Flex used to generate recursive descent. Recognize regular expression and execute some code as soon as some regular expression is recognized.

Bison recognize not only ELR but the socalled generelized ELR.
It is used with flex which generates the ...

Formal Languages and Compilers Laboratory

Syntactic Analysis: Bison

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

Contents

Multiple source files

It is possible to **build a single program** from **multiple source files**

- Each source file is first compiled separately into an **object file**
- Object files are **binaries** but are **not executable** --> are intermediate step between source and executable file
- The process of **combining multiple object files** into an **executable** (or a library) is called **linking**

library.c

```c
#include <stdio.h>

void hello(void)
{
  printf("hello!\n");
}
```

main.c

```c
// prototype of the function
// from library.c
void hello(void);

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

Compilation commands

```
cc -o program main.c library.c
```

Header files

When using multiple source files, we need to repeat prototypes of functions many times

- Solution: **header files**
- They are files including prototypes and definitions (usually) relative to **a specific .c file**
- We include them **when we use the functions in that .c file**

library.c

```c
#include <stdio.h>
#include "library.h"

void hello(void)
{
  printf("hello!\n");
}
```

library.h

header file for library.c, it includes only signatures of functions and can be included in source codes that want to use library.c functions.

```c
void hello(void);
```

main.c

```c
#include "library.h"


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

Compilation commands

```
cc -o program main.c library.c
```

Header files are not passed to the compiler!

struct {
int x,
char y,
float BUFF[30]
}

Unions

The total size of the struct is at least the sum of the sizes of the types of which it is composed (at least because it depends on the disposition in memory, not always it is contiguous

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

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

- Union members **overlap** in memory
- (in contrast with *structs* which allocate their items **sequentially** 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 */
```

The size of union is the size of the biggest type size contained.

Contents

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

A **grammar** defines the syntax of a language.
A syntactic analysis must:

- identify grammar structures --> to define rules as in contex-free grammar
- verify syntactic correctness --> after that we can pass input to the bison generated program from the grammar structure and get checks about syntactic correctness, or report conflicts (reduce-reduce ecc...).
- 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

However since bison is more powerful than ELR method we used, it can resolve some conflicts. Es:
If()
If()
else
can be resolved by means of associativity rules

The syntactic analysis takes as input 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

Here only saying bison uses LR, which is more powerful than ELR

The standard tool to **generate** LR **parser** (i.e. syntax analyzers):

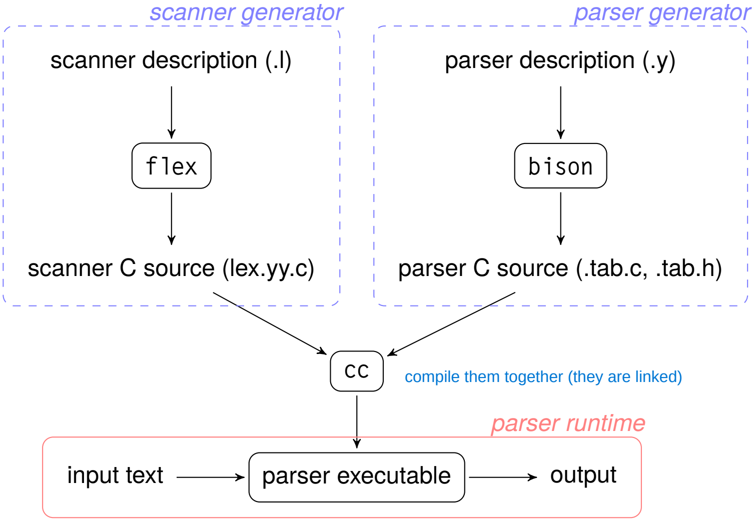
- Improvement on a previous tool called YACC
- Designed to work seamlessly together with `flex`

It is based on the **LALR(1)** theory

- Variant of **LR(1)**, of which **ELR(1)** is another variant...
- Implements a push-down automaton driven by a pilot graph
- The parsing stack is used to keep the parser state at runtime
- Very similar to what you’ve seen in the theory classes

Workflow

Flex generates a scanner

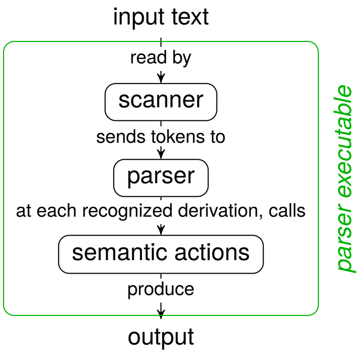


compile them together (they are linked)

we get final parser

Inside the parser

- The **parser** takes the token from the **scanner**
- The **parser** decodes the grammar derivations
- At each decoded derivation, its *semantic action* is invoked
- The semantic actions produce the output of the parsing process



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!

- It is called every time the parser requires a new token

Contents

File format

The structure is the same as the one of flex files

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
```

