Jorge Ruesta, Siro Broton
100432547@alumnos.uc3m.es, 100496683@alumnos.uc3m.es

What lexical items does the rd_lex() function provide?

The function rd_lex() is a built-in function that provides us with everything that we need to write complex arithmetic expressions at the lexical level. That is, it provides us with a token to identify numbers, variables or operators and the value that they have taken. Almost every character that will be printed wll be provided by this function.

Grammar

We will now design a grammar that recognises prefix arithmetic expressions so that, in the future, we can transform them back with the parser. We will first design a grammar that does not contemplate variable use or assignment, so we can build upon it, and we can add all the remaining features. We need to make sure our grammar is always LL(1), because we need it to obtain the First and Follow values. For that we need to avoid ambiguity and left-recursion at all costs. Doing that we obtain the following:

LHS		RHS
S	⟶	oSS
S	⟶	(oSS)
S	⟶	n

Figure 1: First LL(1) formal grammar to read prefix expressions, done in JFLAP.

A brief explanation of the functioning of the grammar to understand it:

S ⇒ Our axiom non-terminal. Allows us to produce as many statements as we want with or without parentheses, even though prefix notation doesn't need them. Later, when we have to translate to other notation, we will add parentheses around every operation regardless of whether they had them or not to maintain the order of operations.
o ⇒ We will use the terminal o for operators (+, -, * and /).
n ⇒ The n terminal will be used to represent integer numbers.
The '(' and ')' characters, evidently, represent left and right parenthesis.

We will now build upon that grammar to allow the use of variables in expressions, the assignment of variables (chained or not) and mixed expressions.

LHS		RHS
S	→	oSS
S	→	(F
S	→	v
S	→	n
S	→	=vS
F	→	oSS)
F	→	=vS)

Figure 2: Complete LL(1) grammar that reads prefix notation.

We added a new non-terminal “F”, which we use to factorize the two possible productions that start with “(“, and a new terminal, “v”, which we use to represent variables. The “=” sign is also a new terminal that we added, but it is used to represent exactly what it is.

Something that we need to keep in mind when looking at this grammar is that we have represented “n”, “v” and “o” as terminals, when in reality they are not. They can both take multiple values, but we did not need to include that in our LL(1) grammar since the given function rd_lex() provides us with valid tokens for each number, variable and operator. That is why the objective of designing this grammar, and what we focused on, was figuring out how the prefix notation worked on a syntactical level, that is, checking that the tokens that rd_lex() provides would form grammatically correct phrases.

Parser Design

Our main objective with this practice is designing a parser that is able to read complex arithmetic expressions in prefix notation and simultaneously writing them in infix notation. We will divide our work into these two different tasks.

For the first task we will need to write a function for each non-terminal in our grammar that we can call whenever we need to. We need to design each function so it detects exactly where it is in the grammar and acts accordingly. For that we can use the First and Follow Sets that we obtained when we did the grammar in JFLAP.

```
C/C++
void ParseYourGrammar() { ParseExpression(); }

void ParseAxiom() {   /// Axiom ::= \n
  ParseYourGrammar(); /// Dummy Parser. Complete this with your design
  if (tokens.token == '\n') {
    printf("\n");
```

```
    MatchSymbol('\n');
  } else {
    rd_syntax_error(
        -1, tokens.token,
        "-- Unexpected Token (Expected:%d=None, Read:%d) at end of
Parsing\n");
  }
}
```

Figure 3: This code snippet shows the ParseAxiom and ParseYourGrammar functions.

The ParseAxiom function serves to call the ParseYourGrammar and detect if the expression is finished. The ParseYourGrammar function just calls another function, ParseExpression, which will be the first one in which we will use conditional functions.

```C/C++
// First = {operator, (, var, num, =}
void ParseExpression() {
  if (tokens.token == T_OPERATOR) {
    MatchSymbol(T_OPERATOR);
    ParseExpression();
    ParseExpression();
  } else if (tokens.token == '(') {
    MatchSymbol('(');
    ParseFactorization();
  } else if (tokens.token == T_VARIABLE) {
    MatchSymbol(T_VARIABLE);
  } else if (tokens.token == T_NUMBER) {
    MatchSymbol(T_NUMBER);
  } else {
    MatchSymbol('=');
    MatchSymbol(T_VARIABLE);
    ParseExpression();
  }
```

Figure 4: This code snippet shows the initial ParseExpression function.

