Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MMP: An Object-Oriented Multi-Machine Parser Generator

Master's Thesis

Youyuan Lu

youylu@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Peter Belcák
Prof. Dr. Roger Wattenhofer

August 4, 2022

Acknowledgements

I thank my supervisor Peter Belcák for inspiring me to dive into the wonderful realm of parsing and guiding me through this journey with his expertise and insights. I also thank Prof. Dr. Roger Wattenhofer for giving me the opportunity to work on this thesis within the wonderful DISCO group. Finally, I owe my great debt to all masterminds behind the theory of formal languages.

Abstract

When compared with the manual labor of crafting parsers, most parser generators currently available are not flexible and powerful enough. They prohibit the free combination of various parsing algorithms/machines, restrict the form of grammar rules, and provide limited support for semantic actions and nonterminals' fields. In this thesis, we present a parser generator Multi-Machine Parsing (MMP) that solves these three problems. MMP features modular design, regular expression (regex) rules, and object-oriented nonterminals, thus providing users with much more freedom to compose their grammars and to tune the performance of the generated parsers.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Mainstream Parser Generators 1

1.2 Goals for MMP 3

2 MMP Frontend **4**

2.1 Grammar File Structure 4

2.1.1 Input Machine 4

2.1.2 Dependency Machine 6

2.1.3 Name Scope 6

2.2 Statement Structure 6

2.2.1 Statement Type 7

2.2.2 Rule Format 10

2.2.3 Statement Field 11

2.2.4 Semantic Action 13

2.3 Machine Structure 14

2.3.1 Machine Type 14

2.3.2 Machine Attribute 15

3 MMP Internals **16**

3.1 Repetition Elimination 16

3.2 Generalized Terminality 18

3.3 Subtype Relation 20

4 MMP Backend **23**
4.1 DFA Parser 23
4.1.1 Additional Restriction 23
4.1.2 NFA Construction 25
4.1.3 NFA-to-DFA Conversion 27
4.2 Lookahead Computation 28
4.2.1 Grammar Preprocessing 29
4.2.2 Sequential Lookahead 30
4.3 LL(k/finite) Parser 31
4.4 LR(k/finite) Parser 34

5 Future Work **37**

Bibliography **38**

A MMP Specification Language **A-1**

B A Working Example **B-1**

Chapter 1

Introduction

Parsing is the process of extracting a structure out of a linear input sequence according to a given grammar [1]. Such structure is used extensively in further processing of the input as it instructs the computer on how we expect the input to be interpreted. Due to the crucial role of this structural view in many text-related fields (e.g., compiling, decoding, etc.), parsing remains a central topic in almost all tasks in these fields.

A parser generator is a computer program that takes a grammar as input and produces the corresponding parser automatically. Given the aforementioned importance of parsing and the tremendous amount of labor one invests on writing a parser oneself, it is desirable to have a flexible and powerful parser generator, thus freeing people from manually crafting parsers and allowing them to focus on composing grammars.

1.1 Mainstream Parser Generators

A variety of parser generators have emerged since the last century, but many of them have drawbacks that hinder people from readily adopting them (for example, the dispute around Yacc as shown in [2] and [3]). We first overview three typically lacked but wanted features, then detail how some mainstream parser generators perform under these three metrics.

- Prohibited free combination of parsing algorithms/machines. When writing a parser for a programming language, it is preferable to apply different parsing techniques to different syntactic entities for optimized performance in terms of parser generation efficiency, parser size, and parsing speed. For example, the shunting yard algorithm is an appropriate choice for arithmetic and logical expressions, but a more general-purpose parsing algorithm (e.g., LL, LR, etc.) might be an overkill in this scenario. However, no current parser generators endorses such differentiated parsing, as they support at most a family of closely-related parsing methods (e.g., LR variants).

- Restricted form of grammar rules. Most current parser generators strictly follow the mathematical definition of grammar, and confine a rule to be a finite sequence of terminals and nonterminals. Such adherence quickly becomes a barrier against writing succinct grammars. To express a list, it is inevitable to resort to an unintuitive recursive form $\langle list \rangle \to \langle item \rangle \langle list \rangle \mid \varepsilon$, which transforms a linear sequence to a right-leaning binary tree. However, with the help of regexes, $\langle list \rangle \to \langle item \rangle^*$ suffices, and it explicitly displays a list as a sequence of items.

- Limited semantic actions and nonterminals' fields. Collecting structural information during parse time is frequently useful for further processing of the input. For example, recording all operators and operands when parsing an expression is a necessity for later analysis and evaluation. Such gathering is achieved by executing semantic actions embedded into the grammar, and a natural implementation is to put all operators into an operator list and to put all operands into an operand list, both of which are fields associated to the expression. However, many current parser generators are not expressive enough to support such sophisticated actions since they impose various restrictions on the number and types of a nonterminal's fields.

Table 1.1 summarizes the performance of five mainstream parser generators: Lex [4], Flex [5], Yacc [6], Bison [7], and ANTLR [8].

features \ generators	Lex/Flex	Yacc/Bison	ANTLR
Machine combination	✗	✗	✗
Regex rules	★	✗	✓
User-defined fields	✗	★	★

Table 1.1: Performance of some mainstream parser generators

✓ means full support; ★ means limited support; ✗ means no support.

Lex/Flex, as its name suggests, is designed to be a tokenizer generator, and most of its limitations are the consequence of such philosophy. Its output is limited to a deterministic finite automaton (DFA), which is an adequate formalism to describe and recognize the contents of tokens. It supports specifying rules using regexes on the character level, but no recursion is allowed as that is beyond the capability of a DFA. Although each rule can be attached with a semantic action, nonterminals cannot be configured by users to have fields.

Yacc/Bison is a general-purpose parser generator. It produces parsers that are among several LR variants, and no free combination of machines is possible. It requires each grammar rule to be a finite sequence of terminals and nonterminals. Limited facilities are provided for semantic actions and nonterminals' fields. Each nonterminal is allowed to possess only one field, and the semantic actions in the

middle of a rule have less capability (i.e., they cannot manipulate the field of the nonterminal being expanded) than their counterparts at the end of the rule.

ANTLR is a modern parser generator. It produces an integrated tokenizer-parser workflow, where the parser adopts LL(*) algorithm [9, 10], a self-adaptive variant of traditional LL(k). ANTLR permits regexes in specifying grammar rules, and tries to eliminate left-recursive rules using Kleene stars. Each nonterminal in an ANTLR grammar file has a predefined, read-only set of fields (e.g., its content, its position in the file, etc.), but users are not allowed to declare any other field for nonterminals of interest.

1.2 Goals for MMP

Our vision for MMP is to tackle the three problems mentioned at the beginning of section 1.1. Three mechanisms are employed to achieve this goal.

- Modular design. MMP allows users to freely partition a huge grammar into small pieces and to assign a parsing technique to each of them. The produced parser consists of several files, each of which is responsible for parsing a grammar fragment using the designated algorithm as mentioned above. This modular organization facilitates the logical decomposition of complex grammars, thus boosting the performance in the entire parser lifecycle, from parser generation to actual parsing.
- Regex rules. MMP supports specifying grammar rules in the form of regexes, which, although do not increase the expressiveness of grammars, provide great ease for writing concise and intuitive grammars. Such allowance comes with the trade-off of burdening the parser generator. Instead of linearly scanning the rule, the generator needs to traverse the tree-like structure of the rule regex.
- Object-oriented nonterminals. Similar to the objects in object-oriented programming languages, nonterminals in MMP have their own types and can be configured by users to possess fields derived from these types. All fields of a nonterminal can be manipulated by compatible semantic actions appearing anywhere in this nonterminal's corresponding rule. Such flexibility augments the recording of structural information during parse time, thus simplifying the further processing of the input.

Chapter 2

MMP Frontend

This chapter is a thorough introduction to the frontend of MMP, namely, how to write a grammar file in MMP format. We first give a high level overview of the structure of MMP grammar files, then explain each component in more detail.

2.1 Grammar File Structure

As stated in section 1.2, MMP adopts modular design that facilitates the logical partition and stratification of a sophisticated grammar. Figure 2.1 provides a hypothetical MMP grammar file and illustrates its structure. Each machine (`m00`, `m01`, etc.) is responsible for parsing a fragment of the entire grammar, and its body (`{/*statements*/}`) contains the corresponding grammar rules and some other MMP-specific utilities.

2.1.1 Input Machine

Parsing is a text processing task innately requiring stratification. To acquire the final representation that carries all needed structural information about the input, a collection of low-level units is formed first to record necessary local information. These low-level units are then served as building blocks for the construction of mid-level entities, which provide a more global view of the input. Finally, these mid-level entities contribute themselves to the establishment of the high-level representation we want. A concrete application of such layered workflow is found in the filed of compiling. An input program (a character stream) is first transformed into a sequence of tokens, from which a syntax tree is constructed to capture the structure of the whole program.

To incorporate stratification into MMP, the keyword `on` is employed to indicate a machine's input machine. If machine u is `on` machine v (v is u's input machine), then the nonterminals outputted by v will become the input fed to u. Due to the single-source nature of the input (since parsing is about analyzing a single input sequence), each machine is allowed to indicate at most one input

machine, and if no such input channel is specified, the machine being defined will accept text input.

The MMP grammar file in figure 2.1 has three layers: machines `m0`i, machines `m1`i, and machines `m2`i. `m00` directly process the text input, and may reference in its body the nonterminals defined in `m01` and `m02`. The keyword `uses` will be explained in more detail in section 2.1.2; for now, it is sufficient to understand that `m01` and `m02` are helpers for `m00`, and that all three machines are in the same layer and directly consume the text input. Layer `m1`i is built upon layer `m0`i in that machines `m1`i take the nonterminals produced by `m00` as their input, and `m10` may reference in its body the nonterminals defined in its helpers `m11` and `m12`.

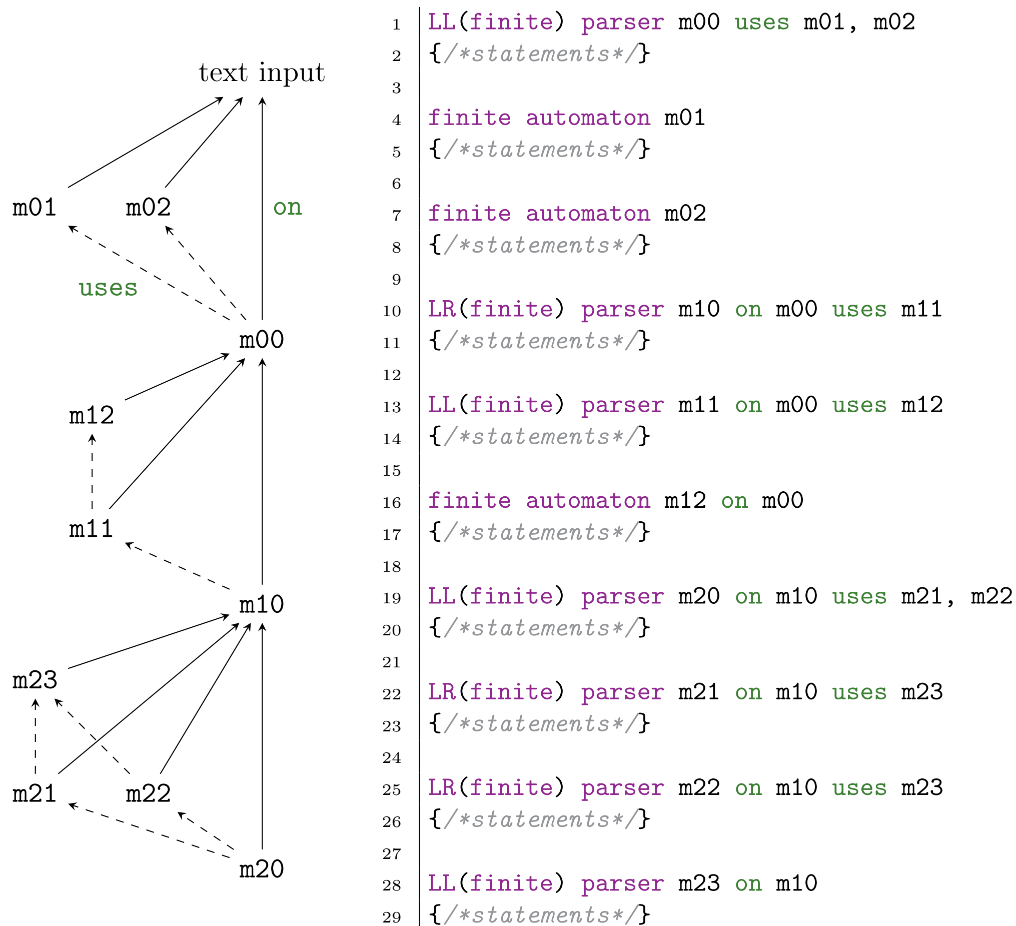


Figure 2.1: A hypothetical MMP grammar file and its illustrated structure

2.1.2 Dependency Machine

Section 1.1 explained in its beginning, through the example of programming languages, the motivation of partitioning huge grammars and applying different parsing techniques to each fragment. To recap, different parts of a grammar require different parsing power, so adaptively applying suitable parsing algorithms to these parts can achieve the optimized performance.

