Principles of Programming Languages

COMP251: Lex (Flex) and Yacc (Bison)

Prof. Dekai Wu

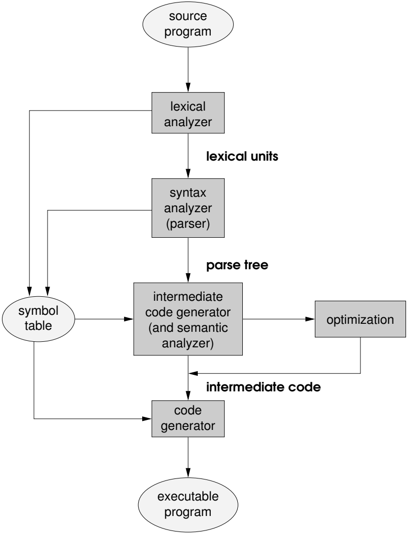
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

Fall 2006

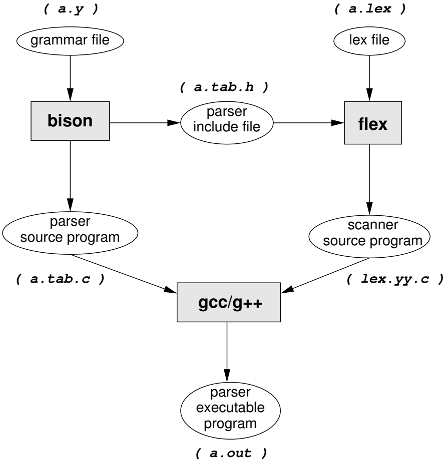
Part II

bison

Compilation (a revisit)



Syntax Analysis using bison and flex



bison : a Parser Generator

- bison is GNU's extended version of the standard UNIX utility yacc, that generates a parser for a given CFG. It is backward compatible with yacc (Yet Another Compiler Compiler), which was perhaps the first popular parser generator.
- bison reads a description of a CFG written in a bison Grammar File, and output a C program containing a routine called `yyparse()`.
- The default name of the output C program is `*.tab.c`. Compile `*.tab.c` to `a.out` which will be the parser.
- bison can only parse a subset of CFGs called LALR(1) grammars, using a bottom-up parsing algorithm with one look-ahead token.
- bison only generates a parser and does NOT provide a scanner automatically. To get both a parser and a scanner:
  - run both bison and flex
  - put the lexical analysis code in the section Additional C Code.

bison Grammar File Format

```
%{
C Declarations
%}

bison Declarations

%%
Grammar Rules + Actions
%%

Additional C Codes
```

- Similar to flex, any statements between the %{ and %}, as well as any additional C code will be copied verbatim to the output.

bison Example 1: Reverse Polish Notation Calculator

```
%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

%% /* grammar rules and actions follow */
input: /* empty */
     | input line
     ;

line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;

exp: NUM { $$ = $1; }
   | exp exp '+' { $$ = $1 + $2; }
   | exp exp '-' { $$ = $1 - $2; }
   | exp exp '*' { $$ = $1 * $2; }
   | exp exp '/' { $$ = $1 / $2; }
   | exp exp '^' { $$ = pow($1, $2); }
   | exp 'n' { $$ = -$1; }
   ;
%%
```

bison Example 1 ..

```
/* additional C code */
#include <ctype.h>
#include <stdio.h>

int yylex(void)
{
    int c;

    while ((c = getchar()) == ' ' || c == '\t') ; /* skip white spaces */

    if (c == '.' || isdigit(c)) {
        /* process numbers */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUM;
    }

    if (c == EOF) return 0;
    return c;
}

int main() { return yyparse(); }

int yyerror(const char* s) { printf("%s\n", s); return 0; }
```

bison Grammar File

1. C Declarations
   - define types and variables
   - `#define`'s and `#include`'s
2. bison Declarations
   - declares names of the terminals/non-terminals symbols
   - describe operator precedence and associativity
   - data types of semantic values of variables
3. Grammar Rules
   - production rules of the CFG
4. Additional C code
   - definition of `yylex()`
   - definition of `yyerror()` and other supporting routines

