INF122 – Functional Programming

Violet Ka I Pun

violet@ifi.uio.no

Today

- What is the course about?
- Practical information
- Overview of programming languages

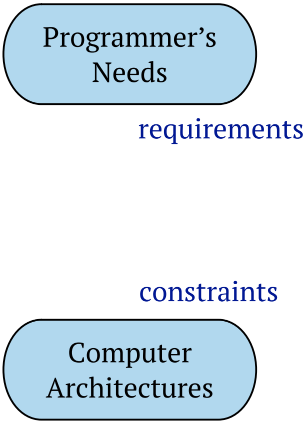
What is the course about?

Programming languages

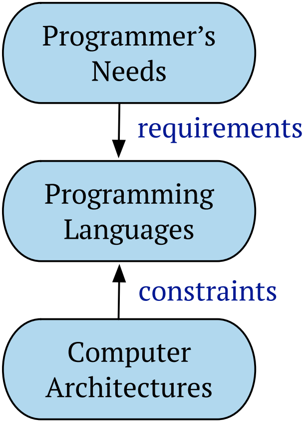
- A means to specify, organise, and reason about computations

Programming languages

- A means to specify, organise, and reason about computations



- A means to specify, organise, and reason about computations



A [edit]

- A# .NET
- A# (Axiom)
- A-0 System
- A+
- A++
- ABAP
- ABC
- ABC ALGOL
- ABSET
- ABSYS
- ACC
- Accent
- Ace DASL
- ACL2
- ACT-III
- Action!
- ActionScript
- Ada
- Adenine
- Agda
- Agilent VEE
- Agora
- AIMMS
- Alef
- ALF
- ALGOL 58
- ALGOL 60
- ALGOL 68
- ALGOL W
- Alice
- Alma-0
- AmbientTalk
- Amiga E
- AMOS
- AMPL
- Apex (Salesforce.com)
- APL
- App Inventor for Android's visual block language
- AppleScript
- Arc
- ARexx
- Argus
- AspectJ
- Assembly language
- ATS
- Ateji PX
- AutoHotkey
- Autocoder
- AutoIt
- AutoLISP / Visual LISP
- Averest
- AWK
- Axum

W [edit]

- WATFIV, WATFOR
- WebDNA
- WebQL
- Whiley
- Windows PowerShell
- Winbatch
- Wolfram Language
- Wyvern

X [edit]

- X++
- X#
- X10
- XBL
- XC (exploits XMOS architecture)
- xHarbour
- XL
- Xojo
- XOTcl
- XPL
- XPL0
- XQuery
- XSB
- XSLT – see XPath
- Xtend

Y [edit]

- Yorick
- YQL

Z [edit]

- Z notation
- ZOPL
- ZPL

Why so many programming languages?

- Why not just make a good language for all kinds of purposes?
- Good reasons that there are many languages:
  - Problems are different in size, complexity, etc,
    and belong to different domains
  - Different requirements to speed, space and security, ...
  - Programmers are different!

What will you learn

- Different way of programming than you have met so far: functional programming
  - In order to choose the language that suits a given problem best
- General mechanisms of most programming languages
  - In order to understand what they can be used for and how they are implemented
  - In order to compare and evaluate (coming) languages
  - In order to be able to design languages

What will you learn

- Overview of programming languages
- Syntax, grammars
- Programming in Haskell

  E.g., Functions, pattern matching, . . .
- Types and classes
- Recursion
- Higher-ordered functions
- Parsing
- I/O
- Hindley-Milners types & unification
- Correctness

Practical Info

Curriculum

- Text book:
  - Graham Hutton:
    Programming in Haskell, 2007.Cambridge University Press.
  - Ravi Sethi:
    Programming Languages, Concepts and Constructs. Second Edition.
  - Additional materials
- Weekly exercises
- Compulsory assignments
  - 2 compulsory assignments

