flex & bison

An Advanced Introduction to Unix/C Programming

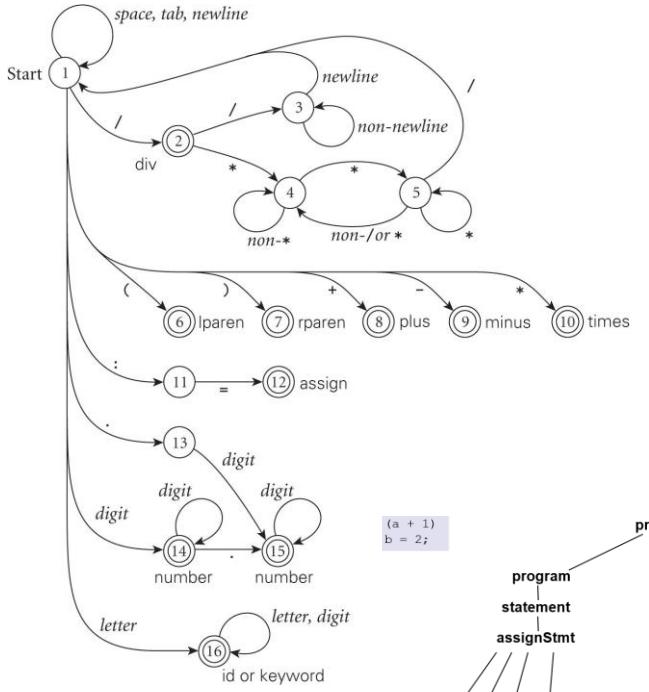
John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands

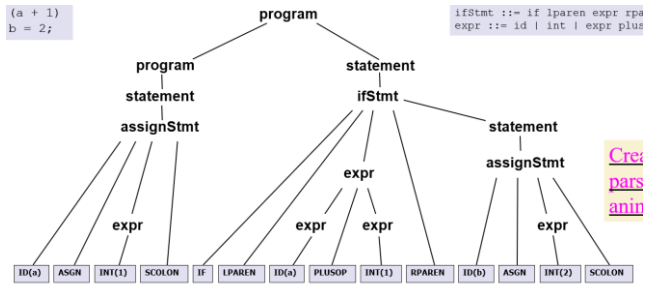
flex & bison

flex – Performs lexical analysis based on regular expressions. Outputs a token stream.

bison – Performs syntactic analysis based on context free grammars. Outputs an abstract syntax tree (AST).

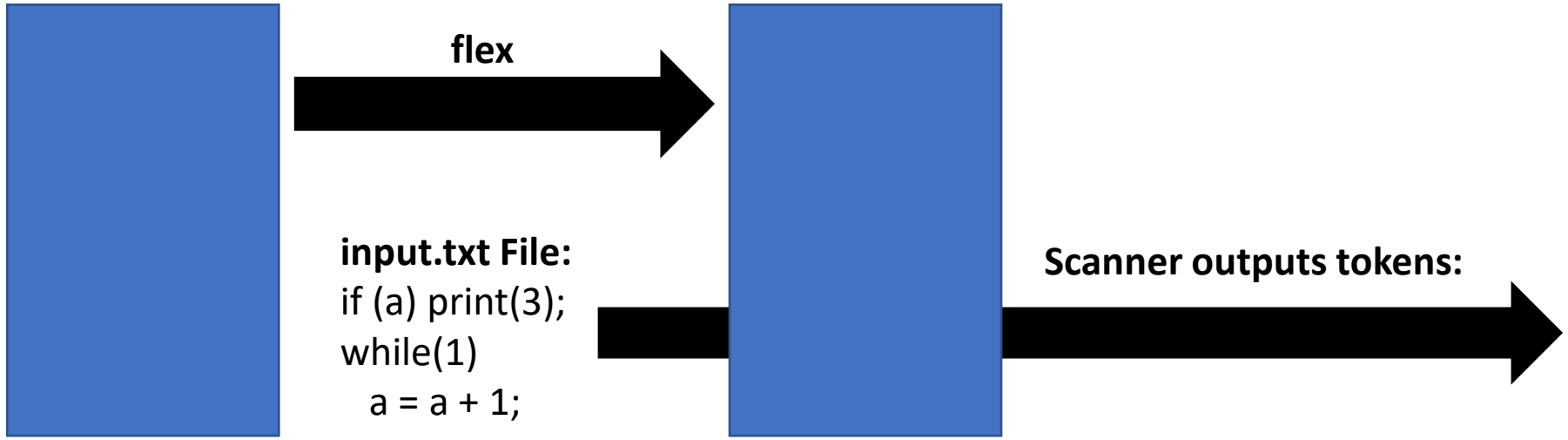
Compilation Task	Specification Mechanism	Implementation Mechanism
Lexical Analysis	Regular Expression	Deterministic Finite State Automata
Syntactic Analysis	Context-Free Grammar	Deterministic Pushdown Automata
Semantic Analysis	Attribute Grammar	Attribute Evaluators





