Journal of Information Processing Vol.24 No.2 256–264 (Mar. 2016)

[DOI: 10.2197/ipsjjip.24.256]

Regular Paper

A Declarative Extension of Parsing Expression Grammars for Recognizing Most Programming Languages

Tetsuro Matsumura[1,†1] Kimio Kuramitsu[1,a)]

Received: May 18, 2015, Accepted: November 5, 2015

Abstract: Parsing Expression Grammars are a popular foundation for describing syntax. Unfortunately, several syntax of programming languages are still hard to recognize with pure PEGs. Notorious cases appears: typedef-defined names in C/C++, indentation-based code layout in Python, and HERE document in many scripting languages. To recognize such PEG-hard syntax, we have addressed a declarative extension to PEGs. The "declarative" extension means no programmed semantic actions, which are traditionally used to realize the extended parsing behavior. Nez is our extended PEG language, including symbol tables and conditional parsing. This paper demonstrates that the use of Nez Extensions can realize many practical programming languages, such as C, C#, Ruby, and Python, which involve PEG-hard syntax.

Keywords: parsing expression grammars, semantic actions, context-sensitive syntax, and case studies on programming languages

1. Introduction

Parsing Expression Grammars [5], or PEGs, are a popular foundation for describing programming language syntax [6], [15]. Indeed, the formalism of PEGs has many desirable properties, including deterministic behaviors, unlimited look-aheads, and integrated lexical analysis known as scanner-less parsing. Due to these properties, PEGs allow grammar developers to avoid the *dangling if-else* problem and to express the *contextual tokens* and the *nested block comment*, which are known problems in traditional LR(k) and LL(k) grammars.

Despite the powerful features of PEGs, several *real* syntax used in popular programming languages is hard to express. This problem comes chiefly from context-sensitive syntax, whose meanings are changed depending on the parsing context. Typical examples of such syntax are:

- Typedef-defined name in C/C++ [5], [6]
- HERE document appearing in Perl, Ruby, and other many scripting languages
- Indentation-based code layout appearing in Python and Haskell [1]
- Contextual keywords used in C# and other evolving languages [3]

Technically, a language grammar involving such context-sensitive syntax is implemented with semantic actions [8], an embedded code that is hocked to execute the extended action at the parsing time. Since semantic actions are written in a host language of the generated parser, the use of semantic actions may invalidate the declarative property of PEGs, thus resulting in reduced reusability of grammars. As a result, many developers need to redevelop grammars for their software engineering tools.

In this paper, we propose a declarative extension of PEGs for recognizing context-sensitive syntax. The "declarative" extension means no arbitrary semantic actions that are written in a general purpose programming language. In our proposal, a variety of semantic actions are abstracted to two modeled states:

- *Symbol Table* – a table that manages a list of symbols that are differently treaded in a specific context
- *Parsing Condition* – a Boolean flag that dynamically switches the parsing behaviors.

Nez is an extended PEG language that is designed to provide additional operators to handle the above states in parsing contexts of PEGs. In this paper, we call *Nez extensions* to distinguish from *pure* PEGs operators.

Using Nez, we have performed extensive case studies by specifying various popular programming languages. The grammars that we have developed for now include C, Java, C#, JavaScript, Lua, Ruby, and Python. Since these grammars include many context-sensitive syntax, we conclude that Nez extensions provide improved expressiveness for programming languages in practical manners.

The remainder of this paper proceeds as follows. Section 2 describes PEG-hard syntax patterns. Section 3 presents a language design for Nez. Section 4 demonstrates case studies with our emphasis on Nez extensions. Section 5 briefly reviews related work. Section 6 concludes the paper. The tools and grammars that are presented in this paper are available online at http://nez-peg.github.io/.

[1] Graduate School of Electronic and Computer Engineering, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[†1] Presently with Zuken Inc.
[a)] kimio@ynu.ac.jp

© 2016 Information Processing Society of Japan

2. Background and Problem Statement

2.1 PEGs

PEGs are a recognition-based foundation for describing syntax, formalized by Ford [5]. While PEGs are relatively new, most of their notations are familiar, coming from EBNF (e.g., productions and recursive nonterminals) and regular expressions (*kleene* operators such as `?` and `*`). **Figure 1** shows a PEG, which expresses the basic mathematical notations.

The interpretation of PEGs significantly differs from CFGs in that PEG's choice is ordered. That is, the first subexpression of a choice is always matched first, and the next subexpression is attempted only if the first fails. This brings us to deterministic parsing behavior, which is regarded as desirable for parsing non-natural languages. The ordered choice, on the contrary, disallows the *left recursion*, because the deterministic interpretation of the left recursion results in unlimited looping. The PEG example in Fig. 1 is defined in a form of eliminating left recursions.

The expressiveness of PEGs is almost similar to that of *deterministic* CFGs (such as LALR and LL family). In general, PEGs can express all LR grammar languages, which are widely used in a standard parser generator such as Lex/Yacc. In addition, PEG's syntactic predicates (& and !) provide us with the expressiveness of unlimited look-aheads, suggesting that PEGs are more expressive than CFGs. As a result, PEGs can recognize non context free languages such as $\{a_n\ b_n\ c_n \mid n > 0\}$.

2.2 Context-Sensitive Syntax

PEGs are very powerful, but, in practice, are not able to express all programming language syntax. This mostly comes from context-sensitive syntax, where the meaning of symbols may vary in different contexts. Note that the same limitation commonly exists in CFG-based grammars. This subsection demonstrates typical examples of context-sensitive syntax.

