Programming with GNU Software

Edition 2, 12 November 2001
$Id: gnuprog2.texi,v 1.10 2001/09/17 21:06:30 akim Exp $

Gary V. Vaughan
gary@gnu.org

Akim Demaille
akim@epita.fr

Paul Scott
pmscott@techie.com

Bruce Korb
bkorb@gnu.org

Richard Meeking
rmeeking@users.sourceforge.net

Copyright © 2000 Gary V. Vaughan, Akim Demaille, Paul Scott, Bruce Korb, Richard Meeking

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table of Contents

1 Parsing 1
1.1 Looking for Balanced Expressions 1
1.2 Looking for Arithmetics 4
1.3 What is Bison 7
1.4 Bison as a Grammar Checker 7
1.5 Resolving Conflicts 10
1.6 Simple Uses of Bison 13
1.7 Using Actions 17
1.8 Advanced Use of Bison 22
1.9 The `yleval` Module 27
1.10 Using Bison with the GNU Build System 32
1.11 Exercises on Bison 33
1.12 Further Reading On Parsing 33

1 Parsing

1.1 Looking for Balanced Expressions

We have seen that Flex supports abbreviations, see ⟨undefined⟩ [Flex Regular Expressions], page ⟨undefined⟩. Using a more pleasant syntax, we could write for instance:

```
digit:  [0-9];
number: digit+;
```

It is then tempting to describe possibly parenthesized expressions using these abbreviations:

```
expr: '(' expr ')' | number
```

Example 7.1: *A Simple Balanced Expression Grammar*

to describe numbers nested in balanced parentheses.

Of course, Flex abbreviations cannot be recursive, since recursion can be used as above to describe balanced parentheses, which fall out of Flex' expressive power: Flex produces finite state recognizers, and FSR cannot recognized balanced parentheses because they have finite memory (see ⟨undefined⟩ [Start Conditions], page ⟨undefined⟩).

This suggests that we need some form of virtually infinite memory to recognize such a language. The most primitive form of an infinite memory device is probably stacks, so let's try to design a recognizer for `expr` using a stack. Such automata are named *pushdown automata*.

Intuitively, if the language was reduced to balanced parentheses without any nested number, we could simply use the stack to push the opening parentheses, and pop them when finding closing parentheses. Slightly generalizing this approach to the present case, it seems a good idea to push the tokens onto the stack, and to pop them when we find what can be done out of them:

Step	Stack	Input
1.		`( ( number ) )`
2.	`(`	`( number ) )`
3.	`( (`	`number ) )`
4.	`( ( number`	`) )`

At this stage, our automaton should recognize that the `number` on top of its stack is a form of `expr`. It should therefore replace `number` with `expr`:

Step	Stack	Input
5.	`( ( expr`	`) )`
6.	`( ( expr )`	`)`

Now, the top of the stack, '`( expr )`', clearly denotes an `expr`, according to our rules:

Step	Stack	Input
7.	`( expr`	`)`
8.	`( expr )`	
9.	`expr`	

Finally, we managed to recognize that '`( ( number ) )`' is indeed an expression according to our definition: the whole input has been processed (i.e., there remains nothing in the input), and the stack contains a single nonterminal: `expr`. To emphasize that the whole input must be processed, henceforth we will add an additional token, '`$`', standing for end of file, and an additional rule:

```
$axiom: expr $;
```

If you look at the way our model works, there are basically two distinct operations to perform:

shift Shifting a token, i.e., fetching the next token from the input, and pushing it onto the stack. Steps performed from the states 1, 2, 3, 5, and 7 are shifts.

reduce Reducing a rule, i.e., recognizing that the top of the stack represents some right hand size of a rule, and replacing it with its left hand side. For instance at stage 4, the top stack contains '`number`' which can be reduced to '`expr`' thanks to the rule '`expr: number;`'. At stages 6 and 8, the top of the stack contains '`( expr )`', which, according to '`expr: '(' expr ')';`' can be reduced to '`expr`'.

The initial rule, '`$axiom: expr $;`' is special: reducing it means we recognized the whole input. This is called *accepting*.

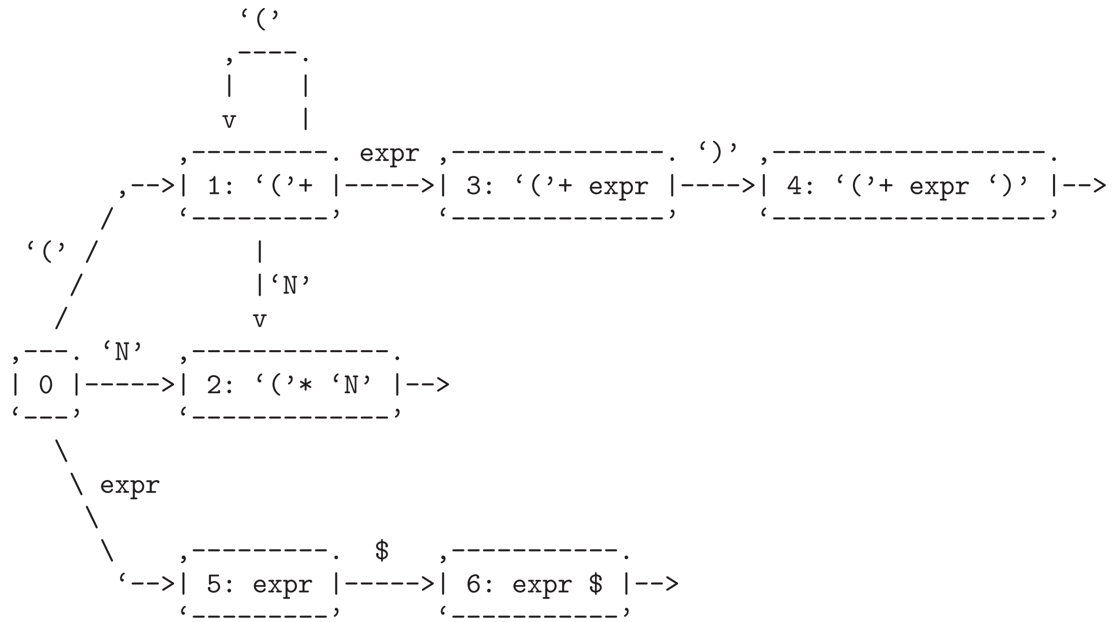
The tricky part is precisely knowing when to shift, and when to reduce: the strategy we followed consists in simply reducing as soon as we can, but how can we recognize stacks that can be reduced? To answer this question, we must also consider the cases where the input contains a syntax error, such as '`number number`'. Finally, our question amounts to "what are the valid stacks ready to be reduced". It is easy to see there are two cases: a possibly empty series of opening parentheses followed by either a lone `number`, or else by a '`(`', then an '`expr`', and a '`)`', or, finally a single '`expr`' and end of file.

Hey! "A possibly empty series of opening parentheses followed by...": this is typically what we used to denote with regular expressions! Here:

```
‘(’* ( expr $ | ‘(’ expr ‘)’ | number )
```

If we managed to describe our reducible stacks with regular expressions, it means we can recognize them using a finite state recognizer!

This FSR is easy to compute by hand, since its language is fairly small. It is depicted below, with an additional terminal symbol, '`$`', denoting the end of file, and '`N`' standing for `number`:



Example 7.2: *A Stack Recognizer for Balanced Expressions*

As expected, each final state denotes the recognition of a rule to reduce. For instance state 4 describes the completion of the first rule, '`expr: '(' expr ')'`', which we will denote as '`reduce 1`'. State 6 describes the acceptance of the whole text, which we will denote '`accept`'. Transitions labelled with tokens are shifts, for instance if the automaton is in state 1 and sees an '`N`', then it will '`shift 2`'. An transition labelled with a non terminal does not change the stack, for instance an '`expr`' in state 1 makes the automaton '`go to 4`'. Any impossible transition (for instance receiving a closing parenthesis in state 0) denotes a syntax error: the text is not conform to the grammar.

