Using the Tokenizing Tools of Maphoon

Dina Muktubayeva and Hans de Nivelle

May 18, 2021

Abstract

We describe how the tokenizing tools of MaphTT are used.

1 Quick Start for the Impatient

Get the files from directory `lexing`. Create a main file that contains `main( int argc, char* argv[] ) { }`. You can also use the existing file `tester.cpp`. Make sure that the file contains the following includes:

```
#include "algorithms.h"
#include "minimization.h"
#include "generators.h"
#include "deterministic.h"
```

Make sure that your compiler compiles it.
Now suppose we want to obtain the automaton for regular expression

$$(a|b)^* \,|\, a^*.$$

Write:

```
using namespace lexing;

acceptor<char> reg = ( just('a') | just('b') ).star( ) |
                     just('a'). star( );

std::cout << reg << "\n";
```

This prints an acceptor that accepts the regular expression. As next step, we want to make this automaton deterministic. Because MaphTT has no operations on `acceptor`, except for constructing them, we need to obtain a classifier first. A classifier has two template parameters, the alphabet and the type that we are using for the token classes. If one cares about efficiency, the token class should be `int` or a dedicated `enum` type. We don't care about efficiency, so we just use `std::string`. When a classifier is constructed, one needs to provide the default classification that will be used when classification fails.

```
classifier< char, std::string > cl( "#error" );
cl. insert( reg, "#ourlang" );
std::cout << cl << "\n";
```

When the classifier is printed, it only prints the classifications of states that do not classify as `#error`. We see a lot of ϵ-transitions, starting from `Q0`, so we write

```
cl = make_deterministic( cl );
std::cout << cl << "\n";
```

Unfortunately, this results in a `logic_error`, with an additional message that `empty word classifies as #ourlang`. This is caused by the fact that the original expression accepts the empty word, which would make it impossible to construct a usable tokenizer. The empty word would be a valid token, and the tokenizer would keep on returning this token forever. The problem can be solved by replacing `.star( )` by `.plus( )` or by adding a terminator. We will add (;) as terminator. Add the line

```
reg *= just( ';' );
```

Now the code runs and constructs an automaton with 5 states. This is ridiculously many of course. We can add:

```
std::cout << "minimization\n";
cl = minimize( cl );
std::cout << cl << "\n";
```

This results in a classifier with two states. The final classifier is readable. State `Q0` loops on a, b and jumps to `Q1` when a (;) is encountered.

2 Creating a Prototype

Assume we want to create a calculator, whose tokens are

- Identifiers.
- Floating point numbers.
- A few operators: `+, -, *, /, %`.
- Opening and closing parentheses: `( )`.
- Semicolon to end the input: (`;`).

Since we are going to care about efficiency eventually, we create an enumeration type:

```
enum tclass
{
   tc_float = 1,
   tc_ident = 2,
   tc_add = 10, tc_sub = 11, tc_mul = 12, tc_div = 13, tc_mod = 14,
   tc_lpar = 20, tc_rpar = 21,
   tc_semicolon = 30
};
```

When creating a tokenizer, we recommend that one creates the classifier in a function. That will be easier for the later steps. So, we define

```
lexing::classifier< char, tclass > buildclassifier( )
{
   using namespace lexing;

   classifier< char, tclass > cl( tc_error );

   cl = make_deterministic(cl);
   cl = minimize(cl);
   return cl;
}
```

Adding tokens is straightforward:

```
   auto idfirst = range('a', 'z') | range('A', 'Z');
   auto idnext = range('a', 'z') | range('A', 'Z') |
                 just('_') | range('0', '9');


   auto number = (just('0') |
               (range('1', '9') * range('0', '9').star()));

   cl. insert( idfirst * idnext.star(), tc_ident );

   auto exp = ( just( 'e' ) | just( 'E' )) *
                ( just( '-' ) | just( '+' )). optional( ) *
                ( range( '0', '9' ). optional( ) *
                  range( '0', '9' ). optional( ) *
                  range( '0', '9' ). optional( ));

   auto floatconst = number *
               ( just('.') * ( range('0', '9').plus()) ). optional( ) *
                 exp. optional( );

   cl. insert( floatconst, tc_float );
```

```
cl. insert( just( '+' ), tc_add );
cl. insert( just( '-' ), tc_sub );
cl. insert( just( '*' ), tc_mul );
cl. insert( just( '/' ), tc_div );
cl. insert( just( '%' ), tc_mod );

cl. insert( just( '(' ), tc_lpar );
cl. insert( just( ')' ), tc_rpar );

cl. insert( just( ';' ), tc_semicolon );
```

