

# LL(1) parser implementation

Sif

January 2024

## 1 Abstract

This report details the construction and execution of an LL(1) parser for a given grammar. The process began with a verification step to ensure that the grammar was free of left recursion, which is a prerequisite for LL(1) parsing. Subsequently, the First and Follow sets for all productions in the grammar were computed, followed by the derivation of Predict sets. These sets were then used to construct a comprehensive LL(1) parsing table. To demonstrate the parser's functionality, instructions for compiling and running the implementation are provided, alongside a sample execution using the input string `id=id*id/integer;`. The execution process is meticulously described using the configurations of a Pushdown Automaton (PDA), highlighting each step of the parsing sequence and the corresponding stack operations.

## 2 Introduction

Parsing is a fundamental process in the field of compilers, where a sequence of tokens is analyzed to determine its grammatical structure under the rules of a formal grammar. LL(1) parsers are a class of top-down parsers that read the input from Left to right, construct a Leftmost derivation of the sentence, and use 1 token of lookahead to make parsing decisions. This report presents a detailed methodology for implementing an LL(1) parser for a specified grammar, which includes mathematical constructs and a step-by-step parsing procedure.

In compiler design, handling recursive productions correctly is crucial for the successful implementation of a parser. As such, the grammar provided was first examined for left recursion, a condition that LL(1) parsers cannot handle directly. The existence of left recursion in the given grammar would need to be dealt with in order to proceed with the computation of the First and Follow sets—critical tools in predicting parser actions.

$$\begin{aligned}
G(P) : \\
P &\rightarrow id := E; \\
E &\rightarrow EOT \\
E &\rightarrow T \\
T &\rightarrow id|integer \\
O &\rightarrow *//
\end{aligned}$$

Above is the grammar that will be used in this paper. As can be observed, there exist a left recursion with the second production rule. This will be dealt with before proceeding with the First and Follow set.

### 3 Eliminating Left Recursions

The grammar contains immediate left recursion in the non-terminal E. The form of left recursion is direct, as E is directly calling itself on the left side of the production:

$$E \rightarrow EOT$$

To remove left recursion, we need to restructure the grammar to eliminate it.

The general strategy to remove immediate left recursion is as follows:

- For a non-terminal A with left recursion, identify productions of the form:
  - $A \rightarrow A\alpha \mid \beta$
  - Where  $\alpha$  represents the recursive part, and  $\beta$  represents the non-recursive alternative(s).
- Replace these productions with non-recursive productions by introducing a new non-terminal A':
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A' \mid \epsilon$
  - Where  $\epsilon$  stands for the empty string.

In our case we identify A as E,  $\alpha$  as OT, and  $\beta$  as T. So, we introduce a new non-terminal E' to eliminate the left recursion providing us with the transformed grammar without left recursion:

$$\begin{aligned}
G'(P) : \\
P &\rightarrow id := E; \\
E &\rightarrow TE' \\
E' &\rightarrow OTE'|\epsilon \\
T &\rightarrow id|integer \\
O &\rightarrow */
\end{aligned}$$

The transformed grammar  $G'(P)$  is now free of left recursion and can be used for constructing LL(1) parsers or other types of top-down parsers.

## 4 First and Follow set

The First set of a non-terminal in a grammar is a set of terminals that begin the strings derivable from the non-terminal. To compute the First set for each non-terminal, we must iterate through the grammar's production rules.

- If a production begins with a terminal, that terminal is added to the First set of the non-terminal being expanded.
- If a production can derive  $\epsilon$ , it is also included in the First set.
- If the production begins with a non-terminal, the First set of that non-terminal is recursively added to the First set of the one being considered, excluding any  $\epsilon$  if present unless it is the only character derivable.

This process is repeated until no more terminals or  $\epsilon$  can be added to any First set. Below is the result following these rules:

$$\begin{aligned}
First(P) &= [id] \\
First(E) &= [id, integer] \\
First(E') &= [*, /, \epsilon] \\
First(T) &= [id, integer] \\
First(O) &= [*, /]
\end{aligned}$$

The Follow set of a non-terminal consists of terminals that can appear immediately to the right of the non-terminal in some "sentential" form derived from the start symbol. To compute the Follow set, we must again consider the production rules. The process begins by placing the end-of-input marker (usually represented by \$) in the Follow set of the start symbol. For each occurrence of a non-terminal B in the production  $A \rightarrow \alpha B \beta$ , where  $\alpha$  and  $\beta$  are any strings of grammar symbols, every terminal in the First set of  $\beta$  (excluding  $\epsilon$ ) is added to the Follow set of B. If  $\beta$  can derive  $\epsilon$  or if B is at the end of the production, then everything in the Follow set of A is added to the Follow set of B. This process

is iterative and continues until no new terminals can be added to the Follow set of any non-terminal. The computation of Follow sets is typically more complex than that of First sets, as it may require several passes over the production rules until all sets reach a fixed point. Below is the resulting Follow set:

$Follow(P) = [\$]$   
 $Follow(E) = [;]$   
 $Follow(E') = [;]$   
 $Follow(T) = [*, /, ;]$   
 $Follow(O) = [id, integer]$

Predict Sets:

The Predict set is essentially the First set of the sequence of symbols following the non-terminal in the production, except when the First set includes  $\epsilon$ , in which case we also add the Follow set of the non-terminal.