2.2.1 Typedef-defined Name (in C/C++)

Typedef-defined name in C/C++ is a typical example of context-sensitive syntax. The identifier is simply supposed as a sequence of word characters; for example, `T` is a legal identifier in C/C++. On the other hand, the following `typedef` statement allows the users to declare `T` as a new type name.

```
typedef unsigned int T;
```

Once `T` is a declared type, `T` is not regarded as an identifier. In general, the interpretation of an identifier class should be performed in the phase of semantic analysis, not in syntactic analysis. However, in C/C++, we need to produce a different syntax tree depending on the contextually varying type of `T`. That is, an expression `(T)-1` can be parsed differently:

- a subtract operator of a variable T and the number 1, or
- a type cast of the number 1 to the type `T`

Note that Java avoids this problem by a careful design of language syntax. The casting `(T)-1` is disallowed by restricting that the unary + and - is only available for primitive number types.

```
Expr      = Sum
Sum       = Product (( '+'  / '-' ) Product )*
Product   = Value (( '*' / '/' ) Value )*
Value     = [0-9]+ / '(' Expr ')'
```

Fig. 1 Mathematical Operators in a PEG.

2.2.2 HERE Document (in Ruby, Bash, etc.)

The *HERE document* is a string literal for multiple lines, widely adopted in scripting languages, such as Bash, Perl, and Ruby. While there are many variations, the users are in common allowed to define a delimiting identifier that stands for an end of lines. In the following, the symbol `END` is an example of the user-defined delimiting identifier.

```
print <<END
  the string
  next line
END
```

In PEGs, the syntax of a HERE document can be specified with multi-lines that continue until the nonterminal `DELIM` is matched in the head of a line. However, we cannot assume specific keywords or possible identifiers to define `DELIM`, since the user-defined delimiting identifiers are totally unpredictable.

2.2.3 Indentation-based Layout (in Python, Haskell, etc.)

Indentation-based code layout is a popular syntax that uses the depth of indentations for representing the beginning and the end of a code block. Typically, Python and Haskell are well known for such indentation-based code layout, a large number of other languages including YAML, F#, and Markdown also use indentation.

```
mapAccumR f = loop
     where loop acc (x:xs) = (acc, x : xs)
              where (acc, x) = f acc x
                    (acc, xs) = loop acc xs
           loop acc [] = (acc, [])
```

The problem with PEGs is that the user is allowed to use arbitrary indentation for their code layout. As with in delimiting identifiers in the HERE documents, it is hard to prepare all possible indentations for code layout.

2.2.4 Contextual Keyword in C#

Popular programming languages are long standing, and then evolve to meet the user's demands. In the language evolution, adding a new keyword is often considered to identify a new syntactical structure. On the other hand, a backward compatibility problem inevitably arises since the added keyword might have already been used as identifiers in legacy code.

A *contextual keyword* is used to avoid the compatibility problem. For example, C#5.0 newly added the `await` keyword, only available in the `async` qualified method.

```
async Task<string> GetPageAsync(string path)
{
  HttpClient c = new HttpClient();
  return await c.GetStringAsync(uri + path);
}
```

As with in typedef names, the different meaning of `await` needs to produce different syntax trees in the phase of syntactic analysis. It is also hard to specify different syntax analysis

© 2016 Information Processing Society of Japan

depending on given contexts.

2.3 Semantic Actions

Semantic actions are a programmed code embedded in grammars definition, which is hocked to perform extra processing in parsing contexts. The following is a Rats!'s grammar fragment that shows an example of semantic actions `&{ ... }` [6]. The parse result of the nonterminal `Identifier` is assigned to a variable `id` in the semantic action and checked by the method `isType()` of a global state `yyState`.

```
TypedefName = id:Identifier &{
     yyState.isType(toText(id))
}
```

As shown above, semantic actions can use a general-purpose programming language, leading to richer operations including AST construction. As a result, semantic actions are most commonly used in today's parser generators to extend the expressiveness of grammars.

An obvious problem with semantic actions is that the grammar definition depends tightly on a parser implementation language, and lacks the declarative properties of grammar definitions. This is really unfortunate, because well-defined grammars are potentially available across many parser applications such as IDEs and other software engineering tools.

3. Language Design of Nez

Nez is a PEG-based grammar specification language that provides pure and declarative notations for AST constructions and enhanced matching for context-sensitive syntax. In this section, we focus on Nez extensions [*1] for enhanced matching of context-sensitive syntax.

3.1 Overview

Nez is a PEG-based language that provides declarative notations for describing syntax without *ad hoc* semantic actions. The extensions range from AST constructions to enhanced matching. **Figure 2** shows an abstract syntax of Nez language.

Nez operators are categorized as follows:

- *PEG Operators* – matching operators based on PEGs
- *AST Constructions* – manipulating abstract syntax tree representations with parsed results
- *Symbol Tables Handlers* – handling the global state, called symbol tables, in parsing contexts.
- *Parsing Conditions* – switching parser behavior depending on a given conditions.