MMP's mechanism for partitioning grammars is the adoption of the keyword `uses`. If machine u `uses` machine v (v is u's dependency machine), then u can reference in its body the nonterminals defined in v. The only restriction is that u and v should accept the same input source. Imagine a huge grammar for some programming language, and let u be the LR parser that is in charge of producing the entire syntax tree, and v be the DFA parser that analyzes single statements. Then u and v are on the same logical level in the sense that the input to both of them is a token stream. Also, u needs to reference some nonterminals in v to grasp the structures of single statements.

The keyword `uses` supports both direct and transitive references. For example, machine `m20` in figure 2.1 can reference in its body the nonterminals defined in machines `m21` and `m22`, as written in line 19 of the code. `m20` can also reference the nonterminals defined in machine `m23`, which is used declaratively by its direct helpers `m21` and `m22`. The boundary of such transitivity is machine `m10`, which provides input to layer `m2`i.

2.1.3 Name Scope

Name scope is about what nonterminals (or in other words, their names) can be referenced from a certain machine. As stated in sections 2.1.1 and 2.1.2, a machine has access to all nonterminals defined within its own body, its input machine, and its direct and transitive dependency machines, thus forming a transitive `uses`-closure inclusively bounded by the previous layer. For example, machine `m10` in figure 2.1 can use the nonterminals defined within machines `m11`, `m12`, and `m00`.

Note that whether a nonterminal can be referenced depends solely on the logical structure of the grammar file. Therefore, a nonterminal can be used when its definition has not been seen yet, as long as it resides in the appropriate `uses`-closure talked above.

2.2 Statement Structure

Before studying the overall structure of machines in section 2.3, we explore in this section the body of a machine, namely, how a statement is formed. A statement

is more than a grammar rule in that it features user-configurable object-oriented characteristics, which, as stated in section 1.2, augments the recording of structural information during parse time.

Since each statement defines uniquely how a nonterminal should be expanded during parse time, we will use the two words "statement" and "nonterminal" interchangeably in the rest of this thesis. The materials in this section are necessary for understanding the machine structure that will be introduced in section 2.3.

2.2.1 Statement Type

There are four types of statements in MMP, as shown in table 2.1. Figure 2.2 provides an MMP grammar file about statement types. Figure 2.3 illustrates the corresponding statement relations. This section overviews statement properties, where the introduction to the four statement types is incorporated.

properties \ statements	`category`	`production`	`pattern`	`regex`
Type-forming	Yes	Yes	No	No
Contextual	Yes	Yes	Yes	No
Rule	No	Yes	Yes	Yes
Terminality	Synthesized	Declared	N/A	N/A

Table 2.1: Four types of MMP statements

- Type-forming. The idea of type-forming statements comes from the types in object-oriented languages. `category` and `production` are the only two type-forming statements, and they are similar to the interface and the class in object-oriented languages, respectively. Type-forming statements make their own types, and a generated parser only returns type-forming statements. On the syntactic level, type-forming statements can explicitly specify their rootness as `root` (like `float`, `decimal`, and `hex` in figure 2.2) or `ignored root` (like `whitespace` in figure 2.2), and can specify their own field lists (like `number` and `hex` in figure 2.2).

  When a machine serves as a dependency machine, the rootnesses of its statements are irrelevant, and it produces whatever type-forming statement requested by the machine that `uses` it.

  Rootness matters when a machine works as an input machine. The keyword `root` means that the statement being defined should be viewed as a starting point of parsing (i.e., the start symbol in the mathematical definition of grammar), and that once successfully parsed, this statement should

```
finite automaton number_scanner
with productions_nonterminal_by_default,
      productions_nonroot_by_default,
      ambiguity_resolved_by_precedence
{
    category number {raw digits; flag is_decimal;};
    category integer : number;

    ignored root production whitespace = [' ' '\t' '\n']+;

    root production float : number = digit+ '.' digit+;
    root production decimal : integer = digit+ @flag:is_decimal;
    root terminal production hex : integer {raw letters;} =
        (digit | letter @append:letters)+;

    pattern digit : float, decimal, hex = ['0'-'9'] @append:digits;
    regex letter = ['a'-'f' 'A'-'F'];

    pattern stranger : number = ('#'+ | '$'+) @empty:digits;
}
```

Figure 2.2: An MMP grammar file about statement types

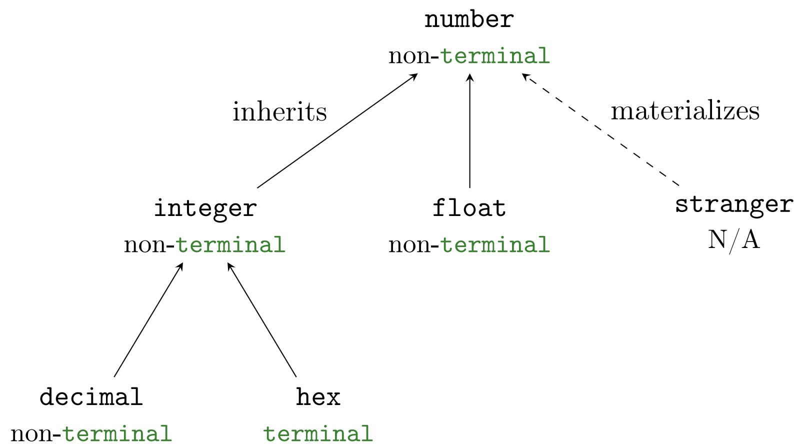


Figure 2.3: Relations among statements in figure 2.2

be placed into the output sequence, a sequence of root statements[1]. The keyword ignored root also designates the statement being defined as a starting point, except that this statement will be discarded once the generated parser finishes parsing it. The difference between these two keywords

[1]Actually, the return value is a tangible subtype of the successfully parsed root, and each element in the output sequence is a tangible subtype of some root. The concept of subtypes will be explored further in section 3.3.

is demonstrated in figure 2.2. The output of `number_scanner` is a sequence of `float`, `decimal`, and `hex`, and all `whitespace` are discarded.

- Contextual. Each type-forming statement creates its own context in that all semantic actions executed in its rule manipulate its own fields. On the syntactic level, contextual statements can claim their base contexts via base context lists. The purpose of adopting contextual statements is threefold.

  First, `category` and `production` are contextual statements, and their base context lists can only contain `category`. In this case, each base context (a `category`, to be exact) is similar to an inherited interface in object-oriented languages. For example, in figure 2.2, `production decimal` inherits a field `is_decimal` transitively via `category integer` from `category number`, and therefore can operate this field in its own rule.

  Second, although `category` is intended to be intangible and play the role of interface, materializing it is a handy feature to have, and `pattern` is designed as a contextual statement for this purpose. `pattern` itself is not type-forming; it only manipulates the fields on behalf of its base contexts, a sequence of `category`. For example, `pattern stranger` in figure 2.2 acts on field `digits` of its base context `number`. When a `pattern` is successfully parsed, what will be returned is its base context.

  Third, sometimes it is convenient to factor out common parts shared by several statement rules. For example, statements `float`, `decimal`, and `hex` in figure 2.2 use digits in their rules, so we factor out this shared piece and make it another statement `digit`. `digit` itself is not type-forming since this grammar is about recognizing numbers instead of single digits. However, `digit` requires some contextual information as it is used by `float`, `decimal`, and `hex` to record each digit appearing in a number for the inherited field `digits`. To serve this purpose, `pattern`'s base context list is allowed to be a sequence of `production`, as shown in line 16 of the code.

- Rule. Except for the interface imitant `category`, all three other types of statements have a grammar rule for expansion. The format of grammar rules in MMP is discussed in section 2.2.2.

- Terminality. The keyword `terminal` indicates that the `production` being defined has an internal field that automatically records the sequence of input symbols that constitutes this `production`. For example, when facing text input `1234BEEF`, `production hex` in figure 2.2 stores `1234` in its inherited field `digits`, and stores the entire stream `1234BEEF` in its automatically created internal field.

  `category` has an automatically synthesized terminality that accumulates its nominal subtypes' behavior. The concept of subtypes will be explored further in section 3.3; for now, it is sufficient to understand nominal subtypes as the solid arrows in figure 2.3. Unlike `production`, a `terminal category`

does not imply the existence of an internal field; instead, this terminality is only used by MMP internally to facilitate some sanity checks.

A `category` is a synthesized `terminal` when all its strict nominal subtypes (i.e., not including itself) are `terminal` (either declared by users or synthesized automatically). Figure 2.3 illustrate this bottom-up synthesis. Note that `stranger` is a `pattern` and is ignored during this process, since only type-forming statements are considered and accumulated.

We have not mentioned `regex` in our discussion above. `regex` is designed to be an actionless counterpart of `pattern` in terms of factoring out common parts shared by several statement rules. Because of its actionless nature, `regex` requires no contextual information and therefore is not a contextual statement.

2.2.2 Rule Format

As stated in section 1.2, MMP supports writing grammar rules using regexes for conciseness and intuitiveness. The complete structure of a grammar rule is listed in appendix A, and here we provide a brief overview of it.

- A grammar rule in MMP is a disjunctive regex, which is a collection of conjunctive regexes, each of which serves as an alternative during parsing.
- A conjunctive regex is a sequence of root regexes. The entire conjunctive regex is matched only when all its component root regexes are matched.
- A root regex is a unit to carry semantic actions. There are two types of root regexes: atomic regex and repetitive regex.
- A repetitive regex wraps an atomic regex with a repetition specification that indicates how many times the wrapped atomic regex should be repeated.
- An atomic regex can be either primitive or disjunctive (thus achieving nested structure). Primitive ones can be a single input symbol, a range of input symbols, a reference to some other nonterminal, etc.

Some extra care needs to be taken when referencing a statement from a grammar rule. Suppose statement s is referenced by a grammar rule in machine m.

- If s is defined within m, then no restriction is imposed on s.
- If s is defined in the input machine of m, then s must be a nominal subtype of a `root`. This restriction is reasonable since as stated before, a machine outputs a sequence of its `root`[2], and therefore m should only be allowed to

[2] Actually, each element in the output sequence is a tangible subtype of some `root`. The concept of subtypes will be explored further in section 3.3.

access its input machine's `root` (as well as their nominal subtypes), namely, the symbols in m's input stream.

- If s is defined in a dependency machine of m, then s can be any type-forming statement. Compared with the previous case, the rootness of s is ignored here, thus easing the combination of different parsing algorithms. The reason why non-type-forming statements are disallowed is that in some parsers (e.g., DFA), non-type-forming statements are integrated into the type-forming statements that use them, and therefore cannot be readily referenced from other machines.

2.2.3 Statement Field

An important feature of object-oriented design is to allow each object to possess its own fields. MMP adopts this philosophy and allows type-forming statements to declare their fields, as mentioned in section 2.2.1. Table 2.2 summarizes for each type of field its analogy in normal programming languages and its compatible semantic actions. Figure 2.4 gives an MMP grammar file demonstrating how to use fields and semantic actions to record structural information.

fields / description	`flag`	`raw`	`item`	`list`
Analogy	Boolean	Deque	Object	Stack

Actions	`@flag`
	`@unflag`
		`@capture`
		`@append`
		`@prepend`
		`@empty`
			`@set`
			`@unset`
				`@push`
				`@pop`
				`@clear`

Table 2.2: Fields and Semantic Actions

As shown in table 2.2, there are four types of fields. We introduce each of them, and briefly mention their corresponding semantic actions, which will be discussed in more detail in section 2.2.4.

- `flag` field is similar to a boolean variable in the sense that it has two possible states, FLAGGED and UNFLAGGED. Action `@flag` sets it to FLAGGED, and action `@unflag` resets it to UNFLAGGED, which is its initial state. For example, field `has_add` in figure 2.4 is initially UNFLAGGED, but when an `ADD_OP` in statement `expression`'s rule is matched in line 30 of the code, this field is set to FLAGGED by action `@flag`.

