BITS Pilani, Hyderabad Campus
Department of Computer Science and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Lab-8: More on Bison / Yacc

1 Objectives

The objectives of this lab sheet are given below.

- Implement and disambiguate arithmetic expression parsers using Bison's precedence and associativity.
- Resolve shift/reduce conflicts in Bison using precedence declarations.
- Apply attributed grammars in Bison to perform semantic analysis on binary strings (counting, decimal conversion).
- Extend attributed grammars to handle floating-point binary conversion.

2 Precedence and Associativity

The following `CFG` evaluates simple arithmetic expressions. The Lex program matches numbers and operators, returning them while ignoring white space. It returns newlines and gives an error message for anything else (this problem is seen in Lab sheet 7).

```
expr:   expr + mulex {$$ = $1 + $3;}
    | expr - mulex {$$ = $1 - $3;}
    | mulex {$$ = $1;}
    ;
mulex: mulex * term {$$ = $1 * $3;}
    | mulex / term {$$ = $1 / $3;}
    | term {$$ = $1;}
    ;

term: ( expr ) {$$ = $2;}
    | INTEGER {$$ = $1;}
    ;
```

The above example had separate rules for addition/subtraction and multiplication/division. We could simplify the grammar by writing:

```
expr: expr + expr {$$ = $1 + $3;}
    | expr - expr {$$ = $1 - $3;}
    | expr * expr {$$ = $1 * $3;}
    | expr / expr {$$ = $1 / $3;}
    | ( expr ) {$$ = $2;}
    | INTEGER {$$ = $1;}
    ;
```

However, this grammar is ambiguous. For example, the expression `1+2*3` would be evaluated as `(1+2)*3`, yielding 9 instead of the correct result, 7.

To resolve this ambiguity, we use Bison's precedence and associativity declarations:

```
%left + -
%left * /
%right ^   /* not in the previous example */
```

These declarations define precedence levels, with later lines having higher precedence. Additionally, they specify associativity:

- The operators $+$ and $-$ are left-associative, ensuring that $5-1-2$ evaluates as $(5-1)-2 = 2$.
- The operators $*$ and $/$ are also left-associative.
- The exponentiation operator `^` (if included) is right-associative, ensuring that 2^3 is evaluated as $2^{(2^3)} = 256$ rather than $(2^2)^3 = 64$.

Using precedence and associativity rules allows Bison to correctly parse arithmetic expressions without ambiguity.

Task 1 Write a Bison program to evaluate an arithmetic expression using the above ambiguous grammar. Add power operator `^` to the grammar.

2.1 Handling Unary Operators

Unary operators such as negation ('-') need special handling in Bison. If we define '-' as left-associative along with '+', it may interfere with unary negation. A common approach is to give unary '-' higher precedence:

```
%right UMINUS
```

Then, we define a separate rule for unary minus:

```
expr: - expr %prec UMINUS { $$ = -$2; }
```

The `%prec UMINUS` directive ensures that unary minus is parsed with higher precedence than other binary operators, preventing ambiguities like -3^2 from being incorrectly evaluated as $(-3)^2$ instead of $-(3^2) = -9$.

Task 2 Write a Bison program to evaluate an arithmetic expression containing a unary operator (along with other binary operators) using the above ambiguous grammar (you need to add a new rule to the grammar).

2.2 Using Precedence to Resolve Shift/Reduce Conflicts

In cases where Bison encounters shift/reduce conflicts due to ambiguous expressions, operator precedence can resolve them. For instance, an `if-else` ambiguity can be resolved using precedence:

```
%nonassoc IFX
%nonassoc ELSE
```

This ensures that an `else` clause always associates with the nearest `if`, following standard programming language conventions.

Bison can parse complex expressions efficiently and unambiguously by carefully assigning precedence and associativity to operators and rules.

```
//Bison Code

%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);
void yyerror(const char *s);

%}

%token IF ELSE ID NUMBER NL
%nonassoc IFX
%nonassoc ELSE

%%
sentence : stmt NL {printf("Valid if"); return 0;}
 ;

stmt:
    IF expr stmt %prec IFX
    | IF expr stmt ELSE stmt
    | ID '=' expr ';'
    | '{' stmt_list '}'
    ;

stmt_list:  stmt
    | stmt_list stmt
    ;

expr: NUMBER
    | ID
    ;

%%

int main() {

    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

```
//Flex code

%{
#include ".tab.h" /* Include the header file generated by Bison */
#include <stdlib.h>
#include <stdio.h>
%}

