ECE 468: Intro to Compilers and Translation Systems Engineering

Fall 2017

MWF, 1:30–2:20, WALC 3087

Project Step 1 — Scanner

Due: September 6th

The first *real* step of the project is building the first phase of a compiler: the scanner (sometimes called a tokenizer). A scanner's job is to convert a series of characters in an input file into a sequence of *tokens* -- the "words" in the program. So, for example, the input

```
A := B + 4
```

Would translate into the following tokens:

```
IDENTIFIER (Value = "A")
OPERATOR (Value = ":=")
IDENTIFIER (Value = "B")
OPERATOR (Value = "+")
INTLITERAL (Value = "4")
```

The way that we define tokens in a programming language is with *regular expressions*. For example, a regular expression that defines an integer literal token looks like:

`[0-9]+` (read: "1 or more digits"),

while a regular expression that defines a float literal token looks like:

`[0-9]+\.[0-9]* | \.[0-9]+` (read: "Either 1 or more digits followed by a decimal followed by 0 or more digits; or a decimal followed by 1 or more digits")

(See the notes for Lecture 2 for details).

While you can write a scanner by hand, it is very tedious. Instead, we typically use tools to help us *automatically generate* scanners. The tools we recommend you to use are either `flex` (if you're planning on writing your compiler in C or C++) or `ANTLR` (if you're planning on writing your compiler in Java). `flex` is available on most Unixes/Linux (including the ecegrid machines), while ANTLR requires a download. If you want to use other tools to generate your scanner, feel free, but we will be able to provide less help.

If you use ANTLR, you will find its API documentation. If you use `flex`, the manual is a good resource, and a working example using `flex` and its accompanying tool `bison` (which you will use in step 2) is available here

Token definitions

We will be building a compiler for a simple language called MICRO in this class. The token definitions (written in plain English) are as follows:

```
an IDENTIFIER token will begin with a letter, and be followed by any number of
letters and numbers.
IDENTIFIERS are case sensitive.

INTLITERAL: integer number
            ex) 0, 123, 678
FLOATLITERAL: floating point number available in two different format
                yyyy.xxxxxx or .xxxxxxx
            ex) 3.141592 , .1414 , .0001 , 456.98

STRINGLITERAL: any sequence of characters except '"'
```


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 Assistanta
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

```
            between '"' and '"'
            ex) "Hello world!" , "***********" , "this is a string"

COMMENT:
      Starts with "--" and lasts till the end of line
      ex) -- this is a comment
      ex) -- any thing after the "--" is ignored

Keywords

PROGRAM,BEGIN,END,FUNCTION,READ,WRITE,
IF,ELSE,FI,FOR,ROF,
RETURN,INT,VOID,STRING,FLOAT

Operators
:= + - * / = != < > ( ) ; , <= >=
```

What you need to do

You should build a scanner that will take an input file and output a list of all the tokens in the program. For each token, you should output the token type (e.g., `OPERATOR`) and its value (e.g., `+`).

There are sample inputs and outputs here. These are the only inputs we will test your compiler on. Your outputs need to match our outputs *exactly* (we will be comparing them using `diff`, though we will ignore whitespace).

Hints

Note that even though our sample outputs combine together a bunch of different tokens as a single type (e.g., all keywords have the token type `KEYWORD`), you will be better served by defining every keyword and operator as a *different* token type (so your scanner will have different tokens for, say, `:=` and `<`), and then writing a little bit of extra code to print the output we expect for that token type.

While it might seem weird, you *will* need to define a token that eats up any whitespace in your program (recall that your compiler really only sees a list of characters; it has no reason to think that a tab character isn't an important character). Make sure that when you recognize a whitespace token, you just silently drop it, rather than printing it out.

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.

You should tag your step 1 submission as `step1-submission`