In practice, because we want to produce efficient parsers, we won't restart the automaton from the beginning of the stack at each turn. Rather, we will remember the different states we reached. For instance the previous stack traces can be decorated with the states and actions as follows:

Step	Stack	Input	Action
1.	`0`	`( ( number ) ) $`	shift 1
2.	`0 ( 1`	`( number ) ) $`	shift 1
3.	`0 ( 1 ( 1`	`number ) ) $`	shift 2
4.	`0 ( 1 ( 1 number 2`	`) ) $`	reduce 2
5.	`0 ( 1 ( 1 expr`	`) ) $`	go to 3
6.	`0 ( 1 ( 1 expr 3`	`) ) $`	shift 4
7.	`0 ( 1 ( 1 expr 3 ) 4`	`) $`	reduce 1
8.	`0 ( 1 expr`	`) $`	go to 2
9.	`0 ( 1 expr 2`	`) $`	shift 4
10.	`0 ( 1 expr 2 ) 4`	`$`	reduce 1
11.	`0 expr`	`$`	go to 5
12.	`0 expr 5`	`$`	shift 6
13.	`0 expr 5 $ 6`		accept

Example 7.3: *Step by Step Analysis of* '`( ( number ) )`'

This technology is named LR(0) parsing, "L" standing for Left to Right Parsing, "R" standing for Rightmost Derivation[1], and "0" standing for no lookahead symbol: we never had to look at the input to decide whether to shift or to reduce.

Again, while the theory is simple, computing the valid stacks, designing an finite state stack recognizer which shifts or reduces, etc. is extremely tedious and error prone. A little of automation is most welcome.

1.2 Looking for Arithmetics

Except for a few insignificant details, the syntax introduced in the previous section is called BNF, standing for Backus-Naur form, from the name of its inventors: they used it to formalize the Algol 60 programming language. It is used to define *grammars*: a set of *rules* which describe the structure of a language, just as we used grammars to describe the English sentences at school. As for natural languages, two kinds of *symbols* are used to describe artificial languages: *terminal symbols* (e.g., '`he`', '`she`', etc. or '`+`', '`-`', etc.) and *nonterminal symbols* (e.g., "subject", or "operator"). Examples of grammars include

```
sentence:    subject predicate;
subject:     'she' | 'he' | 'it';
predicate:   verb noun-phrase | verb;
verb:        'eats';
noun-phrase: article noun | noun ;
article:     'the';
noun:        'bananas' | 'coconuts';
```

or

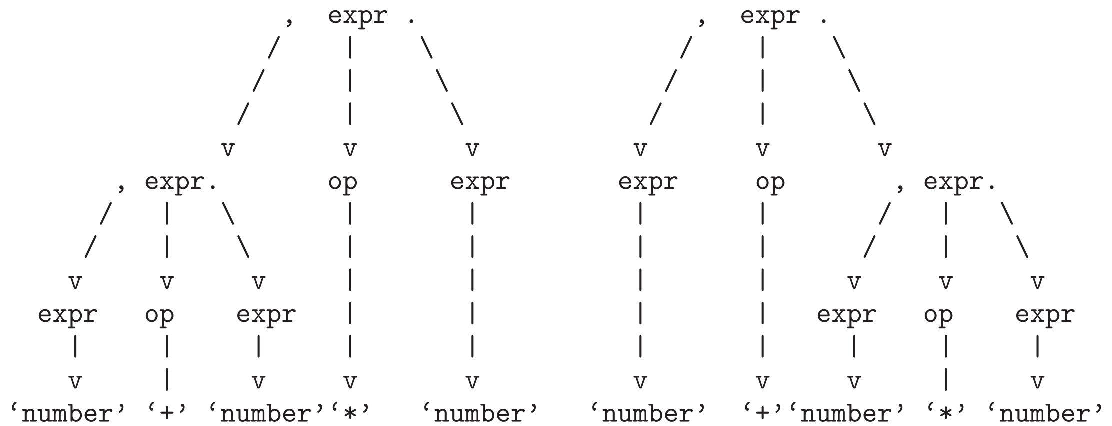
```
expr: expr op expr | '(' expr ')' | 'number';
op:   '+' | '-' | '*' | '/';
```

Example 7.4: *A Grammar for Arithmetics*

Such rules, which left hand side is always reduced to a single nonterminal, define so called *context free grammars*. Context free grammars are properly more powerful than regular expressions: any regular expression can be represented by a context free grammar, and there are context free grammars defining languages that regular expressions cannot denote (e.g., the nested parentheses).

Grammars can be *ambiguous*: some sentences can be understood in different ways. For instance the sentence '`number + number * number`' can be understood in two different ways by the grammar of the *example 7.4*:

[1] When we reduce a rule, it is always at the top of our stack, corresponding to the rightmost part of the text input so forth. Some other parsing techniques, completely different, first handle the leftmost possible reductions.



Example 7.5: *Non Equivalent Parse Trees for* '`number + number * number`'

Because having different interpretations of a given sentence cannot be accepted for artificial languages (a satellite could easily be wasted if programmed in an ambiguous language), we will work with unambiguous grammars exclusively. For instance, we will have to refine the grammar of the *example 7.4* in order to use it with Bison.

Please note that even if limited to the minus operator, this grammar is still ambiguous: two parse trees represent '`number - number - number`'[2]. We, humans, don't find it ambiguous, because we know that by convention '`-`' is executed from left to right, or, in terms of parenthesis, it forms groups of two from left to right. This is called *left associativity*. There exist *right associative* operators, such as power, or assignment in C, that handle their rightmost parts first.

The following unambiguous grammar denotes the same language, but keeps only the conventional interpretation of subtraction: left associativity.

```
expr: expr '-' 'number' | 'number';
```

Example 7.6: *An Unambiguous Grammar for* '`-`' *Arithmetics*

Let's look at our stack recognizer again, on the input '`number - number`':

Step	Stack	Input
1.		`number - number`
2.	`number`	`- number`
3.	`expr`	`- number`
4.	`expr -`	`number`
5.	`expr - number`	
6.	`expr`	

Example 7.7: *An LR(1) Parsing of* '`number - number`'

[2] These two parse trees are those of the *example 7.5*, with '`-`' replacing '`*`' and '`+`'.

This time, we no longer can systematically apply the rule '`expr: 'number'`' each time we see a '`number`' on the top of the stack. In both step 2 and step 5, the top of the stack contains a `number` which can be reduced into an `expr`. We did reduce from step 2, but in step 5 we must not. If we did, our stack would then contain '`expr '-' expr`', out of which nothing can be done: all the '`number`'s *except* the last one, must be converted into an `expr`. We observe that looking at the next token, named the *lookahead*, solves the issue: if the top stack is '`number`', then if the lookahead is '`minus`', shift, if the lookahead is end-of-file, reduce the rule '`expr: 'num'`', and any other token is an error (think of '`number number`' for instance).

The theory we presented in the Section 1.1 [Looking for Balanced Expressions], page 1 can be extended to recognize patterns on the stack plus one lookahead symbol. This is called LR(1) parsing. As you will have surely guessed, LR(k) denotes this technique when peeking at the k next tokens in the input.

Unfortunately, even for reasonably simple grammars the automaton is quickly huge, so in practice, one limits herself to $k = 1$. Yet with a single lookahead, the automaton remains huge: a lot of research has been devoted to design reasonable limitations or compression techniques to have it fit into a reasonable machine. A successful limitation, which description falls out of the scope of this book, is known as LALR(1). A promising approach, DRLR, consists in recognizing the stack from its top instead of from its bottom. Intuitively there are less patterns to recognize since the pertinent information is usually near the top, hence the automaton is smaller.

In *A Discriminative Reverse Approach to LR(k) Parsing*, Fortes Gálvez José compares the handling of three grammars:

arithmetics
: A small grammar similar to that of the *example 7.4*, but unambiguous. There are 5 terminals, 3 nonterminals, and 6 rules.

medium
: A medium size grammar comprising 41 terminals, 38 nonterminals, and 93 rules.

programming
: A real size grammar of a programming language, composed of 82 terminals, 68 nonterminals, and 234 rules.

by the three approaches. The size of the automaton is measured by its number of states and *entries* (roughly, its transitions):

Grammar	LR(1) States	LR(1) Entries	LALR(1) States	LALR(1) Entries	DRLR(1) States	DRLR(1) Entries
arithmetics	22	71	12	26	8	23
medium	773	6,874	173	520	118	781
programming	1,000+	15,000+	439	3,155	270	3,145

As you can see, LR(1) parsers are too expensive and as matter of fact there is no wide spread and used implementation, LALR(1) is reasonable and can be found in numerous tools such as Yacc and all its clones, and finally, because DRLR(1) addresses all the LR(1) grammars, it is an appealing alternative to Yacc. Bison is an implementation of Yacc, hence it handles LALR(1) grammars, but might support DRLR(1) some day, providing the full expressive power of LR(1).

