Modularity'14

Composable User-Defined Operators That Can Express User-Defined Literals

Kazuhiro Ichikawa, Shigeru Chiba
The University of Tokyo

User-Defined Operators

useful for implementing internal DSLs

- can introduce DSL-like syntax
- can be used together with other operators

An example program using internal DSLs (in Scala)

Squeryl (OR Mapper)

```
val ids = from (DB.students) (s =>
            where (s.entranceYear == 2013) select (s.id))
for ( id <- ids ) {
  id should fullyMatch regex "48-13(6|7)6[0-9]{2}"
}
```

ScalaTest (unit test DSL)

regular expression

Existing User-Defined Operators

Their syntax is strictly restricted

In Scala, users can define only infix binary operators and postfix unary operators.

An example program using internal DSLs (in Scala)

() cannot be removed!

```
val ids = from (DB.students) (s =>
            where (s.entranceYear == 2013) select (s.id))
for ( id <- ids ) {
  id should fullyMatch regex "48-13(6|7)6[0-9]{2}"
}
```

combination of infix binary operators

literals cannot be expressed by operators

Existing User-Defined Operators

Their syntax is strictly restricted

In Scala, users can define only infix binary operators and postfix unary operators.

An example program using internal DSLs (in Scala)

```
val ids = from (DB.students) (s =>
            where (s.entranceYear == 2013) select (s.id))
for ( id <- ids ) {
  id should fullyMatch regex 48-13(6|7)6[0-9]{2}
}
```

integer number (not regular expression)

Desired User-Defined Operators

Accept flexible syntax

- not only infix, prefix, postfix, or outfix
- not only unary, binary, ...
- can express literals by combining operators

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                      where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

select _ from _ where _

_ should _

match _

user-defined literals !

Problem: Parsing is Difficult

The grammar may be highly ambiguous

- A DSL developer cannot know all DSLs that are used together with his/her DSL
- Every operator expresses an expression
  – Expression's rule would be complex
- Especially, literal rules introduce a large number of ambiguities. (cf. regex)

Naïve Solution is Inefficient

Generate all possible parse trees
and then choose the most suitable one

- Common scanner-less CFG parser takes $O(n^3)$ time if the grammar is ambiguous
  - * n = # of characters
- The number of trees might exponentially explode
  - * choosing the most suitable tree is difficult in a naïve way

Proposal:
Using Expected Type Info for Parsing

Parser uses only operators with the expected return type

- when the parser tries to parse an expression
- an operand is parsed by operators whose return type is the operand type.
- it can reduce ambiguities since operators with the same syntax can be distinguished by types

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Parsing Algorithm

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine expected type of the next expr
3) pick up an operator with expected return type, and try to parse the expr by the operator's rule
4) if the parser encounters an operand, go to 2
5) if an attempt succeeds, return the result. otherwise, go to 3 and try another operator

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

local variable declaration statement

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

the expected type is ResultSet

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by `select _ from _ where _`

returns `ResultSet`

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expression statement

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expects `void`

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by _ should _

returns void

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                      where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by _ should _

String

Matcher

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expects `Matcher`

Example

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by `match _`

returns `Matcher`

Parsing Precedence

For efficient parsing, we also propose to introduce parsing precedence.

- precedence rule among operators
  with the same return type
  (and the same operator precedence)
- which operator is chosen if an expr is ambiguous
- can remove all ambiguities,
  but may change the grammar

Efficiency

O(n) time for practical grammar

- ambiguities are removed by
  * using static types as non-terminal symbols
  * parsing precedence
- using memoization
  * for reducing the cost of backtrack
  * packrat parsing supporting left-recursion

Benefits

Operators can express literals

- literal is an expression with special whitespace rule
- literals overloads token rules

An example program using operators (in Java)

```
ResultSet ids = select id from DB.students
                      where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

