

# Generating GUI based Domain Specific languages

A Major Qualifying Project (MQP) Report  
Submitted to the Faculty of  
WORCESTER POLYTECHNIC INSTITUTE  
in partial fulfillment of the requirements  
for the Degree of Bachelor of Science in

Computer Science,

By:

Finn Wander  
Peter Husman

Project Advisors:

Garry Pollice

Date: Apr 2023

*This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.*

## **Abstract**

A modification of Earley parsing is presented that allows incremental parsing of incomplete user input to provide some text editing features in a structured editor. A code generator produces files for the editor based on an input grammar. We make use of the tools presented in this project to make a vector graphics toy language and a tool for learning LLVM IR Builder.

# Acknowledgements

1. Michael Engling for his help as a temporary advisor.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	LLVM . . . . .	2
2.1.1	LLVM IR and IR Builder . . . . .	2
2.2	Syntax Directed Editors . . . . .	5
2.3	Context Free Grammars . . . . .	7
2.4	Earley Parsing . . . . .	8
2.5	Error Correcting Earley Parsing . . . . .	9
2.6	Incremental LR Parsing . . . . .	9
2.7	Tools and Technologies Used . . . . .	10
<b>3</b>	<b>Methodology</b>	<b>11</b>
3.1	Timeline of the project . . . . .	11
3.2	Architecture overview . . . . .	14
3.3	GUI generation . . . . .	14
3.4	Parsing input . . . . .	21
3.4.1	Imagination and Concreteness . . . . .	21
3.4.2	Recovering a Parse After Earley Parsing . . . . .	25
3.5	Incremental Parsing . . . . .	30
<b>4</b>	<b>Results</b>	<b>31</b>
4.1	Code Metrics . . . . .	31
4.2	Performance . . . . .	31
4.3	Toy Language . . . . .	32
4.4	Ambiguity Resolution . . . . .	32
4.5	Future Work . . . . .	32
	<b>Appendices</b>	<b>35</b>
	<b>A Grammar for Toy Language</b>	<b>35</b>
	<b>B Grammar File for LLVM Pattern Tool</b>	<b>36</b>
	<b>References</b>	<b>37</b>

# List of Figures

1	An example of Human Readable LLVM IR . . . . .	3
2	Architecture Diagram . . . . .	15
3	A grammar and two partial parses for “ba” on that grammar. Imagined symbols are depicted in blue. figure 7c depicts a minimal partial parse while figure 7d does not because the second tree does not minimize imagined symbols. This demonstrates that Non terminals can be imagined. . . . .	22
4	Grammar before and after normalization . . . . .	24
5	A Parse Forest for the input “aa” for the grammar in figure 3. Many simplifications were made to this representation. Edges leave out the number of times they can be traversed and imagined non terminal symbols have been left out. . . . .	26
6	An example program and its output. . . . .	32
7	Resulting tree structure before and after edit at different cursor positions. Note the cursor was moved after the edit to show the structure of the resulting trees . . . . .	33

# 1 Introduction

This MQP was originally set out to address a critical lack of beginner friendly documentation of LLVM IR Builder. This MQP investigated the current state of resources for learning LLVM IR Builder and identified opportunities for improvement. It also examined existing tools and resources that are available for learning the LLVM IR Builder, and explored the challenges that students could be facing as they are learning this technology.

After much iteration, the project objective was to provide modifiable templates that would take in user input and provide well formed and documented code to interface with the LLVM C++ API. The code for that tool was then abstracted because the inputs for the templates could be thought of as a context free grammar. In turn, the project shifted scope to be about a tool that would translate this abstracted version of user input to a GUI that could collect user input into an abstract syntax tree.

There are many components to getting such a tool right and this project explored a few of them. These aspects included visual customizability, normalizing input grammar, recovering abstract syntax trees after the normalization, dealing with intuitive user navigation through the tree, and parsing user input while allowing for incomplete input.

The Background section will cover topics in LLVM to provide an understanding for the kinds of concepts we wanted to convey in the templates. It will also cover concepts in parsing such as context free grammars, Earley parsing, and incremental parsing. We will also cover related work in structured editors, and the tools and technologies used in this project. The Methodology section will cover the project timeline, as well as a general architecture overview. It will also cover the challenges faced throughout this project. The results section will outline the usability of the tools we have made as well as a toy language which could be used for user studies.

## 2 Background

### 2.1 LLVM

Today's software applications have grown increasingly complex, large, and are often created using various programming languages. This requires ongoing analysis and adjustments throughout the lifetime of the software. The LLVM compiler framework is designed to enable these continuous transformations for all kinds of software in a seamless way. To achieve this, LLVM relies on two main components:

1. A distinctive code representation that acts as a universal medium for analysis, modification, and code sharing;
2. A compiler architecture that takes advantage of this representation, offering a unique blend of features not present in earlier compilation approaches.

The LLVM code representation employs an abstract RISC-style instruction set to portray a program, while incorporating vital high-level details to enable effective analysis. This includes type information, well-defined control flow graphs, and a clear dataflow representation. LLVM maintains its independence from specific source languages by using a low-level instruction set and memory model, which are only slightly more feature rich than traditional assembly languages [Lattner and Adve, 2004].

#### 2.1.1 LLVM IR and IR Builder

LLVM IR is a virtual instruction set That can be represented in 3 forms: plain-text, binary, and in-memory. The plain-text representation is based on a human-readable syntax, while the binary representation is a more compact form used for storage and transmission. The in-memory representation is used for optimization and analysis, and it is designed to be

```

define i64 @foo(ptr %0, ptr %1, ptr %2, i8 %3) {
  store ptr %1, ptr @x
  %5 = getelementptr inbounds i8, ptr %2, i64 5
  %6 = load i8, ptr %5
  %7 = getelementptr inbounds i8, ptr %2, i8 %3
  tail call void @bar(i8 %3, ptr %7)
  %8 = load i64, ptr %0
  ret i64 %8
}

```

Figure 1: An example of Human Readable LLVM IR  
[noa, 2023]

both efficient and flexible. [noa, 2023]

LLVM has a collection of classes that are used to represent modules, functions, basic blocks, and instructions in IR, this is the "in program memory" representation of IR. For example the `AllocaInst` object represents an `alloca` instruction in IR, this makes space the stack for a given type and returns a pointer to that space for you. To create an `alloca` instruction in a program, you could either create the `AllocaInst` object and then add it in the correct location of a basic block object, that you've already added to a function object, that you've already added to a Module object, or, you could use the IR Builder to manage all of that for you. The IR Builder class keeps track of where it is adding instructions and then exposes an interface that lets you make a call to `CreateAlloca()`. This creates an `alloca` instruction for you in a program at the location after you've last inserted an instruction.