Time plan

Lectures

- Thursdays, 10:15 – 12:00
  Høyteknologisenteret, Stort auditorium
- Fridays, 10:15 – 12:00
  Carl L. Godskes hus, Auditorium 307 ("pi")

Group sessions

Group 1: Mondays, 10:15 – 12:00
Høyteknologisenteret, Datalab 3

Group 2: Thursdays, 8:15 – 10:00
Høyteknologisenteret, Datalab 3

Group 3: TBA
TBA

Contact Info

Violet Ka I Pun

violet@ifi.uio.no

Office: 407P2

Andreas Johnsen Lind

Andreas.Lind@uib.no

Jonas Berdal

jonas.berdal@student.uib.no

Morten Lohne

M.Lohne@student.uib.no

Overview of Programming languages

Why higher-level

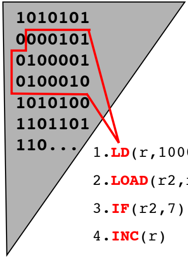
- Machine language is unintelligible
- Assembly language is low level
  - The control is not visible

$\Rightarrow$ Higher-level languages

Abstraction

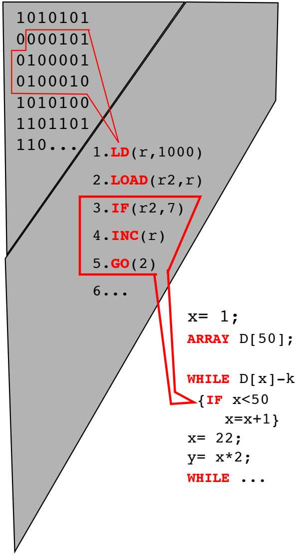
```
1010101
0000101
0100001
0100010
1010100
1101101
110...
```

Abstraction

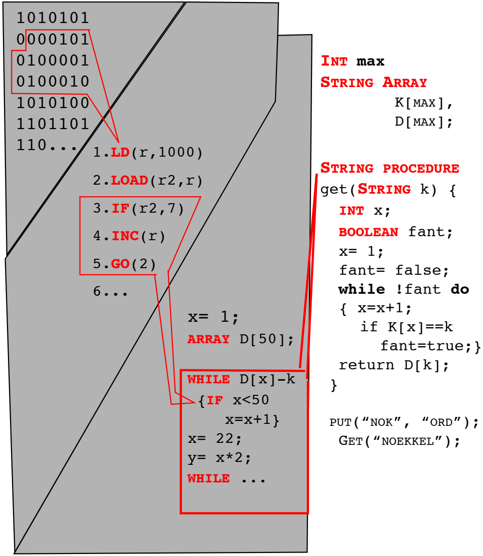


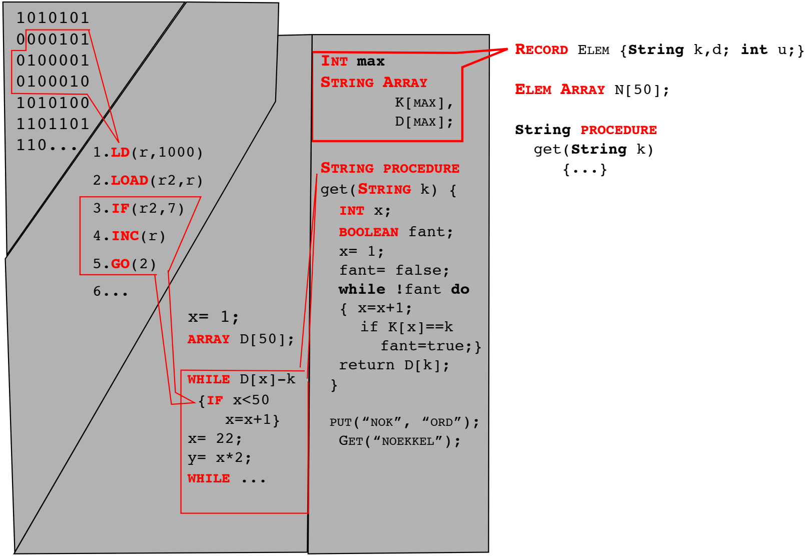

```
1.LD(r,1000)
2.LOAD(r2,r)
3.IF(r2,7)
4.INC(r)
5.GO(2)
6...
```

