The Expression Problem

Gracefully

Andrew P. Black
Portland State University

Portland State University


eattle
WASHINGTON
Spokane
82
Walla Walla
Salem
Eugene
OREGON
Snake
Boise
IDAHO
Owyhee
Pit
Helena
Great Falls
Missouri
MONTANA
90
Billings
Sheridan
Minot
NORTH DAKOTA
Bismarck
James
Red River
MINNESOTA
Duluth
Minneapolis
SOUTH DAKOTA
Sioux Falls
WYOMING
25
Cheyenne
15
80
Reno
NEVADA
San Francisco
Sacramento
San Jose
Salt Lake City
UTAH
Colorado
Denver
Colorado Springs
COLORADO
Arkansas
Fresno
CALIFORNIA
Las Vegas
St. George
Bakersfield
Los Angeles
San Diego
Mexicali
ARIZONA
Phoenix
Gila
10
Tucson
Nogales
40
Albuquerque
NEW MEXICO
NEBRASKA
Lincoln
Omaha
UNITED STATES
Topeka
Kansas City
KANSAS
Wichita
Amarillo
Lubbock
Oklahoma City
OKLAHOMA
Lawton
Tulsa
Joplin
Fort Smith
Paris
Red
Fort Worth
Dallas
Abilene
TEXAS
Waco
Tyler
Pecos
Austin
San Antonio
Houston
35
Corpus Christi
IOWA
Des Moines
80
Rockford
Chicago
WISCONSIN
Green Bay
Milwaukee
MICHIGAN
Lansing
Detroit
Toledo
Fort Wayne
ILLINOIS
Springfield
Terre Haute
St. Louis
MISSOURI
Springfield
ARKANSAS
Little Rock
Shreveport
Natchitoches
LOUISIANA
Beaumont
New Orleans
Memphis
Jackson
MISSISSIPPI
Jackson
Hattiesburg
Mobile
INDIANA
Indianapolis
Evansville
Louisville
KENTUCKY
Nashville
TENNESSEE
Huntsville
Tupelo
Birmingham
ALA.
Montgomery
Dothan
Tallahassee
Knoxville
Chattanooga
Atlanta
GEORGIA
Macon
Savannah
Jacksonville
Gainesville
Ocala
Orlando
Tampa
FLORIDA
Miami
OHIO
Columbus
W. VA.
Richmond
VIRGINIA
Virginia Beach
Winston-Salem
Durham
N.C.
Greenville
40
Charlotte
SOUTH CAROLINA
PENNSYLVANIA
Baltimore
Washington, D.C.
Dover
Philadelphia
New York
Erie
NEW YORK
Syracuse
Watertown
CONN.
Boston
N.H.
VT.
MAINE
Sherbrooke
Augusta

The Expression Problem

Oliveira & Cook (ECOOP 2012):

"The "expression problem" (EP) [38, 10, 46] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants."

Extensibility for the Masses
Practical Extensibility with Object Algebras

Bruno C. d. S. Oliveira[1] and William R. Cook[2]

[1]National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

Abstract. This paper presents a new solution to the expression problem (EP) that works in OO languages with simple generics (including Java or C#). A key novelty of this solution is that advanced typing features, including F-bounded quantification, wildcards and variance annotations, are not needed. The solution is based on *object algebras*, which are an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the VISITOR pattern, but without many of its drawbacks: they are extensible, remove the need for accept methods, and do not compromise encapsulation. We show applications of object algebras that go beyond toy examples usually presented in solutions for the expression problem. In the paper we develop an increasingly more complex set of features for a mini-imperative language, and we discuss a real-world application of object algebras in an implementation of remote batches. We believe that object algebras bring extensibility to the masses: object algebras work in mainstream OO languages, and they significantly reduce the conceptual overhead by using only features that are used by everyday programmers.

1 Introduction

The "expression problem" (EP) [38, 10, 46] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult. The VISITOR Pattern [13] is often used to allow operations to be added to object-oriented data abstractions, but the common approach to visitors prevents adding new classes. Extensible visitors can be created [43, 50, 31], but so far solutions in the literature require complex and unwieldy types, or advanced programming languages.

In this paper we present a new approach to the EP based on *object algebras*. An object algebra is a class that implements a generic abstract factory interface, which corresponds to a particular kind of *algebraic signature* [18]. Object

Portland State UNIVERSITY

Monday, 6 July 2015

What *is* the Expression Problem?

