03/03/2010

Recitation #7

Administrative

- Assignments 1b and 2 are graded
    - see grading criteria and test/reference files on my Wiki page and course website
    - Questions about grading criteria?
    - Please don't hesitate if you think I graded something wrong
- Dr. Hughes is done grading the exam → He will comment in class.
- Withdrawal date is tomorrow (03/04/2010)

Assignment 3

- Dr. Hughes modified the assignment, so the $S \rightarrow BAB$ rule is not part of the grammar anymore
- Review of CKY (= CYK) algorithm (see extensive notes from recitation #6)
- **Example**: Input string "baabb"

	b	**a**	**a**	**b**	**b**
1	B,C	A,C	A,C	B,C	B,C
2	S,E	D	S,D	E	
3	A	A	B		
4	S,D	S,D			
5	**A**				

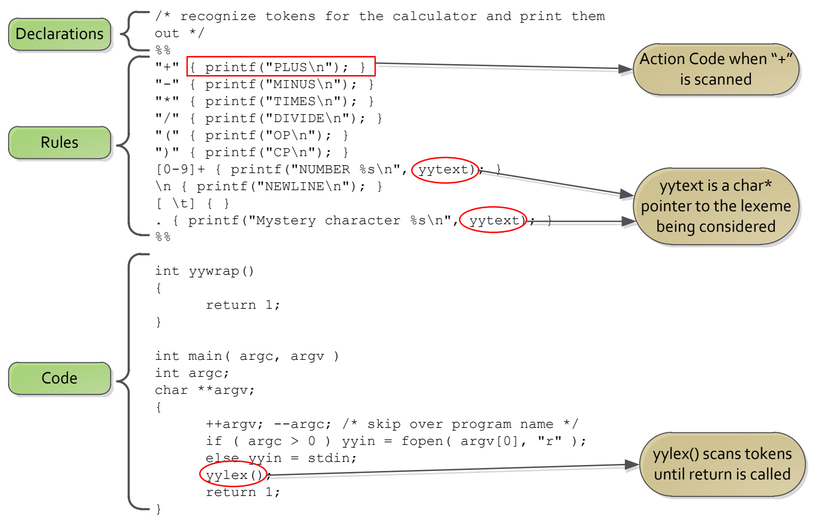
- This string is not part of the language (cannot be parsed), because the lower left cell doesn't contain the "S" non-terminal

Flex and Bison

- Flex is the scanner (lexical analyzer) and Bison is the parser (syntactic analyzer) part of the compiler
- Example of calculator grammar:
    - Recognizes simple expressions of numbers with +, -, *, /, and parentheses
    - We can design standard precedence / associativity grammar for this problem
    - $G = (\{\text{calclist, exp, factor, term}\}, \{+, -, *, /, (,), NUM\}, \text{calclist}, P)$, where P is:

$$\begin{aligned}
\text{calclist} &\rightarrow \text{calclist exp} \mid \varepsilon \\
\text{exp} &\rightarrow \text{exp + factor} \mid \text{exp - factor} \mid \text{factor} \\
\text{factor} &\rightarrow \text{factor} * \text{term} \mid \text{factor / term} \mid \text{term} \\
\text{term} &\rightarrow NUM \mid (\text{ exp })
\end{aligned}$$

