Oppsummering Progspråk

Ting med variable:

- Lexeme
- Token
- Keyword: is a lexeme that has some special meaning in the language, and may be related to some specific syntactic and semantic rules. A reserved word is a lexeme that can be used only as a keyword.
- Literal
- **Atom:** Starter med lower-case letter etterfulg av hva tall og bokstaver.
    - atom er et eksempel på et atom!
    - Atom og 123 er ikke atomer!
- Dataflytvariable
- Stream
- Mutable variable
- Lists:
    - In abstract terms, a list is an ordered collection of items
    - In Oz, lists are chains of nested records
    - The record that form a list have the label '|' and exactly two fields with the features 1 and 2
    - The feature 1 is the first (topmost) redcord names the field which keeps the first elemnt of the list (the head of the list).
    - The feature 2 of the record naed a field which keeps the rest of the list (the tal og the list)
    - The rest of the list is another list - either another record with analogous structure, or empty list
    - [1 2 3] == '|' (1 '|' (2 '|' (3 nil))) %true

- Record and tuple: A record has a label and any number of named feauture-field pairs (or key-value pairs, with feature=key, field=value)

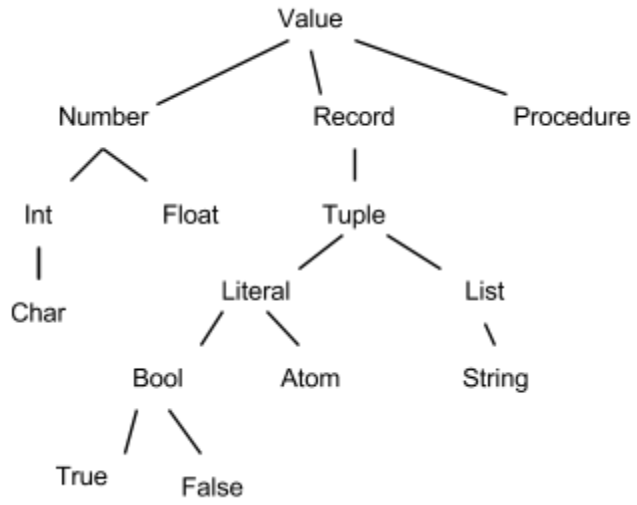
- A record without any field is a atom
  atom == atom() %true
  {Record.label atom} == atom() %true
- A record without explicit features is a tuple:
  tuple(a b c) == tuple(1:a 2:b 3:c) %true
- the tupling constructor # is a convenient syntactic sugar for building records.
- X#Y#Z is a record with the label '#' and three features named 1,2 and 3, with the fields X,Y and Z:
  X#Y#Z == '#'(1:X 2:Y 3:Z) %true
  X#Y#Z == '#'(X Y Z) %true

- **Abstract data types (ADT):** In an implementation of the ADT, the instance data structures can be
The three properties of datastructures are: **Bundledness, Security and statefulness.**

- Bundled, with operations embedded in the data structures
- Unbundled, with operations applied to instances from outside
- Secure, with access to the underlying representation only through the operations

- Insecure, with open access to the underlying representation.
- Stateful

- Types:



“Oz spesifikke ting”:

- SolveAll
- ‘_’ the unbound variable
- ‘#’ record and tuple
- ‘|’ list and records
- **{Browse …}**
  - Is not declarative
  - Is not sequential
- **{Show …}**
  - Is not declarative
- Send/NewPort (Porter og strømmer)
- NewCell (@)
- **NewName:** Is not declarative because it gives a unique name each time it is called.
- ByNeed
- ?Variable
- fun/proc {$ ….}
- **{Record.arity …..}:** The arity of a record is a collecction of all its features sorted alphanumerically.
  - {Browse {Record.arity record(one two three)}} → [1 2 3]
  - {Browse {Record.arity record(1:one 2:two 3:three)}} → [1 2 3]
  - {Browse {Record.arity record(one two:two three)}} → [1 2 two]
  - {Browse {Record.arity record(4:b 12:c a '4':d '12':e)}} → [1 4 12 ‘12’ ‘4’]
- **Map:** Higher order programming. Maps the function on every element in the input list.
  - {Map [1 2 3] fun {$ Item} Item+1 end} → [2 3 4]
- **Enumerate:** is a producer!
  - {Enumerate 1 5} → [1 2 3 4 5]
- **Filter:** (Tranduction)Takes in a list an filter elements based on a Criterion input!
  - {Browse {Filter {Enumerate 1 10} IsOdd}} → [1 3 5 7 9]
- **Zip:** (Tranduction) is a multiplexer! It tranduces two lists into one list
- **Unzip:** Is a demultiplexer! It transduces one list into two lists.
- **FoldRight/FoldRight:**
- **Read only:** X =!!Y → X is a read-only view of Y
- **Thread operations:** may lead to non-declarativeness and they are NOT a part of the declarative model.
  - {Thread.this} returns the name of the curent thread
  - {Thread.suspend T} stops thread T; T remains stopped until it is explicityly resumed
  - {Thread.resumeT} reactivates thread T if it was suspended
  - {Thread.preemt T} preemts thread T
  - {Thread.terminate T} terminates thread T; can no longer be run.
- **{NewPort Stream}** returns a new port associated with Stream
- **{Send Port Value}** causes Port to extend Stream with Value and a new unbound variable.

DSKL: (The declarative sequential kernal language).

Bestanddelene i den abstrakte maskinen DSKL:

- Environment: is a mapping between identifiers in statements and variables in the single assignment store: E = {<id>1 → v1, <id>2 → v2 ….}

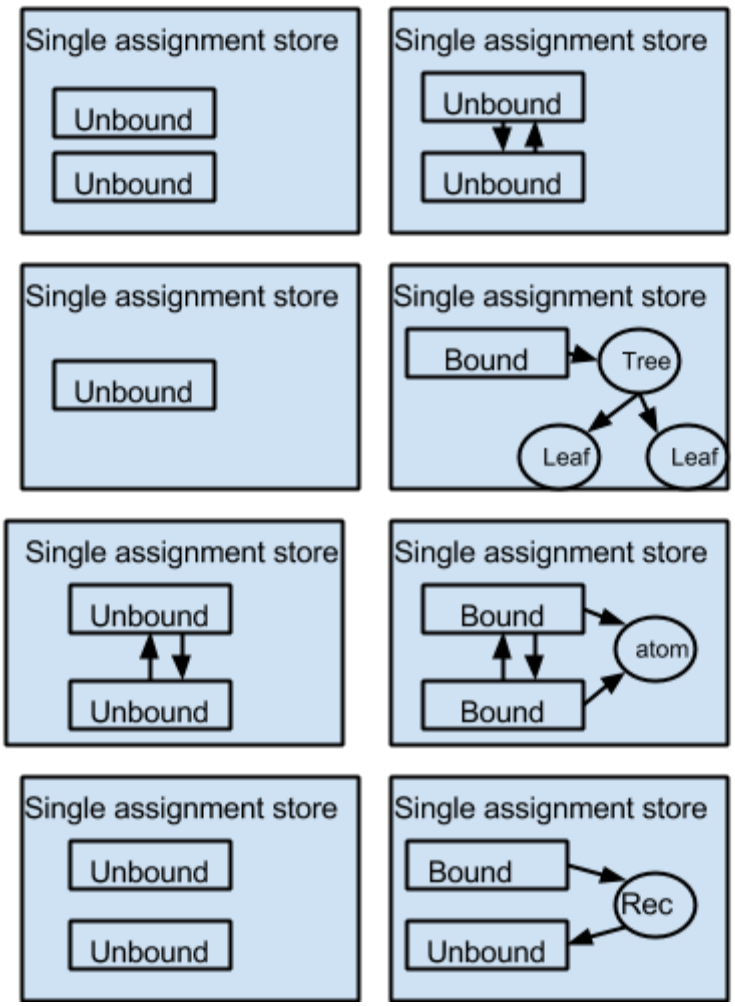
- Operations on environments:

- Adjunction:
  E = {A → v1}
  E'= E + {B → v2} = {A → v1, B → v2}
  E = {A → v1}
  E' = E + {A → v2} = {A → v2}
- Restriction:
  E = {A → v1, B → v2}
  E' = E|{A, C} = {A → v1}
  E = {A → v1, B → v2}
  E' = E|{ } = { }
- Lookup:
  E = {A → v1, B → v2}
  E(A) = v1
  E = {A → v1, B → v2}
  E(C) causes a lookup error!