- Consider a simple implementation of (immutable) lists

		Operations		
		first	rest	isEmpty
Repres-entations	ConsList (e, l)	return e	return l	false
	EmptyList	error	error	true

Algebraic data types:

- Organize program by columns

		Operations		
		first	rest	isEmpty
Repres-entations	ConsList (e, l)	return e	return l	false
	EmptyList	error	error	true
		first function	rest function	isEmpty function

Algebraic data types:

- Organize program by columns
  - easy to add a new column, but hard to add a new row

		Operations		
		first	rest	isEmpty
Repres-entations	ConsList (e, l)	return e	return l	false
	EmptyList	error	error	true
		first function	rest function	isEmpty function

Objects

- Organize program by rows

		Operations			
		first	rest	isEmpty	
Repres-entations	ConsList (e, l)	return e	return l	false	ConsList class
	EmptyList	error	error	true	EmptyList class

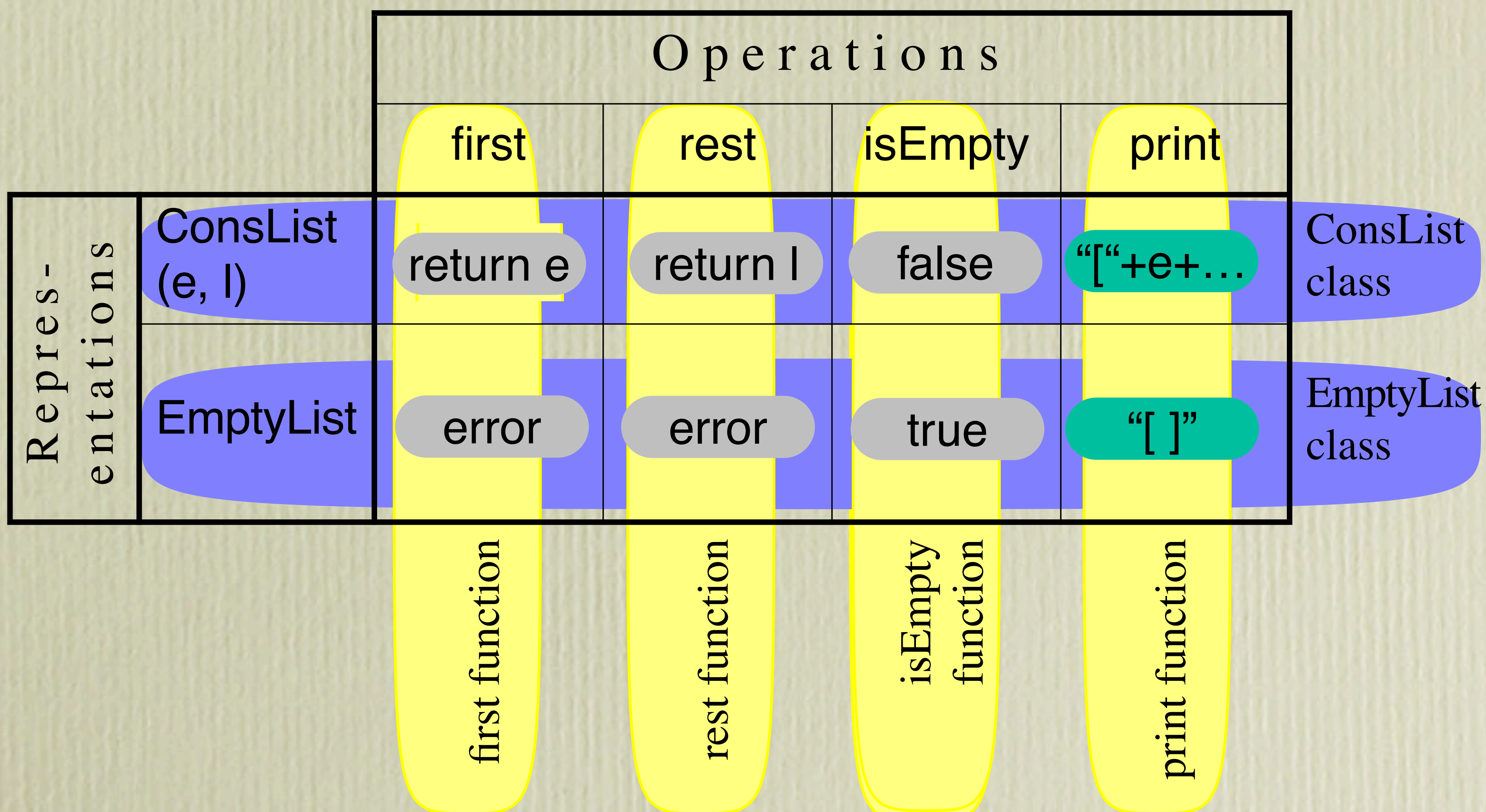
Objects

- Organize program by rows
  - easy to add a new row, but hard to add a new column

Repres-entations		Operations			
		first	rest	isEmpty	
	ConsList (e, l)	return e	return l	false	ConsList class
	EmptyList	error	error	true	EmptyList class

Example: add an operation

- *One* new function with algebraic data, but *two* new methods in two classes with objects



Why is this "difficult"?

- An editing problem
  - assumption: adding methods to two classes involves editing two files
- A packaging problem
  - assumption: the class is the smallest unit of modularity, so editing a class breaks modularity
- A typing problem
  - assumption: fields of the objects have been given types that allow just the base operations

Some History ...

Oliveira & Cook (ECOOP 2012):

"The "expression problem" (EP) [38, 10, 46] is now a classical problem in programming languages."

38. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)

10. Cook,W.R.: Object-oriented programming versus abstract data types. In:Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages. pp. 151–178. Springer-Verlag (1991)

46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Some History ...

Krishnamurthi et al. captured the issue:

"*A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to*

1. *add a new variant of data and adjust the existing tools accordingly, and*
2. *extend the collection of tools.*"

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In ECOOP'98 — Object-Oriented Programming, E. Jul, Ed. LNCS vol. 1445. Springer, pp. 91–113.

46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Some History ...

Krishnamurthi et al. captured the issue:

"A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. *add a new variant of data and adjust the existing tools accordingly, and* new rows
2. *extend the collection of tools."*

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In ECOOP'98 — Object-Oriented Programming, E. Jul, Ed. LNCS vol. 1445. Springer, pp. 91–113.

46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Some History ...

Krishnamurthi et al. captured the issue:

"A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. *add a new variant of data and adjust the existing tools accordingly, and* new rows
2. *extend the collection of tools."* new columns

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In ECOOP'98 — Object-Oriented Programming, E. Jul, Ed. LNCS vol. 1445. Springer, pp. 91–113.

46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Some History ...

Krishnamurthi et al. captured the issue:

“A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. *add a new variant of data and adjust the existing tools accordingly, and*
2. *extend the collection of tools.”*

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. 1998. Synthesizing object-oriented and functional design to promote re-use. In ECOOP’98 — Object-Oriented Programming, E. Jul, Ed. LNCS vol. 1445. Springer, pp. 91–113.

46

G

Restriction: “ideally, these extensions should not require any changes to existing code”

new rows

new

columns

Some History ...

Wadler made this "problem" famous in 1998 (by coining a catchy name)

"The Expresion Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing."

38. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)

10. Cook,W.R.: Object-oriented programming versus abstract data types. In:Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages. pp. 151–178. Springer-Verlag (1991)

46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Some History ...

Wadler made this "problem" famous in 1998 (by coining a catchy name)

"*The Expresion Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing.*"

38. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)

10. Cook,W.R.: Obj… programming versus a… ngs of … of …

46 … G…

Restrictions:

1. Static type safety (no casts)
2. No *recompilation* of existing code

Why wait until 1998?

- Simula '67, C++
  - have algebraic data as well as objects
- Smalltalk 80
  - classes are not the unit of modularity
- Visitor Pattern (name *Visitor* coined 1993)
  - solves the problem, at the cost of pre-planning

Portland State UNIVERSITY

Why wait until 1998?

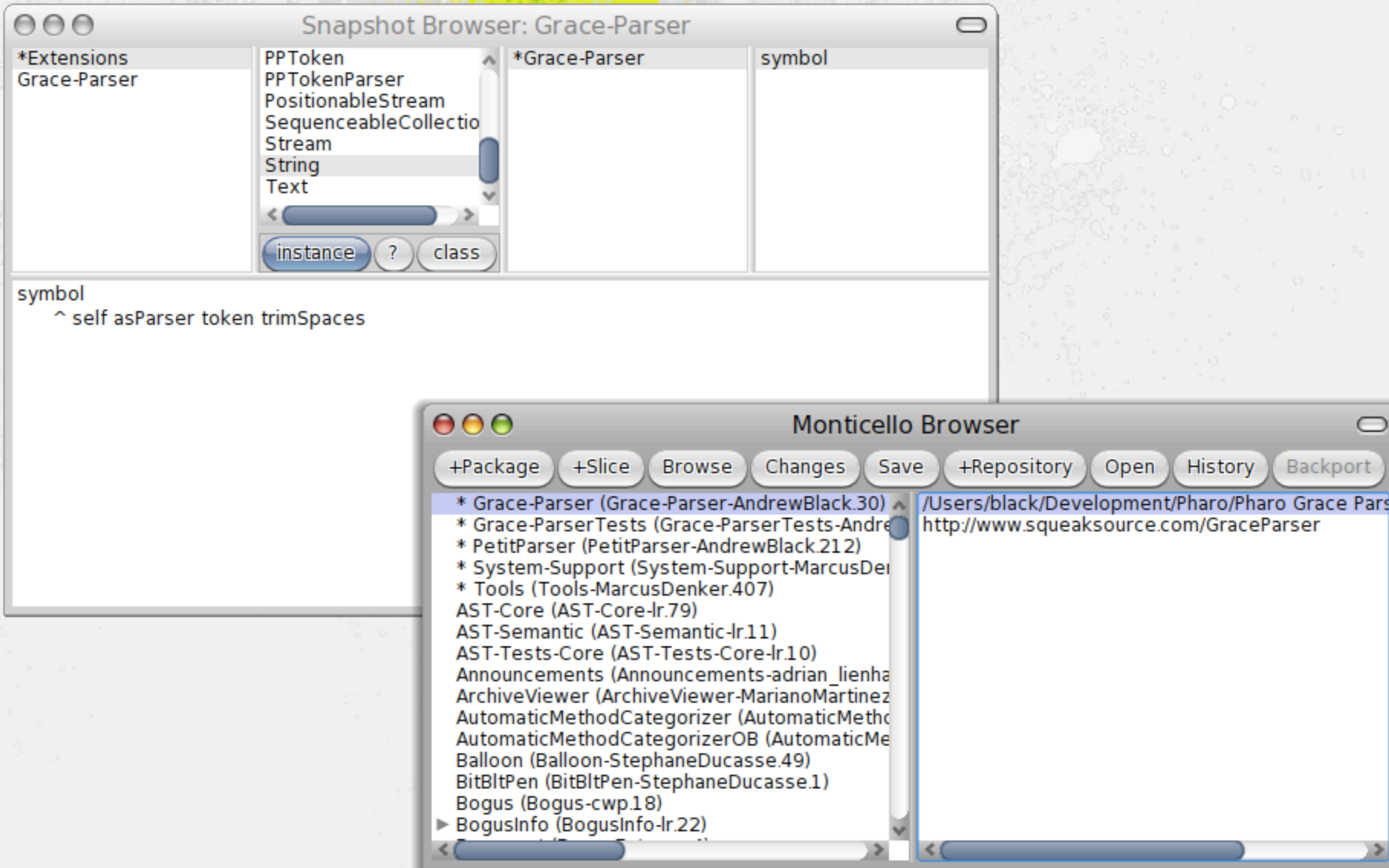
Why wait until 1998?

