

Manual of Maphoon

Hans de Nivelle

March 19, 2026

Abstract

Maphoon is a tool that automatically generates a working 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^{++} – 17. If the attribute classes have correct resource management, the parser will automatically have correct 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 a 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 that a grammar is well-formed, 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.
%symbol S EOF
%symbol a b c A B

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

Call `./maphoon impatient.m`. Maphoon will analyse the file, and print the parse tables. (The file is present in directory `examples` under the name `lalr.m`.)

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^{++} . Good quality means: When the attribute types handle their resource management properly (use RAII), the constructed parser will also do that.
2. The parser must support attribute types that have no copying operations, which means that they can only be moved.
3. The parser must correctly implement the algorithms of the dragon book ([1]).
4. It must be possible to extend the syntax at run time. Concretely, we want to support the possibility of defining Prolog-style operators (see Sterling and Shapiro [4]) In Prolog, it is possible to declare `op('+', 'yfx', 200)`, after which `+` will be a leftassociative 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 if there is one.

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 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 when there is a conflict.

5. We want nice error reporting. A bottom-up parser, as generated by Maphoon, detects an error at the earliest possible point, which is as soon as it has become impossible to continue the input in such a way that the final result will be syntactically correct. We added an approach to error reporting where the presence of certain symbols in the input triggers an expectation. This is not automatic. The user specifies sequences of symbols with their resulting expectation. When the parser gets stuck, it looks for the most recent trigger, and reports what it expects, and what it received instead. In our experience, this gives good error messages.

3 Running Maphoon

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

1. The first argument must be a text file with extension `.m` or `.M`. It must be the name of a text file containing the 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 directory of the Maphoon sources. 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 Maphoon source directory, because it also contains files `symbol.h` and `symbol.cpp` which 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`. It is normally the directory containing the Maphoon sources (and the executable.)

The recommended use sequence is as follows:

1. Prepare a file `grammar.m`, and make sure that Maphoon outputs tables. Check the warnings. The best way to do this is to redirect the output, and read the output in a file. Have a look at the parse tables, and check that they look reasonable.
2. Call Maphoon with a second argument, and let it generate the symbol class. Write the tokenizer, using `symbol` that was produced by Maphoon. It is usually necessary to create additional symbols (e.g. whitespace symbols) that do not occur in the grammar. This will result in warnings. These can be ignored.
3. When the tokenizer is complete, you can call Maphoon with three arguments, start adding semantic actions, compile 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 contain 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 implies that they should not be references. They also should not be `const`. Maphoon will warn when attribute types are reference or `const`. Pointers are possible but not recommended, because they don't have value semantics. For efficiency, it is recommended that the types have efficient moving operators.

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

5 Managing Source Information

It is often useful to attach source information to symbols. This information can be used for generating error messages. For simple parsers, one can report the current line number of the tokenizer, but this will not work for errors that are detected at a later stage, for example type checking errors. If one wants to accurately report those, one needs to keep track of the source of a symbol.

Maphoon views source information as an interval of file positions from the start of the symbol to the end of the symbol. When a grammar rule is reduced, the intervals of the right hand side symbols are merged to obtain the interval of the left hand side symbol. The user should create an interval class, and define the merging operation, and make sure that the tokenizer includes the intervals. The rest is automatic. In case one does not care about intervals, it is sufficient to use only the start positions. In that case, the merging operator does not need to do anything. The steps are as follows:

1. Create a location or interval class. The name can be chosen freely. For simplicity, we assume the name is `sourceinfo`. The `sourceinfo` class should be lightweight, because it will be included in every symbol. If you want to include a filename (or a stack of filenames), then it is better to use a pointer. Class `sourceinfo` must be printable with operator `<<`, and must have a `merge` method, whose signature must be `void merge(sourceinfo)`, `void merge(const sourceinfo&)`, or `void merge(sourceinfo&&)`. If the location contains only the start position, the body can be empty. For an interval class, the method must copy the end location of its argument to the end location of `this`.

In our experience, class `sourceinfo` is always simple enough to be defined header only.

The location class does not need to (and probably should not) have a default constructor.

2. Add `%symbolcode_h{ #include "sourceinfo.h" }` to the grammar, so that the `symbol` class knows about it.
3. Add `%infotype{ sourceinfo }` to the grammar. The name `sourceinfo` can be changed to any name, as the file `sourceinfo.h`.
4. Make sure that the tokenizer adds `sourceinfo` when it constructs a symbol. It is always the second constructor argument after `symbolval`.