1.3 What is Bison

Yacc is a generator of efficient parsers. A *parser* is a program or routine which recognizes the structure of sentences. Yacc's input is composed of *rules* with associated actions. The rules must be *context free*, i.e., their left hand side is composed of a single nonterminal symbol, and their right hand side is composed of series of terminal and nonterminal symbols. When a rule is reduced, the associated C code is triggered.

Yacc is based on pushdown automata. It is a implementation of the LALR(1) parsing algorithm, which is sufficient for most programming languages, but can be too limited a framework to describe conveniently intricate languages.

Yacc, and all its declinations (CAMLYacc for CAML etc.) are used in numerous applications, especially compilers and interpreters. Hence its name: Yet Another Compiler Compiler.

Bison is a free software implementation of Yacc, as described by the POSIX standard. It provides a wide set of additional options and features, produces self contained portable code (no library is required), supports a more pleasant syntax, and stands as a standard of its own. Since most Yacc do have problems (inter-Yacc compatibility and actual bugs), all the reasons are in favor of using exclusively Bison: the portable C code it produces can be shipped in the package and will compile cleanly on the user's machine. It imposes no restriction on the license of the produced parser.

It is used by the GNU Compiler Collection for C, C++[3], the C preprocessor. Many other programming language tools use Bison or Yacc: GNU AWK, Perl, but it proves itself useful in reading structured files: a2ps uses it to parse its style sheets. It also helps decoding some limited forms of natural language: numerous GNU utilities use a Yacc grammar to decode dates such as '`2 days ago`', '`3 months 1 day`', '`25 Dec`', '`1970-01-01 00:00:01 UTC +5 hours`' etc.

GNU Gettext deserves a special mention with three different uses: one to parse its input files, another one to parse the special comments in these files, and one to evaluate the foreign language dependent rules defining the plural forms.

1.4 Bison as a Grammar Checker

Bison is dedicated to artificial languages, which, contrary to natural languages, must be absolutely unambiguous. More precisely it accepts only LALR(1) grammars, which formal definition amounts exactly to "parsable with a Bison parser". Although there are unambiguous grammars that are not LALR(1), we will henceforth use "ambiguous", or "insane", to mean "not parsable with Bison".

While the main purpose of Bison is to generate parsers, since writing good grammars is not an easy task, it proves itself very useful as a grammar checker (e.g., checking it is unambiguous). Again, a plain "error: grammar is insane" would be an information close to useless, fortunately Bison then provides means (i) to understand the insanity of a grammar, (ii) to solve typical ambiguities conveniently.

---

[3] There are plans to rewrite the C++ parser by hand because the syntax of the language is too intricate for LALR(1) parsing.

How exactly does an ambiguity reveal itself to Bison? We saw that Bison parsers are simple machines shifting tokens and reducing rules. As long as these machines know exactly what step to perform, the grammar is obviously sane. In other words, on an insane grammar it will sometimes hesitate between different actions: should I shift, or should I reduce? And if I reduce, which rule should I reduce?

In Section 1.2 [Looking for Arithmetics], page 4, we demonstrated that the naive implementation arithmetics is ambiguous, even if reduced to subtraction. The following Bison file focuses on it. The mark '%%' is needed for Bison to know where the grammar starts:

```
%%
expr: expr '-' expr | "number";
```

Example 7.8: 'arith-1.y' – *A Shift/Reduce Conflict*

Running bison on it, as expected, ends sadly:

```
$ bison arith-1.y
error arith-1.y contains 1 shift/reduce conflict.
```

As announced, ambiguities lead Bison to hesitate between several actions, here '1 shift/reduce conflict' stands for "there is a state from which I could either shift another token, or reduce a rule". But which? You may ask bison for additional details:

```
$ bison --verbose arith-1.y
error arith-1.y contains 1 shift/reduce conflict.
```

which will output the file 'arith-1.output'[4]:

```
State 4 contains 1 shift/reduce conflict.

Grammar
    rule 0    $axiom -> expr $
    rule 1    expr   -> expr '-' expr
    rule 2    expr   -> "number"

Terminals, with rules where they appear
    $         (-1)
    '-'       (45) 1
    error    (256)
    "number" (257) 2

Nonterminals, with rules where they appear
    expr (5) on left: 1 2, on right: 1

State 0    $axion -> . expr $           (rule 0)
           expr   -> . expr '-' expr    (rule 1)
           expr   -> . "number"         (rule 2)
           "number"    shift, and go to state 1
           expr        go to state 2

State 1    expr  ->  "number" .         (rule 2)
           $default    reduce using rule 2 (expr)
```

[4] In addition to minor formatting changes, this output has been modified. In particular the rule 0, which Bison hides, is made explicit.

```
State 2    $axiom -> expr . $           (rule 0)
           expr   -> expr . '-' expr    (rule 1)
           $           go to state 5
           '-'         shift, and go to state 3
State 3    expr -> expr '-' . expr      (rule 1)
           "number"    shift, and go to state 1
           expr        go to state 4
State 4    expr -> expr . '-' expr      (rule 1)
           expr -> expr '-' expr .      (rule 1)
           '-'         shift, and go to state 3
           '-'         [reduce using rule 1 (expr)]
           $default    reduce using rule 1 (expr)
State 5    $axiom -> expr . $           (rule 0)
           $           shift, and go to state 6
State 6    $axiom -> expr $ .           (rule 0)
           $default    accept
```

You shouldn't be frightened by the contents: aside from being applied to a different grammar, this is nothing but the FSR we presented in *example 7.2*. Instead of being presented graphically, it is described by the list of its states and the actions that can be performed from each state. Another difference is that each state is represented by the degree of recognition of the various rules, instead of regular expressions.

For instance the state 3 is characterized by the fact that we recognized the first two symbols of '`expr '-' expr`', which is denoted '`expr -> expr '-' . expr`'. In that state, if we see a '`number`', we shift it, and go to state 1. If we see an '`expr`', we go to state 4. The reader is suggested to draw this automaton.

Bison draws our attention on the state 4, for "State 4 contains 1 shift/reduce conflict". Indeed, there are two possible actions when the lookahead, the next token in the input, is '-':

```
State 4    expr  ->  expr . '-' expr    (rule 1)
           expr  ->  expr '-' expr .    (rule 1)
           '-'         shift, and go to state 3
           '-'         [reduce using rule 1 (expr)]
           $default    reduce using rule 1 (expr)
```

Example 7.9: *State 4 contains 1 shift/reduce conflict*

We find again the ambiguity of our grammar: when reading '`number - number - number`', which leads to the stack being '`expr '-' expr`', when finding another '`-`', it doesn't know if it should:

– shift it ('`shift, and go to state 3`'), which results in

Stack	Input	shift 3
`0 expr 2 - 3 expr 4`	`- number $`	shift 1
`0 expr 2 - 3 expr 4 - 3`	`number $`	shift 1
`0 expr 2 - 3 expr 4 - 3 number 1`	`$`	reduce 2
`0 expr 2 - 3 expr 4 - 3 expr`	`$`	go to 4
`0 expr 2 - 3 expr 4 - 3 expr 4`	`$`	reduce 1

`0 expr 2 - 3 expr`	`$`	go to 4
`0 expr 2 - 3 expr 4`	`$`	reduce 1
`0 expr`	`$`	go to 5
`0 expr 5`	`$`	shift 6
`0 expr 5 $ 6`		accept

i.e., grouping the last two expressions, in other words understanding '`number - (number - number)`', or

– reduce rule 1 ('`[reduce using rule 1 (expr)]`'), which results in

`0 expr 2 - 3 expr 4`	`- number $`	reduce 1
`0 expr`	`- number $`	go to 2
`0 expr 2`	`- number $`	shift 3
`0 expr 2 - 3`	`number $`	shift 1
`0 expr 2 - 3 number 1`	`$`	reduce 2
`0 expr 2 - 3 expr`	`$`	goto 4
`0 expr 2 - 3 expr 4`	`$`	reduce 1
`0 expr`	`$`	go to 5
`0 expr 5`	`$`	shift 6
`0 expr 5 $ 6`		accept

i.e., grouping the first two expressions, in other words understanding '`(number - number) - number`'.

Well meaning, it even chose an alternative for us, shifting, which Bison signifies with the square brackets: possibility '`[reduce using rule 1 (expr)]`' will not be followed. Too bad, that's precisely not the choice we need...