Abstraction



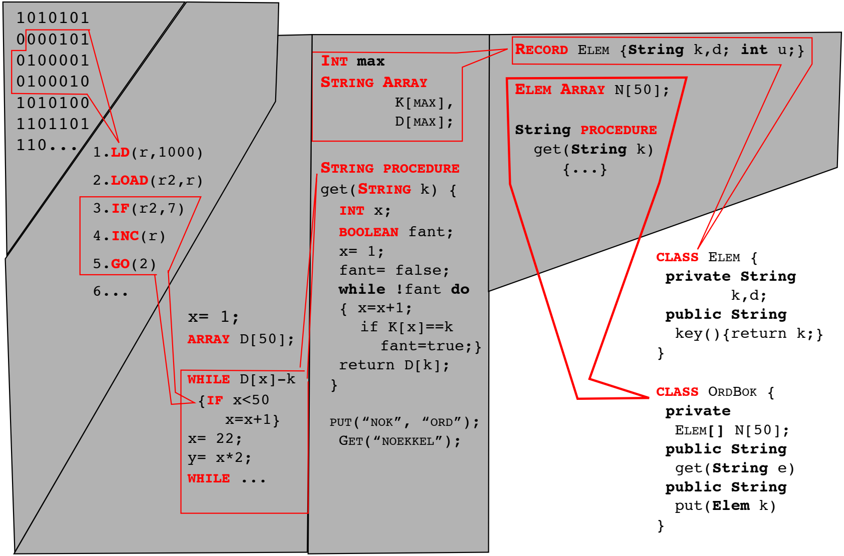
Abstraction



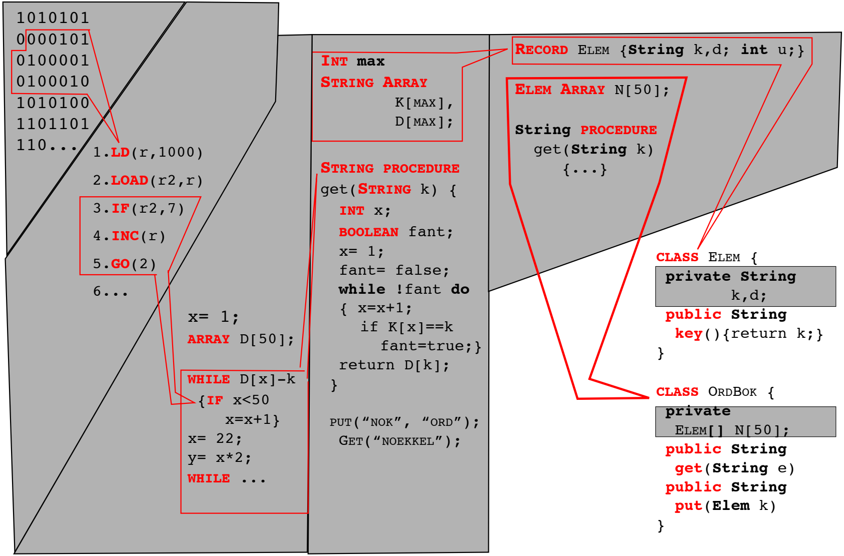


1010101
0000101
0100001
0100010
1010100
1101101
110...
1.LD(r,1000)
2.LOAD(r2,r)
3.IF(r2,7)
4.INC(r)
5.GO(2)
6...
x= 1;
ARRAY D[50];
WHILE D[x]-k
{IF x<50
x=x+1}
x= 22;
y= x*2;
WHILE ...
INT max
STRING ARRAY
K[MAX],
D[MAX];
STRING PROCEDURE
get(STRING k) {
INT x;
BOOLEAN fant;
x= 1;
fant= false;
while !fant do
{ x=x+1;
if K[x]==k
fant=true;}
return D[k];
}
PUT("NOK", "ORD");
GET("NOEKKEL");
RECORD ELEM {String k,d; int u;}
ELEM ARRAY N[50];
String PROCEDURE
get(String k)
{...}

