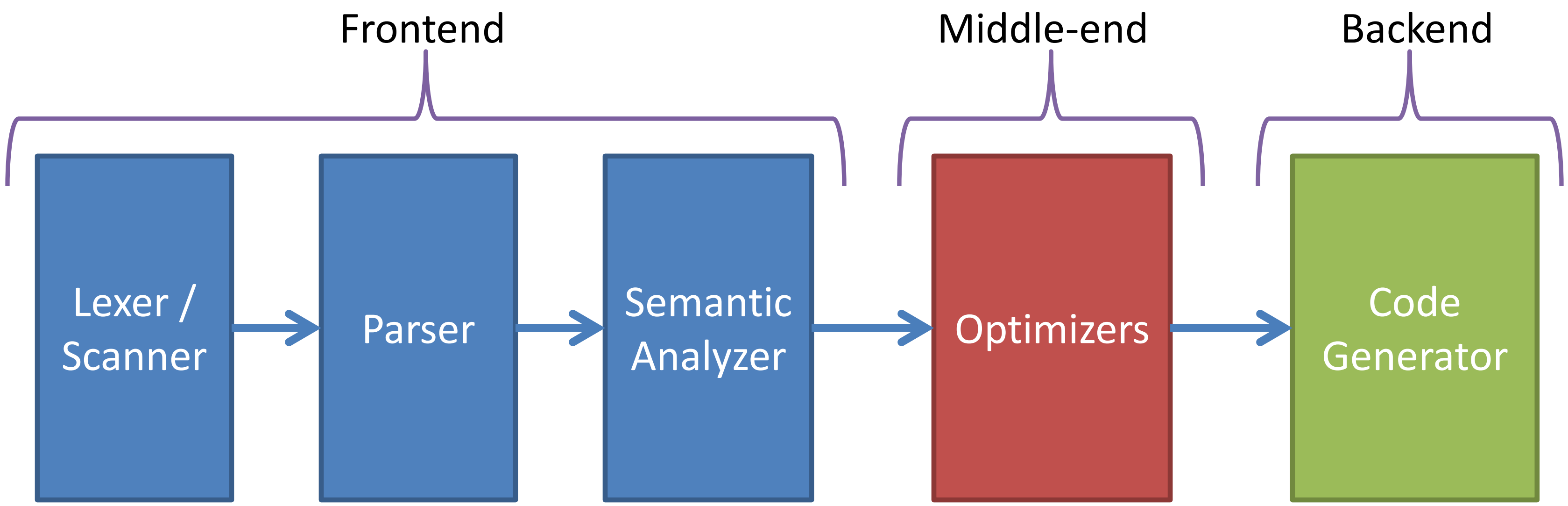
Flex/Bison Tutorial

Aaron Myles Landwehr
aron+ta@udel.edu

GENERAL COMPILER OVERVIEW

Compiler Overview



Lexer/Scanner

- Lexical Analysis
  - process of converting a sequence of characters into a sequence of tokens.

