Manual of Maphoon 2021

Hans de Nivelle

July 22, 2021

Abstract

Maphoon is a tool that automatically generates a parser from the description of a grammar. Maphoon is written in C^{++}, and it constructs a parser in C^{++}. In its functionality, Maphoon is similar to Yacc or Bison. The main improvement is that it allows the programmer to use all advantages of C^{++}. If the attribute classes have proper resource management, the parser will automatically have proper resource management as well. The developer of the parser does not need to know anything about the internals of the attribute classes. Maphoon automatically generates a symbol class. Both the symbol class and the parser that it generates can be put in arbitrary namespace, and they have no static fields. Maphoon can generate parsers that allow run time definition of operators, as used for example by Prolog. Maphoon comes with separate tokenizer generation tools, which are described in separate document.
This document describe the use of the system only. For the theory we refer to [1]. Maphoon is released with the 3-clause BSD license.

1 Quick Start for the Impatient

If you only want to print parse tables, and check a grammar for consistency, then it is sufficient to prepare a single file with extension `.m` in the following format:

```
// Language with tricky lookaheads:
%startsymbol S EOF    // Start symbol with end marker.
%rules

S =>  c b c a | A a A b ;
A =>  B ;
B =>  c ;
```

Call `./maphoon impatient.m`. Maphoon will analyse the file, print the parse tables, and complain a bit about undeclared symbols.

2 Design Goals

In the design of Maphoon, we tried to meet the following requirements:

1. The resulting parser must use good quality C^{++}, and the user must be able to write the semantic actions in good quality C^{++}. Main advantage of C^{++} is that user defined types can hide their resource management, by using copy constructors, assignment and destructors. The parser must support this for the semantic attributes.

2. The parser must correctly implement the algorithms of the dragon book ([1]).

3. It must be possible to extend the syntax at run time. Concretely, we want to support the possiblity of defining Prolog-style operators (see [4]) In Prolog, it is possible to declare `op( '+', 'yfx', 200 ),` after which `+` will be a left associative infix operator. If one wants to allow such dynamic syntax extensions, parse conflicts cannot be resolved earlier than at run time.

   Maphoon solves the problem by attaching preconditions to reduction rules. Preconditions return a `short int` with the following meaning:

   1 Agree to be reduced.

   0 Refuse to be reduced.

   -1 Refuse to be reduced, and also block the shift is there is one.

   The general form of a conflict is (Shift)^m (Reduce)^n, where $m \in \{0, 1\}$, $n \geq 0$, and $m + n > 1$. The conflict is between possibly one shift, and an unbounded number of reductions. At parser generation time, Maphoon stores all possibilities in the parse table in the order specified by the user. At parse time, Maphoon will attempt the reductions and perform the first reduction whose precondition returns an integer > 0. If all preconditions return 0, it will perform the shift if there is one. Otherwise, one of the preconditions returned -1, and the result is a syntax error. The possibility of blocking the shift was created in order to be able to enforce nonassociativity. A non-associative operator is an operator that can neither shift nor reduce.

3 Running Maphoon

Maphoon can be called with one, two or three arguments. The arguments are as follows:

1. The first argument must have extension `.m` or `.M`. It must be the name of a file containing a description of a grammar, for which Maphoon will try to create a parser.

2. If present, the second argument must be a directory, possibly the current directory (`.`), but never the source directory. In this directory, Maphoon will try to write files **symbol.h**, **symbol.cpp**, and possibly also **parser.h** and **parser.cpp**. The first two files will contain a definition of `struct symbol`. The second two files will contain the parser. The output directory should not be the directory of the sources of Maphoon, because they also contain files **symbol.h** and **symbol.cpp** and they will be overwritten.

3. If present, the third argument must be the position of the file `idee.x`. This file contains the starter code for the generated parser. Without it, Maphoon will not be able to generate files **parser.h** and **parser.cpp**.

The recommend use sequence is as follows:

1. Prepare a file `grammar.m`, and make sure that Maphoon accepts it without complaints. Don't worry about semantic actions yet. Have a look at the parse tables, and check if they look reasonable.

2. Call Maphoon with a second argument, and generate the symbol class. Write the tokenizer, using `symbol` that was produced by Maphoon.

