Formal Languages and Compilers

Syntactic Analysis: Bison

Alessandro Barenghi & Michele Scandale

January 8, 2020

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 syntactic analysis must:

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

The structure is defined by means of a grammar. The syntactic analysis is performed over a stream of terminal symbols:

- typically, the input terminal symbols are a token stream
- obtained as the output of a lexical analysis (lexing) pass
- nonterminal symbols are only generated through reduction

Purpose of Semantic Analysis

The semantic analysis aims to verify the correctness at semantic level: it verifies if a syntactically correct sentence is meaningful. This verification can be described as a decoration of the AST (Abstract Syntax Tree) performing:

- type checking (check coherency of types in expressions)
- symbol definition before use
- generated code
- value of expressions

`bison`: The GNU Parser Generator

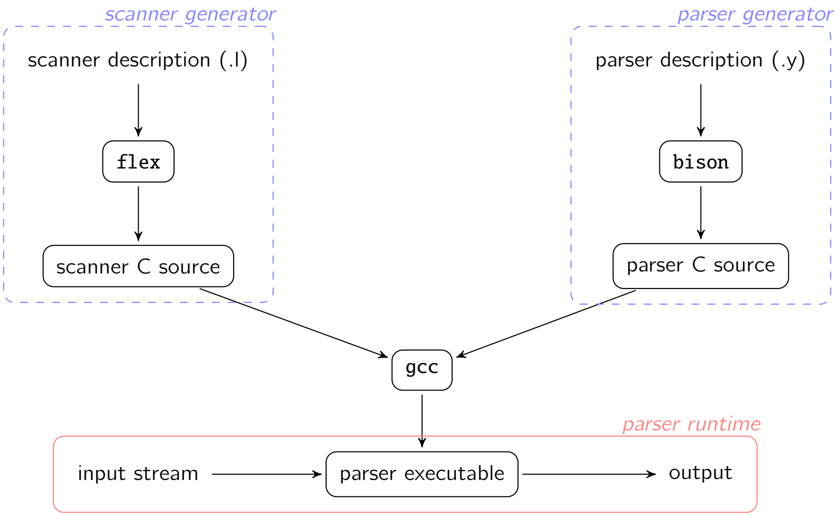
The standard tool to generate LR parsers:

- YACC compatible
- naturally coupled with **`flex`**
- by default LALR(1) parser generator

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

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

Workflow


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

bison

File format

A bison file is structured in four sections separated by %%:

- **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
```

bison

Definitions: tokens

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

Token definitions:

```
%token IF WHILE DO FOR GOTO
%token VOID INT FLOAT STRING
%token ID INT_CONST FLOAT_CONST STRING_CONST
%token GT LT LE GE EQ NE
```

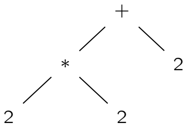
bison

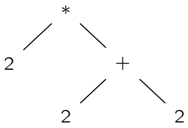
Definitions: operator precedence

Operator precedence and associativity rules can be used to remove some ambiguities from a grammar.

$$
\begin{aligned}
S &\rightarrow E \mid \epsilon \\
E &\rightarrow NUMBER \\
E &\rightarrow E + E \mid E * E
\end{aligned}
$$

The given grammar is ambiguous! Consider the input $2 * 2 + 2$.





Grammar solution: encode operator precedence and associativity within the grammar.

bison

Definitions: operator precedence

Bison solution: specify the operator precedence and associativity keeping the grammar small.

Operator precedence and associativity definitions:

- **precendence** specify only precedence level
- **left** left associativity (e.g., addition, multiplication)
- **right** right associativity (e.g., exponentiation)
- **nonassoc** operator is non associative (x **op** y **op** z is considered a syntax error)

The precedence is implicitly defined in growing order (lower precedence ones come first):