flex – Fast Lexical Analyzer Generator

- flex is a program that writes a scanner program (i.e., a lexical analyzer program) based on regular expressions.




```
{<if> "if"}
{<lparen> "("}
{<ident> "a"}
{<rparen> ")"}
{<print> "print"}
{<lparen> "("}
{<int> "3"}
{<rparen> ")"}
{<semicolon> ";"}
{<while> "while"}
{<lparen> "("}
{<int> "1"}
{<rparen> ")"}
{<ident> "a"}
{<assign> "="}
{<ident> "a"}
{<addop> "+"}
{<int> "1"}
{<semicolon> ";"}
{<eof> ""}
```

flex Uses Regular Expressions To Define Tokens

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

flex File Format

flex files end in .l and are defined by three sections separated by %%:

Section 1: Definitions
Defines macros and imports include files.
Any C code here will be copied verbatim into the generated source file.

Section 2: Rules
Associates regular expressions patterns with C source code.
When the lexer sees text in the input that matches a given pattern, it will execute the associated C code.

Section 3: C Code
C code found here is called by the rules section and is copied verbatim into the generated source file.

flex.l Example

```
john@oho:~/flex_example$ cat flex.l
%{     /* Definition Section */
#include <stdio.h>
%}

%%     /* Rules Section */
[0-9]+     { printf("See: %s\n", yytext); }
print      { printf("See: %s\n", yytext); }
while      { printf("See: %s\n", yytext); }
;          { printf("See: %s\n", yytext); }
.|\n       { }     /* Ignore all other characters. */

%%     /* C Code Section */
int main(void)
{
       yylex();       /* Call the lexer then quit. */
       return 0;
}

void yyerror(char *string) { /* Dummy */ }
```

```
john@oho:~/flex_example$ cat dummy.y
%{
       int yylex();
       void yyerror(char *);
%}
%token EOF

%%
program: EOF

%%


john@oho:~/flex_example$ cat run
yacc -d dummy.y
lex flex.l
gcc lex.yy.c y.tab.c -ll -lm -o scanner
```

Run scanner program created by flex

```
john@oho:~/flex_example$ run

john@oho:~/flex_example$ cat input.txt
if (a) print(123);
while(1)
   a = a + 1;

john@oho:~/flex_example$ scanner < input.txt
See: print
See: 123
See: ;
See: while
See: 1
See: 1
See: ;
```

bison

- **bison is an open source parser generator.**
- **bison uses BNF notation to define a context-free grammar.**
- **bison's generated parser reads a sequence of tokens and determines if the sequence conforms to the syntax specified by the grammar.**
- **By default, bison generates an Look-Ahead Left-to-Right (LALR) parser.**
- **flex provides tokens as input to bison.**

