# CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

---

## Parsing with Bison/Yacc

## Parsing with Bison

So we now know that parsers take the tokens output from the lexer and attempt to create parse trees.

We aren't really going to see the parse trees being made; what we are going to do is practice writing grammars.

# Parsing with Bison

A note about parsers vs lexers: hopefully you have some idea by now about the different goals of the two (building a syntax tree vs finding tokens)

It is also good to note that they are described in different, though often similar-looking, formalisms

Lexers use regular expressions, whereas parsers use CFGs. The technical details are not important here, but regular languages are a subset of context-free languages

# Parsing with Bison

Just as Flex/Lex is a lexer generator, Bison (or the older version Yacc) is a parser generator.

We give Bison a grammar file, it uses that file to customize parser code, and it outputs that parser code.

We can then compile it and use it how we wish.

## Parsing with Bison

---

With newer libraries like LLVM we can even (fairly) easily take the parsed information from Bison and compile it into executables for our language!

In this class we won't be going that far though. Instead, we are going to create more of an interpreted language.

# Parsing with Bison

Just as Flex files had three sections, Bison files have three sections as well:

- **Definitions and Declarations:** include statements, function prototypes(signatures), and any Bison options.
- **Grammar Rules:** Our grammar (CFG).
- **Additional C Code:** Any functions or code we need to customize our parser.

## Parsing with Bison

These files are going to look remarkably similar to our Flex files, except instead of regular expressions defining tokens we are going to have grammars defining our language.

So for example:

## Parsing with Bison

Suppose we wanted to create a language that recognized floating point and integer numbers, or variables. If we have proper regex described in our lexer file, we may have the following Bison file

```
%{
  #include <stdio.h>
}

void yyerror(const char *s);

%union {
  int iVal;
  float fVal;
  char* sval;
}

// Now, we provide the type information
// for our tokens (tokens are defined in)
// our Flex file.  We want the value to be
// stored from Flex in the proper data
// structure.
%token <iVal> INT
%token <fVal> FLOAT
%token <sVal> VAR

// We may have other tokens in our Flex file.
// They may not need type information, but we
// still need to let Bison know about them.
%token START
%token END
```

```
%%

program:        START statement_list END;
statement_list: statement
              | statement statement_list
              ;
statement:      INT         { printf("Found %d, an int.\b", $2); }
          |     FLOAT       { printf("Found %d, a float.\b", $2); }
          |     VAR         { printf("Found %d, a var.\b", $2); }
          ;

%%

int main(int argc, char** argv){
  yyparse();
  return 0;
}
```

## Parsing with Bison

---

This program doesn't do much right now. But it could.

You can have a look at theSubsetC project:

https://github.com/kranthikiran01/subc-compiler

Or even Ruby: https://github.com/ruby

Other projects that use Bison in some way are

- PHP
- Go
- Bash
- MySQL
- PostgreSQL
- GNU Octave
- Perl 5

Now, you try it!

When working on our Flex tokenizer we began working on a simple calculator. Let's make a real calculator now.

It should

- Add, subtract, multiply, and divide (check for 0 in denominator!) ints and floats
- Reset the total when asked
- Have three variables, A, B, and C. We should be able to assign values to them.
- Recognize an end token of some sort that causes the parsing to stop.