40-414 – Compiler Design, spring 1401, Sharif University of Technology

Programming Assignment IV

(Semantic Analyser)

(Plus an optional task!)

Released: Monday, 09/03/1401

Due: Sunday, 22/03/1401 at 11:59pm

1 Introduction

In programming assignment III, you implemented an intermediate code generator for Py-minus. In this assignment, you are to implement a semantic analyser for Py-minus. Please note that you may use codes from textbooks, with a reference to the used book in your code. However, using codes from the internet and/or other students in this course is **strictly forbidden** and may result in **Fail** grade in the course. Besides, even if you did not implement the parser in the previous assignment, you may not use the parsers from other students/groups. In such a case, you need to implement a parser, too.

2 Semantic Analyser Specification

In this assignment, you will implement a semantic analyser with the following characteristics for your compiler:

- The semantic analyser is called by the parser to perform semantic checks.
- Semantic analysis is performed in the same pass as other compilation tasks are performed (because the compiler is supposed to be a **one pass compiler**).
- Semantic analysis is performed in a manner very similar to one explained in Lecture 8 for the intermediate code generation. That is, the parser calls a function (let's call it '**semantic_check**') an action symbol appears on top of the parsing stack. Parser then pops the action symbol and passes it as an argument to the semantic analyser (i.e., '**semantic_check**' function). Semantic analyser then executes the associated **semantic routine**, and the control will return to the parser.
- Semantic errors are reported by appropriate error messages that are saved in an output text file called '**semantic_errors.txt**'.

3 Augmented Py-minus Grammar

To implement your semantic analyser, you should first add the required action symbols to the grammar of Py-minus that was included in section 3 of programming assignment II. For each action symbol, you need to write an appropriate semantic routine in **Python** that performs the required semantic check. Note that **you should not change the given grammar in any ways other than adding the required action symbols to the right hand side of the production rules**. The sample/test '**input.txt**' files for the main part of this assignment will be Py-minus programs, which may contain certain semantic errors. This assignment also includes an optional part, which is the implementation

of the final code generator (see section **6**). In implementing the required semantic routines for the semantic analyser, you should pay attention to the points that were mentioned in section 4 of assignment III.

4 Required Semantic Checks

All the semantic checks that are to be performed by the semantic analyser in this assignment are **static**. There is no need to implement any form of dynamic semantic checks. As it was mentioned before, possible semantic errors should be reported by an appropriate error message, which is saved in an output text file called '**semantic_errors.txt**'. The semantic analyser is supposed to detect the following **five** semantic error types. **Any other possible types of semantic error can be simply ignored.** Besides, for the sake of simplicity of the task, you can assume that every statement of the input program may include only **one** semantic error.

a) **Function 'main'**: Every input program should have a global main function with the signature 'def main()'. If the input file lacks such a signature, signal this error by the message: #lineno: Semantic Error! main function not found, in which the #lineno is the last line number of the input program. Note that in the correct test cases, function 'main' is always the last global definition of the input program.
b) **Scoping**: all globally variables must be declared **before** the definitions of functions. Besides, every function should be defined **before** it can be invoked. These are required in order to be able to implement a one pass compiler. If a variable or a function identifier with token ID lacks such an appropriate declaration, the error should be reported by the message: #lineno: Semantic Error! 'ID' is not defined appropriately, where 'ID' is the ill-defined variable/function.
c) **Actual and formal parameters number matching**: when invoking a function, the number of arguments passed via invocation must match the number of parameters that have been given in the function definition. Otherwise, the error should be reported by the message: #lineno: semantic error! Mismatch in numbers of arguments of 'ID', where 'ID' is the function that has been invoked illegally.
d) **Break statement**: if a 'break' statement is not within any while loops, signal the error by the message: #lineno: Semantic Error! No 'while' found for 'break'.
e) **Continue statement**: if a 'continue' statement is not within any while loops, signal the error by the message: #lineno: Semantic Error! No 'while' found for 'continue'.
f) **Type mismatch**: in a numerical and comparison operation, the type of operand in each side of the operation should not be void (calling a function that doesn't return a numerical value). Otherwise, the error should be reported by the message: #lineno: Semantic Error! Void type in operands.
g) **Overloading**: when there are multiple functions with the same name, if there are multiple functions with the same name, the number of arguments should be different. Otherwise, the error should be reported by the message: #lineno: Semantic Error! Function 'ID' has already been defined with this number of arguments, where 'ID' is the related function.

In the case that the input program is semantically correct, the file '**semantic_errors.txt**' should contain a sentence such as: '**The input program is semantically correct**.

5 Semantic Error Handling

There is no need to handle semantic errors except that errors must be appropriately reported. Therefore, your compiler should continue its normal tasks after reporting a semantic error so that it can detect other possible existing errors. However, there is no need to generate the address codes

if the input program contains any semantic error. In such cases, the '**output.txt**' will contain the sentence '**The output code has not been generated**'.