- Semantic stack(S=[s1, s2, ...] , s1=(<statement>1, E1), E er environment): A stack of semantic statements, where the topmost element is the next one to be executed. A semantic statement is a statement and an environment (a mapping from identifiers to variables)

- Single assignment store(σ): All variables ever declared and their values (for those that are not unbound). A number of operations on the variables in the store are allowed:

- Creation of a new, unbound variable
- Binding of an unbound variable to another unbound variable
- Binding of an unbound variable to a bound variable (to a value)
- Unification of two bound variables (of two values)
- Testing whether a variable is bound or not
- Retrieving the value of a bound variable
- Testing the equivalence between two variables.



- Execution state: The execution state of the machine is a snapshot of the semantic stack and the single assignment store.

- Computations: A computation is a stepwise execution of a program. One step involves the following operations on the abstract machine:

1. Pop the topmost semantic statement from the semantic stack
2. Execute the enclosed statement using mappings in the enclosed environment and bindings in the single assignment store.
3. Modify the semantic stack and the store as needed
4. If the semantic stack is not empty, repeat!

- How does a computation proceed? Let <statement>0 be a program. The abstract machine is initialized as follows:

1. Create an empty environment:
   E0 = {}
2. Create a semantic statement:
   s0 = (<statement>0, E0)
3. Create a semantic stack and push onto it the initial semantic statement

S0 = [s0]

4. Create an empty single store:
   σ0 = {}
5. Create the initial state of the abstract machine:
   M0 = (S0, σ0)

Syntaktisk sukker for DSKL:

<statement> ::== local <id> in <statement> end --->\ <statement> ::== local {<id>}+ in <statement> end

Unification:

If two complete values are of the same type and have the same structure and content, unification succeeds; no additional binding is performed.	Declare X Y in X = tree(leaf leaf) Y = tree(leaf leaf) X = Y
If a partial value is unified with another (possibly partial) value, unification succeeds if the existing content can be unified; additional binding is performed as needed.	declare X Y in X = tree(_ leaf) Y = tree(leaf _) X=Y
If two(partial or complete) values do not match at some place of their structure, unification fails; Additional binding may be performed.	declare X Y in X = tree(leaf leaf) Y = tree(leaf bird) X=Y
Compile-time unfication failure: In some cases, unification failures are discovered at the compile time. (1) This statement causes a compile-time failure (2) The program is actually not executed, and this statement produces no output	declare X Y in {Browse Y} % (1) X = tree(leaf leaf) Y = tree(bird Y) % (2)
Run-time unification failure: In some cases, unification failures are discovered at the run time. Y will be displayed.	declare X Y P in X = tree(leaf leaf) proc {P X Y} X = tree(worm Y) end {Browse Y} {P X Y}

Syntax & Semantics

Semantics:

- Semantics is a spesification of the meaning of (program in) the language, what a program does when executed.
- Semantics of a (programming) language can be specified by description, by calculus, by a set of rules defining what happens when a program is executed.

Syntax:

- A syntax explains the allowed structure in a language, usually by using grammar.
- The syntax of a (programming) language is a set of rules defining which sequences of symbols are recognized as sentences, i.e., which constructs are legal in the language; a specification of the form of (programs in) the language.
- It decides how groups of symbols(tokens) can fit together to make a program(macrosyntax) and also how a token can be built up (macrosyntax).

