Computer Languages

Putting it all together
Looking forward

March 12

www2.hh.se/staff/vero/languages

What is it we have?

For each method in the minijava program we have:

a list of assembler instructions implementing the body of the method

an instance of the class `Frame` (in our case an instance of `MipsFrame`)

Frames

```
class Frame{
/*
Global Information:
  - registers
Local Information:
  - label
  - arg addresses
Methods for:
  - code generation
    - instr. sel.
    - reg. alloc.
    - call prelude/epilogue
*/
}
```

What is it we have?

For each method in the minijava program we have:

a list of assembler instructions implementing the body of the method

an instance of the class `Frame` (in our case an instance of `MipsFrame`)

Frames

```
class Frame{
/*
Global Information:
  - registers
Local Information:
  - label
  - arg addresses
Methods for:
  - code generation
    - instr. sel.
    - reg. alloc.
    - call prelude/epilogue
*/
}
```

What is it we have?

For each method in the minijava program we have:

a list of assembler instructions implementing the body of the method

an instance of the class `Frame` (in our case an instance of `MipsFrame`)

Frames

```
class Frame{
/*
Global Information:
  - registers
Local Information:
  - label
  - arg addresses
Methods for:
  - code generation
    - instr. sel.
    - reg. alloc.
    - call prelude/epilogue
*/
}
```

What is it we have?

For each method in the minijava program we have:

a list of assembler instructions implementing the body of the method

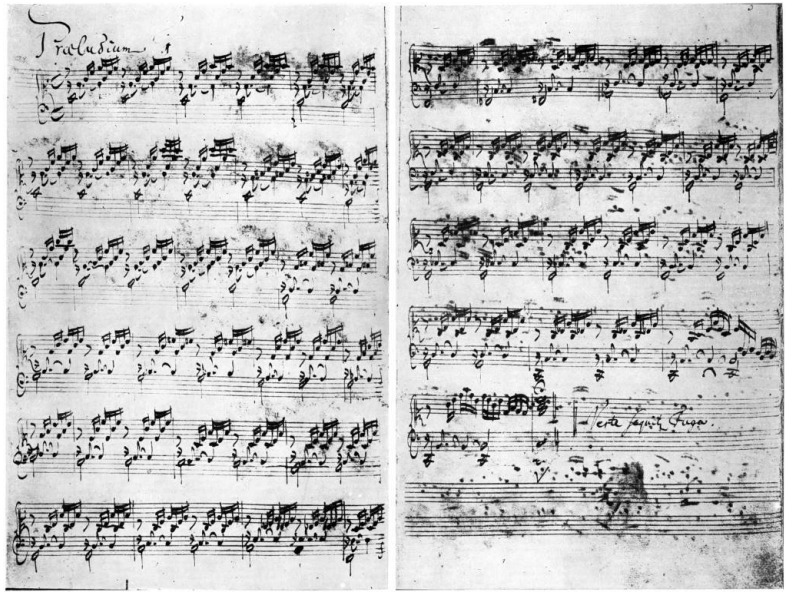
an instance of the class `Frame` (in our case an instance of `MipsFrame`)

Frames

```
class Frame{
/*
Global Information:
  - registers
Local Information:
  - label
  - arg addresses
Methods for:
  - code generation
    - instr. sel.
    - reg. alloc.
    - call prelude/epilogue
*/
}
```

The prelude

An action or event serving as an introduction to something more important



The frame that has to go on top of the stack will start where the stack ends now:

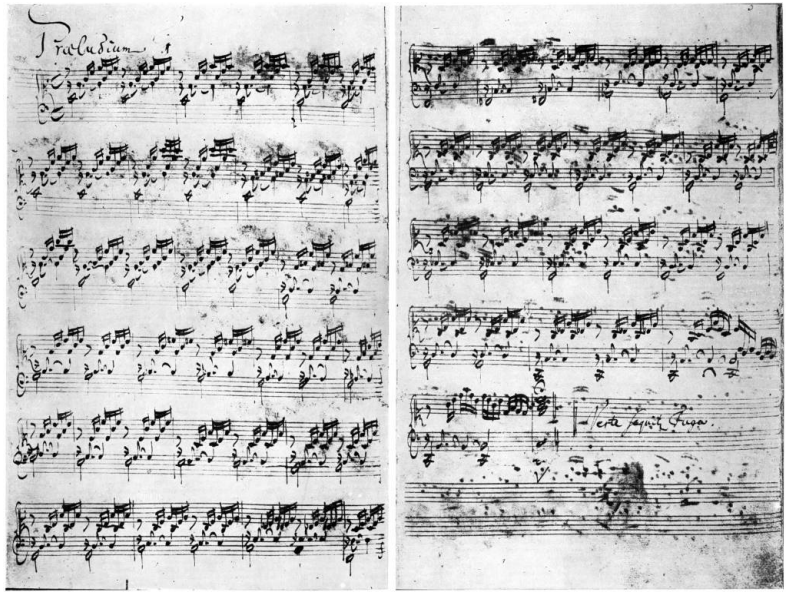
```
fp <- sp
```

The stack pointer has to be moved!

```
sp <- sp - framesize
```

The prelude

An action or event serving as an introduction to something more important



The frame that has to go on top of the stack will start where the stack ends now:

