Formal Languages and Compilers Laboratory

Syntactic Analysis: Bison

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

Syntax

“The study of the rules whereby words or other elements of sentence structure are combined to form grammatical sentences.”

The American Heritage Dictionary

Purpose of Syntactic Analysis

What syntax is valid or not is defined by the **grammar**.
A syntactic analysis must:

- identify grammar structures
- verify syntactic correctness
- build a (possibly unique) derivation tree for the input

Syntactic analysis does **not** determine the **meaning** of the input!

- That is the task of the **semantic** analysis

The syntactic analysis is performed over a stream of **terminal symbols**:

- terminal symbol $=$ **token** produced typically by the lexer
- nonterminal symbols are only generated through **reduction** of grammar rules

`bison`: The GNU Parser Generator

The standard tool to **generate** LR parsers:

- YACC compatible
- designed to work seamlessly together with `flex`
- generated parser uses **LALR(1)** methodology
  - variant of **LR(1)**, of which **ELR(1)** is another variant...

The generated parser implements a table driven push-down automaton:

- the pilot automaton is described as finite state automaton
- the parsing stack is used to keep the parser state at runtime
- acts as a typical *shift-reduce* parser

Contents

File format

A bison file is structured in four sections:

prologue	useful place where to put header file inclusions, variable declarations
definitions	definition of tokens, operator precedence, non-terminal types
rules	grammar rules
user code	C code (generally helper functions)

```
%{
Prologue
%}
Definitions
%%
Rules
%%
User code
```

Same structure as a **flex** file!

Tokens

Different syntactic elements can be defined: f.i., tokens, operator precedence, representation types for non-terminal symbols, language axiom

```
%token IF ELSE WHILE DO FOR
```

In the generated parser each token is assigned a number

- in this way you can use them in the lexer

```
enum {
  /* ... */
  IF = 258,
  ELSE = 259,
  WHILE = 260,
  /* ... */
}
```

Grammar rules

Grammar rules are specified in **BNF** notation.

If not specified, the l.h.s. of the first rule is the **axiom**.

Example: simple parser for configuration files

```
sections : sections section
         | /* empty */
         ;

section  : LSQUARE ID RSQUARE options
         ;

options  : options option
         | option
         ;

option   : ID EQUALS NUMBER
         | ID EQUALS STRING
         ;
```

Semantic actions

Just like `flex`, `bison` allows to specify **semantic actions** in grammar rules:

- a semantic action is a conventional **C code block**
- a semantic action can be specified at the end of each rule alternative

Example

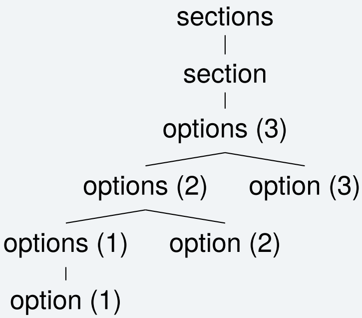
```
option : ID EQUALS NUMBER
         {
            /* associate the number with
             * the specified key */
         }
       | ID EQUALS STRING
         {
            /* same but for string values */
         }
       ;
```

Semantic actions

Semantic actions are executed when the rule they are associated with **has been completely recognized**

- **Consequence:** the order of execution of the actions is **bottom-up** (with respect to the syntactic tree)

Syntactic tree (AST)



Execution order

1. option (1)
2. options (1)
3. option (2)
4. options (2)
5. option (3)
6. options (3)
7. section
8. sections

Mid-rule semantic actions

You can also place semantic actions in the middle of a rule.

Internally bison normalizes the grammar in order to have only **end-of-rule actions**:

With mid-rules

```
section:
  LSQUARE ID RSQUARE
    { /* 1 */ }
  options
    { /* 2 */ }
;
```

No mid-rules

```
section:
  LSQUARE ID RSQUARE $@1 options
    { /* 2 */ }
;
$@1:
  %empty
    { /* 1 */ }
;
```

Mid-rule actions can introduce ambiguities for this reason!

