Project Step 4: Code Generation

Your goal is to generate code for assignments and *simple* expressions. To do this, we will build semantic actions that generate code in an *intermediate representation* (IR) for assignment statements and simple expressions, and then translate that intermediate representation to assembly code. Expressions with only two source operands are called simple expressions here (e.g. x = y + z).

You can do this in three steps, but you can also choose to do it in two steps (step 1 is optional):

1. Generate an *abstract syntax tree* (AST) for the code in your function.
2. Convert the AST into a sequence of *IR Nodes* that implement your function using three address code.
3. Traverse your sequence of IR Nodes to generate assembly code.

We will discuss each of these steps next. (Note: we will only have one function in our program, `main`. You can assume that all variables are defined globally. There will not be any additional variables defined in main().)

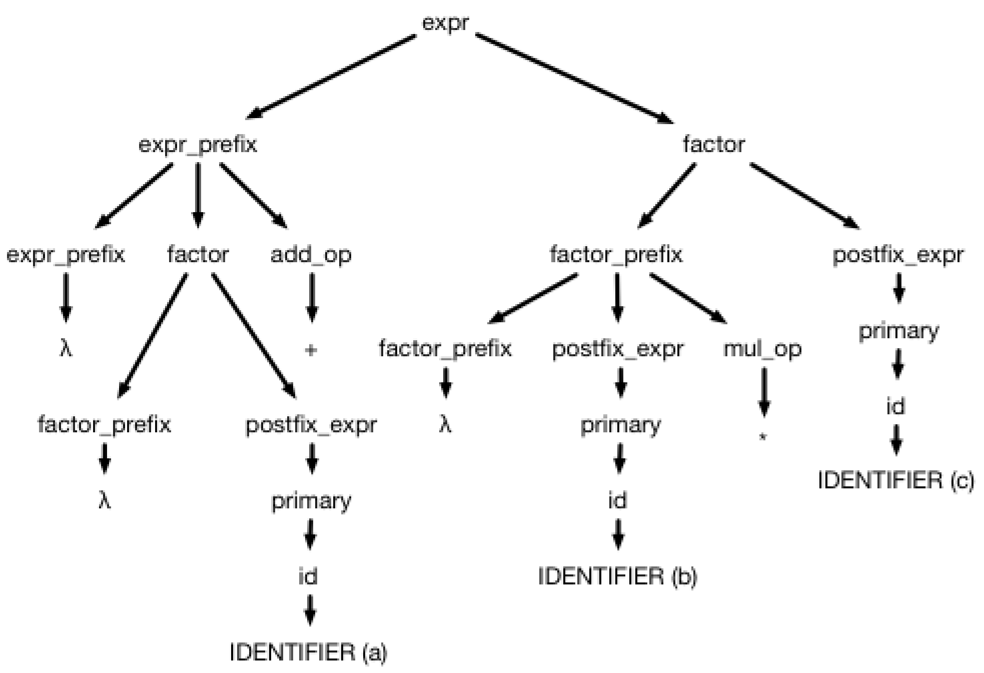
Abstract Syntax Tree

An Abstract Syntax Tree is essentially, a cleaned-up form of your parse tree that more straightforwardly captures the structure your program. For many compilers, the AST *is* the intermediate representation, though we will further convert the AST into another intermediate representation.

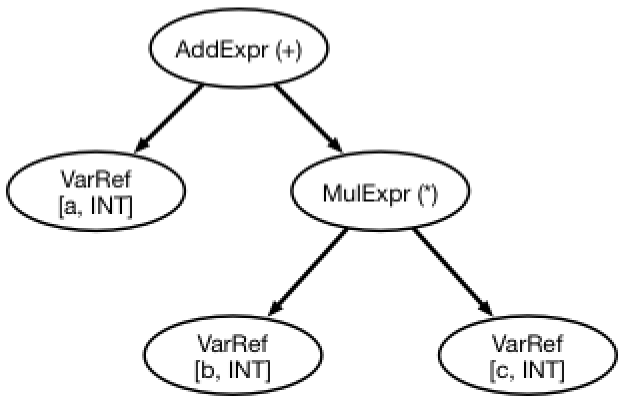
What is the difference between a parse tree and an AST?