- `raw` field is used to store a sequence of input symbols of the current layer, and it is the type of the internal field automatically created by MMP for `terminal` statements mentioned in section 2.2.1. `raw` field resembles a

```
finite automaton tokenizer
with ambiguity_resolved_by_precedence
{
    // productions have root rootness and terminal terminality
    // by default in DFA
    root category operator;
    production ADD_OP : operator = '+';
    production SUB_OP : operator = '-';

    root category operand;
    production ID : operand = ['a'-'z']+;
    production CONST : operand = ['0'-'9']+;

    ignored root production whitespace = [' ' '\t' '\n']+;
}

LL(finite) parser expression_parser on tokenizer
{
    root production expression
    {
        flag has_add;
        raw content;
        operand item head;
        operator list operators;
    }
    =
    (
        operand @set:head
        (
            (ADD_OP @flag:has_add | SUB_OP) @push:operators
            operand
        )*
    ) @capture:content;
}
```

Figure 2.4: An MMP grammar file about fields and semantic actions

deque instead of a stack, because input symbols can be inserted either at its head (by action `@prepend`) or at its tail (by action `@append`).

For example, field `content` in figure 2.4 is initially empty, but when an `expression` is successfully parsed in line 33 of the code, it is commanded by action `@capture` to hold all input symbols that constitute the parsed `expression`. For example, after feeding `expression_parser` with the token stream `ID ADD_OP CONST SUB_OP ID`, an `expression` will be returned and its field `content` will be [`ID`, `ADD_OP`, `CONST`, `SUB_OP`, `ID`].

- `item` field requires the name of a type-forming statement to complete its

declaration, and is used to store a subtype[3] of this statement. For example, field `head` in figure 2.4 initially stores nothing, but when the first `operand` in the rule of `expression` is matched in line 28 of the code, this field is commanded by action `@set` to store this `operand` (either an `ID` or a `CONST`). An `item` field can also be cleared by action `@unset`, the opposite of `@set`.

- `list` field differs from `item` field in that it stores a sequence of subtypes[3] of the designated type-forming statement. It resembles a stack instead of a deque (as `raw` does), because element insertion (by action `@push`) and deletion (by action `@pop`) only happens at its tail. For example, field `operators` in figure 2.4 is initially empty, but each time a binary operator (either an `ADD_OP` or a `SUB_OP`) is matched in line 30 of the code, this operator is appended by action `@push` to `operators`.

2.2.4 Semantic Action

We have discussed in section 2.2.3 about how semantic actions interacts with their target fields. This section will focus on the input of semantic actions. The input to a semantic action is the root regex this action associated to.

- `@flag` and `@unflag` toggle the state of their two-valued target fields, and therefore require no input. They can be placed anywhere in a grammar rule, as shown by line 12 in figure 2.2 and line 30 in figure 2.4.
- `@empty`, `@unset`, `@pop`, and `@clear` delete predefined portions of their target fields, and therefore require no input. They can be placed anywhere in a grammar rule, as shown by line 19 in figure 2.2.
- `@capture`, `@append`, and `@prepend` require their inputs to encompass the input symbols of the current layer, the only elements accepted by `raw` fields, as stated in section 2.2.3. We will discuss further in section 3.2 about generalizing the concept of terminality to all regexes in grammar rules to determine whether theses three actions are compatible with their inputs. For now, check lines 14 and 16 of figure 2.2 and line 33 of figure 2.4 for some examples.
- `@set` and `@push` resemble the generic operations in object-oriented languages, and require their inputs to be nominal subtypes of their designated type-forming statements, as shown by line 28 in figure 2.4.

To explain intuitively how fields and semantic actions shape the parsed output, figures 2.5 and 2.6 give two output results visualized automatically by MMP.

[3]Actually, the stored object is a tangible subtype of the type-forming statement in the field declaration. The concept of subtypes will be explored further in section 3.3.

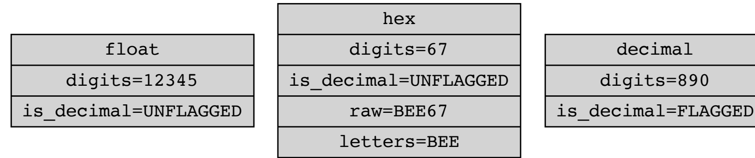


Figure 2.5: Output of figure 2.2 visualized by MMP

Acquired by feeding text input stream `12.345 BEE67 890` to the grammar in figure 2.2.

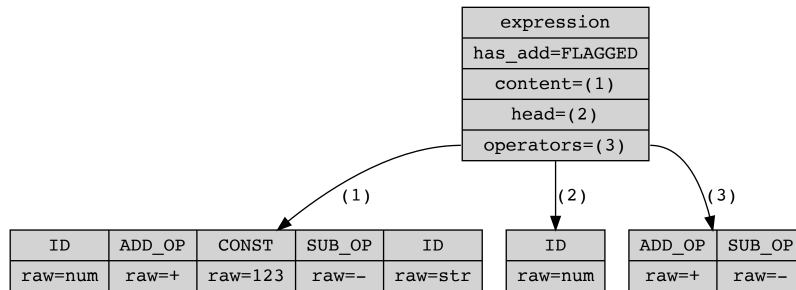


Figure 2.6: Output of figure 2.4 visualized by MMP

Acquired by feeding text input stream `num + 123 - str` to the grammar in figure 2.4.

2.3 Machine Structure

Each machine in an MMP grammar file is a collection of statements to be processed by the designated parsing algorithm with customizable settings. We first introduce the machine types (in other words, parsing algorithms) supported by MMP, then talk about how machine settings can be tuned by supplying different values for a machine's attributes.

2.3.1 Machine Type

MMP currently allows three machines:

(i) DFA with specifier `finite automaton`;

(ii) LL(k/finite) with specifier `LL(⟨k⟩)`, where the lookahead length $\langle k \rangle$ is either a nonnegative integer or keyword `finite`; and

(iii) LR(k/finite) with specifier `LR(⟨k⟩)`, where the lookahead length ⟨k⟩ is either a nonnegative integer or keyword `finite`.

To achieve the three goals mentioned in section 1.2 (i.e., modular design, regex rules, and object-oriented nonterminals), some extra efforts need to be invested into realizing these machines, and the resulting parsers are more complicated than their textbook counterparts. Chapters 3 and 4 dive into these technicalities in more depth.

2.3.2 Machine Attribute

The behavior of an MMP machine depends not only on its type (i.e., its designated parsing algorithm elaborated in section 2.3.1), but also on the properties of its statements as discussed in section 2.2.1. These statement-wise settings can be initialized conveniently by a machine's four boolean-valued attributes:

(i) `ProductionsTerminalByDefault`, which says whether every `production` should be initialized as a `terminal`;

(ii) `ProductionsRootByDefault`, which says whether every `production` should be initialized as a `root`;

(iii) `CategoriesRootByDefault`, which says whether every `category` should be initialized as a `root`; and

(iv) `AmbiguityResolvedByPrecedence`, which says whether ambiguities should be resolved by precedence. For example, in figure 2.2, both `decimal` and `hex` can parse the text input stream `1234`, but a `decimal` is deterministically returned because of line 4.

Note that these initializations will be overwritten by the explicit property elaboration of a statement. For example, line 2 in figure 2.2 (non-`terminal` by default) conflicts with the definition of `hex` (declared `termianl`), and the latter prevails as shown by the internal `raw` field of `hex` in figure 2.5.

Chapter 3

MMP Internals

MMP analyzes an input grammar file lexically, syntactically, and semantically to detect various problems that may arise from the grammar composer's careless planning and typing. Before performing the semantic check, MMP transforms the appearance of an input grammar file without modifying the language it describes to facilitate the parser generation. This chapter first discusses the grammar appearance transformation performed by MMP, then addresses two key points in the MMP semantic analysis.

3.1 Repetition Elimination

The purpose of transforming the appearance of a grammar file is to eliminate all repetitive regexes, thus making this normalized grammar file suitable to be fed into some parser generation algorithms. For example, the rules in LL and LR grammars are assumed to be finite sequences of terminals and nonterminals, and the existence of repetition notations complicates the processing of rules and the generation of corresponding parsers.

The basic idea of elimination is to recursively replace each repetitive regex r with two new statements r_{list} and r_{item}. r_{list} captures r's overall structure, and r_{item} records the pattern being repeated in r. Figure 3.1 demonstrates in detail how this idea works with different kinds of statements. Note that each pair (r_i, c_i) in figure 3.1 represents the location (row and column) of the repetitive regex being replaced in the original grammar file.

- When a repetitive regex appears in a `production`'s rule, as shown by statement `float` in figure 3.1, the two newly-created statements will have type `pattern` and declare the involved `production` as their base contexts. The reason why `pattern` is employed here instead of `regex` is that the repetitive regex being replaced might contain some semantic actions that manipulate the fields of the involved `production`, and `pattern` allows the two new-created statements to receive this contextual information and at the same time be kept as components of a type.

```
finite automaton tokenizer
{
    // productions have root rootness and terminal terminality
    // by default in DFA
    category number;

    // float, original
    production float : number {raw integer; raw fraction;} =
        digit+ @capture:integer '.' tail;
    // float, transformed
    production float : number {raw integer; raw fraction;} =
        float__list_r1_c1 @capture:integer '.' tail;
    pattern float__list_r1_c1 : float =
        float__item_r1_c1 | float__item_r1_c1 float__list_r1_c1;
    pattern float__item_r1_c1 : float =
        digit;

    // tail, original
    pattern tail : float =
        digit @append:fraction {1,2};
    // tail, transformed
    pattern tail : float =
        tail__list_r2_c2;
    pattern tail__list_r2_c2 : float =
        tail__item_r2_c2 | tail__item_r2_c2 tail__item_r2_c2;
    pattern tail__item_r2_c2 : float =
        digit @append:fraction;

    // digit, original
    regex digit =
        ['0'-'9'] '_'*;
    // digit, transformed
    regex digit =
        ['0'-'9'] digit__list_r3_c3;
    regex digit__list_r3_c3 =
        () | digit__item_r3_c3 digit__list_r3_c3;
    regex digit__item_r3_c3 =
        '_';
}
```

Figure 3.1: An MMP grammar file about repetition elimination

- When a repetitive regex appears in a pattern's rule, as shown by statement `tail` in figure 3.1, the two newly-created statements will, for the same reason as mentioned in the case of production, have type pattern. This time, however, their base contexts will be the same as those of the involved pattern, which itself manipulates fields on behalf of its base contexts.

- When a repetitive regex appears in a `regex`'s rule, as shown by statement `digit` in figure 3.1, no contextual information are required by the two newly-created statements, as the involved `regex` is itself actionless. Thus, `regex` serves sufficiently as the type of the two newly-created statement.

Note that although figure 3.1 does not show, the elimination process recurses in the case where the rule r' of r_{item} also contains some repetitive regex ans thus two new statements r'_{list} and r'_{item} are required.

3.2 Generalized Terminality

This section explores, as hinted in section 2.2.4, how to generalize the concept of terminality to all regexes in grammar rules to determine whether the three `raw`-related semantic actions (i.e., `@capture`, `@append`, and `@prepend`) are compatible with their inputs.

The basic idea of computing a regex's terminality is to accumulate the terminality of each of its components, thus assuring that this regex always encompasses the input symbols of the current layer. The following recursive definition delineates the bottom-up process of computing terminality for each regex in a grammar rule, and is illustrated by figure 3.3, which focuses on statement `VAR`'s rule in figure 3.2.

- A disjunctive regex is a `terminal` when each of its alternatives (a conjunctive regex) is a `terminal`.
- A conjunctive regex is a `terminal` when each of its component (a root regex) is a `terminal`.
- A primitive regex is a `terminal` when it is
  - (i) a non-reference, which encompasses text input symbols; or
  - (ii) a reference to a non-type-forming statement, which is assured by section 2.2.2 to be defined within the current machine and thus encompasses the input symbols of the current layer; or
  - (iii) a reference to a `terminal` statement defined within the current machine or its dependency machine, which is embedded with a `raw` field that stores a sequence of input symbols of the current layer; or
  - (iv) a reference to a nominal subtype of a `root` statement defined within the current machine's input machine, which is itself an input symbol of the current layer.