The cure is well known: implement the grammar of the *example 7.6*. The reader is invited to implement it in Bison, and check the output file. In fact, the only difference between the two output files is that the state 4 is now:

```
State 4    expr  ->  expr '-' "number" .    (rule 1)
           $default    reduce using rule 1 (expr)
```

With some practice, you will soon be able to understand how to react to conflicts, spotting where they happen, and find another grammar which is unambiguous. Unfortunately...

1.5 Resolving Conflicts

Unfortunately, naive grammars are often ambiguous. The naive grammar for arithmetics, see *example 7.4*, is well-known for its ambiguities, unfortunately finding an unambiguous equivalent grammar is quite a delicate task and requires some expertise:

```
expr: expr '+' term
    | expr '-' term
    | term;
term: term '*' factor
    | term '/' factor
    | factor;
factor: '(' expr ')'
      | "number";
```

Example 7.10: *An Unambiguous Grammar for Arithmetics*

Worse yet: it makes the grammar more intricate to write. For instance, a famous ambiguity in the naive grammar for the C programming language is the "dangling `else`". Consider the following grammar excerpt:

```
%%
stmt: "if" "expr" stmt "else" stmt
    | "if" "expr" stmt
    | "stmt";
```

Example 7.11: '`ifelse-1.y`' – *Ambiguous grammar for* `if`/`else` *in C*

Although innocent looking, it is ambiguous and leads to two different interpretations of:

```
if   (expr-1)
if   (expr-2)
     statement-1
else statement-2
```

depending whether *statement-2* is bound to the first or second `if`. The C standard clearly mandates the second interpretation: a "dangling else" is bound to the closest `if`. How can one write a grammar for such a case? Again, the answer is not obvious, and requires some experience with grammars. It requires the distinction between "closed statements", i.e., those which `if` are saturated (they have their pending `else`), and non closed statements:

```
%%
stmt: closed_stmt
    | non_closed_stmt;
closed_stmt: "if" "expr" closed_stmt "else" closed_stmt
           | "stmt";
non_closed_stmt: "if" "expr" stmt
               | "if" "expr" closed_stmt "else" non_closed_stmt;
```

Example 7.12: '`ifelse-2.y`' – *Unambiguous Grammar for* `if`/`else` *in C*

And finally, since we introduce new rules, new symbols, our grammar gets bigger, hence the automaton gets fat, therefore it becomes less efficient (since modern processors cannot make full use of their caches with big tables).

Rather, let's have a closer look at *example 7.9*. There are two actions fighting to get into state 4: reducing the rule 1, or shifting the token '`-`'. We have a match between two opponents: if the rule wins, then '`-`' is right associative, if the token wins, then '`-`' is left associative.

What we would like to say is that "shifting '`-`' has priority over reducing '`expr: expr '-' expr`'". Bison is nice to us, and allows us to inform it of the associativity of the operator, it will handle the rest by itself:

```
%left '-'
%%
expr: expr '-' expr | "number";
```

Example 7.13: '`arith-3.y`' – *Using '`%left`' to Solve Shift/Reduce Conflicts*

```
$ bison --verbose arith-3.y
```

This is encouraging, but won't be enough for us to solve the ambiguity of the *example 7.5*:

```
%left '+'
%%
expr: expr '*' expr | expr '+' expr | "number";
```

Example 7.14: '`arith-4.y`' – *An Ambiguous Grammar for '`*`' vs. '`+`'*

```
$ bison --verbose arith-4.y
error arith-4.y contains 3 shift/reduce conflicts.
```

Diving into '`arith-4.output`' will help us understand the problem:

```
Conflict in state 5 between rule 2 and token '+' resolved as reduce.
State 5 contains 1 shift/reduce conflict.
State 6 contains 2 shift/reduce conflicts.
...
State 5     expr  ->  expr . '*' expr   (rule 1)
            expr  ->  expr '*' expr .   (rule 1)
            expr  ->  expr . '+' expr   (rule 2)
            '*'         shift, and go to state 3
            '*'         [reduce using rule 1 (expr)]
            $default    reduce using rule 1 (expr)
State 6     expr  ->  expr . '*' expr   (rule 1)
            expr  ->  expr '*' expr .   (rule 1)
            expr  ->  expr . '+' expr   (rule 2)
            '+'         shift, and go to state 3
            '*'         shift, and go to state 4
            '+'         [reduce using rule 1 (expr)]
            '*'         [reduce using rule 1 (expr)]
            $default    reduce using rule 1 (expr)
```

First note that it reported its understanding of '`%left '+'`': it is a match opposing token '`+`' against rule 2, and the rule is the winner.

State 6, without any surprise, has a shift/reduce conflict because we didn't define the associativity of '`*`', but states 5 and 6 have a new problem. In state 5, after having read '`expr + expr`', in front of a '`*`', should it shift it, or reduce '`expr + expr`'? Obviously it should reduce: the rule containing '`*`' should have precedence over the token '`+`'. Similarly, in state 6, the token '`*`' should have precedence over the rule containing '`+`'. Both cases can be summarized as "'`*`' has precedence over '`+`'".

Bison allows you to specify precedences simply by listing the '`%left`' and '`%right`' from the lowest precedence, to the highest. Some operators have equal precedence, for instance series of '`+`' and '`-`' must always be read from left to right: '`+`' and '`-`' have the same precedence. In this case, just put the two operators under the same '`%left`'.

Finally, we are now able to express our complete grammar of arithmetics:

```
%left '+' '-'
%left '*' '/'
```

```
%%
expr: expr '*' expr | expr '/' expr
    | expr '+' expr | expr '-' expr
    | '(' expr ')'
    | "number";
```

Example 7.15: 'arith-5.y' – *Using Precedence to Solve Shift/Reduce Conflicts*

which is happily accepted by `bison`:

```
$ bison --verbose arith-5.y
```

The reader is invited to:

Implement the grammar of the *example 7.10*
: Read the output file, and see that indeed it requires a bigger automaton than the grammar of the *example 7.15*.

Play with the option '`--graph`'
: A complement of '`--verbose`' is '`--graph`', which outputs a VCG graph, to be viewed with `xvcg`. View the automata of the various grammars presented.

Explain the Failure of Factored Arithmetics
: Bison refuses the following grammar:

```
%left '+' '-'
%left '*' '/'
%%
expr: expr op expr | "number";
op: '+' | '-' | '*' | '/';
```

Example 7.16: 'arith-6.y' – *Failing to Use Precedence*

```
$ bison arith-6.y
error arith-6.y contains 4 shift/reduce conflicts.
```

Can you explain why? Bear in mind the nature of the opponents in a shift/reduce match.

Solve the dangling `else`
: The precedence of a rule is actually that of its *last* token. With this in mind, propose a simple implementation of the grammar of *example 7.10* in Bison.

We are now fully equipped to implement real parsers.

1.6 Simple Uses of Bison

Bison is a source generator, just as Gperf, Flex, and others. It takes a list of rules and actions, and produces a function triggering the action associated to the reduced rule. The input syntax allows for the traditional prologue, containing Bison directives and possibly some user declarations and initializations, and the epilogue, typically additional functions:

```
%{
  user-prologue
%}
bison-directives
%%
/* Comments. */
lfs:
    rhs-1 { action-1 }
  | rhs-2 { action-2 }
  | ...
  ;
...
%%
user-epilogue
```

Example 7.17: *Structure of a Bison Input File*

When run on a file 'foo.y', `bison` produces a file named 'foo.tab.c' and possibly 'foo.tab.h', 'foo.output', etc. This atrocious naming convention is a mongrel between the POSIX standards on the one hand requiring the output to be *always* named 'y.tab.c', and on the other hand the more logical 'foo.c'. You may set the output file name thanks to the `%output="`*parser-filename*`"` directive.

This file is a C source including, in addition to your *user-prologue* and *user-epilogue*, one function:

int yyparse () Function

Parse the whole stream of tokens delivered by successive calls to `yylex`. Trigger the *action* associated to the reduced rule.

Return 0 for success, and nonzero upon failures (such as parse errors).

You must provide `yylex`, but also `yyerror`, used to report parse errors.

For a start, we will implement the *example 7.1*. Because writing a scanner, which splits the input into tokens, is a job in itself, see ⟨undefined⟩ [Scanning with Gperf and Flex], page ⟨undefined⟩, we will simply rely on the command line arguments. In other words, we use the shell as a scanner (or rather, we let the user fight with its shells to explain where start and end the tokens).