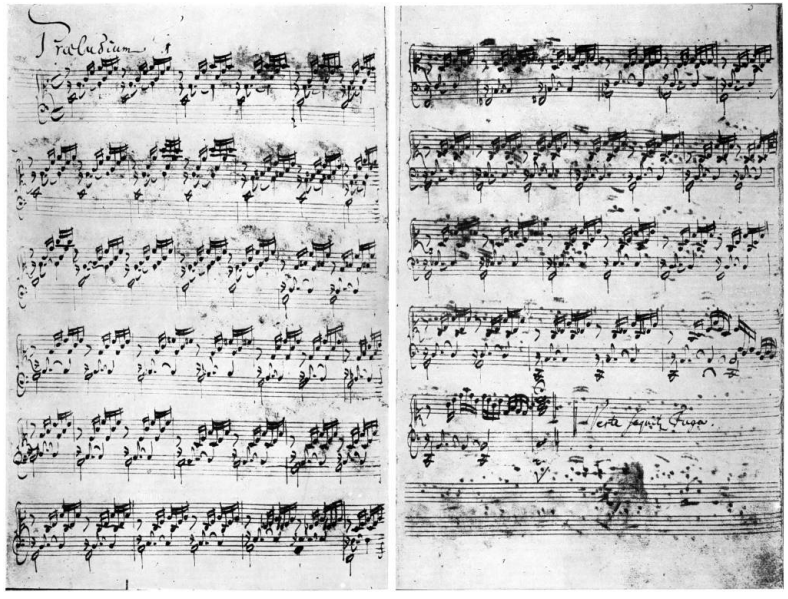
```
fp <- sp
```

The stack pointer has to be moved!

```
sp <- sp - framesize
```

The prelude

An action or event serving as an introduction to something more important



The frame that has to go on top of the stack will start where the stack ends now:

```
fp <- sp
```

The stack pointer has to be moved!

```
sp <- sp - framesize
```

Epilogue

A section or speech at the end of a book or play that serves as a comment on or a conclusion to what has happened.

The frame has to be removed from the stack:

```
sp <- sp + framesize
```

Control has to be returned to the address below the jump to this functions code

```
jump ra
```

Epilogue

A section or speech at the end of a book or play that serves as a comment on or a conclusion to what has happened.

The frame has to be removed from the stack:

```
sp <- sp + framesize
```

Control has to be returned to the address below the jump to this functions code

```
jump ra
```

Epilogue

A section or speech at the end of a book or play that serves as a comment on or a conclusion to what has happened.

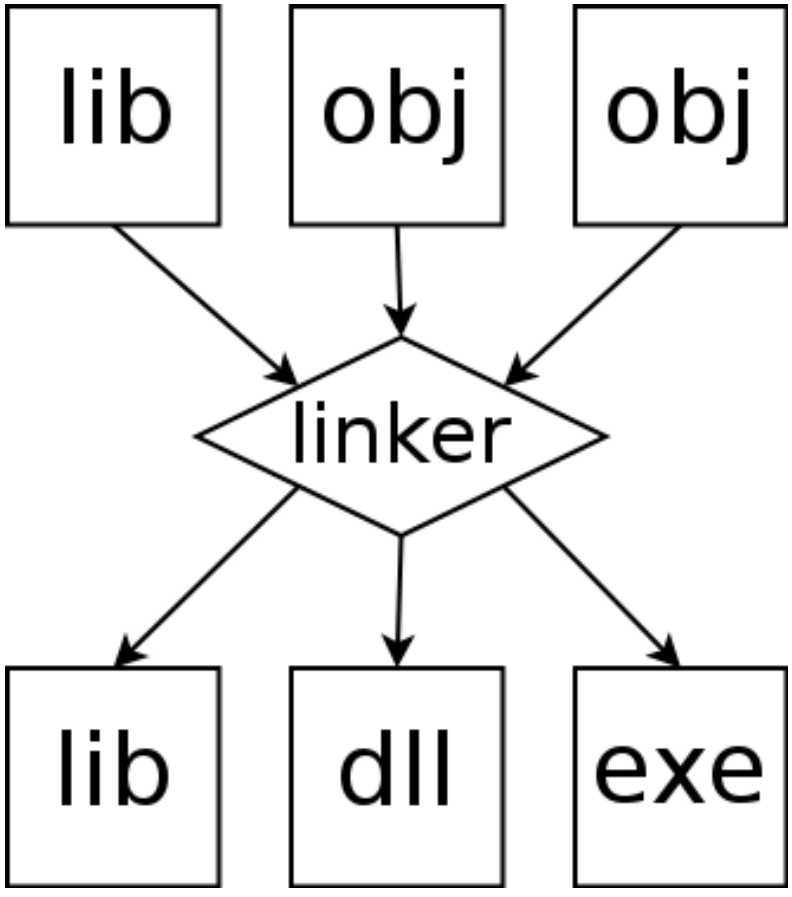
The frame has to be removed from the stack:

```
sp <- sp + framesize
```