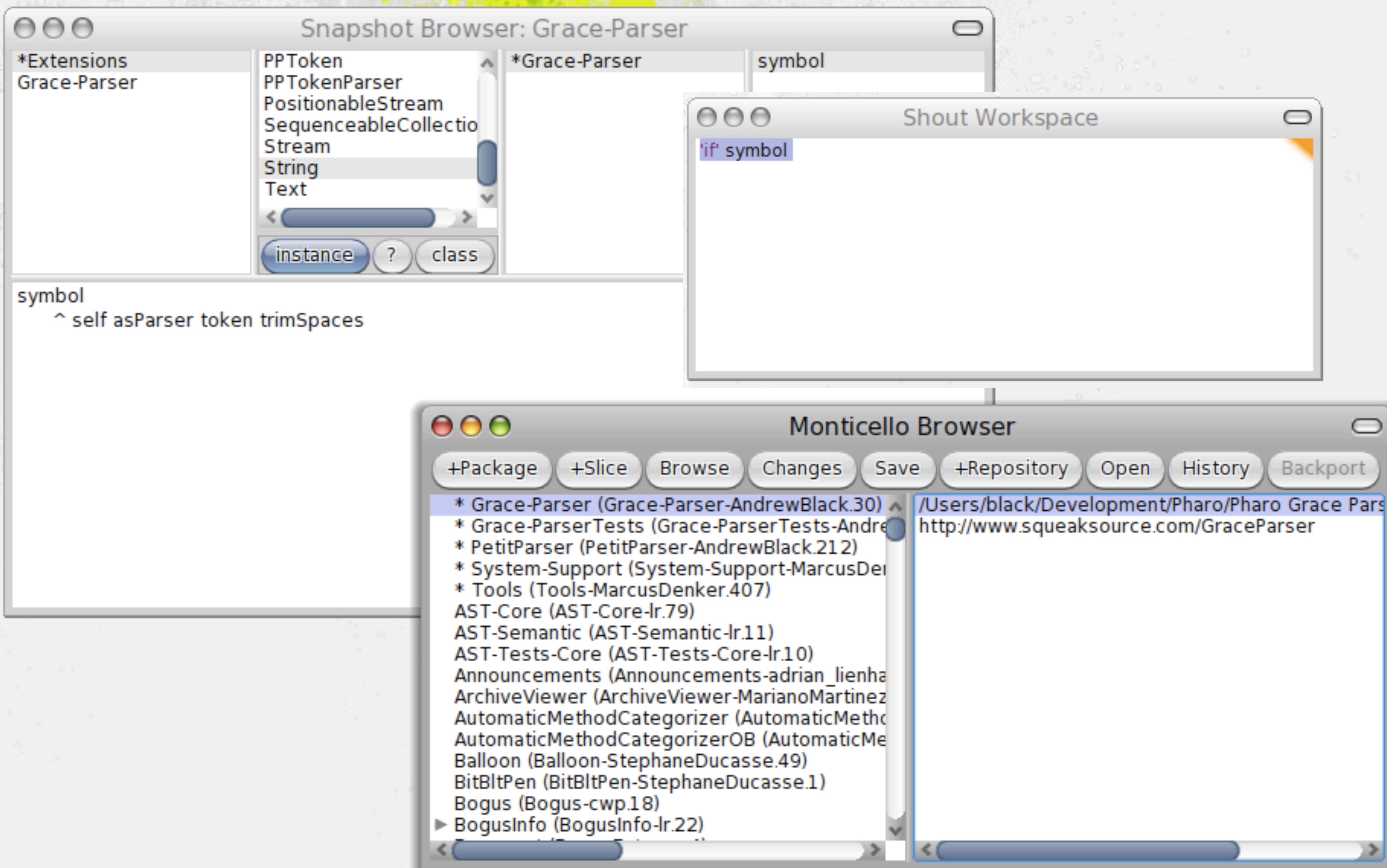
Why wait until 1998?