To explain intuitively how `raw`-related semantic actions set their target fields using the input symbols encompassed in their input regexes (`terminal` regexes,

```
finite automaton tokenizer
with productions_nonterminal_by_default
{
    // productions have root rootness by default in DFA
    category entity {raw name;};

    terminal production VAR : entity {raw type; raw index;} =
    (
        // MMP decomposes a string into a sequence of characters
        // at the very beginning, so adding parenthesis is necessary
        ("str_") @capture:type digits
        | "num_" num_type @capture:type digits
    ) @capture:name;

    production ARRAY : entity = VAR @capture:name "[#]";

    pattern digits : VAR = ['0'-'9'] @append:index +;
    regex num_type = "int_" | "float_";

    ignored root production whitespace = [' ' '\t' '\n']+;
}

LR(finite) parser entity_parser on tokenizer
{
    root production entities {raw vars; raw arrays_reversed;} =
    (VAR @append:vars | ARRAY @prepend:arrays_reversed)+;
}
```

Figure 3.2: An MMP grammar file about generalizing terminality

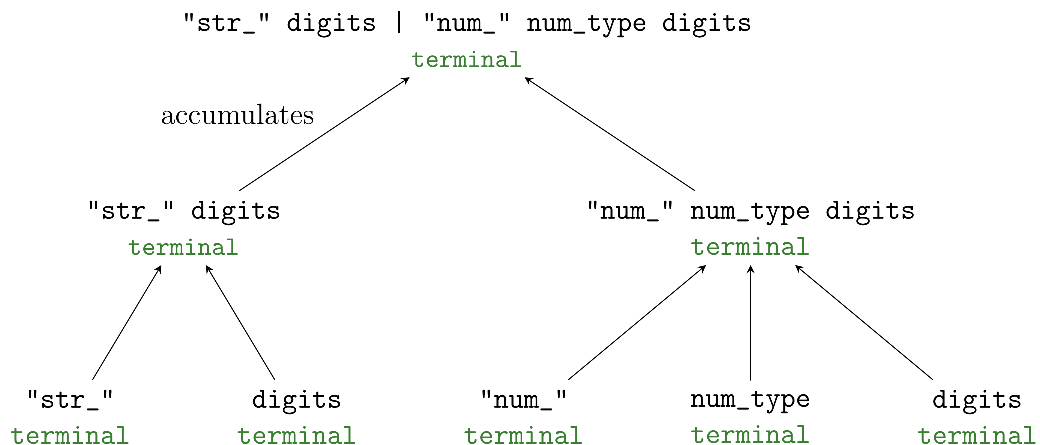


Figure 3.3: Computing terminality for a regex in figure 3.2

to be exact), figure 3.4 gives an output result visualized automatically by MMP. This visualization shows that the purpose of `raw`-related semantic actions is to extract the input symbols of the current layer encompassed in their input regexes and to put them into their target fields. For example, the `raw` fields of **`entities`** store **`VAR`** and **`ARRAY`**, whereas the `raw` fields of **`VAR`** and **`ARRAY`** store characters.

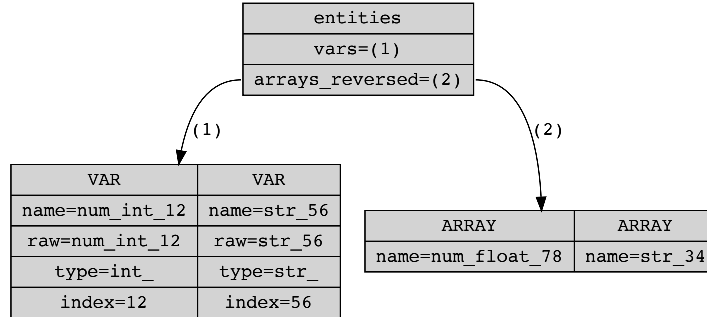


Figure 3.4: Output of figure 3.2 visualized by MMP

Acquired by feeding text input stream **`num_int_12 str_34[#] str_56 num_float_78[#]`** to the grammar in figure 3.2.

3.3 Subtype Relation

This section introduces formally the concepts of subtypes in MMP, which is only mentioned cursively in previous sections. There are two sorts of subtypes in MMP, nominal subtypes and tangible subtypes. Figure 3.6 illustrates the subtype relations among statements defined in figure 3.5.

Nominal subtypes are the same as those in object-oriented languages, since they are defined in a declarative way through base context lists. The only restriction is that no diamond is allowed to exist in the inheritance hierarchy. Consider statement **`unreal`** in figure 3.5. It is problematic because when trying to manipulate the field **`not_real`** in line 15 of the code, **`unreal`** does not know whether this field is inherited from `category` **`cat`** or from `category` **`dog`**.

Tangible subtypes are subtly different, and are delineated by the following recursive definition.

- The only tangible subtype of a `production` is itself.
- The tangible subtypes of a `category` consists of the tangible subtypes of all those type-forming statements that declare this `category` directly as a base

context. In addition, this category itself should be counted as a tangible subtypes if it has any pattern materializing it.

For example, in figure 3.6 (ignore the problematic `unreal`), `lion`, `leopard`, `wolf`, and `fox` are production, so their tangible subtypes are {`lion`}, {`leopard`}, {`wolf`}, and {`fox`}, respectively. `cat` is a category and there is no pattern materializing it, so its tangible subtypes are {`lion`, `leopard`}. `animal`, on the other hand, has a pattern `unknown` materializing it, so its tangible subtypes are {`lion`, `leopard`, `wolf`, `fox`, `animal`}.

```
1  finite automaton subtypes
2  {
3      category animal {flag not_real;};
4      pattern unknown : animal = "trill";
5
6      category cat : animal;
7      production lion : cat = "roar";
8      production leopard : cat = "growl";
9
10     category dog : animal;
11     production wolf : dog = "howl";
12     production fox : dog = "bark";
13
14     // problematic
15     production unreal : cat, dog = "talk" @flag:not_real;
16 }
```

Figure 3.5: An MMP grammar file about subtypes

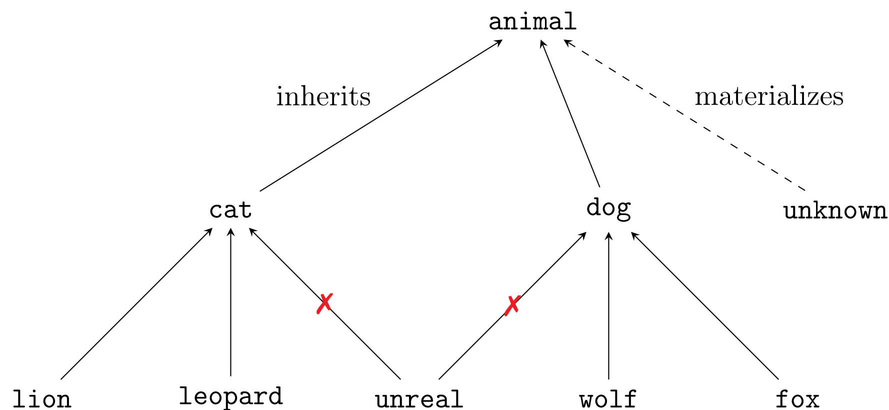


Figure 3.6: Subtypes in figure 3.5

A tangible subtype in MMP is perceived as a type-forming statement that directly has some grammar rule for expansion, and such grammar rule can either

be its own (like `production`) or comes from a helper (like a `pattern` materializing a `category`). This idea of directly accessible grammar rules achieves a disjoint partition where no two tangible subtypes share any grammar rule, and such result is crucial for some deterministic steps in parser generation (e.g., NFA-to-DFA conversion, lookahead computation, etc.).

We hinted in previous sections that (i) when a storage place (either a field or the output stream) is designated to store instances of a type-forming statement s, what actually stores there are tangible subtypes of s; and (ii) when a semantic action requires its input regex to reference a type-forming statement s, every nominal subtype of s is a qualified candidate. These two regulations comply with our vision for MMP subtypes, namely, nominal subtypes are used to check the validity of declarations, and tangible subtypes are actually produced objects.

Chapter 4

MMP Backend

This chapter focuses on the backend of MMP, namely, how the internal representations of various machines are constructed and how their corresponding parser are generated.

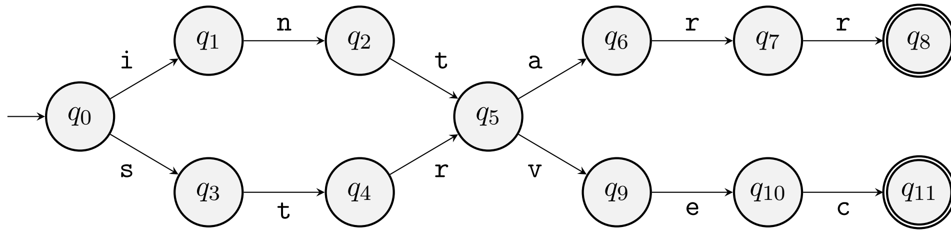
4.1 DFA Parser

DFA parsers are simple in terms of their state-based structure and memoryless nature. In the case of MMP, however, building DFA parsers is more involved due to the allowance of regex rules and embedded semantic actions. This section first talks about the additional restrictions imposed on MMP grammar files intended for DFA parsers, then details the two key steps in DFA construction in MMP.

4.1.1 Additional Restriction

An MMP grammar file with unrestricted grammar rules is equivalent to a context-free grammar (CFG), which is beyond the capability of DFA. For example, this grammar $s \rightarrow \texttt{(}s\texttt{)} \mid \varepsilon$ generates all nested parentheses, but does not have an equivalent DFA according to the pumping lemma [11].

There are several ways to constrain the expressiveness of a CFG to the level where an equivalent DFA can be constructed. One method [11] is to restrict the form of each grammar rule to be a terminal followed optionally by a nonterminal, which directly mirrors the transition function in the definition of DFA. Another method [12] is to stipulate that only those completely defined nonterminals can be referenced. Consider the three representations of regex (`int` | `str`)(`arr` | `vec`) shown in figure 4.1. Figure 4.1b is acquired by the first method and can be viewed as a straightforward translation of the DFA drawn in figure 4.1a. Figure 4.1c comes from the second method, and compared with figure 4.1b, it preserves more essence of the described language (i.e., a prefix denoting the element type and a postfix denoting the storage type) from a reader's perspective.



(a) DFA

$$
\begin{aligned}
q_0 &\to \mathtt{i}q_1 \mid \mathtt{s}q_3 \\
q_1 &\to \mathtt{n}q_2 \\
q_2 &\to \mathtt{t}q_5 \\
q_3 &\to \mathtt{t}q_4 \\
q_4 &\to \mathtt{r}q_5 \\
q_5 &\to \mathtt{a}q_6 \mid \mathtt{v}q_9 \\
q_6 &\to \mathtt{r}q_7 \\
q_7 &\to \mathtt{r} \\
q_9 &\to \mathtt{e}q_{10} \\
q_{10} &\to \mathtt{c}
\end{aligned}
$$

(b) CFG using method one

$$
\begin{aligned}
\langle \mathit{prefix} \rangle &\to \mathtt{int} \mid \mathtt{str} \\
\langle \mathit{postfix} \rangle &\to \mathtt{arr} \mid \mathtt{vec} \\
\langle \mathit{name} \rangle &\to \langle \mathit{prefix} \rangle \langle \mathit{postfix} \rangle
\end{aligned}
$$

(c) CFG using method two

Figure 4.1: Three representations of regex $(\mathtt{int} \mid \mathtt{str})(\mathtt{arr} \mid \mathtt{vec})$

In MMP, we adopt the second method and make it more flexible in that a reference regex can point to a statement defined anywhere in the current machine as stated in section 2.1.3, as long as no circular reference chain exist. Note that we ignore the self-loops caused by the statements generated for eliminating repetition notations mentioned in section 3.1 (i.e., $r_{\text{list}} \to r_{\text{item}} r_{\text{list}}$). Figure 4.2 demonstrates this idea, showing that `prefix` and `postfix` are used in `name`'s rule before their own definitions.

```
1 finite automaton name_scanner
2 {
3     // productions have root rootness and terminal terminality
4     // by default in DFA
5     production name = prefix postfix;
6     pattern prefix = "int" | "str";
7     pattern postfix = "arr" | "vec";
8 }
```

Figure 4.2: An MMP grammar file equivalent to figure 4.1

4.1.2 NFA Construction

Compared with DFA, a nondeterministic finite automaton (NFA) is easier to build from a grammar, and thus serves as the first step towards constructing a DFA parser. The NFA construction in MMP roughly follows the bottom-up approach introduced in [13], but some extra care needs to be taken since the semantic actions specified in the MMP grammar file need to be embedded into NFA states systematically to preserve their original intensions in the grammar. Figure 4.3 is an MMP grammar file for NFA construction. Figure 4.4 is the NFA equivalent to figure 4.3 visualized automatically by MMP.

```
1  finite automaton number_scanner
2  with productions_nonterminal_by_default
3  {
4      // productions have root rootness by default in DFA
5      category number {raw digits;};
6      production float : number = digit+ '.' digit{0,2};
7      production hex : number = ("0x" | "0X") (digit | letter)+;
8      pattern digit : float, hex = ['0'-'9'] @append:digits;
9      regex letter = ['a'-'z' 'A'-'Z'];
10 }
```

Figure 4.3: An MMP grammar file for NFA construction

We use figures 4.3 and 4.4 as an example to elaborate some key ideas in NFA construction in MMP.

