

# Writing a Compiler Targeting CLR

**Tomáš Matoušek**

[tm.matfyz.cz/teaching/clr.htm](http://tm.matfyz.cz/teaching/clr.htm)

# Compilers in General

- frontend
  - lexer/scanner/tokenizer
    - reads a stream of characters (source code)
    - produces tokens (basic lexical elements)
    - available tools: Flex, Improved CsLex, ...
  - parser
    - reads a stream of tokens
    - generates
      - intermediary form: e. g. AST (Abstract Syntax Tree)
      - code
    - available tools: Bison, Garden's Point Parser Generator (GPPG), ...
- backend
  - code generator
    - generate IL Assembler source code
    - Reflection.Emit
      - run-time generated code
    - PERWAPI or other third party API
    - unmanaged API

# Lexer Generators

- Flex
  - classics
    - included in Unix
  - written in C, generates C code
  - works well but the generated code is messy
    - many macros, global variables, ...
  - can be adjusted to generate thread-safe C++/CLI compatible code
    - tricky
- Improved CsLex
  - based on CsLex
    - rewritten from JLex (Java)
    - Improved version fixes many bugs and adds new features
  - similar to Flex
    - but written in C# and generating C# code
    - goal: to be able to use Flex style tokens description but generate C# code
  - input: text file
    - usually with .l or .lex extension
  - output: text file
    - C# source file containing a class that implements the lexer

# CsLex Input File

- three sections
  - separated by %% at the beginning of the line
- section #1
  - contains user defined code copied to the beginning of the output file
- section #2
  - properties specification
    - specifies various properties of the resulting code
    - each on a separate line starting with %
      - e.g. %namespace GeneratedClassNameSpace
  - macro definitions
    - format: *macro-identifier white-space regular-expression*
    - maps a macro identifier to a regular expression
      - e.g. HexDigit [0-9A-Za-z]
    - macros can be used in the section #3
- section #3
  - defines a sequence of finite state automata descriptions (by regular expressions)
    - the order matters
  - associates a piece of C# code with the accepting states of each automaton
    - resulting lexer applies the automaton on the input source code
    - the code is executed when the automaton accepts
    - the code decides what the automaton acceptance really means
      - it needn't mean token outputting

# Section #3 in Detail

- Section #3 syntax
  - a sequence of rules:

```
((('<' lexical-states '>')? regular-expression white-space '{' C#-code '}' eoln)+
```

- lexical states
  - a lexical state determines currently applicable automata
  - defined by %x or %s in Section #2
    - %x means the state has to be listed explicitly in each applicable rule
    - %s means the state is implicitly present in each rule
  - automaton preceded by a list of lexical states is applicable only in the states listed
  - a stack on the lexer allows to switch to a new lexical state (push) and back (pop)
- regular expression
  - meta-characters
    - .? \* + < > { } [ ] ( ) | ^ \$
  - double quotes escape values
    - e.g. "/"\* means slash and asterisk
  - identifiers in curly braces refer to macros
    - e.g. {HexDigit} means ([0-9A-Za-z])
  - currently missing some constructs available in Flex
    - e.g. {n, m} quantifiers

# Commands in C# Code

- tokens
  - enumeration provided from outside
    - manually written or generated by a parser generator
- automaton acceptance action
  - outputting token

```
return Tokens.T_FOREACH;
```
  - ignoring acceptance (implicit action)

```
break;
```
  - extending token

```
yy_more();
```
  - shortening token

```
yy_less();
```
  - switching the current lexical state

```
BEGIN(LexicalState.ST_COMMENT);  
yy_pop_state();  
yy_push_state();  
yy_top_state();
```

# Lexer Properties

- %ignorecase
  - whether to generate case-insensitive lexer
- %class
  - lexer class name
- %attributes
  - attributes for lexer class, e.g. public
- %namespace
  - namespace where the lexer class should be declared
- %function
  - defines name of the token producing method, e.g. GetNextToken
- %type
  - type name of the tokens enumeration type
- %eofval
  - defines a token representing end of file
- %errorval
  - defines a token representing an error
- %char
  - count tokens character offsets
- %line
  - count tokens line offsets
- %column
  - count tokens column offsets