3. When the tokenizer is complete, you can call Maphoon with three arguments, start adding semantic actions, compiler the parser and test it.

4 Declaring Symbols

Maphoon automatically generates C^{++} code that defines a symbol class. The files are called **symbol.cpp** and **symbol.h**. They are written in a directory that is determined by the second command line argument. It is possible to specify a namespace for the symbol class with the `%symbolspace` command. **Warning**: Make sure that the output directory is not the directory of the Maphoon sources. They also conctain files with names **symbol.h** and **symbol.cpp**, and they will be overwritten. In order to declare a symbol, write

```
%symbol sym1 sym2 sym3
   // Declare symbols with trivial (void) attribute type.
%symbol{ C++-type } sym1 sym2 sym3
   // Declare symbols with non-trivial attribute type.
```

All symbols that occur in grammar rules must be declared, otherwise no parser and no symbol class will be generated. It is possible to declare symbols with type `void`, which has the same effect as declaring them without type. The C^{++}-types should be value types, which means that they should not be references. They should also not be **const**. Maphoon will warn when attribute types are a reference or **const**. For efficiency, it is recommended that the types have efficient moving constructors and assignment operators.

It is possible to declare symbols that do not occur in the grammar. This may sometimes be necessary. For example, one could declare a whitespace token or a comment token for internal use in the tokenizer, which will not be returned to the parser. Maphoon will list the declared, but unused symbols in the output.

If a symbol has a non-trivial attribute type, every rule that has it as a left hand side, must have reduction code. The following example will result in an error.

```
%symbol { int } A

%rules
   A => b ;  // Error, don't know which integer to use as attribute.
```

5 Source Information

It is often useful to attach source information to symbols. This information can be used for generating error messages. In order to include a field with source information in the symbol class generated by Maphoon, use `%infotype`, followed by the type between braces `{ }`:

```
%infotype { unsigned int }   // Just a line number.
%infotype { std::pair<unsigned,unsigned> }
   // An interval of two file positions.
%infotype { position }   // A defined class.
```

If an infotype definition is present, the generated symbol class has a field `std::optional<I> info`, where `I` is the type declared by `%infotype`. In addition, `std::optional<I>` will become a parameter of every constructor of class `symbol`. It will always be the second argument after the `symboltype`.

The `info` field must be declared as `std::optional`, because symbols that originate from an empty reduction `Sym => ;` have no source information.

At a reduction, the parser will try to compute the info field of the left hand side from the info fields of the right hand sides, by merging the ones that are defined. In order to merge infotypes, it will call

```
static void parser::merge_infos( infotype& , infotype&& ),
```

which the user must define. The simplest definition is

```
void parser::merge_infos( infotype& inf1, infotype&& inf2 ) { }
```

which means that the left hand side receives the infotype of the first right hand side symbol that has an infotype, and undefined if there is none.

6 Parameters

It is possible to declare parameters that will be included in the parser as reference fields, and which can be used in reduction code. They can be used for example

for storing type declarations, assignments to variables, for remembering the input file, for logging errors, etc. If the tokenizer is defined as a class object, it has to be passed as parameter to the parser. If there is no separate tokenizer class, the parser probably reads its input from a file, and it will be necessary to pass this file as parameter. A parameter declaration has the following form:

```
%parameter { } name
```

Examples are:

```
%parameter{ std::map< std::string, double > } varstore
%parameter{ std::istream } inputsource
%parameter{ std::vector< std::string > } errorlog
```

Don't add a reference symbol `&` to the type, because this is done by Maphoon. It will complain if you try. Pointers are no problem. The parameter fields are initialized when the parser is constructed. They are parameters of the constructor `parser::parser( )`, in the order in which they appear in the input file.

7 Specifying the Tokenizer

Symbols need to come from somewhere, and the place where they come from is usually called *tokenizer*. The input source is declared by

```
%source { expr }
```

Maphoon inserts `lookahead = expr` whenever it needs a symbol. `expr` must include a semicolon at the end. The expression must be such that it can be called from the namespace of the parser. If no source is given, Maphoon will not generate a parser.