Control has to be returned to the address below the jump to this functions code

```
jump ra
```

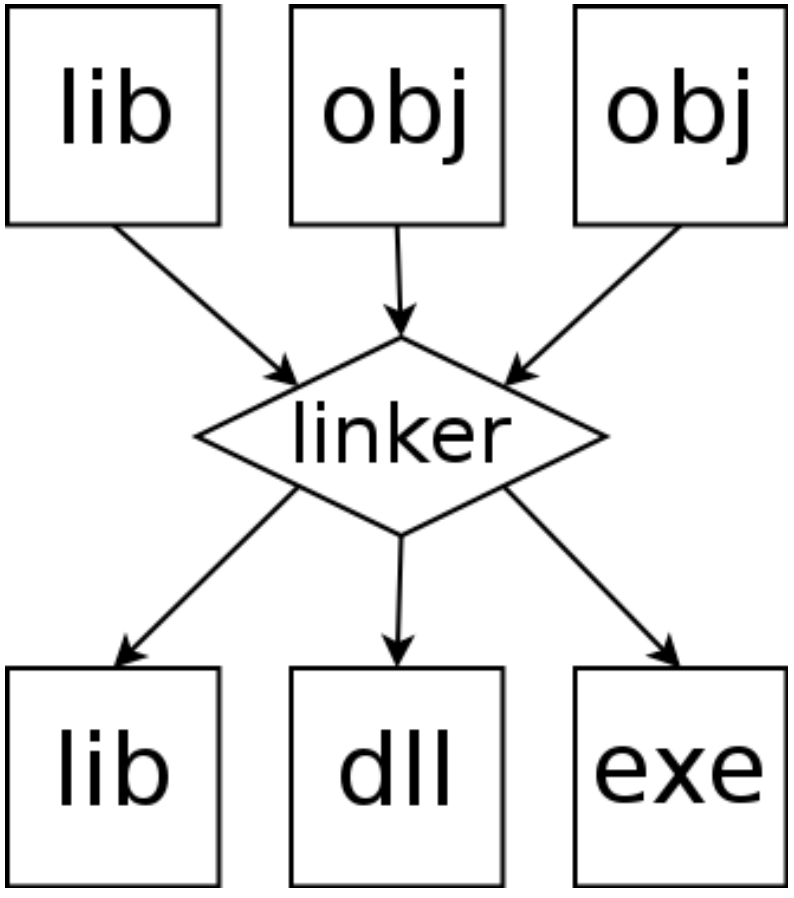
Making it work



Generate a file with the assembler instructions and assemble it!

Link it with the object file including all used external functions to produce an executable!

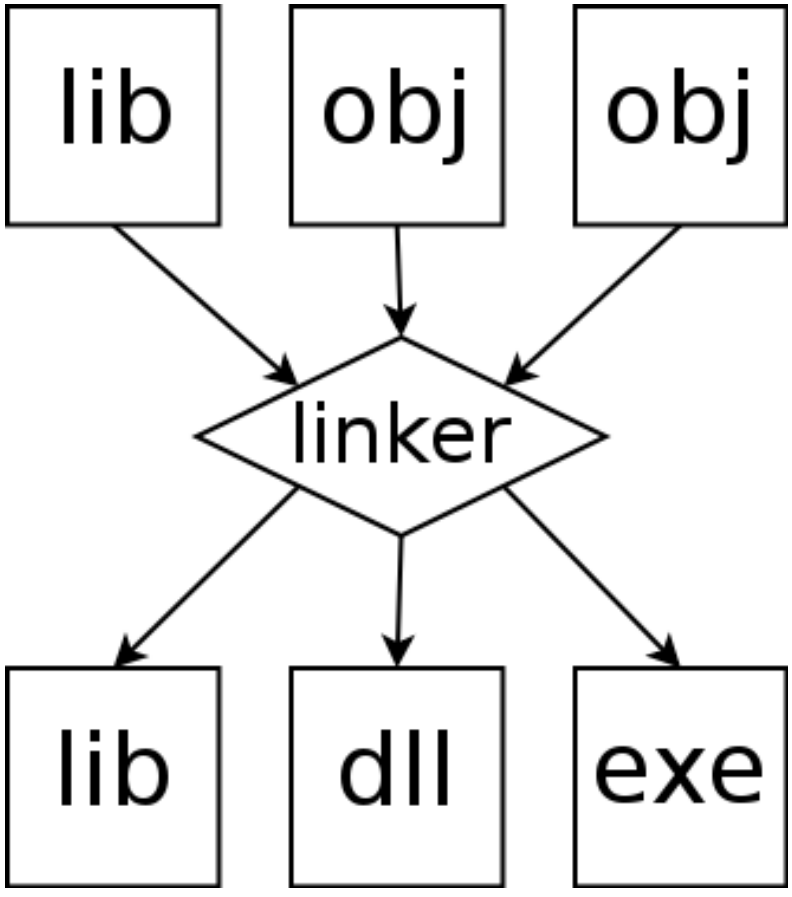
Making it work



Generate a file with the assembler instructions and assemble it!

