NezCC: A Cross-Language PEG Parser Generator

Kimio Kuramitsu 倉光 君郎

Since Yacc was invented in 1970s, many parser generators have been designed to generate a language-specific parser written in a parser host language such as C and Java. This is a little mysterious design choice since its grammar specification is based on a formal grammar such as LR(k), LL(k), and PEGs. We rethink a language agnostic parser generation from the beginning. First, a PEG-based grammar specification is designed in a way that any action code is unnecessary to generate a parser. Second, the parsing algorithm is simplified so that we can write it with language constructs that commonly appear in most programming languages. Finally, syntactic code template is also designed to switch a target source code. The resulting NezCC parser generator can produce a variety of target languages such as C, Java, JavaScript, Python, Lua, Scala, Racket, and F#. The NezCC parser is full-fledged, which includes flexible tree construction, extended context-sensitive parsing, and efficient packrat parsing. More interestingly, our performance study shows that NezCC is as fast as existing language-specific parser generators such as Yacc and ANTLR.

1 Introduction

Many might have asked themselves why a parser generator such as Lex/Yacc produces only C parsers despite a fact that the LALR(1) grammar is by nature independent of a particular programming language. Two reasons are considered. First, LALR(1) and other formal grammars cannot specify a full specification of programming language parsers. Indeed, the use of embedded code called *semantic actions* is very common in most parser generators. Another reason is that an efficient parser requires many language-specific coding techniques.

However the question eminds us a significant demand of parser generators: *reusability* of grammar specifications. As the parser generation depends on a target programming language, we rarely reuse grammar specifications that are found on an open source repository.

The goal of Nez parser is, say, *Write Grammar Once, Parse Text Anywhere*. A key challenge includes an elimination of action code from a grammar specification and a language abstraction of an efficient parser runtime.

To begin with, we have designed OPEG [15], an extended Parsing Expression Grammar [5], which provides a small set of declarative operators in a way that they enables major PEG's impossible features, such as flexible construction of syntax trees and context-sensitive pattern matching. Since OPEG has no longer programmed code. we can generate a parser for any target languages in theory.

The cross-language generation of a parser runtime however is a remaining challenge. Especially, a practical PEG parser is more complicated than a plain recursive descent parser. The memoization

*This is an unrefereed paper. Copyrights belong to the Author(s).

YNU, 横浜国立大学, Graduate School of Electronic and Computer Engineering, Yokohama National University, JAPAN.

technique (called packrat parsing [4]) is required to avoid the worst-case time complexity. In addition, state changes (constructed trees and other extended states) must be consistently managed when backtracking occurs [6]. As a result, the handporting of the parser runtime is prone to errors and requires many software engineering efforts. In fact, parser runtimes that we have ported by hand are limited to C, Java, and JavaScript.

NezCC is a language-agnostic parser generator that maximizes the language independence of OPEG. A key idea of the NezCC is an adaptive generation of the parser runtime itself. A target language is specified as a language profile with a set of syntactic code templates and type mappings. On the other hand, a NezCC parser runtime is coded in abstracted language constructs that are common in many programming languages. NezCC generates a parser by selecting available language constructs from a given language profile.

The following is a list of the NezCC target language, which we have developed so far.

- C, Go
- Java8, C#, Kotlin, Scala
- Python, JavaScript, Lua, PHP
- F#, Racket, Haskell

The list above shows that NezCC generator includes a broad variety of programming languages ranging from low-level system languages, dynamic languages, and functional languages. This indicates that NezCC can cover almost all practical programming languages.

The generated parsers are fast despite a fact that they are based on high-level language abstraction. In our experiments, NezCC/C parser is as fast as Yacc-generated parser, while NezCC/Java parser is faster than ANTLR's generated parsers. Since Yacc and ANTLR are widely accepted in industries, we conclude that our approach is practical in terms of parser performance.

This paper reports an experience on a language agnostic code generation throughout on the development of NezCC. Section 2 introduces OPEG and demonstrates the NezCC tool. Section 3 describes a code structure of NezCC parser and Section 4 describes a design of the language profile. Section 5 discuss performance evaluation, and experiences on language-agnostic code generators. Section 6 briefly reviews related work. Section 7 summarizes this paper.

2 A Short Tour of NezCC

This section is a brief introduction of NezCC parser generator using our implemented open source tool and their sample grammars. The tool ORIGAMI is available online at:

`http://github.com/kkuramitsu/origami`.

2.1 OPEG

OPEG is a declarative grammar specification language, originally designed in Nez parser interpreter [15]. In OPEG, a PEG is extended in a way that we enable flexible tree construction and context-sensitive parsing without any embedded action code. Table 1 shows a summary of PEG and extended OPEG operators. The semantics of OPEG is presented in [15].