```
_ _   : sequence                  _ { _ }   : repetition N times
_ | _ : alternative               [ _ - _ ] : character class
```

Drawbacks

Limited places where operators are available

- only in expressions whose expected type is statically determined before parsing
- depends on the host language
- e.g. the receiver of a method call in Java

not available	available
a = b	a = b
a.method(x, y)	a.method(x, y)
a.field	e;
(Type)a	return a;
	…

Implementation: ProteaJ

A subset of Java with

- flexible user-defined operators
- provides a module system for operators
- does not support generics

Source and test programs are available from:

https://github.com/csg-tokyo/proteaj.git

Operators in ProteaJ

Operator syntax = { name | operand }+

- Not only infix, prefix, unary, binary, ...
- Expressiveness is equivalent to PEG

Extra features

- operator precedence and associativity
- two whitespace rules : expression / literal level

A definition of an operator in ProteaJ

name operand

return type

```
ResultSet "select" col "from" table "where" cond
  (Column col, Table table, Condition cond) {
  return prepareSQLStmt(col, table, cond).execute();
}
```

Experiment

Experiment Environment

CPU: 2.67 GHz Core i5
Memory: 8GB
OS: OpenSUSE 12.1
Java: OpenJDK 1.7.0

Our compiler vs. JSGLR parser

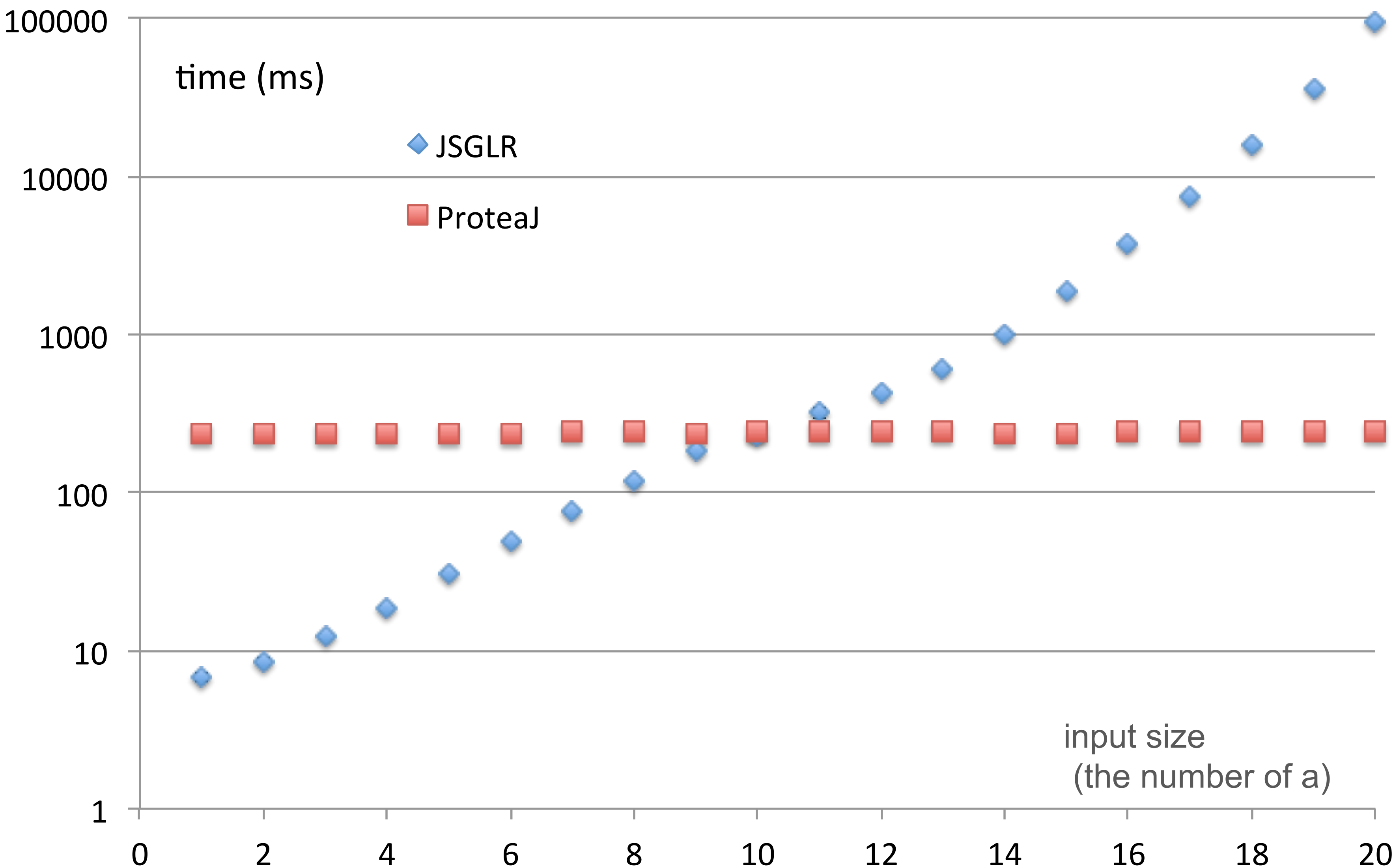
JSGLR: well-known scanner-less CFG parser
It can generate all possible trees

Problem settings

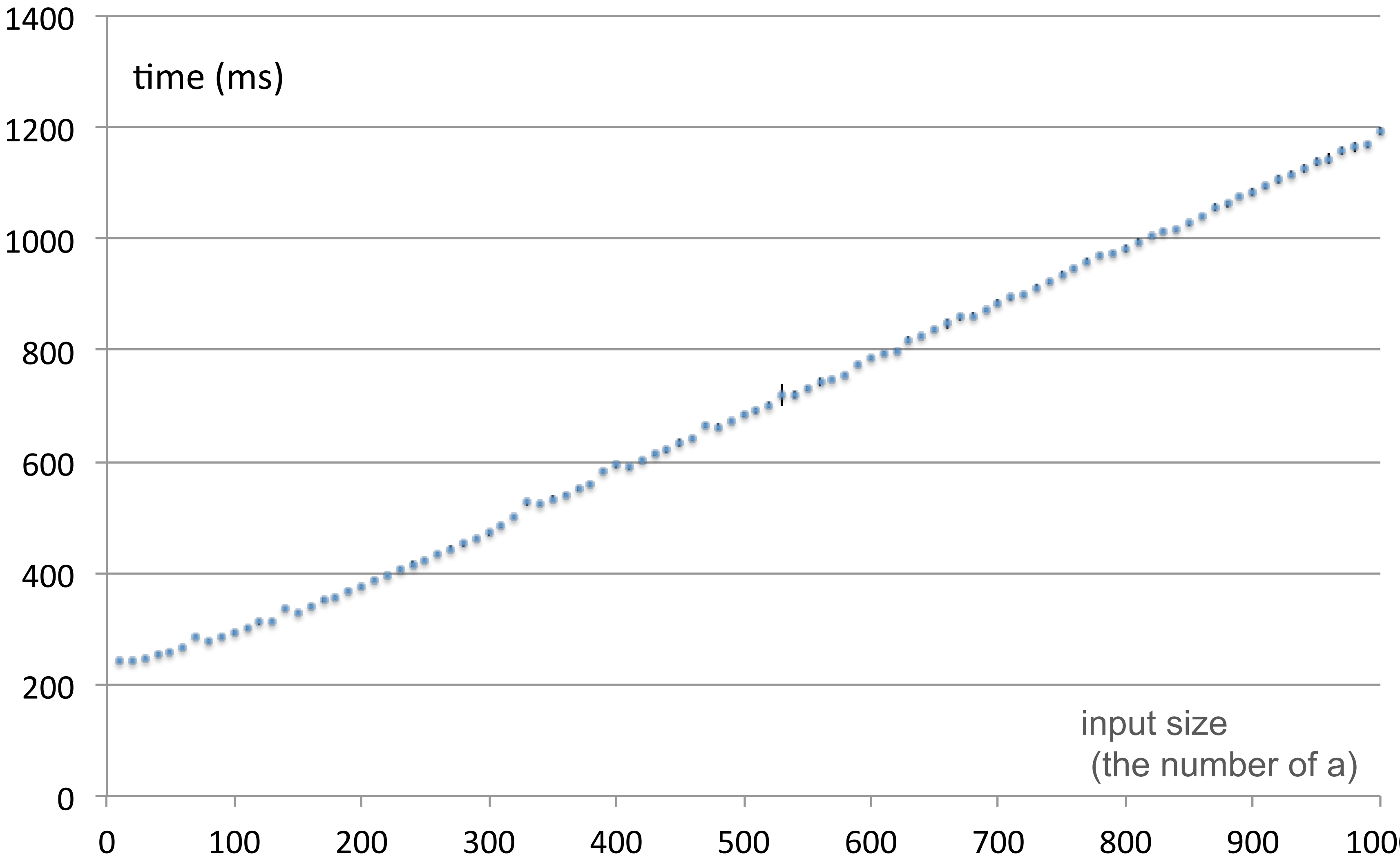
- grammar: arithmetic operators + file path
- input: a/a/a/…/a (input size = # of a)

note: In the ProteaJ experiment, the input is embedded in a minimal program. In the case of JSGLR, it parsed as is.

Result (semi-log graph)



Compilation time by ProteaJ (linear-scale)



Related Work: User-Defined Operators

Mixfix operators

- a class of user-defined operators
- only infix, prefix, postfix, or outfix
- Coq, Agda, Pure, OBJ3, Isabelle, ...

Mixfix operators + implicit (empty) operators

- mixfix + operator having no name
- poorly supports user-defined literals
- OBJ3, Isabelle

Related Work: Parsing

CFG parser + type-based disambiguation

- generate all ASTs => type check
- inefficient for highly ambiguous grammar
- Metaborg, Agda, OBJ3, Isabelle, ...

Type-oriented island parsing [Silkenson '12]

- bottom-up parsing using type information
- cannot define new (complex) literals

Conclusion

Parsing method for flexible operators

- using expected type information
- precedence rule: parsing precedence
- O(n) parse time for practical grammar

Benefits

- Operators can express literals

Drawbacks

- Limited places where operators are available

Efficiency

O(n) time <u>for practical grammar</u>

- ambiguities are removed by
  - * using static types as non-terminal symbols
  - * parsing precedence
- operators ≒ PEG including left-recursion
  - * operator name ≒ terminal
  - * return type, operand type ≒ non-terminal
  - * parsing precedence ≒ ordered choice

Parsing Precedence

For efficient parsing, we also propose to introduce parsing precedence.

- precedence rule among operators
  with the same return type
  (and the same operator precedence)
- which operator is chosen if an expr is ambiguous
- parsing precedence ≒ ordered choice rule
- it is declared by programmers

User-Defined Literals

Protean operators can express literals

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                 where entranceYear == 2013;
for (String id : ids.toList()) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