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

#define yyerror(Msg) error (EXIT_FAILURE, 0, "%s", Msg)
static int yylex (void);
%}
%output="brackens-1.c"
```

```
%%
expr: '(' expr ')'
    | 'N'
    ;
%%
/* The current argument. */
static const char **args = NULL;

static int
yylex (void)
{
  /* No token stands for end of file. */
  if (!*++args)
    return 0;
  /* Parens stand for themselves.  'N' denotes a number. */
  if (strlen (*args) == 1 && strchr ("()N", **args))
    return **args;
  /* An error. */
  error (EXIT_FAILURE, 0, "invalid argument: %s", *args);
  /* Pacify GCC that knows ERROR may return. */
  return -1;
}

int
main (int argc, const char *argv[])
{
  args = argv;
  return yyparse ();
}
```

Example 7.18: '`brackens-1.y`' – *Nested Parentheses Checker*

Which we compile and run:

```
$ bison brackens-1.y
$ gcc -Wall brackens-1.c -o brackens-1
$ ./brackens-1 '(' 'N' ')'
$ ./brackens-1 '(' 'n' ')'
error ./brackens-1: invalid argument: n
$ ./brackens-1 '(' '(' 'N' ')' ')'
$ ./brackens-1 '(' '(' 'N' ')' ')' ')'
error ./brackens-1: parse error
```

It works quite well! Unfortunately, when given an invalid input is it quite hard to find out where the actual error is. Fortunately you can ask Bison parsers to provide more precise information by defining `YYERROR_VERBOSE` to 1. You can also use the Bison directive `%debug` which introduces a variable, `yydebug`, to make your parser verbose:

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

#define yyerror(Msg) error (EXIT_FAILURE, 0, "%s", Msg)
static int yylex (void);
#define YYERROR_VERBOSE 1
%}
%debug
%output="brackens-2.c"
%%
expr: '(' expr ')'
    | 'N'
    ;
%%
int
main (int argc, const char *argv[])
{
  yydebug = getenv ("YYDEBUG") ? 1 : 0;
  args = argv;
  return yyparse ();
}
```

Example 7.19: '`brackens-2.y`' – *Nested Parentheses Checker*

which, when run, produces:

```
$ ./brackens-2 '(' '(' 'N' ')' ')' ')'
./brackens-2: parse error, unexpected ')', expecting $
```

Wow! The error message is *much* better! But it is still not clear which one of the arguments is provoking the error. Asking traces from the parser gives some indication:

```
$ YYDEBUG=1 ./brackens-2 '(' '(' 'N' ')' ')' ')'
Starting parse
Entering state 0
Reading a token: Next token is 40 ('(')
Shifting token 40 ('('), Entering state 1
Reading a token: Next token is 40 ('(')
Shifting token 40 ('('), Entering state 1
Reading a token: Next token is 78 ('N')
Shifting token 78 ('N'), Entering state 2
Reducing via rule 2 (line 32), 'N'  -> expr
state stack now 0 1 1
Entering state 3
Reading a token: Next token is 41 (')')
Shifting token 41 (')'), Entering state 4
Reducing via rule 1 (line 32), '(' expr ')'  -> expr
state stack now 0 1
```

```
Entering state 3
Reading a token: Next token is 41 (')')
Shifting token 41 (')'), Entering state 4
Reducing via rule 1 (line 32), '(' expr ')'  -> expr
state stack now 0
Entering state 5
Reading a token: Next token is 41 (')')
error ./brackens-2: parse error, unexpected ')', expecting $
```

The reader is invited to compare these traces with the "execution" by hand of the *example 7.3*: it is just the same!

Two things are worth remembering. First, always try to produce precise error messages, since once an error is diagnosed, the user still has to locate it in order to fix it. I, for one, don't like tools that merely report the line where the error occurred, because often *several* very similar expressions within that line can be responsible for the error. In that case, I often split my line into severals, recompile, find the culprit, educate it, and join again these lines... And second, never write dummy `printf` to check that your parser works: it is a pure waste of time, since Bison does this on command, and produces extremely readable traces. Take some time to read the example above, and in particular spot on the one hand side the tokens and the shifts and on the other hand the reductions and the rules. But forget about "states", for the time being.

1.7 Using Actions

Of course real applications need to process the input, not merely to validate it as in the previous section. For instance, when processing arithmetics, one wants to compute the result, not just say "this is a valid/invalid arithmetical expression".

We stem on several problems. First of all, up to now we managed to use single characters as tokens: the code of the character is used as token number. But all the integers must be mapped to a single token type, say `INTEGER`.

Bison provide the `%token` directive to declare token types:

```
%token INTEGER FLOAT STRING
```

Secondly, tokens such as numbers have a value. To this end, Bison provides a global variable, `yylval`, which the scanner must fill with the appropriate value. But imagine our application also had to deal with strings, or floats etc.: we need to be able to specify several value types, and associate a value type to a token type.

To declare the different value types to Bison, use the `%union` directive. For instance:

```
%union
{
  int   ival;
  float fval;
  char *sval;
}
```

This results in the definition of the type “token value”: `yystype`[5]. The scanner needs the token types and token value types: if given ‘`--defines`’ `bison` creates ‘`foo.h`’ which contains their definition. Alternatively, you may use the `%defines` directive.

Then map token types to value types:

```
%token <ival> INTEGER
%token <fval> FLOAT
%token <sval> STRING
```

But if tokens have a value, then so have some nonterminals! For instance, if `iexpr` denotes an integer expression, then we must also specify that (i) it has a value, (ii) of type `INTEGER`. To this end, use `%type`:

```
%type <ival> iexpr
%type <fval> fexpr
%type <sval> sexpr
```

We already emphasized the importance of traces: together with the report produced thanks to the option ‘`--verbose`’, they are your best partners to debug a parser. Bison lets you improve the traces of tokens, for instance to output token values in addition to token types. To use this feature, just define the following macro:

YYPRINT (*File, Type, Value*) Macro

Output on *File* a description of the *Value* of a token of *Type*.

For various technical reasons, I suggest the following pattern of use:

```
%{
...
#define YYPRINT(File, Type, Value) yyprint (File, Type, &Value)
static void yyprint (FILE *file, int type, const yystype *value);
...
%}
%%
...
%%
static void
yyprint (FILE *file, int type, const yystype *value)
{
  switch (type)
    {
    case INTEGER:
      fprintf (file, " = %d", value->ival);
      break;
    case FLOAT:
      fprintf (file, " = %f", value->fval);
      break;
    case STRING:
      fprintf (file, " = \"%s\"", value->sval);
      break;
```

[5] Because of the compatibility with POSIX Yacc, the type `YYSTYPE` is also defined. For sake of inter Yacc portability, use the latter only. But for sake of your mental balance, use Bison only.

```
    }
}
```

The most important difference between a mere validation of the input and an actual computation, is that we must be able to equip the parser with actions. For instance, once we have recognized that '`1 + 2`' is a '`INTEGER + INTEGER`', we must be able to (i) compute the sum, which requires fetching the value of the first and third tokens, and (ii) "return" 3 as the result.

To associate an action with a rule, just put the action after the rule, between braces. To access the value of the various symbols of the right hand side of a rule, Bison provides pseudo-variables: '`$1`' is the value of the first symbol, '`$2`' is the second etc. Finally to "return" a value, or rather, to set the value of the left hand side symbol of the rule, use '`$$`'.

Therefore, a calculator typically includes:

```
iexpr: iexpr '+' iexpr { $$ = $1 + $3 }
     | iexpr '-' iexpr { $$ = $1 - $3 }
     | iexpr '*' iexpr { $$ = $1 * $3 }
     | iexpr '/' iexpr { $$ = $1 / $3 }
     | INTEGER         { $$ = $1 }
     ;
```

Please, note that we used '`$3`' to denote the value of the third symbol, even though the second, the operator, has no value.