Semantic values

Problem: we need to **keep track of what each token/non-terminal represents**

- Just looking at the token identifier is not enough
- ‘1234’ and ‘5432’ are both *NUMBER*s, but they are not **the same** number

The solution: **Semantic Values**

- We associate **a variable** to each token or non-terminal parsed
- For **tokens**: its value is assigned **in the lexer**
- For **non-terminals**: its value is assigned **in the semantic action(s) of that non-terminal**
- We **read*** that variable in the rules that use that token or non-terminal

---

*We could obviously also assign a new value to the variable, but that’s not particularly useful typically

Semantic values

- `%union` declaration specifies the entire collection of possible data types
- Type specification for **terminals** (tokens) in the **token declaration**
- Type specification for **non-terminals** in special **%type declarations**

```
%union {
  int int_val;
  const char *str_val;
  option_t option_val;
}
```

```
%token <str_value> ID
%token <str_value> STRING
%token <int_val> NUMBER
%type <option_val> option
```

Accessing semantic values

The semantic value of each grammar symbol in a production is a variable called `$`i, where i is the position of the symbol

- `$$` corresponds to the semantic value of **the rule itself**
- Mid-rule actions "count" in the numbering
- Mid-rule actions have additional restrictions:
  - You cannot access values of symbols that come later
  - You **cannot use `$$`***

```
$$ → section : LSQUARE ← $1
               ID ← $2
               RSQUARE ← $3
               { printf("%s", $2); } ← $4
               options ← $5
               { $$ = create_section($2, $5); }
               ;
```

*Actually you can, it will be the semantic value of the **action itself** – and that's why mid-rule actions count in the numbering

Interface of `bison`

The generated parser is a **C file** with suffix **.tab.c**

- Also generates an header with declarations: suffix **.tab.h**

Main parsing function:

```
int yyparse(void);
```

For reading tokens the parser uses **the same *yylex()* function** that flex-generated scanners provide!

- Additionally, it declares the *yylval* global variable
- Contains the semantic value of the last token returned by *yylex()*
- Type: **union** of the types declared in the bison source

Digression: the *union* type

Unions are a kind of compound data type defined by the **C language**

- Not specific to bison
- Used by bison to associate multiple types to semantic values

Unions are **like structs**, but **assigning a value to one item invalidates the others**

- Structs allocate their items sequentially in memory
- Union members **overlap** in memory

```
typedef union {
  int a;
  double b;
} an_union_t;

an_union_t an_union;
```

```
an_union.a = 10;
/* an_union.b == garbage */

an_union.b = 999.5;
/* an_union.a == garbage */
```

Effects of the %union declaration

Let's go back to the previous example:

```
%union {
  int int_val;
  const char *str_val;
  option_t option_val;
}
```

```
%token <str_value> ID
%token <str_value> STRING
%token <int_val> NUMBER
%type <option_val> option_t
```

The *yylval* variable is declared like this:

```
typedef union YYSTYPE {
  int int_val;
  const char *str_val;
  option_t option_val;
} YYSTYPE;

YYSTYPE yylval;
```

Integration of `flex` and `bison`

In the flex source:

1. Include the *.tab.h header generated by bison
2. In the semantic actions:
    - Assign the semantic value of the token (if any) **to the correct member of the *yylval* variable**
    - Return **the token identifiers declared in bison**

