Manual of Maphoon 2021

Hans de Nivelle

April 17, 2021

Abstract

Maphoon is a tool that automatically generates a parser from a description of an LALR language. The resulting parser is in C^{++}. In its functionality, Maphoon is similar to Yacc or Bison. The main difference is that it allows the user to make better use of the advantages of C^{++}. It allows to define a clean `token`-class, which has object semantics. It can create a parser with multiple entry points, and it can assist the user when checking wellformedness of attributes.

1 Quick Start for the Impatient

If you want only 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 ;
```

Give the file name as argument to Maphoon. Maphoon will analyze the input, print the parse tables, and that's it.

2 Design Goals

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

1. The resulting parser should use good style C^{++}, and the user should be able to write the semantic actions in good style C^{++}. One of the nice features of C^{++} is that classes can completely hide their memory management (using copy constructors, copying assignment and destructors) The parser should support this for the semantic attributes.

2. Movement of heavy attributes should be avoided. Suppose one has a grammar rule of form `List -> Exp | List` with attached semantic action `List := cons( Exp1, List3 ).` When this rule is repeatedly applied, the list can become arbitrarily big. Each time the rule is reduced, the expression is inserted into List3, after which List3 is moved to the position of Exp1 on the parse stack. Since the list grows linearly, this has the potential of making a linear procedure quadratic. The problem is solved by using an `std::list` for the parse stack. Lists support insertion and deletion in the middle without moving other objects in the list. Objects can be moved from one list to another list by pointer manipulations without need to move the real object.

3. Run time extension of the syntax must be possible. Concretely, we want to support the possiblity of defining Prolog-style operators. In Prolog, it is possible to define `op( '+', 'yfx', 200 )`, after which + is a left associative infix operator that can be used in expressions of form $E+E$. If one wants to allow such dynamic syntax extensions, parse conflicts cannot be resolved earlier than at run time.

   Maphoon solves the problem as follows: The general form of a conflict is $(\text{Shift})^m$ $(\text{Reduce})^n$, 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. At parse time, Maphoon will first attempt the reductions, in the order specified by the order in the grammar. The associated semantic actions can either terminate succesfully, or they can throw an exception indicating that the action refused to reduce. Maphoon commits to the first reduction that does not refuse. If all reductions refuse, and there is a shift, Maphoon will perform the shift. Otherwise, it generates an error.

   When deciding whether or not to reduce, a semantic action can see the lookahead. Consider the following example where the parser sees an identifier (for example '+') and it has to decide whether it wants make an expression out of it, or an operator.

```
% Operator : Identifier
   if( identifier occurs in table of defined operators )
      accept the reduction, and return index in table
   else
      refuse;
% ;

% Expression : Identifier
   if( identifier has a value )
      accept the reduction and return the value
   else
      refuse;
% ;
```

When the parser encounters an identifier in appropriate context, it is first pushed on the parse stack as usual. After that the parser sees that two reductions are possible. We assume that the first rule occurs first in the grammar, so that the parser first tries to reduce the Identifier into an Operator. If the symbol does not occur in the operator table, the reduction will refuse and the parser tries to reduce the Identifier into an Expression. If that also fails, and there are no other possibilities, the parser generates an error message.

In the next example, the parser encounters input of the form Exp1 Op1 Exp2 Op2 Exp3 and it has to decide which of the two operators gets the priority.

```
% Exp : Exp Op Exp
   if( lookahead is non-empty and
       lookahead is an operator, which
       has higher priority than Op )
      refuse;
   else
      return Op( Exp1, Exp3 ).
% ;
```

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 create parse tables.
2. If present, the second argument must be a directory, possibly the current directory (`.`). In this directory, Maphoon will try to write files **symbol.h** and **symbol.cpp**. The files contain a definition of `struct symbol`. The output directory should not be the directory of the sources of Maphoon, because they also contain files **symbol.h** and **symbol.cpp**.
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**.

4 Recommended Use Sequence

The recommend use sequence is as follows:

1. Prepare a file `grammar.m`, and make sure that Maphoon accepts it without complaints. 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 run the parser and debug it. When the parser runs correctly, you can start adding semantic actions.

parser.h/parser.cpp: The declaration and definition of the parser. The output files `parser.h` and `parser.cpp` cannot be changed, but it is possible to tell maphoon to put the definition of the parser in any namespace of choice.

Once the parser and token are generated, you can try to compile the result. In order to get a working parser, a couple of more things are required.

- File `parser.h` includes a file `tokenizer.h`, in the assumption that this file contains the declaration of a tokenizer. The exact requirements of the tokenizer are given in Section 7. Since the tokenizer must be declared in `tokenizer.h`, it is natural (but not obligatory) to implement the token tokenizer in file `tokenizer.cpp`.

  The tokenizer must have a field `std::list< token > lookahead` and a method `scan( )` that produces a `token` and appends it to the lookahead. In addition, the tokenizer must have a function for processing error messages.

  It is possible to tell maphoon to assume that the tokenizer is defined in any namespace of choice.

- A main program. It is possible, but not recommended, to define the main program in the input file. It is better to put it in (yet) another file.

5 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 copy constructors and assignment operators.

It is possible to declare symbols that do not occur in the grammar. This may sometimes be useful. 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.
```

The `info` field is used for storing source information.