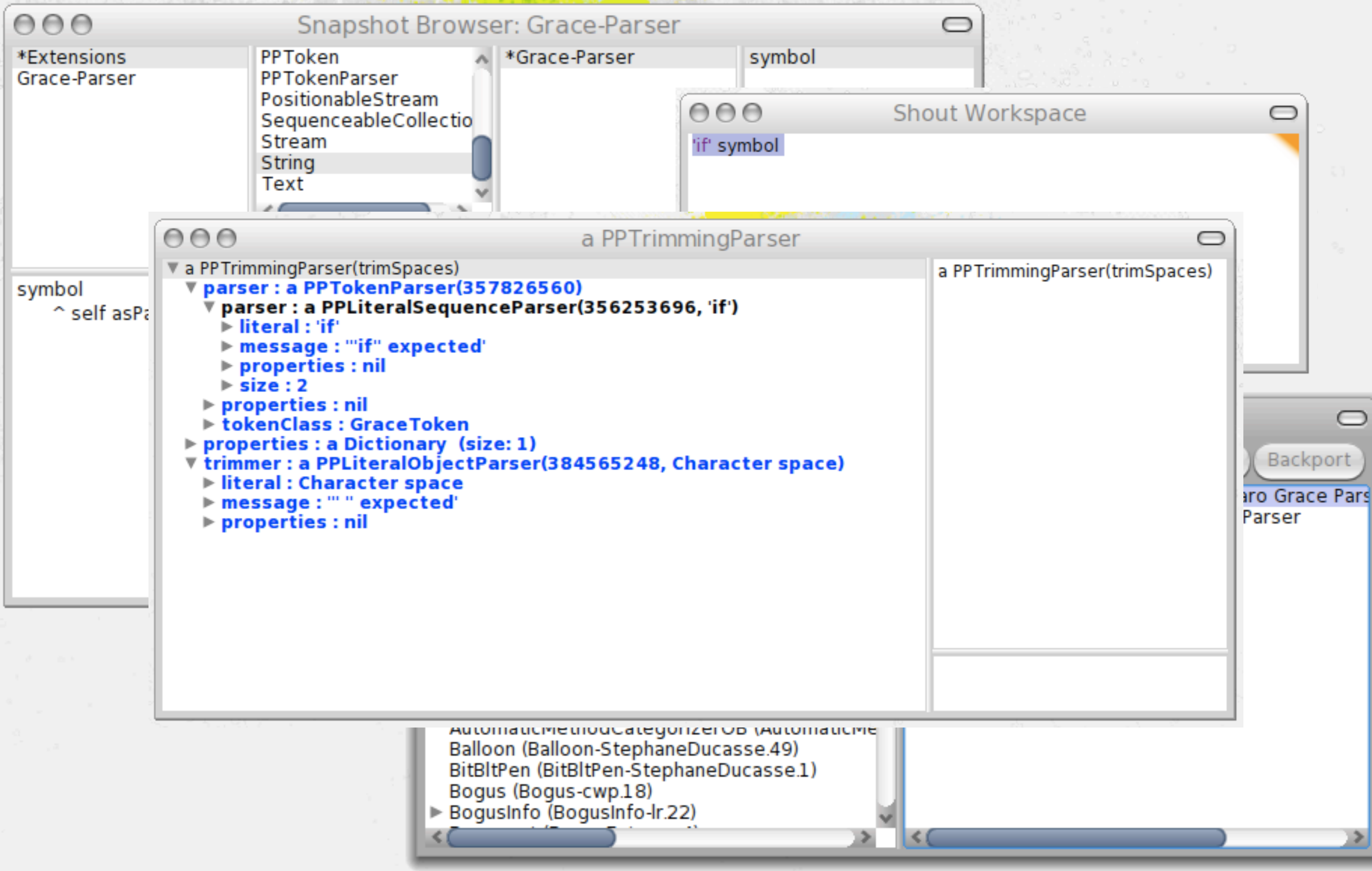
< Java™ >

Back to the future ...