Token classes are printed as integers, which is a bit unpleasant. This problem can be solved by defining `operator <<` on `tclass`. The parser generator[1] automatically creates `enum` classes from the grammar with print functions.

Currently, the classifier will reject whitespace, so we add `tc_whitespace = 40` to `tclass` and add the following code to `buildclassifier`:

```
cl. insert( just( ' ' ) | just( '\t' ), tc_whitespace );

auto linecomment = word("//") *
                 ( every<char>(). without('\n') ).star() * just('\n');
cl. insert( linecomment, tc_whitespace );

auto blockcomment = word("/*") *
  ( every<char>().without('*') |
    ( just('*').plus() * every<char>().without('/').without('*')).star()
  ).star() * just('*').plus( ) * just('/');

cl. insert( blockcomment, tc_whitespace );
```

Now the classifier classifies whitespace and both types of C^{++}-comments.
In addition to the constructors shown in the examples, one can also use `word( )`, which can be called with a string constant.

3 Using a Classifier

In the previous section we constructed a classifier. MaphTT does not create complete tokenizers, it only helps with classifying. In particular, it doesn't help with computing token attributes. We think it is no problem because most tokens have no attribute, and computing attributes is easy when the token has been classified.

In order to classify input, one needs a class with at least the following methods:

- `bool has(n)`. Make sure that there are at least n characters in the buffer. Return `true` if this is guaranteed.

[1] not ready for release

- `C peek(i)`. Gets the i-th character. `has(n)` with $n > i$ must have been called before, and must have succeeded.
- `commit(i)`. Commit to i characters. This means that i characters will be removed from the buffer. `has(n)` with $n \geq i$ must have been called before, and must have succeeded.

We provide two classes that provide these methods: `filereader` and `stringreader`.

- A filereader can be obtained as follows:

  ```
  filereader inp( &std::cin, "stdin" );
  ```

  or

  ```
  std::string name = "filename";
  std::ifstream inputfile( name );
  filereader inp( &inputfile, name );
  ```

- A stringreader can be obtained as follows:

  ```
  stringreader inp( "3.14 / pi + 10" );
  ```

  or

  ```
  std::string s = "input400 /*xxxx*/ ;";
  stringreader inp(s);
  ```

  or

  ```
  stringreader inp( std::move(s) );
  ```

Once one has chosen the reader of one's fancy, one can call

```
auto p = readandclassify( cl, inp );
```

The result has type `std::pair<tclass,size_t>`, where `tclass` is the recognized class and `size_t` the number of characters read. We recommend to use `readandclassify` as follows:

```
symbol gettoken( reader& inp )
{
   static auto cl = buildclassifier( );

restart:
   if( !inp. has(1))
      return a symbol that marks end of file.
```

```
   auto p = readandclassify( cl, inp );
   if( p. second == 0 )
   {
      inp. commit(1); // to avoid looping forever
      return a garbage token
   }

   if( p. first is among the whitespace tokens )
   {
      inp. commit( p. second );
      goto restart;
   }

   if( p. first is a token that has an attribute )
   {
      attr = the attribute (use inp. get( )).
      inp. commit( p. second );
      return symbol( p. first, attr );
   }

   inp. commit( p. second );
   return symbol( p. first ); // without attribute.
}
```

If choosing is hard, one can also write:

```
template< typename S > symbol gettoken( S& inp )
{
 ...
}
```

If the user wants to create a different type of reader, this can be easily done. Both `filereader` and `stringreader` have short implementations.