Putting this all together leads to the following implementation of the *example 7.15*:

```
%{ /* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>
#include "calc.h"

#define YYERROR_VERBOSE 1
#define yyerror(Msg) error (EXIT_FAILURE, 0, "%s", Msg)

#define YYPRINT(File, Type, Value) yyprint (File, Type, &Value)
static void yyprint (FILE *file, int type, const yystype *value);

static int yylex (void);
%}
```

```
%debug
%defines
%output="calc.c"
%union
{
  int ival;
}
%token <ival> NUMBER
%type  <ival> expr input
%left '+' '-'
%left '*' '/'
%%
input: expr         { printf ("%d\n", $1) }
     ;
expr: expr '*' expr { $$ = $1 * $3 }
    | expr '/' expr { $$ = $1 / $3 }
    | expr '+' expr { $$ = $1 + $3 }
    | expr '-' expr { $$ = $1 - $3 }
    | '(' expr ')'  { $$ = $2 }
    | NUMBER        { $$ = $1 }
    ;
%%
/* The current argument. */
static const char **args = NULL;
static void
yyprint (FILE *file, int type, const yystype *value)
{
  switch (type)
    {
    case NUMBER:
      fprintf (file, " = %d", value->ival);
      break;
    }
}
```

```
static int
yylex (void)
{
  /* No token stands for end of file. */
  if (!*++args)
    return 0;
  /* Parens and operators stand for themselves. */
  if (strlen (*args) == 1 && strchr ("()+-*/", **args))
    return **args;
  /* Integers have a value. */
  if (strspn (*args, "0123456789") == strlen (*args))
    {
      yylval.ival = strtol (*args, NULL, 10);
      return NUMBER;
    }
  /* An error. */
  error (EXIT_FAILURE, 0, "invalid argument: %s", *args);
  /* Pacify GCC which knows ERROR may return. */
  return -1;
}

int
main (int argc, const char *argv[])
{
  yydebug = getenv ("YYDEBUG") ? 1 : 0;
  args = argv;
  return yyparse ();
}
```

Example 7.20: ‘`calc.y`’ – *A Simple Integer Calculator*

Its execution is satisfying:

```
$ bison calc.y
$ gcc calc.c -o calc
$ ./calc 1 + 2 \* 3
7
$ ./calc 1 + 2 \* 3 \*
error ./calc: parse error, unexpected $, expecting NUMBER or '('
$ YYDEBUG=1 ./calc 51
Starting parse
Entering state 0
Reading a token: Next token is 257 (NUMBER = 51)
Shifting token 257 (NUMBER), Entering state 1
Reducing via rule 7 (line 56), NUMBER  -> expr
state stack now 0
Entering state 3
Reading a token: Now at end of input.
Reducing via rule 1 (line 48), expr  -> input
51
state stack now 0
```

```
Entering state 14
Now at end of input.
Shifting token 0 ($), Entering state 15
Now at end of input.
```

well, almost

```
$ ./calc 1 / 0
floating point exception  ./calc 1 / 0
```

1.8 Advanced Use of Bison

The example of the previous section is very representative of real uses of Bison, except for its scale. Nevertheless, there remains a few issues to learn before your Bison expert graduation. In this section, we complete a real case example started in ⟨undefined⟩ [Advanced Use of Flex], page ⟨undefined⟩: a reimplementation of M4's `eval` builtin, using Flex and Bison.

Bison parsers are often used on top of Flex scanners. As already explained, the scanner then needs to know the definition of the token types, and the token value types: to this end use '`--defines`' or `%defines` to produce a header containing their definition.

The real added value of good parsers over correct parsers is in the handling of errors: the accuracy and readability of the error messages, and their ability to proceed as much as possible instead of merely dying at the first glitch[6].

It is an absolute necessity for error messages to specify the precise location of the errors. To this end, when given '`--locations`' or `%locations`, `bison` provides an additional type, `yyltype` which defaults to:

```
typedef struct yyltype
{
  int first_line;
  int first_column;

  int last_line;
  int last_column;
} yyltype;
```

and another global variable, `yylloc`, which the scanner is expected to set for each token (see ⟨undefined⟩ [Advanced Use of Flex], page ⟨undefined⟩). Then, the parser will automatically keep track of the locations not only of the tokens, but also of nonterminals. For instance, it knows that the location of '`1 + 20 * 300`' starts where '`1`' starts, and ends where '`300`' does. As with '`$$`', '`$1`' etc., you may use '`@$`', '`@1`' to set/access the location of a symbol, for instance in "division by 0" error messages.

It is unfortunate, but the simple result of history, that improving the error messages requires some black incantations:

```
#define YYERROR_VERBOSE 1
#define yyerror(Msg) yyerror (&yylloc, Msg)
```

in other words, even with `%locations` Bison does not pass the location to `yyerror` by default. Similarly, enriching the traces with the locations requires:

[6] No one would ever accept a compiler which reports only the first error it finds, and then exits!

```
#define YYPRINT(File, Type, Value) yyprint (File, &yylloc, Type, &Value)
```

In the real world, using a fixed set of names such as `yylex`, `yyparse`, `yydebug` and so on is wrong: you may have several parsers in the same program, or, in the case of libraries, you must stay within your own name space. Similarly, using global variables such as `yylval`, `yylloc` is dangerous, and prevents any recursive use of the parser.

These two problems can be solved with Bison: use '`%name-prefix="`*prefix*`"`' to rename of the '`yyfoo`'s into '*prefix*`foo`'s, and use `%pure-parser` to make Bison define `yylval` etc. as variables of `yyparse` instead of global variables.

If you are looking for global-free parsers, then, obviously, you need to be able to exchange information with `yyparse`, otherwise, how could it return something to you! Bison provides an optional user parameter, which is typically a structure in which you include all the information you might need. This structure is conventionally called a *parser control structure*, and for consistency I suggest naming it `yycontrol`. To define it, merely define `YYPARSE_PARAM`:

```
#define YYPARSE_PARAM yycontrol
```

Moving away from '`yy`' via `%name-prefix`, and from global variables via `%pure-parser` also deeply change the protocol between the scanner and the parser: before, `yylex` expected to find a global variable named `yylval`, but now it is named `foolval` and it is not global any more!

Scanners written with Flex can easily be adjusted: give it '`%option prefix="`*prefix*`"`' to change the '`yy`' prefix, and explain, (i) to Bison that you want to pass an additional parameter to `yylex`:

```
#define YYLEX_PARAM yycontrol
```

and (ii) on the other hand, to Flex that the prototype of `yylex` is to be changed:

```
#define YY_DECL \
  int yylex (yystype *yylval, yyltype *yylloc, yycontrol_t *yycontrol)
```

Putting this all together for our `eval` reimplementation gives:

```
%debug
%defines
%locations
%pure-parser
%name-prefix="yleval_"

%{
#if HAVE_CONFIG_H
#  include <config.h>
#endif

#include <m4module.h>
#include "yleval.h"
```

```
/* Pass the control structure to YYPARSE and YYLEX. */
#define YYPARSE_PARAM yleval_control
#define YYLEX_PARAM yleval_control
/* YYPARSE receives YLEVAL_CONTROL as a void *.  Provide a
   correctly typed access to it.  */
#define yycontrol ((yleval_control_t *) yleval_control)

/* Request detailed parse error messages, and pass them to
   YLEVAL_ERROR. */
#define YYERROR_VERBOSE 1
#undef  yyerror
#define yyerror(Msg) \
        yleval_error (yycontrol, &yylloc, Msg)

/* When debugging our pure parser, we want to see values and locations
   of the tokens.  */
#define YYPRINT(File, Type, Value) \
        yyprint (File, &yylloc, Type, &Value)
static void yyprint (FILE *file, const yyltype *loc,
                     int type, const yystype *value);

%}

/* Only NUMBERS have a value.  */
%union
{
  number number;
};
```

The name of the tokens is an implementation detail: the user has no reason to know `NUMBER` is your token type for numbers. Unfortunately, these token names appear in the verbose error messages, and we want them! For instance, in the previous section, we implemented `calc`:

```
$ ./calc '(' 2 + ')'
error ./calc: parse error, unexpected ')', expecting NUMBER or '('
```

Bison lets you specify the "visible" symbol names to produce:

```
$ ./calc '(' 2 + ')'
error ./calc: parse error, unexpected ')', expecting "number" or '('
```

which is not perfect, but still better. In our case:

```
/* Define the tokens together with there human representation. */
%token YLEVAL_EOF 0 "end of string"
%token <number> NUMBER "number"
%token LEFTP "(" RIGHTP ")"
%token LOR  "||"
%token LAND "&&"
%token OR   "|"
%token XOR  "^"
%token AND  "&"
%token EQ "=" NOTEQ "!="
%token GT ">" GTEQ ">=" LS "<" LSEQ "<="
```

```
%token LSHIFT "<<" RSHIFT ">>"
%token PLUS "+" MINUS "-"
%token TIMES "*" DIVIDE "/" MODULO "%" RATIO ":"
%token EXPONENT "**"
%token LNOT "!" NOT "~" UNARY

%type <number> exp
```

