Manual of Maphoon

Hans de Nivelle

May 11, 2010

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 Introduction

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. Dynamic extension of the syntax should 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 (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. 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 ).
% ;
```

2 Overview

Maphoon reads the description of a grammar, with associated actions, and it constructs a C^{++} implementation of a grammar for this language. It is possible to read the grammar from `stdin`, but it is better to prepare the input in a file of form `grammar.m`.

In order to work properly, Maphoon needs access to a file `idee.x`. This file is included in the main directory of Maphoon. It has to be copied to the current directory, so that Maphoon can find it. If everything goes well, Maphoon produces the files listed below. In addition, it prints a readable description of the parser to the standard output.

token.h/token.cpp: Contains a definition of `struct token`. `token` contains an `enum` which determines the type of the token. Optionally, it can have different fields containing lists of attributes. Which attribute lists can be present, is determined by the attribute declarations in the grammar.

`token` has a method `iswellformed( ) const` that checks if the token is well-formed, (which means that it has the right attributes). In addition, maphoon constructs an `<<`-operator for `token`.

It is not possible to have maphoon write the definition of `token` into a different file, but it is possible to tell maphoon to create the definition of `struct token` in a namespace of choice.

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, one 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 6. 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 definition of ASSERT. The following will do:

  ```
  #define ASSERT( X ) { assert( ( X ) ); }
  ```

  The definition can be placed in the input file.
- 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.

The distribution contains a couple of complete examples. If you preparing a project, then this is the recommended order:

1. Prepare a file `grammar.m`, and make sure that Maphoon accepts it. (Do not forget to include `idee.x` in the current directory.) Do not worry too much about semantic actions at this point.
2. Write the tokenizer, based on `struct token` 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.

3 Tokens

The declaration of `struct token` has the form below: The implementation of `token` is not written by the user!. It is generated automatically and written into the files `token.h` and `token.cpp`.

```
enum tokentype
{
   tkn_X1, tkn_X2, ..., tkn_Xn
};


struct token
{
   tokentype type;

   std::list< T1 > attr1;
   std::list< T2 > attr2;
```

```
      ...
      std::list< Tn > attrn;

      token( tokentype t )
         : type(t)
      {
      }

      token( ); // has no definition.

      bool iswellformed( ) const;
   };

   std::ostream& operator << ( std::ostream& , const token& );
```

As can be seen from the code, a token contains an attribute list for each of the types `T1`, ..., `Tn`. As a consequence, it can contain an unbounded number of attributes of each type `Ti`. The types `T1`, ..., `Tn` must have object semantics, because the elements are stored in lists. In order to tell maphoon, which lists of attributes must be included in the definition of `token`, attributes must be declared as follows:

```
%attribute attr T     // Declares an attribute with name attr
                      // and type T.
                      // It will result in a field
                      // std::list<T> attr;
```

Examples are:

```
%attribute name std::string // Declares attribute name of
                            // type std::string.

%attribute value double     // Declares attribute value of
                            // type double.
```

When the type of an attribute is a defined class or an STL class, its header file must be included in the header file of token. In order to allow this, maphoon has the `%intokenheader` command. The string that follows after `%intokenheader` is copied to the file `token.h` without changes. Here are few examples:

```
%intokenheader #include <string>
%intokenheader #include "../mysubdir/myclass.h"
```

It is possible to declare tokens, but it is not necessary. Tokens are implicitly declared when they are used in the grammar, so that only tokens that do not occur in the grammar need to be declared.

When constructing the parser, Maphoon lists the undeclared tokens that occur in the grammar. It also lists the declared tokens that do not occur in the grammar. The name of a token T must be chosen in such a way that the string attr_T is a valid C^{++} identifier. The general form of a token declaration is:

```
%token T1 T2 ... Tn   // Declares tokens T1, T2, ..., Tn.
```

It follows from the definition of `struct token` above, that attributes are stored in lists that can have unbounded size, but in order to avoid complete chaos, the user has to declare in advance which sizes are possible. This can be done by declaring *constraints*. Maphoon accepts two types of contraints. The first type gives a lower bound on how many occurrences of a certain attribute a certain token can have. The second type gives both a lower bound and an upper bound.

```
%constraint T attr n // Declares that a token of type T has at
     // least n elements in field attr. ( attr. size( ) >= n ).

%constraint T attr n1 n2 // Declares that token T has at least
     // n1 elements in the field attr, but
     // strictly less than n2 elements.
     // ( attr. size( ) >= n1 && attr. size( ) < n2 )
```

Examples are:

```
%constraint DOUBLE value 1 2 // A DOUBLE has exactly one
   // element in value.