Parse trees capture all of the little details necessary to implement your grammar. This means that it often contains extraneous information beyond what is necessary to capture the details of a piece of code (e.g., there are nodes for tokens like ";", and nodes for all of the sub-constructs we used to correctly implement order of operations). ASTs, in contrast, contain exactly the information needed to capture the meaning of an expression, including being structured to preserve order of operations.

For example, consider the parse tree for `a + b * c`:



Complicated, huh? Here's an abstract syntax tree that captures the same thing:



Much simpler! We aren't preserving anything except the bare minimum needed to describe the expression (note that we included the type of each of the variables in the program -- we can get that information from our symbol table!).

Building an AST

Note that the information in the AST is associated with various nodes in the parse tree. We can use semantic actions, just as we did previously, to pass information "up" the parse tree to build up the AST. Instead of passing information about a declaration, we can instead pass partially constructed abstract syntax tree nodes.

IR: 3 Address Code

The next step in our compilation process is to generate *3 Address Code* (3AC), which is our intermediate representation. 3AC is an intermediate representation where each instruction has at most two source operands and one destination operand. Unlike assembly code, 3AC does not have any notion of registers. Instead, the key to 3AC is to generate *temporaries* -- variables that are used to hold the intermediate results of computations. For example, the 3AC for `d := a + b * c` (where all variables are integers) will be:

```
MULTI b c $T1
ADDI a $T1 $T2
STOREI $T2 d
```

Generating 3AC

Generating 3AC is straightforward from an AST. We can perform a post-order walk of the tree, passing up increasingly longer sequences of IR code called `CodeObjects`. Each code object retains three pieces of information:

1. A sequence of IR Nodes (a structure representing a single 3AC instruction) that holds the code for this part of the AST (i.e., that implements this part of the expression)
2. An indication of where the "result" of the IR code is being stored (think: the name of the temporary or variable where the result of the expression is stored)
3. An indication of the type of the result

Then, when we encounter something like a node for "+", we can generate code for the overall expression as follows:

1. Create a new `CodeObject` whose code list is all the code from the left child followed by all the code for the right child.
2. Use the `result` fields of the left and right `CodeObject`s to create a new 3AC instruction performing the add, storing the result in a new temporary. Add this new instruction to the end of your code list.
3. Indicate in your `CodeObject` the temporary where the result is stored, and its type.
4. Return the new `CodeObject` up the AST as part of your post-order walk.

Hint: the `CodeObject` for a simple variable won't have any 3AC code associated with it. Instead, mark the variable itself as the "temporary" the result is stored in.
Hint: You may find it useful to write a helper function to generate "fresh" temporaries.

Then, when you get to the top of the AST, you will have a single `CodeObject` that contains all of the IR code for the entire `main` function.

3AC instructions

Here are the 3AC instructions you should use:

```
ADDI  OP1 OP2 RESULT (Integer add; RESULT = OP1 + OP2)
SUBI  OP1 OP2 RESULT (Integer sub; RESULT = OP1 - OP2)
MULTI  OP1 OP2 RESULT (Integer mul; RESULT = OP1 * OP2)
DIVI  OP1 OP2 RESULT (Integer div; RESULT = OP1 / OP2)

ADDF  OP1 OP2 RESULT (Floating point add; RESULT = OP1 + OP2)
SUBF  OP1 OP2 RESULT (Floating point sub; RESULT = OP1 - OP2)
MULTF  OP1 OP2 RESULT (Floating point mul; RESULT = OP1 * OP2)
DIVF  OP1 OP2 RESULT (Floating point div; RESULT = OP1 / OP2)

STOREI OP1 RESULT (Integer store; store OP1 in RESULT)
STOREF OP1 RESULT (Floating point store; store OP1 in RESULT)

READI RESULT (Read integer from console; store in RESULT)
READF RESULT (Read float from console; store in RESULT)

WRITEI OP1 (Write integer OP1 to console)
WRITEF OP1 (Write float OP1 to console)
WRITES OP1 (Write string OP1 to console)
```

Generating Assembly

Once you have your IR, your final task is to generate assembly code. In this class, we will be using an assembly instruction set called *Tiny*. The tiny simulator is meant to work as a simplified version of a real machine. It works by executing a stream of assembly instructions. The "tinyNew.C" file is the source code for the tiny simulator. You can compile the source code on *osprey* server using the following command: `g++ -o tiny tinyNew.C`

See the *tinyDoc.txt* for details about the instruction set.