- How was this "problem" solved in Smalltalk?
  - Classes named by global *variables*
  - *methods* are the unit of compilation & packaging
  - a package contains both *new classes* (and their methods) and extensions to existing classes (*new methods*)
  - loading a package into a Smalltalk system:
    - *changes* some existing classes (overrides and adds methods, adds instance variables)
    - introduces some new classes


Snapshot Browser: Grace-Parser
*Extensions
Grace-Parser
PPToken
PPTokenParser
PositionableStream
SequenceableCollectio
Stream
String
Text
instance
?
class
*Grace-Parser
symbol
symbol
^ self asParser token trimSpaces
Monticello Browser
+Package
+Slice
Browse
Changes
Save
+Repository
Open
History
Backport
* Grace-Parser (Grace-Parser-AndrewBlack.30)
* Grace-ParserTests (Grace-ParserTests-Andre
* PetitParser (PetitParser-AndrewBlack.212)
* System-Support (System-Support-MarcusDe
* Tools (Tools-MarcusDenker.407)
AST-Core (AST-Core-lr.79)
AST-Semantic (AST-Semantic-lr.11)
AST-Tests-Core (AST-Tests-Core-lr.10)
Announcements (Announcements-adrian_lienha
ArchiveViewer (ArchiveViewer-MarianoMartinez
AutomaticMethodCategorizer (AutomaticMetho
AutomaticMethodCategorizerOB (AutomaticMe
Balloon (Balloon-StephaneDucasse.49)
BitBltPen (BitBltPen-StephaneDucasse.1)
Bogus (Bogus-cwp.18)
BogusInfo (BogusInfo-lr.22)
/Users/black/Development/Pharo/Pharo Grace Pars
http://www.squeaksource.com/GraceParser


Snapshot Browser: Grace-Parser
*Extensions
Grace-Parser
PPToken
PPTokenParser
PositionableStream
SequenceableCollectio
Stream
String
Text
instance
?
class
*Grace-Parser
symbol
symbol
^ self asParser token trimSpaces
Shout Workspace
'if' symbol
Monticello Browser
+Package
+Slice
Browse
Changes
Save
+Repository
Open
History
Backport
* Grace-Parser (Grace-Parser-AndrewBlack.30)
* Grace-ParserTests (Grace-ParserTests-Andre
* PetitParser (PetitParser-AndrewBlack.212)
* System-Support (System-Support-MarcusDer
* Tools (Tools-MarcusDenker.407)
AST-Core (AST-Core-lr.79)
AST-Semantic (AST-Semantic-lr.11)
AST-Tests-Core (AST-Tests-Core-lr.10)
Announcements (Announcements-adrian_lienha
ArchiveViewer (ArchiveViewer-MarianoMartinez
AutomaticMethodCategorizer (AutomaticMetho
AutomaticMethodCategorizerOB (AutomaticMe
Balloon (Balloon-StephaneDucasse.49)
BitBltPen (BitBltPen-StephaneDucasse.1)
Bogus (Bogus-cwp.18)
BogusInfo (BogusInfo-lr.22)
/Users/black/Development/Pharo/Pharo Grace Pars
http://www.squeaksource.com/GraceParser


Snapshot Browser: Grace-Parser
*Extensions
Grace-Parser
PPToken
PPTokenParser
PositionableStream
SequenceableCollectio
Stream
String
Text
*Grace-Parser
symbol
symbol
^ self asPa
Shout Workspace
'if' symbol
a PPTrimmingParser
a PPTrimmingParser(trimSpaces)
parser : a PPTokenParser(357826560)
parser : a PPLiteralSequenceParser(356253696, 'if')
literal : 'if'
message : '"if" expected'
properties : nil
size : 2
properties : nil
tokenClass : GraceToken
properties : a Dictionary (size: 1)
trimmer : a PPLiteralObjectParser(384565248, Character space)
literal : Character space
message : '" " expected'
properties : nil
a PPTrimmingParser(trimSpaces)
Backport
Parser
Balloon (Balloon-StephaneDucasse.49)
BitBltPen (BitBltPen-StephaneDucasse.1)
Bogus (Bogus-cwp.18)
BogusInfo (BogusInfo-lr.22)

Why does this work?

- Classes are (mutable) objects
  - adding (or changing) a method mutates the class
- Classes are named by global *variables*
  - loading a new version of a class definition changes the value of the global variable, and recompiles all existing methods
- Objects created by a methods in a class
- No modular type-checking

Why doesn't it work in Java?

- Classes are *not* objects, and are immutable
  - Classes can be changed only by editing the source and recompiling
- Classes have global names, and cannot be renamed, assigned, or aliased
- Objects created by a language built-in **new**
- Modular type-checking

Java	Smalltalk
e = **new** EmptyList	e := EmptyList new
o = e.append(23)	o := e ++ 23

Java	Smalltalk
e = **new** EmptyList	e := EmptyList new
o = e.append(23)	o := e ++ 23

Data (row) *extensibility* is easy: add a new package defining a new class (but also must change creation code)

Java	Smalltalk
e = **new** EmptyList	e := EmptyList new
o = e.append(23)	o := e ++ 23

Data (row) *extensibility* is easy: add a new package defining a new class (but also must change creation code)

Operation (column) *extensibility* is impossible: can't change an existing class without editing the source.

- What about subclassing?
  - idea: subclass all of the original classes to create new variants with the additional operations.
  - Wadler focussed on generalizing the Java type system to make it possible to write those subclasses.
- But this doesn't help!
  - We still have to *change all the creation code* to use the new classes instead of the existing classes.

