Object Grammars

Compositional & Bidirectional Mapping Between Text and Graphs

Tijs van der Storm, William R. Cook, Alex Loh
Don't Design Your Programs, Program Your Designs

http://www.enso-lang.org/
Models
Text to objects and back

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
Object Grammars

- Interleave grammar with data binding
  - object construction
  - field assignment
  - predicates
- Bind to paths in to create cross references
- Formatting hints to guide pretty printing
Points

P ::= [Point] "(" x:int "," y:int ")"
Points

P ::= [Point] "(" x:int "," y:int ")"
Points

P ::= [Point] "(" x:int "," y:int ")"
Points

P ::= [Point] "(" x:int "," y:int ")"

Constructor  Field binding  Built-in primitives
The most fundamental feature of Object Grammars is the ability to declaratively construct objects. Each alternative in a production can construct an appropriate object. The following example defines a production rule named `Point` which parses the standard notation for points.

```
P ::= [Point] "(" x:int "," y:int ")"
```

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing. For example, the above grammar can be rewritten into BNF. Object Grammar extend BNF with constructs to declaratively construct objects, backus-naur form (EBNF) [41], which integrated regular iteration and optional symbols into syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) [41].

For example, the schema for points is:

```java
class Point  x: int  y: int
```

The schema

```
P = (x, y)
```

Constructor

Field binding

Built-in primitives
The last alternative does not construct an object, but instead returns the value created by

The most fundamental feature of Object Grammars is the ability to declaratively construct

Each alternative in a production can construct an appropriate object. The following

2.2 Alternatives and Object-Valued Fields

2.1 Construction and field binding

Selections that format the

literal symbols are copied directly to the output. The field assignments are treated as

forms. The constructor acts as a guard that specifies that points should be rendered. The

points.

Any pattern in a grammar can be refactored to introduce new non-terminals without

effects on the result of parsing. For example, the above grammar can be rewritten

must be defined in a

to integers extracted from the input steam. The classes and fields used in a grammar

creates a

cartesian points and creates a corresponding

example defines a production rule named

Point x: int y: int

Point

constructor

field binding

Built-in primitives

P ::= [Point] "(" x:int "," y:int ")"

The schema

class Point x: int y: int
The last alternative does not construct an object, but instead returns the value created by the nested example constructs either a constant, or one of two different kinds of objects. Each alternative in a production can construct an appropriate object. The following selections that format the field assignments are treated as literal symbols are copied directly to the output. The constructor acts as a guard that specifies that points should be rendered. The points.

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing. For example, the above grammar can be rewritten to integers extracted from the input steam. The classes and fields used in a grammar must be defined in a class that parses the standard notation for cross links and evaluate predicates.

The Object Grammars given above can also be used to format points into textual form. The most fundamental feature of Object Grammars is the ability to declaratively construct objects and assign their fields with values taken from the input stream. The following syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) which integrated regular iteration and optional symbols into BNF. Object Grammar extend BNF with constructs to declaratively construct objects, syntax and object graphs.

Backus-Naur Form (EBNF) [41], which integrated regular iteration and optional symbols.

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing. For example, the above grammar can be rewritten to integers extracted from the input steam. The classes and fields used in a grammar must be defined in a class that parses the standard notation for cross links and evaluate predicates.

The Object Grammars given above can also be used to format points into textual form. The most fundamental feature of Object Grammars is the ability to declaratively construct objects and assign their fields with values taken from the input stream. The following syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) which integrated regular iteration and optional symbols into BNF. Object Grammar extend BNF with constructs to declaratively construct objects, syntax and object graphs.

2.1 Construction and field binding

2.2 Alternatives and Object-Valued Fields

The schema for points is:

```
Point x: int y: int
```

The constructor for the point is:

```
P ::= [Point] "(" x:int "," y:int ")"
```

The class Point is:

```
class Point x: int y: int
```
The production rule begins with a

Each alternative in a production can construct an appropriate object. The following

2.2 Alternatives and Object-Valued Fields

The Object Grammars given above can also be used to format points into textual

Any pattern in a grammar can be refactored to introduce new non-terminals without

2.1 Construction and field binding

The syntax and object graphs. The syntactic structure is specified using a form of Extended

Backus-Naur Form (EBNF) [41], which integrated regular iteration and optional symbols


The schema

class Point x: int y: int

Points

P ::= [Point] "(" x:int "," y:int ")"
The last alternative does not construct an object, but instead returns the value created by
The production rule begins with a
The most fundamental feature of Object Grammars is the ability to declaratively construct
2.2 Alternatives and Object-Valued Fields
each alternative in a production can construct an appropriate object. The following
2.1 Construction and field binding
selections that format the
Expression

| [Const] value:int
| "(" Exp ")"

class Exp
class Binary < Exp
  op: str
  lhs: Exp
  rhs: Exp
class Const < Exp
  value: int

The literals
and
fields of the point as integers.