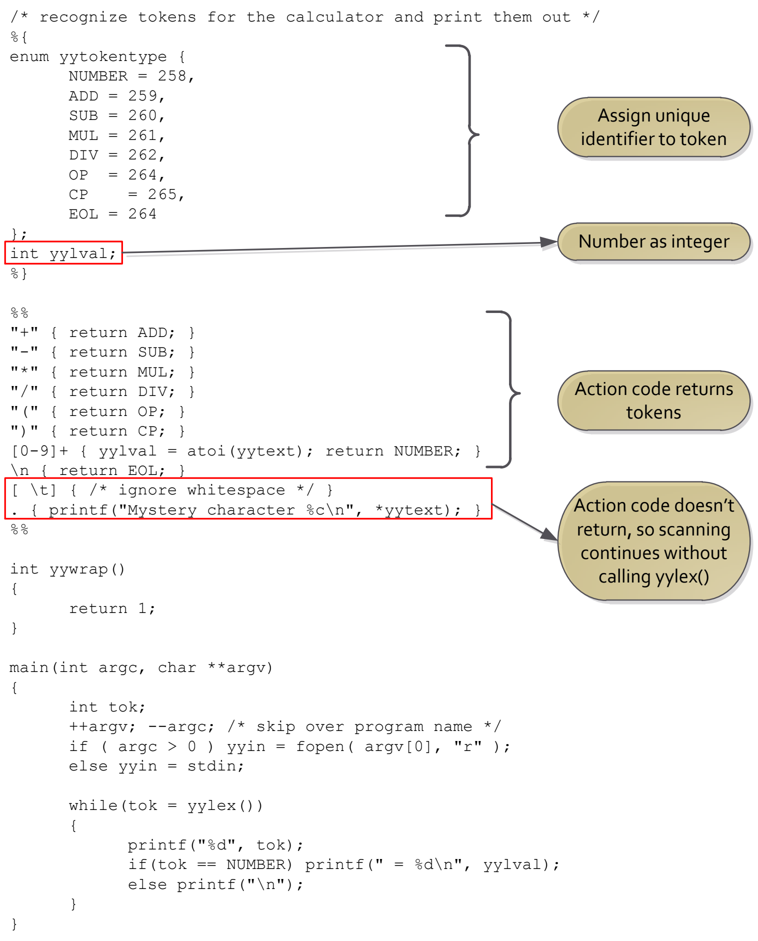
- **First try at calculator scanner with Flex** (`calculator_1.lex`)
  - Needs to recognize numbers and a bunch of special characters
  - yytext is always a pointer to the scanned lexeme
  - yylex can be called to return the next token(s), until one of the tokens returns something. If action code returns, scanning resumes on the next call to yylex(); if it doesn't return, scanning resumes immediately.
  - function definitions of yywrap() and main() necessary when compiling under Windows to avoid linker errors → under Linux linking with "-lfl" will provide defaults


Declarations
Rules
Code
Action Code when "+" is scanned
yytext is a char* pointer to the lexeme being considered
yylex() scans tokens until return is called

```
/* recognize tokens for the calculator and print them
out */
%%
"+" { printf("PLUS\n"); }
"-" { printf("MINUS\n"); }
"*" { printf("TIMES\n"); }
"/" { printf("DIVIDE\n"); }
"(" { printf("OP\n"); }
")" { printf("CP\n"); }
[0-9]+ { printf("NUMBER %s\n", yytext); }
\n { printf("NEWLINE\n"); }
[ \t] { }
. { printf("Mystery character %s\n", yytext); }
%%

int yywrap()
{
      return 1;
}

int main( argc, argv )
int argc;
char **argv;
{
      ++argv; --argc; /* skip over program name */
      if ( argc > 0 ) yyin = fopen( argv[0], "r" );
      else yyin = stdin;
      yylex();
      return 1;
}
```

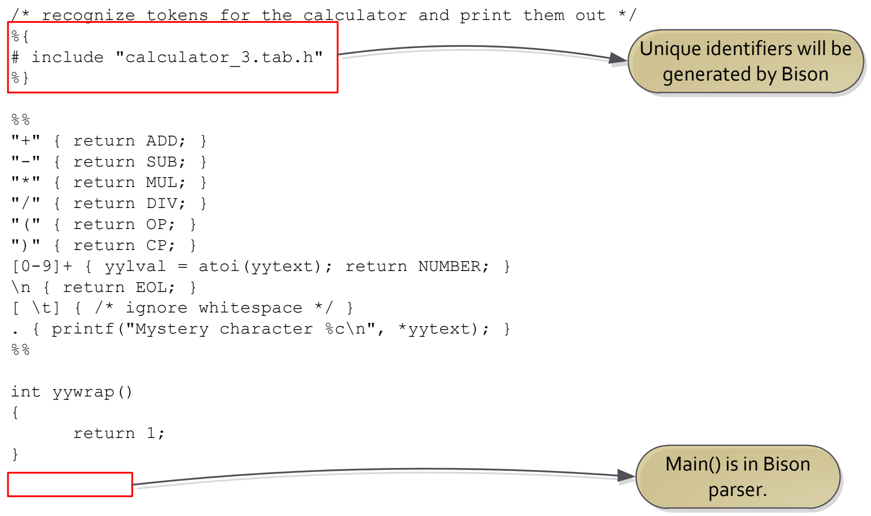
  - compile and run on some input examples