OPEG is a conservative dialect of PEG. All expressive PEG operators in `?[]` are available as they are.

```
SUM = NUM '+' NUM
NUM = DIGIT+
DIGIT = [0−9]
```

PEG however is a syntactic specification of the input string and *not* a schematic specification of the output trees. OPEG provides a declarative operator of a tree construction by enclosing the content pattern with `{ ... }`. This notation is similar to the capturing `(...)` in PCRE, while the users can additionally specify a #Tag for a type of tree and

PEG	Description
' '	Matches text
[]	Matches character class
.	Any character
A	Non-terminal application
(e)	Grouping
$e?$	Option
$e*$	Zero-or-more repetitions
$e+$	One-or-more repetitions
$\&e$	And-predicate
$!e$	Negation
$e_1 e_2$	Sequencing
e_1 / e_2	Ordered Choice
AST	
{ e }	Constructor
$\$(e)$	Connector
{$ e }	Left-folding
$\#t$	Tagging
` `	Replaces a string
Symbols	
`<symbol` A`>`	Symbolize Nonterminal A
`<exists` A`>`	Exists symbols
`<exists` A x`>`	Exists x symbol
`<match` A`>`	Matches symbol
`<is` A`>`	Equals symbol
`<isa` A`>`	Contains symbol
`<block` e`>`	Nested scope for e
`<local` A e`>`	Isolated local scope for e
Conditional	
`<on` B e`>`	e on B is defined
`<on` $!B$ e`>`	e on B is undefined
`<if` B`>`	If B is defined
`<if` $!B$`>`	If B is undefined

表 1 OPEG operators

$label for a sub node.

Here is a tree annotated version of `SUM`. Note that OPEG has the Camel-style naming convention for nonterminals involving a tree construction.

```
Sum = {$left(Num) '+' $right(Num) #Add}
Num = { DIGIT+ #Num}
DIGIT = [0−9]
```

The above example shows a simple tree construction for readability. By combing the tree construction operators with repetitions and recursions, variable-length trees such as lists and nested pairs can be constructed. As a result, a flexible form of trees is specified without any action code.

In addition, OPEG provides a symbol table that manages an extended parser state that is required in some context-sensitive parsing. For example, `<symbol NAME>` ... `<match NAME>` represents a back reference that matches the second NAME exactly matches the same symbol that is matched at the first NAME. Since such symbol operators well model many patterns of context-sensitive syntax [16], OPEG can express many practical programming languages including typedef names, here documents, and indent-based code layouts.

Furthermore information on OPEG is presented in [15]. Also, the ORIGAMI project provides many grammar examples, which are available at:

`https://github.com/nez-peg/nez-grammar` †1

2.2 NezCC

NezCC parser generator is implemented as the `origami nezcc` command. Here is an example of generating a C parser with the JavaScript grammar (`-g js.opeg`).

```
$ origami nezcc −g js.opeg c.nezcc
tree: true
stateful: false
memosize: 53
funcsize: 1889
writing js.c ...
```

The `c.nezcc` is a NezCC language profile that gives a set of syntactic templates. The `js.c` file is a generated parser source, containing all necessary code such as a parser runtime and converted parse functions. Since the generated file also includes the main function as a sample parser, we can simply build the file to run:

```
$ make js CFLAGS=−O3
cc −O3 js.c −o js
```

The compiled parser takes an input text (or an input file) from the command argument and then

†1 This URL is an older version of grammar repository. The new repository is under construction right now.

output the parsed trees:

```
$ js 'var x = 1'
[#Source
  [#VarDecl
    $name=[#NameExpr 'x']
    $expr=[#IntExpr '1']
  ]
]
```

So far, what we have showed is just a C parser generator. The new feature of NezCC is that a target language can be switched by just giving another language profile. Here is another example that are generating a Python3 parser:

```
$ origami nezcc -g xml.opeg python3.nezcc
tree: true
stateful: false
memosize: 53
funcsize: 2085
writing js.py ...
$ python3 js.py 'var x = 1'
[#Source
  [#VarDecl
    $name=[#NameExpr 'x']
    $expr=[#IntExpr '1']
  ]
]
```

Figure 1 shows an excerpt of the NezCC language profile. If we simply define such a profile for a favorite language, we can obtain a full-fledge parser with the same code quality.

3 NezCC Parser

This section describes a code structure of NezCC parser. Here, we use Scala to illustrate code fragments for better presentation, since Scala is rich enough to show several features of NezCC parser code.

3.1 Parser Context

Parser context is a main data structure that maintains several states of a NezCC parser runtime. Using Scala, the parser states are represented the class `ParserContext`, as defined in Figure 2. Each of the fields represents:

- `inputs` – an input text. Note that NezCC parser supports binary data parsing, `inputs` refers to an byte (unsigned 8-bit integer) array.
- `pos` – a parser position
- `length` – the length of the input
- `headpos` – a maximum position that the parser moves forward while parsing. Note that this is

```
# Python nezcc file
extension          = py
comment            = # %s

# Type
String→ Byte[] = %s.encode('utf-8')
Byte[].get         = ord(%s[%s])
Byte[].slice       = %s[%s:%s]
Byte[]''           = '%s'
Byte[].quote       = '
Byte[].esc         = \x%02x

Array.size         = len(%s)
Array.get          = %s[%s]
Array.new          = [None] * %2$s

# Syntax
struct             = class %s :
constructor        = def __init__(self,%2$s):
init               = self.%s = %s
getter             = %s.%s
setter             = %s.%s = %s

const              = %2$s = %3$s
function           = def %2$s(%3$s):
param              = %2$s
return             = return %s

var                = %2$s = %3$s
assign             = %s = %s
if                 = if %s:
while              = while %s:
ifexpr             = (%2$s if (%1$s) else (%3$s))
lambda             = (lambda %s : %s)

and                = %s and %s
or                 = (%s) or (%s)
not                = not (%s)
true               = True
false              = False
null               = None
```

