CS305 – Programming Languages

Fall 2023-2024

HOMEWORK 3

Implementing a Semantic Analyzer and Notifications for MailScript

Due date: December 04, 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 implement a tool which includes a simple semantic analyzer for MailScript language. Detailed information about this programming language was given in the second homework document. You can check it for more information.

The tool that you will implement in this homework will first check if a given MailScript program has any syntax error, grammatically. If there are no syntax errors, the tool will perform some semantic checks for simple cases and if these checks are passed, it will generate notifications for certain statements. Read the rest of the document for more information.

2 Parser and Scanner

The scanner and the parser which you can use to implement this homework will be provided to you. The semantic analysis will require you to implement an attribute grammar. You can start from the scanner/parser files provided, or of course, you can write your own versions of scanner and parser from scratch.

3 Semantic Rules

Your semantic analyzer should start by performing an analysis for the following semantic rules. For each violation of these rules, your semantic analyzer must print out an error message. Note that, after printing out an error message, your semantic analyzer must not terminate and keep working to find further violations, if there exists any.

3.1 Undeclared Variables

A variable has to be set to a certain value before it can be accessed in a statement. This is done by using the **set** keyword.

If a variable is used before it is set to a value, or it is never set to a value the program is supposed to generate an error along with the line information and the identifier. For example, for this program:

```
1     set toBeSent ("How are you?")
2     Mail from john.doe@mail.com:
3         send [Message] to [(user5@mail.com)]
4         send ["I have an early class."] to [(NameInfo, diane@mail.com)]
5     end Mail
6     set Message ("Hello there...")
```

The following should be printed out:

```
    ERROR at line 3: Message is undefined
    ERROR at line 4: NameInfo is undefined
```

Remark 1: A variable defined within a mail block is confined to that block only. Conversely, a variable defined outside a mail block is treated as a global variable, accessible throughout the program.

In the example given below, although the variable `myMessage` is set to a value within the first mail block, this is not visible in the second mail block.

```
1 Mail from john.doe@mail.com:
2   set myMessage ("Hello")
3   send [myMessage] to [(user5@mail.com)]
4 end
5      Mail
6
7 Mail from john.doe@mail.com:
8   send [myMessage] to [(user6@mail.com)]
9 end Mail
```

Hence there is an error in this program. The output of the tool should be:

```
ERROR at line 8: myMessage is undefined
```

Remark 2: In cases where a variable is defined multiple times, the last declaration of the variable should be regarded as its final and effective definition. This overrides any previous declarations.

For detailed information about variable scope see Section 5 item 4.

3.2 Printing Errors

Errors will be printed in the order they appear in the program. That means, the errors with the earliest line number will be printed first. If two or more errors occur on the same line, the errors will also be printed in the order that they are seen (left to right). For the program below:

```
Mail from ben@mail.com:
    send [Message] to [(User1, user1@x.com), (User2, user2@x.com)]
    send ["Hello"] to [(jane@mail.com), (User3, user3@x.com)]
end Mail
```

The output should be:

```
ERROR at line 2: Message is undefined
ERROR at line 2: User1 is undefined
ERROR at line 2: User2 is undefined
ERROR at line 3: User3 is undefined
```

Please note that the line number given is of the line in which the undeclared variable appears.

E-mails with the earliest date and time should be printed first. You **can not** assume that date and time objects will be given in the correct format. You should make the necessary checks for the date and time. The scanner that we provide already checks the structure so you should only validate the date and time on your implementation. (Check HW1 document for date/time structure and for validation check golden outputs)
For example, if your date and time are like this in the given mail block :

```
Mail from admin@mail.com:

    schedule @ [32/12/2023, 24:00]:
        send ["Happy New Year!"] to [(all@mail.com)]
    end schedule

end Mail
```

Then your output should be like:

```
ERROR at line 3: date object is not correct (32/12/2023)
ERROR at line 3: time object is not correct (24:00)
```

4 Notifications

After your semantic analyzer performs the semantic checks mentioned in Section 3 and if no errors are found in the end, it will move onto the second phase, which is generating notifications. **Please keep in mind that notifications will only be generated if the program contains no errors.** For an e-mail that is sent or for an e-mail that is scheduled to be sent, a notification should be printed to specify the related actions. There are two types of notifications: **send** notifications and **schedule** notifications.

