.

In ongoing and future work we are defining the formal semantics of Trieste in order to explore new designs and prove properties of the language. Among other things we are looking at improving the expressivity of WF specifications, automatically detecting patterns which are malformed or which will never match, and verifying the soundness of future optimizations.

In addition to the examples and case studies presented in this paper, Trieste has been used to implement a bytecode compiler for a Python-like language with linear types and in the development of the Verona compiler [18], which targets LLVM bytecode. Trieste is available as open source [17].

A Trieste Examples in Context

The following example shows the definition of our MiniML parser (omitting most rules for brevity). Note the comment rules that switch between two parse modes "start" and "comment". The done method takes a callback for when input is exhausted. It closes any open Groups by default. Here it is extended to check for unterminated comments and close the statement separator ;;.

```
Parse parser() {
  Parse p(depth::file, wf_parser); // Create parse object
  auto depth = std::make_shared<size_t>(0); // Nesting depth
  p("start", // Add parse rules to default mode
    { "[[:digit:]]+\\b" >> [](Make& m) { m.add(Int); },
      "+" >> [](Make& m) { m.add(Add); },
      "\\(" >> [](Make& m) { m.push(Paren); },
      "\\)" >> [](Make& m) { m.term(); m.pop(Paren); },
      "\\(\\*" >> [depth](Make& m) { // Open comment
        ++(*depth); // Increase nesting depth of comment
        m.mode("comment"); }, // Switch to comment mode
      /* Remaining parse rules omitted for brevity */
   });
  p("comment", // Comment mode
    { "\\(\\*" >> [depth](Make&) { ++(*depth); }, // Increase depth
      "\\*\\)" >> [depth](Make& m) { // Decrease depth
        // Switch back to the start mode if depth == 0
        if (--(*depth) == 0) m.mode("start"); },
      ".|\n" >> [](Make&) {} // Ignore other tokens
    });
  p.done([depth](Make& m) // Called when parsing is done
    { *depth = 0; // Reset depth value
      if (m.mode() == "comment")
        m.error("Unterminated comment");
      m.term({SemiSemi}); // Terminate group and ;;
    });
  return p; } // Return parse object
```

A rewrite rule for additive expressions was shown in Section 2.3, here the complete pass definition is given. The pattern -- succeeds only if the provided pattern does *not* match:

```
PassDef add_sub(){
  return { "add_sub", wf_add, dir::topdown, {
    In(Expr) * (T(Expr)[Lhs] * T(Add,Sub)[Op] * T(Expr)[Rhs]) >>
      [](Match& m){ return Expr << (m(Op) << m[Lhs] << m[Rhs]); },
    // Return an Error node if we don't have exactly two Expr nodes as children
    T(Add,Sub)[Add] << --(T(Expr) * T(Expr) * End) >>
      [](Match& m){ return err(m[Add], "invalid expression"); }
  }}; }
```

Figure 3 shows the tree from Figure 2 immediately before and after the add_sub pass. Group and Paren nodes from the initial parse tree have been replaced by Expr nodes in a previous pass. The add_sub pass associates Add and Sub nodes with their operands, giving a shape that is more like an AST than in Figure 2. The superfluous Expr node is a trade-off for keeping the rules simple and is removed by a later pass.

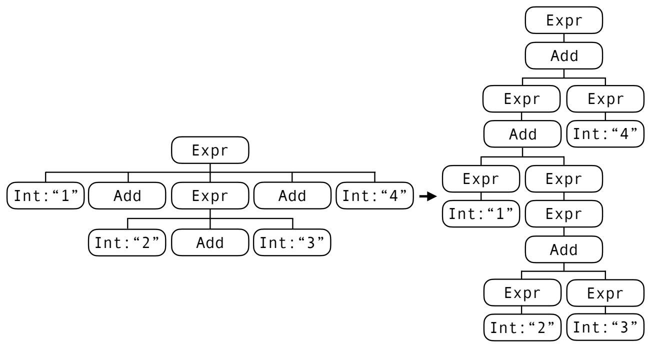

Figure 3. The intermediate tree representations of "1+(2+3)+4" immediately before and after the add_sub pass.

References

[1] Andrej Bauer. 2024. miniml. https://plzoo.andrej.com/language/miniml.html. Accessed: 2024-06-27.
[2] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. 2002. ELAN from a rewriting logic point of view. *Theoretical Computer Science* 285, 2 (Aug. 2002), 155–185. https://doi.org/10.1016/S0304-3975(01)00358-9
[3] T. Bray. 2017. RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format.
[4] Sylvan Clebsch, Matilda Blomqvist, Elias Castegren, Matthew A. Johnson, and Matthew J. Parkinson. 2024. trieste-miniML. https://github.com/fxpl/trieste-miniml.
[5] Open Policy Agent contributors. 2024. Rego. https://www.openpolicyagent.org/docs/latest/policy-language/. Accessed: 2024-06-18.
[6] Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 242 (oct 2023), 30 pages. https://doi.org/10.1145/3622818
[7] Markus Forsberg and Aarne Ranta. 2004. BNF converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. https://doi.org/10.1145/1017472.1017475
[8] Stephen C Johnson. 1978. Yacc: Yet another compiler-compiler.
[9] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax language workbench. In *OOPSLA '10*. Association for Computing Machinery, New York, NY, USA, 237–238. https://doi.org/10.1145/1869542.1869592
[10] Simon Marlow and the Alex developers. 2024. Alex. https://github.com/haskell/alex. Accessed: 2024-06-19.
[11] Terence Parr. 2024. ANTLR. https://www.antlr.org/about.html. Accessed: 2024-06-19.
[12] Vern Paxson et al. 2024. Flex. https://github.com/westes/flex. Accessed: 2024-06-19.
[13] GNU Project. 2024. Bison. https://www.gnu.org/software/bison/. Accessed: 2024-06-19.
[14] YAML Language Development Team. 2024. YAML 1.2.2 Specification. https://yaml.org/spec/1.2.2/. Accessed: 2024-06-18.
[15] Contributors to LLVM. 2024. LibASTMatchers. https://clang.llvm.org/docs/LibASTMatchersReference.html. Accessed: 2024-06-19.
[16] Contributors to the rego-cpp project. 2024. rego-cpp. https://github.com/microsoft/rego-cpp.
[17] Contributors to the Trieste project. 2024. Project Trieste. https://github.com/microsoft/Trieste.
[18] Contributors to the Verona project. 2024. The Verona Language. https://github.com/microsoft/verona.
[19] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In *Rewriting Techniques and Applications*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45127-7_27

Received 2024-06-24; accepted 2024-08-30