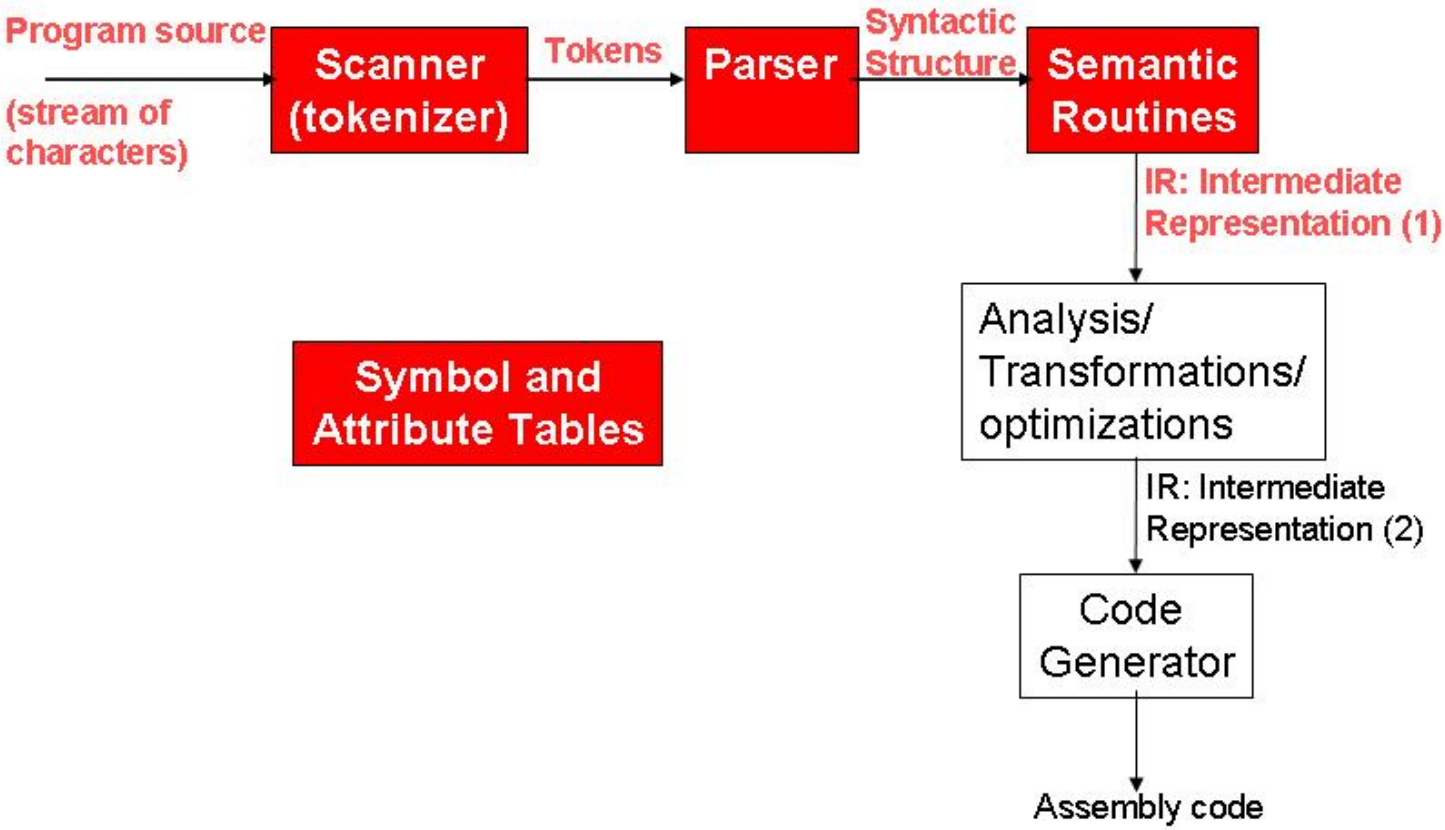
Step 4: Semantic Routines (I) - Due Friday Mar. 11th, 11:59pm

Introduction

Note: the step for [expression,I/O] and [control structures] is split into two steps, step4 and step5. So, the webpage for step4 and step5 are almost the same. Basically, you can refer to step4guide.pdf for instructions on both steps.

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, as well as "step 5 and 6", deals with the code generation routines, also called semantic routines, for the LITTLE 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. The diagram below shows the stage of the compiler construction after the completion of this step:



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 the LITTLE 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.

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 (please, refer to the "step 4" described in the course syllabus). The implementation strategy we suggest is:

(1) Generate 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 here.

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 executing a stream of assembly instructions. The description of the tiny simulator and its source code can be found here.

The linux Tiny version can be found here(64bit)

(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 all each temporary result to one different register and there will be still registers available for intermediate operations that may need extra registers.

(4) Output examples:

- Example of **IR** generated by a micro compiler for step4_testcase.micro, step4_testcase.IR;
- Example of **tiny code** for step4_testcase.micro: step4_testcase.tiny

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: We will grade by running the outputed code on the tiny simulator, and look at the results. Since the IR codes are commented out, they don't affect tiny machine execution. (Although we will look at them too.)

(5) Testcases

- step4_testcase3.micro, step4_testcase3.IR, step4_testcase3.tiny;
- test_expr.micro, test_expr.IR, test_expr.tiny.

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.

(6) Questions about semantics

As you implement the semantic routines you may feel the need for clarification about particular subtleties in the semantics. You are encourged to ask about them on the class newsgroup. Also we will update this section from time to time to help answer the most common questions, so please check back often.

- 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.

- We don't require implicit type conversions from int to float (when int and float values are used mixingly). 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. Of course in later steps this assumption does not hold true any more.

References

- **Charles Fischer, Richard LeBlanc, "Crafting a Compiler with C", The Benjamin/Commings Publishing Company, Inc, 1991.**
  Processing Expressions and Data Structures References - Chapter 11
  Translating Control Structures- Chapter 12