The regular repetition grammar operator
The net effect is that the necessary parentheses are added automatically, to format as
The first case for
This grammar is not very useful, because it is ambiguous. To resolve this ambiguity, we
This grammar refactoring is independent of the schema for expressions; the additional
Object Grammars support regular symbols to automatically map collections of values.

2.3 Collections
also does not match, so parentheses are added and the expression is formatted as a
operator does not match, so it formats the second alternative,
found. For example, to format
allow ambiguous grammars: as long as individual input strings are not ambiguous there
non-terminals.
use the standard technique for encoding precedence and associativity using additional
Prim ::= [Const] value: int
Fact ::= [Binary] lhs:Fact op:"*" rhs:Fact | Fact
Term ::= [Binary] lhs:Term op:"+" rhs:Fact | Fact

| [Const] value:int
| "(" Exp ")"

class Exp
class Binary < Exp
  op: str
  lhs: Exp
  rhs: Exp
class Const < Exp
  value: int

The Object Grammars given above can also be used to format points into textual
The constructor acts as a guard that specifies that points should be rendered. The
points.

During formatting, the alternatives are searched in order until a matching case is
 Prim ::= [Const] value: int
Fact ::= [Binary] lhs:Fact op:"*" rhs:Fact | Fact
Term ::= [Binary] lhs:Term op:"+" rhs:Fact | Fact

| [Const] value:int
| "(" Exp ")"
Expressions

Both + and * become Binary objects

\[
\text{Exp} ::= [\text{Binary}] \text{lhs:Exp op":"+" rhs:Exp} \\
| [\text{Binary}] \text{lhs:Exp op":"*" rhs:Exp} \\
| [\text{Const}] \text{value:int} \\
| "(" \text{Exp ")"}
\]

\begin{align*}
\text{class Exp} \\
\text{class Binary < Exp} & \quad \text{op: str} \\
& \quad \text{lhs: Exp} \\
& \quad \text{rhs: Exp} \\
\text{class Const < Exp} & \quad \text{value: int}
\end{align*}
Expressions

Both + and * become Binary objects

|   [Const] value:int
|   "(" Exp ")"

Parentheses don’t introduce objects

class Exp
class Binary < Exp
   op: str
   lhs: Exp
   rhs: Exp
class Const < Exp
   value: int
Expressions

Refactored grammar for disambiguation

Term ::= [Binary] lhs:Term op:"+" rhs:Fact |
          Fact
Fact ::= [Binary] lhs:Fact op:"*" rhs:Prim |
          Prim
Prim ::= [Const] value:int |
       "(" Term ")"

class Exp
  class Binary < Exp
    op: str
    lhs: Exp
    rhs: Exp
  class Const < Exp
    value: int
State machines

**start** Opened

**state** Opened

  **on** close **go** Closed

**state** Closed

  **on** open **go** Opened

  **on** lock **go** Locked

**state** Locked

  **on** unlock **go** Closed

---

- **start** Opened
- **state** Opened
  - **on** close **go** Closed
- **state** Closed
  - **on** open **go** Opened
  - **on** lock **go** Locked
- **state** Locked
  - **on** unlock **go** Closed
The object grammar

M ::= [Machine] "start" start:</states[it]> states:S*

S ::= [State] "state" name:sym out:T*

T ::= [Transition] "on" event:sym "go" to:</states[it]>
Creating the spine

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```

![Diagram](image-url)
Creating the spine

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```
Creating the spine

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```
Creating the spine

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
Creating the spine

```
start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed
```

Fig. 1. (a) Example state machine in graphical notation, (b) the state machine in textual notation, and (c) the internal representation of the state machine in object diagram notation.
Creating the spine

**start** Opened

**state** Opened
  - **on** close **go** Closed

**state** Closed
  - **on** open **go** Opened
  - **on** lock **go** Locked

**state** Locked
  - **on** unlock **go** Closed

---

**Machine**

- **states**

- **:State**
  - name: "Opened"

- **:State**
  - name: "Closed"

- **:State**
  - name: "Locked"

---

**Transition**

- event: "open"

- event: "unlock"

- event: "lock"

