Module 04

Pralay Mitra & P P Das

Module 04: CS31003: Compilers

Parser Generator: Bison / Yacc

Pralay Mitra
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

pralay@cse.iitkgp.ac.in
ppd@cse.iitkgp.ac.in

August 30 & 31 2021

Module Objectives

- Understand Yacc / Bison Specification
- Understand Parsing (by Parser Generators)

Module Outline

Yacc / Bison Specification

Compiler Phases

- **Lexical Analyser**: We have already discussed how to write a simple lexical analyser using Flex.
- **Syntax Analyser**: We show how to write a parser for a simple expression grammar using Bison.
- **Semantic Analyser**: We extend the parser of expression grammar semantically:
  - To build a Simple Calculator from the expression grammar (computational semantics).
  - To build a programmable calculator from the simple calculator (identifier / storage semantics).

We show how parser / translator generators can be simplified by using Ambiguous Grammar.

Bison Specs – Fundamentals

- Like Flex, has three sections – Definition, Rules, and Auxiliary
- Terminal Symbols
  - Symbolized terminals (like `NUMBER`) are identified by `%token`. Usually, but not necessarily, these are multi-character.
  - Single character tokens (like `'+'`) may be specified in the rules simply with quotes.
- Non-Terminal Symbols
  - Non-Terminal symbols (like `expression`) are identified by `%type`.
  - Any symbol on the left-hand side of a rule is a non-terminal.
- Production Rules
  - Production rules are written with left-hand side non-terminal separated by a colon (`:`) from the right-hand side symbols.
  - Multiple rules are separated by alternate (|).
  - ϵ productions are marked by empty right-hand side.
  - Set of rules from a non-terminal is terminated by semicolon (`;`).
- Start Symbol
  - Non-terminal on the left-hand side of the first production rule is taken as the start symbol by default.
  - Start symbol may be explicitly defined by `%start`: `%start statement`.

Simple Expression Parser

A Simple Expression Grammar

$$
\begin{array}{rlcl}
1: & S & \rightarrow & E \\
2: & E & \rightarrow & E + T \\
3: & E & \rightarrow & E - T \\
4: & E & \rightarrow & T \\
5: & T & \rightarrow & T * F \\
6: & T & \rightarrow & T \,/\, F \\
7: & T & \rightarrow & F \\
8: & F & \rightarrow & (E) \\
9: & F & \rightarrow & - F \\
10: & F & \rightarrow & \mathbf{num}
\end{array}
$$

Expressions involve only constants, operators, and parentheses and are terminated by a \$.

Flex Specs (calc.l) for Simple Expressions

```
%{
#include "y.tab.h" // Generated from Bison
#include <math.h>
%}

%%
[1-9]+[0-9]*    {
                        return NUMBER;
                }

[ \t]           ;  /* ignore white space */

"$"             {
                        return 0; /* end of input */
                }

\n|.            return yytext[0];
%%
```

Bison Specs (calc.y) for Simple Expression Parser

1:	S	$\rightarrow$	E
2:	E	$\rightarrow$	$E + T$
3:	E	$\rightarrow$	$E - T$
4:	E	$\rightarrow$	T
5:	T	$\rightarrow$	$T * F$
6:	T	$\rightarrow$	$T\ /\ F$
7:	T	$\rightarrow$	F
8:	F	$\rightarrow$	(E)
9:	F	$\rightarrow$	$-\ F$
10:	F	$\rightarrow$	**num**

```
%{ /* C Declarations and Definitions */
#include <string.h>
#include <iostream>
extern int yylex(); // Generated by Flex
void yyerror(char *s);
%}

%token NUMBER

%%
statement: expression
         ;
expression: expression '+' term
          | expression '-' term
          | term
          ;
```

```
term: term '*' factor
    | term '/' factor
    | factor
    ;
factor: '(' expression ')'
      | '-' factor
      | NUMBER
      ;
%%

void yyerror(char *s) { // Called on error
    std::cout << s << std::endl;
}

int main() {
    yyparse(); // Generated by Bison
}
```

Note on Bison Specs (calc.y)

- Terminal Symbols
  - Symbolized terminals (like `NUMBER`) are identified by `%token`. Usually, but not necessarily, these are multi-character. These are defined as manifest constants in `y.tab.h`
  - Single character tokens (like `'+'`) may be specified in the rules simply with quotes.
- Non-Terminal Symbols
  - Non-Terminal symbols (like `expression`) are identified by `%type`.
  - Any symbol on the left-hand side of a rule is a non-terminal.
- Production Rules
  - Production rules are written with left-hand side non-terminal separated by a colon (`:`) from the right-hand side symbols.
  - Multiple rules are separated by alternate (`|`).
  - ϵ productions are marked by empty right-hand side.
  - Set of rules from a non-terminal is terminated by semicolon (`;`).
- Start Symbol
  - Non-terminal on the left-hand side of the first production rule is taken as the start symbol by default.
  - Start symbol may be explicitly defined by `%start`: `%start statement`.