%attribute form formula
%constraint FORMULALIST form 0 // A FORMULALIST has an unbounded
                               // number of formulas
```

If, for a given combination of a token and a field, no constraint declaration is present, then for the given token, the given field must be empty. Put differently, not declaring anything for the combination `%constraint T field` is equivalent to declaring `%constraint T field 0 1`.
The method `bool token::iswellformed( ) const` checks that the attribute lists of a token satisfy the constraints. Maphoon uses ASSERT to check that every token that is returned by a semantic action, is wellformed.

4 Semantic Actions

Semantic actions specify what has to be done when a rule is reduced. 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 5 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`.

The left hand side of the rule being reduced is not available as variable. (This is different from Yacc, where the lhs is represented by variable `$$`.) The lhs is returned by a `return`-statement. 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)

5 Global Variables

Global variables are variables that are passed as reference to every semantic action. They can be used for example for storing operator priorities, for declarations of variables, or for storing values of variables. Such parameters cannot be stored in attributes because attributes are passed only bottom up. Global variables enable information flow from left to right through the parse tree. A global variable is declared by a statement of form

```
%global name Type
```

A global variable will become a parameter of the parser (See Section 10 below), and it will be passed to every action (See Section 4 above). Global variables have to be declared and initialized by the function that calls the parser. As an example of a global variable declaration, consider

```
%global memory std::map< std::string, double >
```

`memory` is a data structure that stores values (of type double) of identifiers. An example of its use is shown in the semantic action of the rule `E : IDENTIFIER` above.

6 The Tokenizer

The tokenizer is an object that is passed as a parameter to the parser. Its main task is read input from the input source, and to group it into tokens. When the tokenizer reads a token, it places the token on a list called `lookahead`. In addition, the tokenizer must also have a method `syntaxerror( )` that processes syntax errors. One might argue that processing of syntax errors has nothing to do with reading input, but in our view it makes sense to place the processing of errors in the tokenizer because of the following reasons: In some way, the errors are a property of the input, so that it is reasonable to collect them in the device that reads the input. Also, when a syntax error is reported to the user, one usually wants to say something like: 'syntax error in line L of file F ' The tokenizer knows best about L and F. The tokenizer must have the following interface:

```
struct tokenizer
{
   std::list< token > lookahead;

   void scan( );
      // Get a token from somewhere and push it to the back
      // of lookahead.

   void syntaxerror( );
      // Prints or counts (or whatever) syntax errors.

};
```

When the tokenizer is unable to read input, because for example end-of-file is encountered, it should still append a token to lookahead. It is recommended to define a designated token ENDOFFILE this, and possibly also a designated token READERROR.

7 Errors

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.

8 Description of the Grammar and Start Symbols

Maphoon can construct a parser for more than one grammar at the same time. The different grammars share their set of tokens and rewrite rules, but they have different start symbols. Together which each start symbol, one has to specify the possible tokens that can follow a correct input defined by the start symbol. We will call these tokens the lookaheads of the start symbol.
The following defines a start symbol S with lookaheads L1, . . . , Ln:

```
   %startsymbol S L1 L2 ... Ln
```

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

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

If a start symbol is defined more than once, the lookahead sets are simply merged. The following group of startsymbol declarations is equivalent to the single declaration above:

```
   %startsymbol S L1 L2
   %startsymbol S L3
   ...
   %startsymbol S Ln
```

When a symbol L is declared as a lookahead symbol of a start symbol S, the symbol L must be not reachable from S. More precisely, it is not allowed that it is possible to rewrite S into a word W that contains L. Otherwise, the parser would not know when to stop during error recovery. Maphoon checks that no lookahead symbol 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.

9 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:

```
%parsernamespace N1
   // Define the parser in namespace N1.
%parsernamespace N1 :: N2 :: ... :: Nn
   // Define the parser in namespace N1 :: N2 :: ... :: Nn.

%tokennamespace N1
   // Define struct token in namespace N1.
%tokennamespace N1 :: N2 :: ... :: Nn
   // Define struct token in namespace N1 :: N2 :: ... :: Nn.
```

In a similar way, it is possible to tell the parser that the tokenizer is defined in a separate namespace. This is done by the directive:

```
%tokenizernamespace N1
   // Assume that the tokenizer is defined in namespace N1.
%tokenizernamespace N1 :: N2 :: ... :: Nn
   // Assume that the tokenizer is defined in namespace
   // N1 :: N2 :: ... :: Nn.
```

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.

10 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 interface:

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

The tokenizer is described in Section 6. The variables g1, . . . , gn are the global variables, their purpose is explained in Section 5. The parser passes the global variables to every reduction.

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.

11 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 is 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.