CS39003 Compilers Laboratory
Autumn 2026
Assignment 6
Date of posting: 01-Sep-2026

Using markers of yacc

This assignment deals with unsigned integers and floating-point numbers in arbitrary bases B. For $B \leqslant 10$, use the decimal digits $0, 1, 2, \ldots$. For $B > 10$, first use the decimal digits $0, 1, 2, \ldots, 9$, and then start using the lower-case letters $a, b, c, \ldots$ (standing for the B-ary digits $10, 11, 12, \ldots$). Since there are only 26 letters in the Roman alphabet, we should have $B \leqslant 36$. The base B and a number N in the base is specified either as $B:N$ or as $`N:B$. Here, N is either an integer I or a floating-point number of the form $I.F$ (exponent/scientific notations are not to be considered). Some examples are given below. In these examples, the base B (B is always specified in decimal) is 20, so the allowed digits are $0, 1, 2, \ldots, 9, a, b, c, \ldots, i, j$. The quote is necessary (otherwise how can you decide the base in 15:20?).

```
20:efgh
20:1234.abcd
`efgh:20
`1234.abcd:20
```

The two formats require separate treatments.

The *B*:*N* format

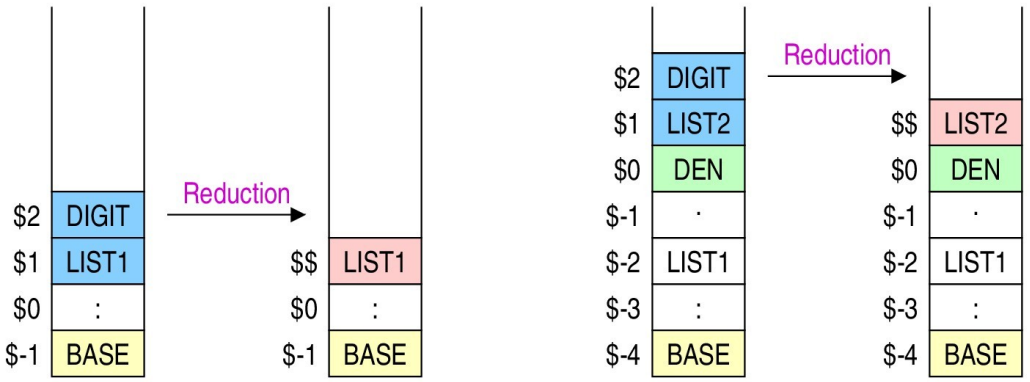
This is the easier format. An integral value N is

$$\begin{aligned} & d_r\, d_{r-1}\, d_{r-2} \ldots d_1\, d_0 \\ = \; & d_r \times B^r + d_{r-1} \times B^{r-1} + d_{r-2} \times B^{r-2} + \ldots + d_1 \times B + d_0 \\ = \; & \big(\ldots \big(((d_r \times B + d_{r-1}) \times B + d_{r-2}) \times B + \ldots \big) \times B + d_1 \big) \times B + d_0 \,. \end{aligned}$$

A fractional part $.e_1 e_2 \ldots e_s$ is the integer $e_1 e_2 \ldots e_s$ divided by the integer B^s. The numerator $e_1 e_2 \ldots e_s$ is evaluated like the integral part (as explained above). The final division by B^s should be a floating-point division. Horner's rule is appropriate to use in the following left-recursive grammar.

```
NUM      →   BASE : FORMAT1
FORMAT1  →   LIST1 | LIST1 . LIST2
LIST1    →   LIST1 DIGIT | ε
LIST2    →   LIST2 DIGIT | ε
```

Since the base B is available at the beginning, it resides in the parse stack until the final reduction of NUM takes place. The post-point part (if present) uses B as well, but should also compute the denominator B^s. Use a marker to store this (not shown in the grammar above; add it at an appropriate place). The first reduction LIST2 → ε initializes the marker to 1. Subsequently, each reduction LIST2 → LIST2 DIGIT multiplies the marker by the base. The left-recursive grammar used here never lets the parse stack grow beyond a small constant size, and both the base and the marker can be located at fixed constant depths from the top of the stack. This is illustrated in the following figure. The handle is shown in cells shaded blue, and the head of the production used for reduction is shown in a cell shaded red. The extra marker cell added for storing the denominator (this is external to the current handles) is shaded green. The BASE B too is stored in a cell (shaded yellow), but this is an input token (not a marker).