Link it with the object file including all used external functions to produce an executable!

Making it work



Generate a file with the assembler instructions and assemble it!

Link it with the object file including all used external functions to produce an executable!

Standard issues we did not deal with

Nested Functions

Some languages like Pascal and Ada allow for functions to be defined locally in other functions.

Example

```
void main(String[] a){
  int abs(int n){
    if(n>=0)return n;
    else return 0-n;
  }
  int integer(String n){
    return new Integer(n);
  }
  print(abs(integer(a[0])));
}
```

Activation records (frames) have to include a pointer to the more recent activation of the enclosing function: static link

Addresses have to be traced along the static link!

Standard issues we did not deal with

Nested Functions

Some languages like Pascal and Ada allow for functions to be defined locally in other functions.

Example

```
void main(String[]a){
   int abs(int n){
     if(n>=0)return n;
     else return 0-n;
   }
   int integer(String n){
     return new Integer(n);
   }
   print(abs(integer(a[0])));
}
```

Activation records (frames) have to include a pointer to the more recent activation of the enclosing function: static link

Addresses have to be traced along the static link!

Standard issues we did not deal with

Nested Functions

Some languages like Pascal and Ada allow for functions to be defined locally in other functions.

Example

```
void main(String[]a){
   int abs(int n){
     if(n>=0)return n;
     else return 0-n;
   }
   int integer(String n){
     return new Integer(n);
   }
   print(abs(integer(a[0])));
}
```

Activation records (frames) have to include a pointer to the more recent activation of the enclosing function: static link

Addresses have to be traced along the static link!

Standard issues we did not deal with

Nested Functions

Some languages like Pascal and Ada allow for functions to be defined locally in other functions.

Example

```
void main(String[]a){
   int abs(int n){
     if(n>=0)return n;
     else return 0-n;
   }
   int integer(String n){
     return new Integer(n);
   }
   print(abs(integer(a[0])));
}
```

Activation records (frames) have to include a pointer to the more recent activation of the enclosing function: static link

Addresses have to be traced along the static link!

Interpreters

Is it necessary to translate to assembler?

NO! A *language processor* might interpret the source after the *analysis phase*.

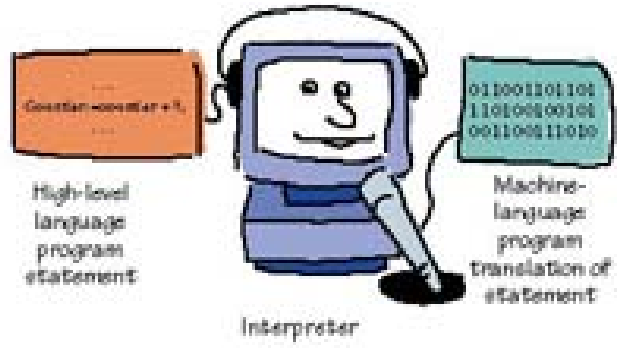
The interpreter can be understood as implementing an abstract machine!

Environments and visitors are still needed!

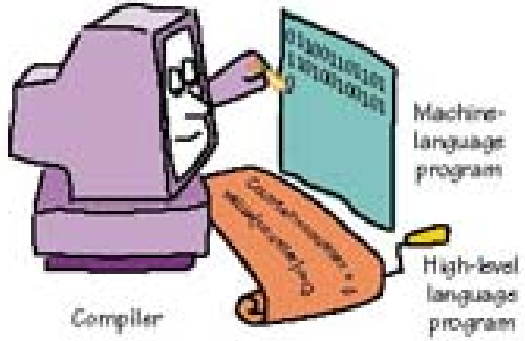
Interpreters

Is it necessary to translate to assembler?

NO! A *language processor* might interpret the source after the *analysis phase*.



The interpreter can be understood as implementing an abstract machine!



Environments and visitors are still needed!