Simple Calculator

A Simple Calculator Grammar

$$
\begin{array}{rlcl}
1: & S & \rightarrow & E \\
2: & E & \rightarrow & E + T \\
3: & E & \rightarrow & E - T \\
4: & E & \rightarrow & T \\
5: & T & \rightarrow & T * F \\
6: & T & \rightarrow & T \ / \ F \\
7: & T & \rightarrow & F \\
8: & F & \rightarrow & (E) \\
9: & F & \rightarrow & - F \\
10: & F & \rightarrow & \textbf{num}
\end{array}
$$

- We build a calculator with the simple expression grammar
- Every expression involves only constants, operators, and parentheses and are terminated by a $
  - Need to bind its *value* to a *constant* (terminal symbol)
  - Need to bind its *value* to an *expression* (non-terminal symbol)
- On completion of parsing (and processing) of the expression, the evaluated value of the expression should be printed

Bison Specs (calc.y) for Simple Calculator

```
%{ /* C Declarations and Definitions */
#include <string.h>
#include <iostream>
extern int yylex();
void yyerror(char *s);
%}

%union { // Placeholder for a value
    int intval;
}

%token <intval> NUMBER

%type <intval> expression
%type <intval> term
%type <intval> factor

%%
statement: expression
               { printf("= %d\n", $1); }
         ;
expression: expression '+' term
                { $$ = $1 + $3; }
          | expression '-' term
                { $$ = $1 - $3; }
          | term
          ;
```

```
term: term '*' factor
          { $$ = $1 * $3; }
    | term '/' factor
          { if ($3 == 0)
              yyerror("divide by zero");
            else $$ = $1 / $3;
          }
    | factor
    ;
factor: '(' expression ')'
            { $$ = $2; }
      | '-' factor
            { $$ = -$2; }
      | NUMBER
      ;
%%

void yyerror(char *s) {
    std::cout << s << std::endl;
}

int main() {
    yyparse();
}
```

Note on Bison Specs (calc.y)

- Attributes
  - Every terminal and non-terminal has an (optional) attribute.
  - Multiple types of attributes are possible. They are bundled in a `C` `union` by `%union`.
  - An attribute is associated with a terminal by the `%token`: `%token <intval> NUMBER`
  - An attribute is associated with a non-terminal by the `%type`: `%type <intval> term`
- Actions
  - Every production rule has an action (`C` code snippet) at the end of the rule that fires when a reduction by the rule takes place.
  - In an action the attribute of the left-hand side non-terminal is identified as `$$` and the attributes of the symbols on the right-hand side are identified as `$1`, `$2`, `$3`, ... counting from left to right.
  - Missing actions for productions with single right-hand side symbol (like `factor` → `NUMBER`) imply a default action of copying the attribute (should be of compatible types) from the right to left: `{ $$ = $1 }` .

Header (y.tab.h) for Simple Calculator


```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers
      know about them.  */
   enum yytokentype {
     NUMBER = 258
   };
#endif
/* Tokens.  */
#define NUMBER 258

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE
{
/* Line 2068 of yacc.c  */
#line 8 "calc.y"

int intval;

/* Line 2068 of yacc.c  */
#line 62 "y.tab.h"
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Note on Header (y.tab.h)

- `y.tab.h` is generated by Bison from `calc.y` to specify the token constants and attribute type.
- `y.tab.h` is automatically included in `y.tab.c` and must be included in `calc.l` so that it can feature in `lex.yy.c`.
- Symbolized tokens are enumerated beyond 256 to avoid clash with ASCII codes returned for single character tokens.
- `%union` has generated a `C union YYSTYPE` .
- Line directives are used for cross references to source files. These help debug messaging. For example:

  ```
  #line 8 "calc.y"
  ```
- `yylval` is a pre-defined global variable of `YYSTYPE` type.

