

Programming Assignment III
(Intermediate Code Generator)
(Plus two optional tasks)

Released: Tuesday, 20/10/1401

Due: Thursday, 13/11/1401 at 11:59pm

1 Introduction

In programming assignment II, you implemented an LR(1) parser for C-minus. In this assignment, you will implement an intermediate code generator for C-minus. Please note that you may use codes from textbooks with 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 Intermediate Code Generator Specification

In this assignment, you will implement the intermediate code generator with the following characteristics:

- The code generator is called by the parser to perform a code generation task, which can be modifying the semantic stack and/or generating a number of three address codes.
- Code generation is performed in the same pass as other compilation tasks are performed (because the compiler is supposed to be a **one-pass compiler**).
- Parser calls a function called '**code_gen**' and sends an **action symbol** as an argument to '**code_gen**' at appropriate times during parsing.
- Code generator (i.e., the '**code_gen**' function) executes the appropriate **semantic routine** associated with the received action symbol (based on the technique introduced in Lecture 8).
- Generated three-address codes are saved in an output text file called '**output.txt**'.

3 Augmented C-minus Grammar

To implement your semantic analyzer and intermediate code generator, you should first add the required action symbols to the grammar of C-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 or code generation tasks, such as modifying the semantic stack and/or generating a number of three address codes. Note that **you will need to change the given grammar (by adding a number of new non-terminal symbols and new epsilon rules) in a way that reduces actions that can occur at the desired points (see pages 51-52 in lecture note 8)**. Consequently, you will need to regenerate your parse table.

4 Intermediate Code Generation

The intermediate code generation is performed with the same method that was introduced in Lecture 8. All the programming constructs defined by the given C-minus grammar must be supported by your intermediate code generator, with the exception of nested procedures/functions and

recursive procedures/functions. The sample/test 'input.txt' files for the main part of this assignment will be C-minus programs, which do not contain any type of errors. This assignment has two optional parts in which input programs may contain certain semantic errors. The first optional part of the assignment is composed of the implementation of a **semantic analyzer** for your compiler (see section 6), and the second optional part includes the implementation of **non-recursive function calls** and **return statements** (see section 9).

In implementing the required semantic routines for the intermediate code generation, you should pay attention to the following points:

- Every input program may include a number of global variables and a main function with the signature '**void main (void)**'.
- All local variables of the functions are declared at the beginning of the functions. That is, there will not be any declaration of variables inside other constructs, such as loops.
- In conditional statements such as 'if' and/or 'repeat ... until', if the expression value is **zero**, it will be regarded as a '**false**' condition; otherwise, it will be regarded to be '**true**'. Moreover, the result of a '**relop**' operation that is **true**, will be '**1**'. Alternatively, if the result of a '**relop**' operation is '**false**', its value will be '**0**'.
- You should implicitly define a function called '**output**' with the signature '**void output (int a);**' which prints its argument (an integer) as the main program's output.

5 Available Three address Codes

In this project, you can only use the following three address codes. Three address codes produced by your compiler will be executed by an interpreter called '**Tester**', which can only interpret the following three address codes. Otherwise, the tester program fails to run your three address codes. Please note that the single and most important factor in evaluating your solution to this assignment is that the output of your intermediate code generator will be successfully interpreted by the '**Tester**' program and produce the expected output value. The '**Tester**' program and its help file are released together with this description.

	Three address code	Explanation
1	(ADD, A1, A2, R)	The contents of A1 and A2 are added. The result will be saved in R.
2	(MULT, A1, A2, R)	The contents of A1 and A2 are multiplied. The result will be saved in R.
3	(DIV, A1, A2, R)	The content of A1 is divided by A2. The result will be saved in R.
4	(SUB, A1, A2, R)	The content of A2 is subtracted from A1. The result will be saved in R.
5	(EQ, A1, A2, R)	The contents of A1 and A2 are compared. If they are equal, '1' (i.e., as a true value) will be saved in R; otherwise, '0' (i.e., as a false value) will be saved in R.
6	(LT, A1, A2, R)	If the content of A1 is less than the content of A2, '1' will be saved in R; otherwise, '0' will be saved in R.
7	(ASSIGN, A, R,)	The content of A is assigned to R.
8	(JPF, A, L,)	If the content of A is 'false', the control will be transferred to L; otherwise, next three address code will be executed.
9	(JP, L, ,)	The control is transferred to L.
10	(PRINT, A, ,)	The content of A will be printed to the standard output.

As it was explained in Lecture 8, in three address codes, you can use three addressing modes of direct address (e.g., 100), indirect address (e.g., @100), and immediate value (e.g., #100). For simplicity, you can suppose that all memory locations are allocated statically. In other words, we don't have a runtime stack or heap. Also, assume that **four** bytes of memory are required to store an integer. Therefore, the address of all data memory locations is divisible by **four**. The following figures show a sample C-minus program and the three address codes produced for it. Note that every three address codes are preceded by a line number starting from **zero**. The tester program outputs a value of '**15**' by running the three address codes in the given sample. For more information about the tester program and the formatting of the three address codes, please read the provided help file very carefully. As it was mentioned earlier, the grading of the code generation part of this assignment is solely based on whether or not the produced three address code can be successfully run by the **Tester** program and produce the expected value.

