CS305 – Programming Languages

Fall 2025-2026

HOMEWORK 2

Implementing a Syntax Analyzer (Parser) for Calculator Scientifique (CS)

Due date: November 2nd, 2025 @ 23:55

NOTE

Only SUCourse submissions are allowed. Submissions by e-mail are not allowed. Please see the note at the end of this document for the late submission policy.

1 Introduction

In this homework, you will write a context-free grammar and implement a parser for the CS language, which you designed as a scanner in the previous homework. The requirements of the homework and the grammar that generate the language are explained below.

2 Calculator Scientifique (CS) Language

The grammar you will design needs to generate the CS language described below. Here is an example program written in this language to give you an idea of what a CS program looks like.

```
1   f(x) = 3x;
2   g()=5  ;
3   h(x) = x + 2
4
5   ;
6   j(x,y) = (3x^2 - 2) (y^3 + 5);
7   k(x) = x^9 - x^3;
8
9   calculate f(x) + f(3);
10  calculate g() * f(y);
11  calculate h(x) j(a,b);
12  calculate k(x) + D(k,x,3);
13  calculate I(f,y,1);
14
```

Here is the description of the syntactic rules of the CS language:

1. A CS program consists of a definition block, followed by a calculation block.
2. Inside a definition block, we have any number of (including 0) function definitions.
3. A function definition consists of a function name (which is a single lowercase letter), followed by a pair of parentheses (i.e., "(" and ")"), followed by an equality sign, followed by an "ordinary–expression" and finally followed by a semicolon. Between the pair of parentheses, there is an optional comma–separated list of variables, where each variable is again a single lowercase letter. An example function definition is given below:

```
f(x,y) = 3x^2 - 5xy - 4y;
```

4. Inside a calculation block, we have any number of (including 0) calculation statements.
5. A calculation statement starts with the `calculate` keyword followed by an "extended–expression", and terminated by a semicolon.
6. An "ordinary–expression" is one of the followings:
   - an integer,
   - a variable,
   - an integer followed by an exponentiation operator followed by an integer,
   - a variable followed by an exponentiation operator followed by an integer,
   - an ordinary–expression enclosed by a pair of parentheses,
   - two ordinary–expressions separated by an addition symbol,
   - two ordinary–expressions separated by a subtraction symbol,
   - two ordinary–expressions separated by a multiplication symbol,
   - two ordinary–expressions (which represent implicit multiplication),
   - **the multiplication symbol has a higher priority level than the addition and the subtraction symbol,**
   - **the exponentiation operator has a higher priority level than the multiplication, the addition, and the subtraction symbol.**
7. An "extended–expression" on the other hand is one of the followings:
   - an integer,
   - a variable,
   - an integer followed by an exponentiation operator followed by an integer,
   - a variable followed by an exponentiation operator followed by an integer,

- an extended–expression enclosed by a pair of parentheses,
- two extended–expressions separated by an addition symbol,
- two extended–expressions separated by a subtraction symbol,
- two extended–expressions separated by a multiplication symbol,
- two extended–expressions (which represent implicit multiplication),
- **within a pair of parenthesis, a comma–separated–list of items (having 0,1, or more elements) where each item is an extended–expression,**
- an integration expression,
- a derivation expression,
- a function expression,
- **the multiplication symbol has a higher priority level than the addition and the subtraction symbol,**
- **the exponentiation operator has a higher priority level than the multiplication, the addition, and the subtraction symbol.**

8. An integration expression starts with the integration keyword "I", followed by a pair of parentheses. Between the parentheses, we have a function name (a single lowercase letter), a variable (a single lowercase letter), and an integer, which are all separated by a comma. An example integration expression is `I(f,x,2)`. Even though it is not important for the purposes of this homework, it means the 2nd integration of the function `f` with respect to `x`.

9. A derivation expression starts with the derivation keyword "D", followed by a pair of parentheses. Between the parentheses, we have a function name (a single lowercase letter), a variable (a single lowercase letter), and an integer, which are all separated by a comma. An example derivation expression is `D(f,x,2)`. Even though it is not important for the purposes of this homework, it means the 2nd derivative of the function `f` with respect to `x`.