図 1 Example of Language Profile: python2.nezcc

```
class ParserContext[T](
  inputs: Array[Byte],
  length: Int,
  var pos: Int,
  var headpos: Int,
  var tree: T,
  var treeLog: Option[TreeLog[T]],
  newFunc: (String,Array[Byte],Int,Int,Int) ⇒ T,
  setFunc: (T,Int,String,T) ⇒ T,
  var state: Option[State[T]],
  var memos: Array[MemoEntry[T]]
)
```

図 2 Main Parser Context

used for syntax error reporting in PEG parsers.

- `tree` – a constructed tree, whose type is parameterized.
- `newFunc`, `setFunc` – tree constructor functions
- `treeLog` – for tree construction logs represented by a linked list of `TreeLog` data
- `state` – an extended state represented by a linked list of `State` data
- `memos` – memoization table (an array of `MemoEntry` data)

`ParserContext` contains three other sub data structures, defined as `TreeLog`, `State`, and `MemoEntry`, while it contains no specific tree data structure. A parameterized type `T` is alternatively used to represent trees. The users may externally give constructor functions ((`newFunc` and `setFunc`) to instantiate arbitrary format of trees.

3.2 OPEG Translation

In general, a code generation of a PEG parser is simply a replacement of parsing expressions with parse functions. In NezCC, an OPEG expression are replaced with a parse function `e`, which has a common interface:

```
e(px: ParserContext) : Boolean
```

where the parse function `e` takes an instance of `ParserContext` and then return the result of matching. (Note that the `true` value represents the success of matching.)

e:: a

```
px.inputs[px.pos++] == 'a'
```

e:: $e_1\ e_2$

```
e1(px) && e2(px)
```

e:: e_1/e_2

```
{val p = px.pos; e1(px)|| backtrack(px, p) && e2(px)}
```

e:: $\&e_1$

```
{val p = px.pos; e1(px) && backtrack(px, p)}
```

e:: $!e_1$

```
{val p = px.pos; !( e1(px) ||  !backtrack(px, p))}
```

e:: $\{e_1\}$

```
beginT(px,0) && e1(px) && endT(px, 0)
```

e:: $\$label(e_1)$

```
e1(px) && linkT(px, "label")
```

e:: $\#Tag$

```
tagT(px, "Tag")
```

図 3 Code Translation of PEG Operators

Figure 3 shows translation rules that compose parse expressions. In NezCC, we take a common code strategy to both imperative programming languages and functional programming languages. All functions (such as `backtrack` as depicted in Figure 4 and tree construction functions) are defined as a function returning a Boolean value, as well as the parse function `e`. As a result, all OPEG parsing expressions can be simple combined by boolean operators (`&&`, `||`, and `!`). Obviously, this leads to an expression style of code, except for local variable declarations. To handle code in a way that everything is an expression, all generated functions are sliced at the point where local variables are needed.

Based on an idea of parser combinators [11], we make a further abstraction of parser functions.

```
def backtrack(px, pos): Boolean = {
  px.pos = pos
  true
}
```

図 4 Backtrack in NezCC

That is, an operator function is in advance generated with a parameterized parse function, and the inner expression of the operator is evaluated by being passed as a high-order function. In other words, the repetition e_1* is generated by a combination of `many1` operator function and a lambda function of `e1`.

```
many1(px, (p) → e1(p));
```

In NezCC, the generation of the combinator-style code is controlled by a cost-based analysis. Thus, NezCC reduces the total size of generated code and then improves the efficiency in code cache.

3. 3 Tree Construction

In OPEG, trees are specified in a declarative annotation of parsing expressions. Tree constructions can be generated by inserting tree operation code before/after parse functions. As shown in Figure3, the tree operation code is implemented in a function returning a boolean value function, which is easier to combine other functions.

The tree construction however is not so trivial due to backtracking. Let us suppose the parse function $\{e_1\}$ fails.

```
beginT(px,0) && e1(px) && endT(px, 0)
```

The operation that is executed at `beginT(px, 0)` must be cancelled in a way that nothing happens.

In NezCC, we take an transactional approach to tree construction. That is, all operations are buffered without any execution and the execution is controlled by commit/abort operations.

The `TreeLog` class (as defined in Figure **??**) is a data structure that buffers a tree operation. A

```
class TreeLog[T](
  var op: Int,
  var pos: Int,
  var tree: T,
)
def log[T](px: ParserContext[T],
  memoPoint: Int): Int = {
  true
}
```

図 5 Logging tree operations in NezCC

linked list allows a stack-based functional data structure. As with the parser position, we can abort all operations when backtracking occurs.

```
var p = px.pos
var treeLog = px.treeLog
...
backtrack(px, p, treeLog)
```

An important exception of the tree operations is the `endT` function. Since this function is called at a safe point where the consistency of subtrees is guaranteed, we commit the transaction and then update a tree in `ParserContext`. Note that NezCC parser constructs trees in a speculative way. This is because it allows significant memory compaction on tree logs. This also means that some constructed trees may be discarded when backtracking occurs. However, the memoization (described in Section 3. 4) avoids redundant tree constructions.