```
_ _   : sequence                 _ { _ }   : repetition N times
_ | _ : alternative              [ _ - _ ] : character class
```

Motivation: Internal DSL

DSL implemented as a library

- It can be used as a part of its host language
- e.g. parser combinator, OR mapper

DSL (Domain Specific Language)

- specialized language for a specific purpose
- e.g. yacc, SQL

Pros/Cons of Internal DSL

Pros: **Composability**

- It can be used together with other DSLs

Cons: **Syntax**

- The syntax is restricted by its host language

Pros/Cons of Internal DSL

Pros: Composability

It can be used together with other DSLs

Cons: Syntax

The syntax is restricted by its host language

An example program using internal DSLs (in Scala)

```
val ids = from (DB.students) (s =>
              where (s.entranceYear == 2013) select (s.id))
for ( id <- ids ) {
  id should fullyMatch regex "48-13(6|7)6[0-9]{2}"
}
```

Squeryl (OR Mapper)

cannot be removed

ScalaTest (unit test DSL)

regular expression

Goal: Composable Syntax Extension

Enabling a DSL to introduce its own syntax

- the syntax is not restricted by the host lang.
- the syntax includes literal-level syntax

Without breaking composability

- multi-DSLs can be used together safely
- without critical penalty of compilation time

Proposal: **Protean Operators**

A class of user-defined operators

- consist of names and operands
  - * not only infix, prefix, postfix, and outfix
- overloaded by its return type
  - * an operator is available only at an expression where the return type is expected
- have a special rule: parsing precedence
  - * Programmers should declare the precedence among operators with the same return type

Protean Operators Introduce DSL Syntax

DSL syntax can be expressed by protean operators !

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                    where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

select _ from _ where _

_ should _

match _

regular expression literals

Protean Operators Introduce DSL Syntax

DSL syntax can be expressed by protean operators !

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

select _ from _ where _

_ should _

match _

regular expression literals

```
_ _   : sequence            _ { _ }   : repetition N times
_ | _ : alternative         [ _ - _ ] : character class
```

Protean Operators are Composable

Compiler can distinguish operators by **types** even if they have the same syntax !

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

2013 is an integer literal
because int is expected.

This part is parsed by regex operators
because Regex is expected here !

Parsing

We developed a parsing method that uses expected type information.

1) parse a statement by the host language rules until the parser encounters an expression part
2) determine the expected type of the next expression
3) parse the expression by the operators that return the expected type
4) if the parser encounters an operand, go to 2

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                    where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

local variable declaration statement

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

the expected type is `ResultSet`

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by `select _ from _ where _`

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expression statement

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expected `void`

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by _ should _

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

expected Matcher

Parsing

We developed a parsing method that uses expected type information.

An example program using protean operators (in Java)