Abstraction



Abstraction



Types

- `int, boolean, char, string, ...`
- Allow programmers to
  - Construct new instances, and
  - Use them through *a given interface*

```
INT x=0, y=5; x=x+2; x=(x % y); y=(y-x); ...
```

```
BOOLEAN a, b; a=x<y; b=(b OR a); b=!b; ...
```

Types

- `int, boolean, char, string, ...`
- Allow programmers to
  - Construct new instances, and
  - Use them through *a given interface*

```
INT x=0, y=5; x=x+2; x=(x % y); y=(y-x); ...
```

1,2,3...	:	$\rightarrow$ Int
_+_	:	INT $\times$ INT $\rightarrow$ INT
_-_	:	INT $\times$ INT $\rightarrow$ INT
_%_	:	INT $\times$ INT $\rightarrow$ INT

```
BOOLEAN a, b; a=x<y; b=(b OR a); b=!b; ...
```

true,false	:	$\rightarrow$ BOOLEAN
_==_	:	INT $\times$ INT $\rightarrow$ BOOLEAN
_<_	:	INT $\times$ INT $\rightarrow$ BOOLEAN
!_	:	BOOLEAN $\rightarrow$ BOOLEAN

Types

- `int`, `boolean`, `char`, `string`, ...
- Allow programmers to
  - Construct new instances, and
  - Use them through *a given interface*

INT `x=0, y=5; x=x+2; x=(x % y); y=(y-x); ...`

1,2,3...	:	$\to$ Int
_+_	:	INT $\times$ INT $\to$ INT
_-_	:	INT $\times$ INT $\to$ INT
_%_	:	INT $\times$ INT $\to$ INT

BOOLEAN `a, b; a=x<y; b=(b OR a); b=!b; ...`

true,false	:	$\to$ BOOLEAN
_==_	:	INT $\times$ INT $\to$ BOOLEAN
_<_	:	INT $\times$ INT $\to$ BOOLEAN
!_	:	BOOLEAN $\to$ BOOLEAN

Without knowing anything about the implementation

User-defined Types

```
PUBLIC CLASS ORDBOK {
  private Elem[] N;
  PUBLIC ORDBOK(int k) { N= new Elem[k]; }
  PUBLIC ELEM GET(STRING e) {...}
  PUBLIC VOID PUT(ELEM k) {...}
}
```

```
  NEW_  :  INT → ORDBOK
_.PUT_  :  ORDBOK × ELEM → ORDBOK
_.GET_  :  ORDBOK × STRING → ELEM
```

User-defined Types

```
PUBLIC CLASS ORDBOK {
  private Elem[] N;
  PUBLIC ORDBOK(int k) { N= new Elem[k]; }
  PUBLIC ELEM GET(STRING e) {...}
  PUBLIC VOID PUT(ELEM k) {...}
}
```

```
   NEW_ : INT → ORDBOK
 _.PUT_ : ORDBOK × ELEM → ORDBOK
 _.GET_ : ORDBOK × STRING → ELEM
```

```
OrdBok ob= NEW ORDBOK(100);
ob.PUT(new Elem("a","aaaaaaa"));
ob.PUT(new Elem("b","bbbb"));
ob.PUT(new Elem("c","cccccc"));
Elem e= ob.GET("b");
```

```
PUBLIC CLASS ORDBOK {
  private Elem[] N;
  PUBLIC ORDBOK(int k) { N= new Elem[k]; }
  PUBLIC ELEM GET(STRING e) {...}
  PUBLIC VOID PUT(ELEM k) {...}
}
```