## Virtual Registers and Values

IR uses virtual registers to represent values. Virtual registers provide a more abstract representation of values than hardware registers in two ways. 1; they can represent aggregate types such as structs or arrays. 2; there is an unbounded amount of them.

Virtual registers are used in LLVM to facilitate the implementation of Single Static Assignment (SSA) form. SSA is a representation of a program in which each variable is



assigned exactly once. Unique virtual register names are required due to SSA representation. The large number of available names solves the uniqueness problem. For example, you can name two registers `%reg.1` and `%reg.2`, and they would be unique. You could also name them `%1`, `%2`, `%3` and so on. Virtual registers are mapped to physical registers of a target architecture. Register allocation assigns physical registers to virtual registers for optimal performance.

Every instruction in IR returns a Value pointer. Using the example from before, the `alloca` instruction returns a pointer to allocated memory. This value is then stored in a virtual register. It is important to understand that Value objects don't actually store data representing the value itself, rather, they store a representation of how that value is computed. This is evidenced by the fact that instruction objects are values. Values also take in other values as parameters, so a value becomes a nested tree of computation. [Lattner and Adve, 2004]

## Basic Blocks and Control Flow

In LLVM IR, a basic block is a set of instructions that are guaranteed to execute in order. There is just one entry point and one exit point for each basic block. A basic block's final instruction must be a terminator instruction like a branch, jump, return, or call. These terminating instructions are Value objects too, however usually being void typed; they represent control semantics rather than data. Because you can't have branching statements in the middle of a basic block, to achieve control constructs in LLVM you need to chain multiple of these basic blocks together. It is simpler to perform optimization and to perform register allocation when the code has been broken into these continuous, non-interrupted blocks of execution. [noa, 2023]

In program memory, basic blocks have an object representation. They are stored in `BasicBlock` objects. The IR Builder can add instructions to a basic block by calling

the `SetInsertPoint` function with a `BasicBlock` object. After that call, instructions created through the IR Builder will be added to that `BasicBlock`. There is also a function called `Create` in the `BasicBlock` namespace that takes in a context, name, and `Function` object, and will return a `BasicBlock` pointer that belongs to that function.

## Types

Every `Value` object in LLVM has an associated `Type` object. LLVM has a type system that includes many categories. Primitive types, Void types, Function types, First-class types, Single value types, and Aggregate Types are the different categories of LLVM IR types. [noa, 2023] A high-level interface for creating and modifying these kinds is provided by LLVM IRBuilder. Because of the robust type system of LLVM IR, optimizations may be carried out with less analysis, leading to more effective code generation and optimization.

The initial reason this UI generator was created was to serve as the front-end for a tool for learning about LLVM IR Builder. LLVM is a useful tool that has been adopted by many prominent software projects such as `rustc` and `clang`. However, the use of LLVM can be challenging for students who are new to this technology. In particular, the LLVM IR Builder, a component of LLVM that is responsible for generating the intermediate representation of the source code, can be difficult to understand and use effectively due to a lack of clear documentation.

## 2.2 Syntax Directed Editors

Syntax directed editors can enhance the programming experience by harnessing the structure and grammar of programming languages. By enforcing syntactic correctness, editors help minimize syntax errors. This allows developers to focus on the logic of their code [Khwaja and Urban, 1993].

Two main categories of syntax-directed editors exist:

1. **Projectional Editors:** These editors enable users to directly manipulate the AST. Bypassing the textual representation of code allows projectional editors to offer powerful features such as language extensibility, language composition, and structural refactoring [Völter and Solomatov, 2010]. Projectional editors allow language designers to specify multiple projections for the same program, enabling a more intuitive editing experience for different tasks. JetBrains Meta Programming System (MPS) is a notable example of a projectional editor.
2. **Structure Editors:** Similar to projectional editors, structure editors use ASTs as their primary representation. However, they offer a more text-like editing experience. The AST is displayed as a visually editable representation resembling text-based code, with placeholders, delimiters, and other visual elements guiding user interaction. The Synthesizer Generator is a tool designed to generate syntax-directed editors for various programming languages. By accepting a language’s grammar specification as input, it produces a syntax-directed editor tailored to that language. The generated editor offers a structured editing experience, enforces syntactic correctness, and supports incremental compilation.[Reps and Teitelbaum, 1984]

The Hazel editor, a bidirectionally typed editor, is designed to uphold well-typed programs, even when incomplete or containing errors. It utilizes a formal calculus called Hazelnut, which provides syntax, bidirectional typing rules, action semantics, and cursor movement rules for handling editing actions and maintaining the abstract syntax tree’s integrity [Omar et al., 2017]. Incorporating the Hazelnut calculus enables Hazel to handle editing in a powerful and user-friendly programming environment.

“bidirectional” refers to the typing rules used in the Hazelnut calculus. Bidirectional typing combines two approaches to type checking: type synthesis, which infers the type of an expression based on its sub-expressions (bottom-up), and type checking, which checks if

an expression has a specific type based on the context in which it appears (top-down). The Hazelnut calculus can efficiently propagate type information throughout the AST during editing, ensuring that the code remains well-typed [Omar et al., 2017].

## 2.3 Context Free Grammars

A context free grammar is defined by  $G = (V, \Sigma, R, S)$ . Where

1.  $V$  is a set containing the non-terminals of a grammar
2.  $\Sigma$  is a set containing the alphabet of terminals of the context-free language defined by  $G$  (with  $\Sigma \cap V = \emptyset$ )
3.  $R \subset V \times (V \cup \Sigma)^*$  is a binary relation representing the rules of the grammar
4.  $S \in V$  is the start symbol of the grammar

Context-free grammars (CFG) provide a means of specifying the structure of a language. They consist of a set of production rules that define how a language can be generated. Each production rule has possible expansions. To generate a valid string in a language defined by a CFG, one starts with the initial production rule, and makes choices of how to replace that rule with one of its possible expansions. If a string can be generated by a grammar in this way it is considered accepted by that grammar.

One common notation for specifying context-free grammars is the Extended Backus-Naur Form (EBNF). EBNF is a language that is used to describe the syntax of programming languages and other structured languages. In EBNF, a grammar is defined as a series of productions, each of which specifies a rule for combining or transforming symbols in the language.

EBNF is particularly useful because it allows for the expression of complex grammatical structures in a compact and intuitive way. For example, EBNF can be used to

express repeated elements through regex notation, specify alternatives (choices), and define recursion in the grammar. [Pattis]

This project uses a variant of EBNF form to express input to the rust program.

## 2.4 Earley Parsing

Earley Parsing is a chart parsing algorithm that can recognize strings from any context free grammars. It works by keeping  $n + 1$  state sets for an input string of length  $n$ . The  $i_{th}$  state set  $S(i)$  represents the information we know about the parse of the string after reading  $i$  tokens. This is achieved by keeping track of all of the rules the  $i_{th}$  token could be in the middle of.