```
ResultSet ids = select id from DB.students
                  where entranceYear == 2013;
for (String id : ids.toList() ) {
  id should match 48-13(6|7)6[0-9]{2};
}
```

parsed by `match _`

Implementation: ProteaJ

A subset of Java + protean operators

- provides module system for operators
- not supports generics

Source and test programs are available from:

www.csg.ci.i.u-tokyo.ac.jp/~ichikawa/ProteaJ.tar.gz

Expressiveness of Protean Operators

Pros: They can express any PEGs

- non-terminal => static type
- PEG (Parsing Expression Grammar) is a type of formal grammar like CFG

Cons: They cannot express declarations

- They do not use meta-programming

Efficiency of Our Parsing Method

O(n) for practical grammar

- n: input source length (# of letters)
- use memoization to reduce back-track cost

Naive method is inefficient

- generate all possible ASTs and then choose most suitable one by using types
- parser that can generate all possible ASTs is inefficient against highly ambiguous grammar

Experiment

Experiment Environment

CPU: 2.67 GHz Core i5
Memory: 8GB
OS: OpenSUSE 12.1
Java: OpenJDK 1.7.0

Our compiler vs. JSGLR parser

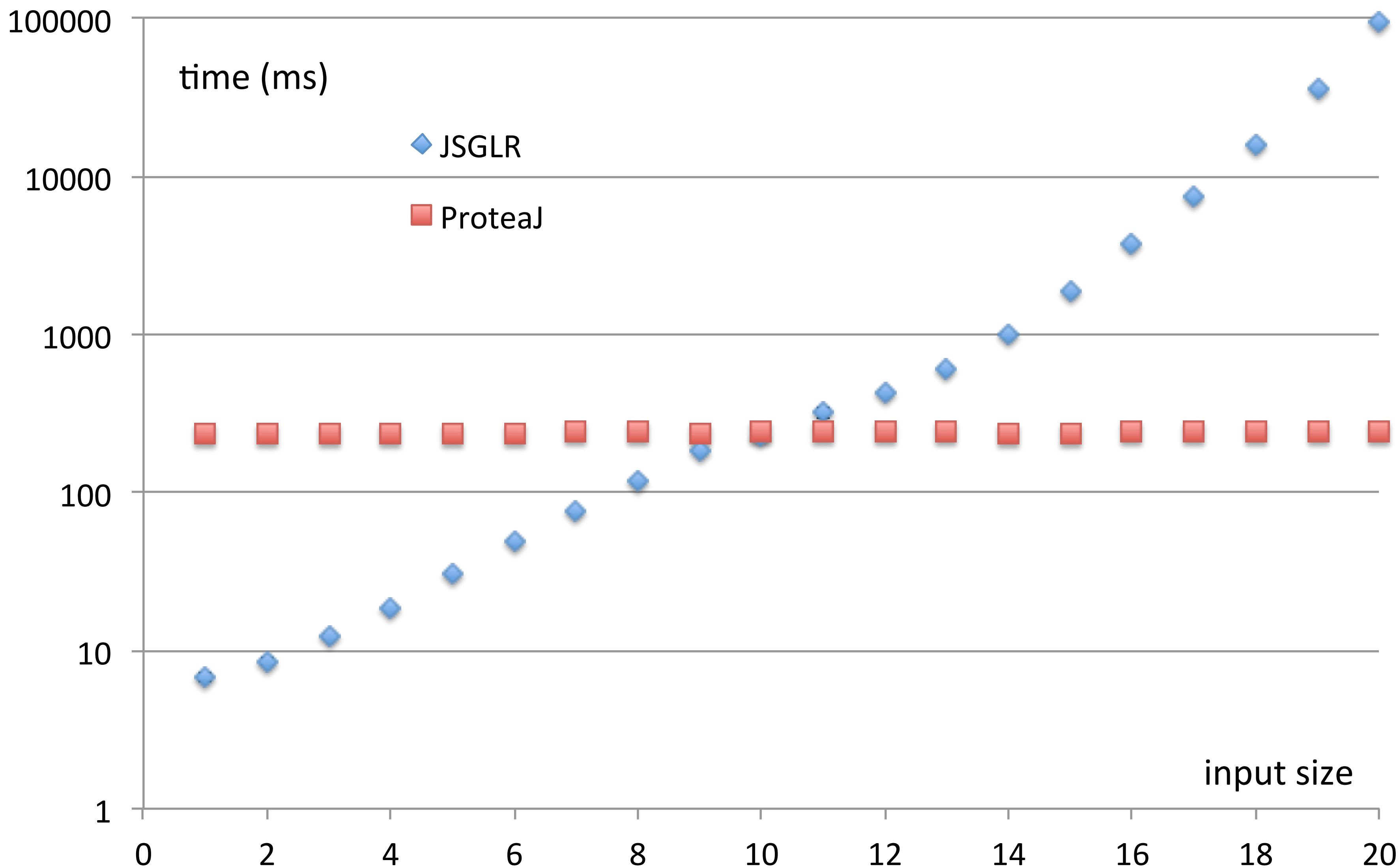
JSGLR: well-known scanner-less CFG parser
It can generate all possible trees

Problem settings

- grammar: arithmetic operators + file path
- input: a/a/a/…/a (input size = # of a)

note: The input for our compiler is more complex
since it must be a valid ProteaJ program.

Result (semi-log graph)



Related Work: User-Defined Operators

Mixfix operators

- a class of user-defined operators
- only infix, prefix, postfix, or outfix
- Agda, Pure, OBJ3, Isabelle, ...

Mixfix operators + empty operators

- mixfix + operator having no name
- cannot define new (complex) literals
- OBJ3, Isabelle

Related Work: Parsing

CFG parser + type-based disambiguation

- generate all ASTs => type check
- inefficient for highly ambiguous grammar
- Metaborg, Agda, OBJ3, Isabelle, ...

Type-oriented island parsing [Silkenson '12]

- bottom-up parsing using type information
- cannot define new (complex) literals

Conclusion

Protean operators

- expressiveness is equivalent to PEG
- multiple operators can be used safely

Parsing method

- uses expected type information
- O(n) for practical grammar