6 Implementing an optimizer (Optional)

In this part of assignment, you can optionally improve your compiler so that it can apply some of local and global optimization techniques introduced in Lecture Notes 11 and 12 to the three address codes generated by the intermediated code. In this exercise, the goal of optimization is to reduce the size of produced code. Therefore, you only need to consider those transformations that can reduce the number of generated three address codes (e.g., copy/constant propagation and local/global dead code elimination). The optimized code must produce the correct expected output. In other words, optimizing incorrect codes has no credit. The optimized codes will be compared against ordinary correct (but not optimized) codes such sample output.txt files distributed together with the programming assignment 3. Each case will be evaluated by comparing the optimized output and the standard output using the following formula:

$$\text{Mark} = 2 * (N-O) / N * 10,$$

in which, N is the number of three address codes in the sample/test case and O is the number codes in the optimized output file. Note that O must be strictly less than N in order to be evaluated. Besides, it is expected that by optimization, you could ideally reduce the size of standard output into half!

7 What to Turn In

Before submitting, please ensure you have done the following:

- It is your responsibility to ensure that the final version you submit does not have any debug print statements.
- You should submit a file named '**compiler.py**', which includes the Python code of scanner, predictive recursive descent parser, semantic analyser, and intermediated code generator modules. Please write your **full name(s)** and **student number(s)**, and any reference that you may have used, as a comment at the top of '**compiler.py**'.
- Your parser should be the main module of the compiler so that by calling the parser, the compilation process can start, and the parser then invokes other modules when it is needed.
- The responsibility of showing that you have understood the course topics is on you. Obtuse code will have a negative effect on your grade, so take the extra time to make your code readable.
- Your parser will be tested by running the command line '**python3 compiler.py**' in Ubuntu using Python interpreter version **3.8**. It is a default installation of the interpreter without any added libraries except for '**anytree**', which may be needed for creating the parse trees. No other additional Python's library function may be used for this or other programming assignments. Please do make sure that your program is correctly compiled in the mentioned environment and by the given command before submitting your code. It is your responsibility to make sure that your code works properly using the mentioned OS and Python interpreter.
- Submitted codes will be tested and graded using several different test cases (i.e., several **'input.txt**' files). Your compiler should read **'input.txt**' from the same working directory as that of '**compiler.py**'. In the case of a compile or run-time error for a test case, a grade of zero will be assigned to the submitted code for that test case. Similarly, if the code cannot produce the expected output (i.e., '**output.txt**') for a test case, or if executing **'output.txt**' by the **Tester** program does not produce the **expected** value, again a grade of zero will be assigned to the code for that test case. Therefore, it is recommended that you test your programs on several different random test cases before submitting your code. If you decided to implement the optional part of the assignment, your compiler will also be tested on a number of relevant inputs. Please note that the test case will be either a fully correct Py-minus program, in which

case the print outs of your generated code will be checked against the '**extected.txt**' and then against 'output.txt' files, or it is a program with a number of semantic errors of those types mentioned in section 4, in which case only the '**semantic_errors.txt**' file produced by your compiler will be evaluated (i.e., the content of output.txt will not matter in these cases).

- A few days after release of this description, you will receive a number of sample input-output files of the obligatory part of the assignment. Seventy percent of the test cases for evaluating your program will be picked up from the released sample inputs. Therefore, if your compiler can generate the expected outputs for all the released samples, you can be sure that your mark for this assignment will be at least 70 out of 100. In the optional part, you can use the same sample/test cases released with Programming assignment 3, as your samples. However, only fifty percent of these samples will be used in the evaluation process, which means you can at least gain 25 out of 50 if you handle those sample cases with maximum expected reduction ratio (i.e., 50%). Overall, in this assignment, you can gain a maximum of 150 marks, which will be scaled to 3 out of 20.
- The decision about whether the scanner, parser, semantic analyser, and intermediate code generator are included in '**compiler.py**' or appear as separate Python files is yours. However, all the required files should be read from the same directory as '**compiler.py**'. In other words, I will place all your submitted files in the same plain directory including a test case, and execute the '**python3 compiler.py**' command.
- You should upload your program files ('**compiler.py**' and any other files that your programs may need) to the course page in Quera (https://quera.ir/course/10726/) **before 11:59 PM, Tuesday, 22/03/1401**.
- Submissions with more than 100 hours delay will not be graded. Submissions with less than 100 hours delay will be penalized by the following rule:

Penalized mark = M * (100 - D) / 100

Where M = the initial mark of the assignment and D is the number of hours passed the deadline. Submissions with 50 < X <= 100 hours delay will be penalized by P.M. =M * 0.5.

Ghassem Sani, Gholamreza
08/03/1401, SUT