We have designed the capability of AST constructions in a decoupled way from any matching capability. That is, they are orthogonal to each other; any constructed ASTs do not influence matching results, while any matching results can be incorporated to tree manipulations. In this paper, we highlight the extended matching capability with symbol table handlers and parsing conditions, due to the space constraints. Further information on the AST construction can be referred to our report [11].

e	::=	ϵ	: empty
	|	A	: non-terminal
	|	a	: terminal character
	|	$e\ e'$	: sequence—
	|	$e\ /\ e'$	: prioritized choice
	|	$e?$	: option
	|	$e*$	: repetition
	|	$\&e$	: and predicate
	|	$!e$	: not predicate
	|	$\{e\}$	: AST constructor
	|	$\$(e)$	: AST connector
	|	`#x`	: AST tagging
	|	`<def` $T\ e$ `>`	: symbol definition
	|	`<exists` T `>`	: symbol existence
	|	`<match` T `>`	: symbol match
	|	`<is` T `>`	: symbol equivalence
	|	`<isa` T `>`	: symbol containment
	|	`<block` $T\ e$ `>`	: nested table scoping
	|	`<local` $T\ e$ `>`	: isolated table scoping
	|	`<if` C `>`	: condition testing
	|	`<on` $C\ e$ `>`	: evaluation on the condition C

Fig. 2 An abstract syntax of Nez language.

3.2 Symbol Tables

Symbol table is a global state used to maintain strings, whose meaning is specialized in parsing contexts. We call such strings *symbols* in this paper. Nez supports multiple symbol tables in a grammar. Let T be a table identifier to distinguish a symbol table from others.

Nez newly defines the following operations on the symbol table T:

- `<def` $T\ e$`>` – symbol definition by extracting a string matched by the subexpression e, and then store it as a symbol to the table T
- `<match` T`>` – match the latest-defined symbols in the table T
- `<is` T`>` – equals the latest-defined symbol in the table T
- `<isa` T`>` – contains one of stored symbols in the table T
- `<exists` T`>` – testing the existence of stored symbols in the T table
- `<local` $T\ e$`>` – isolated local scope of T for the subexpression e
- `<block` $T\ e$`>` – nested local scope of T for the subexpression e

Let us show how a symbol table works with parsing expressions. To start, we consider the absence of any symbol tables. The production XML is intended to accept `<tag> .. </tag>`.

```
XML  = '<' NAME '>' XML? '</' NAME '>'
NAME = [A-z] [A-z0-9]*
```

An obvious problem is that the parsed closing tag can be different from the parsed opening tag because the production `NAME` matches arbitrary names. That is, the production `XML` accepts an illegal input such like `<a></b>`. Since a set of possible names are infinite, it is hard for pure PEGs to ensure the equivalence of opening tags and closing tags.

Now we consider the introduction of a symbol table, named TAG, to maintain names parsed at the opening tag.

[*1] We have planned to revise the Nez operators to improve the usability. The `def` operator was obsolete and replaced with the similar `symbol` operator. The replaced operator ensures the same expressiveness with the `def` operator.

© 2016 Information Processing Society of Japan

```
XML  = '<' <def TAG NAME> '>' XML? '</' <is TAG> '>'
NAME = [A-z] [A-z0-9]*
```

Due to the stored opening tag, we can check the equivalence of the closing tag. However, <is TAG> only accepts the latest-defined symbol in TAG. That is, nested tags, such as <a></a>, are still unacceptable. If we replace <is TAG> with <isa TAG>, then we can match either 'a' or 'b' but the proper order is not guaranteed. (That is, <isa TAG> is a bad example.)

The constructor <block TAG *e*> is introduced to declare a nested local scope of symbol tables. The scope means that all symbols that defined in the subexpression *e* are only available in the context of evaluating *e*. In other words, any symbols defined in *e* are not available outside the block constructor. The following is a scoped-modification to match nested tags correctly.

```
INNER = <block TAG XML>
XML = '<' <def TAG NAME> '>' INNER? '</' <is TAG> '>'
NAME = [A-z] [A-z0-9]*
```

Nez provides another scoping notation, <local *T e*>. The difference is that the local creates an isolated scope for a specific table *T*. The isolated scoping means that symbols defined outside are not referred to from the inside scope. In the above, the replacement of the block scope with <local TAG XML> produce the same matching result.

3.3 Parsing Condition

The idea of parsing condition comes from both *conditional compilation* and *semantic predicates*. The parsing conditions are similar to directives in conditional compilation and switch the parsing behavior by whether the parsing flag is true or not.

Nez supports multiple parsing conditions, identified by the user-specified flag names. Let *C* be a parsing flag name. The notation <if *C*> is a determination of the parser behavior depending on whether *C* is true or not. That is, an parsing expression <if *C*> *e* means that the expression *e* is attempted only if *C* is *true*. The syntactic predicate ! is allowed for a parsing flag, which is simply a negation of *C*. That is, <if !*C*> *e* means that the expression *e* is attempted only if the parsing flag *C* is *false*. The expressions e_1 and e_2 can be distinctly switched with a choice by <if *C*> e_1 / <if !*C*> *e*2.

Here is an example of a parsing condition NL that switches the inclusion of new lines in white spaces. The production WS is regarded as [\t\n] if NL is true, and [\t] if not.

```
WS  = [ \t] / <if NL> [\n]
```

The parsing conditions are not static, rather we switch them on/off in parsing contexts. Switching conditions is only controlled by the following two constructors:

- <on *C e*> – the expression *e* is evaluated under the condition that C is true
- <on !*C e*> – the expression *e* is evaluated under the condition that C is false

Note that <on *C e*> and <on !*C e*> are not an action to be performed, but a condition declaration to be satisfied for the subexpression *e*. We allow nested declarations. If undeclared, conditions are regarded as true by default.

```
<on NL WS> // on
<on !NL WS> // off
<on NL <on !NL WS>>  // nested
```

Note that the parsing condition is a global state and sharable across productions. In the above example, the condition NL used in the production WS is switched from arbitrary contexts of <on NL *e* > whose *e* involves the nonterminal WS.