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 no separate tokenizer exists, 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 { C-type } name
```

Examples are:

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

One should not add the reference symbol `&` to the C-type, because this is done by Maphoon. Pointers are no problem.

7 Specifying the Tokenizer

Symbols need to come from somewhere, and the place where they come from is usually called *tokenizer*, but it can be any function. The source is declared by

```
%source { expr }
```

Maphoon inserts `lookahead = expr` whenever it needs a symbol. `expr` must end with a semicolon. If no source is specified, Maphoon will not generate a parser. If the tokenizer is stored in a variable, it has to be made available to the parser. This is done by declaring it as parameter. If the tokenizer is called `tok`, add

```
   %parameter { tokenizer } tok
   %source{ tok. read( ); }
```

8 Specifying the Grammar and Reduction Code

In order to specify a grammar, one needs to specify the possible start symbols with their terminators and a sequence of rewrite rules with associated reduction code.

Maphoon can construct a parser for more than one grammar at the same time. The different grammars share their symbol sets and rewrite rules, but have different start symbols. Together which each start symbol, one has to specify the possible symbols that terminate correct input derived from the start symbol. We will call these tokens the terminators of the start symbol. Natural choices the end-of-file symbol, or a semicolon.
The following defines 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 defined more than once, the terminator sets are simply merged. The following group of startsymbol declarations is equivalent to the single declaration above:

```
   %startsymbol S T1 T2
   %startsymbol S T3
   ...
   %startsymbol S Tn
```

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.
Grammar rules have form

```
% Leftsymbol : A1 A2 ... Am    / possible actions
   more actions
%            | B1 B2 ... Bn    / actions
   actions actions actions
%            | C1 C2 ... Ck    / actions
   even more actions
% ;
```

The actions are optional. Ideally, actions are of functional nature, which means that they specify how to compute the attribute value of the left hand side from the attribute values of the right hand side, and nothing else. In practice, actions often have side effects. (For example, store some value, or print some value.) In order to facilitate side effects, Maphoon allows the creation of global variables, that are passed as reference to every semantic action. An example of a rule with a (purely functional) semantic action is:

```
   E : E PLUS E / E1 -> value. front( ) += E3 -> value. front( );
                  return E1;
```

This action tells that when the rule `E : E PLUS E` is reduced, first the value of the second E is added to the value of the first E. After that, the first E is returned. In the code of the action, the following variables are available:

- If the rule has form `A : B C D E`, then the tokens corresponding to `B,C,D,E` are available as variables `B1,C2,D3,E4`. The variables have type `std::list< token > :: iterator`

  As an example, consider a rule of form:

  ```
     E : LPAR E RPAR
  ```

  The available parameters are `LPAR1, E2, LPAR3`. The expression `*E2` refers to the token of `E2`. If `E2` has a field with name `value`, it can be accessed with `E2 -> value`. If `E2 -> value` is non-empty, its first element can be accessed with `E2 -> value. front( )`. It is the responsibility of the user to ensure that accessed list elements exist.

- In addition to the parameters originating from the right hand side of the rule, a semantic action can also make use of global variables. If the user declares a global variable, then it is passed as a C^{++}-reference to every semantic action, so that it can be used for example for storing declared variables, defined operators, etc. We explain in Section 6 how global variables are declared.

- 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 addition to the parameters and global variables, there are a few identifiers that the user should avoid. These are `stack, position`.

9 Conflict Handling

Conflicts appear when the grammar is ambiguous, or when parsing requires looking ahead further than one symbol. One frequent 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 non-ambiguous by introducing additional non-terminal symbols, but sometimes this may be tedious, or even impossible. In order to handle difficult conflicts, Maphoon offers the possibility of postponing conflict resolution to run time.

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

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

The strings `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 receives priority. This grammar is ambiguous. 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 do this would refute the goal of making it easy to modify the language. Moreover, some programming languages (most notably Prolog) allow runtime definition of operators. In order to facilitate this, Maphoon supports runtime resolution of conflicts.

Rules can be equipped with conditions, that The `return`-statement must return the complete token of the lhs. It can either return one of the tokens from the right hand side of the rule, or construct a completely new token. In the latter case, the argument of `return` must be a local variable. It is not possible to construct the token in the `return`-statement. It must be constructed before in a local variable, which is then returned. We list the possibilities:

1. A semantic action can refuse to reduce. This is done by executing the statement `refuse;` When deciding to refuse or not, the action can make use of the lookahead. See for example:

   ```
   %  E : E PLUS E