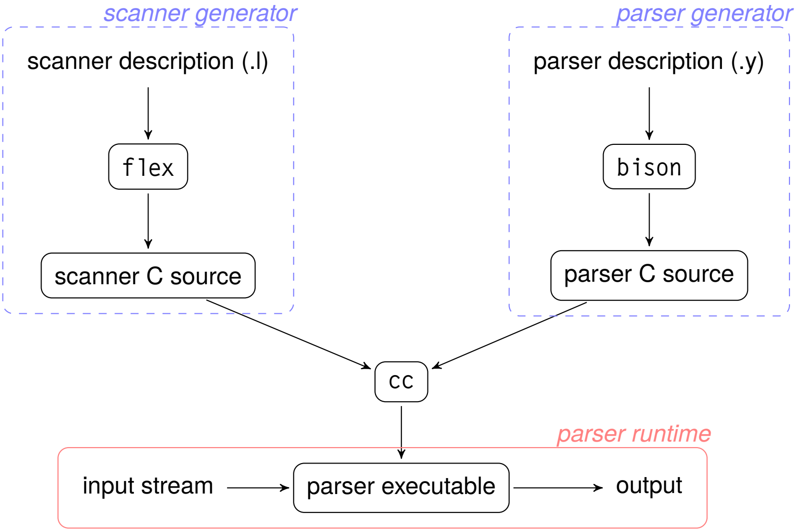
In the bison source:

3. Declare and implement the *main()* function

When compiling:

4. Generate the flex scanner by invoking flex
    - Command line: `flex scanner.l`
5. Generate the bison parser by invoking bison
    - Command line: `bison parser.y`
6. Compile the C files produced by bison and flex **together**
    - Command line: `cc -o out lex.yy.c parser.tab.c`

Workflow


scanner generator
scanner description (.l)
flex
scanner C source
parser generator
parser description (.y)
bison
parser C source
cc
parser runtime
input stream
parser executable
output

Contents

RPN Calculator

Let's look at the implementation of an **RPN calculator**

- RPN = Reverse Polish Notation = postfix notation
- Operators follow **all the operands**
- Non-ambiguous syntax (no need for parenthesis!)
- Used in some scientific calculators

Example

Postfix notation	$5\ 1\ 2\ +\ 3\ *\ -$
Infix notation	$5 - ((1 + 2) * 3)$

RPN Calculator

Grammar

The grammar is simple:

program	:	*lines* NEWLINE
lines	:	*lines line*
	|	ε
line	:	exp NEWLINE
exp	:	NUMBER
	|	*exp exp* PLUS
	|	*exp exp* MINUS
	|	*exp exp* DIV
	|	*exp exp* MUL

RPN Calculator

Parser

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

int yylex(void);
void yyerror(char *);
%}

%union {
  int value;
}

%token <value> NUMBER
%token NEWLINE PLUS
%token MINUS MUL DIV
%type <value> exp

%%
```

```
program : lines NEWLINE
          { YYACCEPT; };

lines : lines line
      | %empty;

line : exp NEWLINE
       { printf("%d\n", $1); };

exp : NUMBER          { $$ = $1; }
    | exp exp PLUS  { $$ = $1 + $2; }
    | exp exp MINUS { $$ = $1 - $2; }
    | exp exp MUL   { $$ = $1 * $2; }
    | exp exp DIV   { $$ = $1 / $2; };

%%

/* ... */
```

RPN Calculator

Scanner

```
%{
#include <stdlib.h>
#include "rpn-calc.tab.h"
%}
%option noyywrap

%%

"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MUL; }
"/"             { return DIV; }
"\n"            { return NEWLINE; }
[0-9]+          { yylval.value = atoi(yytext);
                  return NUMBER; }
[ \t\r]+
```

Some more little details...

In the example there are some thing that we have not seen yet:

- The *yyerror()* function is called by the generated parser when **a syntax error is found**
  - The string parameter contains an error message
  - You must implement it yourself (just *printf* the error)
- The *YYACCEPT* macro is used to tell the parser that at that point it should return successfully without errors
- The *%empty* token corresponds to the empty string (ε)

This bison program **computes the value of the expression while it parses**

- It is an **interpreter**

Let’s modify the example a bit...

RPN Expression Compiler (1/2)

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

int next_var_id = 1;

int yylex(void);
void yyerror(char *);
%}

%union {
  int value;
  int var_id;
}

%token <value> NUMBER
%token NEWLINE PLUS
%token MINUS MUL DIV
%type <reg_id> exp

%%
```

```
program:
  {
    printf("#include <stdio.h>\n\n");
    printf("int main(int argc, char *argv[])\n");
    printf("{\n");
  }
  lines NEWLINE
  {
    printf("  return 0;\n");
    printf("}\n");
    YYACCEPT;
  }
;

lines : lines line
      | %empty;

line:
  exp NEWLINE
  {
    fprintf("  printf(\"%%d\\n\", v%d);\n\n", $1);
  }
;
```