When the grammar contains an `infotype` definition, the generated symbol class has a field `std::optional< sourceinfo > info`, where `sourceinfo` is the name of the source information class. In addition, `std::optional< sourceinfo >` will become the second parameter of every constructor of class `symbol`, after the `symbolval`.

The `info` field has to be optional, because symbols that originate from reducing a rule with empty right hand side `Sym => ;` have no source information.

At reduction time, the parser will try to compute the `info` field of the left hand side from the `info` fields of the right hand sides, by calling the `merge` method of the first right hand side symbol that has `sourceinfo`, with the others that have `sourceinfo`. The result will be copied into the left hand side. The user does not need to worry about this.

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 { type } 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 already done by Maphoon. It will complain if you try. Pointers are possible. The parameters become reference arguments of the constructor `parser::parser()` in the same order in which they appear in the input file. When the parser is constructed, the constructor must be called with reference arguments. Note that there is no

way of declaring local variables whose scope is the complete parser. If one needs such a variable, it must be declared as parameter and initialized outside of the parser. Using global variables is bad style, always use parameters.

7 Specifying the Tokenizer

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

```
%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 itself. 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. In order to do this, one can attach code to grammar rules. The attached code is traditionally called *action code* (See [3]), but *reduction code* would be better fitting. Grammar rules have form $A \Rightarrow B_1 \dots B_n$. We call A the *left hand side (lhs)*, and $B_1 \dots B_n$ the *right hand side (rhs)* of the rule. The action code must specify how the attribute of A is computed from the attributes of $B_1 \dots B_n$. In addition to that, action code can modify parameters of the parser, or have other side effects. If A has an attribute type distinct from `void`, the action code must contain a return statement that returns the attribute of A . If A has an attribute different from `void` and there is no action code, Maphoon will create an error without constructing a parser. For example, the following example will result in an error:

```
%symbol { int } A

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

This can be pretty annoying because one usually wants to add the action code gradually. In order to make this easier, one can write

```

%skip
... here unfinished rules
%endskip

```

around the rules that do not have action code yet.

If some of the B_i 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 declared type of the symbol to which they are attached. The return statements set the attribute value of the lhs symbol. If the lhs symbol has no attribute, the action code can simply fall over the end.

In the given example, all action code consists of a single return statement, but it may use any C^{++} statement, it may extend over multiple lines, and it may contain more than one return statement. We recommend that one keeps action code moderate in size.

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

```

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

```

The action code relies on the following declarations:

```

%parameter{ std::map< std::string, double > } memory
%parameter{ std::vector< std::string > }      errorlog

%symbol{ std::string }          IDENT
%symbol{ double }              E

```

If there were errors during computation of the attribute of **E**, they are printed and the error log is cleared. Otherwise, the value of **E** is stored in the value **IDENT**.

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 with 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 designated quit command. This will be discussed in Section 11.

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 Section 10.

If one wants to be sure that the return value is moved (instead of copied), one must take some precautions: Local variables and expressions returning a value, are always moved. This follows from the *C++* standard. If one returns an attribute originating from the right hand side, it is not moved by default. In that case, one must use `std::move`. Suppose that symbol **A** has attribute type **Atype**, and that the following rule:

```
A => A:a B:b
{
    Atype loc = ...
```

```

    if( ... ) return A(b) // Temporary, will be moved.
    if( ... ) return loc // Local variable, will be moved.
    return std::move(a);
        // Originates from lhs, std::move is needed.
};

```

9 Interacting with the Symbol Class

Maphoon creates a symbol class in files `symbol.cpp` and `symbol.h`. The tokenizer has to create symbols. This is done by calling one of the constructors. If an infotype is present, the constructors have form

```

symbol( symbolval, const std::optional< infotype > &, const A& );
symbol( symbolval, const std::optional< infotype > &, A&& );

```

```

symbol( symbolval, const std::optional< infotype > & );
    // For symbols without attribute.

```

If no infotype is present, the constructors have form

```

symbol( symbolval, const A& );
symbol( symbolval, A&& );

```

```

symbol( symbolval );
    // For symbols without attribute.

```

For primitive types, *A* will be a value parameter, instead of reference. Type `symbolval` is an enumeration type that consists of the names of all declared symbols, preceded by `sym_`. If one defines a symbol `NEWLINE`, `symbolval` will contain `sym_NEWLINE`.

