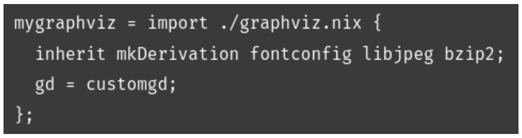
Section title

This is a placeholder for writing contents

- REFs 2x2 subfig, color, 한글 original templates 여백설정, 페이지 넘김

Image

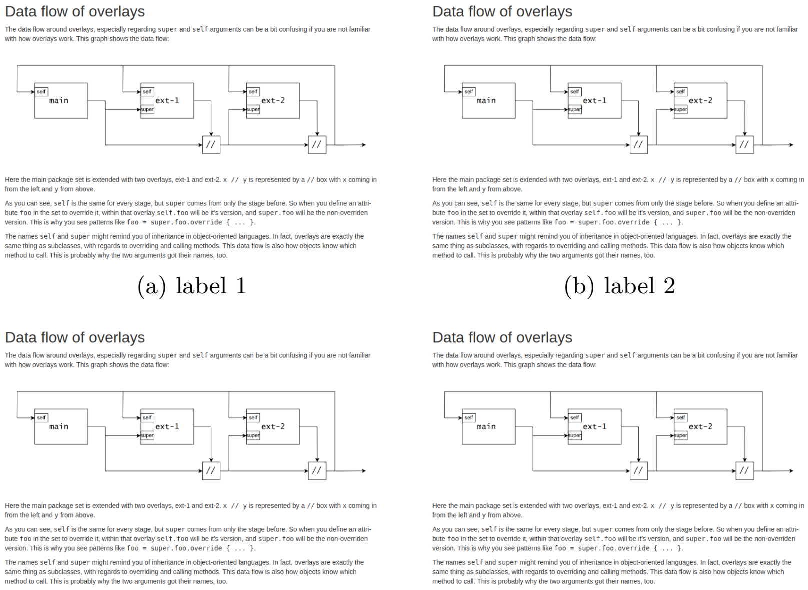
This is an how we can refer to an image, see figure 1.



```
mygraphviz = import ./graphviz.nix {
  inherit mkDerivation fontconfig libjpeg bzip2;
  gd = customgd;
};
```

Figure 1: Leopard icon

There are other ways of showing sub-images and display sub-captions like using in latex, see figure 2



Data flow of overlays

The data flow around overlays, especially regarding super and self arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:

Here the main package set is extended with two overlays, ext-1 and ext-2. x // y is represented by a // box with x coming in from the left and y from above.

As you can see, self is the same for every stage, but super comes from only the stage before. So when you define an attribute foo in the set to override it, within that overlay self.foo will be it's version, and super.foo will be the non-overriden version. This is why you see patterns like foo = super.foo.override { ... }.

The names self and super might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably why the two arguments got their names, too.

(a) label 1

Data flow of overlays

The data flow around overlays, especially regarding super and self arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:

Here the main package set is extended with two overlays, ext-1 and ext-2. x // y is represented by a // box with x coming in from the left and y from above.

As you can see, self is the same for every stage, but super comes from only the stage before. So when you define an attribute foo in the set to override it, within that overlay self.foo will be it's version, and super.foo will be the non-overriden version. This is why you see patterns like foo = super.foo.override { ... }.

The names self and super might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably why the two arguments got their names, too.

(b) label 2

Data flow of overlays

The data flow around overlays, especially regarding super and self arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:

Here the main package set is extended with two overlays, ext-1 and ext-2. x // y is represented by a // box with x coming in from the left and y from above.

As you can see, self is the same for every stage, but super comes from only the stage before. So when you define an attribute foo in the set to override it, within that overlay self.foo will be it's version, and super.foo will be the non-overriden version. This is why you see patterns like foo = super.foo.override { ... }.

The names self and super might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably why the two arguments got their names, too.

(c) label 3

Data flow of overlays

The data flow around overlays, especially regarding super and self arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:

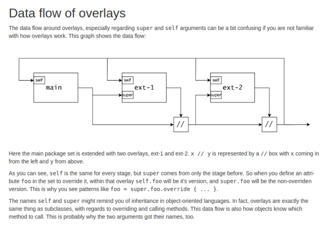
Here the main package set is extended with two overlays, ext-1 and ext-2. x // y is represented by a // box with x coming in from the left and y from above.

As you can see, self is the same for every stage, but super comes from only the stage before. So when you define an attribute foo in the set to override it, within that overlay self.foo will be it's version, and super.foo will be the non-overriden version. This is why you see patterns like foo = super.foo.override { ... }.

The names self and super might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably why the two arguments got their names, too.

(d) label 4

Figure 2: figures with captions



Data flow of overlays

The data flow around overlays, especially regarding `super` and `self` arguments can be a bit confusing if you are not familiar with how overlays work. This graph shows the data flow:

Here the main package set is extended with two overlays, ext-1 and ext-2. `x // y` is represented by a `//` box with `x` coming in from the left and `y` from above.

As you can see, `self` is the same for every stage, but `super` comes from only the stage before. So when you define an attribute `foo` in the set to override it, within that overlay `self.foo` will be it's version, and `super.foo` will be the non-overriden version. This is why you see patterns like `foo = super.foo.override { ... }`.