---

**Transition**

- event: "close"
Creating the spine

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed

Machine
  states
    :State
      name: "Opened"
    :State
      name: "Closed"
    :State
      name: "Locked"

Transition
  event: "open"
  from
  to
  out

Transition
  event: "close"
  from
  to
  out

Transition
  event: "lock"
  from
  to
  out

Transition
  event: "unlock"
  from
  to
  out
Creating the spine

start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed

:Machine
  states

:State
  name: "Opened"
  from
  out
  :Transition
  event: "open"

:State
  name: "Closed"
  from
  out
  :Transition
  event: "unlock"

:State
  name: "Locked"
  from
  out
  :Transition
  event: "lock"
Creating the spine

start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed

(machine)

:State
  name: "Opened"
 :State
  name: "Closed"
 :State
  name: "Locked"

:Transition
  event: "open"
 :Transition
  event: "unlock"
 :Transition
  event: "lock"
Cross links

```
start  Opened

state  Opened
   on close go Closed

state  Closed
   on open go Opened
   on lock go Locked

state  Locked
   on unlock go Closed
```
Cross links

![Cross links diagram]

start  Opened

state  Opened

  on close go Closed

state  Closed

  on open go Opened

  on lock go Locked

state  Locked

  on unlock go Closed
Cross links

start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed
Cross links

start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed
Cross links

- **start** 0pened
- **state** 0pened
  - on close go Closed
- **state** Closed
  - on open go 0pened
  - on lock go Locked
- **state** Locked
  - on unlock go Closed
**Cross links**

Diagram showing a state machine with three states: `Opened`, `Closed`, and `Locked`. The states are connected by transitions:

- **Start State**: `Opened`
- **Transition**:
  - `on close` to `Closed`
  - `on open` to `Opened`
  - `on lock` to `Locked`
- **Locked State**
  - `on unlock` to `Closed`
Object path to find the start state with name it

M ::= [Machine] "start" \start:@/states[it] states:S*

S ::= [State] "state" name:sym out:T*

T ::= [Transition] "on" event:sym "go" to:@/states[it]
Paths

- Navigate the resulting model along
  - Fields
  - Collections (keyed, positional)
- NB: model may not be finished yet
  - Paths may traverse *cross links* too
- Iterative fix point
A path

/start at the root
/navigate into states
/use the parsed identifier as key

/\texttt{states\{it\}}

Paths can also start at current object (.) or parent (..)
Creating cross links

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```
Creating cross links

start **Opened**
state **Opened**
  on close go **Closed**
state **Closed**
  on open go **Opened**
  on lock go **Locked**
state **Locked**
  on unlock go **Closed**
Creating cross links

```
start (Opened) [start:=</states["Opened"]>
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```
Creating cross links

\[
\begin{align*}
\text{start} & \quad \text{Opened} & \text{start:} & \text{states["Opened"]}\quad \\
\text{state} & \quad \text{Opened} & \quad \text{on } & \text{close } \text{go } \text{Closed} \\
\text{state} & \quad \text{Closed} & \quad \text{on } & \text{open } \text{go } \text{Opened} \\
\text{state} & \quad \text{Locked} & \quad \text{on } & \text{unlock } \text{go } \text{Closed}
\end{align*}
\]
Creating cross links

**start** **Opened**

**state** **Opened**

  **on** close **go** **Closed**

**state** **Closed**

  **on** open **go** **Opened**

  **on** lock **go** **Locked**

**state** **Locked**

  **on** unlock **go** **Closed**
Creating cross links

start (Opened) start: /states["Opened"]
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
Creating cross links

start(Openened)  start:</states[“Openened”]>
state Openened
  on close go Closed
state Closed
  on open go Openened
  on lock go Locked
state Locked
  on unlock go Closed
Creating cross links

\[
\begin{align*}
\text{start} & \quad \text{Opened} \\
\text{state} & \quad \text{Opened} \\
& \quad \text{on close go} \quad \text{Closed} \\
\text{state} & \quad \text{Closed} \\
& \quad \text{on open go} \quad \text{Opened} \\
& \quad \text{on lock go} \quad \text{Locked} \\
\text{state} & \quad \text{Locked} \\
& \quad \text{on unlock go} \quad \text{Closed}
\end{align*}
\]
Creating cross links

start (Opened) to: states[“Opened”]
state 0pened on close go (Closed) to: states[“Closed”]
state Closed on open go Opened on lock go Locked
state Locked on unlock go Closed
Creating cross links

start (Opened)
state 0pened
  on close go Closed
to: </states[“Closed”]>
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed

start:
start: </states[“Opened”]>

Start:
states

Machine

:State
name: "Opened"

:State
name: "Closed"

:State
name: "Locked"

:Transition
event: "open"

:Transition
event: "unlock"

:Transition
event: "lock"

:Transition
event: "close"
Creating cross links

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```
Creating cross links

\[
\text{start Opened} \quad \text{state Opened} \quad \text{on close go Closed} \quad \text{state Closed} \quad \text{on open go Opened} \quad \text{on lock go Locked} \quad \text{state Locked} \quad \text{on unlock go Closed}
\]
Creating cross links

\[\begin{align*}
\text{start} & \quad \text{Opened} \\
\text{state} & \quad \text{Opened} \\
& \quad \text{on close go } \text{Closed} \\
\text{state} & \quad \text{Closed} \\
& \quad \text{on open go } \text{Opened} \\
& \quad \text{on lock go } \text{Locked} \\
\text{state} & \quad \text{Locked} \\
& \quad \text{on unlock go } \text{Closed}
\end{align*}\]
Creating cross links

start **Opened**
state **Closed**
on close go **Closed**
on open go **Opened**
on lock go **Locked**
state **Locked**
on unlock go **Closed**

start:\states["Opened"]
state **Closed**
on close go **Closed**
on open go **Opened**
on lock go **Locked**
state **Locked**
on unlock go **Closed**
Creating cross links

start Opened
state Opened
  on close go Closed
te:/states["Closed"]
state Closed
  on open go Opened
to:/states["Opened"]
on lock go Locked
state Locked
  on unlock go Closed
Creating cross links