--> C code here

IN
S-> aSa

a is a token

Same structure as a **flex** file!

Definition in bison:

%TOKEN A

In flex we include the header file generated by bison and then we write

"a" {return A,}

So in flex we don't have to specify tokens anymore, we just write what to recognize and return what bison want (the token)

Definition section

The most important part of the **definitions sections** are the **token declarations**:

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

Bison also generates a *header file* which defines a code for each token. This allows the *scanner* to know these codes.

Partial contents of the .tab.h file generated by Bison:

ex: in flax we could have
"IF"
{Return IF}

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

Rules section

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

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

here we define the syntax of the rule (what we want to recognize)

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
          ;
```

The only one which can be fully recognized without interrogating other non terminals is "option"

Rules section: 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

Here we can define what to do when something is recognized

Example

--> if at some point we recognize "ID EQUALS NUMBER", we execute what is in the square brackets. This means making a reduction: es. in S -> aSa | bSb we could have recognized aSa and execute a reduction

```
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)

```
                sections
                   |
                section
                   |
              options (3)
             /           \
      options (2)      option (3)
       /        \
options (1)   option (2)
    |
option (1)   --> first one executed, then up
```

Execution order

from the leaf to the root

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 *[A-Z]*" {RETURN ?whattoputhere?}

The solution: **Semantic Values**

- We associate **a variable** to each token or non-terminal parsed
- For **tokens** (terminals): 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

Definition of semantic values

The types of each semantic value are specified in the *definition section*:

- `%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 in actions

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

Setting semantic values in the scanner

In the generated code, Bison also declares the *yylval* global variable:

- It contains the semantic value of the last token returned by *yylex()*
- Type: **union** of the types declared in the bison source

This allows the **scanner** to **set the semantic value of a token**:

- The **flex semantic action**, before returning the token value, **sets *yylval* appropriately**

Example flex semantic action

```
[0-9]+  {
            yylval.value = atoi(yytext);
            return NUMBER;
        }
```

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 --> for having the tokens and the union
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`

Contents

RPN Calculator

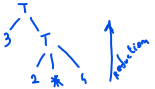
Let's look at the implementation of an **RPN calculator**

- 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	$5-((1+2)*3)$

Usually: 3+2*4 :



In postfix notation we have terminal, terminal, operator. It avoids ambiguity (every operator involves 2 operants and at the end of the two there is the operator).
3 2 4* + (2 and 4 multiplied, 3 added to the result)
3 2 4 + * (2 plus 4 everything multiplied by 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

can be rewritten as:
"lines --> lines line | epsilon"

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

%%
```

prolog

-> for the semantic value

remember:
token for terminal, type for non terminal

Only NUMBER and exp have semantic values

convenction: terminal in capital

```
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; };

%%

/* ... */
```

left part

something provided by flex (In this case a number)

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]+
```

only token here, cause types are non terminal and are managed by bison

we associate the text recognized by flex, stored in yytext, to a value

then flex will recognize the type of "value" looking in the union and then assigned it to one of the $

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...