Given a context free grammar  $G$ , and an input string  $s$  an Earley item is in the form:  $[V \rightarrow \alpha.\beta (f)]$

where  $(V, \alpha\beta) \in R$  and the  $.$  represents how much of the rule has been seen by the item.

Intuitively, an Earley item represents a partially complete production rule in the grammar where the first token produced by that rule is at position  $f$  in the input stream.

If we are looking for the start symbol  $S$  we begin an Earley parse by adding all of the production rules for  $S$  to the  $0_{th}$  state set as Earley items with no completed tokens. These would be all of the items of the form:

$$S \rightarrow .\alpha (0)$$

Then we have three rules to determine if an item  $I$  is in the  $S_{th}$  state set

1. Predict:  $[G \rightarrow \alpha.\Gamma\beta (s)] \in S(s) \wedge (\Gamma, \gamma) \in R \implies [\Gamma \rightarrow .\gamma (s)] \in S(s)$
2. Scan:  $[G \rightarrow \alpha.\gamma\beta (s)] \in S(s-1) \wedge \gamma = s_i \implies [G \rightarrow \alpha\gamma.\beta (s)] \in S(s)$
3. Complete:  $[G \rightarrow \gamma.(n)] \in S(s) \wedge [\Gamma \rightarrow \alpha.G\beta(m)] \in S(n) \implies [\Gamma \rightarrow \alpha G.\beta (m)] \in S(s)$

The Earley parsing algorithm works by filling out  $S_0$  using rules 1 and 3 until fixed point, then using rule 2 to fill out  $S_1$ .  $S_1$  is then filled until fixed point using rules 1 and 3 again, at which point the process continues for  $S_n$  using  $S_{n-1}$  and rule 2 to progress. In the worst case, this algorithm completes in  $O(n^3)$  where the complexity depends on the ambiguity in the grammar you are parsing with [Earley, 1970].

## 2.5 Error Correcting Earley Parsing

Error correcting Earley parsing works almost identically to regular Earley parsing. However, before it can be run, a grammar must be transformed in such a way that allows terminals to be inserted, deleted, or replaced. These extra rules will be referred to as edit rules. The parsing process is then similar the algorithm described above, however an extra number is kept in Earley items that tracks the number of the edit rules were used to create a given item, when a collision in a state set occurs, the item with more edit rules is removed. [Aho and Peterson, 1972]

## 2.6 Incremental LR Parsing

A powerful technique for incremental parsing is outlined in [Wagner and Graham, 1998]. The authors describe a method that relies just on the information in the previous parse tree to obtain a new tree resulting from an edit. This method requires a change to LR parsing that allows shifting non-terminals as well as terminals.

Briefly, LR parsers work by reading the input stream left to right, maintaining a stack of parse information as they go. Using some amount of look-ahead, they decide to either push the next symbol onto the stack, or reduce symbols on the top of the stack. Reduction can be thought of as applying a rule in a grammar. If you had the grammar:

$$S \rightarrow AB$$

$$A \rightarrow aa$$

$$B \rightarrow b$$

and you had “a a” on the top of your stack, and the next symbol in the input stream was “b,” you would reduce those a’s according to the rule that they can be represented as a single “A.” This would result in a new stack only containing “A.”

The Incremental method works with three ideas.

1. Identifying un-modified subtrees.
2. Allowing non-terminals to be shifted
3. Right most and left most breakdown of existing rules.

This incremental algorithm turns un-modified subtrees into the input stream for an LR parser. It then performs right most and left most breakdowns in order to effectively reuse subtrees. In order to perform a right most breakdown of a stream of parse trees, you take the last parse tree in the stream and replace it with it’s children repeatedly, discarding epsilon productions.

## 2.7 Tools and Technologies Used

### Rust

Rust is a general purpose, memory safe, systems programming language. It was used in this project as something like a preprocessing script. Between providing nom, a feature rich parser combinator library, and pattern matching capabilities, this language was perfect for the job. These features were used to parse a grammar in EBNF form, and to run modification and optimization passes on the parsed grammar to transform it into typescript that could be used by the parsing algorithm.

## **React**

React is a popular JavaScript framework for building user interfaces that was developed and maintained by Facebook. React allows developers to create dynamic, reusable UI components that can be easily integrated into complex web applications.

React provides the ability to efficiently update and render components, resulting in a responsive yet performant user interface. This is achieved through the use of a virtual DOM, which allows React to selectively update only the parts of the UI that have changed, rather than re-rendering the entire page. React is also highly customizable, with a large ecosystem of third-party libraries and tools available for developers to extend and enhance its capabilities. This makes it easy to build large scale applications. All of these benefits make React a suitable framework to target for this code generation script. [Venkat Sai Indla and Puranik, 2021]

## **Solid JS**

Solid JS is another javascript framework which allows ui designers to write their code in a declarative manner. This framework achieves this by making use of a publisher subscriber system allowing programmers to write UI's that responsively update when signals change their value.

# **3 Methodology**

## **3.1 Timeline of the project**

The original problem this MQP was created to address was to simplify LLVM IR for beginners. This is a broad topic and can have many approaches. Due to this, this project



took many turns.

## **Making Templates In The Form Of Documentation**

. The first approach taken to solve this problem was to create static web pages that explained LLVM IR Builder patterns and how to use them. This was done through github's web-page hosting service and markdown files. After interviews, it was clear that students were still confused. As a result, we wanted to shift the approach to create a tool that was more interactive.

## **Making a Projection Based Editor and Compiler**

The next approach was to create a web page that allowed for complete editing of a toy language. This toy language would then be compiled to C++ code that used LLVM IR Builder to generate equivalent LLVM IR code to the input toy language code. The problems faced in this approach were.

1. Lack of react experience
2. Lack of a need for an entire compiler
3. User input code was tied to AST code

We were inexperienced with react, so the code ended up being messy due to a lack of understanding of the underlying framework. This led to a code base that was hard to continue adding features to.

Students learning LLVM IR Builder don't need to create entire programs to understand how to use it. This demonstrates a lack of a need for a compiler from a toy language to well formed LLVM IR Builder code. In order to learn a new technology, while it might

be nice to be able to play around with a tool that is completely extensive, all that is usually actually necessary is access to commonly used patterns.

To tackle these problems we started redesigning this code base. We realized that all of the react components we were creating for user input could be generalized into a grammar. Similar to how a parser generator like ANTLR would create a parser based on an input grammar, we created a tool to generate graphical user input based on an input grammar. This tool ended up being useful for the next iteration of the MQP.