```
start Opened
state Opened
  on close go Closed
to: </states[“Closed”]>
state Closed
  on open go Opened
to: </states[“Opened”]>
on lock go Locked
state Locked
  on unlock go Closed
```
Creating cross links

\begin{itemize}
\item \texttt{start} \texttt{(Opened)}
\item \texttt{state} \texttt{Opened}
  \item \texttt{on} close \texttt{go} \texttt{(Closed)}
\item \texttt{state} \texttt{Closed}
  \item \texttt{on} open \texttt{go} \texttt{(Opened)}
  \item \texttt{on} lock \texttt{go} \texttt{(Locked)}
\item \texttt{state} \texttt{Locked}
  \item \texttt{on} unlock \texttt{go} \texttt{Closed}
\end{itemize}
Creating cross links

**start** Opened

**state** Opened
  **on** close go **Closed**

**state** Closed
  **on** open go **Opened**
  **on** lock go **Locked**

**state** Locked
  **on** unlock go Closed
Creating cross links

start **Opened**

state **Opened**

on close go **Closed**

state **Closed**

on open go **Opened**

on lock go **Locked**

state **Locked**

on unlock go **Closed**
Creating cross links

```

start (Opened)  \rightarrow \text{start:<\text{states["Opened"]>}}
state  Opened
on close go (Closed)  \rightarrow \text{to:<\text{states["Closed"]>}
state  Closed
on open go (Opened) \rightarrow \text{to:<\text{states["Opened"]>}
on lock go (Locked) \rightarrow \text{to:<\text{states["Locked"]>}
state  Locked
on unlock go Closed
```
Creating cross links

start (Opened)  
state Closed  
on close go (Closed)  
on open go (Opened)  
on lock go (Locked)  
state Locked  
on unlock go Closed

start: </states[“Opened”]>  
states

start

:State
name: "Opened"
:Transition
event: "open"

:State
name: "Closed"
:Transition
event: "unlock"

:State
name: "Locked"
:Transition
event: "lock"

:Machine
machine

states

states

states

states
Creating cross links

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed

start:\</states["Opened"]>
Creating cross links

start (Opened)
state 0pened
  on close go (Closed)
state Closed
  on open go (Opened)
  on lock go (Locked)
state Locked
  on unlock go (Closed)
Creating cross links

start (Opened)  
state  Opened  
on close go (Closed)  
state  Closed  
on open go (Opened)  
on lock go (Locked)  
state  Locked  
on unlock go (Closed)  

This is illustrated in the object diagram given in Fig. 1(c).

The regular repetition grammar operator field indicates that the case for the field as many-valued. The # annotation marks a field as a primary key, as is the a type, and some optional modifiers. For example, the schema consists of a list of named classes, each having a list of fields defined by a name, the machine itself, its states and the transitions are all represented explicitly as objects. The same state machine is rendered textually in Fig. 1(b). Internally, to introduce a slightly more elaborate example. Consider a small DSL for modeling state machines. Figure 1 displays three representations of a simple state machine representing

In order to explain path-based reference resolution in Object Grammars, it is instructive to the formatter to perform intelligent grouping and indentation. A repeated group is either bound multiple times, rather than using the created by zero-or-more occurrences of . A collection field can also be explicitly replaced by .

For formatting, the regular operators may be optionally followed by a separator . A collection field can also be explicitly provide additional semantics, allowing . The annotation after the field of the State class. As a result, state names must be unique and inverses can be indexed by name. The states are the States. The states are the States. The States. The States. The States. The States. The States.
Creating cross links

start: \textit{Opened} \quad \text{start}: \langle \text{states}[^{\text{"Opened"}}]\rangle

\textbf{state} \textit{Opened}

\textbf{on} close \textbf{go} \textit{Closed} \quad \text{to}: \langle \text{states}[^{\text{"Closed"}}]\rangle

\textbf{state} \textit{Closed}