The `*N*:*B* format

This format offers some difficulty. Since B is available at the end, computations using B cannot proceed as above. We need to store the digits in a list, and eventually when B is read, that list will be processed. We disallow you to use any external list as global variables or stack data type. The parse stack itself can hold all the symbols read from the input, and start processing them after B is available. This necessitates using a right-recursive grammar that will process the digits from right to left. Horner's rule is not readily applicable in this context. We instead use the following formula.

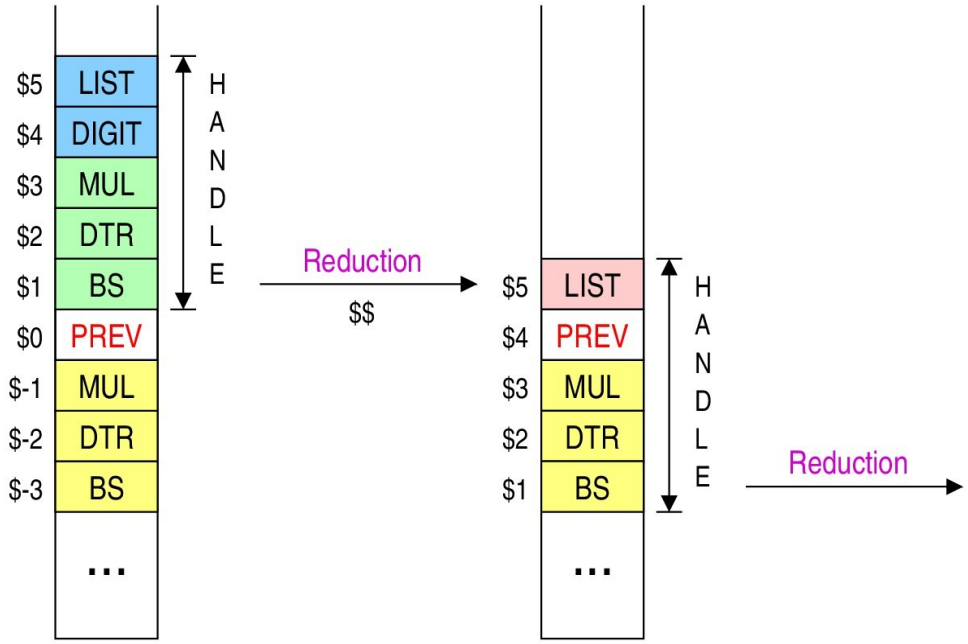
$$
\begin{aligned}
& d_r\, d_{r-1}\, d_{r-2} \ldots d_1\, d_0 \,.\, e_1\, e_2 \ldots e_s \\
= \;& d_r \times B^r + d_{r-1} \times B^{r-1} + d_{r-2} \times B^{r-2} + \ldots + d_1 \times B + d_0 + e_1 \times B^{-1} + e_2 \times B^{-2} + \ldots + e_s \times B^{-s} \\
= \;& \left(d_r \times B^{r+s} + d_{r-1} \times B^{r+s-1} + d_{r-2} \times B^{r+s-2} + \ldots + d_1 \times B^{s+1} + d_0 \times B^s + e_1 \times B^{s-1} + e_2 \times B^{s-2} + \ldots + e_s\right) / B^s
\end{aligned}
$$

The calculation of each term requires a digit, the base, and a power of the base. Moreover, from the fractional part (if present), the final denominator B^s should be computed. You are **not** allowed to store these values in global variables. Instead you store these values in the parse stack itself in the form of markers. Before every digit (or the .), these values must be present at *constant* depths from the current top. When a digit is processed, a term and $$ (the numerator) are computed, and the markers just below the previous input token (digit or . or the beginning) should be updated.

The grammar for this case is given below. It suffices to have three markers: the base BS (the name BASE is used as another grammar symbol), MUL (the multiplier B^i for the current term of the numerator), and DTR (the denominator; do not use the same name DEN given to a marker in the other format). In order to distinguish printing integers from printing floating-point numbers, the denominator may be initialized to 0. As soon as the B-ary point is found, it is changed to 1. In all other cases, it is copied *from* the previous denominator at the time of the reduction of DTR. At the time of the reduction of LIST3 and LIST4, it is copied *to* the previous denominator. Add the markers at appropriate places in the following grammar.

```
NUM      →   FORMAT2
FORMAT2  →   LIST3
LIST3    →   DIGIT LIST3  |  : BASE  |  . LIST4
LIST4    →   DIGIT LIST4  |  : BASE
```

