CS352

Lab 3: Abstract Syntax Trees

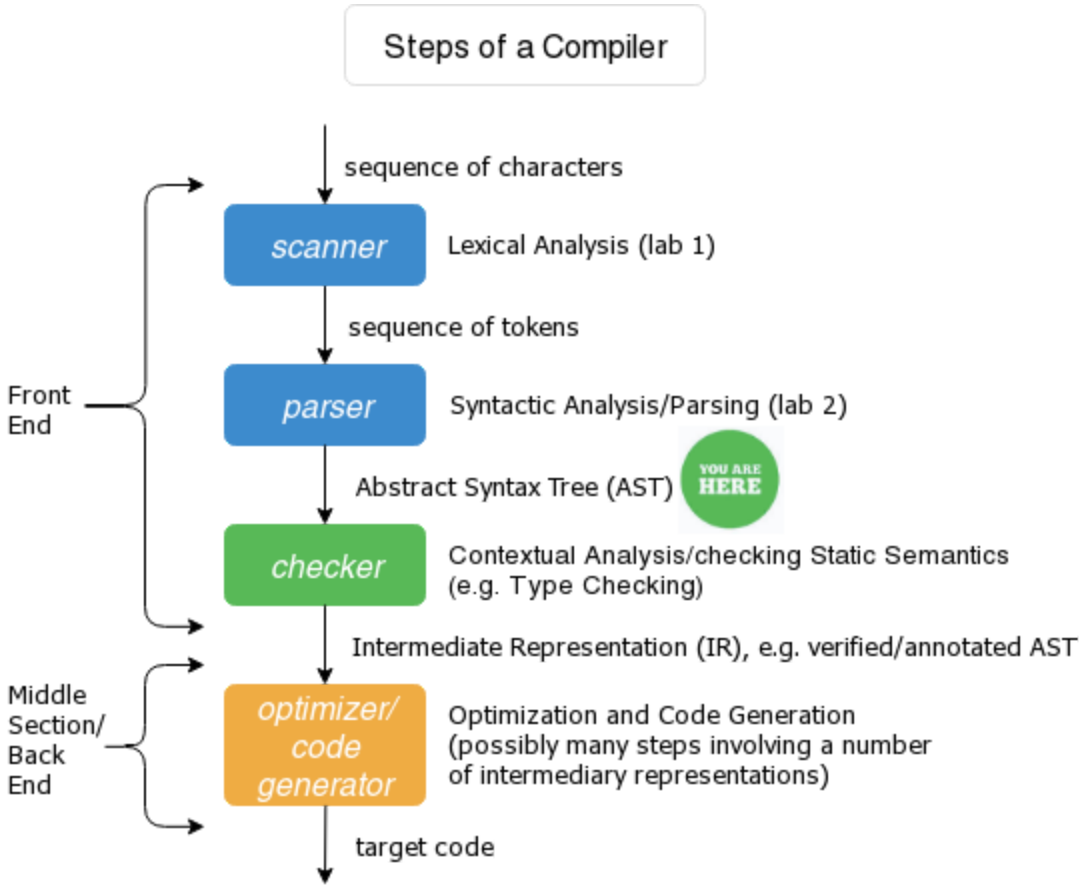
In this lab you will create an Abstract Syntax Tree (AST) for SimpleC, using the base code we provide.

Part 0

Copy the initial files of the project into your home directory.
Note that the code will be released on Thu Feb 16.

```
cd cs352
cp /homes/cs352/Spring2017/lab3-ast/lab3-src.tar .
tar -xvf lab3-src.tar
```

Part 0.5 Background Knowledge



Concrete Syntax Tree (Parse Tree) vs Abstract Syntax Tree

In lab 2, you wrote your own recursive descent parser for small infix notation expressions. You then input the SimpleC grammar into JavaCC, effectively showing it how to generate a recursive descent parser for our language. In this lab, we've given you a completed solution to lab 2, at least the SimpleC grammar part, and you'll be implementing steps to generate an AST inside the JavaCC functions.