NEW_	:	INT → ORDBOK
.PUT	:	ORDBOK × ELEM → ORDBOK
.GET	:	ORDBOK × STRING → ELEM

```
OrdBok ob= NEW ORDBOK(100);
ob.PUT(new Elem("a","aaaaaaa"));
ob.PUT(new Elem("b","bbbb"));
ob.PUT(new Elem("c","cccccc"));
Elem e= ob.GET("b");
```

Extend the language

Objected-orientation and typing

Typing

- Requires higher programming discipline
- Allows to discover many errors during compilation
- Increases reliability of software considerably
- Leads to possibilities of
  - Reusing
  - Modularization of software

Objected-orientation and typing

Typing

- Requires higher programming discipline
- Allows to discover many errors during compilation
- Increases reliability of software considerably
- Leads to possibilities of
  - Reusing
  - Modularization of software

A class declaration defines a type, namely

- An interface
- A sort `T`
- An implementation of `T` and the operations

Object-oriented and typing

```
CLASS ELEM
{ PRIVATE String k,d;
 PUBLIC ELEM(String a, String b){ k= a; d= b; }
 PUBLIC STRING KEY()  {return k;}
 PUBLIC STRING DATA() {return d;}
}
```

```
CLASS ORDBOK
{ PRIVATE Elem  N[]; PRIVATE int max;
 PUBLIC ORDBOK(int m)
   { max= m; N= new Elem[max]; }
 PUBLIC VOID PUT(Elem e)
   { max++; N[max]= e;  }
 PUBLIC ELEM GET(String k)
   { for (int i=1; i<=max; i++)
       if (N[i].key()==k) return N[i];
     return null; }
}
```

```
CLASS ORDBOKLS EXTENDS ORDBOK
{ PUBLIC VOID LISTSORTERT()
  {for (int i=1; i<=max; i++)
     for (int j=1; j<=max-i; j++)
      if (N[j].key()>N[j+1].key()) swap(j,j+1)
   for (int i=1; i<=max; i++)
     skriv(N[i]);
} }
```

Object-oriented and typing

```
class Elem
{ private String k,d;
  public Elem(String a, String b) { k= a; d= b; }
  public String Key()  {return k;}
  public String Data() {return d;}
}
```

```
class OrdBok
{ private Elem N[]; private int max;
  public OrdBok(int m)
    { max= m; N= new Elem[max]; }
  public void Put(Elem e)
    { max++; N[max]= e;  }
  public Elem Get(String k)
    { for (int i=1; i<=max; i++)
        if (N[i].key()==k) return N[i];
      return null;
}}
```

```
class OrdBokLS extends OrdBok
{ public void ListSortert()
  { for (int i=1; i<=max; i++)
      for (int j=1; j<=max-i; j++)
        if (N[j].key()>N[j+1].key()) swap(j,j+1)
    for (int i=1; i<=max; i++)
      skriv(N[i]);
} }
```

- Imperative
  - A program execution is regarded as a sequence of operations manipulating a set of data items
    E.g., Fortran, Basic, Pascal, C, . . .

Programming Paradigms

- Imperative
  - A program execution is regarded as a sequence of operations manipulating a set of data items
    E.g., Fortran, Basic, Pascal, C, ...
- Object-oriented
  - A program execution is regarded as a physical model simulating a real or imaginary part of the world
    E.g., Simula, Java, C++, ...

Programming Paradigms

- Imperative
  - A program execution is regarded as a sequence of operations manipulating a set of data items
    E.g., Fortran, Basic, Pascal, C, ...
- Object-oriented
  - A program execution is regarded as a physical model simulating a real or imaginary part of the world
    E.g., Simula, Java, C++, ...
- Functional
  - A program is regarded as a mathematical function
    E.g., Lisp, Haskell, Erlang, ...