Production	Predict Set
$P \rightarrow id := E;$	{ id }
$E \rightarrow TE'$	{ id, integer }
$E' \rightarrow OTE'$	{ *, / }
$E' \rightarrow \epsilon$	{ ; }
$T \rightarrow id$	{ id }
$T \rightarrow integer$	{ integer }
$O \rightarrow *$	{ * }
$O \rightarrow /$	{ / }

## 5 LL(1) parsing table

Here is the LL(1) parsing table based on the grammar and the First and Follow sets. The table dictates which production rule to use, given a non-terminal on the stack (rows) and the next input symbol (columns).

	id	:=	*	/	integer	;	\$
P	$P \rightarrow id := E;$						
E	$E \rightarrow TE'$				$E \rightarrow TE'$		
E'			$E' \rightarrow OTE'$	$E' \rightarrow OTE'$		$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow id$				$T \rightarrow integer$		
O			$O \rightarrow *$	$O \rightarrow /$			

## 6 Implementing in C++

The structure of the C++ program is as such(omitting the details):

```
enum class TokenType {
    ID,           // "id"
    ASSIGN,       // "!="
    INTEGER,      // "integer"
    STAR,         // "*"
    SLASH,        // "/"
    SEMICOLON,    // ";"
    END,
    INVALID
};

struct Token {
    TokenType type;
    std::string lexeme;
};

class Lexer {
private:
    char getNextChar();
    void putBackChar();
    Token identifierOrKeyword();

public:
    explicit Lexer(const std::string &src) :
        source(src), currentChar(src.begin()), end(src.end()) {}
    Token getNextToken();
};

class Parser {
private:
    Lexer lexer;
    Token currentToken;

    void consume(TokenType type);

    void P();
    void E();
    void E_prime();
    void T();
    void O();

public:
    explicit Parser(const std::string &source)
        : lexer(Lexer(source)) {
        currentToken = lexer.getNextToken();
    }
    void parse();
};
```

**Token:** This is a simple struct that represents a token with a TokenType indicating the kind of token it is (ID, ASSIGN, INTEGER, etc.) and a 'lexeme' holding the actual text of the token.

**Lexer:** The Lexer class is responsible for tokenizing the input string. It converts the string into a sequence of tokens, where each token is a meaningful sequence of characters (like an identifier, an integer, an operator, etc.).

**Parser:** The Parser class uses the Lexer to read the tokens and attempts to match the input against the grammar rules. It contains a series of methods (P(), E(), E\_prime(), T(), O()), each corresponding to a non-terminal in the grammar. These methods call each other recursively to match the production rules.

```
int main() {
    std::string input;
    while(true) {
        std::cout << "Enter an expression to parse: ";
        std::getline(std::cin, input);

        if(input == "exit") return false;

        Parser parser(input);

        try {
            parser.parse();
            std::cout << "Accept" << std::endl;
        } catch (const std::exception &e) {
            std::cerr << "Parsing error: " << e.what() << std::endl;
            return 1;
        }
    }

    return 0;
}
```

Here's a breakdown of the parser's operation:

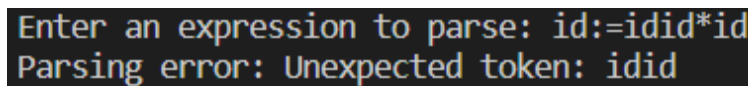
- The parse() method starts parsing by calling the P() method for the starting production  $P \rightarrow id:=E;$ .
- The P() method consumes tokens by matching an identifier (id), an assignment operator (:=), an expression by calling E(), and finally a semicolon (;).
- The E() method parses an expression by calling T() for the term and then E\_prime() for the rest of the expression (E').
- E\_prime() checks if the current token is a multiplication (\*) or division (/) symbol, and if so, it consumes the operator by calling O() and then parses another term and expression tail, effectively applying the production  $E' \rightarrow OTE'$ .