RPN Expression Compiler (2/2)

```
exp:
  NUMBER          {
                    $$ = next_var_id++;
                    printf("  int v%d = %d;\n", $$, $1);
                  }
  | exp exp PLUS  {
                    $$ = next_var_id++;
                    printf("  int v%d = v%d + v%d;\n", $$, $1, $2);
                  }
  | exp exp MINUS {
                    $$ = next_var_id++;
                    printf("  int v%d = v%d - v%d;\n", $$, $1, $2);
                  }
  | exp exp MUL   {
                    $$ = next_var_id++;
                    printf("  int v%d = v%d * v%d;\n", $$, $1, $2);
                  }
  | exp exp DIV   {
                    $$ = next_var_id++;
                    printf("  int v%d = v%d / v%d;\n", $$, $1, $2);
                  }
;
```

Where is the computation?

The modified example
does not compute the value of the expression

Instead, it **produces C code** that computes the value of the expression

The computation only happens when **the C code produced** is compiled and executed, in turn

This is a **compiler**, not an interpreter

Where is the computation?

Input

$5\;1\;2\;+\;3\;*\;-$

Interpreter Output

-4

Compiler Output

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int v1 = 5;
  int v2 = 1;
  int v3 = 2;
  int v4 = v2 + v3;
  int v5 = 3;
  int v6 = v4 * v5;
  int v7 = v1 - v6;
  printf("%d\n", v7);
  return 0;
}
```

Compile Time versus Run Time

In an **interpreter**, execution of the parsed commands happens **immediately**

In a **compiler**, the commands are simply **rewritten in another language**, without executing them. Of course we still need to perform some (but *different*) computations.

Definition: Compile Time

Computations performed **in the compiler** to produce the compiled output

Definition: Run Time

Computations performed **by the compiled program when it is later executed**

Compile Time versus Run Time

Example:

Compile Time Operations

```
printf("#include <stdio.h>\n\n");
printf("int main(int argc, char *argv[])\n");
printf("{\n");
$$ = next_var_id++;
printf("  int v%d = %d;\n", $$, $1);
$$ = next_var_id++;
printf("  int v%d = v%d + v%d;\n",$$,$1,$2);
printf("  return 0;\n");
printf("}\n");
```

And all the stuff that the parser generated by bison and the scanner generated by flex are doing for us to “decode” the input

Run Time Ops.