Makrosyntax:

Syntaktisk sukker:

- Is a syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for humans to use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.
- Examples:

	Without syntactic sugar	**With syntactic sugar**
Nested values	X = 1 Y = y(x:X) Z = z(y:Y)	Z = z(y:y(x:1))

Multiple variable declarations	local X in
	  local Y in
	  ….
	  end
	end
		local X Y in
		  …..
		end

Variable initialization	local X in
	  X = 1
	  …..
	end
		local X = 1 in
		  …..
		end

Nesting markers	local Z in
	  {Procedure X Y Z}
	  {Browse Z}
	end
		local Z = {Procedure X Y $} in
		  {Browse Z}
		end

Linguistic abstraction(Språklige abstraksjoner):

- adding linguistic abstractions -- syntactic conveniences that not only allow for more concise programs, but also introduce new programming structures, without extending the model of computation.

	Without linguistic abstraction	**With linguistic abstraction**
Functions	Procedure = proc {$ Argument Result} Result= 2*Argument end {Procedure 1 X}	Function = fun {$ Argument} 2*Argument end X = {Function 1}
Procedures as functions	Double = proc {$ input Output} Output = 2*Input end local Four in {Double 2 Four}... end local Four = {Double 2 $} in … end local Four = {Double 2} in … end	
Record field accession	case R of label(feature:1_ feature2:_ feature3:Value3) then X= Value3 end	case R of label(feature3:Value...) then X = Value end
Record field accession	case R of record(feature:Value...) then X = Value end	X = R.feature
Record labels	case R of Label(...) then X = Label end	X = {Record.label R}

Syntax tree/parse-tree:

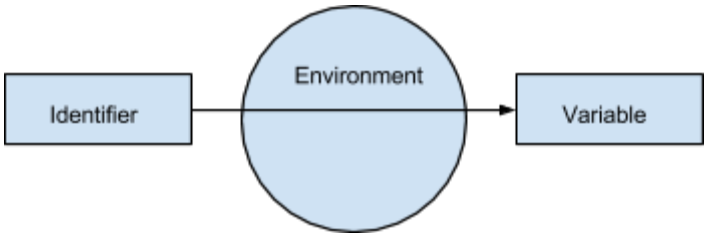
Syntax vs. Semantics:

- Syntax - Definition of the form of programs in the language. Specifies which sequences of symbols are valid , and which are not.
- Semantics - Definition of the meaning of programs in a language. Specifies what the computer has to do during an execution of a program.

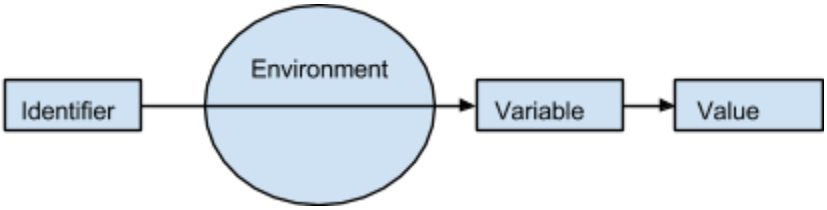
Identifiers and variables:

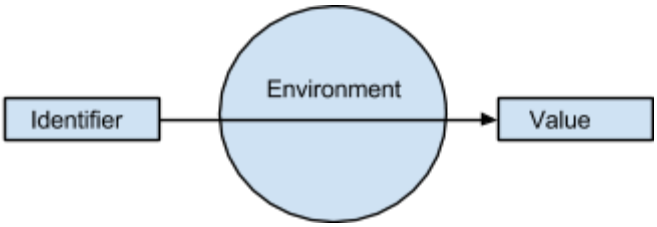
Identifier: is a syntactic entity, it is a part of the program as a sequence of tokens
Variable: is a semantic entity, it is a part of the execution of a program.

- We can declare a variable without binding it to a value



- We can declare a variable and bind it to a value





Deklarativitet :

- An operation is declarative if, whenever called with the same arguments, it returns the same arguments, it returns the same result independent of any other computation state.
- Alternativt: A program is declarative if it specifies what should be achived, not how it schould be achieved.

Variable declaration:
All variables have to be declared before they are used. In other words, any identifier has to be mapped to variable before it appear in statements other than variable declarations

Local scope:	**Global scope:**
local X in ….. end	declare X in ….