In ParseExpression we first need to detect what type of expression we have. If our current token is an operator (rd_lex returned T_OPERATOR), we know we are in an S ⇒ oSS derivation. Then we match the operator symbol and call the ParseExpression function two additional times. If our current token is ‘(‘ we know we are in S ⇒ (F, so we just match the ‘(‘ symbol and call ParseFactorization (which corresponds to the F non-terminal in our grammar). If our token is a variable we immediately know we are in S ⇒ v, so we just need to
MatchSymbol. The same goes if our token is a number, the only difference being that we are in S ⇒ n. The only case that remains is if we are doing an assignment operation S ⇒ =vS, so we match both terminals and call the function ParseExpression again.

```C/C++
// First = {=, operator}
void ParseFactorization() {
  if (tokens.token == T_OPERATOR) {
    MatchSymbol(T_OPERATOR);
    ParseExpression();
    ParseExpression();
    MatchSymbol(')');
  } else {
    MatchSymbol('=');
    MatchSymbol(T_VARIABLE);
    ParseExpression();
    MatchSymbol(')');
  }
}
```

Figure 5: This code snippet shows our last function, ParseFactorization.

In ParseFactorization we need to distinguish if we are in F ⇒ oSS) or in F ⇒ =vS. We just need to check if the current token is an operator or not. If it is, we save it to op, match the first terminal (the operator) and then call the ParseExpression function twice. If it is not, we match the first terminal (the '=' sign) and the variable, and then call the ParseExpression function once.

We now have a program that can read and recognise only expressions in prefix notation, but doesn't print the corresponding infix expression. To accomplish that, we need to introduce print instructions for each terminal symbol in precise locations so that each prefix expression is transformed into our desired infix expression.

```C/C++
void ParseExpression() {
  if (tokens.token == T_OPERATOR) {
    printf("(");
    char op = tokens.token_val;
    MatchSymbol(T_OPERATOR);
    ParseExpression();
    printf("%c", op);
    ParseExpression();
    printf(")");
  } else if (tokens.token == '(') {
    printf("(");
    MatchSymbol('(');
    ParseFactorization();
  } else if (tokens.token == T_VARIABLE) {
    printf("%s", tokens.variable_name);
    MatchSymbol(T_VARIABLE);
```

```
  } else if (tokens.token == T_NUMBER) {
    printf("%d", tokens.number);
    MatchSymbol(T_NUMBER);
  } else {
    MatchSymbol('=');
    printf("(%s", tokens.variable_name);
    MatchSymbol(T_VARIABLE);
    printf("=");
    ParseExpression();
    printf(")");
  }
```

Figure 6: Final version of ParseExpression

As a general rule we will have every expression in between parentheses, even though they might not have been included in the prefix expression or are mostly redundant. This will be done to maintain the original order of operations. So at the beginning and the end of each of these cases S ⇒ oSS and S ⇒ =vS we add parentheses. Of course when S ⇒ (F there are parentheses, but this is because they are already in the prefix expression. Whenever we have S ⇒ oSS we print the operator in between the two calls to the ParseExpression function so that the sign ends in between the two expressions, i.e. Expression (+, -, * or /) Expression. Whenever S ⇒ v or S ⇒ n, we just print the current value of that terminal (stored in tokens.number for numbers and tokens.variable_name for variables). For the '=' we do a similar thing to what we did with the operator, we print it in between the variable (stored in tokens.variable_name) and the expression, i.e. Variable = Expression.

```C/C++
void ParseFactorization() {
  if (tokens.token == T_OPERATOR) {
    char op = tokens.token_val;
    MatchSymbol(T_OPERATOR);
    ParseExpression();
    printf("%c", op);
    ParseExpression();
    printf(")");
    MatchSymbol(')');
  } else {
    MatchSymbol('=');
    printf("(%s", tokens.variable_name);
    MatchSymbol(T_VARIABLE);
    printf("=");
    ParseExpression();
    printf(")");
    MatchSymbol(')');
  }
```

Figure 7: Final version of ParseFactorization

The print instructions added in ParseFactorization are in very similar positions to the ones in ParseExpression. This is because the two production rules of F are almost identical to the first and last production rules in S. So for F ⇒ oSS) we print the operator character in between the expressions and for F ⇒ =vS) we print the '=' sign after we print the variable, and before we call the ParseExpression function, exactly as we did in the previous function.