# Generated Lexer

- public members
  - token fetching method (e.g. GetNextToken)
  - constructor
  - initialization method (sets stream reader and initial lexical state up)
- private members
  - static tables (all automata combined into a single one)
  - lexical states stack
  - accepting method
    - switch over state
    - code pieces associated with automata copied here
  - supportive private routines
    - `yyles`, `yymore`, `BEGIN`, `yy_pop_state`, `yy_push_state`, `yy_top_state`
  - character buffer
    - resizes as needed, loads data from supplied stream reader
    - always contains whole token
      - $\text{token\_start} \leq \text{token\_chunk\_start} < \text{token\_end}$
  - auxiliary private routines for buffer manipulation
  - token position info
    - `token_start_pos`, `token_end_pos`
- additional functionality to be provided by user via partial class
  - e.g. routines for interpretation of the buffer content

# Parser Generators

- Bison
  - classics
  - used along with Flex
  - written in C, generates C code
  - can be adjusted to generate thread-safe C++/CLI compatible code
- Garden's Point Parser Generator (GPPG)
  - some minor improvements also required
    - position tracking, a handful of bug fixes
  - written in C#, generates C# code
  - input file similar to Bison
  - same grammar strength
    - LALR(1), context-free grammars subset

# Bison-like Input File

- three sections
  - separated by %% at the beginning of line
- section #1
  - user code copied at the beginning of the generated file
- section #2
  - properties specification
  - semantic value type declaration
  - tokens declaration
    - tokens are terminals in the grammar
  - optionally, non-terminals can also be declared here
- section #3
  - a sequence of production rules constituting the grammar
    - lhs of the rule must be a non-terminal (recall context-free grammars)
  - semantic actions
    - C# code pieces associated with reductions
    - give semantics to the rules
    - can evaluate the tokens, generate code, build AST, etc.

## Section #2: Properties

- %namespace
  - namespace where to declare the parser in
- %parsertype
  - name of the generated parser class
- %valuetype
  - name of the structure holding the semantic value
  - information associated with terminals and non-terminals
- %positiontype
  - name of the structure holding token position (optional)
  - position (line, column, offset) of the terminal/non-terminal
- %tokentype
  - name of the tokens enumeration
  - enumeration is generated to the resulting file
- %visibility
  - visibility of the generated types
- %start
  - starting non-terminal

## Section #2: Semantic Value Type Declaration

- %union
  - declares semantic value type fields
  - historical name
    - in Bison, fields are stored in a union
  - fields have to capture all possible values of the terminals and non-terminals
  - should be small yet not “compressed”
    - i.e. should allow efficient value getting and setting
  - example:

```
%union
{
    object Object; // stores instances of reference types
    double Double; // stores floating-point values
    int Integer;   // stores integer values
}
```

# Production Rules

- rule format

*non-terminal*: *rule-rhs* | *rule-rhs* | ... | *rule-rhs* ;

- rule-rhs is a white-space separated list of

- terminals
- non-terminals
- semantic actions enclosed in curly braces
  - usually at the end of a *rule-rhs*
    - executed on rule reduction
  - but can be in the middle as well
    - a new auxiliary non-terminal and rule introduced
  - can refer to items of the *rule-rhs* by \$n
    - \$n means the semantic value of the n-th item
    - @n means the position value of the n-th item
  - can refer to the resulting *non-terminal*
    - \$\$ means the semantic value of the non-terminal
      - should be assigned a value in a semantic action within the rule
    - \$@ means the position value of the result

# Declarations in Section #2

- tokens declaration
  - mandatory
  - %token
    - declares a token with a specified name
  - %left, %right, %nonassoc
    - declares a token with specified name and associativity
    - associativity used for grammar disambiguating
      - but make the grammar itself unambiguous is better
  - can optionally be followed by <FieldName>
    - e.g. %token<Integer> HEX\_NUMBER
    - FieldName is a name of the field of semantic value type
    - each \$ reference to the token in the grammar is suffixed with .FieldName
- non-terminals declaration
  - optional
  - %type<FieldName>
    - e.g. %type<Integer> expression
    - each \$ reference to the non-terminal is suffixed with .FieldName

# Generated Parser

- token enumeration
- semantic value type structure
- position structure
- parser class
  - static tables for states and rules
  - semantic actions copied to a method, switch over rule no.
  - derives from `GPPG.ShiftReduceParser<S, P>`
    - S is the semantic value type, P is the position type
    - implements general parser driven by the tables