Project Step 4 — Part 1: Expressions

Your goal in part 1 is to generate code for expressions. To do this, we will build semantic actions that generate code in an *intermediate representation* (IR) for assignment statements and expressions, and then translate that intermediate representation to assembly code.

We recommend that you do this in three steps, as it will make it easier to debug your code, 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: in this step 4, 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().)

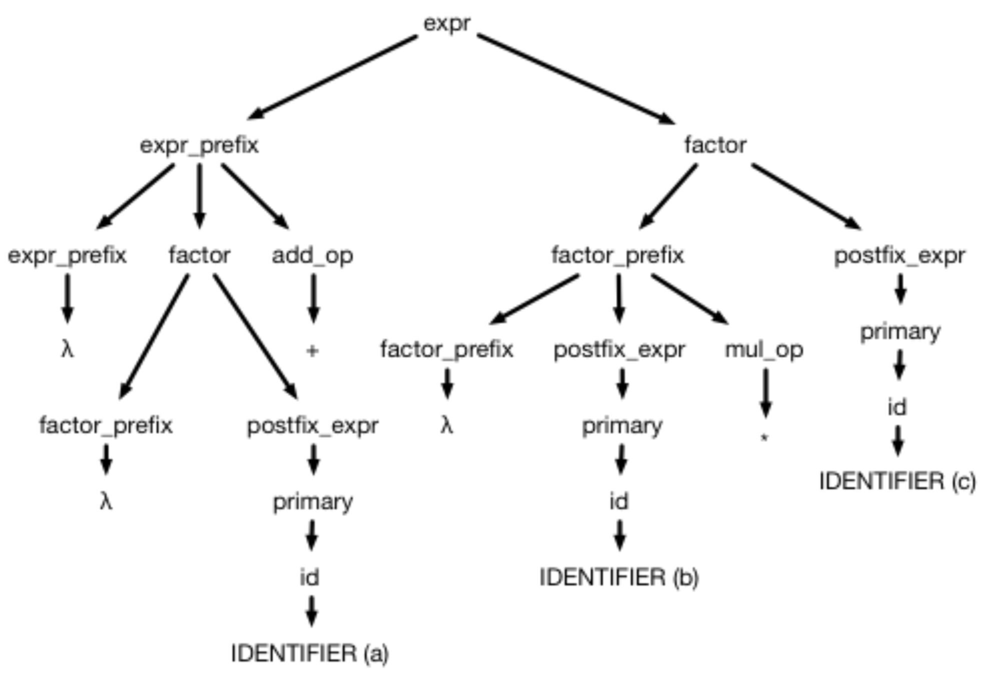
Abstract Syntax Tree

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

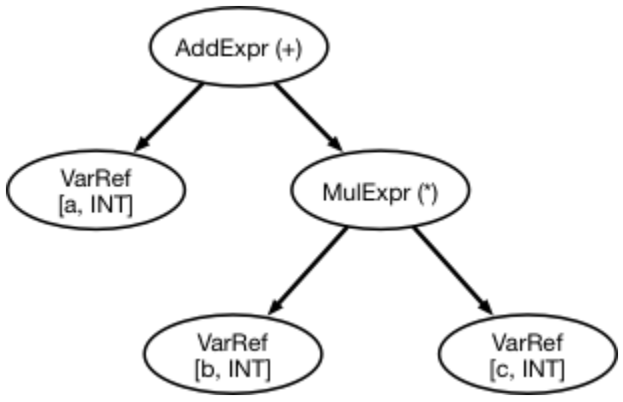
What is the difference between a parse tree and an AST?

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 in Step 3, 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 (you may want to define a class or structure called `ASTNode` that can be the "return type" for the relevant constructs). For example:

- `add_op` : generate an `AddExpr` AST node that has two children (that you leave uninitialized) and keeps track of the operator (`+` or `-`).
- `expr_prefix`: this will have three AST Nodes passed up from its sub rules: one from `expr_prefix` (which may be NULL, but otherwise will be an `AddExpr` node missing its right child), one passed up from `factor` (which will be a complete AST Node with all its fields filled in), and one passed up from `add_op` (which will be an `AddExpr` node that has neither its left nor its right child filled in).
    1. If `expr_prefix` is NULL, make the `add_op` node's left child the node from `factor` and return up the `add_op` node (note that it won't have its right child filled in!)
    2. If `expr_prefix` *isn't* NULL, note that it will be missing its right child. Make the `factor` node its right child, then make the `expr_prefix` node the `add_op` node's left child, which you pass up.