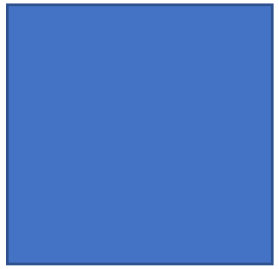
bison – Creates Parser Generator

- **bison reads a sequence of tokens defined by flex and determines if the token sequence conforms to the syntax specified by the grammar.**

lexer.l File

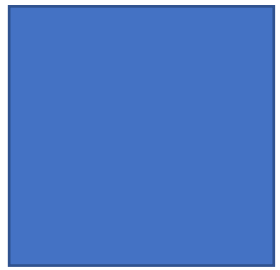
flex

lexer.c
(created by flex)



expression.h

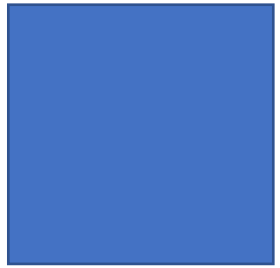
expression.c



main.c

parser.y File

bison



parser.c
(created by bison)

To compile, run make to create calc program.

```
john@oho:~/bison_wiki$ make
flex lexer.l
bison parser.y
gcc -g -ansi lexer.c parser.c expression.c main.c -o calc
```

bison File Format

bison files end in .y and are defined by three sections separated by %%:

Section 1: Definitions
Defines macros and imports include files.
Any C code here will be copied verbatim into the generated source file.

Section 2: Rules
Associates BNF grammars with C source code.
When the parser sees tokens in the input that matches a the defined context free grammar, it will execute the associated C code.

Section 3: C Code
C code found here is called by the rules section and is copied verbatim into the generated source file.

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ ls
Makefile  expression.c  expression.h  lexer.l  main.c  parser.y

john@oho:~/bison_wiki$ cat Makefile
# Makefile

FILES = lexer.c parser.c expression.c main.c
CC = gcc
CFLAGS = -g -ansi

calc: $(FILES)
	$(CC) $(CFLAGS) $(FILES) -o calc

lexer.c: lexer.l
	flex lexer.l

parser.c: parser.y lexer.c
	bison parser.y

clean:
	rm -f *.o lexer.c lexer.h parser.c parser.h calc
```

```
john@oho:~/bison_wiki$ cat lexer.l
%{
#include "expression.h"
#include "parser.h"
#include <stdio.h>
%}
%option outfile="lexer.c" header-file="lexer.h"
%option warn nodefault
%option reentrant noyywrap never-interactive nounistd
%option bison-bridge

%%
[ \r\n\t]*   { continue; /* Skip blanks. */ }
[0-9]+       { sscanf(yytext, "%d", &yylval->value);  return TOKEN_NUMBER; }
"*"          { return TOKEN_STAR; }
"+"          { return TOKEN_PLUS; }
"("          { return TOKEN_LPAREN; }
")"          { return TOKEN_RPAREN; }
.            { continue; /* Ignore unexpected characters. */}

%%
int yyerror(SExpression **expression, yyscan_t scanner, const char *msg) {
   fprintf(stderr, "Error: %s\n", msg);
   return 0;
}
```

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ cat parser.y
%{
#include "expression.h"
#include "parser.h"
#include "lexer.h"
int yyerror(SExpression **expression, yyscan_t scanner, const char *msg);
%}
%code requires {
  typedef void* yyscan_t;
}
%output  "parser.c"
%defines "parser.h"
%define api.pure
%lex-param   { yyscan_t scanner }
%parse-param { SExpression **expression }
%parse-param { yyscan_t scanner }

%union {
   int value;
   SExpression *expression;
}
%token TOKEN_LPAREN   "("
%token TOKEN_RPAREN   ")"
%token TOKEN_PLUS     "+"
%token TOKEN_STAR     "*"
```

```
%token <value> TOKEN_NUMBER "number"

%type <expression> expr

/* Precedence (increasing) and associativity:
   a+b+c is (a+b)+c: left associativity
   a+b*c is a+(b*c): the precedence of "*" is higher than that of "+". */
%left "+"
%left "*"

%%

input
   : expr { *expression = $1; }
   ;

expr
   :  expr[L] "+" expr[R]     { $$  = createOperation( eADD, $L, $R ); }
   | expr[L] "*" expr[R]      { $$  = createOperation( eMULTIPLY, $L, $R ); }
   | "(" expr[E] ")"          { $$  = $E; }
   | "number"                 { $$  = createNumber($1); }
   ;

%%
```