4.1 Send Notification

For an e-mail that is sent, a notification that specifies the sender address, the message, and the recipient should be printed out. The format will be as given below:

```
E-mail sent from X to Y: "Message"
```

Some important points that should be taken into consideration are:

- A new notification should be generated for each unique recipient in the recipient list.
- The sender's information is only limited to their e-mail address. In other words, we only have the e-mail address for the sender. In the template notification message given above, `X` denotes this e-mail address.
- On the other hand, for a recipient, we have the e-mail address, but optionally, we may also have a name for the recipient. In the template notification message, `Y` denotes the recipient information. If the name information is given for the recipient, `Y` will be used as this name (no e-mail information for the recipient will be printed out in this case). However, when there is no name information provided for the recipient, then `Y` will be used as the e-mail address of the recipient. For the code below:

  ```
  Mail from nick@mail.com:
      send ["You're welcome."] to [("Luisa", luisa@mail.com),
      (anne@mail.com)]
  end Mail
  ```

  The output should be:

  ```
  E-mail sent from nick@mail.com to Luisa: "You're welcome."
  E-mail sent from nick@mail.com to anne@mail.com: "You're welcome."
  ```

- The recipient list can contain the same e-mail more than once. It can appear with different names or with no name provided at all. It is crucial that you print only ONE notification for a single e-mail address, no matter how many times it is repeated in the recipients list of a single send statement. In this case, the first (leftmost) one of the recipient objects (that share an e-mail address) will be taken into consideration. Once again, if the first recipient object contains a name, the name will be printed as Y. If the said recipient object does not contain a name, then the e-mail address will be printed as Y. For example:

  ```
  Mail from kate@mail.com:
      send ["Welcome."] to [(mike@mail.com), ("Mike", mike@mail.com)]
      send ["Hi."] to [("Joseph", joe@mail.com), ("Joe", joe@mail.com)]
  end Mail
  ```

  The output will be:

  ```
      E-mail sent from kate@mail.com to mike@mail.com: "Welcome."
      E-mail sent from kate@mail.com to Joseph: "Hi."
  ```

  On the second line, even though a name for the same user is provided in the second recipient object (`Mike`), we will only take the first recipient object into consideration. That is why `mike@mail.com` is printed, instead of `Mike`. On the third line, once again the first recipient object is taken into consideration and the name `Joseph` is printed instead of `Joe`.

- Please note that the message or the recipient's name can be given as a variable and not a string. In this case, the variable's value should be printed out, not the variable's name. For example:

  ```
      Mail from ian@mail.com:
          set newMail ("Welcome.")
          set newUser ("Courtney")
          send [newMail] to [(newUser, courtney@mail.com)]
      end Mail
  ```

  The output will be:

  ```
      E-mail sent from ian@mail.com to Courtney: "Welcome."
  ```

4.2 Schedule Notification

Schedule notifications are similar to send notifications with one key difference between them: schedule notifications are supposed to be printed after all the send notifications are printed. That means, all the send notifications will be printed in the order they are given in the code. Then, the schedule notifications will be printed in chronological order:

- If two or more e-mails are scheduled to be sent on the same date and time, the e-mail that appears first in the code (the statement with the lowest line number), should be printed first.
- The scheduled date (denoted as DATE in the template notification message below) will not be printed as it is. Instead, it will be converted to the Month Day, Year format. The time object will stay the same (denoted as TIME in the template notification message below). The examples of this conversion are given below:

  ```
  17/02/2021 -> February 17, 2021
  06/08/1997 -> August 6, 1997
  ```

- The format of a schedule notification is given below:

  ```
  E-mail scheduled to be sent from X on DATE, TIME to Y: "Message"
  ```

For example, if the input is the code given below:

```
Mail from root@mail.com:
    schedule @ [08/12/1365, 00:00]:
        send ["1"] to [("A", a@mail.com)]
    end schedule
    schedule @ [08/12/1365, 12:05]:
        send ["2"] to [("B", b@mail.com)]
    end schedule
    schedule @ [01/12/1365, 23:59]:
        send ["3"] to [("C", c@mail.com)]
    end schedule
end Mail
```

The output will be:

```
E-mail scheduled to be sent from root@mail.com on December 1, 1365, 23:59 to C: "3"
E-mail scheduled to be sent from root@mail.com on December 8, 1365, 00:00 to A: "1"
E-mail scheduled to be sent from root@mail.com on December 8, 1365 12:05 to B: "2"
```

4.3 Printing Notifications

Some details about printing notifications are clarified further below:

1. Send statements will be printed in the order they are given. That means, the send notification for the statement with the earliest line number will be printed first. If two or more send statements occur on the same line, the notifications will be printed in the order that the recipients are seen (left to right). For the program given below:

```
Mail from root@mail.com:
    send ["Hi."] to [("A", user1@mail.com), ("B", user2@mail.com)]
    send ["Bye."] to [("C", user3@mail.com), ("D", user4@mail.com)]
end Mail
```

The output should be:

```
    E-mail sent from root@mail.com to A: "Hi."
    E-mail sent from root@mail.com to B: "Hi."
    E-mail sent from root@mail.com to C: "Bye."
    E-mail sent from root@mail.com to D: "Bye."
```

2. A similar rule also applies to schedule notifications. Two or more e-mails that are **scheduled at the same date/time will be printed according to their line numbers.** If they also share a line, the leftmost recipient's notification will be printed first. For the program given below:

```
Mail from root@mail.com:
    schedule @ [05/06/2021, 22:00]:
        send ["Hi."] to [("A", user1@mail.com), ("B", user2@mail.com)]
        send ["Bye."] to [("C", user3@mail.com), ("D", user4@mail.com)]
    end schedule
end Mail
```