```
int v1 = 5;
int v2 = 1;
int v3 = 2;
int v4 = v2 + v3;
int v5 = 3;
int v6 = v4 * v5;
int v7 = v1 - v6;
printf("%d\n", v7);
```

Contents

Infix calculator

Now let's stop with that RPN nonsense!
We will modify the calculator to read **normal infix expressions**

The grammar appears simple like before...

program	:	*lines* NEWLINE
lines	:	*lines line*
	|	ε
line	:	exp NEWLINE
exp	:	NUMBER
	|	*exp* PLUS *exp*
	|	*exp* MINUS *exp*
	|	*exp* DIV *exp*
	|	*exp* MUL *exp*
	|	LPAR *exp* RPAR

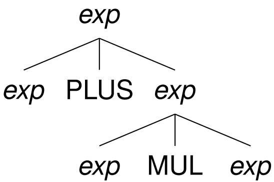
Infix calculator

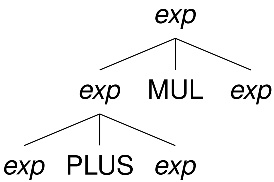
However, this new grammar is **ambiguous**! Why?

Consider this expression:

$$1 + 2 * 3$$

What is the correct parse tree?





By convention, the right tree is the one on the **left**.
How do we tell bison about this convention?

Infix calculator

Solving ambiguities

Solution 1 (bad): remove the ambiguities in the grammar manually

- The grammar must be **left-recursive only** to ensure the non-ambiguity of expressions like $1 + 2 + 3 + 4$
- The exp rule must be split between subexpressions with multiplications and subexpressions with additions to encode the precedence between operators
- **Problem:** The resulting grammar is non-obvious

Solution 2 (good): use the easy grammar; when the parser encounters an ambiguity it is resolved depending on the last tokens read

- Easy to implement in LR-family parsers because ambiguities result in **shift-reduce conflicts**
- **bison supports this solution explicitly!**

Precedence and associativity

Goal: decide which rule takes precedence **depending on the tokens**

Example: $1 + 2 * 3$

- $\boxed{\textit{exp} : \textit{exp}\ \text{PLUS}\ \textit{exp}}$ vs $\boxed{\textit{exp} : \textit{exp}\ \text{MUL}\ \textit{exp}}$
- $+$ higher precedence than $*$: $(1 + 2) * 3$
- $*$ higher precedence than $+$: $1 + (2 * 3)$

If the precedence is the same, then **associativity** kicks in

Example: $1 + 2 + 3$

- $\boxed{\textit{exp} : \textit{exp}\ \text{PLUS}\ \textit{exp}}$ vs $\boxed{\textit{exp} : \textit{exp}\ \text{PLUS}\ \textit{exp}}$
- $+$ **left** associative: $(1 + 2) + 3$
- $+$ **non associative**: *syntax error!*
- $+$ **right** associative: $1 + (2 + 3)$

Precedence declaration in bison

How to declare precedence in *bison*?

- `%precedence` declarations
- Goes into the **definitions** part of the file (the first section)
- List the tokens in **order of growing precedence**
  - Token that comes first: **lowest precedence**
  - Token that comes last: **highest precedence**
- Tokens listed in the same line have the same precedence

```
%precedence PLUS MINUS
%precedence MUL DIV
```

Associativity declaration in bison

How to declare associativity in *bison*?

- **Replace** `%precedence` with one of the following:
  - **`%left`** for left associativity
  - **`%nonassoc`** for not associative
  - **`%right`** for right associativity

Tip: usually **left** associativity is what you want

```
%left PLUS MINUS
%left MUL DIV
```

Infix Calculator

Parser

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

int yylex(void);
void yyerror(char *);
%}

%union {
  int value;
}

%token <value> NUMBER
%token NEWLINE PLUS MINUS
%token MUL DIV LPAR RPAR
%type <value> exp

%left PLUS MINUS
%left MUL DIV

%%
```

```
program : lines NEWLINE
          { YYACCEPT; };

lines : lines line
      | %empty;

line : exp NEWLINE
       { printf("%d\n", $1); };

exp : NUMBER          { $$ = $1; }
    | exp PLUS exp  { $$ = $1 + $3; }
    | exp MINUS exp { $$ = $1 - $3; }
    | exp MUL exp   { $$ = $1 * $3; }
    | exp DIV exp   { $$ = $1 / $3; }
    | LPAR exp RPAR { $$ = $2; };

%%

/* ... */
```

Contents

Homework

1. Modify the infix calculator and transform it into the **infix compiler**

2. Modify the infix compiler (or the RPN compiler if you prefer) to support for **input expressions**
   - In an expression, a question mark enclosed in brackets may appear, followed by an arbitrary string
   - Example: `981 * [? time]`
   - The compiled program must **display the string**, **ask for a value**, and then use that value in the expression
   - **The compiler obviously doesn't ask any additional input**

3. Modify the infix calculator to add support for **variables**
   - Before an expression, the user must be able to specify in which variable it is stored
   - Example: `x = 10 + 3 * 2` stores `16` in the variable `x`
   - When a variable identifier appears in an expression, its value is used in the computation
   - Tip: use a linked list to store all the variables...