Programming Paradigms

- Imperative
  - A program execution is regarded as a sequence of operations manipulating a set of data items
    E.g., Fortran, Basic, Pascal, C, ...
- Object-oriented
  - A program execution is regarded as a physical model simulating a real or imaginary part of the world
    E.g., Simula, Java, C++, ...
- Functional
  - A program is regarded as a mathematical function
    E.g., Lisp, Haskell, Erlang, ...
- Logic
  - A program is regarded as a set of equations describing relations
    E.g., Prolog, ASP, ...

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; }
```
. . call by name

```
z := 3;
swap(z,z)
```
$z = 0$

assignment (state manipulation with side-effects) under aliasing is difficult

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

```
swap(x,y) = (y,x)
```

Imperative

swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }

without extra memory

void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name

z := 3;
swap(z,z)
z = 0

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x)
  swap(2,1) $\rightsquigarrow$ (1,2)

Imperative

swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }

<u>without extra memory</u>

void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name

z := 3;
swap(z,z)
$z = 0$

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!
  swap(2,1) $\leadsto$ (1,2)
  swap(2,True) $\leadsto$ (True,2)
...

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } .. call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

  swap(2,1) $\rightsquigarrow$ (1,2)

  swap(2,True) $\rightsquigarrow$ (True,2)

...

Logical

swap((X,Y), (Y,X)).

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

swap(2,1) $\rightsquigarrow$ (1,2)

swap(2,True) $\rightsquigarrow$ (True,2)

...

Logical

```
swap( (X,Y), (Y,X) ).
  ?– swap( (2,a), (a,2) )
     Yes
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

<u>without extra memory</u>

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

swap(2,1) $\rightsquigarrow$ (1,2)

swap(2,True) $\rightsquigarrow$ (True,2)

...

Logical

```
swap( (X,Y), (Y,X) ).
 ?– swap( (2,a), (a,2) )
    Yes
 ?– swap( (2,True), Z )
    Z = (True,2)
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

swap(2,1) $\leadsto$ (1,2)

swap(2,True) $\leadsto$ (True,2)

...

Logical

```
swap( (X,Y), (Y,X) ).
 ?– swap( (2,a), (a,2) )
    Yes
 ?– swap( (2,True), Z )
    Z = (True,2)
 ?– swap( Z, (True,2) )
    Z = (2,True)
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

  swap(2,1) $\rightsquigarrow$ (1,2)

  swap(2,True) $\rightsquigarrow$ (True,2)

...

Logical

```
swap( (X,Y), (Y,X) ).
 ?– swap( (2,a), (a,2) )
    Yes
 ?– swap( (2,True), Z )
    Z = (True,2)
 ?– swap( Z, (True,2) )
    Z = (2,True)
 ?– swap( (1,2), (X,Y) )
    X = 2, Y = 1
```

Imperative

```
swap-a(x,y: int) {
  var z := y;
  y := x;
  x := z; }
```

without extra memory

```
void swap(x,y: int) {
  x := x + y;
  y := x - y;
  x := x - y; } . . call by name
```

```
z := 3;
swap(z,z)
z = 0
```

assignment (state manipulation with side-effects) under aliasing is difficult

Functional

swap(x,y) = (y,x) polymorphic!

swap(2,1) $\leadsto$ (1,2)

swap(2,True) $\leadsto$ (True,2)

...

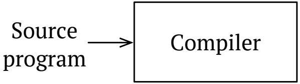
Logical

```
swap( (X,Y), (Y,X) ).
 ?– swap( (2,a), (a,2) )
    Yes
 ?– swap( (2,True), Z )
    Z = (True,2)
 ?– swap( Z, (True,2) )
    Z = (2,True)
 ?– swap( (1,2), (X,Y) )
    X = 2, Y = 1
 ?– swap( (1,2), (Y,Y) )
    No
```