This task is fairly straightforward: iterate over the list of 3AC you generated in the previous step and convert each individual instruction into the necessary Tiny code (note that Tiny instructions reuse one of the source operands as the destination, so you may need to generate multiple Tiny instructions for each 3AC instruction).

We will be using a version of Tiny emulator that supports 1000 registers, so you can more or less directly translate each temporary you generate into a register (i.e. you don't have to worry about efficient register allocation).

What you need to do

In this part, you will be generating assembly code for assignment statements, *simple* expressions, and READ and WRITE commands. Use the steps outlined above to generate Tiny code. Your code should output a list of tiny code (which is Non-graded Output) that we will then run through the Tiny emulator to make sure you generated the right result (which is the Graded Output).

For debugging purposes, it may also be helpful to emit your list of IR code. You can precede a statement with a `;` to turn it into a comment that our simulator will not interpret.

Handling errors

All the inputs we will give you in this step will be valid programs. We will also ensure that all expressions are type safe: a given expression will operate on either INTs or FLOATs, but not a mix, and all assignment statements will assign INT results to variables that are declared as INTs (and respectively for FLOATs).

How and where to implement actions?

We prohibit you to edit the g4 file. It is okay to do minor modifications (especially if your step1/step2 code did not give 100% correct results).

What you cannot do is add any application-specific code (e.g. java code) into the grammar file. In other words, embedding actions within grammar rules is prohibited. You should only use either Listener functionality for implementation. Otherwise, points may be deducted.

The reason for the above restriction is given below (excerpt is taken from the ANTLR book: https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference).

"To get reusable and retargetable grammars, we need to keep them completely clear of user-defined actions. This means putting all of the application-specific code into some kind of listener or visitor external to the grammar. Listeners and visitors operate on parse trees, and ANTLR automatically generates appropriate tree-walking interfaces and default implementations. Since the event method signatures are fixed and not application specific" this ensures that the same grammar can be used for many applications.

What you need to submit

- The ANTLR grammar dentition (.g4) you wrote for LITTLE. Make sure to name it **Little.g4**.
- Your source program that is the driver (i.e. one with the *main* method). Make sure to name it **Driver.java**.
- Any other .java files that were not auto generated by ANTLR (meaning you wrote them).
- IMPORTANT: The grader will be using an executable called 'Micro' that takes care of running ANTLR, compiling source files, and running your code. On *osprey.unf.edu* server, the instructor will type './Micro.sh', followed by an input file name, on the terminal (as shown below), and have your code execute. Use *chmod* Unix command for setting the permission. **Submitting code that cannot be executed using the given Micro script may result in 50% penalty.**

  ```
  $ ./Micro.sh sqrt.micro
  ```

What you will NOT submit

- The ANTLR jar file
- All files in 'step4_files' archive
- All files that were auto generated (i.e., you did not write them)

Grading – VERT IMPORTANT

In this step, we will only grade your compiler on the correctness of the generated Tiny code. We will run your generated code through the Tiny simulator and check to make sure that you produce the same result as our code. When we say the result, we mean the outputs of any WRITE statements in the program (excluding any tiny simulator statistics like num. of clock cycles). Any testcase that requires console input (sys read) will receive random inputs from the grader and result(s) will be calculated based on these random inputs.

We will not check to see if you generate exactly the same Tiny code that we generate. In other words, we only care if your generated Tiny code *works correctly*. You may generate slightly different Tiny code than we do but still get full credit as long as the output from the simulator is same as ours.

Note: This step will be graded automatically, and the tiny outputs generated by your code will be directly compared with the expected tiny outputs using "diff -B -b -s" command. Please make sure your tiny outputs are identical to the expected tiny outputs provided.

Assessment Policy and Extra Credit

For 80% credit on this assignment, your generated code merely needs to work properly (we will not consider how fast your code runs).

For the remaining 20% of the grade, we will also evaluate how fast your Tiny code runs (the "Total Cycles" reported by the Tiny simulator). If your number of clock cycles is 5% or better than the given tiny code, you will get full credit. If not, your score will be proportional to the amount of improvement. The above will be separately computed for each individual test case.

The **top three** submissions whose generated Tiny code runs fastest (averaging across all of the inputs) receive XP points. In this step, how early you submit is NOT factored in for XP points.