

ECE 468: Intro to Compilers and Translation Systems Engineering

Fall 2017

MWF, 1:30–2:20, WALC 3087

Project Step 4 — Expressions

Due: October 18th

Your goal in this step 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. You may also find the Lecture 4 notes helpful.

(Note: in this step, and step 5, we will only have one function in our program, `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`:

Quick Links

[Home](#)
[Syllabus \(PDF\)](#)
[Piazza](#)
[Blackboard](#)
[Calendar](#)

Course details

Instructor

Milind Kulkarni
milind 'at' purdue 'dot' edu
EE 324A
Office hours:

- Mondays, 2:30–4:00 PM
- Thursdays, 10:00–11:30 AM

Teaching Assistant

Chris Wright
wrigh338 'at' purdue 'dot' edu

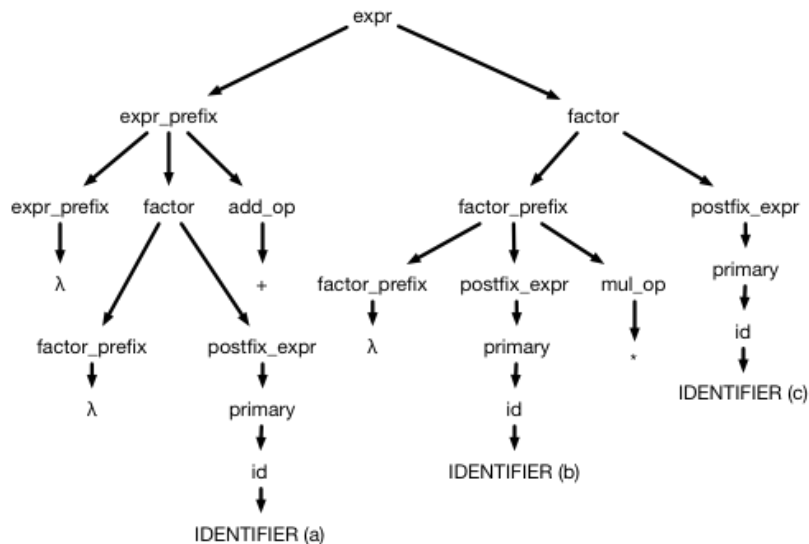
Instructional Lab

Location: EE 207
Hours:

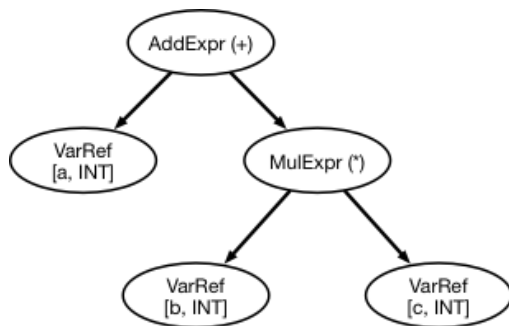
- Wednesdays, 5:00–7:00 PM
- Fridays, 3:00–5:00 PM

Assignments

[Submission instructions](#)
[Step 0: Test submission](#). Due 8/25
[Step 1: Scanner](#). Due 9/6
[Step 2: Parser](#). Due 9/15
[Step 3: Symbol table](#). Due 9/29
[Step 4: Expressions](#). Due 10/18
[Step 5: Control Structures](#). Due 11/1
[Step 6: Functions](#). Due 11/17
[Step 7: Register Allocation](#). Due 12/4



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 `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 (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 `CodeObjects` 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)
MULF  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. See the [Tiny documentation](#) for details about the instruction set.

This step 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).

For now, we will be using a version of Tiny that supports 200 registers, so you can more or less directly translate each temporary you generate into a register.

Testing your Tiny code

You can test your Tiny code by running it on our [Tiny simulator](#). This simulator can be built by running:

```
> g++ -o tiny tinyNew.C
```

You can then run a program with tiny commands using:

```
> ./tiny <code file>
```

What you need to do

In this step, 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 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).

Sample inputs and outputs

The inputs and outputs we will test your program can be found [here](#).

A sample compiler (a .jar file that you can invoke with -jar) is available [here](#).

Grading

In this step, we will only grade your compiler on the correctness of the generated 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 (not details such as how many cycles the code uses, how many registers, etc.)

We will not check to see if you generate exactly the same code that we do -- no need to diff anything. We only care if your generated code *works correctly*. You may generate slightly different code than we did.

Extra credit

For full credit on this assignment, your generated code merely needs to work properly. We will not consider how fast your code runs. *However*, we will also evaluate how fast your Tiny code runs (the "Total Cycles" reported by the Tiny simulator).

The groups whose generated Tiny code runs fastest (averaging across all the inputs) will receive bonus points for this step: 15% for the fastest code, 10% for second, and 5% for third.

What you need to submit

- All of the necessary code for your compiler that you wrote yourself. You do not need to include the ANTLR jar files if you are using ANTLR.
- A Makefile with the following targets:
 1. `compiler`: this target will build your compiler
 2. `clean`: this target will remove any intermediate files that were created to build the compiler
 3. `team`: this target will print the same team information that you printed in step 0.
- A shell script (this *must* be written in bash, which is located at `/bin/bash` on the ecegrid machines) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the scanner and second, the filename where you want to put the scanner's output. You can assume that we will have run `make compiler` before running this script.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation or test cases. If you have a folder named `test` in your repo, it will be deleted as part of running our test script (though the deletion won't get pushed) -- make sure no code necessary for building/running your compiler is in such a directory.

You should tag your step 4 submission as `step4-submission`

Layout based on website design by [Milind Kulkarni](#).