Asian Institute of Technology

AT70.07

Programming Language and Compilers

Professor: Phan Minh Dung

Parser Analysis: Calculator Project

Nuon Puthisoptey (st124390)

An assignment submitted in partial fulfillment of the

requirements for AT70.07

March 2024

I. Introduction

The objective of this project is to create a calculator app capable of performing addition and multiplication operations on integer inputs. The addition operator takes precedence over multiplication in the operations. The application meets all the specified requirements for calculator functionality, ensuring that each input involving both operators yields three outputs: the result of the calculation, an equivalent expression in prefix notation, and an expression in postfix notation.

Furthermore, this report will delve into the grammar employed by the calculator, the types of parsers utilized, and the translation methodology.

II. Language Grammar

The grammar for the calculator language used in this project is as follows:

Rule 0:	$E' \rightarrow E$
Rule 1:	$E \rightarrow E * T$
Rule 2:	$E \rightarrow T$
Rule 3:	$T \rightarrow T + F$
Rule 4:	$T \rightarrow F$
Rule 5:	$F \rightarrow N$

Canonical LR(0) items

state 0:

0:	E' → . E
1:	E → . E * T
2:	E → . T
3:	T → . T + F
4:	T → . F
5:	F → . N

state 1:

0:	E' → E .
1:	E → E . * T

state 2:

2:	E → T .
3:	T → T . + F

state 3:

4:	T → F .

state 4:

5:	F → N .

state 5:

1:	E → E * . T
3:	T → . T + F
4:	T → . F
5:	F → . N

state 6:

3:	T → T + . F
5:	F → . N

state 7:

1:	E → E * T .
3:	T → T . + F

state 8:

3:	T → T + F .

III. Parser

The type of parser used for this project is a Bottom-up SLR(1) parser. In the syntax analysis stage, I have panned out three different types of parser to satisfy the output requirement.

Plainly and simply, I have separated the evaluation parser to perform normal calculation as CalcParser, prefix notation calculation as PrefixParser, and postfix notation calculation as PostfixParser. For prefix parsers, the calculation will return an output whereby the operators are in the front of the arithmetic expression or numbers. Hence, the postfix parsers will return an output that contains the operators after the numbers.

Implementing a bottom-up SLR(1) parser for the calculator project offers efficiency, simplicity, and powerful parsing capabilities. SLR(1) parsing provides linear time complexity, making it suitable for real-time parsing of user input. SLR(1) parsers handle a wide range of grammars, making them sufficient for the calculator's simple arithmetic expressions. Additionally, SLR(1) parsers include built-in error handling mechanisms, ensuring graceful handling of syntax errors in user input.

Parsing Table

Action					goto		
State	*	+	N	$	E	T	F
0			s4		1	2	3
1	s5			acc			
2	r2	s6		r2			
3	r4	r4		r4			
4	r5	r5		r5			
5			s4			7	3
6			s4				8
7	r1	s6		r1			
8	r3	r3		r3			

IV. Method of Translation

The compiler translation method for the parser that outputs three outputs (value evaluation, prefix notation, and postfix notation) involves incorporating semantic rules to ensure correct evaluation and notation generation. Here's a summary with additional detail on semantic rules:

a. Lexical Analysis (Tokenization): The input expression is tokenized into a sequence of tokens representing operands, operators, and other elements. Semantic rules may be applied during tokenization to handle special cases or validate the input format.

b. Parsing: The token sequence is parsed according to the defined grammar rules. In this case, the semantic rule specifies the precedence whereby '+' operator is to be prioritized before '*' operator.

c. Semantic Analysis (Evaluation): Once the parse tree is constructed, semantic rules are applied during tree traversal to evaluate the expression. These rules ensure that arithmetic operations are performed correctly and that any special cases, such as division by zero or overflow, are handled appropriately. For example, division by zero may result in an error message, while overflow may trigger a warning or require special handling.

   i. Prefix and Postfix Notation Generation: While parsing and evaluating the expression, semantic rules are applied to generate the prefix and postfix notations.

   ii. Rules for value evaluation

Production	**Semantic Rules**
$E \rightarrow E_1 * T$ $E \rightarrow T$ $T \rightarrow T_1 + F$ $T \rightarrow F$ $F \rightarrow N$	$E.val := E_1.val * T.val$ $E.val := T.val$ $T.val := T_1.val + F.val$ $T.val := F.val$ $F.val := N.lexval$

iii. Rules for prefix notation

Production	Semantic Rules
$E \rightarrow E_1 * T$	E.pf := ‘*’ || E_1.pf || T.pf
$E \rightarrow T$	E.pf := T.pf
$T \rightarrow T_1 + F$	T.pf := ‘+’ || T_1.pf || F.pf
$T \rightarrow F$	T.pf := F.pf
$F \rightarrow N$	F.pf := N.pf

iv. Rules for postfix notation

Production	Semantic Rules
$E \rightarrow E_1 * T$	E.pf := E_1.pf || T.pf || ‘*’
$E \rightarrow T$	E.pf := T.pf
$T \rightarrow T_1 + F$	T.pf := T_1.pf || F.pf || ‘+’
$T \rightarrow F$	T.pf := F.pf
$F \rightarrow N$	F.pf := N.pf

V. Conclusion

In conclusion, the calculator project employs a bottom-up LR(1) parser augmented with semantic actions to transform input expressions into their respective values, prefix notation, and postfix notation. The integration method for the parsers is not required as parsers are already separated into three. As a result, the calculator was able to satisfy all the conditions and perform the operation smoothly and accurately.