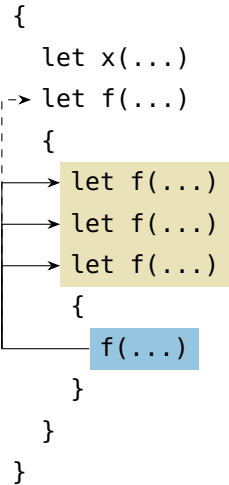
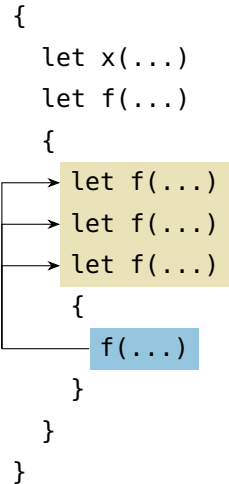
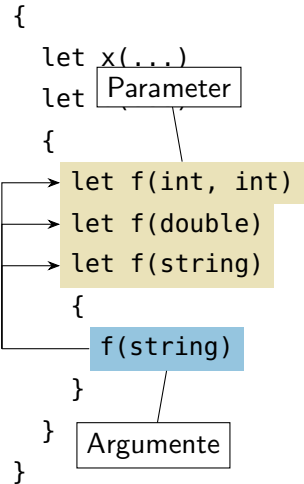


```
{  
  let x(...)  
  let f(...)  
  {  
    let f(...)  
    let f(...)  
    let f(...)  
    {  
      f(...)  
    }  
  }  
}
```







```
{  
  let x(...)  
  let f(...)  
  {  
    let f(int, int)  
    let f(double)  
    let f(string)  
    {  
      f(string)  
    }  
  }  
}
```

The diagram illustrates function overloading. It shows a code snippet with three function signatures: `let f(int, int)`, `let f(double)`, and `let f(string)`. The `let f(double)` and `let f(string)` lines are highlighted in a yellow box. Below the `let f(string)` line, there is a blue box containing `f(string)`. Three arrows originate from the left side of the image: one points to the `let f(double)` line, one points to the `let f(string)` line, and one points to the `f(string)` line in the blue box.

```
{  
  let x(...)  
  let f(...)  
  {  
    let f(int, int)  
    let f(double)  
    let f(string)  
    {  
      f(string)  
    }  
  }  
}
```

The diagram illustrates the resolution of a function call `f(string)` within a nested scope. A horizontal arrow points from the `f(string)` call in the innermost scope to the `let f(string)` definition in the same scope. A vertical line descends from the arrow's start, and another horizontal line connects it to the `let f(string)` definition in the middle scope, indicating that the function is resolved to the definition in the middle scope because the innermost scope does not have a binding for `f`.