One note to make is that the way a recursive descent parser traverses the grammar and visits rules can be thought of as a tree itself, separate from the AST. We call this tree a Parse Tree, or Concrete Syntax Tree. It has a one-to-one mapping with our grammar rules.

The Concrete Syntax Tree is very formal, and doesn't exactly help us down the line with later compiler passes. Once we understand the language, we want a better representation, and this is what the AST is for. This distinction is important, since our AST classes don't have to be one-to-one with our grammar. We've given you some code for certain classes, but anything you implement on your own can be done however you see fit, provided that the output is identical with the test cases.

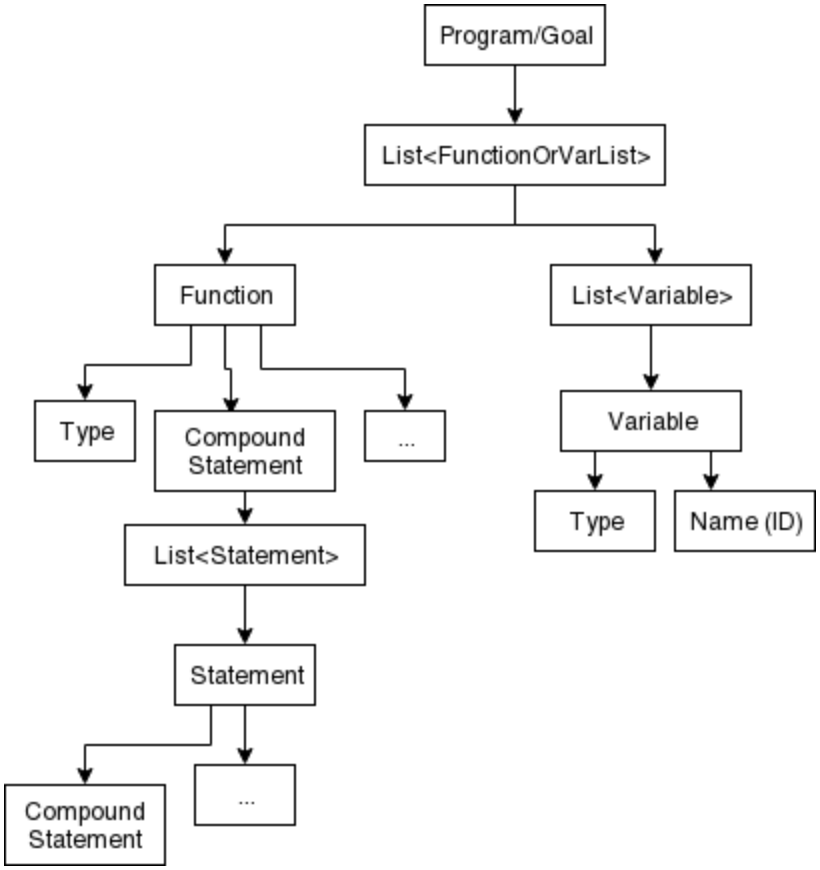
You can find more information about the difference between Parse Trees and Abstract Syntax Trees here: http://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/

Part 1 SimpleC Parse Tree Diagram

An important part of the lab is laying out the foundation of what you are implementing, and understanding it fully **before** you start to code. The first part of the lab will be to take the grammar rules and convert them to an AST representation of your choosing.

Typically, an AST/CST is the tree representation of a given source program or string. For this part of the lab, you need to make a tree containing every type of AST node, and connect them in terms of how they relate, not necessarily for a specific program.

Here is the beginning of the TA's AST representation, and is what you've been given in the code as well. Note again that you are free to represent the AST however you want. This step in the lab is just to get you to think about how each part of the grammar will be represented.



Compared with the original grammar and subsequent parse trees, there are a couple simplifications here.

1. Program and Goal have been inlined into one. It's simpler for our AST to store just one.
2. There is no global variable node type. In the original grammar, a program is a list of functions/global vars. In the AST, we have a Variable type, and a type that represents Variable Lists and Functions.

All of these are design decisions that you can choose to follow or ignore.