  ```
  extern YYSTYPE yylval;
  ```

  This is used by `lex.yy.c`.

Flex Specs (calc.l) for Calculator Grammar

```
%{
#include "y.tab.h" // Bison generated file of token symbols and attributes
#include <math.h>
%}

%%
[1-9]+[0-9]*    {
                        yylval.intval = atoi(yytext); // yylval denotes the attribute
                                                      // of the current symbol
                        return NUMBER;
                }

[ \t]           ;  /* ignore white space */

"$"             {
                        return 0; /* end of input */
                }

\n|.            return yytext[0];
%%
```

Note on Flex Specs (calc.l)

- `y.tab.h` is automatically included in `y.tab.c` and must be included in `calc.l` so that it can feature in `lex.yy.c`.
- `yylval` is a pre-defined global variable of `YYSTYPE` type. So attributes of terminal symbols should be populated in it as appropriate. So for `NUMBER` we have:

  ```
  yylval.intval = atoi(yytext);
  ```

  Recall, in `calc.y`, we specified:

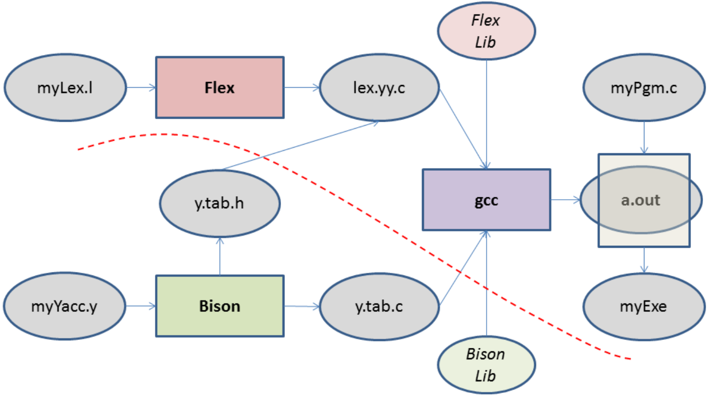
  ```
  %token <intval> NUMBER
  ```

  binding `intval` to `NUMBER`.
- Note how

  ```
  \n|.            return yytext[0];
  ```

  would return single character operators by their ASCII code.
- Newline is not treated as a white space but returned separately so that calc.y can generate error messages on line numbers if needed (not shown in the current example).

Flex-Bison Flow & Build Commands




```
$ flex calc.l
$ yacc -dtv calc.y
$ g++ -c lex.yy.c
$ g++ -c y.tab.c
$ g++ lex.yy.o y.tab.o -lfl
```

Sample Run

```
$ ./a.out
12+8 $
= 20
$ ./a.out
12+2*45/4-23*(7+1) $
= -150
```

Handling of 12+8 $

- In the next slide we show the working of the parser on the input:
  `12 + 8 $`
- We use a pair of stacks – one for the grammar symbols for parsing and the other for keeping the associated attributes.
- We show the snapshot on every reduction (skipping the shifts).

Handling of 12+8 $

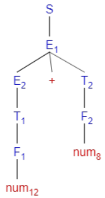
Grammar

1:	S	$\rightarrow$	E	{ printf("= %d\n", $1); }
2:	E	$\rightarrow$	$E + T$	{ $$ = $1 + $3; }
3:	E	$\rightarrow$	$E - T$	{ $$ = $1 – $3; }
4:	E	$\rightarrow$	T	{ $$ = $1; }
5:	T	$\rightarrow$	$T * F$	{ $$ = $1 * $3; }
6:	T	$\rightarrow$	T / F	{ $$ = $1 / $3; }
7:	T	$\rightarrow$	F	{ $$ = $1; }
8:	F	$\rightarrow$	(E)	{ $$ = $2; }
9:	F	$\rightarrow$	$- E$	{ $$ = –$2; }
10:	F	$\rightarrow$	**num**	{ $$ = $1; }

Reductions

$\underline{\mathbf{num}_{12}} + \mathbf{num}_8\ \$$
$\Rightarrow \underline{F} + \mathbf{num}_8\ \$$
$\Rightarrow \underline{T} + \mathbf{num}_8\ \$$
$\Rightarrow E + \underline{\mathbf{num}_8}\ \$$
$\Rightarrow E + \underline{F}\ \$$
$\Rightarrow \underline{E + T}\ \$$
$\Rightarrow \underline{E}\ \$$
$\Rightarrow S\ \$$

Parse Tree



Stack

num	12

||

F	12

||

T	12

||

E	12

||

num	8
+	
E	12

Stack

F	8
+	
E	12

||

T	8
+	
E	12

||

E	20

||

S	

||

Output || || || = 20 ||