Syntactic diagram representing the designed grammar

To illustrate the process we have designed a transition graph for our grammar:

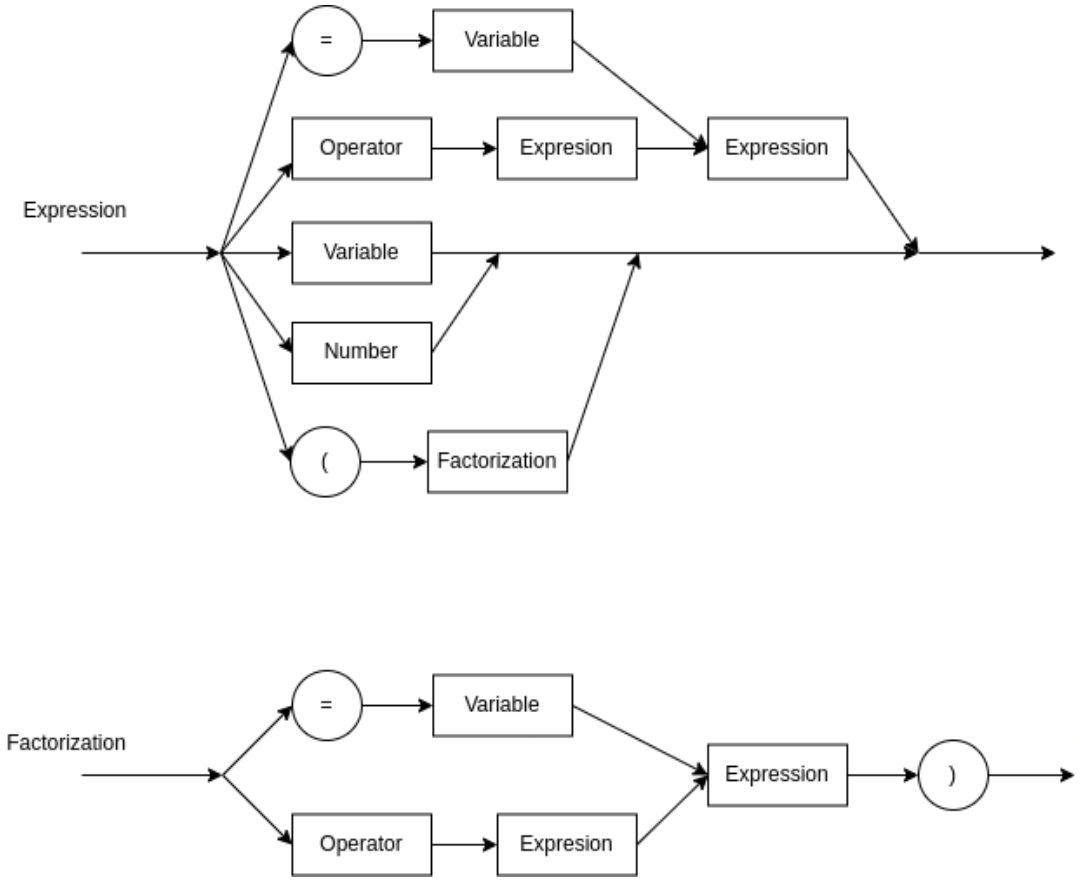

Figure 8: Syntactic diagram for the grammar.

Tests

We have designed a series of tests to make sure that our code and grammar really do what they are supposed to:

Test	Expected result	Result	Explanation
10	10	10	Number
i1	i1	i1	Variables
+ 1 2	(1+2)	(1+2)	Base case for operation
* + 1 2 3	(1+2)*3	((1+2)*3)	Multiple operations
= a1 1	a1=1	(a1=1)	Assignments
= b = a 1	b=a=1	(b=(a=1))	Chained assignments
=a+=b*2 3=c 1	a=(b=2*3)+(c=1)	(a=((b=(2*3))+(c=1))	Mixed expressions
*+a 2+b/3c1	(a+2)*(b+3/c1)	((a+2)*(b+(3/c1)))	Complex operations with operation hierarchy
+a12	a1+2	(a1+2)	Without spaces
a12	error	Unexpected Token (Expected:-1=None, Read:1001) at end of Parsing	Variable with incorrect format or missing operand
*(+1 2)3	(1+2)*3	((1+2)*3)	Good parentheses
*(+1 2 3)	error	ERROR in line 1 token 41 expected, but 1001 was read	Bad parentheses
+*d+*abc	error	ERROR in line 2 token 61 expected, but 10 was read	Not enough arguments
= 1 2	error	ERROR in line 1 token 1003 expected, but 1001 was read	Assignment of a number to another number