## **Making Templates That Rely on User input**

The next iteration of this MQP was to go back to something similar to static documentation of LLVM IR Builder through the use of commonly used patterns. To aid in the communication of the role of variables in patterns, each pattern is has graphical user input that can be used to determine pattern output. The goal of these patterns was to express important concepts in LLVM IR along with their analogs in the LLVM C++ API. The full grammar used to generate this GUI is in appendix B.

## **Functions**

The function pattern would take in a name, a variable list of arguments, a return type, and linkage options from the user through GUI inputs, and output valid C++ code that would emit a function with the user specified properties through the LLVM C++ API.

## **Control Flow**

The Control Flow pattern only had a selection between 4 options: If, If Else, For, and While. The pattern would then output the template for the selected control flow pattern.

## Basic Block

The Basic Block pattern didn't take in any input from the user and would always output C++ code that would emit a standard basic block. This pattern relied there being a function already created, so at the top of the pattern, there was a variable of type Function pointer linking to the Function pattern.

## Boolean Expression

This pattern took in a type and an operator from the user and its output was code that would operate on two Value pointers with the given type with the given operator.

## 3.2 Architecture overview

As shown in figure 2, our tool begins with an input grammar in EBNF form. It is then pre-processed with a rust program. The result of that processing is then used in the final editor in many locations.

## 3.3 GUI generation

A large part of this project was generating a GUI that reflects an input context free grammar. For example, if there was a grammar rule that said that a terminal  $A$  could be 'B' or 'C' written as:

```
A ::= 'B' | 'C';
```

Then the tool would create a front-end component for  $A$  that has a drop down selection box that let's  $A$  turn into either 'B' or 'C.' This works recursively so nested grammar production rules work how you would expect. For example, if the input grammar was this:

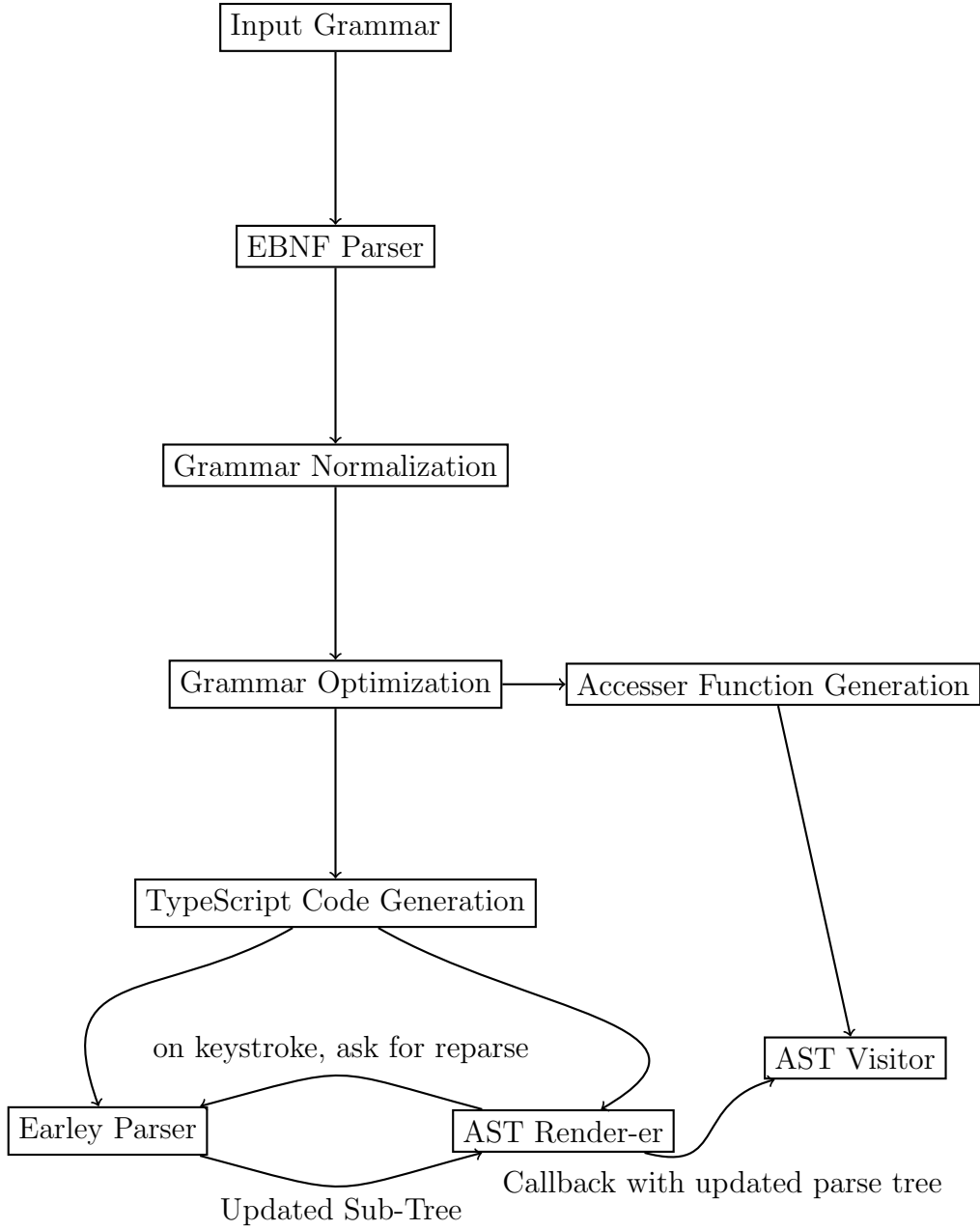


Figure 2: Architecture Diagram

```
A ::= B | 'C';  
B ::= 'one' | 'two';
```

Then the output front-end would prompt the user for the choice between  $\langle B \rangle$  and 'C', if  $\langle B \rangle$  is selected, the input box turns into another input box with a choice between 'one' and 'two.'

This also works with regex extension symbols:

```
A ::= 'c'+;
```

Would indicate that the user can add and remove from a list of 'c' with a minimum of 1 element. The star operator (\*) works the same way as plus, except the minimum is 0. Both of these create graphical plus and minus buttons for adding and removing elements.

The grammar can also contain regex matches. These would correspond to a text input box that verifies user input matches the desired regex. For example, to allow users to input a constant number:

```
A ::= #'[0-9]+';
```

The front-end components that these grammars generate also keep track of their state in an abstract syntax tree. To access node elements more easily, you can label elements in your grammar.

```
A ::= ('a' | 'b') <letter> ('1' | '2') <number>;
```

This grammar would allow you to access the first selection of a node with a call to letter() on that node. These labels also work within variable length lists:

```
A ::= (B <nth>)+;  
B ::= ((#' [a-Z]+' <elem>)*)+;
```

In this case, calling `node.nth(0)` would return the 0th input element of a node of type  $\langle A \rangle$ . Calling `node.elem(0,i)` would return the the  $i^{th}$  element of the  $o^{th}$  list of a node of type  $\langle B \rangle$ . Here the outer list is denoted by the '+' regex symbol and the inner list is denoted by the '\*' symbol. You can nest these calls, for instance, if you had a node of type  $\langle A \rangle$  then you could call `node.nth(n).elem(0,i)`.