Compiler

Interpreter

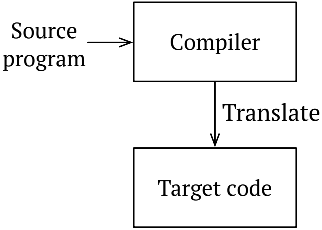
Compiler



Interpreter

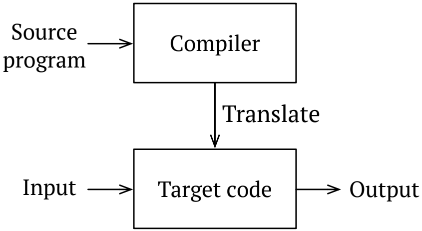
Compiler

Interpreter

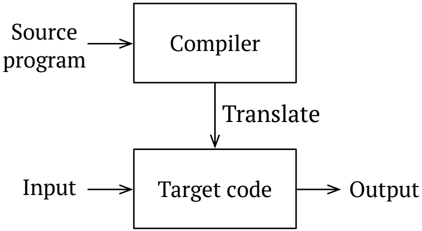


Compiler

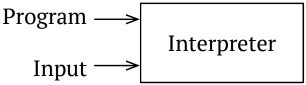
Interpreter



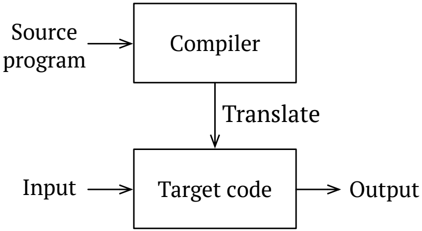
Compiler



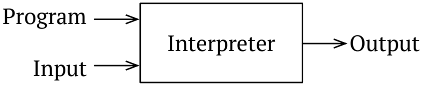
Interpreter



Compiler



Interpreter



Compilation vs. Interpretation

```
void inc(x:int)
  if (x*3 + 3 > 0)
       y := x*3 + 4;
  else y := x*3 + 3;
```

```
void inc(x:int)
  if (x*3 + 3 > 0)
        y := x*3 + 4;
  else y := x*3 + 3;
```

Compilation ↓

```
inc:           (machine code)
1. r1 := x
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt
```

```
void inc(x:int)
  if (x*3 + 3 > 0)
       y := x*3 + 4;
  else y := x*3 + 3;
```

Compilation ↓

inc:	(machine code)
1. r1 := 2	
2. r1 := r1 * 3	
3. r1 := r1 + 3	
4. if r1 > 0 then goto 7	
5. y := r1	
6. goto 8	
7. y := r1 + 1	
8. halt	

inc(2) ↑

void inc(x:int)
  if (x*3 + 3 > 0)
      y := x*3 + 4;
  else y := x*3 + 3;

Compilation ↓

inc: (machine code)
1. r1 := 2
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt

inc(2) ↑ y = 10

result: change of state

Compilation vs. Interpretation

```
void inc(x:int)
  if (x*3 + 3 > 0)
      y := x*3 + 4;
  else y := x*3 + 3;
```

```
inc(x) =
  if (x*3 + 3 > 0)
      x*3 + 4
  else x*3 + 3
```

Compilation ↓

```
inc:            (machine code)
1. r1 := x
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt
```

inc(2) ↑ y = 10

result: change of state

Compilation vs. Interpretation

```
void inc(x:int)
  if (x*3 + 3 > 0)
      y := x*3 + 4;
  else y := x*3 + 3;
```

```
inc(x) =
  if (x*3 + 3 > 0)
      x*3 + 4
  else x*3 + 3
```

Compilation ↓

```
inc:                (machine code)
1. r1 := x
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt
```

inc(2) ↑ y = 10

result: change of state

```
ev( if X a else b ) = ev(if ev(X) a else b)
ev( if True a else b ) = ev(a)
ev( if False a else b) = ev(b)
ev( a "+" b ) = ev(a) + ev(b)
ev( a "*" b ) = ev(a) * ev(b)
...
```

```
void inc(x:int)
  if (x*3 + 3 > 0)
        y := x*3 + 4;
  else y := x*3 + 3;
```

```
inc(2) =
  if (2*3 + 3 > 0)
        2*3 + 4
  else 2*3 + 3
```