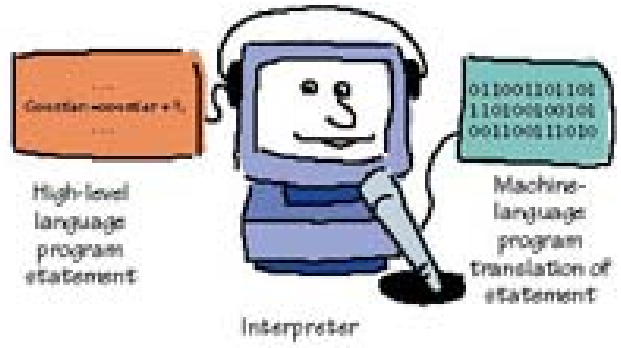
Interpreters

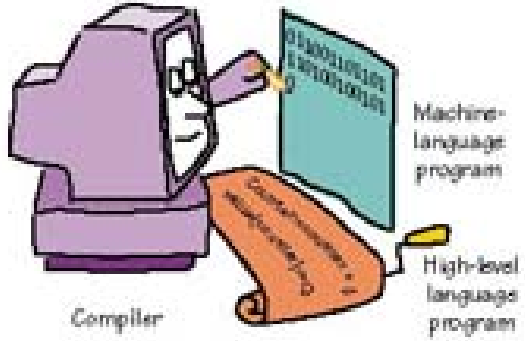
Is it necessary to translate to assembler?

NO! A *language processor* might interpret the source after the *analysis phase*.

The interpreter can be understood as implementing an abstract machine!

Environments and visitors are still needed!





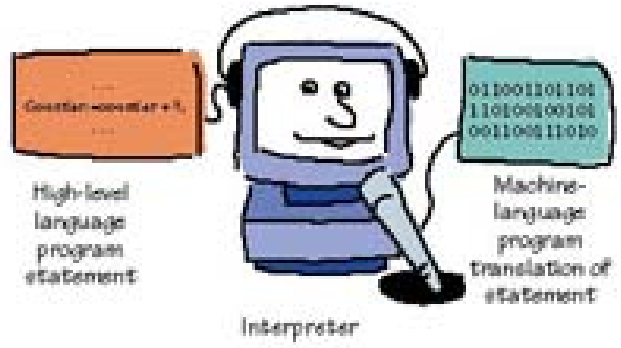
Interpreters

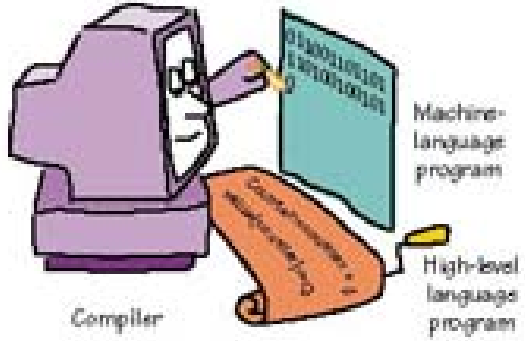
Is it necessary to translate to assembler?

NO! A *language processor* might interpret the source after the *analysis phase*.

The interpreter can be understood as implementing an abstract machine!

Environments and visitors are still needed!





Advanced features

Loop optimizations

Are there expressions in a loop whose value is constant during the iterations?

Maybe some expressions can be moved outside loops!

This is a very well studied field, and there are courses about optimizing compilers that you are in a position to attend!

Advanced features

Loop optimizations

Are there expressions in a loop whose value is constant during the iterations?

Maybe some expressions can be moved outside loops!

This is a very well studied field, and there are courses about optimizing compilers that you are in a position to attend!

Advanced features

Loop optimizations

Are there expressions in a loop whose value is constant during the iterations?

Maybe some expressions can be moved outside loops!

This is a very well studied field, and there are courses about optimizing compilers that you are in a position to attend!

Advanced features

Garbage collection

When an object is created memory is claimed in the heap. How many objects can be created? (the heap is not infinite!)

Memory can be recovered by keeping track of the references made to a specific address. If there are no references, the chunk of memory can be returned to the heap!

A grabage collector gets to run now and then, interrupting execution of the program.

Advanced features

Garbage collection

When an object is created memory is claimed in the heap. How many objects can be created? (the heap is not infinite!)

Memory can be recovered by keeping track of the references made to a specific address. If there are no references, the chunk of memory can be returned to the heap!

A grabage collector gets to run now and then, interrupting execution of the program.

Advanced features

Garbage collection

When an object is created memory is claimed in the heap. How many objects can be created? (the heap is not infinite!)

Memory can be recovered by keeping track of the references made to a specific address. If there are no references, the chunk of memory can be returned to the heap!

A grabage collector gets to run now and then, interrupting execution of the program.