These names are scoped at a per-alternation level. You could therefore have a grammar with the following assessor names:

```
A ::= 'a' <elem> | 'b' <elem>
```

calling `node.elem()` would return two different strings dependent on whichever alternation is actively being used in abstract syntax tree.

## Abstract Syntax Tree Generation

These generated react components update an abstract syntax tree (AST) automatically when user input makes a valid change. You can pass a callback function to the top generator of your grammar to be called every time there is an update to the AST.

The rust code generates one `AST_node` class for every production rule in the input grammar. All of these classes inherit from one base `AST_node` class. Each `AST_node` contains data which is some nested structure of strings and other `AST_nodes`. Labels make interfacing with this data easier by exposing access methods. Every `AST_node` also has an `accept` method used for a visitor.

There is also a generated base visitor class that has an abstract visit method for each node type. This ensures that if you're writing your own visitor and you update the input grammar, all methods for visiting must be implemented. The base visitor class is templated by return type. After calling `accept` on a node with a visitor of type `T`, you can expect a result of type `T`.

For the LLVM learning tool, we created a visitor class called `outputgen`. This visitor would generate JSX to be rendered onto the page based on the AST. We also called `root.accept(outputgen)` in the update callback passed into the root grammar rule component.

## User Input

Whenever a grammar gives the user choice, there needs to be a rendered component that accepts that input. These inputs come in three forms.

1. Alternations
2. Regex
3. Regex Extension

Alternations appear whenever the pipe symbol (`|`) is used in a grammar. For these cases, a select component is rendered. The select component is given the sub components (*children*) and their respective string values (*names*) as input. It renders a text input box that fuzzy finds through *names* based on user input to display auto complete options underneath.

Regex simply renders a text input box that checks user input based on a regex match. If the regex matches, the component sets the corresponding data in the AST.

Regex Extensions render a plus and minus button at the end of the list of components. They are handled with the variadic component. The variadic component takes in a function as input; this function takes in a positive integer `n` and outputs a JSX element. The variadic component renders `n` of these functions from 0 to `n-1` where `n` is determined by user input through the use of the plus and minus buttons.

## Type Checking

There is a special type of visitor class that is considered a type checker. Any visitor class templated with type boolean can be considered a type checker. The intended purpose of such a class would be to type check an AST, returning true if correct and false on a type error. There is a generated defaultChecker class that always returns true.

You can specify a type checker to be used by select components to narrow down selection options. A select component will try to add every one of its children to its options. Running the type checker on the whole abstract syntax tree for each option. On false it will remove that as an option.

## Backwards Element

Consider the following grammar:

```
A ::= line*;  
line ::= id ':=' ([line] | num) ';' ;  
id ::= #'[a-Z][a-Z0-9]*';  
num ::= #'[0-9]+';
```

The [line] put in hard brackets makes this "grammar" no longer a context free grammar. The react code this translates to is a select element where the options are any previous line in  $\langle A \rangle$

For example, if you had a program under this grammar that looked like this:

```
var1 := 5;  
var2 := 10;  
var3 := _
```



Where `_` represents possible user input that has not been input yet. Then the underscore would be rendered as a selection between `[line]` and `num`. And if `[line]` was selected, then a select component would be rendered with two options, `{'var1 := 5;', 'var2 :=10;'}`.

The fact that the options contain the content of the whole line can be clunky, so the angle bracket operator will look for a special label in the node called `name()`. If this label is provided, the option for that line would be represented as only the contents of that label. We can modify the grammar to fit this standard:

```
A ::= line*;
line ::= id <name> ' := ' ([line] | num) ' ; ' ;
id ::= #' [a-Z] [a-Z0-9]* ' ;
num ::= #' [0-9]+ ' ;
```

In this case, using the example above, the rendered options given to the user would look like: `{'var1', 'var2'}`. Because the `<name>` tag was given to the id of line.

## Styling

You can define style classes that elements of your grammar should belong to:

```
A ::= ('b' | 'c' <.style>)
```

In this case, the terminal `'c'` would belong to a css group called `"style."` If you wanted to give an expression a style and a name to access it by you can employ the following syntax:

```
A ::= ('b' | 'c' <name.style>)
```

## 3.4 Parsing input

### Motivation

Though tree navigation and editing can, on their own, provide a fairly robust *structured* editing experience, we sought to improve on that with ease-of-use text-editing capabilities. To this end, we added a parser to the editor, allowing textual changes to be reflected in real time in the parse tree.

#### 3.4.1 Imagination and Concreteness

Useful information can still be gleaned from incomplete input. For instance, a lone + symbol with an arithmetic grammar provides enough information to know the rough shape of the expression tree, with a vague idea that some sort of expression ought to lie on either side of it. To extract and formalize this idea, we introduce *imagined symbols*. Intuitively, these are symbols present in the parse tree that do not produce tokens present in the text. Imagined symbols can be terminal or nonterminal symbols. Symbols that are not imagined are said to be *concrete*. More formally, we define the concrete symbols first and let imagined symbols be those that are not concrete.

A terminal symbol (or terminal node on a parse tree) is *concrete* if and only if it corresponds to a token in the text (that is, if and only if it has a nonzero number of characters associated with it). A nonterminal node is *concrete* if and only if it has at least one concrete child. Imagined nodes are precisely those which are not concrete. We use the term “*imagined symbol*” to refer to a symbol that is used to label an imagined node in the tree, and similarly for concrete symbols.

We modify the Earley parsing algorithm to allow the imagination of any symbols necessary to complete parsing of an input stream. We call a parse tree that may have imagined nodes a *partial parse*.

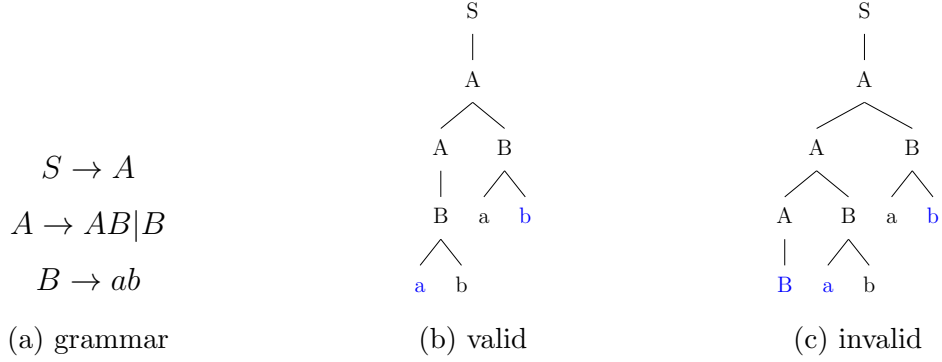


Figure 3: A grammar and two partial parses for “ba” on that grammar. Imagined symbols are depicted in blue. figure 7c depicts a minimal partial parse while figure 7d does not because the second tree does not minimize imagined symbols. This demonstrates that Non terminals can be imagined.