The output should be:

```
E-mail scheduled to be sent from root@mail.com on June 5, 2021, 22:00 to A: "Hi."
E-mail scheduled to be sent from root@mail.com on June 5, 2021, 22:00 to B: "Hi."
E-mail scheduled to be sent from root@mail.com on June 5, 2021, 22:00 to C: "Bye."
E-mail scheduled to be sent from root@mail.com on June 5, 2021, 22:00 to D: "Bye."
```

5 Example Programs and Outputs

1. If the program is not grammatically correct then like the second homework you have to print ***ERROR***. For example, if we have the below program:

```
    send ["This is an invalid program."] to [(cs305@mail.edu)]
```

Then the output should be:

```
    ERROR
```

2. If the inside of a mail block is empty, your code should not complain about it and should not print anything.

   ```
   Mail from cs305@mail.com:

   end Mail
   ```

   Then there should be **no output** for this mail block.

3. If a program is grammatically correct but contains violations of the semantic rules then the output should display all the semantic errors. For example, if we have the following program:

```
1   Mail from cs305@mail.com:
2
3       send ["Hello!"] to [("Daniel", daniel@mail.com),
4       (username, mehmet@mail.com), (mehmet@mail.com)]
5
6       send ["Bye."] to [(gamze@mail.com)]
7
8       schedule @ [18/04/2022, 06:30]:
9           send [Message] to [("Beril", beril@mail.com.tr)]
10      end schedule
11
12  end Mail
13
14
15  Mail from cs305@sabanciuniv.edu:
16
17      schedule @ [02/12/2021, 23:00]:
18          send ["Good morning!"] to [(ali@mail.com),
19          ("Ferhat Yilmaz", ferhat@mail.com), ("Ali", ali@mail.com)]
20      end schedule
21
22      send ["These are the files."] to [(user_45@mail.co.uk),
23      (Name_2, user_45@mail.co.uk)]
24
25  end Mail
```

   Then the output to the above program must be as follows:

   ```
   ERROR at line 4: username is undefined
   ERROR at line 9: Message is undefined
   ERROR at line 23: Name_2 is undefined
   ```

4. If a program is grammatically correct and does not contain any violations of the semantic rules then the output should display the notifications. For example if we have the following program:

```
set Outgoing ("This is a message.")

Mail from cs305@mail.com:
    set Outgoing ("CS 305")
    send ["Hello!"] to [("Daniel", daniel@mail.com),
    ("Ahmet", mehmet@mail.com), (mehmet@mail.com)]

    schedule @ [02/12/2021, 12:00]:
        send ["Good morning!"] to [(ali@mail.com),
        ("John Doe", john@mail.com), ("Ali", ali@mail.com)]
    end schedule


end Mail

set newMessage ("Thank you.")

Mail from cs305@mail.com:

    set Name ("Omer")
    schedule @ [28/11/2021, 04:00]:
        send [Outgoing] to [(Name, omer@mail.com)]
    end schedule

    send [newMessage] to [("u1", u1@x.com), (u2@x.com),
    (u1@x.com), (u1@x.com), (Name, u1@x.com)]

end Mail

```

Then the output for the above program must be as follows:

```
E-mail sent from cs305@mail.com to Daniel: "Hello!"
E-mail sent from cs305@mail.com to Ahmet: "Hello!"
E-mail sent from cs305@mail.com to u1: "Thank you."
E-mail sent from cs305@mail.com to u2@x.com: "Thank you."
E-mail is scheduled to be sent from cs305@mail.com on November 28, 2021, 04:00 to Omer: "This is a message."
E-mail is scheduled to be sent from cs305@mail.com on December 2, 2021, 12:00 to ali@mail.com: "Good morning!"
E-mail is scheduled to be sent from cs305@mail.com on December 2, 2021, 12:00 to John Doe: "Good morning!"
```

6 How to Submit

Submit your Bison file named as `username-hw3.y`, and flex file named as `username-hw3.flx` where `username` is your SU-Net username. You may use additional files, such as a header file. Please also upload those files.

We will compile your files by using the following commands:

```
flex username-hw3.flx
bison -d username-hw3.y
gcc -o username-hw3 lex.yy.c username-hw3.tab.c -lfl
```

So, make sure that these three commands are enough to produce the executable. If we assume that there is a MailScript file named test17.ms, we will try out your parser by using the following command line:

```
username-hw3 < test17.ms
```

7 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 6. 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. Some of the points that we can think of are:**
  - **There should be no extra space at the beginning or at the end of a line.**
  - **There is exactly one space between each word in a line.**
  - **Make sure that the spellings are as it is given in the homework document.**
  - **We check in a case sensitive manner (e.g. ”ERROR” $\neq$ ”error”)**
  - **The format of the error messages and notifications should be exactly the same as it is supposed to be.**
  - **If you are not sure about your outputs you can compare your outputs with the outputs given by the golden.**

- You may get help from our TA or from your friends. However, **you must implement the homework 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 machine 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 or days 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.