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ cat expression.h
#ifndef __EXPRESSION_H__
#define __EXPRESSION_H__

typedef enum tagEOperationType
{
   eVALUE,
   eMULTIPLY,
   eADD
} EOperationType;

typedef struct tagSExpression
{
   EOperationType type;              /* /< type of operation */
   int value;                        /* /< valid only when type is eVALUE */
   struct tagSExpression *left;      /* /<  left side of the tree */
   struct tagSExpression *right;     /* /< right side of the tree */
} SExpression;

SExpression *createNumber(int value);
SExpression *createOperation(EOperationType type,
                             SExpression *left, SExpression *right);
void deleteExpression(SExpression *b);

#endif /* __EXPRESSION_H__ */
```

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ cat expression.c
/* Expression.c Implementation of functions used to build the syntax tree. */
#include "expression.h"
#include <stdlib.h>
static SExpression *allocateExpression()
{
   SExpression *b = (SExpression *)malloc(sizeof(SExpression));
   if (b == NULL)
      return NULL;
   b->type = eVALUE;
   b->value = 0;
   b->left = NULL;
   b->right = NULL;
   return b;
}

SExpression *createNumber(int value)
{
   SExpression *b = allocateExpression();
   if (b == NULL)
      return NULL;
   b->type = eVALUE;
   b->value = value;
   return b;
}
```

```
SExpression *createOperation(EOperationType type,
                             SExpression *left,
                             SExpression *right)
{
   SExpression *b = allocateExpression();
   if (b == NULL)
      return NULL;
   b->type = type;
   b->left = left;
   b->right = right;
   return b;
}

void deleteExpression(SExpression *b)
{
   if (b == NULL)
      return;
   deleteExpression(b->left);
   deleteExpression(b->right);
   free(b);
}
```

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ cat main.c
#include "expression.h"
#include "parser.h"
#include "lexer.h"
#include <stdio.h>
int yyparse(SExpression **expression, yyscan_t scanner);

SExpression *getAST(const char *expr)
{
    SExpression *expression;
    yyscan_t scanner;
    YY_BUFFER_STATE state;
    if (yylex_init(&scanner)) {
        /* could not initialize */
        return NULL;
    }
    state = yy_scan_string(expr, scanner);
    if (yyparse(&expression, scanner)) {
        /* error parsing */
        return NULL;
    }
    yy_delete_buffer(state, scanner);
    yylex_destroy(scanner);
    return expression;
}
```

```
int evaluate(SExpression *e)
{
    switch (e->type) {
        case eVALUE:
            return e->value;
        case eMULTIPLY:
            return evaluate(e->left) * evaluate(e->right);
        case eADD:
            return evaluate(e->left) + evaluate(e->right);
        default:
            /* should not be here */
            return 0;
    }
}

int main(void)
{
    char test[] = " 4 + 2*10 + 3*( 5 + 1 )";
    SExpression *e = getAST(test);
    int result = evaluate(e);
    printf("Result of '%s' is %d\n", test, result);
    deleteExpression(e);
    return 0;
}
```

bison Example – From Wikipedia GNU Bison

```
john@oho:~/bison_wiki$ ls
Makefile  expression.c  expression.h  lexer.l  main.c  parser.y

john@oho:~/bison_wiki$ make
flex lexer.l
bison parser.y
gcc -g -ansi lexer.c parser.c expression.c main.c -o calc

john@oho:~/bison_wiki$ ls
Makefile  expression.c  lexer.c  lexer.l  parser.c  parser.y
calc      expression.h  lexer.h  main.c   parser.h

john@oho:~/bison_wiki$ calc
Result of ' 4 + 2*10 + 3*( 5 + 1 )' is 42
```