10. A function expression starts with a function name (a single lowercase letter), followed by a pair of parentheses. Between the parentheses, we have an optional comma–separated list of extended–expressions. **Please do not confuse comma–separated extended–expressions and implicit multiplications, thus the following is a valid calculation statement:**

```
calculate f(x y*z,t) 3x^2 - 5yz  4t;
```

3 Terminal Symbols

Although you can implement your own scanner, we provide a flex scanner for this homework. The provided flex scanner implements the following tokens.

`tPLUS`: The scanner returns this token when it sees a `+` operator.

`tMINUS`: The scanner returns this token when it sees a `-` operator.

`tMUL`: The scanner returns this token when it sees a `*` operator.

`tEXP`: The scanner returns this token when it sees an `^` operator.

`tLPR`: The scanner returns this token when it sees a `(` symbol.

`tRPR`: The scanner returns this token when it sees a `)` symbol.

`tASSIGN`: The scanner returns this token when it sees an `=` symbol.

`tCOMMA`: The scanner returns this token when it sees a `,` symbol.

`tSEMICOLON`: The scanner returns this token when it sees a `;` symbol.

`tCALCULATE`: The scanner returns this token when it sees a `calculate` keyword.

`tDERIVATION`: The scanner returns this token when it sees a `D` keyword.

`tINTEGRATION`: The scanner returns this token when it sees an `I` keyword.

`tIDENTIFIER`: The scanner returns this token when it sees an `identifier`.

`tINTEGER`: The scanner returns this token when it sees an `integer`.

Besides these tokens, the scanner silently consumes white space characters. Any other character that is not recognized as part of a lexeme of a token is returned to the parser. These tokens and their lexemes are explained in detail in the Homework 1 document.

4 Output

Your parser must print out `OK` and produce a new line if the input is grammatically correct. Otherwise, your parser must print out `ERROR` and produce a new line. In other words, the main part in your parser file must be as follows (and there should be no other part in your parser that produces an output):

```
int main ()
{
   if (yyparse())
   {
      // parse error
      printf("ERROR\n");
      return 1;
   }
   else
   {
      // successful parsing
      printf("OK\n");
      return 0;
```

```
    }
}
```

In short, if the file `example1.cs` includes a grammatically correct CS program, then your output should be `OK`, and otherwise, your output should be `ERROR`.

5 How to Submit

You need to submit:

1. your Bison file
2. your flex file (**IMPORTANT**: Even if you use the flex file we provided, you still need to submit it after **renaming** it as indicated below.)

Please submit both of these files (**without zipping them**) on SUCourse. The name of your bison file must be **username-hw2.y**, and the name of your flex file must be **username-hw2.flx**, where **username** is your SU-Net username.

We will compile your files by using the following commands:

```
flex username-hw2.flx
bison -d username-hw2.y
gcc -o username-hw2 lex.yy.c username-hw2.tab.c -lfl
```

So, ensure these three commands are enough to produce the executable parser. If we assume that there is a text file named example1.cs, we will try out your parser by using the following command line:

```
username-hw2 < example1.cs
```

6 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-20 points otherwise]
- **Important**: Make sure that your parser **must** work in an **unambiguous** way. In other words, you shall not get any ambiguity warnings once it is compiled. [-20 points otherwise]

- **Important**: Make sure you include the correct "`...tab.h`" file in your scanner, and make sure you can compile your parser using the commands given in Section 5. If we are not able to compile your code with those commands **your grade will be zero for this homework.**
- **Important**: Since this homework is evaluated automatically, ensure your output is exactly as it should be. (i.e., OK for grammatically correct CS programs and ERROR otherwise).
- No homework will be accepted if it is not submitted using SUCourse.
- You may get help from our TAs, LA, or friends. However, **you must write your bison file by yourself**.
- Start working on the homework immediately.
- If you develop your code or create your test files on your own computer (not on cs305.sabanciuniv.edu), there can be incompatibilities once you transfer them to the cs305 machine. Since the grading will be done automatically on the cs305 machine, we strongly encourage you to do your development on the cs305 machine, or at least test your code on the cs305 machine before submitting it. If you prefer not to test your implementation on the cs305 machine, this means you accept to take the risks of incompatibilities. Even if you have spent hours on homework, you can easily get 0 due to such incompatibilities.

LATE SUBMISSION POLICY

Late submission is allowed subject to the following conditions:

- Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
- If you submit on time (i.e., before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 minutes of delay.
- We will not accept any homework later than 500 minutes after the deadline.
- SUCourse's timestamp will be used for STF computation.
- If you submit multiple times, the last submitted version and its submission time will be used.