Other kind of compilers

These slides are produced with help of a compiler: pdflatex.

lecture12.tex → beamer.sty pgf.sty → lecture12.pdf

Example

```
\begin{frame}[fragile]
  \frametitle{Functional programming}
\includegraphics[width=5cm]{deck}
  \begin{block}
    \begin{semiverbatim}
deck = [(s,v)| s<-[Spade .. Club],v<-[A .. King]]
    \end{semiverbatim}
  \end{block}
\end{frame}
```

Other kind of compilers

These slides are produced with help of a compiler: pdflatex.

lecture12.tex → beamer.sty pgf.sty → lecture12.pdf

Example

```
\begin{frame}[fragile]
  \frametitle{Functional programming}
\includegraphics[width=5cm]{deck}
  \begin{block}
    \begin{semiverbatim}
deck = [(s,v)| s<-[Spade .. Club],v<-[A .. King]]
    \end{semiverbatim}
  \end{block}
\end{frame}
```

Other kind of compilers

These slides are produced with help of a compiler: pdflatex.

lecture12.tex → beamer.sty pgf.sty → lecture12.pdf

Example

```
\begin{frame}[fragile]
  \frametitle{Functional programming}
\includegraphics[width=5cm]{deck}
  \begin{block}
    \begin{semiverbatim}
deck = [(s,v)| s<-[Spade .. Club],v<-[A .. King]]
    \end{semiverbatim}
  \end{block}
\end{frame}
```

Theory of computation

Compilers are very much related with more theoretical issues in computer science (not just programming!)

Some of these issues are related to questions like Are there some functions that cannot be computed?

The halting problem

Can we program a function h that can be applied to a program p and an argument x

$$h(p, x)$$

so that h answers true when p(x) terminates and false otherwise?

Theory of computation

Compilers are very much related with more theoretical issues in computer science (not just programming!)

Some of these issues are related to questions like Are there some functions that cannot be computed?

The halting problem

Can we program a function h that can be applied to a program p and an argument x

$$h(p, x)$$

so that h answers true when p(x) terminates and false otherwise?

Theory of computation

Compilers are very much related with more theoretical issues in computer science (not just programming!)

Some of these issues are related to questions like Are there some functions that cannot be computed?

The halting problem

Can we program a function h that can be applied to a program p and an argument x

$$h(p, x)$$

so that h answers true when p(x) terminates and false otherwise?

Theory of computation

Such a function could be very useful!

Example

```
int fac(int n){
  if(n==0) return 1;
  else return n*fac(n-1);
}
```

We could use h (that always terminates, we know that we might have to wait but the answer will come!) to decide whether to wait for an answer from fac!

Example

h(fac,123456789)
h(fac,-1)

Theory of computation

Such a function could be very useful!

Example

```
int fac(int n){
  if(n==0) return 1;
  else return n*fac(n-1);
}
```

We could use h (that always terminates, we know that we might have to wait but the answer will come!) to decide whether to wait for an answer from fac!

Example

h(fac,123456789)
h(fac,-1)

Theory of computation

Such a function could be very useful!

Example

```
int fac(int n){
  if(n==0) return 1;
  else return n*fac(n-1);
}
```

We could use h (that always terminates, we know that we might have to wait but the answer will come!) to decide whether to wait for an answer from fac!

Example

h(fac,123456789)
h(fac,-1)

Theory of computation

Such a function could be very useful!

Example

```
int fac(int n){
  if(n==0) return 1;
  else return n*fac(n-1);
}
```

We could use h (that always terminates, we know that we might have to wait but the answer will come!) to decide whether to wait for an answer from fac!

Example

h(fac,123456789)
h(fac,-1)

Incomputability

To show that you can program a given function you just do it!

To show that you cannot program a given function you have to argue! (Prove a theorem!)

Theorem

Any language containing conditionals and recursive function definitions which is powerful enough to program its own compiler cannot be used to program its own terminates function.

Incomputability

To show that you can program a given function you just do it!

To show that you cannot program a given function you have to argue! (Prove a theorem!)

Theorem

Any language containing conditionals and recursive function definitions which is powerful enough to program its own compiler cannot be used to program its own terminates function.

Incomputability

To show that you can program a given function you just do it!

To show that you cannot program a given function you have to argue! (Prove a theorem!)

Theorem

Any language containing conditionals and recursive function definitions which is powerful enough to program its own compiler cannot be used to program its own terminates function.

Incomputability

Assume we can program h according to the specification we gave.

Consider the definition of hetero according to

```
hetero(f)  = if h(f,f) return !f(f);
             else return true;
```

What happens if we apply `hetero(hetero)`?

We get `!hetero(hetero)`!

Conclude that we cannot possibly progam h (in a language where we could use it to program `hetero`!)

Incomputability

Assume we can program h according to the specification we gave.

Consider the definition of hetero according to

```
hetero(f)  = if h(f,f) return !f(f);
             else return true;
```

What happens if we apply `hetero(hetero)`?

We get `!hetero(hetero)`!

Conclude that we cannot possibly progam h (in a language where we could use it to program `hetero`!)

Incomputability

Assume we can program h according to the specification we gave.

Consider the definition of hetero according to

```
hetero(f)  = if h(f,f) return !f(f);
             else return true;
```

What happens if we apply `hetero(hetero)`?

We get `!hetero(hetero)`!

Conclude that we cannot possibly progam h (in a language where we could use it to program `hetero`!)

Incomputability

Assume we can program h according to the specification we gave.

Consider the definition of hetero according to

```
hetero(f)  = if h(f,f) return !f(f);
             else return true;
```

What happens if we apply `hetero(hetero)`?