Description of bison Grammar Rules

- Three ways to represent terminals:
  1. character literals. e.g. '+' for the + operator.
  2. C string constants. e.g. "else" for the keyword **else**.
  3. C-like identifiers. e.g. NUM (for numbers). The convention is to write it in upper case.
- Non-terminals are represented as C-like identifiers. The convention is to write them in lower case.
  e.g. exp for `<Expression>`.
- Use : to represent ::=.
- A rule ends with a ';'.
- Example of a (production) rule in bison:

```
if-stmt : "if" bool-expr "then" stmt ';'
        | "if" bool-expr "then" stmt "else" stmt ';'
        ;
```

Type and Semantic Value

- Most terminals or tokens have
  1. a type
  2. a semantic value

  e.g. the integer 123 has:

type	:	INTEGER
semantic value	:	one hundred twenty-three

- But *some* terminals do NOT. e.g. operator '+'.
- Non-terminals also have semantic values. e.g.
  - the semantic value of a math expression (e.g. $E = a + b$) is a real number — result computed from its constituents.
  - the semantic value of a compiled statement is a parse tree.

Semantic Actions

- Define the semantics of a program!
- Compute the semantic value of the non-terminal on the LHS of a grammar production rule based on the semantic values of the terminals and non-terminals on the RHS of the rule.
  For example,

```
expr : expr '+' term    { $$ = $1 + $3 }
```

  where

```
$$ = semantic value of ``expr'' on the LHS.
$1 = semantic value of the 1st token on the
     RHS, which is the non-terminal ``expr''.
$3 = semantic value of the 3rd token on the
     RHS, which is the non-terminal ``term''.
```

bison Types, Variables, Functions

Entity	Meaning
YYSTYPE	macro for the token type (default: int)
extern YYSTYPE yylval	value of an input token
extern int yyparse(void)	parser function
extern int yyerror(const char*)	error reporting function

- When no action is specified, the default action is: `$$ = $1`.
- Token type code of EOF = any non-positive value (including 0).

bison Example 1 again: rpn-calc.y

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
%}

%token NUM

%% /* Grammar rules and actions follow */
input: /* empty */
     | input line
     ;
line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;
exp: NUM { $$ = $1; }
   | exp exp '+' { $$ = $1 + $2; }
   | exp exp '-' { $$ = $1 - $2; }
   | exp exp '*' { $$ = $1 * $2; }
   | exp exp '/' { $$ = $1 / $2; }
   | exp exp '^' { $$ = pow($1, $2); }
   | exp 'n' { $$ = -$1; }
   ;
%%

int main() { return yyparse(); }
int yyerror(const char* s) { printf("%s\n", s); return 0; }
```

bison Example 1 again: rpn-calc.tab.h

```
bison -d rpn-calc.y
```

produces 2 files:

- rpn-calc.tab.h: Some C declarations needed by the lex file
- rpn-calc.tab.c: Source program of the parser

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUM 257

extern YYSTYPE yylval;
```

bison Example 1 again: rpn-calc.lex

```
%option noyywrap

%{
#define YYSTYPE double      /* type for bison's var: yylval */
#include <stdlib.h>         /* for atof(const char*) */
#include "rpn-calc.tab.h"
%}

digits [0-9]
rn     (0|[1-9]+{digits}*)\.?{digits}*
op     [+*^n/\-]
ws     [ \t]+

%%
{rn}    yylval = atof(yytext); return NUM;
{op}    |
\n      return *yytext;
{ws}    /* eats up white spaces */
%%

/* There is NO main function! */
```

bison Example 2: Infix-Notation Calculator

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
%}

%token NUM
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'

%% /* Grammar rules and actions follow */
input:     /* empty */ | input line ;
line:      '\n' | exp '\n' { printf("\t%.10g\n", $1); } ;
exp: NUM { $$ = $1; }
   | exp '+' exp { $$ = $1 + $3; }
   | exp '-' exp { $$ = $1 - $3; }
   | exp '*' exp { $$ = $1 * $3; }
   | exp '/' exp { $$ = $1 / $3; }
   | '-' exp %prec NEG { $$ = -$2; }
   | exp '^' exp { $$ = pow($1, $3); }
   | '(' exp ')' { $$ = $2; } ;
%%

int main() { return yyparse(); }
int yyerror(const char* s) { printf("%s\n", s); return 0; }
```