%%

"if" { return IF; }
"else" { return ELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return ID; }
[0-9]+ { return NUMBER; }
"=" { return '='; }
";" { return ';'; }
"{" { return '{'; }
"}" { return '}'; }
[ \t] { /* Ignore whitespace */ }
\n return NL;
. { fprintf(stderr, "Invalid character: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}
```

3 Attributed grammar evaluation

Attributed grammars extend context-free grammars by associating attributes with grammar symbols. These attributes can carry information about the symbols, enabling computations and semantic analysis during parsing. Evaluating these attributes allows us to perform tasks beyond simple syntax analysis, such as counting, conversion, and type checking. This section explores the concept of attributed grammar evaluation using the simple binary string grammar $L \to LB \mid B$ and $B \to 0 \mid 1$.

We will demonstrate how to attach attributes to the non-terminal symbols L and B to count the number of ones and zeros in a binary string, convert a binary string to its decimal equivalent, and even handle floating-point binary strings. Through practical examples using Bison, we will illustrate how to define and evaluate these attributes to achieve specific semantic goals.

Example 1 Write a program to count the number of 1s in a binary string generated by CFG:

```
//Bison code: save the code with name zeros.y
%{
#include <stdio.h>
#include <stdlib.h>
%}

%union {
    int count;
}

%token ZERO ONE NL
%type <count> L B

%%

S: L NL { printf("Number of 0's: %d\n", $1); exit(0);}
  ;

L: L B {  $$ = $1 + $2;}
 | B {   $$ = $1;}
 ;

B: ZERO {    $$ = 1; }
| ONE {    $$ = 0; }
;

%%

int main() {
    printf("Give the binary string: ");
    yyparse();
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 1;
}

//Flex code

%{
#include "zeros.tab.h"
%}

%%
0       { return ZERO; }
1       { return ONE; }
\n      { return NL; }
[ \t]   ; /* ignore whitespace */
.       { yyerror("Invalid character"); }

%%
```

Task 3 Use the grammar $L \to LB \mid B$ and $B \to 0 \mid 1$ that generates all the binary strings.

Write a program to convert the binary string generated by CFG to an equivalent decimal:

Example 2 Suppose we want to count zeros and ones using attributes attached to nodes (symbols). In this case, we need to maintain two attributes, say `zc` and `oc` where `zc` denotes the count of zero's and `oc` denotes the count of one's in the parse tree rooted at the non-terminal.

- $L.zc$ (integer): Count of zeros in L
- $L.oc$ (integer): Count of ones in L
- $B.zc$ (integer): Count of zeros in B
- $B.oc$ (integer): Count of ones in B

Semantic Rules

1. $S \to L$ NL
   - Print "Number of zeros: $L.zc$, Number of ones: $L.oc$"
2. $L_1 \to L_2 B$
   - $L_1.zc = L_2.zc + B.zc$
   - $L_1.oc = L_2.oc + B.oc$
3. $L \to B$
   - $L.zc = B.zc$
   - $L.oc = B.oc$
4. $B \to 0$
   - $B.zc = 1$
   - $B.oc = 0$
5. $B \to 1$
   - $B.zc = 0$
   - $B.oc = 1$

This attributed grammar and its semantic rules effectively perform the counting of zeros and ones during the parsing process. A Bison implementation is given below:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
%}

%union {
    struct { int zc; int oc; }t;
}

%token ZERO ONE NL
%type <t> L B

%%

S: L NL {
    printf("Number of zeros: %d, Number of ones: %d\n", $<t.zc>1, $<t.oc>1);
    exit(0);
};

L:L B {$<t.zc>$=$<t.zc>1+$<t.zc>2; $<t.oc>$=$<t.oc>1+$<t.oc>2;}
  | B {$<t.zc>$=$<t.zc>1;$<t.oc>$=$<t.oc>1;}
  ;

B: ONE {$<t.zc>$=0;$<t.oc>$=1;}
 | ZERO {$<t.zc>$=1;$<t.oc>$=0;}
 ;

%%

int main() {
    printf("Give the binary string: ");
    yyparse();
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 1;
}
```

Task 4 The following grammar generates all the binary numbers.

$$L \to BL \mid B \text{ and } B \to 0 \mid 1$$

(a) Write a Bison program to convert the binary string generated by CFG to an equivalent decimal.

(b) Write a Bison program to count the number of zeros and ones in the given binary number using the above grammar.

Task 5 The following grammar generates all the floating binary numbers.

$$N \to L \cdot L, \quad L \to LB \mid B \text{ and } B \to 0 \mid 1$$

Write a Bison program to convert the binary string generated by CFG to an equivalent decimal.

4 Extra problems

Task 1: Parsing and Evaluating Boolean Expressions

- Extend the parser to evaluate Boolean expressions.
- Support operators: AND (&&), OR (||), NOT (!).
- Allow nested expressions with parentheses.

Example:

```
Input:  (1 && 0) || !0
Output: 1
```

Task 2: Mini Language for Variable Assignments

- Implement a symbol table for variable storage.
- Support variable assignment and retrieval.
- Allow basic arithmetic operations on variables.
- Print the values of all the variables in the input program.

Example:

```
Input:
    a = 10;
    b = a + 5;
    c = b * 2;
Output:
    a=10
    b=15
    c = 30
```