We get `!hetero(hetero)`!

Conclude that we cannot possibly progam h (in a language where we could use it to program `hetero`!)

Incomputability

Assume we can program h according to the specification we gave.

Consider the definition of hetero according to

```
hetero(f)  = if h(f,f) return !f(f);
             else return true;
```

What happens if we apply `hetero(hetero)`?

We get `!hetero(hetero)`!

Conclude that we cannot possibly progam h (in a language where we could use it to program `hetero`!)

Programming language paradigms

Imperative	Functional	Logic
Assembler, C, C++, C#, Java, Pascal, Ada.	Lisp, Scheme, ML, Haskell.	Prolog.
A program is a command.	A program is an expression.	A program is a predicate.
The purpose of a command is to change state.	An expression has a value.	Values for the variables in the program are computed that make the predicates true.
State? The values of the variables in the program.	Variables do not have values!	

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Programming language paradigms

Imperative

Assembler, C, C++, C#, Java, Pascal, Ada.

A program is a command.

The purpose of a command is to change state.

State? The values of the variables in the program.

Functional

Lisp, Scheme, ML, Haskell.

A program is an expression.

An expression has a value.

Variables do not have values!

Logic

Prolog.

A program is a predicate.

Values for the variables in the program are computed that make the predicates true.

Functional programming

Why?

Programming is difficult, it is easy to make errors! How can programs be made easier to understand?

Examples in Haskell.

Example

```
data Suit = Spade | Heart | Diamond | Club

data Value = A | Two | Three | Four | Five | Six | Seven
           | Eight | Nine | Ten | Knight | Queen | King

type Card = (Suit,Value)
```

Functional programming

Why?

Programming is difficult, it is easy to make errors! How can programs be made easier to understand?

Examples in Haskell.

Example

```
data Suit = Spade | Heart | Diamond | Club

data Value = A | Two | Three | Four | Five | Six | Seven
           | Eight | Nine | Ten | Knight | Queen | King

type Card = (Suit,Value)
```

Functional programming

Why?

Programming is difficult, it is easy to make errors! How can programs be made easier to understand?

Examples in Haskell.

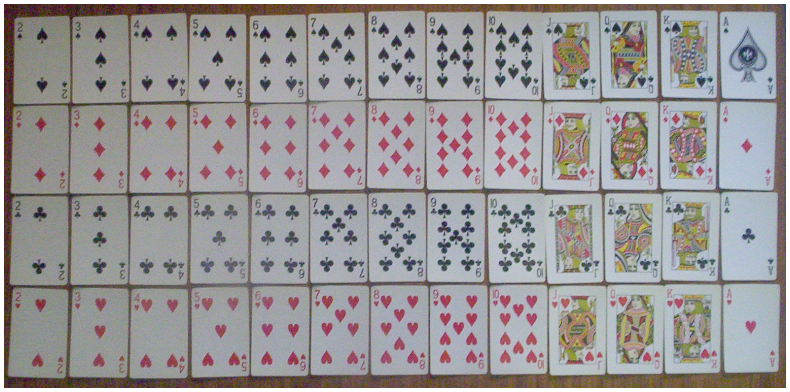
Example

```
data Suit = Spade | Heart | Diamond | Club

data Value = A | Two | Three | Four | Five | Six | Seven
           | Eight | Nine | Ten | Knight | Queen | King

type Card = (Suit,Value)
```

Functional programming

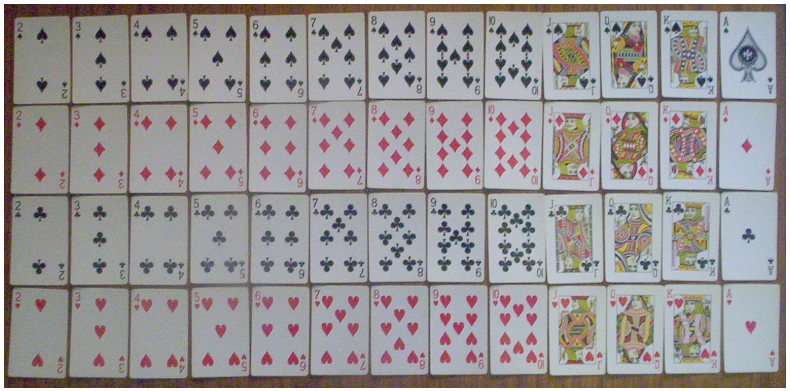


```
deck = [(s,v)| s<-[Spade .. Club], v<-[A .. King]]
```

```
shuffleL []     []     = []
shuffleL (x:xs) (y:ys) = x:y:shuffleL xs ys
```

```
shuffle xs = let n = length xs
                 m = div n  2
             in shuffleL (take m xs) (drop m xs)
```