Additional information needed by `expr` must be passed as parameters to the parser (See Section 6). Typical parameters are the input file, or the tokenizer object. If the tokenizer is defined in a class `tokenizer`, one can define

```
%parameter { tokenizer } tok   // & will be added automatically.
%source{ tok. read( ); }
```

Note that `tok` is not a global name, but a reference field of the parser that has to be initialized when the parser is constructed.

8 Specifying the Grammar and Action Code

The main part of the description of a bottom up parser consists of the grammar rules, and what must be done when a rule is applied. Grammar rules have form `A => B1 ... Bn` and have code attached to them. This code is traditionally called *action code* ([3]), although calling it *reduction code* would be better.

We call `A` the *left hand side (lhs)*, and `B1 ... Bn` the *right hand side (rhs)* of the rule. If the parser finds `B1 ... Bn`, it will replace by `A`, and execute the associated code. If `A` has an attribute, the code must compute the attribute. If some of the `Bi` have attributes, they can be used in the computation.

The following is a simple example. It describes three grammar rules, which share a common lhs `E`.

```
E  => E:e PLUS F:f   { return e + f; }
    | E:e MINUS F:f  { return e - f; }
    | F:f            { return f; }
    ;
```

If one wants to use the attribute of an rhs symbol, one has to attach a variable to it by using a colon (`:`). This is done for `E:e` and `F:f` in the example above. Attached variables have the type with which their attached symbol are declared. The return statements set the attribute value of the `lhs` symbol. If the lhs symbol has no attribute, the action code can fall over the end.

In the given example all actions consist of a single return statement, but it may use any C^{++} statement, and it may extend over multiple lines. We recommend that you keep action code moderate in size.

In addition to computing the lhs value, action code can have side effects. In particular, action code can change and use the values of parameters (See 6). For example, an assignment statement can store assigned value:

```
Command => IDENTIFIER:id BECOMES E:e SEMICOLON
{
   if( errorlog. empty( ))
   {
      std::cout << "assigning: " << id << " := " << e << "\n";
      memory. assign( id, e );
   }
   else
   {
      printerrors( errorlog, std::cout );
      errorlog. clear( );
   }
}
```

The action code relies on the following declarations:

```
   %parameter{ varstore<double> }    memory
   %symbol{ std::string }            IDENTIFIER
   %symbol{ double }                 E
```

If there were errors during computation of `E`, they are printed. Otherwise, the value of `E` is stored in the value `IDENTIFIER`.

In order to decide when parsing is complete, one lhs symbol has to be assigned the role of *start symbol* (although for bottom up parsing, calling it *end*

symbol would be better.) The parse is complete when the input is rewritten into the start symbol.

Maphoon constructs a parser that can be called with different start symbols. In this way, different languages can share grammar rules and symbol sets. Possible start symbols have to be declared. Together which each start symbol, one has to specify the symbols that terminate correct input derived from the start symbol. We call these symbols the *terminators* of the start symbol. Natural choices are the end-of-file symbol, or a semicolon.
The following declares a start symbol S with terminators T1, . . . , Tn:

```
%startsymbol S T1 T2 ... Tn
```

The following defines a start symbol S with terminator EOF, and a start symbol EXP with terminator DOT.

```
%startsymbol S EOF
%startsymbol EXP DOT
```

If a start symbol is declared more than once, the terminator sets are merged.

When the parser is called, it has to be called with the start symbol that one wants to recognize. It is guaranteed that the parser will never read beyond a terminator of the given start symbol. When a symbol T is declared as a terminator symbol of a start symbol S, the symbol T must be not reachable from S. Otherwise, the parser would not know when to stop. Maphoon checks that no terminator is reachable from its start symbol.

In many cases, parsing has to be stopped before the input is reduced into a start symbol. This happens for example when there exists a decided Quit command. This will be discussed in Section 10.

In addition to action code, rules can have conditions attached to them. The conditions are evaluated before the rule is reduced, and if the condition reduces to zero or a negative number, the reduction is not carried out. This makes it possible to parse ambiguous grammars. This will be discussed in detail in the next section.