3.4 Semantics

The semantics of the Nez extensions is built on the formal semantic of PEGs, presented in Ref. [5]. Formally, a Parsing Expression Grammar, G, is a 4-tuple $G = (V_N, V_T, R, e_s)$, where V_N is a finite set of nonterminal, V_T is a finite set of terminal, R is a finite set of rules, e_s is a start expression. Each rule, $A = e$, is a mapping from nonterminal A to parsing expression e. This mapping is written as $R(A)$.

Let x, y, z, w be a string. The symbol table T is defined as a recursive ordered pair of string. The qualifier "recursive" means that a list (x, y, z) equals to a nested pair $(x, (y, z))$. () stands for an empty list. The pair (w, T) stands for a new pair where a string w is added to T. The function $top(T)$ is defined as $top(T) = x$

$$
\begin{aligned}
\text{Empty} &: (\epsilon, x, T) \Rightarrow (1, x, T) \\
\text{Terminal} &: (a, ax, T) \Rightarrow (1, x, T) \\
\text{NonTerminal} &: (A, xy, T) \Rightarrow (n+1, y, T') \ \textbf{if}\ (R(A), xy, T) \Rightarrow (n, y, T') \\
\text{Sequence} &: (e_1\ e_2, xyz, T) \Rightarrow (n_1 + n_2 + 1, z, T'') \\
&\quad \textbf{if}\ (e_1, xy, T) \Rightarrow (n_1, y, T')\ \textbf{and}\ (e_2, yz, T') \Rightarrow (n_2, z, T'') \\
\text{Choice} &: (e_1/e_2, xy, T) \Rightarrow (n_1 + 1, y, T') \\
&\quad \textbf{if}\ (e_1, xy, T) \Rightarrow (n_1, y, T') \\
\text{Choice(2)} &: (e_1/e_2, xy, T) \Rightarrow (n_2 + 1, y, T') \\
&\quad \textbf{if}\ (e_1, xy, T) \Rightarrow (n_1, \bullet, T')\ \textbf{and}\ (e_2, xy, T) \Rightarrow (n_2, y, T') \\
\text{Repetition} &: (e*, xyz, T) \Rightarrow (n_1 + n_2 + 1, y, T'') \\
&\quad \textbf{if}\ (e_1, xyz, T) \Rightarrow (n+1, yz, T')\ \textbf{and}\ (e*, yz, T') \Rightarrow (n_2, z, T'') \\
\text{Not} &: (!e, x, T) \Rightarrow (n+1, x, T')\ \textbf{if}\ (e, x, T) \Rightarrow (n, \bullet, T') \\
\text{Def} &: (\langle def\ T\ e\rangle, xy, T) \Rightarrow (n+1, y, (x, T)) \\
&\quad \textbf{if}\ (e, xy, T) \Rightarrow (n, y, T) \\
\text{Block} &: (\langle block\ T\ e\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (e, xy, T) \Rightarrow (n, y, T') \\
\text{Local} &: (\langle local\ T\ e\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (e, xy, ()) \Rightarrow (n, y, T') \\
\text{Exists} &: (\langle exists\ T\rangle, x, T) \Rightarrow (n+1, x, T)\ \textbf{if}\ \exists w\ \textbf{such that}\ w \in T \\
\text{Match} &: (\langle match\ T\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (x, xy, T) \Rightarrow (n, y, T)\ \textbf{and}\ x = top(T) \\
\text{Is} &: (\langle is\ T\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (e^T, xy, T) \Rightarrow (n, y, T)\ \textbf{and}\ x = top(T) \\
\text{Isa} &: (\langle isa\ T\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (e^T, xy, T) \Rightarrow (n, y, T)\ \textbf{and}\ x \in T \\
\text{If} &: (\langle if\ C\rangle, x, T) \Rightarrow (n+1, x, T)\ \textbf{and}\ C \in T \\
\text{On} &: (\langle on\ C\ e\rangle, xy, T) \Rightarrow (n+1, y, T) \\
&\quad \textbf{if}\ (e, xy, (C, T)) \Rightarrow (n, y, T')
\end{aligned}
$$

Fig. 3 Semantics.

© 2016 Information Processing Society of Japan

where $T = (x, y, z, \ldots)$.

Let xy be a concatenation of x and y. We assume the single table case without the loss of generality. The semantics of PEGs with T is given as a relation $(e, xy, T) \Rightarrow (n, y, T')$, where e is a parsing expression, xy is an input string, n is a step counter, and y is an unconsumed string. That is, the expression e consumes the string x. If the expression fails, we write $\bullet$ for the remaining string. The T and T' represent a certain status of the symbol table, which may be different.

The relation $\Rightarrow$ is defined inductively as shown in **Fig. 3**. As in Ref. [5], we use the abstract form of PEG syntax by omitting syntax sugars. In addition, we regard the condition name C as a string that can be stored in the table T.

In Nez, we presume that strings in the table T are all extracted from the same expression. That is, two different table definitions (<`def` T e> and <`def` T e'> such that $e \neq e'$) are not allowed. We write e^T for representing an expression that is defined in <`def` T e>. More importantly, symbol matching <`is` T> and symbol containment <`isa` T> are based on the sub-string extraction by e^T in order to avoid unintended substring matching.

4. Case Studies and Experiences

We have developed many grammars ranging from programming languages to data formats. All developed grammars are available online at http://nez-peg.github.io/. This section reports our experiences throughout grammar developments with Nez.