- **Return actual tokens** (`calculator_2.lex`)
  - Calculator_1 only prints the tokens, let's assign unique values to them
  - Assign actual unique values to tokens that are returned by action code, e.g. ADD = 259
  - Since action code has returns now, I have to call yylex() repeatedly
  - The pattern that matches whitespace doesn't return, so the scanner just continues to look for what comes next.
  - yylval holds actual number as integer
  - Compile and run on some input example


Assign unique identifier to token
Number as integer
Action code returns tokens
Action code doesn't return, so scanning continues without calling yylex()

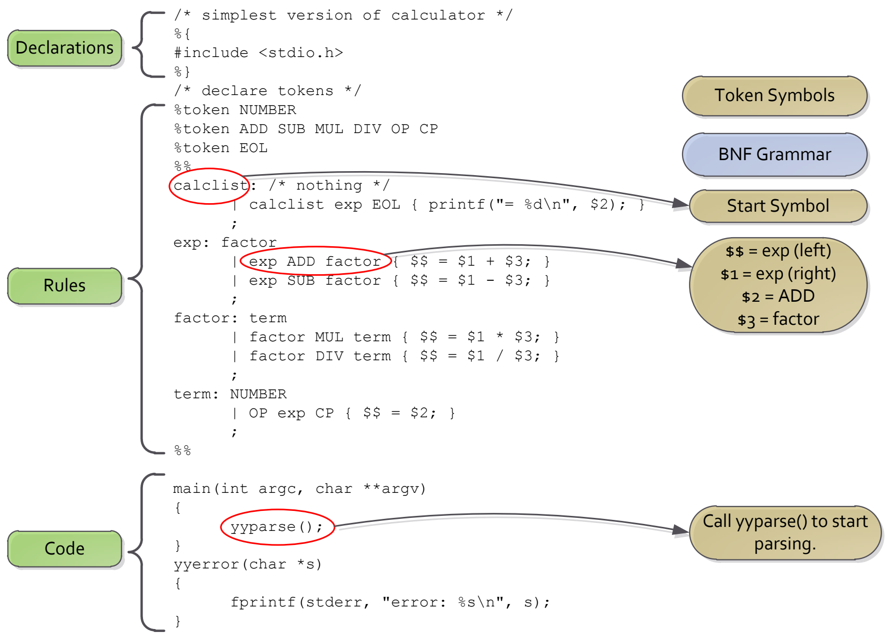
```
/* recognize tokens for the calculator and print them out */
%{
enum yytokentype {
      NUMBER = 258,
      ADD = 259,
      SUB = 260,
      MUL = 261,
      DIV = 262,
      OP  = 264,
      CP     = 265,
      EOL = 264
};
int yylval;
%}

%%
"+" { return ADD; }
"-" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
"(" { return OP; }
")" { return CP; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n { return EOL; }
[ \t] { /* ignore whitespace */ }
. { printf("Mystery character %c\n", *yytext); }
%%

int yywrap()
{
      return 1;
}

main(int argc, char **argv)
{
      int tok;
      ++argv; --argc; /* skip over program name */
      if ( argc > 0 ) yyin = fopen( argv[0], "r" );
      else yyin = stdin;

      while(tok = yylex())
      {
            printf("%d", tok);
            if(tok == NUMBER) printf(" = %d\n", yylval);
            else printf("\n");
      }
}
```

