CS39003 Compilers Laboratory
Autumn 2025
Assignment 2
Date of posting: 11-Aug-2025

Predictive parsing under LL(1) grammar

Consider the grammar of string expressions with exponentiation and concatenation only (no range selection). This time, introduce parentheses for grouping (and altering the usual precedence rules). For example, `( ab ^ 2 . c ) ^ 3 . d ^ 4` evaluates to the string *ababcababcababcdddd*. The parentheses make the language of string expressions non-regular. Therefore you cannot parse an input by pattern matching alone. You need to use a parser to that effect. You will use lex anyway, but only to find the tokens. A **recursive descent parser** based on the **LL(1) parsing table** for this language should be used. Because this language is LL(1), your predictive parser will not require backtracking. The iterative procedure described in the class is good at parsing inputs. But since this should be associated with the actions involving string operations, the iterative version becomes problematic. A recursive implementation can naturally get around these problems.

The grammar

As in the earlier assignments, you read and parse each line of the input file one by one. We take a grammar that generates a single line, so LINE is the start symbol. Multiple invocations of the parser is necessary to read multiple lines.

```
LINE    ⟶    ID = EXPR
EXPR    ⟶    ARG REST
REST    ⟶    ε | . EXPR
ARG     ⟶    BASE XPNT
XPNT    ⟶    ε | ^ NUM
BASE    ⟶    STR | $ID | ( EXPR )
```

The tokens

In the above grammar, the terminal symbols (tokens) are highlighted in **red**. Lex will only identify these tokens, and will do no operations on strings. Each token is a pair (token_type, token_value). Upon the recognition of a token in the input, lex will **return** the token type. The value (if needed) can be obtained from `yytext`. A subsequent call of `yylex()` destroys `yytext`, so you need to keep a copy of a matched value before calling `yylex()` for getting the next token. For punctuation symbols and operators, the value is not important, so `yytext` can be ignored for these tokens.

The complete list of tokens to be identified by lex is given below. Define the return types as suitable non-zero integers.

Token	Return type	More info
Variable name	ID	Copy `yytext` for later use as l-value.
Non-negative integer	NUM	Used in an exponent. Should be stored as a string or as an integer.
Literal string	STR	An alphabetic string.
Variable reference (`$ID`)	REF	Symbol-table lookup needed to get the string.
(	LPN	These tokens do not have values, so the storage of `yytext` for these tokens is not needed.
)	RPN	
^	CRT	
.	DOT	
=	ASG	
\n	EOL	

In order to handle valid strings like `abc = abc . $abc`, you need to use lex states. At the beginning of each line, lex should be in the INITIAL state, and will switch to a new state RVALUE after encountering the assignment operator. In the INITIAL state, `abc` should be interpreted as an ID. In the state RVALUE, `abc` should be interpreted as a literal string (STR), whereas `$abc` is to be interpreted as a reference (REF). Do not use the other states as in Assignment 1 (because it is impossible for lex to carry out the parsing job in the current assignment). Using the state RVALUE is still useful, because the same pattern may mean two different token types on the two sides of the assignment operator.

LL(1) Parsing Table

For your convenience, the table is derived below. Use the table blindly for making decisions about non-terminal expansions. In the parsing table, the row header is the head of the production, so we write only the body of the production as table entries. Also, the columns under the input symbols = and NUM are entirely empty, so these columns are not shown in the table below.

The LL(1) parser (recursive or iterative) requires an end-of-input marker. In our theory classes, we have used \$ for that purpose. Here, we use \$ in tokens of type REF. Every line in a text file (including the last) ends with the new line character `\n`. So this symbol can work as the natural end-of-input marker. Recall that you parse each line individually (not the entire input as a whole).

FIRST(LINE)	= { ID }	
FIRST(EXPR)	=	
FIRST(ARG)	=	
FIRST(BASE)	= { STR , \$, (}	
FIRST(REST)	= { . , ε }	
FIRST(XPNT)	= { ^ , ε }	
FOLLOW(LINE)	= { \n }	
FOLLOW(EXPR)	=	
FOLLOW(REST)	= {) } U FOLLOW(LINE)	= {) , \n }
FOLLOW(ARG)	= (FIRST(REST) – { ε }) U FOLLOW(EXPR)	= { . ,) , \n }
FOLLOW(XPNT)	=	
FOLLOW(ARG)		= { . ,) , \n }
FOLLOW(BASE)	= (FIRST(XPNT) – { ε }) U FOLLOW(ARG)	= { ^ , . ,) , \n }