Partial value: can be a variable that has not yet been assigned to a value (unbound variable).
Unbound variables and freezing: If an unbound variable is used in an operation that needs the value of the variable, the computation freezes.

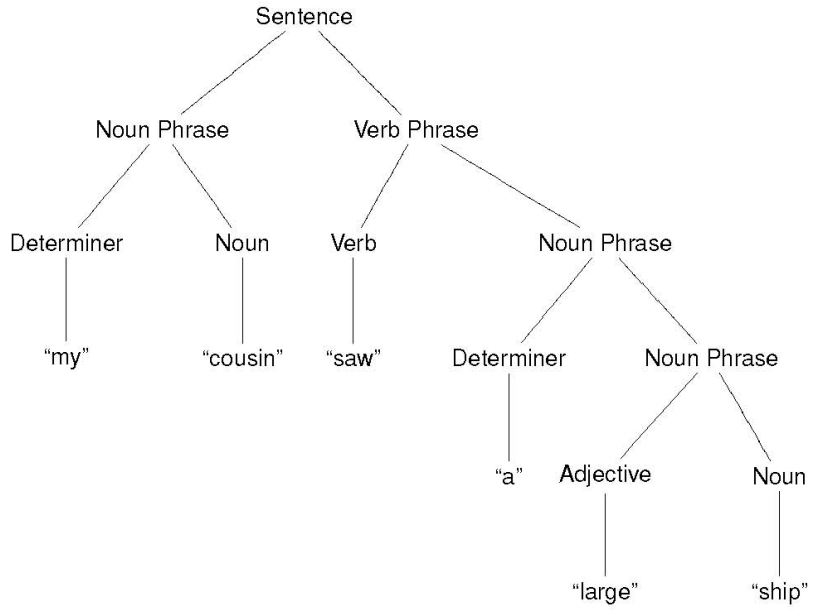
declare X Y in
  Y = X + 2
  {Browse Y}
end
	The computation will freeze because X is never assigned to a value and will be unbound → Y will be unbound until X is bound.

Syntactic analysis of program:

1. The initial input is linear - it is a sequence of symbols from the alphabet of characters
2. A lexical analyzer (scanner, lexer, tokenizer) reads the sequence of characters and outputs a sequence of tokens
3. A parser reads a sequence of tokens and outputs a structures (typically non-linear) internal representation of the program - a syntax tree (parse tree)

Program: if X == 1 then …
Input: 'i' 'f' ' ' 'X' ' ' '=' '=' ' ' 't' 'h' 'e' 'n' ….
Lexemization: 'if' 'X' '==' '1' 'then' …
Tokenization: key('if') var('X') op('==') int(1) key('then')...
Parsing: program(ifthenelse(eq(var('X')
int(1))
…..
…..)
…..)
Interpretation: Actions according to the program and language semantics
Compilation: code generation according to the program and language semantics

Scanning:
Scanning is the process of translating programs from the string-of-characters input format into the sequence-of-tokens intermediate format. Lexemizer + tokenizer = scanning.
Lexer:
The lexemizer took as input a string of characters and returned a sequence of lexemes
Tokenizer:
The tokenizer took as input a sequence of lexemes and returned a sequence of tokens.
Parser:
- Is a function (a program) that takes a input a sequence of tokens (the output of a lexer) and returns a nested data structure corresponding to a parse tree.
- Reads a sequence of tokens and outputs an abstract syntax tree
Interpreter(tolk): reads program code input as text and evaluates and prints the result of executing code.
Compilation:
Syntax tree/parse tree:
- Is a structured representation of a program



Ambiguity/Tvetydighet:

- A grammar is ambiguous if a sentence can be parsed in more than one way :
    1. the program has more than one parse tree
    2. the program has more than one leftmost derivations

- løsninger på tvetydighet:
    1. Obliatory parentheses,
    2. Precendence of operators,
    3. Associativity of operators,
    4. Combine all 3.

