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

Programming Assignment IV

(Intermediate Code Generator - Part 2)

(Plus two Optional Tasks)

Released: Thursday, 04/10/99

Due: Thursday, 25/10/99 at 11:59pm

1 Introduction

In this programming assignment, you are supposed to complete the intermediate code generator of your compiler 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 assignment II, you may not use the parsers from other students. In such a case, you need to implement the parser, too.

2 Intermediate Code Generator Specification

In this assignment, you will implement the second 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 the second part of the intermediate code generator, you need to add a few more action symbols to the grammar of C-minus of the grammar of programming assignment III. For each action symbol, you need to write an appropriate semantic routine in **Python** that performs the required code generation tasks such as modifying the semantic stack and/or generating a number of three address codes. Note that again **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**.

4 Intermediate Code Generation

The intermediate code generation is performed with the same method that was introduced in Lecture **6**. In the second part of implementing intermediate code generation in this assignment, all constructs supported by the given C-minus grammar are to be implemented including **break** statements, **switch** statements, **return** statements, and **function calls**. However, you do not need to consider nested or recursive functions. Besides in this assignment, all the sample and test cases will be lexically, syntactically, and semantically correct programs. In implementing the required sematic 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 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.
- Implementation of **recursive functions** is optional (see section 9 below). In this assignment, implementation of a **semantic analyser** is optional, too (see section 6 below). Successful implementation of these two optional parts can **double** the total mark of this assignment!

5 Available Three address Codes

In this assignment, you can only use exactly the same set three address codes that was explained in section 5 of the description of Programming Assignment III. Besides, three address codes produced by your compiler will be executed by the same '**Tester**' program that was distributed with Programming Assignment III. Please note that again the most important factor in evaluating this assignment is that the output of your code generator will be successfully interpreted by the '**Tester'** program and produce the expected output value.

6 Semantic Analyser Specification (Optional)

In this assignment, implementation of the semantic analyser is optional and if it correctly reports the semantic errors of the given test cases, you can gain up to 50% the assignment's mark. If you choose to implement this optional part, your semantic analyser must have the following characteristics:

- 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 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. 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**'.

7 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 **six** 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.

1. **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 in order to be able to implement a one pass compiler. If a variable or a function identifier with token ID lacks such a declaration or definition, respectively, the error should be reported by the message: #lineno: Semantic Error! 'ID' is not defined, where 'ID' is the undefined variable/function.
2. **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.
3. **Actual and formal parameters number matching**: when invoking a function, the number of arguments passed via invocation must match the number of parameters that has 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.
4. **Break statement**: if a 'break' statement is not within any while loops or switch statements, signal the error by the message: #lineno: Semantic Error! No 'while' or 'switch' found for 'break'.
5. **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.
6. **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

In this project, there is no need to handle sematic errors apart from detecting and reporting the errors. Your compiler should be able to continue its tasks as normal after reporting a semantic error in order to 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 a sentence '**The output code has not been generated**'.

9 An Extra Optional task

In this assignment, you can optionally improve your compiler so that it can produce three address codes for **recursive programs** such as the following example for computing **Factorial** function. Please note that in order to do this task, you should somehow implement a sort of dynamic memory allocation such as a runtime stack. The three address code for this program should print 120 when it is given to the tester program:

lineno	code
1	int fact (int n)
2	{
3	int f;
4	if (n == 1) f = 1;
5	else f = n * fact (n - 1);
6	return f;
7	}
8	void main (void)
9	{
10	int i;
11	i = fact (5);
12	output (i);
13	}

Fig. 1 C-minus recursive program sample

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 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 either of the two optional parts of the assignment, your compiler will also be tested on a number relevant inputs. Please note that the test case will be either a fully correct C-minus program, in which case the print outs of your generated code will be checked against the '**extected.txt**' file, or it is a program with a number of sematic 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).

- 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 (or 100 out of 200 if you can also handle the extra sample cases provided for the optional parts).
- 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, Thursday, 25/10/99**.
- 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

04/10/99, SUT