When : BASE is found at the end, the topmost set of markers in the parse stack are initialized (except DTR). Each subsequent reduction should update the previous set of markers just below the current handle. One reduction step (the processing of a digit) is illustrated in the figure below. The current set of markers (shown in boxes shaded green; these are a part of the current handle), the current DIGIT, and the current value of LIST (may be LIST3 or LIST4) are used to compute $$ (the numerator). Moreover, the previous set of markers (shown in boxes shaded yellow) are updated for the next reduction that will process the previous input token PREV. At the beginning (of LIST3), there is no input token. Use a dummy marker (in addition to the above three markers) there. The number is printed during the reduction NUM → FORMAT2. This printing needs DTR. In order to make the processing of each token in N uniform, the reduction of the first digit (or the B-ary dot if there is no digit before the dot) should update the bottommost set of markers.



Lexical analysis

Use lex to generate the tokens. In this assignment, we have only single-character tokens (`0 – 9`, `a – z`, `.`, `:`, and `` ` ``). Lex must **not** return an entire number $B:N$ or $`N:B$. Of course, these can be defined by regular expressions, and you can post-process the entire string to do the relevant calculations. But that is not the theme of this assignment. You **must** use yacc and its markers; these are the only relevant components of this exercise. So yacc must receive tokens character by character. As usual, you need to ignore all white spaces and empty lines. Only one number appears in each non-empty line. That is, the new-line character `\n` acts as the separator between two consecutive numbers. It is therefore allowed to have spaces inside a single number (like `` ` 1 234 . 567 : 8 ``).

Parsing

The use of yacc is here subject to several restrictions. Since this problem can be solved in various other ways, you must strictly follow the guidelines given below.

- You are forbidden to use even a single global variable. Of course, lex will introduce global variables like `yytext` or `yyin`. Besides these, you must not yourself declare and/or use any other global variable. Carry out all the computations in yacc's actions themselves. Do **not** declare or use any additional functions.

- Use the default parse stack type for yacc (that is, `int`) throughout. This means that you work only with integer arithmetic. Use of floating-point functions like `pow()` is strictly forbidden.

- It is in general allowed to use `struct` pointers as yacc's stack-element types. A `struct` may embed all the markers in the second format inside `$$` itself. You are not allowed to do this in this assignment (because you use only the default stack-element type, and you are supposed to use markers). In particular, your yacc program must **not** have any `%union` directive.

- No external list is allowed. Only the parse stack is to be used as an ordered storage of input tokens.

The start symbol PROG will derive a set of NUM's separated by the new-line character `\n`. For each non-empty line (a number), use the following production.

NUM → BASE : FORMAT1 | FORMAT2

Then use the grammar given earlier (insert appropriate marker symbols at appropriate places inside the production bodies). The final reduction of NUM should print the calculated number (in decimal) in the format demonstrated by the sample output on the next page. During these printings, use floating-point divisions only when needed. For printing floating-point numbers in FORMAT1, the two lists LIST1 and LIST2 are evaluated separately, and the final value is computed as VAL1 + (VAL2 / DEN). On the other hand, the processing of the input in FORMAT2 proceeds differently. If LIST4 is never used (for integers), then DTR should hold 0, and a division is not necessary in this case. If LIST4 is used (because of the presence of the B-ary dot in the input), the final value is computed as LIST3 / DTR. At that time, a floating-point division is needed to compute the value of the input number.

Submission format

Write the `main()` function in the yacc file itself. Submit (in a tar/tgz/zip archive) your lex file, your yacc file, and a makefile with run, compile (on `input.txt`), and clean targets.

Sample Output

Input	Output
20:efgh 20:1234.abcd `efgh:20 `1234.abcd:20 15 : 20 ` 15 : 20  2 : 100 1101 0010 . 1001 0001 0110 0001 01 ` 11 1101 1011 . 1010 0111 1000 0001 1001 : 2 16:16 16:16. `16:16 `16.:16 8:. `.:8  8: `:8 `zzz:36	118337 8864 + 84653 / 160000 = 8864.529081 118337 1418324653 / 160000 = 8864.529081 30 25 1234 + 148869 / 262144 = 1234.567890 1035630617 / 1048576 = 987.654321 22 22 + 0 / 1 = 22.000000 22 22 / 1 = 22.000000 0 + 0 / 1 = 0.000000 0 / 1 = 0.000000 0 0 46655