Feel free to do this on paper and scan/take a picture or do it online. Also, for any recursive steps in the tree, i.e. statement -> compound_statement -> statement -> … feel free to just evaluate it one level just like we did with compound statement in the above example diagram. The main point is to get you more familiar with the intricacies of the language.

Save a pdf **called ast.pdf** with your diagram and include it in your submission.

Part 2 SimpleC Language Questions

Answer these questions in a file called questions.txt and include it in your submission.

1. What are 5 limitations of the SimpleC grammar compared with C?
2. To match an expression, you need to go through the rules of parsing an expression -> logical_or_expr -> logical_and_expr -> ... -> primary_expr for many steps. What is the point of this?
3. For each of the following, state whether the program is valid (in terms of grammar rules alone, no semantic analysis) and if it is not valid, **include why**
    a.

```
int main() {
  {
    x = 5;
  }
}
```

    b.

```
long f() {
  double *x;
  x = "ok";
}
```

    c.

```
double g() {
  return "ok1";
  return "ok2";
  return "ok3";
}
```

    d.

```
char** g() {
  char* g = “gg”;
}
```

    e.

```
double* f1() {
  char* f2() {
  }
}
```

    f.

```
long k() {
  long i;
  for (;;) {} // infinite loop
}
```

Part 3 SimpleC AST

The SimpleC grammar is defined here. Note that we've changed some of it since the previous lab. You'll be given a completed SimpleC.jj file to work with for this lab.

The goal of this lab is to fully implement an Abstract Syntax Tree for SimpleC. We have given you a decent chunk of the code- your job is to fill it in. We have given you working code for:

- Functions
- Some Statements (Compound, Assignment, VariableDecls)
- Global variables
- Some expressions (||, Characters, Numbers, Paren expressions, unary)

It is suggested that you finish expressions, local variables, then finish the statements one by one. We've left notes in the code indicating what you should work on, just search for TODO in the code.

Your code should build the AST, and also define print methods for every kind of AST node, indicating how the Visitor interface will interact with your tree as it traverses. You will take in source code as input, and output the source code generated from the AST.

It may be beneficial to look into how the Visitor pattern works, and there are plenty of online resources you could look into. The Wikipedia article, and this blog, is quite good:
https://en.wikipedia.org/wiki/Visitor_pattern
https://sourcemaking.com/design_patterns/visitor

You've been given a huge test suite, along with 3 utilities for interacting with the tests.

testinput.sh will take a single string argument representing program text, and will output the AST generated code

```
λ ./testinput.sh "long f,g,h;long f(){{{double* x; x = \"ok\";}}}"
long f;
long g;
long h;
long f()
{
  {
    {
      double* x;
      x = "ok";
    }
  }
}
char** g;
```

test.sh will run a single test, and outputs whether or not you've passed it. You can add an additional argument -p to show your input, output, and the expected test case output. Running the script without any arguments will show you an example.

```
λ ./test.sh 2
t2 OK
λ ./test.sh 2 -p
***INPUT***

    1  double d;

***YOUR OUTPUT***
double d;

***EXPECTED OUTPUT***
double d;
```

testall.sh will run all of the tests and give tell you how many you've passed/failed. With the given code, the first few test cases should pass, of the 70 we have provided to you. The rest will be up to you to implement. Getting a 100% on the test cases doesn't mean you'll get 100% on the lab.

Note that this test suite doesn't include the tests from the previous labs, but they've been put in the oldtests folder if you wish to run them against your code.

You can run files from the oldtests directory with your code by going into the bin directory and running something like (from the bin dir)

```
λ cat ../oldtests/queens.c | java simplec.AST
```

Extra Credit

Adding structs and unions

Deadline

The deadline of this project is Monday February 27th at 11:59pm. To turn in your project, in the directory above your solution, run

turnin -c cs352 -p lab3 lab3-src

Make sure that the .git/ is there as well since it is used to verify the evolution of your project.

FAQ (We'll keep adding to this, also see: Piazza)

1. Why is the grammar different than what was there before?
    a. We've fixed some small problems with certain valid programs parsing, and fixed some small errors that were previously in the grammar. This is why we're giving you a filled out SimpleC.jj