- `T()` matches a term which could be an id or an integer.
- `O()` matches an operator, either a star (\*) or slash (/).
- `consume(TokenType type)` is a utility method that checks if the current token matches the expected type. If it does, it consumes the token and gets the next one; if not, it throws an exception.
- The `main()` function sets up a loop where the user can enter expressions to parse. If the expression is "exit", the program stops. Otherwise, it creates a Parser with the input and calls the `parse()` method.
- If the parsing process is successful and reaches the end of the input without any mismatches or leftover input, the program prints "Accept". Otherwise, it prints an error message with details about the parsing error.

This code is a simplified example of a parser and does not cover all error handling or the full complexity you might find in a production-quality parser. However, it illustrates the basic principles of recursive descent parsing and how a parser interacts with a lexer to analyze the syntax of an input string.

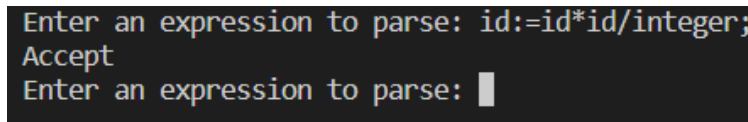
Keep in mind that the provided grammar must be LL(1) because recursive descent parsers require the grammar to be free of left recursion and to have no ambiguity, meaning that the next production rule to apply can always be determined by looking at the next token.

Here are two example string being used, one accepted string and one rejected string (not conforming with the grammar  $G'(P)$ ):



```
Enter an expression to parse: id:=idid*id
Parsing error: Unexpected token: idid
```

Figure 1: Rejected string, `id:=idid*id`



```
Enter an expression to parse: id:=id*id/integer;
Accept
Enter an expression to parse: █
```

Figure 2: Accepted string, `id:=id*id/integer;`

## 6.1 Pushdown Automaton (PDA)

Below is the PDA table for the accepted string `id:=id*id/integer;` as the input:

Stack	Input	Action
P\$	id:=id*id/integer;\$	$P \rightarrow id := E;$
id:=E;\$	id:=id*id/integer;\$	Match <i>id</i>
:=E;\$	:=id*id/integer;\$	Match <i>:=</i>
E;\$	id*id/integer;\$	$E \rightarrow TE'$
TE';\$	id*id/integer;\$	$T \rightarrow id$
idE';\$	id*id/integer;\$	Match <i>id</i>
E';\$	*id/integer;\$	$E' \rightarrow OTE'$
OTE';\$	*id/integer;\$	$O \rightarrow *$
*TE';\$	*id/integer;\$	Match <i>*</i>
TE';\$	id/integer;\$	$T \rightarrow id$
idE';\$	id/integer;\$	Match <i>id</i>
E';\$	/integer;\$	$E' \rightarrow OTE'$
OTE';\$	/integer;\$	$O \rightarrow /$
/TE';\$	/integer;\$	Match <i>/</i>
TE';\$	integer;\$	$T \rightarrow integer$
integerE';\$	integer;\$	Match <i>integer</i>
E';\$	;\$	$E' \rightarrow \epsilon$
;\$	;\$	Match <i>;</i>
\$	\$	Accept

The PDA refers to the parsing table when looking for the appropriate production to be used. Taking the first row as example the desired terminal is 'id', going back to the parsing table we can see that for non-terminal P the production used to obtain terminal 'id' is  $P \rightarrow id:=E;$ . Another example, take the third row from the bottom where the remaining input is ;\$ and the stack E';\$\$. Looking at the parsing table when provided with the non-terminal E' and the required terminal ';', we can see that the production used should be  $E' \rightarrow \epsilon$ .

## 7 Conclusion

This paper has presented a comprehensive exploration of the implementation of an LL(1) parser in C++ for the specific grammar G(P), which required meticulous grammar analysis. By eliminating left recursion and constructing the appropriate First and Follow sets, a solid foundation was established upon which the LL(1) parsing table could be constructed. The Predict sets derived from the intersection of First and Follow sets gave us the necessary guidance to construct a parsing algorithm that anticipates the correct production rule to apply based solely on the next input token.

The transition table, a crucial component of the LL(1) parser, was designed to reflect the relationships between non-terminals, terminals, and production rules. Its implementation in C++ not only showcased the practical application of theoretical computer science principles but also emphasized the importance of a well-structured approach to parsing in compiler design.



The journey from grammar transformation to actual implementation has underscored the significance of a sound theoretical foundation in creating practical tools for language processing. Future work could extend this implementation to support a wider range of grammars, handle error recovery more adequately, and perhaps integrate with a full-fledged compiler front-end, thus contributing further to the field of compiler construction and language design.

## References

- [1] Xiaoyuan Xie. "Lecture 3: Syntax Analysis." School of Computer Science, Wuhan University, 10 Oct. 2023. Lecture presentation.