Grammatikk/Grammar & Language:

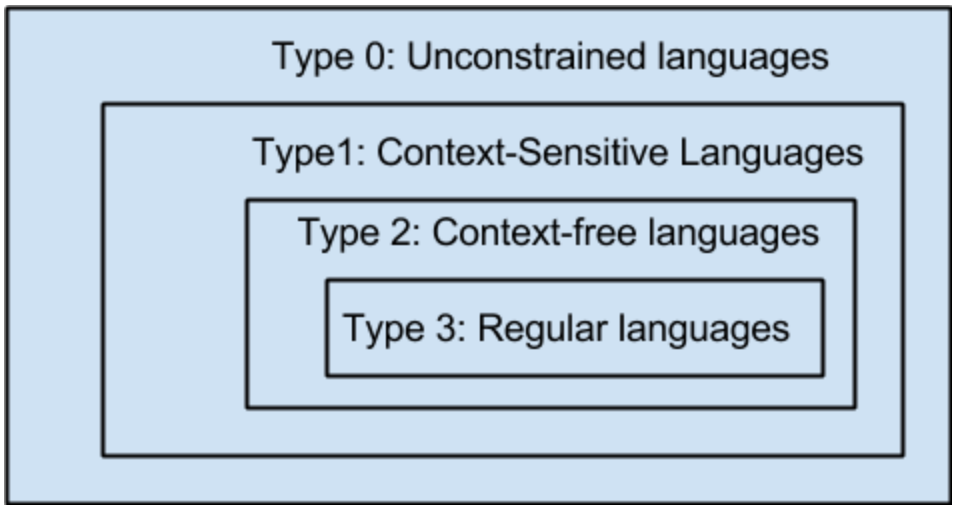
Terminal vs. ikke-terminal:

<sentence> ::== e | a<letter>b | aa<letter> | b<letter>
<letter> ::== b | a

<sentence> og <letter> = ikketerminaler
a,b, etc = terminaler

Forskjellige typer grammatikk:

- unconstrained: include all formal grammars.
- context-sensitiv: These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule.
- context-free: These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals.
- regular: Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule.



Backus-Naur Form:

Metasyntactic extensions:

- '[' and ']' is used to enclose a subsequence that appears in the sting at most once
- '{' and '}' is used to enclose a subsequence that appears in the string any number of times
- '{ <sequence> }^2' means two subsequent occurences of <sequence>
- '{ <sequence> }+' means at least ine occurence of <sequence>
- '{ <sequence> }*' means any number of occurences of <sequence>

Derivations:

We kan derive sentences in the language L(G) specified by a grammar G in a sequence og steps:

1. In each step we transform one sentential form (a sequence of terminals and/or non-terminals) into another sentential form by replacing one non-terminal the right-hand side of a matching rule.
2. The first sentential form is the start variable (root) alone.
3. The last sentential form is a valid sentence, composed only of terminals (Ingen ikke-terminaler!)

Sequences of sentential forms starting with the root and ending with a sentence L(G) obtained as specified above are called derivations.

Leftmost derivation:	**Rightmost derivation:**
1. <stmt> 2. if <var> then <stmt> else <stmt> end 3. if A then <stmt> else <stmt> end 4. if A then skip else <stmt> end ……	1. <stmt> 2. if <var> then <stmt> else <stmt> end 3. if <var> then <stmt> else if <var> then <stmt> else <stmt> end end ……

Formal Language:

- A formal language L is a set of strings of symbols from a finite alphabet A.
- If a particular string cannot be generated from a grammar, it is not a valid sentence in the formal language defined by the grammar.

Formal grammar:

- A formal grammar G contains: a finite set of variables, a finite set of symbols, a finite set of rules and a start variable (root).

Paradigmer: Is a style of programming, the way in witch solutions to problems are implemented in a programming language. Her er noen paradigmer:

Imparativ programming:

- A program that is a sequence of statements. Statements update the variables.

Object oriented programming:

- A program defines a set of objects that are encapsulations of data and operations (methods).
- Models are entities in the real world. Separation of class and instance.
- Separation of state(attributes/properties) and behaviour (methods/features).
- Objects send messages to (call method on) each other.
- Objects execute operations when they receive messages.