9 Runtime Conflict Handling

Conflicts appear when the grammar is ambiguous, or when parsing requires looking ahead further than one symbol. One possible source of ambiguity are errors in the grammar. In that case, the grammar can be corrected, and the conflict will go away. A second source of ambiguity are underspecified priorities between operators. In most cases, the grammar can be made unambiguous by introducing additional non-terminal symbols, we give an example below.

In order to handle difficult conflicts, Maphoon offers the possibility of postponing conflict resolution to run time. We first explain how ambiguous grammars can be made nonambigous in the presence of fixed operators. After that, we explain the runtime conflict handling mechanism of Maphoon.

An example of a naturally occurring ambiguous grammar is the following:

```
Formula => CONST
        |  NOT Formula
        |  Formula AND Formula
        |  Formula OR Formula
        |  Formula IMP Formula
        ;
```

This grammar is ambiguous, because the inputs `CONST AND CONST AND CONST` and `CONST AND CONST OR CONST` can be parsed in different ways, dependent on whether the left or the right operator has the priority. It can be made non-ambiguous by introducing additional non-terminal symbols:

```
Formula  => Formula2 IMP Formula | Formula2 ;  // left associative.
Formula2 => Formula2 OR Formula3 | Formula3 ;
Formula3 => Formula3 AND Formula4 | Formula4 ;
Formula4 => NOT Formula4 | CONST;
```

Unfortunately, this solution is not always acceptable. Some languages have many priority levels (up to 15), and forcing the user to introduce 15 symbols would refute the goal of making it easy to modify the language. Moreover, some programming languages (Prolog [4] is probably the most important one) allow runtime definition of operators. In order to deal with runtime definition of operators, Maphoon supports runtime conflict resolution by attaching preconditions to grammar rules. In addition to runtime conflict resolution, preconditions make it possible to leave the recognition of reserved word to the parser instead of the tokenizer. This has the advantage that identifiers are treated as reserved words only in situations where they can occur.

Semantic actions have access to the lookahead, which has type `const std::list< token > &`. Note that there is no guarantee that `lookahead. size( ) != 0`, so that this has to be checked before the lookahead is accessed. The lookahead can be used for deciding whether the reduction should be accepted.

In order to attach a precondition to a rule, one can use `%requires` followed by the code of the precondition. The code has the same form as action code, but all attribute variables and the parameters are **const**. The code of the precondition must return `short int`. A return value greater than zero means that the reduction will take place. Returning zero means that the reduction will not take place, but shifting is still allowed. Returning a value less than zero means that the reduction will not take place, and in addition, no shift will be allowed. If there are more reduction candidates, they will be considered.

The rules below reduce an identifier into the `Quit` symbol when it equals `"Quit"` and the `Show` symbol when it equals `"Show"`.

```
Quit => IDENTIFIER : id
%requires { return id == "quit"; }
;

Show => IDENTIFIER : id
```

```
%requires { return id == "show"; }
;
```

The rules have no action code, because `Quit` and `Show` have no attribute.

In order for this approach to work, the programmer must have control over the order in which reductions are attempted. The `%redsequence` statement provides this control. If there is a state in which more than one reduction is possible, the lhs symbols of the reducible rules can be listed together in a `%redsequence` statement, in the order in which they must be attempted. If they don't occur together in a `%redsequence` statement, Maphoon will print a warning, and apply the rules in an unpredictable order. The parser will reduce the first rule whose precondition returns a value greater than zero (or has no precondition). If all rules return zero and the state allows a shift, the parser will perform the shift. In the remaining case (when there is no shift, or one of the preconditions returned a value less than zero), the parser creates an error.

Precondition code can use the lookahead to decide if the reduction should be made. This is important when deciding priorities of operators. The following code decides if the rule `Formula => Formula AND Formula` should be reduced:

```
Formula => Formula:f1 AND Formula:f2
%requires
   { return decide( sym_AND, lookahead. value( ). type ); }
%reduces
   { return form( op_and, f1, f2 ); }
;
```

Function `decide` can return 1 if `sym_AND` if has higher priority than `lookahead. value( ). type`, and 0 if the lookahead has higher priority. If the lookahead has the same priority and `sym_AND` is non-associative, it should return -1. In that case, the parser will generate an error. It is guaranteed that `lookahead. value( )` exists when precondition code is called.