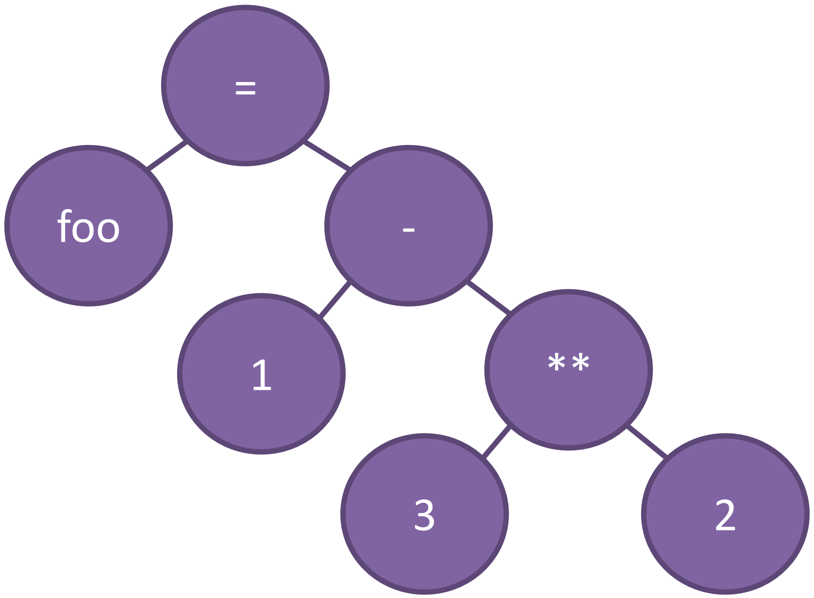
```
foo = 1 - 3**2
```

Lexeme	Token Type
foo	Variable
=	Assignment Operator
1	Number
-	Subtraction Operator
3	Number
**	Power Operator
2	Number

Parser

- Syntactic Analysis
  - The process of analyzing a sequence of tokens to determine its grammatical structure.
  - Syntax errors are identified during this stage.

Lexeme	Token Type
foo	Variable
=	Assignment Operator
1	Number
-	Subtraction Operator
3	Number
**	Power Operator
2	Number



Semantic Analyzer

- Semantic Analysis
  - The process of performing semantic checks.
    - E.g. type checking, object binding, etc.

Code:

```
float a = "example";
```

Semantic Check Error:

```
error: incompatible types in initialization
```

Optimizer(s)

- Compiler Optimizations
  - tune the output of a compiler to minimize or maximize some attributes of an executable computer program.
  - Make programs faster, etc…

Code Generator

- Code Generation
  - process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

```
int foo()
{
        return 345;
}
```

```
foo:
        addiu   $sp, $sp, -16
        addiu   $2, $zero, 345
        addiu   $sp, $sp, 16
        jr      $ra
```