Grace

- new, simple O-O language
  - designed for teaching novice programmers the concepts of object-oriented programming
- block-structured within a module
- modules are objects
- no global variables
  - modules are imported under a name chosen by the *client*

Oliveira and Cook's Example

exp_base

		Operations
		eval
Representations	lit(n)	n
	sum(e_1, e_2)	e_1.eval + e_2.eval

Oliveira and Cook's Example

exp+pretty		Operations	
		eval	pretty
Representations	lit(n)	n	"{n}"
	sum(e_1, e_2)	e_1.eval + e_2.eval	"{e_1.pretty} + {e_2.pretty}"

Portland State UNIVERSITY

Monday, 6 July 2015

Oliveira and Cook's code

from their paper, translated into Grace

exp_base.grace

```
dialect "staticTypes"

type Value = Object
type Exp = { eval -> Value }

factory method lit(i:Number) -> Exp {
    method x -> Number { i }
    method eval -> Value { x }
}
factory method sum(a:Exp, b:Exp) -> Exp {
    method l -> Exp { a }
    method r -> Exp { b }
    method eval -> Value { l.eval + r.eval }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"
// prints:    an object = 7
```

Oliveira and Cook's code

from their paper, translated into Grace

exp_base.grace

```
dialect "staticTypes"

type Value = Object
type Exp = { eval −> Value }

factory method lit(i:Number) −> Exp {
    method x −> Number { i }
    method eval −> Value { x }
}
factory method sum(a:Exp, b:Exp) −> Exp {
    method l −> Exp { a }
    method r −> Exp { b }
    method eval −> Value { l.eval + r.eval }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"
// prints:    an object = 7
```

```
$ apbmg exp_base.grace
self.sum[0x0x7fc6cbc1b9f8] = 7
$
```

Graceful solution

```
dialect "staticTypes"
import "exp_base" as baseExp
type Exp = baseExp.Exp & type { pretty -> String }

factory method lit(i:Number) -> Exp {
    inherits baseExp.lit(i)
    method pretty { x.asString }
}
factory method sum(a:Exp, b:Exp) -> Exp {
    inherits baseExp.sum(a, b)
    method pretty { "{l.pretty} + {r.pretty}" }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour.pretty} = {threePlusFour.eval}"
// prints:    3 + 4 = 7
```

exp+pretty.grace

Graceful solution

```
dialect "staticTypes"
import "exp_base" as baseExp
type Exp = baseExp.Exp & type { pretty -> String }

factory method lit(i:Number) -> Exp {
    inherits baseExp.lit(i)
    method pretty { x.asString }
}
factory method sum(a:Exp, b:Exp) -> Exp {
    inherits baseExp.sum(a, b)
    method pretty { "{l.pretty} + {r.pretty}" }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour.pretty} = {threePlusFour.eval}"
// prints:    3 + 4 = 7
```

exp+pretty.grace

```
$ apbmg exp+pretty.grace
self.add[0x0x7f9d40523c58] = 7
3 + 4 = 7
$
```

Oliveira and Cook's Example

exp and pretty		Operations	
		eval	pretty
Representations	lit(n)	n	"{n}"
	sum(e_1, e_2)	e_1.eval + e_2. eval	"{e_1.pretty} + {e_2.pretty}"

Portland State UNIVERSITY

Oliveira and Cook's Example

exp+pretty+bool		Operations	
		eval	pretty
Representations	lit(n)	n	"{n}"
	sum(e_1, e_2)	e_1.eval + $e_{2.}$ eval	"{e_1.pretty} + {e_2.pretty}"
	bool(b)	b	"{b}"
	iff(c, th, el)	if(c.eval)then {th.eval) else {el.eval}	"if {c.pretty} then {th.pretty} else {el.pretty}"

exp+pretty+bool.grace

```
dialect "staticTypes"
import "exp_and_pretty" as baseExp
type Exp = baseExp.Exp
type Value = Object

method sum(l:Exp, r:Exp) —> Exp { baseExp.sum(l, r) }
method lit(x:Number) —> Exp { baseExp.lit(x) }

factory method bool(b:Boolean) —> Exp {
    method x —> Boolean { b }
    method eval —> Value { x }
    method pretty —> String { b.asString }
}
factory method iff(c:Exp, t:Exp, f:Exp) —> Exp {
    method eval —> Value {
        if (c.eval) then { t.eval } else { f.eval }
    }
    method pretty —> String {
        "if ({c.pretty}) then {t.pretty} else {f.pretty}"
    }
}

def e3plus4:Exp = sum(lit 3, lit 4)
def e2plus6:Exp = sum(lit 2, lit 6)
def ett:Exp = bool(true)
def ifExpr:Exp = iff(ett, e3plus4, e2plus6)
print "{ifExpr.pretty} = {ifExpr.eval}"
// prints:    if (true) then 3 + 4 else 2 + 6 = 7
```