4.1 Summary

To evaluate the language design of Nez, we have performed extensive case studies by developing grammars for major programming languages. Our examined grammars are developed by the following approaches:

- C – Based on two PEG grammars written in Mouse and Rats!. Semantic actions for handling the `typedef` statement are ported into Nez's symbol table.
- Java8 – Ported from Java8 grammar written in ANTLR4 [*2]
- JavaScript – Based on JavaScript grammar for PEG.js [*3].
- C# – Developed from the scratch, referencing C#5.0 Language Specification (written in a natural language)
- Lua – Developed from the scratch, referencing Lua 5.1 Reference Manual.
- Ruby – Developed from the scratch and in part referred as the yacc grammar for CRuby.
- Parser – Ported from Python 2.7 abstract grammar [*4]
- Konoha –Developed from the scratch, based on Konoha paper [9]
- MinCaml –Developed from the scratch, based on MinCaml paper [16]
- Haskell – Developed from the scratch.

Table 1 shows a list of developed grammars.The column labeled "#" indicates the number of production rules, implying the complexity of grammars. Table 1 confirms a substantial trend in the expressiveness of PEGs, while our developed grammars are in part incomplete. Only two languages (including Java8 and MinCaml) can be fully specified with *pure* PEGs. This suggests that most of programming languages require semantic actions or equivalent extensions such as the Nez extensions in order to parse with PEGs.

[*2] https://github.com/antlr/grammars-v4/blob/master/java8/Java8.g4
[*3] https://github.com/pegjs/pegjs/blob/master/examples/javascript.pegjs
[*4] https://docs.python.org/2.7/library/ast.html

Table 1 List of developed grammars for programming languages.

Language	# of Rules	Nez ext.
C	101	<def> <isa>
C#5.0	454	<if> <on>
Haskell98	110	<def> <is> <block> <if>
Java8	160	
JavaScript	132	<if> <on>
Konoha	124	<if> <on>
MinCaml	65	
Lua	96	<def> <is> <block> <if>
Python	55	<def> <match> <block> <if>
Ruby	200	<def> <is> <block> <on> <if>

```
W  = [A-ZA-z_0-9]
S  = [ \t\r\n]

TypeDef
   = 'typedef' S* TypeName S* <def TYPE W+> S* ';'

TypeName
   = BuildInTypeName / <isa TYPE>

BuiltInType
   = 'int' !W / 'long' !W / 'float' !W ...
```

Fig. 4 Productions for `typedef` and type names in C grammars.

Throughout our case studies on Nez extensions, we have obtained positive forecasts for expressing each of full language specifications. A significant exception is Haskell. As described below, Haskell's code layout is too amended to programmers. In addition, Haskell supports syntax extensions, which allow the users to even change the precedence of binary operators. This extensibility is not good for PEG's deterministic parsing based on the operator associativity.

The remainder of this section describes each language syntax that focuses on the Nez extensions. Note that all examples presented in this subsection are modified for improved readability.

4.2 C and Typedef-Defined Name

Parsing `typedef`-defined names is a classic problem of parser generators even with LALR and LL families. **Figure 4** shows excerpted productions from our C grammar. The production `TypeDef` describes a syntax of the `typedef` statement with defining matched symbols in the TYPE table. The production `TypeName` is defined to first match built-in type names and then match one of symbols stored in the TYPE table.

If the scope of type-defined names were global only, the production `TypeName` could work in any contexts. In reality, the `typedef` statement allows nested local scoping in a mixed manner with local variables. For example, the following is an legal C code:

```
typedef int T;

/* T is a type name */
```

© 2016 Information Processing Society of Japan

```
int main() {
  int T = 0;
  /* Here, T is a variable */
  {
    typedef double T;
    /* Here, T is a type name */
    printf("T=%f\n", (T)-1);
  }
  /* Again, T is a variable */
  printf("T=%d\n", (T)-1);
  return T;
}
/* Again, T is a type name */
```

To parse the above concisely, we need to maintain all local variable names with another symbol table, as well as to introduce the nested scoping with <block>. Let VAR be a symbol table for local variables. The following is a modified TypeName production to test the local variable name.

```
TypeName
   = BuildInTypeName / !<isa VAR> <isa TYPE>
```

This modification however still has difficulty in handling the inner nested typedef statement. Since Nez provides no supports for distinguishing table types, we cannot perform a concise parsing of duplicated names. However, the typedef statement is mostly used at the top level of source code. This is why we consider it not to be serious in the most practical cases.

4.3 HERE Document

The HERE document is a popular syntax of multiple line strings, by allowing the user to define a delimiting identifier of the end of those lines. The user-defined identifier, as with in typedef-defined names, can be maintained in a symbol table. **Figure 5** shows a fragmentation of the Ruby grammar.

The statement definitions involving HereDocu are defined inside a local scope. The production HereDocu matches a delimiting identifier to store in the DELIM table. The body of HERE document follows the statement, depending on the DELIM table. We use <exists DELIM> to test the existence of table entries. If it exists, we parse subsequent lines until the head of the line starts with <is DELIM>.

Note that the defined delimiting identifier is available only inside StatementDocu and Document productions. If a statement includes another statement, the isolation of scope is necessary. In such cases, we may use <local>, instead of <block>.

