Flex/Bison Tutorial

Aaron Myles Landwehr
aron+ta@.udel.edu
Compiler Overview

Frontend
- Lexer / Scanner
- Parser
- Semantic Analyzer

Middle-end
- Optimizers

Backend
- Code Generator
Lexer/Scanner

• Lexical Analysis
  – process of converting a sequence of characters into a sequence of tokens.

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>Variable</td>
</tr>
<tr>
<td>=</td>
<td>Assignment Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>3</td>
<td>Number</td>
</tr>
<tr>
<td>**</td>
<td>Power Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
</tbody>
</table>

foo = 1 - 3**2
• **Syntactic Analysis**
  – The process of analyzing a sequence of tokens to determine its grammatical structure.
  – Syntax errors are identified during this stage.

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>Variable</td>
</tr>
<tr>
<td>=</td>
<td>Assignment Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>3</td>
<td>Number</td>
</tr>
<tr>
<td>**</td>
<td>Power Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
</tbody>
</table>
Semantic Analyzer

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

Code:

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

Semantic Check Error:

```plaintext
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 Generation

- 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.

```c
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 generate 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

Frontend

Lexer / Scanner → Parser → Semantic Analyzer

Middle-end

Optimizers

Backend

Code Generator

Lex/Flex (.l spec file) → Yacc/Bison (.y spec file)
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

```c
/* 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);
 }

/** ( C Code section ***/

```
Flex Rule Format

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

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

2/17/2012
Flex Regex Matching Rules

• Flex matches the token with the *longest match*:
  – Input: \textit{abc}
  – Rule: \([a-z]+\)
    \(\text{Token: abc (not "a" or "ab")}\)

• Flex uses the *first applicable rule*:
  – Input: \textit{post}
  – Rule1: \texttt{"post" \{ printf("Hello,"\); \}}
  – Rule2: \([a-zA-Z]+\) \{ printf ("World!"\); \}
  \(\text{It will print Hello, (not “World!”)}\)
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

[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]; }
[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.
[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 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 the token type. Declared in the .y file.
  }

  {  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]; }
[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]; }
BISON IN DETAIL
How Bison Works

• Bison uses a `.y spec file` to generate a parser.

  ![Diagram](image)

  `.y spec file` ➔ `bison` ➔ `somename.tab.c`
  ➔ `somename.tab.h`

• 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.
Above is a simple grammar that allows recursive math operations...

E → E + E
→ E - E
→ E * E
→ E / E
→ id
Simple Example Grammar

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

These are productions
Simple Example Grammar

\[ E \rightarrow E + E \]
\[ E \rightarrow E - E \]
\[ E \rightarrow E \ast E \]
\[ E \rightarrow E / E \]
\[ E \rightarrow id \]

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

Suppose we had the following tokens:

2 + 2 - 1

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>

\[
E \rightarrow E + E \\
E - E \\
E * E \\
E / E \\
id
\]
Simple Example Grammar

We start by parsing from the left. We find that we have an \textbf{id}.

Suppose we had the following tokens: \[2 + 2 - 1\]

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>
Simple Example Grammar

An id is an expression.

Suppose we had the following tokens:

2 + 2 - 1

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>
### Simple Example Grammar

#### Grammar Rules

- \( E \rightarrow E + E \)
- \( E \rightarrow E - E \)
- \( E \rightarrow E * E \)
- \( E \rightarrow E / E \)
- \( E \rightarrow \text{id} \)

#### Lexeme Token Type

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>

Suppose we had the following tokens:

\[ 2 + 2 - 1 \]

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

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

Suppose we had the following tokens:

2 + 2 - 1

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

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>
Simple Example Grammar

We know that $E + E$ is an expression. So we can apply the same ideas and move on until we finish parsing...

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction Operator</td>
</tr>
<tr>
<td>1</td>
<td>Number</td>
</tr>
</tbody>
</table>

Suppose we had the following tokens:

$2 + 2 - 1$
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 ***/
2/17/2012

CAPSL 41
/* C code to be copied verbatim */

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

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

%type <dval> expression
Bison: definition Section Example

```c
/*** 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

```c
/*** Definition section ***/
{%
    /* C code to be copied verbatim */
%
%
%token <symp> NAME
%token <dval> NUMBER

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

%type <dval> expression
```

Operator Precedence and Associativity

Higher

Lower

2/17/2012
Bison: definition Section Example

```c
/* 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

```c
/*** 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

```c
/*** 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

```c
/*** 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

```c
/*** 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`.

2/17/2012
Bison: rules Section Example

```c
/*** 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.
This simply says that an expression is a **number** or a **name**.
Bison: rules Section Example

```c
/*** 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'
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.
ABOUT YOUR ASSIGNMENT
What you need to do

• You are given a prefix calculator.
  \[ + \ 2 \ 4 \]

• You need to make infix and postfix versions of the calculator.
  \[ 2 + 4 \quad 2 4 + \]

• You then need to add support for additional operators to all three calculators.
Hints

• Name your calculators “infix” and “postfix.”
• You don’t need to change the c code section of the .y.
• You may need to define new tokens for parts of the assignment.
Credit

• Wikipedia
  – Most of the content is from or based off of information from here.

• Wookieepedia
  – Nothing was taken from here.
  – Not even this picture of Chewie.

• 2008 Tutorial

*From Wikipedia: qualifies as fair use under United States Copyright law.