40-414 – Compiler Design, Fall 1399, Sharif University of Technology

Programming Assignment III

(Intermediate Code generator - Part 1)

Released: Sunday, 02/09/99

Due: Sunday, 23/09/99 at 11:59pm

1 Introduction

In programming assignment II, you implemented an LL(1) parser for C-minus. In this assignment you are to implement the first part of an intermediate code generator for C-minus. Please note that you may use codes from text books, 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. In such a case, you need to implement parser, too.

2 Intermediate Code Generator Specification

In this assignment, you will implement the first part of 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 9).
- Generated three-address codes are saved in an output text file called '**output.txt**'.

3 Augmented C-minus Grammar

To implement your semantic analyser 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. You should also implement a **Semantic Stack,** which can be used by these modules to store the necessary information (i.e., similar to the technique introduced in Lecture 6 for the intermediate code generation). 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**. Also note that the semantic analyzer will have a structure very similar to that of the intermediate code generator introduced on page 12 of Lecture note 9.

4 Intermediate Code Generation

The intermediate code generation is performed with the same method that was introduced in Lecture **9**. In the first part of implementing intermediate code generation in this assignment, all constructs

supported by the given C-minus grammar are to be implemented except for: **break** statements, **switch** statements, **return** statements, and **function calls**. Therefore, the sample/test '**input.txt**' files for this assignment will be a simple C-minus program, which does not contain any type of errors. In implementing the required sematic routines for the intermediate code generation, you should pay attention to the following points:

- Every input program may include only a number of global variables and contain just a main function with the signature '**void main (void)**'.
- All local variables of the main function are declared at the beginning of the function. That is, there will not be any declaration of variables inside other constructs such as while loops.
- In conditional statements such as 'if' and/or 'while', 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	(SUB, A1, A2, R)	The content of A2 is subtracted from A1. The result will be saved in R.
4	(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.
5	(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.
6	(ASSIGN, A, R,)	The content of A is assigned to R.
7	(JPF, A, L,)	If content of A is 'false', the control will be transferred to L; otherwise, next three address code will be executed.
8	(JP, L, ,)	The control is transferred to L.
9	(PRINT, A, ,)	The content of A will be printed to the standard output.

As it was explained in Lecture 6, 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 code is 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 to be identical to the code given in Fig 2. There can be 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 < 7) {
7	prod = i * prod;
8	i = i + 2;
9	}
10	output (prod);
11	}

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

	produced three address codes
0	(ASSIGN, #0, 500,)
1	(ASSIGN, #0, 504,)
2	(ASSIGN, #18, 508,)
3	(ASSIGN, #0, 516,)
4	(ASSIGN, #0, 520,)
5	(ASSIGN, #1, 516,)
6	(ASSIGN, #1, 520,)
7	(JP, 9, ,)
8	(JP, 16, ,)
9	(LT, 520, #7, 1000)
10	(JPF, 1000, 16,)
11	(MULT, 516, 520, 1004)
12	(ASSIGN, 1004, 516,)
13	(ADD, 520, #2, 1008)
14	(ASSIGN, 1008, 520,)
15	(JP, 9, ,)
16	(ASSIGN, 516, 500,)
17	(PRINT, 500, ,)

Fig. 2 'Output.txt' Sample

6 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.
- Together with this description, you will receive a number of sample input-output files. Half of the test cases for evaluating your program will be picked up from the released sample inputs. Therefore, if your code can generate the expected outputs for all the released samples, you can be sure that your mark for this assignment will be equal to or more than 50 out of 100.
- 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/6364/) **before 11:59 PM, Sunday, 23/09/99**.
- 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:

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

Where M = the initial mark of the assignment and D is number of hours passed the deadline.

Good Luck!

Ghassem Sani, Gholamreza

02/09/99, SUT