Functional programming:

- A program defines a set of functions.
-Functions, when called with appripriate arguments, compute values based on input.
- Functions are first-class objects: they can be both arguments to and return values from calls to other functions.
- In pure functional programming there are non side-effects -- non mutable variables, no I/O.
- in functional programming, a subroutine (a function) specifies the relation between the arguments and the result of a computation, which is independent of any mutable state.

Concurrent programming:

Programming with lazy execution:

Declarative programming:

- Given a specific input always produce the sameoutput
- The result cannot be influenced by changes in the state of some objects external to the operation;
- The operation does not have any internal state that can be changed

Logical Programming:

- A program is a set of logical assertions (acts and rules).
- A program may be read as a logical expression or as a set of operations to be executed.
- An execution of a program follows an inference pattern to prove or disprove a query.

Multiparadigm language: OZ!

Memory management, bindings and scope:

Bindings:

- Binding time: is the time at which the meaning of a syntactic entity is established.
- Language implementation time bindings: The meaning of syntactic entities is provided as a part of an implementation of the language.
- Compile time bindings: The meaning of syntactic entities is established when the compiler produces object code.
- Link time bindings: The meaning of syntactic entities is established when the object code is liked with, e.g., static libraries.
  → References to variables, procures, etc. from external modueles are typically resolved at the link time.
- Load time bindings: The meaning of static entities is established when the (compiled and linked) program is loaded into memory, still prior to its execution
  → The physical (or virtual) addresses of variables, procedure, ets are typically resolved at the load time.
- Run time bindings: The meaning of syntactic entities is established when the program is actually executed.
  → The location and content of, e.g., local variables are typically fixed at the run time.
  → Scripting kanguages and pure interpreted languages usually rely to a lagre extent on run time binding
- Late and early bindings:
  → Early binding times: all binding times up to the load time, inclusive.
  → Late binding times: run time binding , including start-up time, module entry time, elaboration time, procedure call time, block entry time, etc.

Memory allocation:

- Static: The location of objects in memory is fixed at load time, and the allocation persists throughout the whole execution of the program.
- Stack-based: The allocation of objects in memory is done at the run time in a last-in, first-out order, and the allocation persist as long as needed.
- Heap-based: The allocation of objects in memory is done at the run time within a region of memory called the “heap”.
- Automatic and manual memory allocation and dellocation

Lifetime: The lifetime of an object or of reference is the interval during an execution of a program when the object or the reference persist in memory

- Object lifetime:
- Reference lifetime:

Garbage collecotor: finds out which objects on the heap are no longer accessible, and removes them.

Scope: The scope of a binding is an enclosing context where the binding is in effect.

- Static scoping: a scope is a textual region of code
- Dynamic scoping: a scope is a fragment of a proceeding computation.
- Lexiacal scope: A lexically scoped bindings is in effect from the point o declaration to the end of the respective block (local X in ….. end), but expect for when it is shadowed (hidden) by a

binding in a nested block (local X in (local Y in …..end) end) → Y er hidden inni sitt scope.

Concurrency/Samtidighet

Concurrency: is the progressing of two or more activities (processes, tasks, threads) in parallel.

- Data-driven concurrency: a thread of computation can be executed as soon as it has all data it needs
- Demand-driven concurrency: a thread of computation can be executed no sooner than when the result of its execution is needed by another thread.
- With concurrent execution is many possible orders of execution possible
- Non-determinism: is a feature of computation which, at a certain point, can arbitrarily choose among two or more ways to proceed further. A non-deterministic program can be declarative!