- State `0` has two outgoing branches, one starting from state `1` and one starting from state `13`, and they correspond to the two `root` statements `float` and `hex`, respectively.
- The branch corresponding to `hex` is a concatenation of three components: state set $\{1, 2, 3, 4, 5\}$ responsible for `digit+`, state set $\{5, 6\}$ responsible for `'.'`, and state set $\{6, 7, 8, 9, 10, 11, 12\}$ responsible for `digit{0,2}`.
- State set $\{1, 2, 3, 4, 5\}$ uses a loop ($3 \xrightarrow{0\ldots9} 4 \xrightarrow{\varepsilon} 3$) to continuously match the repeated pattern `digit` since the repetition notation `+` specifies no maximum occurrence. On the other hand, state set $\{6, 7, 8, 9, 10, 11, 12\}$ lists all possibilities (7, $8 \xrightarrow{0\ldots9} 9$, and $10 \xrightarrow{0\ldots9} 11 \xrightarrow{0\ldots9} 12$) since the repetition notation `{0,2}` only allows a finite number of repetition.
- Some structural information about the grammar file is lost during the NFA construction, which is compensated by automatically inserting some auxiliary semantic actions into suitable states.
  - The three `raw`-related semantic actions (i.e., `@capture`, `@append`, and `@prepend`) do not know when to start recording input symbols in NFA,

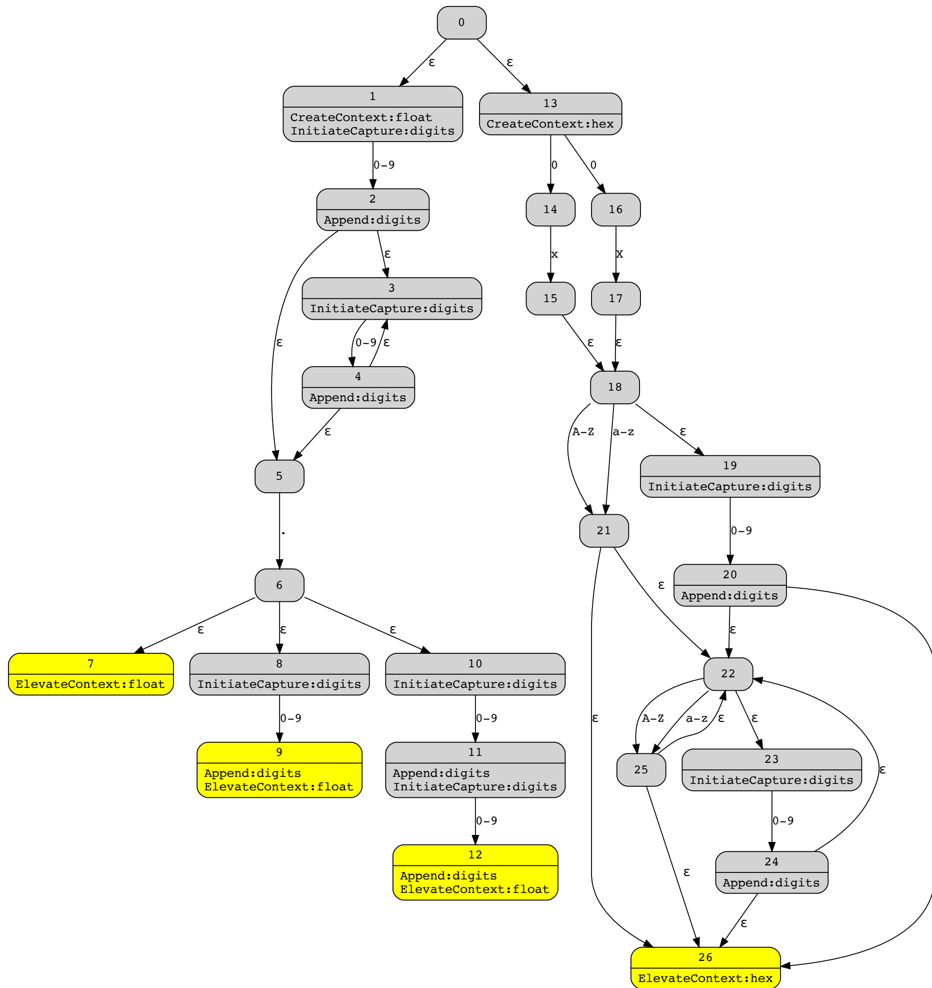


Figure 4.4: NFA equivalent to figure 4.3 visualized by MMP

Accept states are colored yellow; all other states are colored gray.

a purely state-based model. Therefore, a new action `InitiateCapture` is employed to signal the start of recording. For example, in transition `1` $\xrightarrow{\texttt{0...9}}$ `2`, the `InitiateCapture` in state `1` echos the `Append` in state `2`, thus indicating the range of recording.

– The justification for adopting `InitiateCapture` also applies to creating and concluding an instance of a type-forming statement. Thus, two new indicator actions `CreateContext` and `ElevateContext` are invented. Observe how the `CreateContext` in state `1` pairs with the

`ElevateContext` in states `7`, `9`, and `12` to instantiate `float`.

- When combining two components as mentioned in the first two bullet points, some states that can be merged in an actionless setting should remain separated to avoid action conflicts. For example, the start state `1` for statement `float` and the start state `13` for statement `hex` cannot be merged since the action `InitiateCapture` in state `1` is incompatible with `hex`, which has a non-`digit` prefix (`0x` or `0X`). Therefore, a new start state `0` is created and connected to states `1` and `13` via ε-transitions.

4.1.3 NFA-to-DFA Conversion

The final step of building a DFA parser is to convert the NFA discussed in section 4.1.2 to an equivalent DFA. Such conversion roughly follows the subset construction method introduced in [14], but some extra information is recorded in this process to facilitate tracing NFA states and their embedded semantic actions when paring input streams. Figure 4.5 is the DFA equivalent to figure 4.4 visualized automatically by MMP.

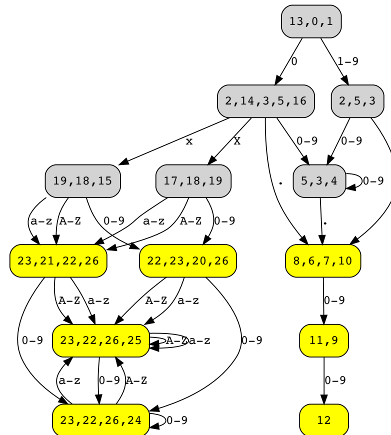


Figure 4.5: DFA equivalent to figure 4.4 visualized by MMP

Accept states are colored yellow; all other states are colored gray.

We use the transition $\{17, 18, 19\} \xrightarrow{\texttt{0}} \{22, 23, 20, 26\}$ in figure 4.5 as an example to elaborate some key ideas in NFA-to-DFA conversion in MMP.

- `0`-closure $\{20\}$ is computed by examining which NFA states are reachable from NFA states in $\{17, 18, 19\}$ via `0`-transition. In our example, the `0`-closure is a singleton set $\{20\}$ since the only applicable NFA transition is

$\texttt{19} \xrightarrow{\texttt{0}} \texttt{20}$, which is recorded to track how this 0-closure is derived.

- ε-closure $\{22, 23, 20, 26\}$ is computed by examining which NFA states are reachable from NFA states in the 0-closure $\{20\}$. In our example, the ε-closure is $\{22, 23, 20, 26\}$ since the applicable NFA transitions are $20 \xrightarrow{\varepsilon} 22 \xrightarrow{\varepsilon} 23$ and $20 \xrightarrow{\varepsilon} 26$, which are recorded to track how this ε-closure is derived.
- The spanning diagram shown in figure 4.6 tracks how $\{17, 18, 19\}$ goes to $\{22, 23, 20, 26\}$ via input symbol 0. It is derived from the recordings in the first two bullet points and will be consulted during parse time for performing semantic actions.

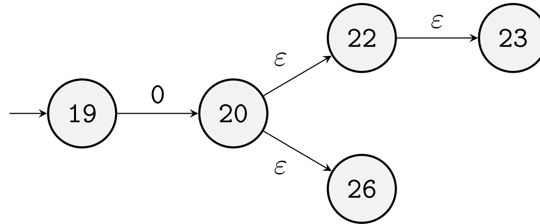


Figure 4.6: Spanning diagram for $\{17, 18, 19\} \xrightarrow{\texttt{0}} \{22, 23, 20, 26\}$

We use the text input stream 0xF as an example to show how to find the underlying NFA state path and perform the corresponding semantic actions. The DFA state path is

$$\{13, 0, 1\} \xrightarrow{\texttt{0}} \{2, 14, 3, 5, 16\} \xrightarrow{\texttt{x}} \{19, 18, 15\} \xrightarrow{\texttt{F}} \{23, 21, 22, 26\}$$

Since the accept NFA state 26 announces $\{23, 21, 22, 26\}$ as an accept DFA state, the spanning diagrams of the three DFA transitions are used in a reversed manner to find the underlying NFA state path

$$\left[0 \xrightarrow{\varepsilon} 13\right] \xrightarrow{\texttt{0}} \left[14\right] \xrightarrow{\texttt{x}} \left[15 \xrightarrow{\varepsilon} 18\right] \xrightarrow{\texttt{F}} \left[21 \xrightarrow{\varepsilon} 26\right]$$

By following this NFA state path and executing the embedded semantic actions of each NFA state, 0xF is successfully parsed and returned as a `hex`.

4.2 Lookahead Computation

Lookahead is a decision-making mechanism that resolves nondeterministic and conflicting situations during parse time by peeking some extra symbols in the input stream, and it is an essential component in many parsers. For example, in LL parsing, lookahead is utilized to help choosing the most appropriate alternative among several ones belonging to a nonterminal [15]. Another example is

LR parsing, where lookahead is employed to identify from an item set the most suitable one, whose associated action (shift or reduce) will be performed [16].

This section introduces the notion of sequential lookahead, namely, given a finite sequence of input symbols that leads to a specific position in the grammar, what the next expected input symbol would be. It will be applied in sections 4.3 and 4.4. This idea of incorporating contextual information into lookahead computation mainly comes from [17]. Figure 4.7 is an MMP grammar file for computing sequential lookahead.

```
1 LL(finite) parser strings
2 with productions_root_by_default
3 {
4     production string =
5         'a' letter ('b' | 'c') | 'a' letter ('d' | 'e');
6     production letter = 'x' | 'y';
7 }
```

Figure 4.7: An MMP grammar file for lookahead computation

The two `a` appearing in the first and the second alternatives of `string` will be referred to as $\mathtt{a}_1$ and $\mathtt{a}_2$ respectively to signal that they are different instances. Similarly, the two references to `letter` used in `string` will be addressed as $\mathtt{letter}_1$ and $\mathtt{letter}_2$.

4.2.1 Grammar Preprocessing

Some structural information about the grammar file is required for computing lookahead. Figure 4.8 illustrates the three dictionaries S (start), N (next), and F (follow) that are build based on figure 4.7 to facilitate the lookahead computation.

$$S(\mathtt{string}) = \{\mathtt{a}_1, \mathtt{a}_2\} \qquad S(\mathtt{letter}) = \{\mathtt{x}, \mathtt{y}\}$$

$$N(\mathtt{a}_1) = \{\mathtt{letter}_1\} \qquad N(\mathtt{letter}_1) = \{\mathtt{b}, \mathtt{c}\} \qquad N(\mathtt{a}_2) = \{\mathtt{letter}_2\} \qquad N(\mathtt{letter}_2) = \{\mathtt{d}, \mathtt{e}\}$$

$$F(\mathtt{x}) = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\} \qquad F(\mathtt{y}) = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$$

Figure 4.8: Three dictionaries S, N, and F built upon figure 4.7

All empty entries (i.e., have value $\varnothing$) are omitted in this figure.