	ID	STR	\$ID	(	)	^	.	\n
LINE	ID = EXPR							
EXPR		ARG REST	ARG REST	ARG REST				
REST					ε		. EXPR	ε
ARG		BASE XPNT	BASE XPNT	BASE XPNT				
XPNT					ε	^ NUM	ε	ε
BASE		STR	\$ID	(EXPR)				

The parsing algorithm

Write (mutually) recursive functions for the non-terminal symbols. At the beginning of each line from the input, call `LINE()`. You also maintain a global variable to store the next token waiting to be consumed from the input (can be EOL among others). If you are in a function for the non-terminal N, look at the next token. If the (N, next_token)-th entry in the above parsing table is empty, then LL(1) parsing fails. Report this error, and exit. Otherwise, pick the applicable production from that table entry. If that production is $N \rightarrow X_1X_2\ldots X_k$, take actions for X_1, X_2, …, X_k in that sequence. If X_i is a terminal, check whether the same symbol waits as the next token. If so, consume it from the input (call `yylex()` so that another input token is read from the input). Otherwise, report error, and exit. If X_i is a non-terminal, call the function for X_i. Parsing completes when the outermost call of `LINE()` finishes successfully (no other function calls `LINE()`, so this was the only call of this function for this input line).

As an example, suppose that you are in the `BASE()`. If the next token is neither STR nor \$ID nor (, then parsing fails. Let the next token at this point be (. Then the applicable production is BASE → (EXPR). Consume (from the input. Then call `EXPR()`. When this call returns,) must be the next token. If not, report error, and exit. Otherwise, consume) from the input, and the current call of `BASE()` returns. As another example, suppose that the function `EXPR()` is called, and the next token is \$ID. In this case, the applicable production is EXPR → ARG REST. So call `ARG()` and `REST()`, and then return.

So far, only the parsing steps are explained. For performing the string operations (concatenation and exponentiation), the function for each non-terminal should return a string. Only `LINE()` need not return anything; it should set an entry in the symbol table, print a message to that effect, and return. The return value of each of the other non-terminal functions should be the partial result (a string) computed beneath that non-terminal in the parse tree. For example, consider a call of `EXPR()` that uses the production EXPR → `ARG REST`, invokes `ARG()` and `REST()`, concatenates their return values, and return that concatenated string. Likewise, for the production ARG → BASE XPNT, the string returned by `BASE()` should be concatenated with itself as many times as XPNT suggests by its return value. The resulting string is returned by the call of `ARG()`. For the production `BASE` → `$ID` in a call of `BASE()`, a symbol-table lookup is needed.

Make the convention that if ε is the body of a production used during the parsing, then the NULL string is returned. For instance, `REST()` and `XPNT()` should return NULL, when the next token is) .

Handling errors

You do not have to worry about all types of errors. Detect only the following two types of errors.

- Lex finds an invalid character.
- At parse time, an empty entry in the parsing table is accessed.

There is no need to correct any error. Ignore lex errors (with a warning message), whereas exit on a parsing error.

In addition, you should handle invalid symbol-table references (undefined variables) by returning the empty (not NULL) string.

What to submit

Submit in an archive (zip, tar, or tgz) the following three files.

- Your lex file. The user code section should only contain the default implementation of `yywrap()` (and nothing else).
- A separate program (a C/C++ file) that implements LL(1) parsing recursively.
- A makefile with all, run, and clean targets.

Submit a single file, not the individual files. Do not include other files (like `lex.yy.c` or `a.out`) in the archive.

Sample Output

```
$ cat input.txt
S1 = a

S2 = b^2 . c

S3 = ( $S1 . (b . c^4) ^2 ) ^ 3

S4 = (($S1 . d) ^ 3 . $S2 . d^2)^4 . (EF^2 . GH) ^ 5
$ make run
lex tokens.l
gcc LLparser.c
./a.out input.txt
+++ Storing S1 = a
+++ Storing S2 = bbc
+++ Storing S3 = abccccbccccabccccbccccabccccbcccc
+++ Storing S4 = adadadbbcddadadadbbcddadadadbbcddadadadbbcddEFEFGHEFEFGHEFEFGHEFEFGHEFEFGH
$ make clean
rm -f a.out lex.yy.c
$
```