Contents

During parsing of an expression, in case of ambiguity, a **shift/reduce conflict** occurs when the parser reaches **the second operator**

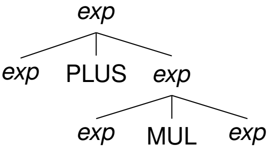
Example: $1 + 2 * 3$

Action $\rightarrow$	**Stack**	**Lookahead and input**
...	*exp* PLUS *exp*	MUL NUMBER
	Option 1: shift MUL	
Shift MUL	*exp* PLUS *exp* MUL	NUMBER
Shift NUMBER	*exp* PLUS *exp* MUL NUMBER	
Reduce *exp*	*exp* PLUS *exp* MUL *exp*	
Reduce *exp*	*exp* PLUS *exp*	
Reduce *exp*	*exp*	
	Option 2: reduce *exp*	
Reduce *exp*	*exp*	MUL NUMBER
Shift MUL	*exp* MUL	NUMBER
Shift NUMBER	*exp* MUL NUMBER	
Reduce *exp*	*exp* MUL *exp*	
Reduce *exp*	*exp*	

Precedence/associativity: how it works

Shifting results in the **multiplication** having precedence:

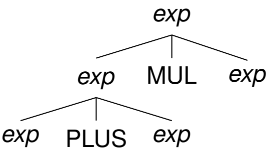
Action $\rightarrow$	**Stack**	**Lookahead and input**
…	*exp* PLUS *exp*	MUL NUMBER
Shift MUL	*exp* PLUS *exp* MUL	NUMBER
Shift NUMBER	*exp* PLUS *exp* MUL NUMBER	
Reduce *exp*	*exp* PLUS *exp* MUL *exp*	
Reduce *exp*	*exp* PLUS *exp*	
Reduce *exp*	*exp*	



Precedence/associativity: how it works

Reducing results in the **addition** having precedence:

Action $\rightarrow$	**Stack**	**Lookahead and input**
…	*exp* PLUS *exp*	MUL NUMBER
Reduce *exp*	*exp*	MUL NUMBER
Shift MUL	*exp* MUL	NUMBER
Shift NUMBER	*exp* MUL NUMBER	
Reduce *exp*	*exp* MUL *exp*	
Reduce *exp*	*exp*	



Contents

Context-dependent precedence

Let's extend the calculator to handle the *unary minus*:

```
exp : NUMBER
    | exp PLUS exp
    | exp MINUS exp
    | exp MUL exp
    | exp DIV exp
    | LPAR exp RPAR
    | MINUS exp      // <- new!
    ;
```

Problem: with the precedence we have specified we get **strange behavior**:

- The expression $-3 * 5$ is parsed like $-(3 * 5)$
- We wanted it to parse like $(-3) * 5$!
- The reason is that the multiplication is higher priority than the minus sign!

We need to somehow set a **different priority** for the minus **only when we are parsing that new rule**

Context-dependent precedence

Solution: define a non-existing token with the desired associativity and precedence and use it to override the precedence in the context of the rule.

```
%left PLUS MINUS
%left MUL DIV
%right UMINUS

%%

exp : NUMBER
    | exp PLUS exp
    | exp MINUS exp
    | exp MUL exp
    | exp DIV exp
    | LPAR exp RPAR
    | MINUS exp %prec UMINUS
    ;
```

`%prec` specifies that the rule where it appears must have the precedence of a different given token (in this case `UMINUS`)