\textbf{on} open \textbf{go} \textit{Opened} \quad \text{to}: \langle \text{states}[^{\text{"Opened"}}]\rangle

\textbf{on} lock \textbf{go} \textit{Locked} \quad \text{to}: \langle \text{states}[^{\text{"Locked"}}]\rangle

\textbf{state} \textit{Locked}

\textbf{on} unlock \textbf{go} \textit{Closed} \quad \text{to}: \langle \text{states}[^{\text{"Closed"}}]\rangle
Creating cross links

start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed

start:<states["Opened"]>
next:<states["Closed"]>
next:<states["Locked"]>

:Machine
  states
  :State
    name: "Opened"
    start
    transit: "close"
    to:<states["Closed"]>
  :State
    name: "Locked"
    transit: "lock"
    out
    :State
    name: "Closed"
    transit: "unlock"
Assessment

- Bi-directional & compositional
- Flexible:
  - interleaved data binding
  - path-based references & predicates
  - formatting hints
- Self-described
Fig. 11. Language composition in Ensō. Each arrow $A \rightarrow B$ indicates an invocation of $B$. The arrow points in the direction of the result. For instance, the Stencil and Web languages are, independently, merged into the Command language. As a result both Stencil and Web include, and possibly override and/or extend the Command language. If a language reuses or extends multiple other languages, the merge operator is applied in sequence.

For instance, Grammar is first merged into Path, and then merged into Expr.

The core languages in Ensō include both the Schema and Grammar languages, as well as Stencil, a language to define graphical model editors. Additionally, Ensō features a small set of library languages that are not deployed independently but reused in other languages. An example of a library language is Expr, an expression language with operators, variables and primitive values. It is, for instance, reused in Grammar for predicates and in Schema for computed fields. Command is a control-flow language that captures loops, conditional statements and functions. The Command language reuses the Expr language for the guards in loops and conditional statements. Another example is the language of paths (Path), shown in Fig. 4, which provides a model to address nodes in object graphs.

The reuse of Expr and Path are examples of a simple embedding. The languages are reused as black boxes, without modification. The composition of Command with Stencil and Web, however is different. Stencil is created by adding language constructs for user-interface widgets, lines, and shapes to the Command language as valid primitives. The Command language can now be used to create diagrams. A similar extension is realized in the Web language: here a language for XML element structure is mixed with the statement language of Web. The extension works in both directions: XML elements are valid statements, statements are valid XML content. The Piping and Controller languages are from a domain-specific modeling case-study in the domain of piping and instrumentation for the Language Workbench Challenge 2012 [25]. Fig. 11 only shows the Controller part which reuses Expr.

An overview of the number source lines of code (SLOC) is shown in Table 1(a). We show the number for the full languages in Ensō as well as the reused language modules (Path, Command, Expr and XML). A language consists of a schema, a grammar and an interpreter. The interpreters are all implemented in Ruby. Table 1(b) shows the reuse percentage for each language [17]. This percentage is computed as $\frac{\#SLOC \text{ reused}}{\#SLOC \text{ total}} \times 100$. Which languages are reused in each case can be seen from Fig. 11. As can be seen from this table, the amount of reuse in schemas and grammars is consistently high, with the exception of the Piping language, which does not reuse any.
Conclusion

- Object grammars: mapping text to objects and vice versa
- Declarative paths for resolving cross-references
- Flexible, bi-directional and compositional
- Foundation of Ensō
Don't Design Your Programs, Program Your Designs

http://www.enso-lang.org/