The names `self` and `super` might remind you of inheritance in object-oriented languages. In fact, overlays are exactly the same thing as subclasses, with regards to overriding and calling methods. This data flow is also how objects know which method to call. This is probably why the two arguments got their names, too.

this is how to insert figures between paragraphs. asdkfjl akjwe jk ajsk jflkjflkj i akkjdxf l skdjflkj safij kasfl jlkjaij kljwekj jkzxnnlk; wmnzkjojsaijra . jsalkjfpjwet askljlks flk a sdklj asdlkfj jtqwijsf kj kjasj 한글도 사용해보자!! 색까지 넣어서!!

Table

Author	Email	Institution-ID
Gene Ting-Chun Kao	your.email@email.edu	1
Your name		2
another name		3

Section title

Mathematics in latex

Check equation 1.

$$f(x) = s_0 = \frac{\sum\limits_i n_i^T (x - x_i)\Phi_i(x)}{\sum\limits_i \Phi_i(x)} \tag{1}$$

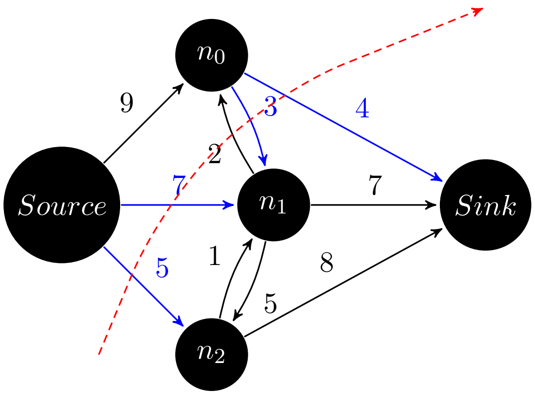
To have a set of equations and to align them:

$$\begin{aligned} \max \quad & \mathbf{c}^T\mathbf{x} \\ s.t. \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2}$$

Graph

Check out the graph in figure 3.

Figure 3: Max flow min cut, max flow = 19



Algorithm

Algorithm 1 How to write algorithms

```
Data: Initial bounding-box $Q_0$ for $\Theta$, $QBest = Q_0$, $delta = 3$, stack $\Omega = \{Q_0\}$
Result: Optimal Q$^* = QBest \in \Omega$
while $U_k - L_k > 1$ do
    Pop $Q_k \in \Omega$
     Prune $\Omega$ if current node is impossible solution node
     Compare $L_k$ from $Q_k$ and $QBest$
     if $Q_k.L_k > QBest.L_k$ then
        $QBest = Q_k$
    end
    Split $Q$ into $Q_I$ and $Q_{II}$
     Find best condidate from $Q_I$ and $Q_{II}$ and add them to stack $\Omega$
end
```

Flowchart

This flowchart in Fig. 4 is modified from this latex code.

Citation

This is how we can cite paper [**?**]

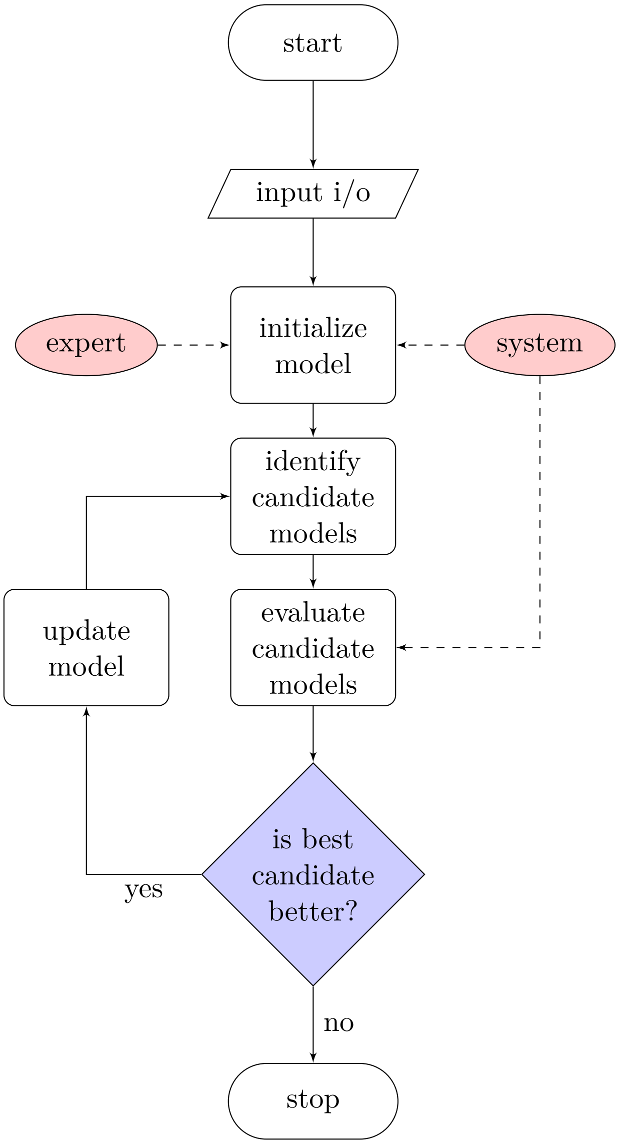

start
input i/o
expert
initialize model
system
identify candidate models
update model
evaluate candidate models
is best candidate better?
yes
no
stop

Figure 4: This is my flow chart