The constructors do not check that the attribute type *A* corresponds to the declared attribute type of the symbol. For example, one can declare `%symbol{ std::string } IDENTIFIER` and construct `symbol(sym_IDENTIFIER, 3.14)`. This will eventually cause the parser to throw `bad_variant_access` when the field is accessed. If one assigns 1 to the field `checkattrtypes`, the parser will complain when the attribute type does not correspond to `symbolval`. If one assigns `checkattrtypes = 2`, the parser will complain and stop when the attribute type is wrong.

If debugging is switched on (by assigning a value different from 0 to the `debug` field), the parser will print the symbols on the parse stack, including the attributes. In order for an attribute to be printable, define the function `void print_attr(const A&, std::ostream& out)` for attribute type *A*. Use `symbolcode_cpp` (see Section 18). `void print_attr` is already defined for most of the primitive types, and also for `std::pair`. The definition for `double` is:

```
void print_attr( double d, std::ostream& out )
    { out << d; }
```

The definition for `std::pair` is:

```
template< typename T1, typename T2 >
void print_attr( const std::pair<T1,T2> & pr, std::ostream& out )
{
    out << '[';
    print_attr( pr. first, out ); out << ',';
    print_attr( pr. second, out ); out << ']';
}
```

Defining `print_attr` is easy in most cases. If `A` is a container type, one can call `void print_range(Iter i0, Iter i1, char c0, char c1, std::ostream& out);` from `print_attr`. The function will print the range `i0..i1` between a `c0` and a `c1`. For example, in order to print a set of characters, include

```
%symbolcode_cpp {
    void print_attr( const std::set< char > & set, std::ostream& out )
    {
        print_range( set. begin( ), set. end( ), '{', '}', out );
    }
}
```

For the rest, if anything is unclear, one can always look at the code in the files `symbol.cpp` and `symbol.h`. It is designed to be readable.

Note that in action code, one does not deal directly with the `symbol` class. One only needs to compute the attribute of the left hand side and return it.

10 Runtime Conflict Resolution

Conflicts arise when the grammar is ambiguous, or when parsing requires looking ahead further than one symbol. This can be caused in principle by mistakes in the grammar. If this is the case, the grammar can be corrected, and the conflict will go away.

A common source of ambiguity are underspecified priorities between operators. In most cases, the grammar can be made unambiguous by introducing additional non-terminal symbols. Consider for example:

```
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 receives priority. It can be made non-ambiguous by introducing additional non-terminal symbols:

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

This solution does not always work, because some programming languages (Prolog [4] is probably the most important one) allow runtime definition of operators. In addition, some languages have so many levels of operator priorities, that it may be easier to use the runtime conflict resolution of Maphoon.

Another form of ambiguity are reduction rules that are only possible when the right hand side meets certain requirements. For example, instead of making the tokenizer recognize reserved words, one can use grammar rules to recognize them. This has the advantage that reserved words can be used as ordinary identifier at positions where the grammar does not allow the reserved word. For example, one can define:

```
While => IDENT:s; // With condition that s == "while"
```

A similar situation occurs in *C*, where an identifier can be a type name if it is declared as type:

```
TypeName => IDENT:s; // Assuming that s was defined as type.
```

In this case, one can also pass the type information to the tokenizer and have the tokenizer make the decision, but making the decision in the parser will ensure that no `TypeNames` are created at places where a type name cannot occur. Perhaps the most famous ambiguity is the dangling else problem:

```
Statement => IF Expr THEN Statement
           | IF Expr THEN Statement ELSE Statement
           ;
```

In the input `IF Expr THEN IF Expr THEN Statement ELSE Statement`, the `ELSE Statement` can be connected either to the first or to the second `IF`. Usually the `ELSE` is connected to the nearest `IF`. This can be enforced by requiring that lookahead is not `ELSE` in the first rule.

In order to attach a precondition to a grammar rule, use `%requires`, followed by code that checks the precondition. This code has the same form as action code, but attribute variables and parameters are only `const` accessible. Otherwise the precondition code could make changes to the state of the parser and after that, decline the reduction.

The code of the precondition must return a `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 still considered. In most cases, there is no need to block shifts, and returning a `bool` is sufficient.

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

```
Quit => IDENT : id
%requires { return id == "quit"; }
    // booleans are handled as expected.
;