LEX/FLEX AND YACC/BISON OVERVIEW

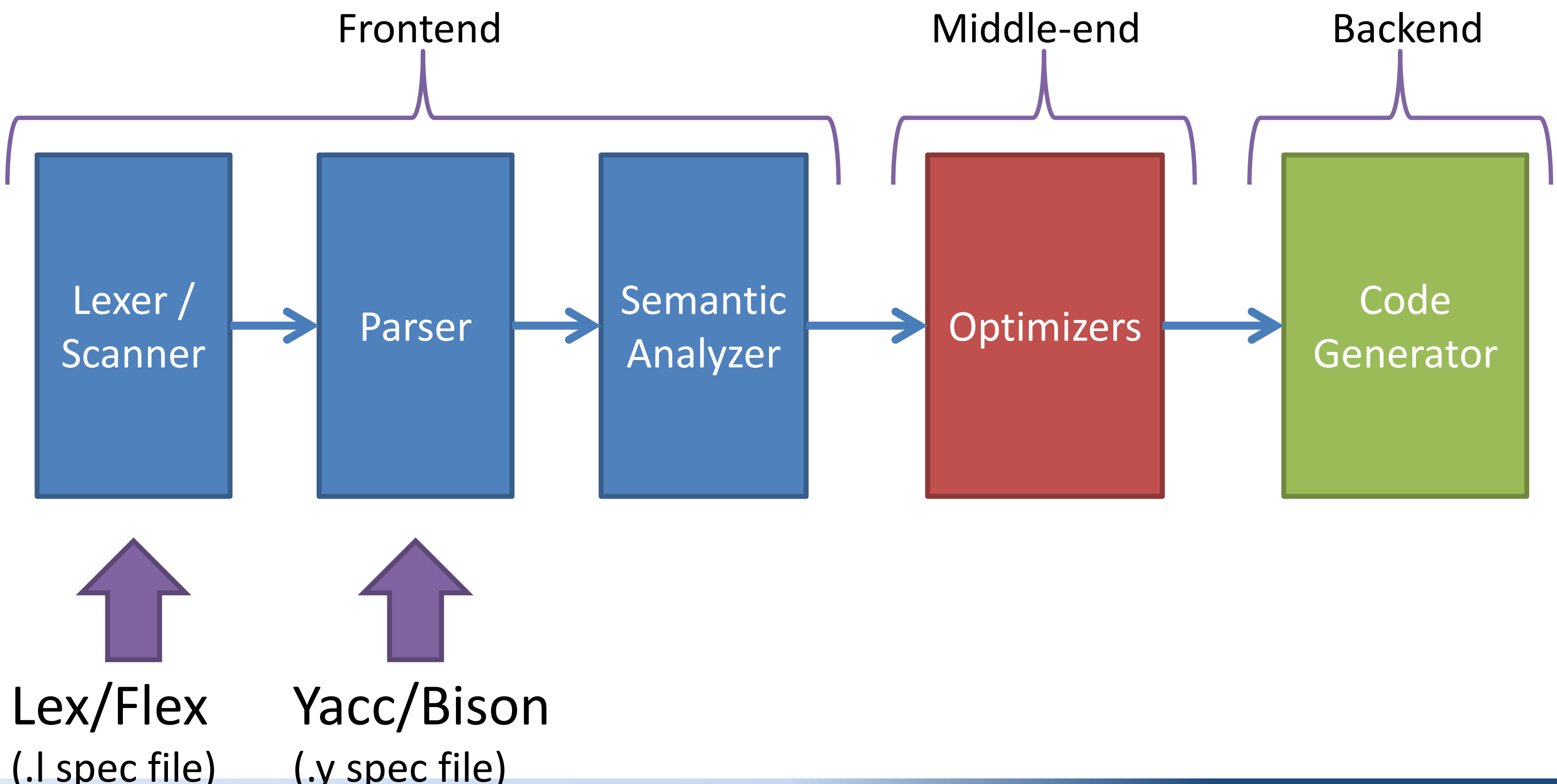
General Lex/Flex Information

- lex
  - is a tool to generator lexical analyzers.
  - It was written by Mike Lesk and Eric Schmidt (the Google guy).
  - It isn't used anymore.
- flex (fast lexical analyzer generator)
  - Free and open source alternative.
  - You'll be using this.

General Yacc/Bison Information

- yacc
  - Is a tool to generate parsers (syntactic analyzers).
  - Generated parsers require a lexical analyzer.
  - It isn’t used anymore.
- bison
  - Free and open source alternative.
  - You’ll be using this.

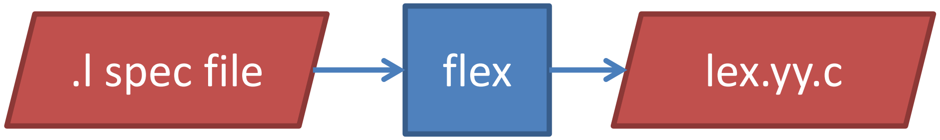
Lex/Flex and Yacc/Bison relation to a compiler toolchain



FLEX IN DETAIL

How Flex Works

- Flex uses a *.l spec file* to generate a tokenizer/scanner.



- The tokenizer reads an *input file* and chunks it into a series of *tokens* which are passed to the parser.

Flex .l specification file

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
%}

/* This tells flex to read only one input file */
%option noyywrap

%%
    /*** Rules section ***/

    /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }

.|\n    {   /* Ignore all other characters. */   }

%%
/*** C Code section ***/
```

Flex Rule Format

- Matches text input via Regular Expressions
- Returns the token type.
- Format:

```
REGEX   {
                /*Code*/
                return TOKEN-TYPE;
        }

  …
```

Flex Regex Matching Rules

- Flex matches the token with the *longest match*:
  - Input: *abc*
  - Rule: `[a-z]+`
  - ➢ Token: `abc(not "a" or "ab")`
- Flex uses the *first applicable rule*:
  - Input: *post*
  - Rule1: `"post"            { printf("Hello,");  }`
  - Rule2: `[a-zA-z]+    { printf ("World!"); }`
  - ➢ It will print Hello, (not “World!”)

Flex Example

```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Flex Example

Match one or more characters between 0-9.

```
[0-9]+  {
            /*Code*/
            yylval.dval = atof(yytext);
            return NUMBER;
        }

[A-Za-z]+  {
            /*Code*/
            struct symtab *sp = symlook(yytext);
            yylval.symp = sp;
            return WORD;
           }

.          { return yytext[0]; }
```

Flex Example

```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.          { return yytext[0]; }
```

Store the Number.

