CS305 – Programming Languages
Fall 2023-2024

HOMEWORK 2

Implementing a Syntax Analyzer (Parser) for MailScript

Due date: November 03, 2023 @ 23:55

NOTE

Only SUCourse submission is allowed. No submission by e-mail. Please see the note at the end of this document for late submission policy.

1 Introduction

In this homework you will write a context free grammar and implement a simple parser for MailScript language for which you designed a scanner in the previous homework. The language that will be generated by your grammar and other requirements of the homework are explained below.

2 MailScript Language

The grammar you will design needs to generate the MailScript language as described below. Here is an example program written in this language. This is to give you an idea how a MailScript program looks like.

```
set Message ("Welcome to CS305.")

Mail from nermin@mail.com:

    schedule @ [27/10/2021, 07:48]:
        send [Message] to [("Kerem", kerem@mail.com.tr)]
    end schedule

    set Message ("Our project is due tomorrow.")

    send [Message] to [("Mine", mine@mail.com),
    (selim@mail.com)]
```

```
end Mail

Mail from nermin@sabanciuniv.edu:

end Mail

Mail from serkan@sabanciuniv.edu:

    set UserName ("There is a meeting at 3pm.")

    schedule @ [02/10/2021, 16:00]:

        send [MorningMessage] to [(ali@mail.com),
        ("John Doe", emre@mail.com),
        (UserName, ali@mail.com)]

        send ["How are you?"] to [("Omer", omer@mail.com)]

    end schedule

end Mail
```

Note that the set of statements given above does not have to make sense. It may even have errors as a MailScript program. For example: There can be a usage of an undeclared variable or the recipient list can contain the same e-mail more than once.

We have some examples of such errors in the example MailScript program given above. However, in this homework we will not aim to detect such errors. In this homework, we will aim to implement a parser for the basic syntax rules of the MailScript language that are given in detail below.

1. A MailScript program consists of a sequence of components, where each component is either a mail block or a set statement. Any number of mail blocks and set statements can be given in any order in a MailScript program. An empty program is also considered as a MailScript program.

2. A mail block starts with the "Mail" and the following "from" keyword, followed by the e-mail address that the user wants to send an e-mail from, followed by a colon symbol. Each mail block ends with the keyword "end Mail". Note that the space inside "end Mail" can be more than one but cannot be zero. For further information, you can check the relevant part in the Homework 1 document. Inside of a mail block can be empty or contain a statement list. A statement list is defined as one or more statements given one after another.

3. Each statement in the MailScript language can be a "set", "send" or "schedule" statement.

   (i) A set statement is used to assign a variable a value. It starts with the keyword "set" followed by the name of the variable, followed by a left parenthesis, followed by a string, followed by a right parenthesis. Please note that a set statement can exist both in and out of mail blocks.

   ```
   set Notification ("Don't forget to set the alarm.")
   Mail from melis@sumail.com:
       set Content ("Meeting at 5pm.")
   end Mail
   set Notification ("Set the alarm for 9am.")
   ```

   (ii) A send statement is used to send an e-mail. It starts with the keyword "send", followed by a left bracket, followed by either a variable name or a string, followed by a right bracket, followed by the keyword "to", followed by a recipient list. Some examples are given below:

   ```
   set Message ("How are you?")
   send ["Good morning!"] to [(user1@mail.com)]
   send [Message] to [(user2@mail.com)]
   ```

   (iii) A schedule statement is used to schedule a time for one or more e-mails to be sent. It also is formed like a block. Each schedule block starts with the keyword "schedule", followed by the @ symbol, followed by a left bracket, followed by a date and a time object separated by a comma, followed by a right bracket, and finally followed by a colon symbol. Each schedule block ends with the keyword "end Schedule". Note that the space inside "end Schedule" can be more than one but cannot be zero. For further information, you can check the relevant part in the Homework 1 document. Inside of a schedule block, there can ONLY be a list of one or more send statements, and it cannot be empty. An example is given below:

   ```
   schedule @ [18/10/2021, 12:00]:
       send ["HW is assigned!"] to [(example@mail.com)]
       send ["Please check the gradebook."] to
       [(example@mail.com), (admin123@mail.com)]
   end schedule
   ```

4. Please note that the schedule and send statements can only appear in mail blocks. On the other hand, a set statement can exist in or out of a mail block.