A data format of tree is another important design choice. Actually, trees can be formed in a variety of data structures, which may differ from languages to languages and, even if the same language, can differ in types of parser application. NezCC is designed to customize such a various data structure by tree constructor functions in `ParserContext`:

- `newFunc(tag,inputs,spos,epos,nsubs)` - creating a new tree node with `tag` by extracting a substring from the input.
- `setFunc(parent,index,label,child)` - adding a child node into a parent node with the spec-

ified label.

Note that the `setFunc` function is called many times, while each time requires the mutation of trees. Some languages are unlike to handle such a mutation. In such cases, NezCC optionally provides immutable tree construction although tree structures are not configurable at runtime.

3.4 Memoization

Memoization is a standard technique to avoid the worst-case exponential parsing time in a backtracking parser. NezCC allows the user to switch the use of memoization when the parser is produced. Once the user chooses the use of memoization, NezCC assigned an unique number, called *memo point*, to an parsing expression to be memoized.

In general, The memo points are assigned to nonterminals in a OPEG grammar, while NezCC eliminates some of cost-ineffective nonterminals for important.

In NezCC parser, the memoization table is a 1-dimensional array of `MemoEntry` data. Let M be a set of memo points and m be a memo point such that $m \in M$. The memo entry is located by the following key index:

$$key = m + |M| \times pos$$

This memoization table however requires the linear space in the length of the input. To make the constant space consumption, NezCC relies on the sliding window idea presented in [12]. That is, the table index is recomputed by the modulo of the key index, and older entries are rewritten as the parser move forwards. As a result, NezCC can generate a simple and an efficient packrat parsing.

4 NezCC Template

This section describes the design and the implementation of NezCC generator and templates.

```
class MemoData[T](
  var result: Int,
  var pos: Int,
  var tree: T,
  var state: Option[State[T]],
)
def lookup[T](px: ParserContext[T],
  memoPoint: Int): Int = {
  val m: = px.memos(longkey(px.pos,memoPoint))
  if (m.key == key)
  { px.pos = m.pos; px.tree = m.tree; m.result } else 2
}
def store[T](px: ParserContext[T],
 memoPoint: Int, pos: Int, matched: Boolean): Boolean = {
  val m: = px.memos(longkey(px.pos,memoPoint))
  m.key = key
  m.result = (if (matched) 1 else 0)
  m.pos = (if (matched) px.pos else pos)
  m.tree = px.tree
  matched
}
```

図 6 Memoization (lookup and store functions) in NezCC

4.1 NezCC Generator

NezCC generator is a code generator specialized for parser generation. At first, an OPEG grammar file is parsed to syntax trees, which are traversed by an OPEG visitor. The core of NezCC generator is an OPEG visitor that calls code emitting APIs, which represent an abstract operation of NezCC parser. Table 2 is a list of emiting APIs.

Each of emitting APIs binds one and more templates that synthesize a target code. The suffix `?` stands for a fact that the template is optional. For example, the `emitIfExpr`() API is associated with the `ifexpr` template, which can be defined in C-like languages:

```
ifexpr = %1$s ? %2$s : %3$s
```

Note that formatters such as `%1$s` and `%s` follows the Java's formatter specification. Here, `%1$s` stands for the *first* argument of formatter parameters. The users may change the order of parameters, which can allows the Python-style if expres-

sion:

```
ifexpr = %2$s if %1$s then %3$s
```

There are many templates that are not explicitly supported in a particular programming language. We may *undefine* such code templates. Alternative, the NezCC generator tries to produce syntactic sugar code with other available templates. For example, if `ifexpr` is undefined, the `emitIfExpr`() synthesizes a lambda function that emulates an if-expression by the if statement. Accordingly, the emitting APIs are designed to take some extra parameters that enables synthesizing alternative code.

4.2 Types and Operators

In NezCC, basic types that are used in a Nez parser are identified as language independent names, and then mapped into target types. Here are basic types and type mappings in the C profile.

```
Bool            = int
Byte            = unsigned char
Int             = int
String          = char *
Array           = %s *
Tree            = void *
```

Once these basic types are set, NezCC composes necessary data types recursively. For example, the type of a byte array `Byte[]` is derived from the definition of `Byte` and `Array` above and set to be `unsigned char *`.

As described in Section 3.1, NezCC parser handles a byte array for binary parsing. In several languages, however, the implementation of byte arrays is heavy and result in poor performance. In such cases, NezCC allows a specialized type instead of the automatically derived type.

```
Byte[]          = const unsigned char*
```

Note that no types are required in a dynamic language.

4.3 Declarators

Based on basic types, NezCC synthesizes data structures, data constructors, constant values, and functions. The syntax of these code is defined with the following declarator templates:

- `struct` - complex data definition, optional
- `functype` - function type definition, optional
- `const` - constant definition, mandatory
- `prototype` - prototype declaration, optional
- `function` - function definition, mandatory

Here we focus on `struct` and `functype` to specify the necessary syntax. In c.nezcc, the struct and functype are defined as follows.

```
struct          = struct %s {
end struct      = };
field           = %s %s;
functype        = typedef %1$s (*%2$s)(%3$s);
```

Here is the produced struct by NezCC.

```
struct ParserContext {
  unsigned char * inputs;
  long length;
  long pos;
  long headpos;
  void * tree;
  struct TreeLog * treeLog;
  TreeFunc newFunc;
  TreeSetFunc setFunc;
  struct State * state;
  struct MemoEntry * memos;
};
```

Names such as `inputs` and `pos` are common across target languages, while prefixes and suffixes can be customized by `varname` template. Annotations on variables are not user-controlled, but derived from the names.