There remains to define the precedences for all our operators, which ends our prologue:

```
/* Specify associativities and precedences from the lowest to the
   highest.  */
%left LOR
%left LAND
%left OR
%left XOR
%left AND
/* These operators are non associative, in other words, we forbid
   '-1 < 0 < 1'.  C allows this, but understands it as
   '(-1 < 0) < 1' which evaluates to... false.  */
%nonassoc EQ NOTEQ
%nonassoc GT GTEQ LS LSEQ
%nonassoc LSHIFT RSHIFT
%left  PLUS MINUS
%left  TIMES DIVIDE MODULO RATIO
%right EXPONENT
/* UNARY is used only as a precedence place holder for the
   unary plus/minus rules.  */
%right LNOT NOT UNARY
%%
```

The next section of '`ylparse.y`' is the core of the grammar: its rules. The very first rule deserves special attention:

```
result: { LOCATION_RESET (yylloc) } exp { yycontrol->result = $2 }
```

it aims at initializing `yylloc` before the parsing actually starts. It does so via a so called *mid-rule action*, i.e., a rule which is executed before the whole rule is recognized[7]. Then it looks for a single expression, which value consists in the result of the parse: we store it in the parser control structure.

The following chunk addresses the token `NUMBER`. It relies on the default action: '`$$ = $1`'.

---

[7] The attentive reader will notice that the notion of mid-rule action does not fit with LALR(1) techniques: an action can only be performed once a whole rule is recognized. The paradox is simple to solve: internally Bison rewrites the above fragment as

```
result: @1 exp    { yycontrol->result = $2 };
@1: /* Empty. */  { LOCATION_RESET (yylloc) };
```

where '`@1`' is a fresh nonterminal. You are invited to read the '`--verbose`' report which does include these invisible symbols and rules.

```
/* Plain numbers. */
exp:
  NUMBER
;

/* Parentheses. */
exp:
  LEFTP exp RIGHTP { $$ = $2 }
;
```

The next bit of the grammar describes arithmetics. Two treatments deserve attention:

Unary Operators
: We want the unary minus and plus to bind extremely tightly: their precedence is higher than that binary plus and minus. But, as already explained, the precedence of a rule is that of its last token... by default... To override this default, we use `%prec` which you can put anywhere in the rule. The rules below read as "the rules for unary minus and plus have the precedence of the precedence place holder `UNARY`".

Semantic Errors
: Not all the exponentiations are valid ('`2 ^ -1`'), nor are all the divisions and moduli ('`1 / 0`', '`1 % 0`'). When such errors are detected, we abort the parsing by invoking `YYABORT`.

```
/* Arithmetics. */
exp:
  PLUS  exp        { $$ =   $2 }   %prec UNARY
| MINUS exp        { $$ = - $2 }   %prec UNARY
| exp PLUS     exp { $$ = $1 + $3 }
| exp MINUS    exp { $$ = $1 - $3 }
| exp TIMES    exp { $$ = $1 * $3 }
| exp DIVIDE   exp { if (!yldiv (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp MODULO   exp { if (!ylmod (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp RATIO    exp { if (!yldiv (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
| exp EXPONENT exp { if (!ylpow (yycontrol, &@$, &$$, $1, $3)) YYABORT; }
;
/* Booleans. */
exp:
  LNOT exp     { $$ = ! $2 }
| exp LAND exp { $$ = $1 && $3 }
| exp LOR  exp { $$ = $1 || $3 }
;
```

```
/* Comparisons. */
exp:
  exp EQ    exp { $$ = $1 == $3 }
| exp NOTEQ exp { $$ = $1 != $3 }
| exp LS    exp { $$ = $1 <  $3 }
| exp LSEQ  exp { $$ = $1 <= $3 }
| exp GT    exp { $$ = $1 >  $3 }
| exp GTEQ  exp { $$ = $1 >= $3 }
;

/* Bitwise. */
exp:
  NOT exp         { $$ = ~ $2 }
| exp AND    exp { $$ = $1 & $3 }
| exp OR     exp { $$ = $1 | $3 }
| exp LSHIFT exp { $$ = $1 << $3 }
| exp RSHIFT exp { $$ = $1 >> $3 }
;
%%
```

Finally, the epilogue of our parser consists in the definition of the tracing function, `yyprint`:

```
/*------------------------------------------------------------------.
| When debugging the parser, display tokens' locations and values.  |
`------------------------------------------------------------------*/

static void
yyprint (FILE *file,
         const yyltype *loc, int type, const yystype *value)
{
  fputs (" (", file);
  LOCATION_PRINT (file, *loc);
  fputs (")", file);
  switch (type)
    {
    case NUMBER:
      fprintf (file, " = %ld", value->number);
      break;
    }
}
```

1.9 The `yleval` Module

Our job is almost done, there only remains to create the leader: the M4 module which receives the expression from M4, launches the parser on it, and return the result or an error to M4. The most relevant bits of this file, '`yleval.c`', are included below. The reader is also referred to '`yleval.h`' and '`ylscan.l`', see ⟨undefined⟩ [Advanced Use of Flex], page ⟨undefined⟩.

```
/*------------------------------------------------------------------.
| Record an error occurring at location LOC and described by MSG.  |
`------------------------------------------------------------------*/

void
yleval_error (yleval_control_t *control,
              const yyltype *loc, const char *msg, ...)
{
  va_list ap;
  /* Separate different error messages with a new line. */
  fflush (control->err);
  if (control->err_size)
    putc ('\n', control->err);
  LOCATION_PRINT (control->err, *loc);
  fprintf (control->err, ": ");
  va_start (ap, msg);
  vfprintf (control->err, msg, ap);
  va_end (ap);
}
```

This first function, `yleval_error`, is used to report the errors. Since there can be several errors in a single expression, it uses a facility provided by the GNU C Library: pseudo streams which are in fact hooked onto plain `char *` buffers, grown on demand. We will see below how to initialize this stream, `err`, part of our control structure defined as:

```
typedef struct yleval_control_s
{
  /* To store the result. */
  number result;

  /* A string stream.  */
  FILE *err;
  char *err_str;
  size_t err_size;
} yleval_control_t;
```

The following function computes the division, and performs some semantic checking. `ylmod` and `ylpow` are similar.

```
/*-------------------------------------------------------------------.
| Compute NUMERATOR / DENOMINATOR, and store in RESULT.  On errors, |
| produce an error message, and return FALSE.                       |
`-------------------------------------------------------------------*/

boolean
yldiv (yleval_control_t *control, const yyltype *loc,
       number *result, number numerator, number denominator)
{
  if (!denominator)
    {
      yleval_error (control, loc, "Division by zero");
      *result = 0;
      return FALSE;
    }

  *result = numerator / denominator;
  return TRUE;
}
```

Now, the conductor of the whole band, the builtin `yleval` is defined as follows. Its first part is dedicated to initialization, and to decoding of the optional arguments passed to the builtin:

```
/*-----------------------------------.
| yleval(EXPRESSION, [RADIX], [MIN]) |
`-----------------------------------*/

M4BUILTIN_HANDLER (yleval)
{
  int radix = 10;
  int min = 1;

  /* Initialize the parser control structure, in particular
     open the string stream ERR. */
  yleval_control_t yleval_control;
  yleval_control.err =
    open_memstream (&yleval_control.err_str,
                    &yleval_control.err_size);

  /* Decode RADIX, reject invalid values. */
  if (argc >= 3 && !m4_numeric_arg (argc, argv, 2, &radix))
    return;

  if (radix <= 1 || radix > 36)
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: radix out of range: %d"),
                M4ARG(0), radix));
      return;
    }
```

```
  /* Decode MIN, reject invalid values. */
  if (argc >= 4 && !m4_numeric_arg (argc, argv, 3, &min))
    return;

  if (min <= 0)
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: negative width: %d"),
                M4ARG(0), min));
      return;
    }
```