Ruby and many other scripting languages allows multiple HERE documents in a single statement. That is, the following is a legal code:

```
puts <<FIRST, <<SECOND
...
FIRST
...
SECOND
```

Nez provides no meta-variables for tables. Accordingly, we need to define a fixed number of tables in preparation. How many tables we need upfront depends on the language specification, while Ruby's reference manual does not mention the maximum number.

```
HereDocu
  = '<<' (!<exists DELIM> <def DELIM W+ > )
      /  (!<exists DELIM2> <def DELIM2 W+ > )
      / ..

Document
  = <exists DELIM>
   !<is DELIM> (!NL .)* NL <is DELIM> NL
    ( <exists DELIM2>
     !<is DELIM2> (!NL .)* NL <is DELIM2> NL
     ...
    )?
```

```
NL = '\r\n' / '\n'

Statement
  = <block StatementDocu NL
           Document?   >

HereDocu
  = '<<' <def DELIM W+ >

Document
  = <exists DELIM>
   !<is DELIM> (!NL .)* NL <is DELIM> NL
```

Fig. 5 Productions for HERE Document in Ruby.

4.4 Contextual Keywords

A contextual keyword is used to avoid the backward compatibility problem with language evolutions. An added new keyword is used to provide a specific meaning in a specific context of code; outside the context, programmers can still use it as an identifier. **Figure 6** shows a list of contextual keywords in C#5.0, implying that the evolution of C# relies largely on many contextual keywords. The same ideas are extensively discussed in many cases of language evolutions, including the future version of JavaScript and C++0x.

Fundamentally, PEGs are based on scanner-less parsing, and allows any tokens to be recognized as either keywords or identifiers depending on its context. Let us recall the await case, described in Section 2.2.4. The token await can be regarded as either an identifier by Identifier or a keyword by IdentiferNonAwait:

```
W = [A-z_0-9]
/* await can be an identifier */
Identifier = !Keyword [A-z_] W+
Keyword    = 'abstract' !W / 'as' !W / 'base' !W ...

/* await is a keyword */
IdentifierNonAwait
           = !KeywordAwait [A-z_] W+
```

```
add alias ascending async await descending dynamic
from get global group into join let orderby partial
remove select set value var where yield
```

Fig. 6 Contextual Keywords in C#5.0.

© 2016 Information Processing Society of Japan

```
KeywordAwait
           = 'abstract' !W / 'as' !W / 'await' !W
            / 'base' !W ...
```

The specification of contexts is the hard part. The keyword `await` is only available in `async` modified methods. The production `MethodDecl` needs to dispatch two different cases depending on the `async` modifier.

```
MethodDecl
    = 'async'Spacing MethodDeclAwaitContext
    / MethodDeclContext
```

The production `MethodDeclContext` is a standard version of method declaration, including such language syntax as blocks, statements, expressions, and variables. A problem with specifying `MethodDeclAwaitContext` is that we need to rewrite all await versions of these sub-productions that involving `Identifier` and `Keyword`. In our experience, the rewrites are needed for approximately 107 productions. As easily imagined, this approach would involve a considerable number of tedious specification tasks and would be then prone to errors.

Nez's conditional parsing, on the other hand, allows a single definition of the `Keyword` production to be differently recognized on a giving condition.

```
W = [A-z_0-9]

Identifier = !Keyword [A-z_] W+
Keyword    = 'abstract' !W / 'as' !W
           / <if AWAIT> 'await' !W
           / base !W ...
```

In addition, a single `MethodDeclContext` definition is also allowed. This reduces the duplication tasks of similar productions by hands.

```
MethodDecl
    = "async" __ <on AWAIT MethodDeclContext>
    / <on !AWAIT MethodDeclContext>
```

It is important to note that the conditional constructs such as `<if` *C*`>` and `<on` *C e*`>` can be removed from grammars by converting into condition-specific nonterminals. Actually, Nez parser performs such conversions upfront. As a result, the conditional parsing is an extension for improving the productivity of specification tasks.

4.5 Indentation-based Code Layout

Indentation-based code layout is a popular style of code layout, typically used in Python and Haskell. At the same time, it is a known fact [1], [2] that CFGs and PEGs are not able to recognize it without semantic actions.

While the symbol table is not specifically designed to handle indentations, we can store white spaces as a specialized symbol for representing indentation. To illustrate, let `S` be a spacing production such that `S = [ \n]`.

An indentation of a line heading can be defined as a white spacing symbol on the INDENT table:

```
<def INDENT S*> Statement
```

Note that `S*` is greedy matching that consumes all white spaces before `Statement`.

Now we can test the same length of white spaces with `<match INDENT>`, and a deeper indentation can be controlled by `<match INDENT>` followed by one-and-more repetition of white spaces:

```
<match INDENT> S+ Statement
```

The Python-style nested indentation layout can be handled by nested scope of the INDENT table. **Figure 7** is an excerpted grammar from our Python syntax, where the parsing condition `LO` is used to switch either layout-sensitive or layout-insensitive styles of code.

In addition, Python's indentation has an offside rule, or an exception of indentation-based layout; the layout is ignored inside parenthesized expression. For example, the indentation ahead of `2` is regarded as nothing:

```
if cond:
      a = (1 +
  2)
```

As the readers imagine, the offside rule is similar to contextual keywords, which implies that conditional parsing can switch behaviors. In this case, we express an offside rule by simply surrounding the parenthesized expression such as `'(' <on !LO Expr> ')'`.