exp+pretty+bool.grace

```
dialect "staticTypes"
import "exp_and_pretty" as baseExp
type Exp = baseExp.Exp
type Value = Object

method sum(l:Exp, r:Exp) —> Exp { baseExp.sum(l, r) }
method lit(x:Number) —> Exp { baseExp.lit(x) }

factory method bool(b:Boolean) —> Exp {
    method x —> Boolean { b }
    method eval —> Value { x }
    method pretty —> String { b.asString }
}
factory method iff(c:Exp, t:Exp, f:Exp) —> Exp {
    method eval —> Value {
        if (c.eval) then { t.eval } else { f.eval }
    }
    method pretty —> String {
        "if ({c.pretty}) then {t.pretty} else {f.pretty}"
    }
}

def e3plus4:Exp = sum(lit 3, lit 4)
def e2plus6:Exp = sum(lit 2, lit 6)
def ett:Exp = bool(true)
def ifExpr:Exp = iff(ett, e3plus4, e2plus6)
print "{ifExpr.pretty} = {ifExpr.eval}"
// prints:    if (true) then 3 + 4 else 2 + 6 = 7
```

```
…
3 + 4 = 7
if (true) then 3 + 4 else 2 + 6 = 7
$
```

Object Algebras

- Oliveira and Cook. "Extensibility for the masses". ECOOP 2012
  - Avoids typing issues (beyond type parameters) and permits re-use of creation code.
  - Basic idea: abstract over creation by defining a method that builds the structure on demand
  - Argument to that method is the "Object Algebra" — a factory object

```
dialect "staticTypes"
import "exp_base" as exp
type Exp = exp.Exp

// define the Object Algebra machinery
type IntAlg<A> = {
    lit(x:Number) —> A
    sum(e1:A, e2:A) —> A
}
factory method intFactory —> IntAlg<Exp> {
    method lit(x:Number) —> Exp { exp.lit(x) }
    method sum(a:Exp, b:Exp) —> Exp { exp.sum(a, b) }
}
method mk3Plus4<A>(v:IntAlg<A>) —> A {
    v.sum(v.lit(3), v.lit(4))
}
// compare the above with the normal expression:
// def e3Plus4:Exp = sum(lit 3, lit 4)
```

objectAlgerbra.grace (2)

```
// add pretty-printing to expressions "retroactively"
type Pretty = { pretty -> String }
factory method prettyFactory -> IntAlg<Pretty> {
    factory method lit(x:Number) {
        method pretty -> String { x.asString }
    }
    factory method sum(a:Pretty, b:Pretty) {
        method pretty -> String { "{a.pretty} + {b.pretty}" }
    }
}

// demonstration
def x = mk3Plus4(intFactory)
// print "{x.pretty} = {x.eval}"
//   fails: no method 'pretty' in object x
def s = mk3Plus4(prettyFactory)
// print "{s.pretty} = {s.eval}"
//   fails: no method 'eval' in object s
print "{s.pretty} = {x.eval}"
//   prints:    3 + 4 = 7
```

objectAlgerbra.grace (2)

```
// add pretty–printing to expressions "retroactively"
type Pretty = { pretty –> String }
factory method prettyFactory –> IntAlg<Pretty> {
    factory method lit(x:Number) {
        method pretty –> String { x.asString }
    }
    factory method sum(a:Pretty, b:Pretty) {
        method pretty –> String { "{a.pretty} + {b.pretty}" }
    }
}

// demonstration
def x = mk3Plus4(intFactory)
// print "{x.pretty} = {x.eval}"
//   fails: no method 'pretty' in object x
def s = mk3Plus4(prettyFactory)
// print "{s.pretty} = {s.eval}"
//   fails: no method 'eval' in object s
print "{s.pretty} = {x.eval}"
//   prints:    3 + 4 = 7
```

```
...
3 + 4 = 7
$
```

Independent Extensibility

In real life, a much more common scenario than Fig. 1 followed by Fig. 2 followed by Fig. 3 would be like this. Some party A defines *exp_and_pretty*. Another party B independently defines *exp_and_bool*. A third party C finds those and wants to combine them to *exp_and_pretty_and_bool*. This should be possible so that C need only define pretty for bool (in addition to importing the two previous modules). Can Grace handle that?

- Adding *pretty* uses inheritance, while adding *bool* uses composition.
  - If both the original extensions used inheritance, we couldn't guarantee that we could combine them

Independent Extensibility

In real life, a much more common scenario than Fig. 1 followed by Fig. 2 followed by Fig. 3 would be like this. Some party A defines *exp_and_pretty*. Another party B independently defines *exp_and_bool*. A third party C finds those and wants to combine them to *exp_and_pretty_and_bool*. This should be possible so that C need only define pretty for bool (in addition to importing the two previous modules). Can Grace handle that?

Yes!

- But solution is not fully general

- Adding *pretty* uses inheritance, while adding *bool* uses composition.
  - If both the original extensions used inheritance, we couldn't guarantee that we could combine them

Conclusions

- Wadler's version of the expression problem is unsolvable
- Wadler saw it as a challenge for type systems
- I see it as a challenge for even more fundamental features of a language:
  - global constants *vs* local namespaces
  - presence of built-in "non-objects",
  - client object creation with method request or primitive