We want the final partial parse tree we select to minimize the number of imagined symbols so that it, intuitively, matches the input stream as closely as possible. Figures 7c and 7d show two partial parse trees for the input string “ba” on the grammar in Figure 3a.

### Concrete-aware Earley Parsing

We modify Earley parsing to support imagined symbols through addition of a fourth rule for item propagation by imagining the next symbol instead of scanning it. We also modify the structure of an Earley item. Items now take the form

$$[V \rightarrow \alpha.\beta (f, c, i)]$$

with, as before,  $(V, \alpha\beta) \in R$  and  $f \in \mathbb{N}$ .  $c \in \{\mathbf{F}, \mathbf{T}\}$  is a boolean flag signifying whether  $V$  is concrete or not.  $i \in \mathbb{N}$  signifies the number of imagined symbols required for this item so far.

1. Predict:  $[G \rightarrow \alpha.\Gamma\beta (s, c, j)] \in S(s) \wedge (\Gamma, \gamma) \in R \implies [\Gamma \rightarrow \cdot\gamma (s, \mathbf{F}, 0) \in S(s)]$
2. Scan:  $[G \rightarrow \alpha.\gamma\beta (s, c, j)] \in S(s-1) \wedge \gamma = s_i \implies [G \rightarrow \alpha\gamma.\beta (s, \mathbf{T}, j) \in S(s)]$
3. Complete:  $[G \rightarrow \gamma. (n, \mathbf{T}, j)] \in S(s) \wedge [\Gamma \rightarrow \alpha.G\beta(k, c, p)] \in S(n) \implies [\Gamma \rightarrow \alpha G.\beta (k, \mathbf{T}, p+j) \in S(s)]$

4. Imagine:  $[G \rightarrow \alpha.\gamma\beta (s, c, m)] \in S(s-1) \wedge \gamma \in \Sigma \cup V \implies [G \rightarrow \alpha\gamma.\beta (s, c, m+1)] \in S(s)$

## Grammar Transformations

There were two steps we took to transform input grammars. Normalization and optimization. The normalization step was necessary for use of the Earley parsing algorithm, and optimization was necessary for making up for the redundancies introduced by this normalization process.<sup>1</sup>

The normalization process is repeatedly applied to each rule in the input grammar until fixed point. The rules for moralization are as follows:

Pattern	Produces	Explanation
$R \rightarrow \alpha x \beta$	$R \rightarrow \alpha K \beta$ $K \rightarrow x$	Where $\alpha$ and $\beta$ are 0 or more symbols from $V$ or $\Sigma$ and $x$ is not
$R \rightarrow x^*$	$R \rightarrow K$ $K \rightarrow Kx$ $K \rightarrow \epsilon$	Lists of 0 or more
$R \rightarrow x^+$	$R \rightarrow K$ $K \rightarrow Kx$ $K \rightarrow x$	Lists of 1 or more
$R \rightarrow x^?$	$R \rightarrow x$ $K \rightarrow \epsilon$	Optional regex
$R \rightarrow (x)$	$R \rightarrow x$	Remove parenthesis
$R \rightarrow x y$	$R \rightarrow x$ $R \rightarrow y$	Remove alternations

<sup>1</sup>Preserving accessor name and css styling when normalizing and optimizing is an important detail to get right and the details of which are not outlined here.

	$A \rightarrow A'$
	$A' \rightarrow A'A''$
	$A' \rightarrow$
	$A'' \rightarrow B$
	$A'' \rightarrow C$
$A \rightarrow (B C)^*$	
(a) before	(b) after

Figure 4: Grammar before and after normalization

Optimization will only apply to the rules generated through normalization. It also is applied until fixed point. Essentially the algorithm removes two kinds of rules, redundant rules and identical rules.

### Redundant Rules

If you had the following grammar:

$$A \rightarrow a(bc)^*$$

Then at some point in the normalization process you would have:

$$A \rightarrow aA'$$

$$A' \rightarrow A''$$

$$A'' \rightarrow A''A'''$$

$$A'' \rightarrow \epsilon$$

$$A''' \rightarrow bc$$

In this case,  $A' \rightarrow A''$  is a redundant rule.

### Identical Rules

If you had the following grammar:

$$A \rightarrow (B|C)^*$$

At some point in the normalization process you would generate:

$$A' \rightarrow A'(B|C)$$

$$A' \rightarrow (B|C)$$

The purpose of identical rule removal would be to mitigate the fact that the two  $(B|C)$  expressions would generate many identical rules after further normalization.

### 3.4.2 Recovering a Parse After Earley Parsing

Initially we thought that recovering a parse tree from the modified Earley parsing method would be difficult. For this reason used parse forests in our code and then tried a few techniques to recover information from those objects.

#### Parse Forests

A *parse forest* is an object with a symbol and an array of collections of possible children. A parse forest represents the possible expansions of a symbol detected by the concrete-aware Earley process. We build a collection of parse forests during the Earley parsing process as follows:

Each item now has a reference to an associated parse forest. The initial items with the grammar start symbol S on the lefthand side each get a new parse forest with symbol S to start.

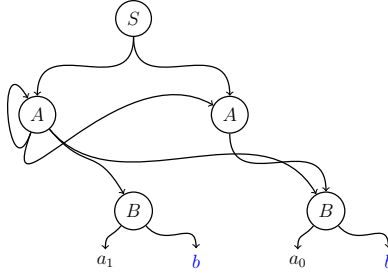


Figure 5: A Parse Forest for the input “aa” for the grammar in figure 3. Many simplifications were made to this representation. Edges leave out the number of times they can be traversed and imagined non terminal symbols have been left out.