```
%precedence THEN
%precedence ELSE
%left AND OR XOR
%nonassoc GT LT LE GE EQ NE
%left PLUS MINUS
%left MUL DIV
%right EXP
```

Note that `%left`, `%right`, `%nonassoc` serve the purpose of `%token` declaration.

bison

Definitions: operator precedence

Precedence and associativity information are employed during parsing to solve ambiguities:

- each rule containing at least an operator has the precedence of the last terminal symbol

During parsing, shift/reduce conflicts may occur:

shift if the precedence of the look ahead symbol is higher than the one of the rule

reduce if the precedence of the look ahead symbol is lower than the one of the rule

If the precedences are the same, check the associativity:

left forces a reduction

right forces a shift

bison

Definitions: semantic values

Semantic values:

- **%union** declaration specifies the entire collection of possible data types for semantic values (the type defined is named **YYSTYPE**)
- semantic value field can be associated to *terminal* (**%token** declaration) and *non-terminal* symbols alike (**%type** declaration)

```
%union {
  int i_value;
  float f_value;
  const char *str_value;
  struct Expr expr;
  struct TreeNode *node;
}

%token <str_value> ID
%token <i_value> INT
%token <f_value> FLOAT
%type <expr> expr
```

bison

Grammar rules

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

- if not specified, the l.h.s. of the first rule is the axiom

```
program: type_defs_opt var_decls_opt function_decls_opts
       ;

type_defs_opt: type_def_opt type_def
             | /* epsilon */
             ;

type_def : ID ':' types
         ;

types : basic_types
      | aggregate_types
      | ID
      ;
```

bison

Grammar Rules: Context-dependent precedence

Given a simple arithmetic expressions grammar that handling the *unary minus*:

```
%token NUMBER
%left '+' '-'
%left '*' '/'
%%
exp : exp '+' exp | exp '-' exp |
      exp '*' exp | exp '/' expr |
      '-' exp | '(' exp ')' | NUMBER
    ;
```

The grammar is still ambiguous! Consider the following input: $-2+5$.

Problem: The - token is used with two different semantic values (subtraction and negation). How do we handle different associativity and precedence for the same symbol?

bison

Grammar Rules: 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 '+' '-'
%left '*' '/'
%right OP_UMINUS
%%
exp : exp '+' exp | exp '-' exp
    | exp '*' exp | exp '/' exp
    | '-' exp %prec OP_UMINUS
    | '(' exp ')' | NUMBER
    ;
```

bison

Grammar Rules: Semantic Actions

In order to allow semantic analysis implementation within the parsing process, **bison** allows to specify semantic actions in grammar rules:

- a semantic action is a piece of C code wrapped by curly braces {,}
- a semantic action can be specified at the end of each rule alternative
- a semantic action can be specified also in the middle of a production rule (mid-rule actions)

```
function_call : ID
                { /* mid-rule action */ }
                '(' args_list_opt ')'
                { /* end-of-rule action */}
              ;
```

bison

Grammar Rules: Semantic Actions

- the semantic value of each grammar symbol in a production is labelled as `$i`, where *i* is the position of the symbol
- mid-rule actions have their own unique identifier too, and thus they are to be counted
- semantic values can be propagated towards the root of the syntax tree (bottom-up parsing), to this end `$$` identifies the *left-hand-side* of the rule (accessible only from *end-of-rule actions*)

```
$$ -> function_call : ID                          <- $1
                      { /* mid-rule action */ }   <- $2
               $3 ->  '('
                      args_list_opt               <- $4
               $5 ->  ')'
                      { /* end-of-rule action */}
                    ;
```

bison

Mid-rule actions

Mid-rule actions have their behaviour conditioned by the (LALR) parser runtime behavior:

- from mid-rule actions you can access **$n** only with *n* lesser than the mid-rule action index itself (greater values refer to elements that haven't been parsed yet!)
- $$ in a mid-rule action will refer to the semantic value synthetized by the action itself
- they can introduce ambiguities (bison does not inspect C code)

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

The following chunk:

```
stmt : ID { foo($1); } '(' args ')' { check_call_args($1, $4); }
     ;
```

is transformed into the following:

```
stmt : ID $@1 '(' args ')' { check_call_args($1, $4); }
     ;
$@1 : %empty { foo($0); }
```

Integration of `flex` and `bison`

The generated parser:

- assumes that an **`yylex`** function will be provided
- provides a **`yyparse`** function that returns 0 when the parsing is successful
- provides a **`yylval`** global variabile (for non-reentrant parser) that can be used from **`flex`** to setup some lexical value (e.g., constant values)
- using **`bison -d`** an header file will be generated: this contains token definitions, and declaration of extern accessible variable (e.g., yylval, yylloc): this can be included in flex in order to know the token encoding expected by the parser.