- Dictionary S maps a rule statement to the set of its start primitive regexes.

For example, statement `letter` in figure 4.7 starts with either `x` or `y`, so we have $S(\texttt{letter}) = \{\texttt{x}, \texttt{y}\}$.

- Dictionary N maps a primitive regex to the set of primitive regexes that are next to it in the grammar rule. For example, $\texttt{letter}_1$ in figure 4.7 has either `b` or `c` as its descendent in the alternative `'a' letter ('b' | 'c')`, so we have $N(\texttt{letter}_1) = \{\texttt{b}, \texttt{c}\}$.

- Dictionary F maps a primitive regex that is at the end of a grammar rule to the set of primitive regexes that are followers in callers' contexts. For example, `x` in figure 4.7 can be followed by `b`, `c`, `d`, and `e`, which are what follow the use of `letter` in the rule of `string`. Thus, we have $F(\texttt{x}) = \{\texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$.

These three dictionaries can be constructed by traversing (e.g., in a depth-first way) the tree-like structure of each grammar rule discussed in section 2.2.2 and bookkeeping necessary information.

4.2.2 Sequential Lookahead

The idea of sequential lookahead is to compute the next expected input symbol based on a finite sequence of input symbols that leads to a specific position in the grammar. Formally, the sequential lookahead is a function L that returns a set of input symbols when fed with three inputs: (i) a finite sequence of input symbols, and (ii) a primitive regex that tracks the consumption of the sequence in the grammar, and (iii) a stack where each element is a set of primitive regexes that serve as callbacks.

Figure 4.9 illustrates how the value $\{\texttt{b}, \texttt{c}\}$ of $L(\texttt{ax}, \texttt{a}_1, [\,])$ is recursively computed and accumulated. We use it as an example to elaborate some key ideas in computing sequential lookahead.

- When computing $L(\texttt{ax}, \texttt{a}_1, [\,])$, the position indicator $\texttt{a}_1$ matches the first symbol in sequence `ax`, so $\texttt{a}_1$ is advanced to $\texttt{letter}_1$ according to dictionary N, and `ax` pops its head `a`. Thus, $L(\texttt{ax}, \texttt{a}_1, [\,])$ is reduced to $L(\texttt{x}, \texttt{letter}_1, [\,])$.

- When computing $L(\texttt{x}, \texttt{letter}_1, [\,])$, the position indicator $\texttt{letter}_1$ calls statement `letter`, so $\texttt{letter}_1$ is advanced to its start primitive regexes listed in $S(\texttt{letter})$, and the callback stack is populated with $N(\texttt{letter}_1)$ to record the return address after exploring statement `letter`. Thus, $L(\texttt{x}, \texttt{letter}_1, [\,])$ is reduced to $L(\texttt{x}, \texttt{x}, [\{\texttt{b}, \texttt{c}\}])$ and $L(\texttt{x}, \texttt{y}, [\{\texttt{b}, \texttt{c}\}])$.

- When computing $L(\texttt{x}, \texttt{x}, [\{\texttt{b}, \texttt{c}\}])$, the position indicator `x` matches the first symbol in sequence `x`. Moreover, since it is an end of its residing

rule, the callback stack is consulted for the next position indicator. Thus, $L(\mathtt{x}, \mathtt{x}, [\{\mathtt{b}, \mathtt{c}\}])$ is reduced to $L(\varepsilon, \mathtt{b}, [\,])$ and $L(\varepsilon, \mathtt{c}, [\,])$.

- When computing $L(\mathtt{x}, \mathtt{y}, [\{\mathtt{b}, \mathtt{c}\}])$, the position indicator $\mathtt{y}$ does not match (and thus incompatible with) the first symbol in sequence $\mathtt{x}$, so the result is trivially $\varnothing$.
- When computing $L(\varepsilon, \mathtt{b}, [\,])$ and $L(\varepsilon, \mathtt{c}, [\,])$, the sequence is empty, so the position indicator is returned as result ($\{\mathtt{b}\}$ and $\{\mathtt{c}\}$).

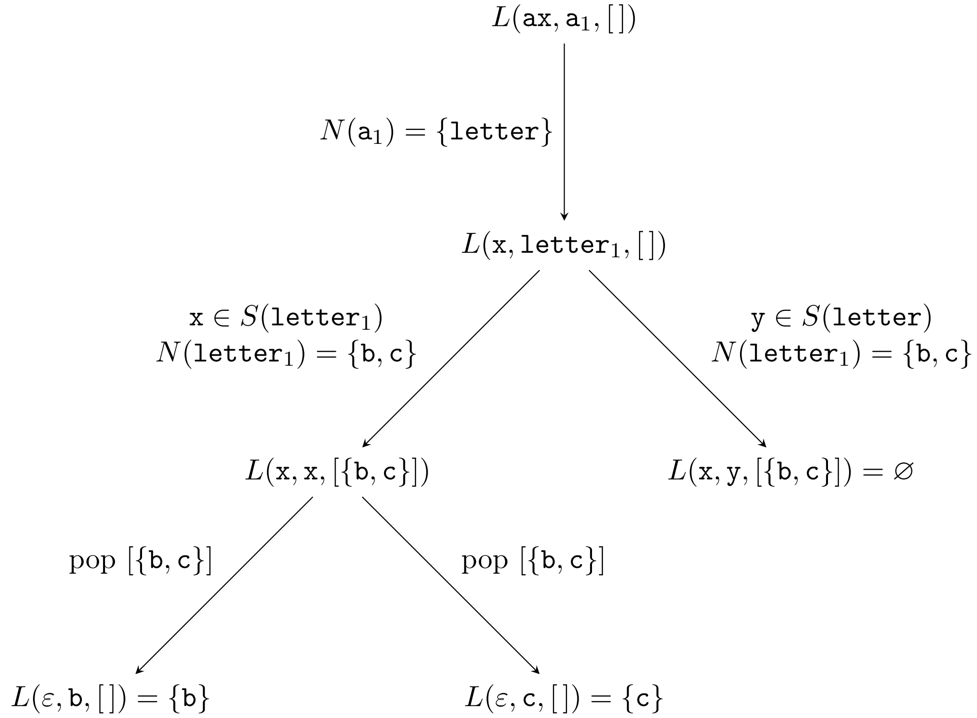


Figure 4.9: Computing $L(\mathtt{ax}, \mathtt{a}_1, [\,])$ based on figure 4.7

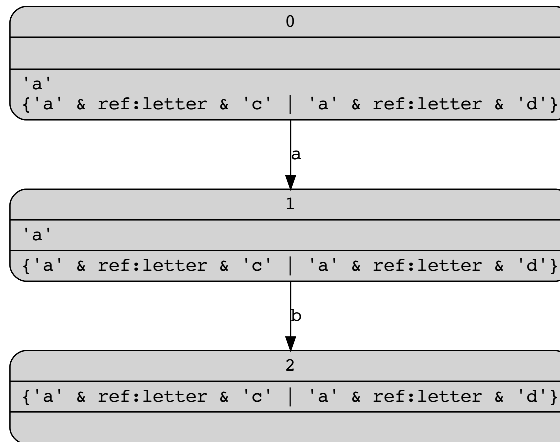
4.3 LL(k/finite) Parser

An LL parser is a collection of branching point processors, each of which is responsible for picking the most suitable path for its corresponding branching point. In our LL parser implementation, each branching point processor is equipped with a tree-shape DFA called lookaheader that employs the sequential lookahead mechanism discussed in section 4.2.2 to continuously partition a set of alternatives until a unique alternative is determined for each situation.

```
1 LL(finite) parser strings
2 with productions_root_by_default
3 {
4     production string = 'a' | ('a' letter 'c' | 'a' letter 'd');
5     regex letter = 'b';
6 }
```

Figure 4.10: An MMP grammar file for LL lookaheaders

The three `a` and two references to `letter` in `string`'s rule will be addressed as $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$, $\mathtt{letter}_1$, and $\mathtt{letter}_2$ respectively to signal that they are different instances.



(a) LL lookaheader visualized by MMP

state 0, prefix ε

$$L(\varepsilon, \mathtt{a}_1, [\,]) = \{\mathtt{a}\} \quad \text{go to state } 1$$

$$L(\varepsilon, \mathtt{a}_2, [\,]) \cup L(\varepsilon, \mathtt{a}_3, [\,]) = \{\mathtt{a}\} \cup \{\mathtt{a}\} = \{\mathtt{a}\} \quad \text{go to state } 1$$

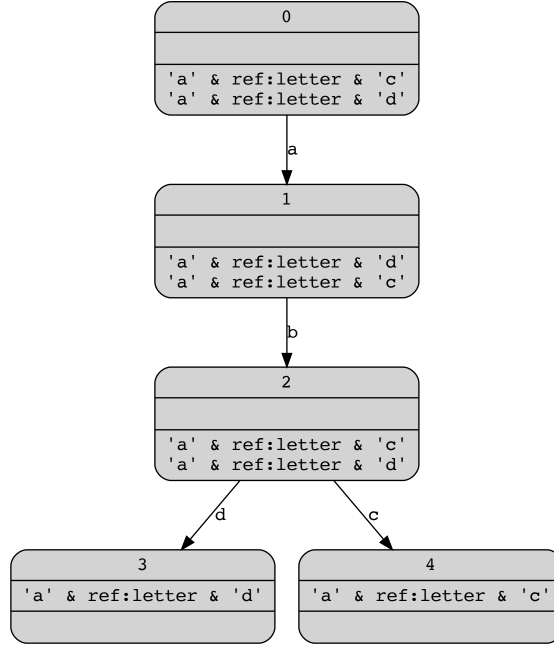
state 1, prefix $\mathtt{a}$

$$L(\mathtt{a}, \mathtt{a}_1, [\,]) = L(\varepsilon, \dashv, [\,]) = \varnothing \quad \text{stay at state } 1$$

$$\begin{aligned} & L(\mathtt{a}, \mathtt{a}_2, [\,]) \cup L(\mathtt{a}, \mathtt{a}_3, [\,]) \\ =& L(\varepsilon, \mathtt{letter}_1, [\,]) \cup L(\varepsilon, \mathtt{letter}_2, [\,]) \\ =& L(\varepsilon, \mathtt{b}, [\{\mathtt{c}\}]) \cup L(\varepsilon, \mathtt{b}, [\{\mathtt{d}\}]) \\ =& \{\mathtt{b}\} \cup \{\mathtt{b}\} = \{\mathtt{b}\} \quad \text{go to state } 2 \end{aligned}$$

(b) Construction of LL lookaheader

Figure 4.11: `'a' | ('a' letter 'c' | 'a' letter 'd')` in figure 4.10



(a) LL lookaheader visualized by MMP

state 0, prefix ε

$$L(\varepsilon, \mathtt{a}_2, [\,]) = \{\mathtt{a}\} \quad \text{go to state } \mathtt{1}$$

$$L(\varepsilon, \mathtt{a}_3, [\,]) = \{\mathtt{a}\} \quad \text{go to state } \mathtt{1}$$

state 1, prefix a

$$\begin{aligned} &L(\mathtt{a}, \mathtt{a}_2, [\,]) = L(\varepsilon, \mathtt{letter}_1, [\,]) \\ =&L(\varepsilon, \mathtt{b}, [\mathtt{c}]) = \{\mathtt{b}\} \qquad \text{go to state } \mathtt{2} \\ &L(\mathtt{a}, \mathtt{a}_3, [\,]) = L(\varepsilon, \mathtt{letter}_2, [\,]) \\ =&L(\varepsilon, \mathtt{b}, [\mathtt{d}]) = \{\mathtt{b}\} \qquad \text{go to state } \mathtt{2} \end{aligned}$$

state 2, prefix ab

$$\begin{aligned} &L(\mathtt{ab}, \mathtt{a}_2, [\,]) = L(\mathtt{b}, \mathtt{letter}_1, [\,]) \\ =&L(\mathtt{b}, \mathtt{b}, [\mathtt{c}]) = L(\varepsilon, \mathtt{c}, [\,]) = \{\mathtt{c}\} \qquad \text{go to state } \mathtt{4} \\ &L(\mathtt{ab}, \mathtt{a}_3, [\,]) = L(\mathtt{b}, \mathtt{letter}_2, [\,]) \\ =&L(\mathtt{b}, \mathtt{b}, [\mathtt{d}]) = L(\varepsilon, \mathtt{d}, [\,]) = \{\mathtt{d}\} \qquad \text{go to state } \mathtt{3} \end{aligned}$$

(b) Construction of LL lookaheader

Figure 4.12: `('a' letter 'c' | 'a' letter 'd')` in figure 4.10

Figures 4.11 and 4.12 illustrate the construction of two lookaheaders built for the two branching points in figure 4.10. Each state is associated with a prefix recording how this state is reached from the starting point by a finite sequence of input symbols. This prefix serves as the first argument passed to function L that computes for each alternative enclosed in this state the next expected input symbol, which is the criterion for continuous partitioning until each state possesses exactly one alternative. For example, state `2` in figure 4.12 has prefix `ab` and encompasses two alternatives, `'a' letter 'c'` and `'a' letter 'd'`. The function L is invoked to compute the next expected input symbol based on the prefix, thus distinguishing the two alternatives by creating transitions $\mathtt{2} \xrightarrow{\mathtt{c}} \mathtt{4}$ and $\mathtt{2} \xrightarrow{\mathtt{d}} \mathtt{3}$.