NezCC resolve the dependency of definitions and produces them in a topological order.

4.4 Controls

A PEG parser involves several control constructs (such as branches and iterations), while supports for control constructs significantly differ from languages to languages.

In NezCC, we take a minimum-common approach to handling the varieties of control constructs. That is, all controls of a parser are written in restricted constructs(i.e., function call, boolean operators, and `if−then−else` branch), which are considered to be common in almost all programming languages. In other words, NezCC can generate a parser if these three common constructs are supported in a target language.

Here are code templates that produce control constructs in C.

```
if              = if(%s) {
else if         = else if(%s) {
return          = return %s;
while           = while(%s) {
switch          = switch(%s) {
case            = case %s : %s
default         = default : %s
```

Note that there is no return statement in many functional programming languages. In other words, we have to define each function with a code fragment that can be controlled without explicit return statements. This results in many of small function definitions.

The minimum-common approach however is likely to result in poor performance. To improve the performance, we can selectively use while-looping and switch dispatching, depending on language supports in each target.

A NezCC parser relies on many recursive calls, assuming the absence of the while construct. However, thoughtless recursive calls may cause performance degradation and stack overflows in many languages. For this reason, NezCC synthesizes the while version of code only in very limited parts that are required for performance consideration. The while version of code is selected only if `while` is set.

Likewise, NezCC generates dispatching code with a single character lookahead. If `switch` and `case` are undefined, NezCC alternatively generates a function mapping from a lookahead character to lambda version of parse functions. If `lambda` is also undefined, NezCC finally generates `else if` version, which is slower than the first two versions.

4.5 Code Replacement

The syntactic templates cover the core of all NezCC parser code. However, there are several corner cases where template-based code generation is hard in practice.

- Very unique language features
- Performance (faster code is available)
- Main function (due to many language-specific code)

To integrate these corner cases, NezCC allows a direct replacement with predefined code fragments. Since the function names of a parser runtime are fixed, we can replace some with the predefined functions.

Here is an example of a function replacement in the C language. The replaced `nextbyte` function relies on a pointer-based `px->pos`, instead of `px->inputs[px->pos++]`.

```
def nextbyte = '''
static int nextbyte(struct ParserContext *px) {
  return *px→ pos++;
}
'''
```

In principle, the code replacement is quite powerful. However, the strength of NezCC templates is that we can generate a parser with the same code quality. This results in reduced costs of software testing.

5 Evaluation and Experience

In this section, we evaluate the NezCC generator in terms of language supports, code quality, and performance on generated code.

5.1 Languages

We would like to start by showing a list of programming languages that can generate a NezCC

parser without syntax and type errors[†2]. The list is in our development order, since the development of NezCC results from an iterative process of software engineering in a such way that we add new code templates for newly found unsupported language features.

- Java – original (ported from hand-written Nez parser)
- C – malloc/free, prototype declaration, array declaration
- Python – untyped code, `switch/case` alternation
- JavaScript – nothing
- Scala – `null` alternation (`Option[T]`)
- Perl – variable names(`$pos`)
- F# – 'let rec' modifier (annotation for recursive functions), explicit type conversion
- Lua – the staring index of array
- Racket – operator templates such as (+ `pos` 1), while-less version.
- Haskell – State monad, eliminating duplicated names in records, and many others.

Table **??** shows a comparison of supported constructs in each of our developed languages.

In the reminder of this subsection, we would like to mention about languages that we have not accomplished the full code generation yet.

Haskell was listed as a primary target language from the beginning of the NezCC project, but at the same time it was considered to be the most difficult one. Indeed, NezCC combinator ideas and function output units were introduced as a functional programming. Nonetheless, it took us several weeks to integrate the State monad into a NezCC code structure. The compilation has passed successfully, but we still need some rewrites to run. Compared to Haskell, F# could be ported in half a day work even if a F# beginner developed. This

†2 At this moment, bugs remain in some programming languages

```
static int e6(struct ParserContext * px) {
  return bits32(charset127,nextbyte(px));
}
```

図 7 Parse function of [0-9]+ in C

```
private static <T> boolean e6(ParserContext<T> px) {
  return many9(px,(p0) → charset127[nextbyte(p0)]);
}
```

図 8 Parse function of [0-9]+ in Java

difference mainly results from mutable data structures supported in F#.

Rust, a new system language that Mozilla project developed, is also our interesting target language but we have not generated any working code yet. The main difficulty is that Rust's memory management is integrated with Rust's type system such as ownership and lifetime concepts. This is bad compatibility with NezCC's tree construction where external constructor function. We consider that Rust 1.18 is too strict and not suitable for code generation.