Each rule for item propagation in concrete-aware Earley parsing has an associated action to continue the parse forests:

1. Predict: When an item  $I$  predicts a nonterminal  $\Gamma$  after the dot and produces a new item  $J$ , the new item  $J$  is associated with a reference to a new parse forest with symbol  $J$  and an array of empty collections of possible children with the length of the array equal to the number of symbols on the righthand side of the rule in item  $J$ .
2. Scan: When an item  $I$  scans a symbol  $\gamma$  after the dot and produces a new item  $J$ ,  $J$  is associated with a reference to the same parse forest referenced by  $I$ . If the dot in  $I$  is in position  $d$ , the symbol  $\gamma$  is added to the  $d$ th collection of possible children in the parse forest.
3. Complete: Consider the item  $J$  that is completed (has the dot at the end) and the item  $I$  into which it is completed (the dot is not at the end). Let  $K$  be the new item generated/implied by the completion.  $K$  is associated with a reference to the same parse forest referenced by  $I$ . If the dot in  $I$  is in position  $d$ , the parse forest associated with  $J$  is added to the  $d$ th collection of possible children in the parse forest referenced by  $I$  and  $K$ .
4. Imagine: When an item  $I$  imagines a symbol  $\gamma$  after the dot and produces a new item  $J$ ,  $J$  is associated with a reference to the same parse forest referenced by  $I$ . If the dot

in  $I$  is in position  $d$ , an imagined  $\gamma$  symbol is added to the  $d$ th collection of possible children in the parse forest.

## The Flatten Method

The flatten method operates on a parse forest. It is essentially a depth first search with pruning in the space of possible parses.

```
def flatten(node, edge_context):
    if node is leaf:
        return [node]
    pq <- empty priority queue
    enqueue [initial_tree, 0] into pq
    ret <- []
    min_imagined <- empty map
    while pq is not empty:
        [current_tree, index] <- pq.dequeue()
        if min_imagined.has(current_tree.end)
            and current_tree.num_imagined >= min_imagined(current_tree.end):
            continue
        if index == node.children.length:
            ret.push(current_tree)
            min_imagined(current_tree.end) <- min(current_tree.num_imagined, min_imagined)
        else:
            for each [edge, child_forest] in node.children[index]:
                if edge_context(this, child_forest) != edge:
                    update edge_context
                    child_trees <- flatten(child_forest, edge_context)
                    restore edge_context
                    for each child_tree in child_trees:
                        if child_tree.start == current_tree.end or child_tree.start == -1:
                            next_tree <- current_tree + child_tree
                            enqueue [next_tree, index + 1] into pq
    return ret
```



## Applying Dijkstra's Algorithm

Dijkstra's shortest path algorithm can also be applied to the problem of reconstructing the parse tree. To this end, we introduce the concept of NTrees. A node of a NTree has a symbol and an array of children that can be parse trees, parse forests, or other NTree nodes. NTrees represent partially (or fully) expanded parse trees starting from the grammar start symbol S at the root. Leaves of a NTree are parse trees if they are fully resolved and parse forests if they have not yet been expanded by choosing exact children. Essentially, leaves that are parse forests could be resolved to a NTree node with any combination of children from the parse forest.

Dijkstra's search operates on these NTrees. It begins with an unresolved S for each original parse forest with symbol S. Edges in the graph traversed by the search connect one NTree to all NTrees obtained by resolving one parse forest leaf to a NTree node with any combination of children allowed by the parse forest. Nodes are visited in the order determined by a heuristic function. The heuristic is the sum of the minimum imagined symbols of all the leaves that are parse forests. The search continues until a fully-resolved tree is obtained that describes the input stream.

## A Straight Forward Approach

After trying both of these methods to recovering parse information from the Earley recognizer, we found that we could apply the same method that [Aho and Peterson, 1972] employs to recover parse information. The method is simply to track pointers to parent Earley Items as we are parsing. This simply involves setting a single parent in the case of the imagine, scan and predict steps, and setting two parents in the complete step. The structure of a minimally imagined parse can then be recovered by simply traversing these pointers back to the start.

Although this new method replaces the idea of parse forests, they still have a use. As we conduct the Earley parse of our input, we discard ties and parse forests can be used to resolve those ties.

## Navigation

Cursor position is a position in the parse tree – either left or right of a nonterminal node, or at any position in the token of a terminal node – instead of a one-dimensional index into the character stream as it would be in a text editor.

As an important note, the right side of a nonterminal and the left side of its next sibling are *distinct* cursor positions.

The selected node is the node the cursor is on either edge or inside of.

For purposes of cursor navigation, the *depth* of a node  $n$  is the minimum path length from the root to  $n$  not counting edges originating from a parent with only one child.

Despite their internal left-recursive representation, lists are always interpreted for purposes of navigation and rendering as though they are one nonterminal node with a variable number of children, where each child is one element of the list.

Cursor movement works in the following way. If the Left Arrow Key is pressed, then

1. if the cursor is on the right of a nonterminal node, the cursor changes to the left of the nonterminal node;
2. if the cursor is inside a token and not on the left edge, the cursor moves one character left;
3. if the cursor is on the left of a nonterminal node or the left edge of a token:
  - (a) if the selected node has a sibling immediately to the left, the cursor moves to the

right edge of that sibling;

- (b) if the selected node is the root, the cursor does not move;
- (c) if the selected node is the leftmost child of its parent, a temporary variable is first set to the lowest ancestor of the selected node that has an immediate left sibling. If no such ancestor exists, the cursor does not move and the algorithm terminates. If it does exist, the temporary variable is set equal to that sibling. Then, the temporary variable is set equal to its own rightmost child repeatedly, as long as that child exists and the depth of the child does not exceed the depth of the selected node. Then, the cursor is placed at the right edge of the node indicated by the temporary variable.

If the Right Arrow key is pressed, the cursor moves according to the same rules as above but with all occurrences of “left” replaced with “right” and vice versa.

If the Up Arrow key is pressed, the cursor moves to the left of the parent of the selected node. If the selected node has no parent (meaning it is the root), the cursor does not move.

If the Down Arrow key is pressed, the cursor moves to the most recently occupied position of the cursor in or next to any of the selected node’s children. If it has no children, the cursor does not move. If the cursor has never been in or next to any of the selected node’s children, it moves to the left of the leftmost child.

### **3.5 Incremental Parsing**

We employed a method influenced by [Wagner and Graham, 1998] for incremental parsing in our structured editor. We recursively induce right and left most breakdowns of the input stream around the cite of an edit. For simple grammars, this step only needs to happen once. We also enforce that direct siblings of and edited nodes can’t be broken down

to allow for ambiguity resolution.

## 4 Results

### 4.1 Code Metrics

**Typescript Editor** is 2544 lines of code across 19 files.

**Rust Pre-Processor** is 3197 lines of code across 15 files.

Our pre-processor and parser have 80 unit tests.

### 4.2 Performance

#### Tree Reconstruction

Both the Flatten and Dijkstra's Search approaches for reconstructing partial parse trees produced editor behavior that was suitably real-time for edits on the first layers of the tree, but became intractably slow beyond the fifth layer of depth in the tree.

#### Incremental

Because the method we use for incremental parsing lowers the size of the input stream for LR(1) grammars to  $O(\log n)$  and the Earley parsing algorithm we use is  $O(n^3)$ . Our parser works in  $O(\log^3 n)$ . We have found this to be viable enough for real time applications.

```
(subtract(circle 30)(add(rotate(rect 60 5) 45)(translate(rect 5 50) 20 0)));
```



Figure 6: An example program and its output.