5. A recipient object can be given in two forms. The first form starts with a left parenthesis, followed by an e-mail address, followed by a right parenthesis. The second form starts with a left parenthesis, followed by either a variable name or a string, followed by a comma, followed by an e-mail address, followed by a right parenthesis. Some examples are given below:

```
(user_475@some-company.co.uk)
("Mehmet", user_475@some-company.co.uk)
```

6. A recipient list starts with a left bracket, followed by a comma separated list of recipient objects (or a single recipient object), followed by a right bracket. It cannot be empty. An example is given below:

```
[("Ayse", ayse3@thismail.com), ("Unknown User", user8564@mail.com),
(seyma@example.de)]
```

3 Terminal Symbols

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

`tMAIL`: The scanner returns this token when it sees `Mail` in the input.

`tENDMAIL`: The scanner returns this token when it sees `end Mail` in the input.

`tSCHEDULE`: The scanner returns this token when it sees `schedule` in the input.

`tENDSCHEDULE`: The scanner returns this token when it sees `end schedule` in the input.

`tSEND`: The scanner returns this token when it sees `send` in the input.

`tSET`: The scanner returns this token when it sees `set` in the input.

`tTO`: The scanner returns this token when it sees `to` in the input.

`tFROM`: The scanner returns this token when it sees `from` in the input.

`tAT`: The scanner returns this token when it sees an `@` in the input.

`tCOMMA`: The scanner returns this token when it sees a comma in the input.

`tCOLON`: The scanner returns this token when it sees a `:` in the input.

`tLPR`: The scanner returns this token when it sees a `(` in the input.

`tRPR`: The scanner returns this token when it sees a `)` in the input.

`tLBR`: The scanner returns this token when it sees `[` in the input.

`tRBR`: The scanner returns this token when it sees `]` in the input.

`tIDENT`: The scanner returns this token when it sees an identifier in the input.

`tSTRING`: The scanner returns this token when it sees a string in the input.

`tADDRESS`: The scanner returns this token when it sees an e-mail address in the input.

`tDATE`: The scanner returns this token when it sees a date in the input.

`tTIME`: The scanner returns this token when it sees a time in the input.

Besides these tokens, it silently consumes any white space character. Any other character which is not recognized as a lexeme of the tokens is returned to the parser.

These tokens and their lexemes are explained 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;
   }
}
```

5 How to Submit

Submit your Bison file named as `username-hw2.y`, and flex file named as `username-hw2.flx` where `username` is your SU-Net username and **do not zip your files.** 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, make sure that these three commands are enough to produce the executable parser. If we assume that there is a text file named test17.ms, we will try out your parser by using the following command line:

```
username-hw2 < test17.ms
```

If the file `test17` includes a grammatically correct MailScript program then your output should be `OK`, and otherwise, your output should be `ERROR`.

6 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points otherwise]
- **Important**: Make sure you include the right file in your scanner and make sure you can compile your parser using the commands given in the 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 make sure your output is exactly as it is supposed to be. (i.e. OK for grammatically correct and ERROR otherwise).
- No homework will be accepted if it is not submitted using SUCourse.
- You may get help from our TA or from your 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 flow.sabanciuniv.edu), there can be incompatibilities once you transfer them to flow.sabanciuniv.edu. Since the grading will be done automatically on the flow.sabanciuniv.edu, we strongly encourage you to do your development on flow.sabanciuniv.edu, or at least test your code on flow.sabanciuniv.edu before submitting it. If you prefer not to test your implementation on flow.sabanciuniv.edu, this means you accept to take the risks of incompatibility. Even if you may have spent hours on the 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 mins of delay.
- We will not accept any homework later than 500 mins after the deadline.
- SUCourse's timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.