5. 2 Code Adaptation

Here, we examine the code adaptation of NezCC parser generator. First, we would like to compare the syntactic variations of generated code. Figure 7, 8, 9, 10, and **??** show parse functions of the same parsing expression `'-'? [0-9]+`, where `charset127` is a boolean array encoding the `[0-9]` pattern. Note that the numbering of parse functions are sequentially automated, resulting in different numbers in each languges.

As described in 4. 4, the while support switch different versions of code. Figure 11 and 12 compares the while version (in JavaScript) and the recursive version (in Racket).

```
def e6(px):
  return many9(px,(lambda p0:charset127[nextbyte(p0)]))
```

図 9　Parse function of [0-9]+ in Python

```
func e3(px * ParserContext) (ret bool) {
    var _ret interface{}
    defer func() bool{
        ret, _ = _ret.(bool)
        return ret
    }()
  _ret = charset127[nextbyte(px)]
  return ret
}
```

図 10　Parse function of [0-9]+ in Go

```
function many7(px,f){
  var pos = px.pos
  var treeLog = px.treeLog
  var tree = px.tree
  var state = px.state
  while (f(px)){
    pos = px.pos;
    treeLog = px.treeLog;
    tree = px.tree;
    state = px.state;
  }
  return back7(px,pos,treeLog,tree,state);
}
```

図 11　While version of many7 function in JavaScript

5.3 Performance Study

We will turn to the performance evaluation of parsers that are generated by NezCC. Here, we focus on XML parsing. A reason for this is that XML parsing is easier to compare with hand-written optimized parsers, such as Xerces [27] and libxml2 [17]. Another reason is that the XML grammar is simple and easy to port into the existing parser generators. In this experiment, we compare NezCC generator with Yacc(Bison [3]), ANTLR3/4, and PegJS [18].

Figure 13 shows the parsing time of XML data in each of tested parsers. We have parsed an XMark 10 MB file [25] and elapsed times are measured in millisecond. Tested parsers are labeled by a tool/language pair. We run the same test sets several time and the average time is plotted on the graph. All tests are measured on DELL XPS-8700 with 3.4GHz Intel Core i7-4700, 8GB of DDR3 RAM, and running on Linux Ubuntu 14.04.3 LTS.

```
(define (many7 px f)
  (define pos (get-field _pos px))
  (define treeLog (get-field _treeLog px))
  (define tree (get-field _tree px))
  (define state (get-field _state px))
  (if (f px) (begin (many7 px f))
    (begin (back7 px pos treeLog tree state)))
)
```

図 12　Recursion version of many7 function in Racket

The NezCC/C parser (`nezcc/c`) indicates the fastest parsing time in all the tested sets, including handwritten XML parsers and yacc-generated parser. This suggests that the C compiler (GCC4.5) can well handle a functional code style of NezCC. In the comparison of the Java parsers, we confirm that NezCC/java shows not the best but competitive performance to the hand-written Xerce/java, and ANTLR3/4-generated parsers. In JavaScript comparisons, NezCC/JS/ is also two time as fast as PegJS/JS. As long as we have tested, NezCC generates a very efficient parser code in every language.

Table 13 lists the parsing time of other parsers that are faster than the PegJS/JS parser. The slower parsers that are not listed are NezCC/Lua5 in 7.2 [s], NezCC/Haskell in 8.2 [s], NezCC/Python2 in 9.7[s], NezCC/Racket in 60[m], and NezCC/PHP in 74[m]. The reader might consider that very slow parsers seem useless in practical. However, many applications need to parse not so large but complex input data. The strength of

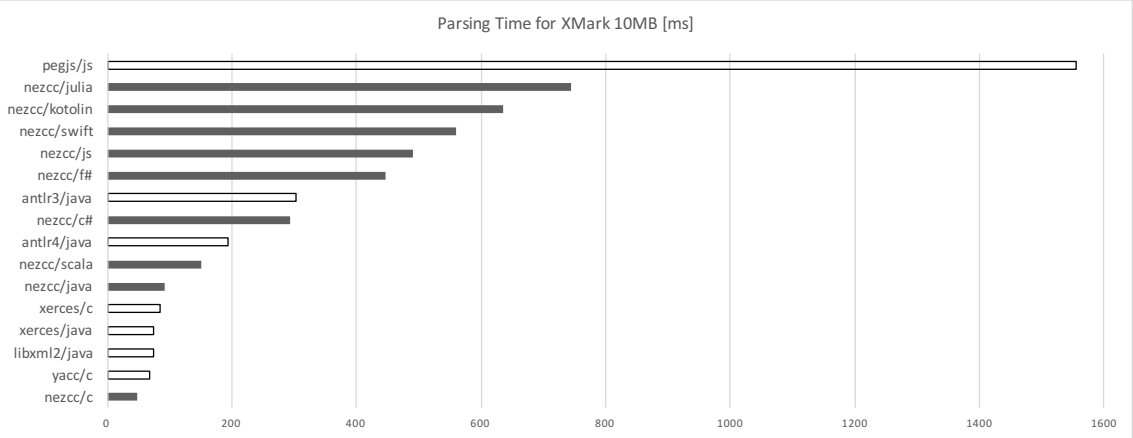

図 13　Performance comparions: XMark 10MB

NezCC is that parsing is all the same way in any language environments.