Note that the three address codes produced for an input program, such as the given sample in Fig. 1, need not be identical to the code given in Fig 2. There can be a virtually infinite number of correct three-address codes for such programs. As long as the produced code can be executed by the **Tester** program and prints the expected value(s), it is acceptable.

lineno	code
1	void main(void) {
2	int prod;
3	int i;
4	prod = 1;
5	i = 1;
6	while(i < 6) {
7	prod = i * prod;
8	i = i + 2;
9	} until (6 < i)
10	output (prod);
11	}

Fig. 1 C-minus input sample (saved in "input.txt")

	produced three address codes
0	(JP, 1, ,)
1	(ASSIGN, #1, 100,)
2	(ASSIGN, #1, 104,)
3	(LT, 104, #6, 108)
4	(JPF, 108, ,)
5	(MULT, 104, 100, 500)
6	(ASSIGN, 500, 100,)
7	(ADD, 104, #2, 504)
8	(ASSIGN, 504, 104,)
9	(JP, 3, ,)
10	(PRINT, 100, ,)

Fig. 2 'Output.txt' Sample

6 Semantic Analyser Specification (Optional task 1)

As it was mentioned above, in this assignment, the implementation of the semantic analyzer is. If you choose to implement this optional part, your semantic analyzer must have the following characteristics:

- The semantic analyzer 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 the one explained in Lecture 9 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. The parser then pops the action symbol and passes it as an argument to the semantic analyzer (i.e., the '**semantic_check**' function). The semantic analyzer 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**'.

7 Required Semantic Checks

All the semantic checks that are to be performed by the semantic analyzer 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 analyzer is supposed to detect the following **six** semantic error types. Any other possible types of semantic error can be simply ignored. Besides, for the sake of the simplicity of the task, you can assume that every statement of the input program may include only **one** semantic error.

- Scoping**: all variables must be declared, either globally or in the current scope before they can be used in any expression. Besides, every function should be defined before it can be invoked. These are required to be able to implement a one-pass compiler. If a variable or a function identifier with a token ID lacks such a declaration or definition, respectively, the error should be reported by the message: #lineno: Semantic Error! 'ID' is not defined, whereas 'ID' is the undefined variable/function.

- b) **Void type:** when defining a single variable or an array, the type cannot be void. In such a case, report the error by the error message: #lineno: Semantic Error! Illegal type of void for 'ID', where ID is the variable or array with the illegal type.
- 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' or 'switch case' statements, signal the error by the message: #lineno: Semantic Error! No 'while' or 'switch case' found for 'break'..
- e) **Type mismatch:** in a numerical and/or comparison operation, the types of operands on both sides of the operation should match. Otherwise, the error should be reported by the message: #lineno: Semantic Error! Type mismatch in operands, Got 'Y' instead of 'X', where 'Y' is the mismatched type, and 'X' is the expected type.
- f) **Actual and formal parameter type matching:** when invoking a function, the type of each argument passed via invocation must match the type of associated parameter in the function definition. Otherwise, the error should be reported by the message: #lineno: Semantic Error! Mismatch in type of argument N for 'ID'. Expected 'X' but got 'Y' instead', where 'N' is the number of the argument with the illegal type, 'ID' is the function's name, 'X' is the expected type, and 'Y' is the illegal type.

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**'.

8 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**'.

9 Generating code for function calls and return statements (Optional task 2)

In this assignment, you can optionally improve your compiler so that it can produce three address codes for **programs** that contain non-recursive function calls and return statements, such as the given example in Figure 3. The three-address code for this program should print numbers 0, 2, 10, and 12 when it is given to the tester program.

lineno	code
1	int foo(int x){
2	if (x < 0){
3	output(x);
4	return 1;
5	}
6	else {
7	output(x);
8	return x + 2;
9	} endif
10	}

11	void main(void){
12	int i;
13	i= foo(0);
14	output(i);
15	i=foo(10);
16	output(i);
17	}

Fig. 3 C-minus program sample with function call

10 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 the scanner, LR(1) parser, semantic analyzer, 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 compiler 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 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 either of the two optional parts 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 C-minus program, in which case the printouts of your generated code will be checked against the '**extected.txt**' file, or it is a program with a number of semantic errors of those six types mentioned in section 7, 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 the release of this description, you will also receive 10 input-output sample files. You also receive 5 extra input test cases without having access to their expected output.
- Your compiler will be evaluated by Quora's Judge System (QJS). These 15 cases will be added to QJS. After the assignment's deadline is passed, the five test cases plus half of the 10 samples will be used to evaluate your compiler.

- Similarly, you will also receive a number of samples and test cases for the optional parts of this assignment. The total mark of each optional part will be 20% of the main part of the assignment. In other words, if you pass all the test cases for both the main part and the two optional parts, your mark from this assignment will increase by a 1.4 multiplication factor.
- The decision about whether the scanner, parser, semantic analyzer, 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/11864/>) **before 11:59 PM, Thursday, 13/11/1401**.
- Submissions with more than 100 hours of delay will not be graded. Submissions with less than 100 hours delay will be penalized by the following rule:

$$\text{Penalized mark} = M * (100 - 0.5 * D) / 100$$

Where M = the initial mark of the assignment and D is the number of hours passed the deadline. Submissions with $50 < X \leq 100$ hours delay will be penalized by P.M. = $M * 0.75$.

Ghassem Sani, Gholamreza
20/10/1401, SUT