Flex Example

```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Return the token type.
Declared in the .y file.

Flex Example

```
[0-9]+  {
              /*Code*/
              yylval.dval = atof(yytext);
              return NUMBER;
        }

[A-Za-z]+  {
              /*Code*/
              struct symtab *sp = symlook(yytext);
              yylval.symp = sp;
              return WORD;
           }

.          { return yytext[0]; }
```

Match one or more alphabetical characters.

Flex Example

```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Store the text.

Flex Example

```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Return the token type.
Declared in the .y file.

Flex Example

```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Match any single character

Flex Example

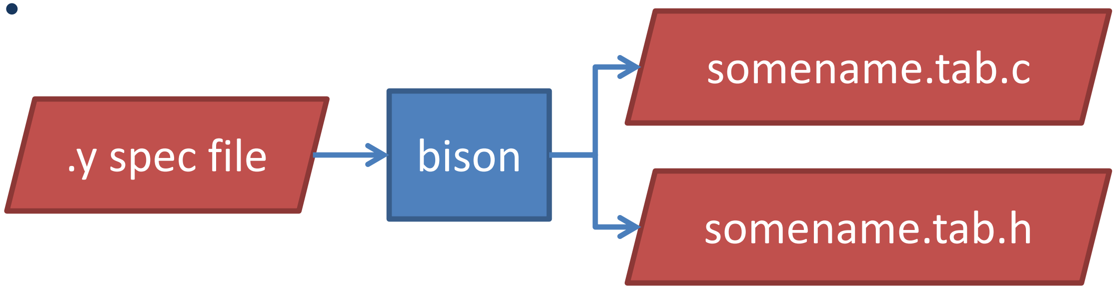
```
[0-9]+  {
                /*Code*/
                yylval.dval = atof(yytext);
                return NUMBER;
        }

[A-Za-z]+  {
                /*Code*/
                struct symtab *sp = symlook(yytext);
                yylval.symp = sp;
                return WORD;
           }

.          { return yytext[0]; }
```

Return the character. No need to create special symbol for this case.

BISON IN DETAIL

How Bison Works

- Bison uses a *.y spec file* to generate a parser.



- The parser reads a *series of tokens* and tries to determine the grammatical structure with respect to a given *grammar*.

What is a Grammar?

- A grammar
  - is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet (tokens) that are valid according to the language's syntax.

Simple Example Grammar

```
E → E + E
  → E - E
  → E * E
  → E / E
  → id
```

Above is a simple grammar that allows recursive math operations…

Simple Example Grammar

```
E → E + E
  → E - E
  → E * E
  → E / E
  → id
```

These are productions

Simple Example Grammar

$$
\begin{aligned}
\boxed{E} &\rightarrow E + E \\
&\rightarrow E - E \\
&\rightarrow E * E \\
&\rightarrow E \,/\, E \\
&\rightarrow \mathrm{id}
\end{aligned}
$$

In this case expressions (E) can be made up of the statements on the right.

*Note: the order of the right side doesn't matter.

Simple Example Grammar

```
E → E + E
  → E - E
  → E * E
  → E / E
  → id
```

How does this work when parsing a series of tokens?

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

```
E → E + E
  → E - E
  → E * E
  → E / E
  → id
```

Suppose we had the following tokens:

2 + 2 - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E - E \\
&\rightarrow E * E \\
&\rightarrow E \;/\; E \\
&\rightarrow \boxed{\mathtt{id}}
\end{aligned}
$$

We start by parsing from the left. We find that we have an **id**.