Haskell has a different style of indentation-based code layout, where the depth of indentation is determined by specific keywords such as `let` and `where`. That is, for example, the following indentation of `y = b` must start with white spaces at the same position of `x = a`.

```
let x = a
     y = b
```

Apparently, we cannot extract white spaces for the INDENT

```
/* LO: a flag for indent-based layout */

Layout
    = <if LO> <def INDENT <match INDENT> S+>
    / S*

Block
    = EOS <on LO <block INDENT Statement*>>
    / <on !LO Statement>

Statement
    = IfStatement / WhileStatement /

IfStatement
    = Layout 'if' Expression ':' Block
     (Layout 'else' ':' Block)?

WhileStatement
    = Layout 'while' Expression ':' Block
```

Fig. 7 Fragment of Python Grammars.

© 2016 Information Processing Society of Japan

table from a matched string. To express the Haskell-style indentation, we require a language-specific symbol handler, such as <defindent>. While we experimentally support such language-specific handlers, we don't mention them due to the generality of the Nez extensions.

5. Performance Study

Linear time parsing is a central concern of backtracking parsers because backtracking may easily impose exponential costs in the worst cases. In PEGs, packrat parsing [4] is a known implementation method to avoid such potential exponential costs. However, the Nez extensions would give rise to a problem with packrat parsing, because the linear time guarantee of packrat parsing is based on the fact that PEG-based parsing is stateless. In other words, the trick of packrat parsing is the memoization of nonterminal calls at distinct positions to avoid redundant repeated calls. Apparently, the Nez extensions invalidates the feature of stateless parsing, as we described in the previous sections.

In terms of the lack of stateless features, a semantic action approach involves the same problem. In this light, Grimm, the author of Rats!, has assumed in Ref. [6] that the state changes in parsing programming languages always flow forward the input and then previously memoized results need to be invalidated. That is, the linearity of packrat parsing can be preserved. We examine this assumption with our developed grammars and Nez parser.

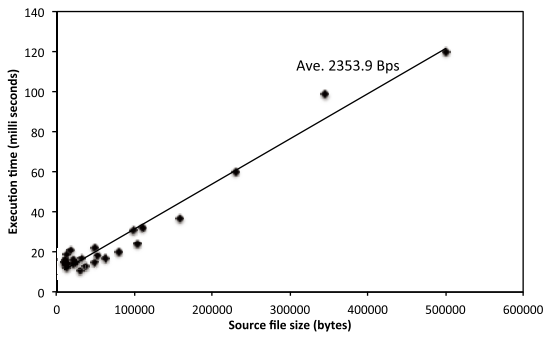

Fig. 8 Parsing Time in Java.

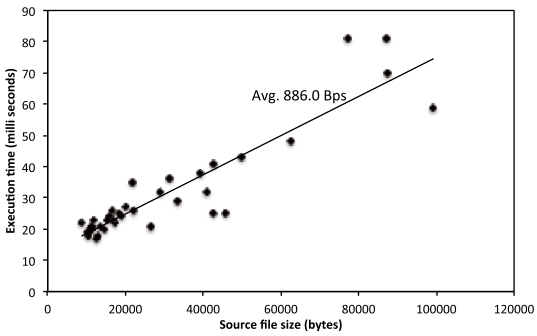

Fig. 9 Parsing Time in Ruby.

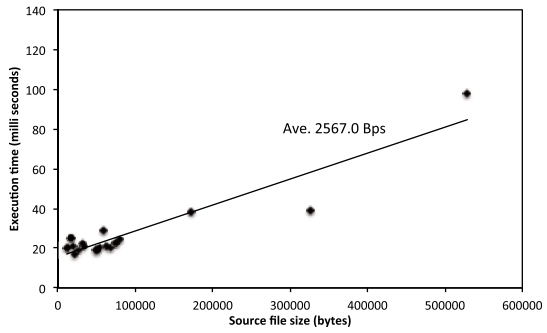

Fig. 10 Parsing Time in C#.

Figures 8, **9**, and **10** show the parsing time plotted against file sizes in, respectively, Java, Ruby, and C#. These tests were measured on an Apple's Mac Book Air, with 2 GHz Intel Core i7, 4 MB of L3 Cache, 8 GB of DDR3 RAM, running on Mac OS X 10.8.5 and Oracle Java Development Kit version 1.8. Tested files are collected from various open source repositories in order to examine different styles of coding. Tests run several times and we record the best time for each iteration. The execution time is measured by `System.nanoTime()` in Java APIs. The Nez parser that we have tested is based on Ref. [10].

As explained int Section 4, the Java grammar contain no Nez extension and the C# grammar contains parsing conditions, which are converted to condition free PEGs before parsing. These are stateless parsing, leading to no invalidation of packrat parsing. The Ruby grammar contains the symbol table handlers, <def> and <is>, which requires the treatment of state changes in packrat parsing. However, we haven't observed any significant difference on the linearity of parsing time, compared to other grammars. The reason is that the state changes operated by symbol definitions are perhaps localized and do not cause any significant invalidation of memoized results. We confirm Grimm's assumption with supporting evidence.

6. Related Work

Due to the popularity of PEGs, many grammar developers have attempted the grammar specification for their interesting languages. While PEGs, in some sense, are more powerful than CFGs, several limitations on their expressiveness have been pointed out in Refs. [5], [15].

Since YACC [8] has been broadly accepted as a standard parser, the semantic action (embedded code in a grammar) is a traditional and common approach to enhance the expressiveness of formal grammars, such as LR(k) and LL(k) [13]. Grimm presents that the semantic actions can be applied even into the speculative parsing such as PEGs [6]. More recently, most PEG-based parser generators (e.g., Mouse [14], PEGTL [7], and PEGjs [12]) have the semantic action supports for recognizing PEG-hard syntax, but the embedded action code depends on a host language of a parser.