The lookahead length in our LL parser implementation specifies the maximum depth of a lookaheader, and it can be either a nonnegative integer k or a keyword `finite`. For example, the grammar in figure 4.10 is allowed to be `LL(3)`, `LL(4)`, . . . , `LL(finite)`, since the depth of the lookaheader in figure 4.12 is 3.

4.4 LR(k/finite) Parser

Compared with LL parsers, LR parsers make decisions in a delayed manner. An LL parser determines the path before actually walking on it; an LR parser, however, tries all paths simultaneously and then announces the one it just finished. The backbone of an LR parser is a pushdown automaton (PDA), a state-based formalism augmenting DFA with an additional stack for memory. To track all possible paths simultaneously, each PDA state encompasses a set of items, each of which is a marked grammar rule recording how much this rule is consumed.

Since each grammar rule in MMP is a tree-like structure as discussed in section 2.2.2, two logically consecutive primitive regexes (or figuratively, leaves) are required to trace the consumption of the rule. For example, there are 8 configurations for rule `('a' | 'b')('c' | 'd')`: $(\vdash, \mathtt{a})$, $(\vdash, \mathtt{b})$, $(\mathtt{a}, \mathtt{c})$, $(\mathtt{a}, \mathtt{d})$, $(\mathtt{b}, \mathtt{c})$, $(\mathtt{b}, \mathtt{d})$, $(\mathtt{c}, \dashv)$, $(\mathtt{d}, \dashv)$. Figure 4.15 illustrates the PDA constructed for figure 4.13. Observe how the consumption of `expr`'s rule is tracked by the three locator pairs within item `2` in state `0`, item `8` in state `2`, and item `16` in state `5`.

To determine whether to continue reading input or to announce a finished path, an LR parser equips each of its PDA states with a lookaheader similar to the one introduced in section 4.3. For example, in state `2` in figure 4.15, items `6` and `10` are incompatible with item `9` since the first two expect to read the next input symbol (action `Shift`), whereas the last one tries to conclude its corresponding rule (action `Reduce`). The lookaheader shown in figure 4.14 resolves this conflict by peeking the next input symbol. If the end of input stream is encountered, `Reduce` is the only viable option; otherwise, the next input symbol is expected to be an `op` (either `+` or `-`), and `Shift` is executed to consume the input.

```
LR(finite) parser expression
{
    root production expr = bit (op bit)*;
    regex bit = '0' | '1';
    regex op = '+' | '-';
}
```

Figure 4.13: An example MMP grammar file for PDA

```
def proceed2(currIter: list[Iterator[str]], stack: list[StackElement])
    -> StackElement:
    initIter = copy(currIter[0])
    currState = 0
    currInputSymbol = next(currIter[0], None)
    inputEnd = currInputSymbol == None

    # Lookaheader for PDA state 2
    while currInputSymbol:
        match currState:
            case 0:
                match currInputSymbol:
                    case '+':
                        currState = 1
                    case '-':
                        currState = 2
                    case _:
                        break
            case 2:
                match currInputSymbol:
                    case _:
                        break
            case 1:
                match currInputSymbol:
                    case _:
                        break
        currInputSymbol = next(currIter[0], None)

    currIter[0] = initIter

    # Execute action (shift/reduce/use) for PDA state 2
    match currState:
        case 0:
            return reduce(stack, 'expr__list_3_29', 9)
        case 2:
            return shift(currIter)
        case 1:
            return shift(currIter)
```

Figure 4.14: Python code for PDA state 2 in figure 4.15 generated by MMP

Figure 4.15: PDA based on figure 4.13 visualized by MMP

Chapter 5

Future Work

This chapter outlines some future directions to extend our current work.

- More machines. As stated in section 2.3, MMP currently supports three types of machines: DFA, LL(k/finite), and LR(k/finite). More parsing algorithms can be integrated into MMP to provide users with a broader selection of machines.
- More fields and semantic actions. As discussed in section 2.2.3, the fields in MMP are either primitive (only a boolean-like `flag`, to be exact) or linear generic containers, and their corresponding semantic actions are not fine-grained enough. It might be beneficial to incorporate fields of other primitive types (e.g., integer, float, etc.) and of nonlinear generic containers (e.g., priority queue, graph, etc.), as well as semantic actions of higher granularity (e.g., bit/arithmetic/logical operations on primitive types, set the i-th element of a container type, etc.).
- More flexible grammar rules. As shown in appendix A, MMP currently has limited support for range selection in that users can only specify what they want in text input layer. It might be helpful to add range selection on the level of nonterminals, as well as reverse selection (i.e., users specify what they do not want).
- Callable statements. ANTLR [8] allows a nonterminal to have parameters that receive arguments from its callers (i.e., those places where this nonterminal is referenced). It might be favorable to have this feature of callable nonterminals in MMP as well.

Bibliography

[1] P. M. Lewis and R. E. Stearns, “Syntax-directed transduction,” *Journal of the ACM (JACM)*, vol. 15, no. 3, pp. 465–488, 1968.

[2] M. Might and D. Darais, “Yacc is dead,” 2010. [Online]. Available: https://arxiv.org/abs/1010.5023

[3] R. Cox. Yacc is not dead. [Online]. Available: https://research.swtch.com/yaccalive

[4] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[5] V. Paxson, W. Estes, and J. Millaway, “Lexical analysis with flex,” *University of California*, p. 28, 2007.

[6] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[7] C. Donnelly, “Bison the yacc-compatible parser generator,” *Technical report, Free Software Foundation*, 1988.

[8] T. Parr. Antlr official webside. [Online]. Available: https://www.antlr.org/

[9] T. Parr and K. Fisher, “Ll (*) the foundation of the antlr parser generator,” *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[10] T. Parr, S. Harwell, and K. Fisher, “Adaptive ll (*) parsing: the power of dynamic analysis,” *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, 2014.

[11] D. Grune and C. J. H. Jacobsk, *Parsing Techniques: A Practical Guide*, 2nd ed. Springer New York, NY, 2008.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed. Addison-Wesley, 2007.

[13] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[14] M. O. Rabin and D. Scott, “Finite automata and their decision problems,” *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959.

[15] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top-down grammars," *Information and Control*, vol. 17, no. 3, pp. 226–256, 1970.

[16] D. E. Knuth, "On the translation of languages from left to right," *Information and control*, vol. 8, no. 6, pp. 607–639, 1965.

[17] P. Belcak, "The ll(finite) strategy for optimal ll(k) parsing," 2020. [Online]. Available: https://arxiv.org/abs/2010.07874

Appendix A

MMP Specification Language

Below is the complete grammar (on token level) for MMP specification language written in MMP.

```
/*
    KW_USES,
    KW_ON,
    KW_WITH,

    KW_FINITE,
    KW_AUTOMATON,
    KW_LL,
    KW_LR,
    KW_PARSER,

    KW_PRODUCTIONS_TERMINAL_BY_DEFAULT,
    KW_PRODUCTIONS_NONTERMINAL_BY_DEFAULT,
    KW_PRODUCTIONS_ROOT_BY_DEFAULT,
    KW_PRODUCTIONS_NONROOT_BY_DEFAULT,
    KW_CATEGORIES_ROOT_BY_DEFAULT,
    KW_CATEGORIES_NONROOT_BY_DEFAULT,
    KW_AMBIGUITY_DISALLOWED,
    KW_AMBIGUITY_RESOLVED_BY_PRECEDENCE,

    KW_IGNORED,
    KW_ROOT,
    KW_TERMINAL,
    KW_NONTERMINAL,
    KW_CATEGORY,
    KW_PRODUCTION,
    KW_PATTERN,
    KW_REGEX,

    KW_ITEM,
    KW_LIST,
    KW_RAW,

```

```
34      KW_FLAG,
35      KW_UNFLAG,
36      KW_CAPTURE,
37      KW_EMPTY,
38      KW_APPEND,
39      KW_PREPEND,
40      KW_SET,
41      KW_UNSET,
42      KW_PUSH,
43      KW_POP,
44      KW_CLEAR,
45
46      IDENTIFIER,
47      STRING,
48      NUMBER,
49
50      PAR_LEFT,
51      PAR_RIGHT,
52      SQUARE_LEFT,
53      SQUARE_RIGHT,
54      CURLY_LEFT,
55      CURLY_RIGHT,
56
57      OP_COLON,
58      OP_EQUALS,
59      OP_LEFTARR,
60      OP_SEMICOLON,
61      OP_COMMA,
62      OP_DOT,
63      OP_CARET,
64      OP_DOLLAR,
65
66      OP_STAR,
67      OP_PLUS,
68      OP_QM,
69      OP_OR,
70      OP_FWDSLASH,
71
72      OP_AMPERSAND,
73      OP_DASH,
74      OP_AT,
75
76      EOS
77  */
78
79  /* ~~~~~~~~~~~~~~~~~~~~ top-level entities ~~~~~~~~~~~~~~~~~~~~ */
80
81  production SpecificationFile =
82      (MachineDefinition | UsesStatement)*
```

```
83  ;
84
85  production UsesStatement =
86      KW_USES STRING OP_SEMICOLON
87  ;
88
89  /* ~~~~~~~~~~~~~~~~~~~~ machine entities ~~~~~~~~~~~~~~~~~~~~ */
90
91  production MachineDefinition =
92      machineType IDENTIFIER
93      (KW_WITH machineOption (OP_COMMA machineOption)*)?
94      (KW_ON IDENTIFIER)?
95      (KW_USES IDENTIFIER (OP_COMMA IDENTIFIER)*)?
96      (CURLY_LEFT MachineStatement* CURLY_RIGHT | OP_SEMICOLON)
97  ;
98
99  regex machineType =
100     KW_FINITE KW_AUTOMATON
101     | KW_LL PAR_LEFT (NUMBER | KW_FINITE) PAR_RIGHT KW_PARSER
102 ;
103
104 regex machineOption =
105     KW_PRODUCTIONS_TERMINAL_BY_DEFAULT
106     | KW_PRODUCTIONS_NONTERMINAL_BY_DEFAULT
107     | KW_PRODUCTIONS_ROOT_BY_DEFAULT
108     | KW_PRODUCTIONS_NONROOT_BY_DEFAULT
109     | KW_CATEGORIES_ROOT_BY_DEFAULT
110     | KW_CATEGORIES_NONROOT_BY_DEFAULT
111     | KW_AMBIGUITY_DISALLOWED
112     | KW_AMBIGUITY_RESOLVED_BY_PRECEDENCE
113 ;
114
115 /* ~~~~~~~~~~~~~~~~~~~~ statement entities ~~~~~~~~~~~~~~~~~~~~ */
116
117 production MachineStatement =
118     CategoryStatement
119     | ProductionStatement
120     | PatternStatement
121     | RegexStatement
122 ;
123
124 production CategoryStatement =
125     rootnessElaboration? KW_CATEGORY IDENTIFIER
126     baseContextList? fieldDeclarationList? OP_SEMICOLON
127 ;
128
129 production ProductionStatement =
130     rootnessElaboration? terminalityElaboration? KW_PRODUCTION? IDENTIFIER
131     baseContextList? fieldDeclarationList? rule
```

```
132 ;
133
134 production PatternStatement =
135     KW_PATTERN IDENTIFIER baseContextList? rule
136 ;
137
138 production RegexStatement =
139     KW_REGEX IDENTIFIER rule
140 ;
141
142 regex terminalityElaboration =
143     KW_TERMINAL | KW_NONTERMINAL
144 ;
145
146 regex rootnessElaboration =
147     KW_ROOT | KW_IGNORED KW_ROOT
148 ;
149
150 regex baseContextList =
151     OP_COLON IDENTIFIER (OP_COMMA IDENTIFIER)*
152 ;
153
154 regex fieldDeclarationList =
155     CURLY_LEFT Field* CURLY_RIGHT
156 ;
157
158 regex rule =
159     OP_EQUALS DisjunctiveRegex OP_SEMICOLON
160 ;
161
162 /* ~~~~~~~~~~~~~~~~~~~~ field entities ~~~~~~~~~~~~~~~~~~~~ */
163
164 production Field =
165     (KW_FLAG | KW_RAW | IDENTIFIER (KW_LIST | KW_ITEM)?)
166     IDENTIFIER OP_SEMICOLON
167 ;
168
169 /* ~~~~~~~~~~~~~~~~~~~~ regex entities ~~~~~~~~~~~~~~~~~~~~ */
170
171 production DisjunctiveRegex =
172     ConjunctiveRegex (OP_OR ConjunctiveRegex)*
173 ;
174
175 production ConjunctiveRegex =
176     RootRegex+
177 ;
178
179 production RootRegex =
180     AtomicRegex | RepetitiveRegex
```

```
181 ;
182
183 production AtomicRegex =
184     ( PAR_LEFT DisjunctiveRegex PAR_RIGHT
185     | SQUARE_LEFT (STRING | regexRange)* SQUARE_RIGHT
186     | SQUARE_LEFT OP_CARET (STRING | regexRange)* SQUARE_RIGHT
187     | STRING
188     | KW_EMPTY | PAR_LEFT PAR_RIGHT
189     | OP_DOT
190     | IDENTIFIER
191     ) Action*
192 ;
193
194 production RepetitiveRegex =
195     AtomicRegex repetition Action*
196 ;
197
198 regex repetition =
199     OP_QM
200     | OP_STAR
201     | OP_PLUS
202     | CURLY_LEFT NUMBER OP_COMMA NUMBER CURLY_RIGHT
203 ;
204
205 regex regexRange =
206     STRING OP_DASH STRING
207 ;
208
209 /* ~~~~~~~~~~~~~~~~~~~~ action entities ~~~~~~~~~~~~~~~~~~~~ */
210
211 production Action =
212     OP_AT actionType OP_COLON IDENTIFIER
213 ;
214
215 regex actionType =
216     KW_FLAG
217     | KW_UNFLAG
218     | KW_CAPTURE
219     | KW_EMPTY
220     | KW_APPEND
221     | KW_PREPEND
222     | KW_SET
223     | KW_UNSET
224     | KW_PUSH
225     | KW_POP
226     | KW_CLEAR
227 ;
```

