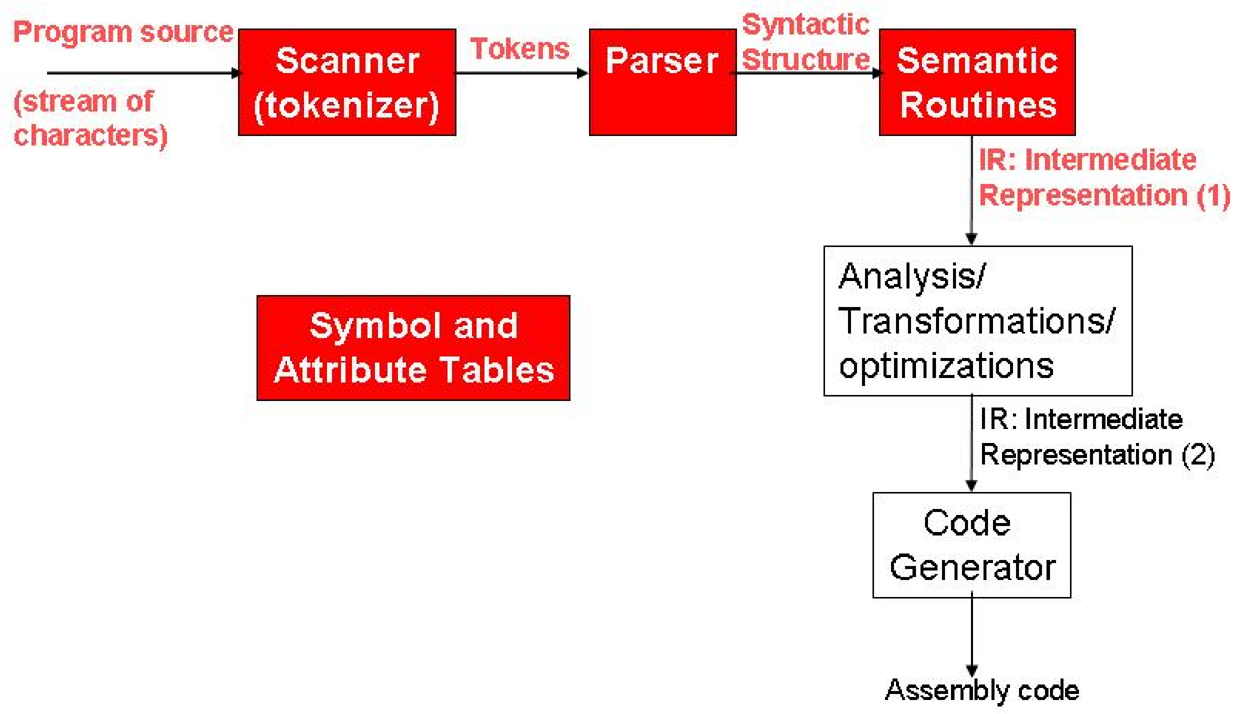
Step 4: Semantic Routines

In the so-called parser-driven compilers, the compilers call code-generation routines every time a grammar rule is recognized by the parser. This step deals with the code generation routines, also called semantic routines, for your language. This step assumes that you have fixed all the problems with the parser developed in "step 2" as well as the symbol table developed in "step 3". You may find useful to get familiar with the material listed in the "Readings" section of this handout before you start the implementation, especially *Processing Expressions and Data Structures References* (Chapter 11) and *Translating Control Structures* (Chapter 12) of (3). The diagram below shows the stage of the compiler construction after the completion of this step:


Program source
(stream of characters)
Scanner (tokenizer)
Tokens
Parser
Syntactic Structure
Semantic Routines
IR: Intermediate Representation (1)
Symbol and Attribute Tables
Analysis/ Transformations/ optimizations
IR: Intermediate Representation (2)
Code Generator
Assembly code

The Task: The compiler's semantic routines for expressions and I/O operations

Your goal in this step is to generate executable code for a simple program written using your language (a program that only has the main function, which calls no other functions). For simplicity reasons, the code that your compiler will generate is meant to be executed on the Tiny simulator (see below). In order to achieve this goal you have to add the semantic routines that will be executed by the parser every time a grammar rule is recognized. The rules you will have to deal with are the ones related to statements (assignment, read/write), expressions and IF/WHILE statements. The implementation strategy suggested is:

1. Generating an intermediate representation (IR):

The intermediate representation is the representation of the program in a language internal to the compiler. Many code optimization techniques operate on the intermediate code. However, for now, the IR will serve only as an intermediate step for the generation of the final tiny code. The intermediate representation is implemented as a linked-list that has every intermediate instruction as a node. For branch instructions the node also has a pointer to the target node. The intermediate instructions are machine independent. We suggest you to follow the IR description provided in the “step4_guide.pdf”.

For this step you can assume that all variables are defined globally. There will not be any additional variables defined in main().

2. The tiny simulator

The tiny simulator is meant to work as a simplified version of a real machine. It works by executing a stream of assembly instructions. A description of the tiny simulator is in “tinyDoc.txt”. The “tinyNew.C” file is the source code for the tiny simulator. You can compile the source code using the following command: g++ -o tiny tinyNew.C

3. Code generation

You implementation of item 1 will generate an internal representation of the micro program and store that representation in memory using a linked-list. In the code generation process you have to implement a procedure that will traverse the linked-list and translate the IR instructions to tiny instructions. Note that your IR code consists in three operand instructions as opposed to the two operand instructions of the tiny simulator. Therefore, some instructions in the IR representation will be translated in more than one tiny instruction. Please read carefully the tiny manual to understand how the tiny instructions are defined.

For this step you don't need to care about register allocation or semantic records, therefore the symbols of the final tiny code can be the same ones used in the intermediate representation. You will use a version of the tiny simulator that allows your code to use up to 1000 registers. That would be enough to map each temporary result to one different register and there will still be registers available for intermediate operations that may need extra registers.

Input/Output format requirements

Execution of the compiler would output BOTH the IRcode and tiny code and combine them together on the standard output. The output forms an executable file on the tiny simulator. It should start with the IRcode, with each line commented out by prepending a ';' to its front. Then the second half of the output should be the tiny code. Please take a look at the example outputs given below.

Grading: This step will be graded by running the outputed code on the tiny simulator, and looking at the results. Since the IR codes are commented out, they don't affect tiny machine execution.

Testcases

Available in “Step4_files.zip” archive.

Please note that these files are only examples -- it is perfectly ok if your output is different from them. What really matters is the execution result of your code. In other words, it is perfectly ok if your generated IR or Tiny is different.

Readings

Several books introduce the theory necessary for a good understanding of lexical analysis and how a compiler tokenizer works, as well a good description of how a simple compiler is structured. You may find it helpful to check some of the following references:

1. Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers, Principles, Techniques*. Addison wesley.
2. Andrew, W. A., & Jens, P. (2002). *Modern compiler implementation in Java*.
3. Fischer, C. N., LeBlanc. RJ Jr. (1991). *Crafting a Compiler with C*.
4. Levine, John R., Tony Mason, and Doug Brown. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

Q&A

- READ(a,b,c); means to do READ(a); READ(b); READ(c);. In other words, with READ (id_list) you should read from left to right. Same thing is true for WRITE.
- Implicit type conversion from int to float (when int and float values are used together) is not required. Indeed these cases will not appear in the testcases.
- Method-local variables don't appear in step 4. Assume all variables are declared in the global domain, and we create code for the main function only.