The only purpose of `%reduces` is to make the input look better. It can be omitted, but it looks bad:

```
Formula => Formula:f1 AND Formula:f2
%requires
   { return decide( sym_AND, lookahead. value( ). type ); }
   { return form( op_and, f1, f2 ); }
;
```

The purpose of allowing preconditions to return negative values is to be able to deal with non-associative operators. If one would use only yes/no, it would be impossible to define non-associative operators.

10 Stopping the Parser

The parser can be stopped by assigning `timetosaygoodbye = true` in action code. The parser will stop immediately after the reduction, and return the

resulting lhs.

11 Syntax Errors and How to Recover from Them

When it encounters an error, the parser first calls method `unsigned int parser::syntaxerror( )`. The returned `unsigned int` determines how many symbols will be ignored in a recovery attempt. Returning 0 means that the parser will not attempt to recover.

Function `syntaxerror( )` should first decide if the error is new, or the result of a failed recovery attempt. This can be done by looking at `lasterror`. The Yacc manual ([2]) suggests that 3 is a good cut-off value. The default implementatation is follows:

```
unsigned int parser::syntaxerror( )
{
   if( lasterror > 3 )
   {
      std::cout << "this looks like a new syntax error\n";
      return 6;
   }
   else
   {
      std::cout << "this looks like a failed recovery\n";
      return 6 - lasterror;
         // Possible because lasterror <= 3.
   }
}
```

In order to define an own error function, include `%usererror` in the input file, so that Maphoon knows that the default function should not be added.

Recovery works the same as in Yacc. The parser inspects the stack for `_recover_` transitions and it collects the symbols that occur after these transitions. After that, it throws away input symbols until it encounters one of the collected symbols. It pushes `_recover` and the symbol after it, and assumes that it is recovered.
An example of the use of `_recover` in a rule is as follows:

```
% E   => _recover_ SEMICOLON
```

This means that if somewhere, while attempting to parse an E, something goes wrong, the parser will resynchronize when it sees a `SEMICOLON`. If token E has an attribute, the recovery rule needs to invent a reasonable value for the attribute. If after seeing the `SEMICOLON` one decides that recovery is not a good idea after all, one can assign `timetosaygoodbye = true;`.

12 Movability of Symbols

The parser will be more efficient when the symbol class is nothrow-movable. In order to obtain that, every attribute must be movable, including the info-type if it is present. If one sets the `debug` field ≥ 2, the parser will check `std::is_nothrow_move_constructible<symbol> :: value` and warn if it is false.

13 Remaining Options

- If `%selfcheck` is selected, the parse tables will be rechecked for completeness. Maphoon will also print information about the quality of the hash tables used. This option is important for the developer only.
- Option `%printclosed` prints the itemsets closed instead of simplified. This may be useful for teaching or finding the cause of conflicts. It doesn't affect the parser constructed.
- It is possible to specify C^{++} code that will be copied into the symbol or parser definition. No substitutions will be applied.

  ```
  %symbolcode_h{       }      // Goes into symbol.h
  %symbolcode_cpp{     }      // Goes into symbol.cpp
  %parsercode_h{       }      // Goes into parser.h
  %parsercode_cpp{     }      // Goes into parser.cpp
  ```

  As usual, **.h** files should be used for declarations and `#include`s, while the corresponding **.cpp** files should be used for the corresponding definitions. One should not define anything in a **.cpp** file without declaring it in the corresponding **.h** file, unless one defines it in anonymous namespace. One must be careful with namespaces. If `%symbolspace` or `%parserspace` is set, the code that is copied into the **.cpp** file will be in the corresponding namespace, while any code that is copied into the **.h** file will not be in any namespace. It must be like this, because `#include`s cannot be in a namespace. As a general rule, one should not define many things in **parser.h** and **symbol.h**, but rather in a separate file.
- If `%nodefaults` is set, the parser will not reduce without lookahead in states where reduction is the only possibility. Otherwise, it will always take a lookahead from the tokenizer. Suppose that one has rule `S => E SEMICOLON`. When the semicolon has been pushed, the parser is in a state where the only possibility is reducing the rule. If `%usedefaults` is set, the parser will do this. Otherwise, it will still take a lookahead, and create an error when the lookahead is not in the follow set. The advantage is that in case of error, the chances for recovery are better. The disadvantages are a bigger parse table, and the fact that in interactive applications, the tokenizer will

ask for an additional symbol before reducing the rule. That is annoying if the associated reduction code prints the outcome.