Show => IDENT : id
%requires { return id == "show"; }
;
```

The rules have no action code, because symbols `Quit` and `Show` have no attribute. In nearly all cases, the precondition can simply return a `bool` that will be correctly converted into 0 or 1. Returning a negative number is needed only for nonassociative operators.

In case of a conflict, `maphoon` will try all reductions, and pick the first one that accepts. In order for this approach to have predictable behaviour, the programmer must have control over the order in which reductions are attempted. The `%reductionseq` statement provides this control. If there is a state in which more than one reduction is possible, the lhs symbols of the reducible rules must be listed together in a `%reductionseq` statement in order of preference. If they do not occur together in a `%reductionseq` statement, `Maphoon` will print a warning, and create a parser that applies 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 a syntax error.

Precondition code can use the lookahead to decide if the reduction should be made. This is important when deciding between priorities of operators. It is guaranteed that there is a lookahead when `%requires` is evaluated. The lookahead can be accessed by calling `getlookahead()` and has type `const symbol&`.

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, getlookahead( ). val ); }
%reduces
    { return form( op_and, f1, f2 ); }
;
```

Function `decide` can return 1 if `sym_AND` if has higher priority than `getlookahead(). val`, and 0 if the lookahead is an operator with 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. Its implementation could be as follows:

```
short int decide( symbolval s1, symbolval s2 )
{
    // std::cout << "Deciding between: " << s1 << " " << s2 << "\n";

    if( s1 == sym_NOT || s1 == sym_AND )
        return 1; // Reduce, both have highest priority.

    if( s1 == sym_OR )
    {
        if( s2 == sym_AND )
            return 0; // AND has higher priority, so we shift.
        return 1;
    }

    if( s1 == sym_IMP )
    {
        if( s2 == sym_AND || s2 == sym_OR || s2 == sym_IMP )
            return 0; // AND/OR have higher priority, and IMP
                // is rightassociative, so we shift.
        return 1;
    }
    ...
}
```

It is guaranteed that `getlookahead()` can be called when precondition code is called in case there is more than one option. The parser makes reductions without lookahead only in states where this reduction is the only option. It is in principle possible to add preconditions to such reductions, but it is a bad design, because the error should have been caught earlier.

The only purpose of `%reduces` is to make the input look better. It can be omitted, but it looks not nice:

```
Formula => Formula:f1 AND Formula:f2
%requires
    { return decide( sym_AND, getlookahead( ). val ); }
{ 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.

11 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.

12 Syntax Errors: Reporting and Recovery

Maphoon uses an optional, experimental approach to error reporting, which we will explain shortly. Alternatively, the user can write an own error reporting function.

When the parser gets stuck, it calls method `void syntaxerror(std::ostream& errorstream)`. The parameter `errorstream` determines where errors will be reported. The `errorstream` can be provided as a parameter to `parser::parse`. The default is `std::cout`.

The Maphoon implementation of `void syntaxerror(std::ostream& errorstream)` uses a table of sequences of symbols that trigger expectations. The expectations are used to create informative error messages. Alternatively, the user can provide an own implementation of `syntaxerror()`. In that case, `%usererror` must be included in the grammar file, so that the standard implementation will not be generated. The actual implementation can be defined using `parsercode_cpp`.

If the user decides to create an own implementation, then `syntaxerror()` must first decide if the error is new, or the result of a failed recovery attempt. This can be done by looking at `timesincelasterror` and comparing it to `maxtrialperiod`. The Yacc manual [2] suggests that 3 is a good cut-off value, so this is the default value of `maxtrialperiod`. It can be reassigned.

The implementation provided by Maphoon tries to create an informative error message. The default message has form 'unexpected X', where X is the current lookahead symbol. If there is no lookahead, the message will be just 'syntax error'. If there is an `infotype`, the default `syntaxerror()` will try to include source information in the message.

The error message can be made more informative by adding what was expected instead of the lookahead. In order to define an expectation, one must specify a sequence of symbols that possibly occur on the parse stack, together with the expectation that one gets from seeing them. It is possible to add integers between the symbols that specify how far the triggering symbols can be apart. Adding *n* means that the symbols can be at most *n* positions apart. No integer means that there can no other symbol between the triggering symbols. Instead of distance 1, a star (*) can be used. One can also use multiple stars to obtain bigger integers. A simple example is

```
%errors
    LPAR 5 => "closing parenthesis";
```

This rule can be alternatively written as