Appendix B

A Working Example

Below is an MMP grammar file for parsing C expressions.

```
finite automaton Tokenizer with ambiguity_resolved_by_precedence {
    root category seperator;

    root category unary_op;
    root category unary_post : unary_op;
    root category unary_pre : unary_op;

    root category binary_op;
    root category access_op : binary_op;
    root category multiplicative_op : binary_op;
    root category additive_op : binary_op;
    root category shift_op : binary_op;
    root category relational_op : binary_op;
    root category equality_op : binary_op;
    root category binary_bit_op : binary_op;
    root category binary_logical_op: binary_op;
    root category assign_op : binary_op;

    root category ternary_op;

    root category storage_class;

    root category type_specifier;
    root category primitive_type : type_specifier;
    root category composite_type : type_specifier;
    root category homogeneous_type : composite_type;
    root category heterogeneous_type : composite_type;

    root category type_qualifier;

    ignored root WHITESPACE = [' ''\t''\n']+;

    PAR_LEFT : seperator = '(';
    PAR_RIGHT : seperator = ')';
```

```
35    SQUARE_LEFT : seperator = '[';
36    SQUARE_RIGHT : seperator = ']';
37    CURLY_LEFT : seperator = '{';
38    CURLY_RIGHT : seperator = '}';
39    SEMICOLON : seperator = ';';
40    COMMA : seperator, binary_op = ',';
41
42    SIZEOF : unary_pre = "sizeof";
43    BIT_NOT_OP : unary_pre = '~';
44    NOT_OP : unary_pre = '!';
45    INC_OP : unary_pre, unary_post = "++";
46    DEC_OP : unary_pre, unary_post = "--";
47    STAR_OP : unary_pre, multiplicative_op = '*';
48    CROSS_OP : unary_pre, additive_op = '+';
49    DASH_OP : unary_pre, additive_op = '-';
50    AMP_OP : unary_pre, binary_bit_op = '&';
51
52    DOT_OP : access_op = '.';
53    PTR_OP : access_op = "->";
54    DIV_OP : multiplicative_op = '/';
55    MOD_OP : multiplicative_op = '%';
56    LEFT_OP : shift_op = "<<";
57    RIGHT_OP : shift_op = ">>";
58    LT_OP : relational_op = '<';
59    GT_OP: relational_op = '>';
60    LE_OP : relational_op = "<=";
61    GE_OP : relational_op = ">=";
62    EQ_OP : equality_op = "==";
63    NE_OP : equality_op = "!=";
64    BIT_XOR_OP : binary_bit_op = '^';
65    BIT_OR_OP : binary_bit_op = '|';
66    AND_OP : binary_logical_op = "&&";
67    OR_OP : binary_logical_op = "||";
68    QUESTION_OP : ternary_op = '?';
69    COLON_OP : ternary_op = ':';
70
71    ASSIGN : assign_op = "=";
72    MUL_ASSIGN : assign_op = "*=";
73    DIV_ASSIGN : assign_op = "/=";
74    MOD_ASSIGN : assign_op = "%=";
75    ADD_ASSIGN : assign_op = "+=";
76    SUB_ASSIGN : assign_op = "-=";
77    LEFT_ASSIGN : assign_op = "<<=";
78    RIGHT_ASSIGN : assign_op = ">>=";
79    AND_ASSIGN : assign_op = "&=";
80    XOR_ASSIGN : assign_op = "^=";
81    OR_ASSIGN : assign_op = "|=";
82
83    TYPEDEF : storage_class = "typedef";
```

```
84     EXTERN : storage_class = "extern";
85     STATIC : storage_class = "static";
86     AUTO : storage_class = "auto";
87     REGISTER : storage_class = "register";
88
89     CHAR : primitive_type = "char";
90     SHORT : primitive_type = "short";
91     INT : primitive_type = "int";
92     LONG : primitive_type = "long";
93     SIGNED : primitive_type = "signed";
94     UNSIGNED : primitive_type = "unsigned";
95     FLOAT : primitive_type = "float";
96     DOUBLE : primitive_type = "double";
97     VOID : primitive_type = "void";
98     ENUM : homogeneous_type = "enum";
99     STRUCT : heterogeneous_type = "struct";
100    UNION : heterogeneous_type = "union";
101
102    CONST : type_qualifier = "const";
103    VOLATILE : type_qualifier = "volatile";
104
105    ELLIPSIS = "...";
106
107    CASE = "case";
108    DEFAULT = "default";
109    IF = "if";
110    ELSE = "else";
111    SWITCH = "switch";
112    WHILE = "while";
113    DO = "do";
114    FOR = "for";
115    GOTO = "goto";
116    CONTINUE = "continue";
117    BREAK = "break";
118    RETURN = "return";
119
120    regex num = ['0'-'9'];
121    regex char =['a'-'z''A'-'Z'];
122
123    CONSTANT = num+;
124    STRING_LITERAL = '\"' (char | num)* '\"';
125    IDENTIFIER = char (char | num)*;
126 }
127
128 LR(finite) parser Type on Tokenizer {
129    root terminal type_name =
130        full_base declarator
131    ;
132
```

```
133     pattern full_base : type_name =
134         ( storage_class
135         | type_qualifier
136         | primitive_type
137         | composite_type_definition
138         )+
139     ;
140
141     pattern half_base : type_name =
142         ( type_qualifier
143         | primitive_type
144         | composite_type_definition
145         )+
146     ;
147
148     pattern composite_type_definition : type_name =
149         homogeneous_type IDENTIFIER?
150             (CURLY_LEFT homogeneous_list CURLY_RIGHT)?
151         | heterogeneous_type IDENTIFIER?
152             (CURLY_LEFT heterogeneous_list CURLY_RIGHT)?
153     ;
154
155     pattern homogeneous_list : type_name =
156         IDENTIFIER (COMMA IDENTIFIER)*
157     ;
158
159     pattern heterogeneous_list : type_name =
160         half_base declarator (COMMA half_base declarator)*
161     ;
162
163     pattern declarator_prefix : type_name =
164         STAR_OP (STAR_OP | type_qualifier)*
165     ;
166
167     pattern declarator_postfix : type_name =
168         ( SQUARE_LEFT CONSTANT SQUARE_RIGHT
169         | PAR_LEFT parameter_list? PAR_RIGHT
170         )+
171     ;
172
173     pattern declarator : type_name =
174         declarator_prefix?
175         (IDENTIFIER? | PAR_LEFT declarator PAR_RIGHT)
176         declarator_postfix?
177     ;
178
179     pattern parameter_list : type_name =
180         parameter (COMMA parameter)* (COMMA ELLIPSIS)?
181     ;
```

```
182
183     pattern parameter : type_name =
184         full_base declarator?
185     ;
186 }
187
188 LR(finite) parser Expr on Tokenizer uses Type {
189     category expr {expr list operands;};
190     category binary_expr : expr {binary_op list operators;};
191
192     primary_expr : expr =
193         IDENTIFIER
194         | CONSTANT
195         | STRING_LITERAL
196         | PAR_LEFT comma_expr PAR_RIGHT
197         | SIZEOF PAR_LEFT type_name PAR_RIGHT
198     ;
199
200     regex arg_list =
201         assign_expr (COMMA assign_expr)*
202     ;
203
204     regex unary_postfix =
205         SQUARE_LEFT comma_expr SQUARE_RIGHT
206         | PAR_LEFT arg_list* PAR_RIGHT
207         | access_op IDENTIFIER
208         | unary_post
209     ;
210
211     unary_expr : expr =
212         unary_pre* primary_expr unary_postfix*
213     ;
214
215     multiplicative_expr : binary_expr =
216         unary_expr@push:operands
217         (multiplicative_op@push:operators unary_expr@push:operands)*
218     ;
219
220     additive_expr : binary_expr =
221         multiplicative_expr@push:operands
222         (additive_op@push:operators multiplicative_expr@push:operands)*
223     ;
224
225     shift_expr : binary_expr =
226         additive_expr@push:operands
227         (shift_op@push:operators additive_expr@push:operands)*
228     ;
229
230     relational_expr : binary_expr =
```

```
231         shift_expr@push:operands
232         (relational_op@push:operators shift_expr@push:operands)*
233     ;
234
235     equality_expr : binary_expr =
236         relational_expr@push:operands
237         (equality_op@push:operators relational_expr@push:operands)*
238     ;
239
240     bit_and_expr : binary_expr =
241         equality_expr@push:operands
242         (AMP_OP@push:operators equality_expr@push:operands)*
243     ;
244
245     bit_xor_expr : binary_expr =
246         bit_and_expr@push:operands
247         (BIT_XOR_OP@push:operators bit_and_expr@push:operands)*
248     ;
249
250     bit_or_expr : binary_expr =
251         bit_xor_expr@push:operands
252         (BIT_OR_OP@push:operators bit_xor_expr@push:operands)*
253     ;
254
255     and_expr : binary_expr =
256         bit_or_expr@push:operands
257         (AND_OP@push:operators bit_or_expr@push:operands)*
258     ;
259
260     or_expr : binary_expr =
261         and_expr@push:operands
262         (OR_OP@push:operators and_expr@push:operands)*
263     ;
264
265     conditional_expr : expr =
266         or_expr@push:operands
267         (QUESTION_OP comma_expr@push:operands COLON_OP conditional_expr@push:operands)?
268     ;
269
270     assign_expr : binary_expr =
271         conditional_expr@push:operands
272         (assign_op@push:operators conditional_expr@push:operands)*
273     ;
274
275     root comma_expr : binary_expr =
276         assign_expr@push:operands
277         (COMMA@push:operators assign_expr@push:operands)*
278     ;
279 }
```

When given the text input stream `sizeof(int) == sizeof(char) * 4 ? (a & 15) + (78 - 12) :  ++b % 3 & 1 && (c == d)`, the generated parser produces the following output.