- `%symbolspace s1 :: ... :: sn` determines the namespace of the symbol class.
- `%parserspace s1 :: ... :: sn` determines the namespace of the parser class

14 C^{++} Errors

User code is preceeded by a `#line` directive, so that errors will be reported with their place in the `.m` file. This is convenient if the error indeed originates from user code in the `.m` file. Otherwise, it is annoying. If you suspect that this is the case, just delete the `#line` directives.

If the compiler complains about duplicate constructors in the `symbol` class, the most likely reason is that symbols were declared with equivalent types, whose equivalence was not detected by Maphoon. This may be caused by a `using` directive, or by use of `const` in the attribute type of a symbol declaration.

15 Interface to the Parser

If everything goes well, Maphoon creates a parser in files **parser.h** and **parser.cpp**. The parser is defined in a class `parser` in the namespace specified by `%parserspace`. The constructor has form:

```
parser( P1& p1, P2& p2, ..., Pn& pn ) :
   p1(p1), p2(p2), ..., pn(pn)
{ }
```

where `p1, ..., pn` are the parameters declared by `parameter` . The parameters are reference fields of the class:

```
P1& p1;
P2& p2;
...
Pn& pn;
```

Action code and preconditions become methods of the parser class, so that the parameters are accessible. In addition the the parameters, the parser has a field `std::optional<symbol> lookahead;`.

The parser is started by calling method `symbol parse( symboltype start )`. The `symboltype` is the start symbol that one wants to recognize. Only symbols that were declared as `%startsymbol` in the input can be used as start symbol. Otherwise, the parser throws an `std::out_of_range`.

The returned symbol depends on the way the parser terminated. The possibilities are as follows:

1. The parse was succesful, and it reduced the start symbol. It returns a start symbol. If a `lookahead` is present, it will be a terminator of the original start symbol.
2. The parse was succesful, and ended by assigning `timetosaygoodbye=true;`. In that case, the returned symbol is the lhs of the action code in which the assignment took place.
3. The parser could not recover from a syntax error and reached a terminator symbol while trying to recover. In that case the parser returns `sym__recover_`.
4. The parser could not recover from a syntax error and gave up. It returns `sym__recover_`.

Note that, if a syntax error occurred from which the parser recovered, it will return in state (1) or (2). It is the responsibility of the user keep track of encountered errors.

If the parser behaves in an unexpected way, it can be debugged by assigning $1, 2$ or 3 to the `short int debug` field. A higher number results in more output.

16 Missing Features (that are worth considering)

Here are some features that we don't have, but which are worth considering:

- Some parser generators allow the use of of regular expression in grammar rules. For example

  ```
  Formula => Formula::f1 ( (PLUS | MINUS ) Formula :f2 ) * ;

  Statement => IF Expr THEN Statement ( ELSE Statement ) ? ;
  ```

  In fact, it is totally easy to allow DFAs on right hand sides in rules, because items can be viewed as the state of a linear DFA. The problem is that we do not know how to compute attributes in case of DFAs.
- Automatic construction of AST.
- Currently there is no way to define private fields of the parser, only parameters, which have to be initialized in the main program.

17 Acknowledgements

Thanks to Danel Batyrbek, Aleksandra Kireeva, Tatyana Korotkova, Dina Muktubayeva, Cláudia Nalon and Olzhas Zhangeldinov.
We thank Nazarbayev University for supporting this research through the Faculty Development Competitive Research Grant Program (FDCRGP), grant number 021220FD1651.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers (Principles, Techniques and Tools)*. Pearson, Addison Wesley, 2007.

[2] Akim Demaille and Paul Eggert. Yacc system. `https://www.gnu.org/software/bison/`, 2014.

[3] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.

[4] Leon Sterling and Ehud Shapiro. *The Art of Prolog (Advanced Programming Techniques)*. The MIT Press, 1994.