4 Making it Efficient

The tokenizer in the previous is convenient, but not efficient at all. The classifier is recomputed every time the program is restarted, and direct use of the classifier is not efficient, even when it is deterministic. MaphTT provides two solutions:

1. Create a table based classifier, like Lex or JFLEX.
2. Translate the classifier into directly executable C^{++} code.

The interface is primitive: In the function that builds the classifier, one prints code that can be later compiled, and which can replace the call to `buildclassifier( )`.

Since classifiers are polymorphic in the character set `C` and the type `T` used for classification, one needs to provide functions that print `C` and `T`. In case of `C = char`, this is easy, because `char` can be printed as integers:

```
[]( std::ostream& out, char c ) { out << (int)c; }
```

One can also print `char` with single quotes, but then one has to take care of non-printable characters, which makes the function too complicated to define in a single lambda expression.

Dealing with `tclass` is a bit more tricky, because `enum` is not well-designed. The best solution is to print them by their names, but then one has to define a print function for `tclass`, and we didn't do that[2]. For the moment, we hack ourselves out of the situation by creating code that casts integers into tclass:

```
[]( std::ostream& out, const tclass t )
     { out << "((tclass) " << t << ")"; } );
```

We describe the details:

- If one wants to create a table based classifier, one must use

  ```
  template< typename C, typename T >
  void printdeterministic(
     const std::string& Cname, const std::string& Tname,
     const std::string& name,
     const lexing::classifier<C,T> & cl, std::ostream& file,
     const std::function< void( std::ostream& out, const C& c ) > & printC,
     const std::function< void( std::ostream& out, const T& t ) > & printT )
  ```

  The parameter `Cname` is the name of the character type, which `"char"` in our case. Similarly `Tname` must be `"tclass"`. If `C` or `T` is in a namespace, the namespace must be included. Parameter `name` is the name that the function will receive. It should be something like `"builddetclassifier`.

  Choose an unused file name (it will be overwritten!), and insert

  ```
  std::ofstream file( "det.h" );
  lexing::printdeterministic< char, tclass > (
     "char", "tclass", "builddetclassifier",
     cl, file,
     []( std::ostream& out, char c ) { out << (int)c; },
     []( std::ostream& out, const tclass t )
        { out << "((tclass) " << t << ")"; } );
  file. close( );
  ```

  before the return statement.

  When `buildclassifier( )` is called, it will construct the deterministic classifier.

  Include the resulting file before function `gettoken`, make sure that it compiles, and replace

[2] the parser generator will do it automatically

```
symbol gettoken( reader& inp )
{
   static auto cl = buildclassifier( );
```

by

```
symbol gettoken( reader& inp )
{
   // static auto cl = calculator( );
   auto cl = builddetclassifier( );
```

- The procedure for creating a classifier in C^{++} code is similar. The function that creates the code is:

```
template< typename C, typename T >
void printcode(
   const std::string& Cname, const std::string& Tname,
   const std::string& space,
   const lexing::classifier<C,T> & cl, std::ostream& file,
   const std::function< void( std::ostream& out, const C& t ) > & printC,
   const std::function< void( std::ostream& out, const T& t ) > & printT )
```

The generated function will be called `"readandclassify"`, preceded by the namespace `space`. If `space == "myspace::"`, then the generated function will be called `"myspace::readandclassify"`.

At the end of `buildclassifier`, insert:

```
std::ofstream file( "directlycoded.h" );
lexing::printcode< char, tclass > (
   "char", "tclass", "",
   cl, file,
   []( std::ostream& out, char c ) { out << (int)c; },
   []( std::ostream& out, const tclass t )
      { out << "((tclass) " << t << ")"; } );
file. close( );
```

Include the resulting file before function `gettoken`, make sure that it compiles, and replace

```
symbol gettoken( reader& inp )
{
   static auto cl = buildclassifier( );
```

by

```
symbol gettoken( reader& inp )
{
   // static auto cl = calculator( );
   auto cl = lexing::trivial( );
```

Since the directly coded classifier does not need any additional data, one could in principle remove the variable `cl`, but if one does that, one may have to put it back later when one wants to change the classifier again. Because of this, we created a trivial class `trivial` without fields, which acts as a placeholder for the classifier. The function `readandclassify` has `trivial` as first parameter.