      if( !lookahead. size( ) ||
          shouldreduce( *PLUS2, lookahead. front( )))
      {
         E1 -> val. front( ) += E3 -> val. front( );
         return E1;
      }
      else
         refuse;
   ```

   It is assumed that the user implemented a function `shouldreduce( )` that compares the priority of the token in the lookahead to the priority of PLUS. If the lookahead is empty, does not contain an operator, or an operator with lower priority, then the rule reduces and returns a token of type E. Otherwise it refuses.

2. It can return a token of the type of the left hand side of the rule. In the example above, when the semantic action accepts the reduction, it returns a token of type E. Returning a token can be done in different ways. The easiest way is by reaching the end of the semantic action. This is possible when the token on the left hand side has no attributes. The parser will automatically generate a token of the right type.

   Alternatively, the user can explicitly return a token through a statement of form `return p;` p must be an identifier. It can be a variable from the right hand side if this variable has the right token type. (This was done in the example above.) If the token does not have the right type, it is possible to change its type before returning it, as in the following example:

   ```
   %  E : DOUBLE

      DOUBLE1 -> type = tkn_E;
      return DOUBLE1;
                // We change the type of the token from DOUBLE
                // to E, but do not modify the attribute.
   ```

   It is the responsibility of the user to ensure that the token that is returned satisfies the attribute constraints. Inside the actions, the user can do whatever he likes with the attributes, but when a token is returned, the parser checks the attribute contraints by calling `ASSERT( iswellformed( ));`

Instead of returning an existing token (after possibly changing its type), it is also possible to construct a completely new token, as in the following example:

```
% E : IDENTIFIER
   token t = tkn_E;   // implicit call of token constructor.
   t. val. push_back( memory [ IDENTIFIER1 -> id. front( ) ]);
      // Look up value of identifier in memory.
   return t;
```

If attributes are big, one should avoid moving tokens, and as much as possible try to return an existing token. It was one of the design goals of the attribute mechanism to allow this as much as possible.

When a semantic action returns a token, Maphoon checks that its type corresponds to the left hand side of the rule. After that, it checks that the attributes satisfy the constraint declarations by calling method `iswellformed( ) const`. If all tests are passed, the parser cleans up the elements from the parse stack that are no longer needed. If the returned element is a parameter from the right hand side, it will not be moved. If the returned element originates from a local variable, all elements from the rhs are cleaned up, and the new element is copied onto the parse stack.

In the rest of this section, we explain how actions are copied into the code of the parser. Every action is translated into a function of form

```
void reduction_X(
   stack, position,        // should not be touched by user.
   G1& g1, ... Gn& gn,     // global variables, will be
                           // explained later.
   std::list< token > :: iterator T1, ...,
   std::list< token > :: iterator Tn, // one parameter for each
                                      // token in the right
                                      // hand side.
   const std::list< token > & lookahead )
throw( refused )
```

The names of the parameters T1,..., Tn are derived from the types of the tokens on the right hand side of the rule. If the right hand side has form E PLUS E MINUS F, then the parameters will be E1, PLUS2, E3, MINUS4, F5. They have type `std::list< token > :: iterator` . In order to acces the token, the `*` operator has to be used. The attributes of the token can be accessed through the `->` operator. It is the responsibility of the user to ensure that accessed list elements exist. (But in the future, we may decide to use a checked list instead of std::list)

10 Syntax Errors and How to Recover from Them

When it encounters an error, the parser first calls method `tokenizer::syntaxerror( )`. After that, it will try to resynchronize by throwing away tokens until the parser is either resynchronized or it has thrown away more than **recoverlength** tokens. Resynchronization points are set by rules of the form