5.4 Experience

Finally, we summarize our experiences on cross-language techniques that we have learnt through our development of NezCC. Here, we start with two assumptions that we set before the development of NezCC.

Assumption 1: *If we generate a parser code based on only common language features of C and Java, it would be easy to map to other languages.*

This is assumed simply because many existing languages are influenced from C and Java. But, what we have experienced is the complexity of type mapping. In particular, richly-typed languages are likely to be more difficult. For example, many modern languages provide some static null check support. This requires us adding the `Option`-like type (`Maybe` in Haskell) template after all. However, the `Option`-like type has little common operators across languages, making it difficult to design code template. (As a result, NezCC generates very miserable code).

Assumption 2: *Software test is unnecessary since a generated parser is tested with strongly-typed code such as Scala and F#.*

We prepared a test grammar (named `testcase.opeg`), while most of a newly generated parser passed without serious failures. Compared to a hand-written parser generator, the quality of initial code is high and software testing seems unnecessary. Static type checkers are always useful for finding mistakes in code templates, while dynamic languages require print debugging at the runtime. The most serious bug was caused by an integer overflow when parsing a very large input.

In origin, NezCC is designed on an assumption that all programming languages support, at least, function call, boolean operators, and if-expression (ternary operator). However, the lack of our knowledge is gradually clear since Lua and Go language have no if-expression support. Thus, there are no minimal common language features that allow parsing.

This fact also led to a crisis of NezCC project, since the code structure of NezCC relies largely on a functional style. What saved us was a lambda expression, which allows us to map imperative code into an expression. In the end, the lambda support is very important in developing a cross-language code generation.

6 Related Work

Parser generator is a typical scenario of generative programming, and has a long history in the field of programming languages. Lex/Yacc [8] is the first modern parser generator, which has invented semantic actions embedded on a formal grammar specification. Since the semantic action hits a sweet spot [7] between the grammar expressiveness and the practical performance, most of parser generators inherit this idea from Yacc. However, semantic action code significantly reduces the reusability of grammar specifications, as many researchers [21][10][15] have been pointed out.

Generalization of semantic action code is the first step attempt toward a language agnostic generator. In this context, attribute grammars [20] or abstracted operations [10][2][9][1][26] have been addressed instead of semantic action code. OPEG in NezCC has been developed to eliminate arbitrary action code [15]. Even if a grammar is purely declarative, the porting problem remains.

As of 2017, there are more than 140 kinds of parser generator implementations on the Wikipedia's comparison list[†3]. 26 of the listed parser generators produces two or more target languages. Notably, ANTLR 3/4 [22][23] provides an abstract and standardized APIs between the ANTLR parser runtime. Many parser runtimes have been ported so far, while they relies on a significant amount of software engineering efforts and strong community supports. To my knowledge, NezCC is the first template-based parser generator, which enables a wide support of target languages at a very short period of the development time.

Last not but least, NezCC totally relies on the simple and efficient parser runtime, which results from many previous research efforts in the context of PEGs and PEG parsers [5][4]. Notably, we base the partial DFA conversion of PEGs [19], a transactional state management (tree construction) [6][14], and cache-based efficient packrat parsing [24][12].

7 Conclusion

We have addressed a language agnostic generation of a full-edged parser, including flexible tree construction, extended context-sensitive parsing, and efficient packrat parsing. The NezCC generator is based on two levels of abstraction: a declarative modeling of extended PEG-based parsing and a language abstraction of an efficient parser runtime. In this paper, we have focused mainly on the latter; a set of syntactic code templates and type mappings allows us to generate parser code of very various target languages, including C, Java, JavaScript, Python, Lua, Scala, Racket, and F#.

Throughout this study, we have found that the parser generation is a good subject of cross-language code generation, which would be increasingly demanded in the current development of software. In that respect, many interesting challenges remained, including cross-language optimization, and code reliability and verification across languages. We hope that this research will lead to furthermore generalized cross-language transpliation techniques.

†3 https://en.wikipedia.org/wiki/Comparison_of_parser_generators

参 考 文 献

[1] Adams, M. D.: Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, ACM, 2013, pp. 511–522.

[2] Atkey, R.: The Semantics of Parsing with Semantic Actions, *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, Washington, DC, USA, IEEE Computer Society, 2012, pp. 75–84.

[3] Donnelly, C. and Stallman, R.: Bison. The YACC-compatible Parser Generator, (2004).

[4] Ford, B.: Packrat Parsing:: Simple, Powerful,

Lazy, Linear Time, Functional Pearl, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, New York, NY, USA, ACM, 2002, pp. 36–47.

[5] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, 2004, pp. 111–122.

[6] Grimm, R.: Better Extensibility Through Modular Syntax, *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, New York, NY, USA, ACM, 2006, pp. 38–51.

[7] Jim, T., Mandelbaum, Y., and Walker, D.: Semantics and Algorithms for Data-dependent Grammars, *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, New York, NY, USA, ACM, 2010, pp. 417–430.

[8] Johnson, S. C.: *Yacc: Yet another compiler-compiler*, Vol. 32, ell Laboratories Murray Hill, NJ, 1975.

[9] Johnstone, A. and Scott, E.: Tear-Insert-Fold Grammars, *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, New York, NY, USA, ACM, 2010, pp. 6:1–6:8.