- **Make Scanner usable for Bison Parser** (`calculator_3.lex`)
  - Unique identifiers will be generated by Bison, as well as yylval variable
  - Main() is obsolete, because scanner will be directly called through Bison


Unique identifiers will be generated by Bison
Main() is in Bison parser.

```
/* recognize tokens for the calculator and print them out */
%{
# include "calculator_3.tab.h"
%}

%%
"+" { return ADD; }
"-" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
"(" { return OP; }
")" { return CP; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n { return EOL; }
[ \t] { /* ignore whitespace */ }
. { printf("Mystery character %c\n", *yytext); }
%%

int yywrap()
{
      return 1;
}
```

- **Generate Calculator Parser with Bison** (`calculator_3.y`)
  - Same three-part structure as flex (declarations, rules, code)
  - Bison uses “:” instead of “::=” for grammar rules; they are ended with “;”
  - The values of tokens are whatever was in yylval when the scanner returned the token
  - The values of other symbols are set in rules in the parser. In this parser, the values of the factor, term, and exp symbols are the value of the expression they represent.
  - In the absence of an explicit action on a rule, the parser assigns $1 to $$.




```
/* simplest version of calculator */
%{
#include <stdio.h>
%}
/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV OP CP
%token EOL
%%
calclist: /* nothing */
        | calclist exp EOL { printf("= %d\n", $2); }
        ;
exp: factor
        | exp ADD factor { $$ = $1 + $3; }
        | exp SUB factor { $$ = $1 - $3; }
        ;
factor: term
        | factor MUL term { $$ = $1 * $3; }
        | factor DIV term { $$ = $1 / $3; }
        ;
term: NUMBER
        | OP exp CP { $$ = $2; }
        ;
%%
main(int argc, char **argv)
{
        yyparse();
}
yyerror(char *s)
{
        fprintf(stderr, "error: %s\n", s);
}
```

- **Put it all together** (`calculator_3.lex` and `calculator_3.y`)
  - Execute the following commands:
    ```
    bison -d calculator_3.y
    flex -ocalculator_3.c calculator_3.lex
    ```
  - Then compile your calculator and run it (in this example through Visual Studio command prompt):
    ```
    cl calculator_3*.c
    calculator_3.exe
    ```
  - Play with some arithmetic expressions

Assignment 4

- Dr. Hughes gives you the Flex and Bison files for his WHILE language
- You can generate the parser for this language with the following commands:
  ```
  flex -owhile.lex.c While.l
  bison -d While.y
  cl /FeWhile.exe While.tab.c
  ```

- **Show output** for good and bad program
- You have the choice on how to implement the lexical analyzer:
  - Modify your assignment 1 submission
  - Use Dr. Hughes' solution to assignment 1
  - Use the Flex-generated lexical analyzer

- Show **lexical analyzer** "While.l"
  - `ECHO` is equivalent to `fprintf(yyout, "%s", yytext);`
  - Returned tokens
  - For NUM also set `yylval` with `setnum()` function → `atoi()` would work here, because we have the whole lexeme
  - For ID call `setid()` → copy name into 0-th element of symbol table, linear search for symbol, `strcmp()` will return 0 if found, so `yylval` will be > 0 if duplicate symbol; otherwise `yylval` will be set to 0
- Show **parser** "While.y"
  - `%left` means left-associative token → use `%right` for right-associative
  - `%nonassoc` implies no associativity → these token cannot be found in the same expression
  - Generates intermediate code → see `emit()` function
  - Generates symbol table → var.def rules, if not a duplicate, call `enter()` function to add to symbol table
  - Read/Write reference counter → counter in symbol table of `emit()` function
  - `error { … }` code is executed after error is encountered (and `yyerror()` was called)
  - WHILE construct: triple (code pointer) is returned on start of while statement (while.start), after whole WHILE construct is parsed ("while" grammar rule), set JUMP point to negative of saved triple → similar handling of DO

- **Biggest Task**: Implement a recursive descent parser (top-down) for the language described in the assignment
  - Can't use left-recursive grammar
  - Needs to recognize the same language
  - Needs to implement same functionality that Dr. Hughes' version has