```
 % E    | _recover DOT
```

This means that if somewhere, while attempting to parse a E, something goes wrong, the parser will resynchronize when it sees a DOT. If token E has attributes, the recovery rule needs to find reasonable values for the attributes.

11 Remaining Options

- If `%selfcheck` is selected, the parse tables will be rechecked for completeness. This option is important for the developer only, which is me and not you.
- Option `%printclosed` prints the itemsets closed instead of simplified. This may be useful for teaching or finding the cause of conflicts.
- 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 the corresponding **.cpp** files 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.
- 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

12 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.

13 Namespaces

The definition of token and the parser are by default put in the top level namespace. In big projects, it may be useful to have the definition of token and the parser in dedicated namespaces. This can be obtained by adding the following directives to the input:
Using different namespaces, it is possible to generate different types of tokens, and different parsers in the same project, but one must be careful that the generated files do not overwrite each other. This can be obtained by putting the different versions of `token.h, token.cpp, parser.h, parser.cpp` in different directories.

If one uses namespaces, one must be careful how to address types and functions: Attribute definitions of form `%attribute name type` must name `type` in such a way that `N::type` is the correct addressing, where `N` is the namespace given by `%tokennamespace`. Global variable definitions of form `%global name type` must declare `type` in such a way that `type` is a correct name on top level. If `type` contains any namespaces, they have to be mentioned explicitly in `type`.

Maphoon puts the semantic action in an anonymous namespace on toplevel. If one wants to address functions or classes from inside a semantic action, they must be addresssed by the full name, including all namespaces.

14 Interface to the Parser

Maphoon reads from standard input, or from a file that is specified as parameter. Lines that do not start with `%` are copied to the file `parser.cpp` without change. Lines that start with `%` must contain a valid Maphoon directive. The implementation of the parser is written at the end of the file `parser.cpp`. The parser is declared in the file `parser.h`, and it has the following constructor:

```
parser( G1& g1, G2& g2, ..., Gn& gn,
        tokentype start,
        unsigned int recoverlength );
```

The tokenizer is described in Section 7. The variables g1, . . . , gn are the global variables, their purpose is explained in Section 6. The parameters are accessible in reduction code.

When the parser is called, **start** denotes the start symbol that the user wants to parse. Only symbols that were declared as startsymbol in the input can be used as start symbol. Otherwise, the parser quits with an error message `could not find startsymbol`. The parser uses `read. lookahead` for returning the result of the parse. There are four possibilities:

1. The parse was succesful and the parser did not need a lookahead for deciding that it reached the end of the input. This happens when the accepted input is not a prefix of another acceptable input. In that case, `read. lookahead` has size 1 and consists of the start symbol.

2. The parse was successful, but the parser needed a lookehead for deciding that it reached the end of the input. This happens when the accepted input is the prefix of another acceptable input string. In that case, `lookahead` has size 2 and consists of a start symbol followed by the encountered lookahead symbol.

3. The parse could not recover from a syntax error and reached a lookahead symbol while trying to recover. In that case, `read. lookahead` has size 1, and consists of the encountered lookahead symbol.

4. The parser could not recover from a syntax error and gave up, because it threw away more than recoverlength symbols. In that case, `read. lookahead` has size 1, and consists of a single `_recover` symbol.

Note that, if a syntax error occurred from which the parser could recover, it will return in state (1) or (2). It is the responsibility of the user (either in function `read. syntaxerror( )` or by the value of the returned attributes) to keep track of encountered errors.

If the parser behaves in an unexpected way, it can be debugged by including a definition of form `#define MAPH_DEBUG 1` in the file `parser.cpp`. When included, the parser prints a lot of debugging information about its state and its decisions while running.

15 Bugs, Missing Features, Possible Improvements

- The parse tables are quite space efficient, but not time efficient. Maybe using the original Yacc compression technique is better.
- There is a strange feature originating from the mixture of syntax and semantics. Suppose that one has a rule `E : E DIVIDES E` . It is natural to attach the following action to it:

  ```
  if( E3 -> val. front( ) == 0.0 )
  {
     std::cout << "division by zero\n";
     refuse;
  }

  if( !lookahead. size( ) ||
           shouldreduce( *DIVIDES2, lookahead. front( )))
  {
     E1 -> val. front( ) = E1 -> val. front( ) /
                           E3 -> val. front( );
     return E1;
  }
  else
     refuse;
  ```

  This causes a curious behaviour on $1/0 + 8$. The first parse $(1/0) + 8$ will fail, but Maphoon will simply construct the alternative parse $1/(0 + 8)$. It can be concluded that using `refuse;` for semantic errors was not a good idea, but maybe it has a useful application. Until I really understand the consequences, I do not intend to do anything about it.
- At present, there is no mechanism for graceful termination before the parse is complete. We will probably add an **accept** command in the future. At present, the user can only terminate the parser by throwing an exception.
- It it in principle possible that the tokenizer constructs more than one token at the same time, and appends them to `lookahead`. Maphoon was not tested for this case. This should be done.

16 Acknowledgement