Compilation ↓

inc:	(machine code)
1. r1 := x	
2. r1 := r1 * 3	
3. r1 := r1 + 3	
4. if r1 > 0 then goto 7	
5. y := r1	
6. goto 8	
7. y := r1 + 1	
8. halt	

inc(2) ↑ y = 10

result: change of state

↓ **Interpretation**

inc(2)

```
ev( if X a else b ) = ev(if ev(X) a else b)
ev( if True a else b ) = ev(a)
ev( if False a else b) = ev(b)
ev( a "+" b ) = ev(a) + ev(b)
ev( a "*" b ) = ev(a) * ev(b)
...
```

Compilation vs. Interpretation

```
void inc(x:int)
  if (x*3 + 3 > 0)
      y := x*3 + 4;
  else y := x*3 + 3;
```

Compilation ↓

inc: (machine code)

1. r1 := x
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt

inc(2) ↑ y = 10

result: change of state

inc(2) =
  if True
      2*3 + 4
  else 2*3 + 3

↓ **Interpretation**

inc(2)

ev(if X a else b) = ev(if ev(X) a else b)
ev(if True a else b) = ev(a)
ev(if False a else b) = ev(b)
ev(a "+" b) = ev(a) + ev(b)
ev(a "*" b) = ev(a) * ev(b)
...

```
void inc(x:int)
  if (x*3 + 3 > 0)
      y := x*3 + 4;
  else y := x*3 + 3;
```

Compilation ↓

inc:	(machine code)
1. r1 := x	
2. r1 := r1 * 3	
3. r1 := r1 + 3	
4. if r1 > 0 then goto 7	
5. y := r1	
6. goto 8	
7. y := r1 + 1	
8. halt	

inc(2) ↑ y = 10

result: change of state

```
inc(2) =
  if True
      2*3 + 4
  else 2*3 + 3
```

↓ **Interpretation**

inc(2)

ev(if X a else b) = ev(if ev(X) a else b)
ev(if True a else b) = ev(a)
ev(if False a else b) = ev(b)
ev(a "+" b) = ev(a) + ev(b)
ev(a "*" b) = ev(a) * ev(b)
...

$\rightsquigarrow$ 10

result: a value

Compilation vs. Interpretation

```
void inc(x:int)
  if (x*3 + 3 > 0)
       y := x*3 + 4;
  else y := x*3 + 3;
```

```
inc(x) =
  if (x*3 + 3 > 0)
       x*3 + 4
  else x*3 + 3
```

Compilation ↓

inc: (machine code)

1. r1 := x
2. r1 := r1 * 3
3. r1 := r1 + 3
4. if r1 > 0 then goto 7
5. y := r1
6. goto 8
7. y := r1 + 1
8. halt

inc(2) ↑ y = 10

result: change of state

↓ **Interpretation**

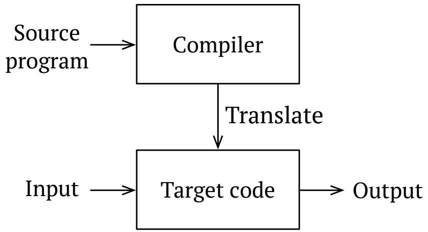
inc(2)

```
ev( if X a else b ) = ev(if ev(X) a else b)
ev( if True a else b ) = ev(a)
ev( if False a else b) = ev(b)
ev( a "+" b ) = ev(a) + ev(b)
ev( a "*" b ) = ev(a) * ev(b)
...
```

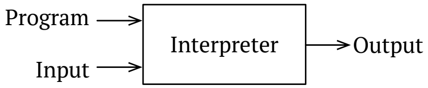
$\leadsto$ 10

result: a value

Compiler



Interpreter



- Compilation can be more efficient than interpretation
- Interpretation can be more flexible than compilation