More informations and details can be found here:
`http://www.gnu.org/software/bison/manual`

Example

Lets start with a basic calculator for simple expression:

$$
\begin{aligned}
&S \rightarrow E | S; E \\
&E \rightarrow E + E | E - E | - E \\
&E \rightarrow NUMBER
\end{aligned}
$$

Example

The code for the scanner:

```
%{
#include "exp.parser.h"
#include <stdio.h>
%}
DIGIT [0-9]
%option noyywrap
%%
[ \t\r\n]+    {}
{DIGIT}+ { yylval.value = atoi(yytext); return NUMBER; }
"+"      { return '+'; }
"-"      { return '-'; }
";"      { return ';'; }
.        { return LEX_ERR; }
%%
```

Example

To tackle the parser development we:

- implement a pure syntax analyzer
- add the semantic actions to the syntax analyzer

```
%union {
  int value;
}
%token <value> NUMBER
%token LEX_ERR
%type <value> exp
%left '+' '-'
%right OP_UMINUS
%%
program : exp
        | program ';' exp
        ;

exp : exp '+' exp
    | exp '-' exp
    | '-' %prec OP_UMINUS exp
    | NUMBER
    ;
%%
// ...
```

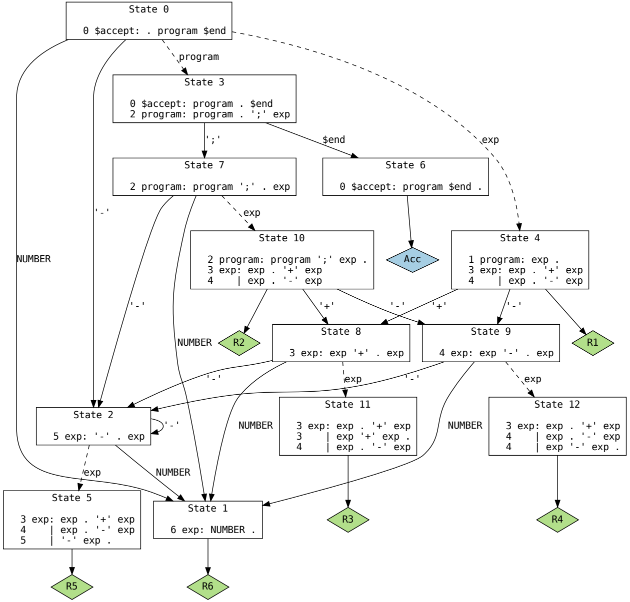
Example

Now we add the semantic analysis in order to compute the result:

```
%union {
  int value;
}
%token <value> NUMBER
%token LEX_ERR
%type <value> exp
%left '+' '-'
%right OP_UMINUS
%%
program : exp { printf("Result: %d\n", $1); }
        | program ';' exp { printf("Result: %d\n", $3); }
        ;

exp : exp '+' exp { $$ = $1 + $3; }
    | exp '-' exp { $$ = $1 - $3; }
    | '-' %prec OP_UMINUS exp { $$ = -$2; }
    | NUMBER { $$ = $1; }
    ;
%%
// ...
```

Here's the pilot automaton (see **-g** bison option):



Class task

Extend the previous example in order to add a full set of arithmetic expressions and introduce the support for scoped variables:
An input example:

```
-2 * [let x = 3, y = ?] { x + 2 + y};
11 - [let z = ?, y = 2] {2 - z - [let k = ?] {k * 5}}
```

With `?` we represent values that user must provide interactively.

Hints:

- you can assume a maximum number of variable declarations (globally or for each scope)
- use a global variable to represent a stack of scopes to track the scope nesting
- write functions to traverse the scope stack for symbol resolution