The previous expression was an interpreter, here a compiler.
The difference is that the letter "translates" , the first calculate a result.

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 <var_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);
                  }
;
```

in exp we are not saving a value but "storing" an identifier used to printf

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;
}
```

we can imagine substituting the input with an a scanf. We know how to handle the input the user will insert.

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

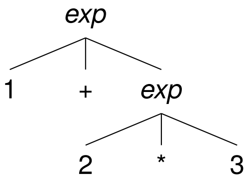
Ambiguities in infix expressions

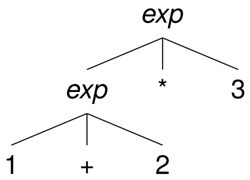
Usually expressions are written in **infix notation**.
Each operator is placed **between** its two operands.

Consider this expression:

$$1 + 2 * 3$$

What is the correct parse tree?





By convention, the correct tree is the one on the **left**,
because multiplication takes precedence.

Infix expressions are inherently ambiguous.

Precedence and associativity

We solve ambiguities in an infix expression by introducing **precedence** and **associativity** rules.

Example: $1 + 2 * 3$

- $+$ higher precedence than $*$: $(1 + 2) * 3$
- $*$ higher precedence than $+$: $1 + (2 * 3)$

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

Example: $1 + 2 + 3$

- $+$ **left** associative: $(1 + 2) + 3$
- $+$ **non associative**: *syntax error!*
- $+$ **right** associative: $1 + (2 + 3)$

Implementation in BNF grammars

A **strictly** BNF grammar for infix expressions must encode precedence and associativity manually:

- Left associativity is expressed using **left-recursive rules**
- Right associativity is expressed using **right-recursive rules**
- Precedence level is determined by using multiple non-terminals

exp	:	*exp* PLUS *term*
	|	*exp* MINUS *term*
	|	*term*
term	:	*term* MUL *factor*
	|	*term* DIV *factor*
	|	*factor*
factor	:	NUMBER
	|	LPAR *exp* RPAR

Infix calculator with bison

Bison allows us to use a **much simpler grammar**:

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

Precedence and associativity are **handled with a separate mechanism** controlled by definitions in the bison file.

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
- Rules take precedence from **the last token**

```
%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
```

precedente doesn't let you specify associativity --> usually not used for that reason.

the other three keywords (left, nonassoc, right) keep the meaning of precedence but also specify associativity

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

Action	**Stack**	**Lookahead and input**
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*	

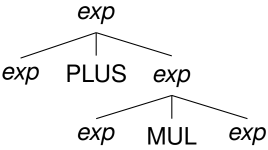
Option 2: reduce *exp*

Action	**Stack**	**Lookahead and input**
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:

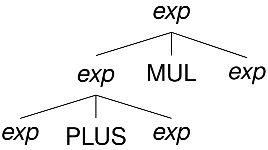
Action $\rightarrow$	**Stack**	**Lookahead and input**
…	*exp* PLUS *exp*	MUL NUMBER
Shift MUL	*exp* PLUS *exp* MUL	NUMBER
Shift NUMBER	*exp* PLUS *exp* MUL NUMBER	
Reduce *exp*	*exp* PLUS *exp* MUL *exp*	
Reduce *exp*	*exp* PLUS *exp*	
Reduce *exp*	*exp*	



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*	



Precedence/associativity: how it works

When the bison-generated parser encounters a **shift/reduce conflict**, it decides what to do depending on the precedence/associativity set on the tokens.

- Unless otherwise specified, consider the precedence/associativity of a rule to be equal to the one of its last terminal symbol.
- Let $a =$ the rule we would reduce. --> for rule the priority is given by the last token
- Let $b =$ the token we would shift.
- If precedence$(a) >$ precedence(b) then reduce;
- else, if precedence$(a) =$ precedence(b):
  - Note that associativity$(a) =$ associativity(b) due to how the declarations work.
  - If associativity(a) is *left* then reduce;
  - else shift.
- else shift.

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 token than the default (in this case `UMINUS` rather than `MINUS`)