### 4.3 Toy Language

We created a toy language for the rendering of simple vector graphics. The grammar for this language is visible in Appendix A. Expressions in this language are either numbers, colors, or function calls. Function calls are in parentheses, with the name of the operation to be performed placed before the operands. An example expression is shown in Figure 6, along with its generated figure. This language was heavily inspired by the testing language used in a related work [Hempel et al., 2018].

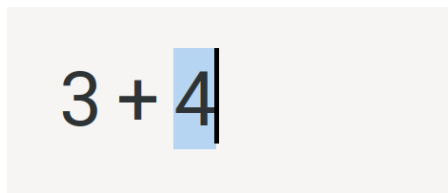
### 4.4 Ambiguity Resolution

We found that due to how we implemented our incremental parser, specifying operator precedence is intuitively tied to cursor location. Figure 7 shows how one could use this editor with an ambiguous grammar for arithmetic.

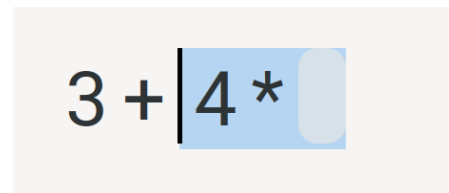
### 4.5 Future Work

#### User Studies

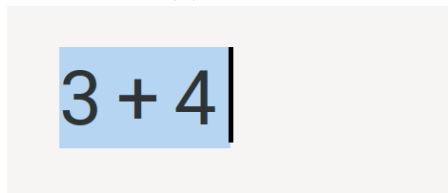
We have yet to conduct user studies to investigate the efficacy and appeal of the new editor. An example of one type of user study could compare users' experience with the

$$S \rightarrow expr$$
$$expr \rightarrow binop \mid number$$
$$binop \rightarrow expr \ op \ expr$$
$$op \rightarrow + \mid * \mid - \mid \div$$
$$number \rightarrow [0 - 9]^+$$


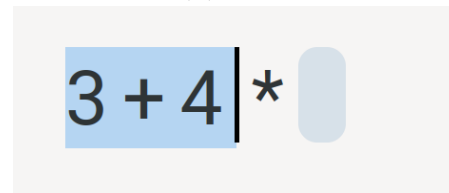
(a) before edit



(b) after edit



(c) before edit



(d) after edit

Figure 7: Resulting tree structure before and after edit at different cursor positions. Note the cursor was moved after the edit to show the structure of the resulting trees

new editor to that of a plain text editor, such as Notepad, while writing programs to achieve specific tasks in the vector graphics toy language. Users would be presented with a series of goal shapes and asked to reproduce them using the vector graphics language. Users in the control group would write their code in Notepad. Users in the experimental group would write their code in the new editor. All users would be asked to describe their experience both qualitatively and quantitatively through the use of ratings. They would be surveyed both early and late in the series of problems to compare their perspectives before and after they have had the opportunity to acclimate to the language and the editor.

### **Integration of Old Features**

There are features present in an old version of this project that were removed when adding the ability to parse user input. These features, such as rendering alternations as drop down menus, still need to be integrated with the new feature of parsing.

# Appendices

## A Grammar for Toy Language

```
S ::= (expr <nthExpr> ';' \n)* <exprsList>;
expr ::= opexpr <opact>
      | number <num>
      | color <col>;
opexpr ::= '(' op <action> expr* <args> ')';
number ::= #'-?[0-9]+' <text>;
op ::= keyword <kw> | id <name>;
keyword ::= ('rect'
            | 'circle'
            | 'rotate'
            | 'translate'
            | 'add'
            | 'subtract') <word.keyword>;
id ::= #'[a-z]+' <text>;
color ::= #'[A-Z]+' <text>;
```



## B Grammar File for LLVM Pattern Tool

```
program      ::= 'patterns' \n block;
block        ::= contextAndModule | functions | controlFlow | basicBlock | binaryOp;
contextAndModule ::=
    'no Input' <na> ;
functions ::=
    'linkage:' ('internal' | 'external') <linkage> \n
    'name:' id <name> \n
    'params:' \n \t ('-' type <nthType>)* <typesList> \n
    'return type:' type <retType>
    ;
basicBlock ::=
    'name:' id <name>;
controlFlow ::=
    'if' | 'if else' | 'while' | 'for loop';
binaryOp ::=
    'type:' ('unsigned' | 'signed' | 'float') <type> \n
    'operation:' op <op>;
type        ::= int | pointer | arrayty;
int          ::= '1 bit' | '2 bits' | '4 bits' | '8 bits' | '16 bits' | '32 bits';
pointer      ::= 'pointer' '(' type ')';
arrayty      ::= 'array' '(' type ')';
op           ::= '+' | '-' | '*' | '/' | '%' |
    '&' | '|' | '^' | '<<' | '>>' | '==' | '!=' |
    '<' | '<=' | '>' | '>=';
id           ::= #' [a-zA-Z] [a-zA-Z0-9_]*';
num          ::= #' [0-9]+';
```

## References

- Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *ieee*, August 2004. URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1281665&tag=1>.
- LLVM Language Reference Manual*. February 2023. URL <https://llvm.org/docs/LangRef.html>.
- Amir A. Khwaja and Joseph E. Urban. Syntax-directed editing environments: Issues and features. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 230–237, 1993. doi: 10.1145/162754.162882. URL <http://doi.acm.org/10.1145/162754.162882>.
- Markus Völter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. 01 2010.
- Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48, apr 1984. ISSN 0362-1340. doi: 10.1145/390011.808247. URL <https://doi.org/10.1145/390011.808247>.
- Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. *SIGPLAN Not.*, 52(1):86–99, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009900. URL <https://doi.org/10.1145/3093333.3009900>.
- Pattis. EBNF: A Notation to Describe Syntax. *University of California Irvine*. URL <https://www.ics.uci.edu/~pattis/ICS-33/lectures/ebnf.pdf>.
- Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, feb 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL <https://doi.org/10.1145/362007.362035>.
- Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972. doi: 10.1137/0201022. URL <https://doi.org/10.1137/0201022>.
- Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Trans. Program. Lang. Syst.*, 20(5):980–1013, sep 1998. ISSN 0164-0925. doi: 10.1145/293677.293678. URL <https://doi.org/10.1145/293677.293678>.
- Bhupati Venkat Sai Indla and Yogeshchandra Puranik. Review on React JS. *IJTSRD*, 5(4), June 2021. URL <https://www.ijtsrd.com/papers/ijtsrd42490.pdf>.
- Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: A lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 654–664, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180165. URL <https://doi.org/10.1145/3180155.3180165>.