```
%errors
  LPAR * * * * * => "closing parenthesis";
```

If the parser gets stuck, and there is an LPAR on the parse stack at a position not deeper than 5 symbols, the error method will report that it expected a closing parenthesis instead of the lookahead. Here is another example, containing different possible expectations:

```
%errors
  LPAR 10 => "a )";
  IDENTIFIER LPAR 5 => "a function argument";
  ( TIMES | DIVIDES | MODULO ) => "a factor";
  ( PLUS | MINUS ) => "a summand";
```

If the parser gets stuck with PLUS or MINUS on the top of the stack, the error function will report that it expected a summand (instead of the lookahead). Similarly, if the parser gets stuck after seeing IDENTIFIER LPAR and not more than 5 other symbols, it will report that it expected a function argument. If more than one rule can be applied the rule that matches highest on the parse stack will be used to define the expectation. The easiest way to find suitable expectations is by setting `debug = 1`, creating the error, and looking at the parse stack for characteristic symbols.

If the default error method is not good enough, and the user wishes to define an own error function, it is possible to include `%usererror` in the input file, and write an own `void syntaxerror()` method. Inside `syntaxerror()`, one can access all parameters and to the lookahead. In order to get the lookahead, call `const symbol& getlookahead()`.

Since the parser uses default reductions, it may lose information when it encounters an error. This can be prevented by setting `%nodefaults`. It will result in more informative error messages. Unfortunately, it may create problems with interactive applications, see Section 16.

Recovery works the same as in Yacc (See [2]). The parser inspects the stack for `_recover_` transitions and 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 the parser cannot recover within the limit specified by `maxrecovery`, or a terminator is encountered while trying to recover, the parser will return `sym__recover_`.

If after seeing the SEMICOLON one decides that recovery was not a good idea after all, one can assign `timetosaygoodbye = true`; but one must be sure to report the error or to log it.

13 Movability of Symbols

The parser will be more efficient when the symbol class is nothrow-movable, because it will be used when the parse stack is reallocated. In order to obtain that, every attribute must be nothrow-movable, and the `infotype` must be nothrow copyable, if it is present. If one sets the field `checkmovable` to 1, the parser will create a warning when `symbol` is not movable. If `checkmovable` is set to 2, it will quit when `symbol` is not movable. It is sufficient to make this check once when the set of possible attributes changes. Also see the remarks at the end of Section 8 about ensuring that attributes are moved. Maphoon can be used with non-copyable attributes.

14 Dealing with C++ Errors

User code is preceded by a `#line` directive, so that errors will be reported with their correct 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 the error does not originate from user code, delete the `#line` directives from the parser and the symbol files.

- If the compiler complains about duplicate constructors in the `symbol` class, the most likely cause 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.
- If the compiler complains that it cannot find some constructor of the `symbol` class, the most likely cause is that no symbol was declared with the required attribute type.
- If the compiler cannot resolve one of the attribute types while compiling `parser.cpp`, it will produce monumental error messages. Check that all files where the attribute types are defined, are included. Use `symbolcode.h`.
- If the compiler says `error: forming reference to void`, this is likely caused by the fact that one did not declare the type of an attribute that one is trying to use in action code.

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 parser class:

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

Action and preconditions code become methods of the parser class, so that the parameters are accessible. The lookahead can be accessed by calling `getlookahead()`. Presence of a lookahead can be checked by calling `haslookahead() const`.

The parser is started by calling method `symbol parse(symbolval start, std::ostream& errorrep`. The `symbolval` 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 `std::out_of_range`. The `errorreport` is the stream into which the default error reporting function `syntax_error(std::ostream&)` will report errors.

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

1. The parse was succesful and the input reduced into the start symbol. The start symbol is returned, containing a possible attribute. 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. In that case, it also 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 errors that were recovered.

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 Interactive Applications

Interactive applications are applications where the program repeatedly asks for input, and immediately processes the given input. Parsers constructed by Maphoon can be used for interactive applications, but there are a few problems that one has to take into account: One has to take care of the moment at which commands are reduced, one has to make sure that the parser can be ended by

means of a quit command (in addition to encountering EOF), and one has to choose whether the parser is called once for the complete session, or separately for each command.

In an interactive implementation, it has to be ensured that the user receives feed back immediately after typing a command. The moment when reductions are made depends on whether `%nodefaults` is set. If the grammar contains a rule of form `Command => Expr:e SEMICOLON { Process e }` and `%nodefaults` is set, the parser will reduce the rule only when the symbol after the SEMICOLON has been read, which implies that the user has to start typing the next command before `Expr` is processed. Without `%nodefaults`, the parser will reduce immediately after reading the SEMICOLON (assuming it is the only reduction possible, and nothing can be shifted). If one wishes to use `%nodefaults`, the rule can be rewritten as follows:

```
Command => E SEMICOLON ;
E => Expr:e { Process e };
```

In our experience, there is no need to use `%nodefaults`. Multiple commands can be processed either completely inside the parser, or by designing a parser that processes a single command and returns. The former option is easier to implement:

```
%startsymbol Session EOF
%rules

Session => Session Expr:e SEMICOLON
{ process e.
  If e is a quit command, set timetosaygoodbye = true }
| _recover_ SEMICOLON
;
```

The parser will stop when it reached EOF, or when a quit command is encountered.

If one wants to call the parser separately for each command, one has to be careful with terminators, and implement resynchronization after a syntax error by hand. We show from an example how it should be done, because an example is easier to read than explanations.

```
parser prs = parser( ... );

while( forever )
{
  prs. ensurelookahead( );
  if( prs. getlookahead( ). val == sym_EOF )
    (end-of-file reached in normal fashion.)

  symbol res = prs. parse( sym_Expr, std::cout );
  // parser will start with existing lookahead,
```

```

        // syntax errors will be reported std::cout.

if( res. val == sym_Expr &&
    prs. getlookahead( ). val == sym_SEMICOLON )
{
    // Everything went well.

    if( res. attr is a quit command )
        exit normally.

    (process res. attr.)
}
else
{
    // It was a syntax error:

    while( prs. getlookahead( ). val != sym_SEMICOLON &&
           prs. getlookahead( ). val != sym_EOF )
    {
        prs. resetlookahead( );
        prs. ensurelookahead( );
    }
}

if( prs. getlookahead( ). val == sym_SEMICOLON )
    prs. resetlookahead( );
    // If lookahead is EOF, we keep it.

prs. reset( );
    // This does not reset the lookahead, so if we
    // encountered EOF, we will quit in the next iteration.
}

```

17 Possible Problems

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

- Some parser generators allow the use 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. It's not worth it.
- It has been reported that parsers generated by Maphoon compile unreasonably slow when the parse tables are big, but I have not seen a concrete example. In order to fix this, I need an example where the problem shows up. If you have one, please contact me.
- At present, it doesn't seem possible to use `constexpr` to implement a practical parser generator, but it might change in the future.

18 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.
- By adding option `%showclosures`, the itemsets are shown closed instead of simplified. This may be useful for teaching or finding the cause of conflicts. It does not 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 without namespace.
                    // Intended for including attribute types.
%symbolcode_cpp{ } // Goes into anonymous namespace in
                    // symbol.cpp. Intended for defining print_attr
%parsercode_h{    } // Goes into parser.h without namespace.
%parsercode_cpp{ } // Goes into parser.cpp in the
                    // same namespace.
```

We recommend that the `.h` files are used for `#includes` only.

One should not define anything in `%parsercode_cpp`, unless one declares it in `%parsercode_h`, or puts it in anonymous namespace. As a general rule, one should not define anything important in `parser.h` or `symbol.h`, but use a separate file and include it. It is not possible to declare anything that is accessible from outside using `%symbolcode_cpp`, because it is put in anonymous namespace.

- If `%nodefaults` is set, the parser will never reduce without lookahead even in states where reduction is the only possibility. Suppose that one has a rule `S => E SEMICOLON`. When the semicolon has been pushed, the parser is in a state where the only possibility is reducing the rule. In the default mode, the parser will reduce without lookahead. If `%nodefaults` is set, the parser will still take a lookahead, and create an error when the

lookahead is not in the follow set of S . The advantage is that in case of error, the chances for recovery are better. The disadvantages are a bigger parse table, and a radically changed interactive behaviour of the parser: If in the action code, the system has to react to the input, this reaction will come only after the user typed the first symbol after the semicolon.

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

19 Acknowledgements

I thank Danel Batyrbek, Aleksandra Kireeva, Tatyana Korotkova, Dina Mukhtubayeva, Cláudia Nalon and Olzhas Zhangeldinov. Special thanks to Zoltan Teki for extensively trying out Maphoon in January 2023. Zoltan found a bug in the tokenizer generator, and insisted on adding support for move-only attributes.

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.