[10] Kats, L. C., Visser, E., and Wachsmuth, G.: Pure and Declarative Syntax Definition: Paradise Lost and Regained, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, ACM, 2010, pp. 918–932.

[11] Koopman, P. W. M. and Plasmeijer, M. J.: Efficient Combinator Parsers, *Selected Papers from the 10th International Workshop on 10th International Workshop*, IFL '98, London, UK, UK, Springer-Verlag, 1999, pp. 120–136.

[12] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window, *Journal of Infomration Processing*, Vol. 23, No. 4(2015), pp. 505–512.

[13] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window, *Journal of Infomration Processing*, Vol. 23, No. 4(2015), pp. 505–512.

[14] Kuramitsu, K.: Fast, Flexible, and Declarative Consturction of Abstract Syntax Trees with PEGs, *Journal of Infomration Processing*, Vol. 24, No. 1(2016), pp. 256264.

[15] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, ACM, 2016, pp. 29–42.

[16] Kuramitsu, K.: A Symbol-Based Extension of Parsing Expression Grammars and Context-Sensitive Packrat Parsing, *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, 2017.

[17] Libxml2: Validation and DTDs, http://xmlsoft.org/xmldtd.html, 2015.

[18] Majda, D.: PEG.js - Parser Generator for JavaScript, 2015.

[19] Nariyoshi, C. and Kuramitsu, K.: Linear Parsing Expression Grammar, *Proceedings of 11th International Conference on Language and Automata Theory and Applications*, LATA 2017, New York, NY, USA, 2017, pp. (to appear).

[20] Paakki, J.: Attribute Grammar Paradigms&Mdash;a High-level Methodology in Language Implementation, *ACM Comput. Surv.*, Vol. 27, No. 2(1995), pp. 196–255.

[21] Parr, T.: The Reuse of Grammars with Embedded Semantic Actions, *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, Washington, DC, USA, IEEE Computer Society, 2008, pp. 5–10.

[22] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, New York, NY, USA, ACM, 2011, pp. 425–436.

[23] Parr, T., Harwell, S., and Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, New York, NY, USA, ACM, 2014, pp. 579–598.

[24] Redziejowski, R. R.: Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking, *Fundam. Inf.*, Vol. 79, No. 3-4(2007), pp. 513–524.

[25] Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R.: XMark: A Benchmark for XML Data Management, *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, VLDB Endowment, 2002, pp. 974–985.

[26] Thielecke, H.: On the Semantics of Parsing Actions, *Sci. Comput. Program.*, Vol. 84(2014), pp. 52–76.

[27] Xerces: Apache Xerces Project, http://xerces.apache.org/, 2015.

表 2　List of Emitting APIs

Declarators	Templates
`declStruct(name, fields)`	`struct`
`declFuncType(ret, name, params)`	`functype?`
`declConst(type, name, size, data)`	`const,const_array?`
`declProtoType(ret, name, params)`	`prototype?`
`declFunc(acc,ret,name,params,block)`	`function`
Statements	
`beginBlock()`	`begin?`
`emitStmt(block, expr)`	`semicolon;`
`endBlock(block)`	`end?`
`emitVarDecl(mut,name,expr,expr2)`	`letin?,var?`
`emitIfStmt(expr, elseIf, stmt)`	`if,else if?`
`emitWhileStmt(expr, stmt)`	`while?`
Expressions	
`emitAssign(name, expr)`	`assign?`
`emitReturn(expr)`	`return?`
`emitOp(expr, op, expr2)`	`+,-,*,%,eq,ne`
`emitConv(var, expr)`	`T->T`
`emitNull(name,)`	`null?`
`emitArrayIndex(self, index)`	`Array.get`
`emitNewArray(type, index)`	`Array.new`
`emitGetter(self, name)`	`getter`
`emitSetter(self, name, expr)`	`setter`
`emitNew(type, params)`	`object?`
`emitFunc(func, params)`	`funccall`
`emitApply(func, params)`	`apply`
`emitNot(expr)`	`not`
`emitSucc()`	`true`
`emitFail()`	`false`
`emitAnd(expr, expr2)`	`and`
`emitOr(expr, expr2)`	`or`
`emitIfExpr(expr, type, expr2, expr3)`	`ifexpr?`
`emitFuncRef(name)`	`funcref?`
`emitLambdaExpr(params, body)`	`lambda?`
`emitDispatch(index, cases)`	`switch?`

表 3　Comparison of Supported Constructors in Languages

Language	constructor	lambda	ifexpr	while	switch	null	mutable
C	malloc	x	o	o	o	o	o
C#	new	o	o	o	o	o	o
F#	record	o	o	o	match	x	o
Go	new	o	x	o	o	o	o
Haskell	record	o	o	x	match	x	x
Java8	new	o	o	o	o	o	o
JavaScript	function	o	o	o	o	o	o
Lua	record	o	x	o	x	o	o
Perl	record	o	o	o	x	x	o
PHP5/7	new	o	o	o	o	o	o
Python2/3	new	o	o	o	x	o	o
Racket	record	o	o	x	match	o	o
Scala	new	o	o	o	match	x	o
Swift	record	o	o	o	o	x	o