Then it parses the expression, and outputs it result.

```
  /* Feed the scanner with the EXPRESSION. */
  yleval__scan_string (M4ARG (1));
  /* Launch the parser. */
  yleval_parse (&yleval_control);

  /* End the ERR stream.  If it is empty, the parsing is
     successful and we return the value, otherwise, we report
     the error messages. */
  fclose (yleval_control.err);
  if (!yleval_control.err_size)
    {
      numb_obstack (obs, yleval_control.result, radix, min);
    }
  else
    {
      M4ERROR ((warning_status, 0,
                _("Warning: %s: %s: %s"),
                M4ARG (0), yleval_control.err_str, M4ARG (1)));
      free (yleval_control.err_str);
    }
}
```

Example 7.21: '`yleval.y`' – *Builtin* `yleval` *(continued)*

It is high time to play with our module! Here are a few sample runs:

```
$ echo "yleval(1)" | m4 -M . -m yleval
1
```

Good news, we seem to agree on the definition of 1,

```
$ echo "yleval(1 + 2 * 3)" | m4 -M . -m yleval
7
```

and on the precedence of multiplication over addition. Let's exercise big numbers, and the radix:

```
$ echo "yleval(2 ** 2 ** 2 ** 2 - 1)" | m4 -M . -m yleval
65535
$ echo "yleval(2 ** 2 ** 2 ** 2 - 1, 2)" | m4 -M . -m yleval
```

```
1111111111111111
```

How about tickling the parser with a few parse errors:

```
$ echo "yleval(2 *** 2)" | m4 -M . -m yleval
error m4: stdin: 1: Warning: yleval: 1.5: parse error, unexpected "*": 2 *** 2
```

Wow! Now, *that*'s what I call an error message: the fifth character, in other words the third '*', should not be there. Nevertheless, at this point you may wonder how the stars were grouped[8]. Because you never know what the future is made of, I strongly suggest that you *always* equip your scanners and parsers with runtime switches to enable/disable tracing. This is the point of the following additional builtin, `yldebugmode`:

```
/*-----------------------------.
| yldebugmode([REQUEST1], ...) |
`-----------------------------*/

M4BUILTIN_HANDLER (yldebugmode)
{
  /* Without arguments, return the current debug mode. */
  if (argc == 1)
    {
      m4_shipout_string (obs,
                         yleval__flex_debug ? "+scanner" : "-scanner", 0, TRUE);
      obstack_1grow (obs, ',');
      m4_shipout_string (obs,
                         yleval_debug ? "+parser" : "-parser", 0, TRUE);
    }
  else
    {
      int arg;
      for (arg = 1; arg < argc; ++arg)
        if (!strcmp (M4ARG (arg), "+scanner"))
          yleval__flex_debug = 1;
        else if (!strcmp (M4ARG (arg), "-scanner"))
          yleval__flex_debug = 0;
        else if (!strcmp (M4ARG (arg), "+parser"))
          yleval_debug = 1;
        else if (!strcmp (M4ARG (arg), "-parser"))
          yleval_debug = 0;
        else
          M4ERROR ((warning_status, 0,
                    _("%s: invalid debug flags: %s"),
                   M4ARG (0), M4ARG (arg)));
    }
}
```

[8] Actually you should already know that Flex chose the longest match first, therefore it returned '**' and then '*'. See ⟨undefined⟩ [Looking for Tokens], page ⟨undefined⟩.

Applied to our previous example, it clearly demonstrates how '2 *** 2' is parsed[9]:

```
$ echo "yldebugmode(+parser)yleval(2 *** 2)" | m4 -M . -m yleval
error Starting parse
error Entering state 0
error Reducing via rule 1 (line 100),  -> @1
error state stack now 0
error Entering state 2
error Reading a token: Next token is 257 ("number" (1.1) = 2)
error Shifting token 257 ("number"), Entering state 4
error Reducing via rule 3 (line 102), "number"  -> exp
error state stack now 0 2
error Entering state 10
error Reading a token: Next token is 279 ("**" (1.3-4))
error Shifting token 279 ("**"), Entering state 34
error Reading a token: Next token is 275 ("*" (1.5))
error Error: state stack now 0 2 10
error Error: state stack now 0 2
error Error: state stack now 0
error m4: stdin: 1: Warning: yleval: 1.5: parse error, unexpected "*": 2 *** 2
```

which does confirm '***' is split into '**' and then '*'.

1.10 Using Bison with the GNU Build System

Autoconf and Automake provide some generic Yacc support, but no Bison dedicated support. In particular Automake expects 'y.tab.c' and 'y.tab.h' as output files, which makes it go through various hoops to prevent several concurrent invocations of yacc to overwrite each others' output.

As Gperf, Lex, and Flex, Yacc and Bison are maintainer requirements: someone changing the package needs them, but since their result is shipped, a regular user does not need them. Really, don't bother with different Yaccs, Bison alone will satisfy all your needs, and if it doesn't, just improve it!

If you use Autoconf's AC_PROG_YACC, then if bison is found, it sets YACC to 'bison --yacc', otherwise to 'byacc' if it exist, otherwise to 'yacc'. It seems so much simpler to simply set YACC to 'bison'! But then you lose the assistance of Automake, and in particular of missing (see ⟨undefined⟩ [Using Gperf with the GNU Build System], page ⟨undefined⟩). So I suggest simply sticking to AC_PROG_YACC, and let Automake handle the rest.

Then, in your 'Makefile.am', merely list Bison sources as ordinary source files:

```
LDFLAGS = -no-undefined

pkglibexec_LTLIBRARIES = yleval.la
yleval_la_SOURCES = ylparse.y ylscan.l yleval.c yleval.h ylparse.h
yleval_la_LDFLAGS = -module
```

and that's all.

[9] The same attentive reader who was shocked by the concept of mid-rule actions, see Section 1.8 [Advanced Use of Bison], page 22, will notice the reduction of the invisible '@1' symbol below.

1.11 Exercises on Bison

Error Recovery

There is one single important feature of Yacc parsers that we have not revealed here: the use of the `error` token to recover from errors. The general idea is that when the parser finds an error, it tries to find the nearest rule which claims to be ready to return to normal processing after having thrown away embarrassing symbols. For instance:

```
exp: '(' error ')'
```

But including recovery means you must pretend everything went right. In particular, an `exp` is expected to have a value, this rule *must* provide a valid `$$`. In our case, the absence of a rule means proceeding with a random integer, but in other applications it may be a wandering pointer!

Equip `yleval` with proper error recovery. See section "Writing rules for error recovery" in *Bison – The YACC-compatible Parser Generator*, for detailed explanations on `error` and `yyerrok`.

LR(2) Grammars

Consider the following excerpt of the grammar of Bison grammars:

```
grammar: rule
       | grammar rule
rule: symbol ':' ride-hand-side
right-hand-side: /* Nothing */
               | right-hand-side symbol
symbol: 's'
```

Convince yourself that Bison cannot accept this grammar either by (i) feeding it to `bison` and understanding the conflict, (ii) drawing the LR(1) automaton and understanding the conflict, (iii) looking at the strategy *you*, human, use to read the text '`s : s s s : s : s`'.

Yes, the bottom line is that, ironically, Yacc and Bison cannot use themselves! (More rigorously, they cannot handle the natural grammar describing their syntax.)

Once you have understood the problem, (i) imagine you were the designer of Yacc and propose the grammar for a better syntax, (ii) stop dreaming, you are not Steven C. Johnson, so you can't change this grammar: instead, imagine how the *scanner* could help the parser to distinguish a `symbol` at the right hand side of a rule from the one at the left hand side.

1.12 Further Reading On Parsing

Here a list of suggested readings.

Bison – The YACC-compatible Parser Generator

Written by The Free Software Foundation
Available with the sources for GNU Bison

Definitely one of the most beautiful piece of software documentation.

Lex & Yacc
Written by John R. Levine, Tony Mason and Doug Brown
Published by O'Reilly; ISBN: (**FIXME:** *I have the french one :).*)

As many details on all the Lexes and Yaccs as you may wish.

Parsing Techniques – A Practical Guide
Written by Dick Grune and Ceriel J. Jacob
Published by the authors; ISBN: 0-13-651431-6

A remarkable review of all the parsing techniques. Freely available on the web pages of the authors since the book was out of print. (**FIXME:** *url.*).

Modern Compiler Implementation in C, Java, ML
Written by Andrew W. Appel
Published by Cambridge University Press; ISBN: 0-521-58390-X

A nice introduction to the principles of compilers. As a support to theory, the whole book aims at the implementation of a simple language, Tiger, and of some of its extensions (Object Oriented etc.).