A few researchers have attempted to extend the expressive power of PEGs itself. Notably, Adams newly introduced Indent-Sensitive CFGs [1] and its PEG-version [2] to recognize indentation-based code layout. The idea is based on constraint-based annotations on all nonterminals and terminals. As we described in Section 4.5, Nez can define an `INDENT` table and provide similar (not the same) effects to the Indent-Sensitive CFGs, as shown in **Table 2**.

To our knowledge, Nez is the first attempt to the declarative supports for recognizing various context-sensitive syntax patterns, including limitations that Ford's first pointed out.

© 2016 Information Processing Society of Japan

Table 2 Simple Correspondence between IS-CFGs and Nez INDENT table.

ISCFG	Nez/PEG
$e^{=}$	`<match INDENT>` e
$e^{>}$	`<match INDENT>` $S+$ e
$e^{\geq}$	`<match INDENT>` $S*$ e
$\lvert e \rvert$	`<block <def INDENT S*>` e`>`
e^{*}	`<local INDENT` e`>`

7. Conclusion

Parsing Expression Grammars are a popular foundation for describing syntax. Unfortunately, pure PEGs find it difficult to recognize several syntax patterns appearing in major programming languages. Notorious cases include typedef-defined names in C/C++, indentation-based code block in Python, and HERE document used in many scripting languages. To recognize such PEG-hard patterns, we have designed Nez as a pure and declarative extension to PEGs.

We present the language design of Nez, including symbol table handlers and conditional parsing. Using Nez, we have performed extensive case studies on programming language grammars, which include C, C#, Java8, JavaScript, Lua, Python, Ruby, etc. Our case studies indicate that the Nez extensions are a practical extension to improve the expressiveness of PEGs for recognizing major programming languages. Our developed artifacts will be available online, at http://nez-peg.github.io/.

Acknowledgments The authors thank the IPSJ PRO102 and PRO103 attendees for their feedback and discussions.

References

[1] Adams, M.D.: Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule, *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '13*, New York, NY, USA, pp.511–522, ACM (online), DOI: 10.1145/2429069.2429129 (2013).

[2] Adams, M.D. and Ağacan, O.S.: Indentation-sensitive Parsing for Parsec, *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, New York, NY, USA, pp.121–132, ACM (online), DOI: 10.1145/2633357.2633369 (2014).

[3] Bravenboer, M., Tanter, E. and Visser, E.: Declarative, Formal, and Extensible Syntax Definition for aspectJ, *Proc. 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, *OOPSLA '06*, New York, NY, USA, pp.209–228, ACM (online), DOI: 10.1145/1167473.1167491 (2006).

[4] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, *ICFP '02*, New York, NY, USA, pp.36–47, ACM (online), DOI: 10.1145/581478.581483 (2002).

[5] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '04*, New York, NY, USA, pp.111–122, ACM (online), DOI: 10.1145/964001.964011 (2004).

[6] Grimm, R.: Better Extensibility Through Modular Syntax, *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '06*, New York, NY, USA, pp.38–51, ACM (online), DOI: 10.1145/1133981.1133987 (2006).

[7] Hirsch, C. and Frey, D.: Parsing Expression Grammar Template Library (2014), available from ⟨https://code.google.com/p/pegtl/⟩.

[8] Johnson, S.C. and Sethi, R.: UNIX Vol.II, W.B. Saunders Company, Philadelphia, PA, USA, chapter Yacc: A Parser Generator, pp.347–374 (1990) (online), available from ⟨http://dl.acm.org/citation.cfm?id=107172.107192⟩.

[9] Kuramitsu, K.: Konoha: Implementing a static scripting language with dynamic behaviors, *S3 '10: Workshop on Self-Sustaining Systems*, New York, NY, USA, pp.21–29, ACM (online), DOI: http://doi.acm.org/10.1145/1942793.1942797 (2010).

[10] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window, *Journal of Infomration Processing*, Vol.23, No.4, pp.505–512 (online), DOI: http://doi.org/10.2197/ipsjjip.23.505 (2015).

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

[12] Majda, D.: PEG.js - Parser Generator for JavaScript (2015).

[13] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proc. 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '11*, New York, NY, USA, pp.425–436, ACM (online), DOI: 10.1145/1993498.1993548 (2011).

[14] Redziejowski, R.R.: Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking, *Fundam. Inf.*, Vol.79, No.3-4, pp.513–524 (2007) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2367396.2367415⟩.

[15] Ryu, S.: Parsing Fortress Syntax, *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, *PPPJ '09*, New York, NY, USA, pp.76–84, ACM (online), DOI: 10.1145/1596655.1596667 (2009).

[16] Sumii, E.: MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language, *Proc. 2005 Workshop on Functional and Declarative Programming in Education*, *FDPE '05*, New York, NY, USA, pp.27–38, ACM (online), DOI: 10.1145/1085114.1085122 (2005).

Tetsuro Matsumura received his master degree from Yokohama National University in engineering in 2015.

Kimio Kuramitsu is an Associate Professor, leading the Language Engineering research group at Yokohama National University. His research interests range from programming language design, software engineering to data engineering, ubiquitous and dependable computing. He has received the Yamashita Memorial Research Award at IPSJ. His pedagogical achievements include Konoha and Aspen, the first programming exercise environment for novice students. He earned his B.E. at the University of Tokyo, and his Ph.D. at the University of Tokyo under the supervision of Professor Ken Sakamura.

© 2016 Information Processing Society of Japan