Suppose we had the following tokens:

$\boxed{2}$ + 2 - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

```
E → E + E
  → E - E
  → E * E
  → E / E
  → id
```

An **id** is an **expression**.

Suppose we had the following tokens:

2 + 2 - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

$$
\begin{aligned}
E &\rightarrow \boxed{E} + E \\
&\rightarrow E - E \\
&\rightarrow E * E \\
&\rightarrow E / E \\
&\rightarrow id
\end{aligned}
$$

Next it will match one of the rules based on the next token because the parser know **2** is an **expression**.

Suppose we had the following tokens:

2 + 2 - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

$$
\begin{aligned}
E &\rightarrow E \boxed{+} E \\
&\rightarrow E - E \\
&\rightarrow E * E \\
&\rightarrow E / E \\
&\rightarrow id
\end{aligned}
$$

The production with the **plus** is matched because it is the next token in the stream.

Suppose we had the following tokens:

2 + 2 - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

$$
\begin{aligned}
E &\rightarrow \boxed{E} + E \\
&\rightarrow E - E \\
&\rightarrow E * E \\
&\rightarrow E \,/\, E \\
&\rightarrow \boxed{id}
\end{aligned}
$$

Next we move to the next token which is an **id** and thus an **expression**.

Suppose we had the following tokens:

2 + $\boxed{2}$ - 1

Simple Example Grammar

Lexeme	Token Type
2	Number
+	Addition Operator
2	Number
-	Subtraction Operator
1	Number

$$\boxed{E} \rightarrow \boxed{E + E}$$
$$\rightarrow E - E$$
$$\rightarrow E * E$$
$$\rightarrow E / E$$
$$\rightarrow id$$

We know that **E + E** is an **expression**.

So we can apply the same ideas and move on until we finish parsing…

Suppose we had the following tokens:

$\boxed{2 + 2}$ - 1

Bison .y specification file

```
/*** Definition section ***/
%{ /* C code to be copied verbatim */ %}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'
%type <dval> expression
```

```
%%
/*** Rules section ***/
statement_list: statement '\n'
                | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
           | expression { printf("= %g\n", $1); }

expression: NUMBER
            | NAME { $$ = $1->value; }
```

```
%%
/*** C Code section ***/
```

Bison: definition Section Example

```
/*** Definition section ***/
%{
        /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Bison: definition Section Example

```
/*** Definition section ***/
%{
       /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Declaration of Tokens:
%token <TYPE> NAME

Bison: definition Section Example

Lower

Higher

```
/*** Definition section ***/
%{
        /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Operator Precedence and Associativity

Bison: definition Section Example

```
/*** Definition section ***/
%{
       /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Associativity Options:

```
%left     -        a OP b OP c
%right    -        a OP b OP c
%nonassoc -        a OP b OP c (ERROR)
```

Bison: definition Section Example

```
/*** Definition section ***/
%{
       /* C code to be copied verbatim */
%}

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'

%type <dval> expression
```

Defined non-terminal name (the left side of productions)

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

This is the grammar for bison. It should look similar to the **simple example grammar** from before.

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

What this says is that a **statement list** is made up of a **statement** OR a **statement list** followed by a **statement.**

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

The same logic applies here also. The first production is an assignment statement, the second is a simple expression.

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

This simply says that an expression is a **number** or a **name**.

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'

statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

This is an executable statement. These are found to the right of a production. When the rule is matched, it is run. In this particular case, it just says to return the value.

Bison: rules Section Example

```
/*** Rules section ***/
statement_list: statement '\n'
              | statement_list statement '\n'
                1    2      3
statement: NAME '=' expression { $1->value = $3; }
         | expression { printf("= %g\n", $1); }

expression: NUMBER
          | NAME { $$ = $1->value; }
```

The numbers in the executable statement correspond to the tokens listed in the production. They are numbered in ascending order.