Functional programming

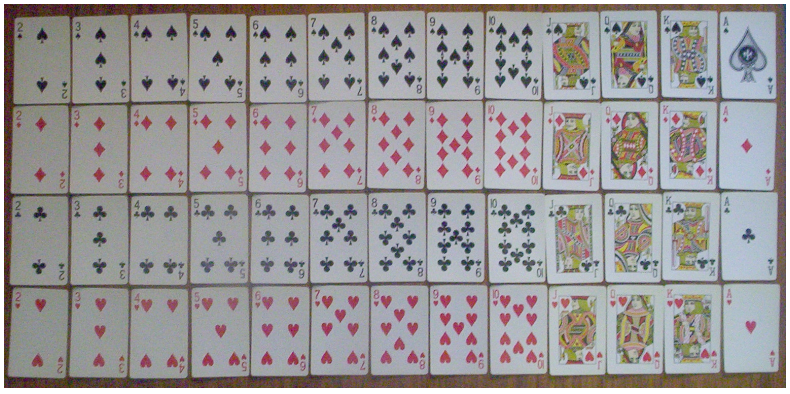


```
deck = [(s,v)| s<-[Spade .. Club], v<-[A .. King]]
```

```
shuffleL []     []     = []
shuffleL (x:xs) (y:ys) = x:y:shuffleL xs ys
```

```
shuffle xs = let n = length xs
                 m = div n  2
             in shuffleL (take m xs) (drop m xs)
```

Functional programming



```
deck = [(s,v)| s<-[Spade .. Club], v<-[A .. King]]
```

```
shuffleL []     []     = []
shuffleL (x:xs) (y:ys) = x:y:shuffleL xs ys
```

```
shuffle xs = let n = length xs
                 m = div n  2
             in shuffleL (take m xs) (drop m xs)
```

Functional programming

Glueing

How easy is it to put together smaller programs to do more advanced things?

```
mix n xs = (iterate shuffle xs) !! n
```

What is `iterate`?

```
iterate :: (a -> a) -> a -> [a]

iterate f a = a : iterate f (f a)
```

Example

```
iterate (+2) 0

[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,^C
```

Functional programming

Glueing

How easy is it to put together smaller programs to do more advanced things?

```
mix n xs = (iterate shuffle xs) !! n
```

What is `iterate`?

```
iterate :: (a -> a) -> a -> [a]

iterate f a = a : iterate f (f a)
```

Example

```
iterate (+2) 0

[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,^C
```

Functional programming

Glueing

How easy is it to put together smaller programs to do more advanced things?

```
mix n xs = (iterate shuffle xs) !! n
```

What is `iterate`?

```
iterate :: (a -> a) -> a -> [a]

iterate f a = a : iterate f (f a)
```

Example

```
iterate (+2) 0

[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,^C
```

Functional programming

Glueing

How easy is it to put together smaller programs to do more advanced things?

```
mix n xs = (iterate shuffle xs) !! n
```

What is `iterate`?

```
iterate :: (a -> a) -> a -> [a]

iterate f a = a : iterate f (f a)
```

Example

```
iterate (+2) 0

[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,^C
```

Functional programming - exotic issues

Functions are values! They can be passed as arguments and returned as results!

Types are inferred!

Evaluation is lazy! A program might terminate even if it includes non-terminating parts!

Functional programming - exotic issues

Functions are values! They can be passed as arguments and returned as results!

Types are inferred!

Evaluation is lazy! A program might terminate even if it includes non-terminating parts!

Functional programming - exotic issues

Functions are values! They can be passed as arguments and returned as results!

Types are inferred!

Evaluation is lazy! A program might terminate even if it includes non-terminating parts!

Functional programming

Many bachelor education programmes in computer science and computer engineering include courses on functional programming (Chalmers, Lund, KTH, Luleå, Linköping in Sweden)

Examination

The project

You have to have submitted all labs part of the project. They need not be perfect, but they must compile and work in some of the cases!

The questions

There are questions related to each part the project that all students have to be prepared to answer.

Students are asked questions individually: each student gets her/his own grade!

Each student has to book a 30 minutes pass with me, I will have your project available!

The lexer

Explain some of the regular expressions you used.

1. What is the purpose of the lexer?
2. What errors can be captured by the lexer?
3. What tools can we use for producing lexers?
4. How are the tools used?

The parser

Explain some of the grammar rules you used. Explain some of the abstract syntax trees you constructed.

1. What is the purpose of the parser?
2. What errors can be captured by the parser?
3. What tools can we use for producing parsers?
4. How are the tools used?

The typechecker

Explain some of the cases of your type checker.

1. What is the purpose of the typechecker?
2. What errors can be captured by the typechecker?
3. What datastructure does the typechecker work on?

Translation to intermediate representation

Explain some of the cases of the translation.

1. What is the purpose of the translation phase?
2. What is the output of this phase?

Instruction selection

Explain some of the patterns you used. Explain the assembler instructions you generate.

1. What is the purpose of the instruction selectio phase?
2. What is left to be done after instruction selection?