



Question

- Where did the keywords/id come from?
 - Answer: The scanner
 - The scanner often uses regular expressions in order to help tokenize the program. The tokens that come out of the scanning phase are the terminals used in the BNF parsing phase.
 - Flex is the “fast lexical scanner” and bison is the parser we will be using



Regular Expressions in Flex

- [see the handout]
- How do I:
 - Create an id for Java?
 - Create a real number?
 - Create the keyword “for”?
 - Create a leet X?



The Format of Flex Files

```
%{
.h files, variables, and c-constructs for the program
go here
}%
Definitions go here
%%
Rules go here
%%
User subroutines go here
```



An Example – What’s it do?

```
%{
extern int yyval;
extern int sym[];
#include "calc.tab.h"
}%
%%
[a-z] { yyval = "yytext - 'a'; return VARIABLE; }
[0-9]+ { yyval = atoi(yytext); return INTEGER; }
"==" return ASSIGN;
"+" return PLUS;
"*=" return TIMES;
"\n" return NEWLINE;
[ \t ] ;
. yyerror("Invalid character");
%%
int yywrap() {
return 1;
}
```



Can You Write?

- A flex program that
 - ignores all white space and tabs
 - Prints “%s is a verb” for all verbs (you can think of)
 - Prints “%s is not a verb” for all other words and symbols



Bison

- A parser that parses BNF
 - It may be used to do semantic analysis, but this is not normally what’s done with a larger language. Normally, the parser would create a symbol table and parse tree and semantic analysis would be separate.



A Simple Bison Example

```

%{
#include <stdio.h>
%}
%token INTEGER PLUS TIMES
expr: INTEGER      { printf("INTEGER\n"); }
    | expr PLUS expr { printf("PLUS\n"); }
    | expr TIMES expr { printf("TIMES\n"); }
;
%%
int yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main() {
    yyparse();
    return 0;
}

```



Ambiguity

- Suppose we have the following ambiguous grammar:
 1. $E \rightarrow E + E$
 2. $E \rightarrow id$
 - Let's parse $x + y + z$.



A reduce-reduce conflict

- If the grammar is:
 - $E \rightarrow T$
 - $E \rightarrow id$
 - $T \rightarrow id$
 - There are two different reductions possible with id on the stack. In that case, Bison will take the first rule in the listing.
 - Of course it's better to avoid ambiguity. Often, you do not have to rewrite the whole grammar. You can specify in Bison whether operators are left or right associative, and you can define their precedence.



An Example

```

%{
#include <stdio.h>
int sym[26];
%}
%token VARIABLE ASSIGN INTEGER PLUS MINUS NEWLINE
%left PLUS
%left TIMES

```



Bison Ex. Cont'd

```

program: program statement
        | /* empty */
;
statement: expr NEWLINE { printf("%d\n", $1); }
          | VARIABLE ASSIGN expr NEWLINE
            { sym[$1] = $3; }
;
expr: INTEGER      { $$ = $1; }
    | VARIABLE     { $$ = sym[$1]; }
    | expr PLUS expr { $$ = $1 + $3; }
    | expr TIMES expr { $$ = $1 * $3; }
;
%%

```



Bison Ex. Cont'd

```

%%
int yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main() {
    yyparse();
    return 0;
}

```