This basic idea: creating AST nodes when you have the information for a new node, then filling in various fields of the node as you work your way up the parse tree, will let you eventually create an AST for all the statements in the function.

Hint: you should also create an AST node to capture lists of statements; each element of the list will point to an AST node for a single `assign_stmt`.

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 `CodeObject`s. 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 (INT or FLOAT)

Then, when we encounter something like an `AddExpr` Node, 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 of the `AddExpr` 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.

Note: We are generating code by performing a post-order walk of the AST. You can also generate code using this strategy by performing a post-order walk of the parse-tree (which is why you can optionally skip building the AST).

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 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, 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 will be displayed as Non-graded Output) that we will then run through the Tiny emulator to make sure you generated the right result (which will be displayed as 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).

Grading

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 result, we mean the outputs of any WRITE statements in the program. Any testcase that requires console input (sys read) will receive random inputs 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. In other words, we only care if your generated Tiny code *works correctly*. You may generate slightly different Tiny code than we did.

Project Step 4 — Part 2: Control Structures

This step builds on Part 1. Now that we are able to generate code for lists of statements, what happens if those lists of statements are embedded in control structures? (IF statements and FOR loops)?

(Note: as in part 1, 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())

ASTs for Control Structures

ASTs for control structures are, intuitively, simple: each control structure will have several children (3 in the case of an IF statement, etc.) that are themselves ASTs (ASTs for statement lists in the case of the bodies of IF statements and WHILE loops, ASTs for conditional expressions in the case of the conditions in the IF statements and WHILE loops, etc.). Because you already have working code for building an AST for statement lists (and can readily adapt your code for binary expressions to build ASTs for conditional expressions), all you have to do is create semantic actions for the control structures that "stitch together" the existing ASTs.

Generating 3AC for Control Structures

Generating 3AC for control structures builds directly on your code for part 1, which is able to generate code for lists of statements. This means that when you are generating code for an IF AST node, you know that the 3AC for the three children already exists. All that is left is to put them together in the correct order and insert any necessary labels and jumps.

There are two things that you need to pay attention to when putting together 3AC:

Generating Labels

At various points in your code, you will need to insert labels and jumps to allow control to transfer from one part of your code to another. You will need to make sure that you can generate *unique* labels every time (since your code will not work properly if there are multiple labels with the same name).

The 3AC you will generate for labels looks like:

```
LABEL STRING
```

Where STRING is whatever name you decide to give to your label

Generating Jumps

Unconditional jumps (like you might use to jump over an ELSE block) are easy:

```
JUMP STRING
```

Where STRING is the label you want to jump to.

Conditional jumps are a little bit tricky in our 3AC (and in Tiny): you need to generate the right kind of jump:

```
GT  OP1 OP2 LABEL (Jump to Label if OP1 > OP2)
GE  OP1 OP2 LABEL (Jump to Label if OP1 >= OP2)
LT  OP1 OP2 LABEL (Jump to Label if OP1 < OP2)
LE  OP1 OP2 LABEL (Jump to Label if OP1 <= OP2)
NE  OP1 OP2 LABEL (Jump to Label if OP1 != OP2)
EQ  OP1 OP2 LABEL (Jump to Label if OP1 == OP2)
```

To generate the right kind of jump for a conditional expression, you will need to inspect the AST node for the comparison operation and use that information to select the right 3AC instruction.

Note: the 3AC does not preserve type information about what kind of comparison you are doing, but the Tiny code for jumps does; you may want to extend either your 3AC or your data structure to keep track of this information.

What you need to do

In this part, you will be generating assembly code for IF statements and WHILE loops. Use the steps outlined above to generate Tiny code. Your compiler should output a list of tiny code that we will then run through the Tiny simulator to make sure you generated the right result.

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).

Grading

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 result, we mean the outputs of any WRITE statements in the program.

We will not check to see if you generate exactly the same Tiny code that we. In other words, we only care if your generated Tiny code *works correctly*. You may generate slightly different Tiny code than we did.