Threads:

Concurrency and exceptions:

```
local X in
   thread try X=1 catch _ then skip end end
   thread try X=2 catch _ then skip end end
end
```

- On every execution, there will be a unification failure! → **declarative**
- BUT, the failure is caught by the exception handling statement catch, and the program completes gracefully with X bound to 1 or to 2 → **non-deterministic**

Deadlock with threads:

```
local X in
    thread if X==1 then skip else X=0 end end
    thread if X==0 then skip else X=1 end end
    {Browse X}
end
```

- There will be no output! Both threads freeze over the comparison statement since X is unbound; the program never terminates.
- This is a simple example of a deadlock, where X is the resource and both threads wait for the resource to become bound (....that it never will be).

```
local T1 T2 in
  thread T1 = {Thread.this} {Thread.terminate T2} X=1 end
  thread T2 = {Thread.this} {Thread.terminate T1} X=2 end
  {Browse X}
end
```

- One of the threads succeeds in terminating the other, and then binds X
- Either 1 or 2 will be displayed, non-deterministically

Thread extension to the model of computation:

- the semantic stack is replaced by a multiset of semantic stacks

- each semantic stack in the multiset corresponds to one thread
- Why multiset?: Because two threads may happen to have identical semantic statements in their corresponding stacks → their stacks can be identical.
- The choice of a stack is not specified by the semantics(it is from this point of view non-deterministic). It is an implementational detail of the scheduler.

<statement> ::== ….. **| thread <statement> end**
Semantics of the thread statement (thread <statement> end, E) 1. Create a new semantic stack 2. Put onto the new stack the semantic statement (<statement>, E)

Lazy: Lazy evaluation is an approach to execution of programs in which a computation is postponed until its result is needed.

- Lazy evaluation is extensively used for demand-driven computations
- Lazy evaluation is the opposite of eager evaluation, used in data-driven computations
- Some programming languages have features that enable programming with lazy evaluation; in some, lazy evaluation is the default approach to computation
- Lat utførelse kan være nyttig å bruke i uendelige strømmer.
- {ByNeed Procedure Result} means that procedure should be applied to Result in a separate thread, and only when there is a need for the value of result.
- Lazy functions are a linguistic abstraction for ByNeed execution

When is a variable needed (lazy execution)?:

1. There is a thread that has been suspended waiting for the variable to become bound
2. There is an attempt to bind (unify) the variable
3. The variable is already bound

fun lazy {LazyIncrease N} N+1 end (1) M = {LazyIncrease N} (2) N = {LazyIncrease 1} (3) O = {LazyIncrease M+1} Hvis jeg putter en browse mellom hver utregning vil jeg få: (1) → **3** (2) → **2** (3) → **_**	(1) M og N trengs ingen plass (2) N trengs ingen plass (3) O trengs ingen plass, men M trengs. Det at M trengs medfører at N og trengs! O → M → N **NOTE:** We do not need to compute (M+1)+1, but we need to compute M+1 to pass it to LazyIncrease, thus M is needed.
{Browse [{LazyIncrease 1} {LazyIncrease 1}+1]}	Resultatet blir: **[_ 3]**

<statement> ::== ....
| {ByNeed <id> <id>}

Need an extension of the abstract kernel language machine → **we add a trigger store**

Semantics of ByNeed statement
({ByNeed <id>1 <id>2}, E)

1. If E(<id>2) is not bound, add to the trigger store the pair (a trigger):
   (E(<id>1), E(<id>2))
2. Otherwise, create a new thread with the semantic statement
   ({<id>1 <id>2}, E)

The Execution rule for a ByNeed statement:
If there is a need for the variable v and the trigger store contains a trigger for that variable - a pair (v',v) with v as the second element, then:

1. Remove the trigger, ans
2. Create a new thread with the semantic statement:
   ({<id>1 <id>2}, {<id>1 → v', <id>2 → v})

Rules related to memory management:

1. If there is in the trigger store a trigger (v1, v2), and v2 is reachable, then v1 is reachable
2. If there is in the trigger store a trigger (v1, v2), and v2 is not reachable, then the trigger can be removed!

Streams: A stream is a potentially infinite data structure (read and extend):

- elements can be read form the beginning
- elements can be added to the end
- A stream is an abstraction for the flow of data.
- Producer (adds elements..) and consumer (reads the elements that the producer produce)

fun {EagerFib N} {Fib N} end **fun lazy {LazyFib} {Fib N} end** **{Browse [{EagerFib 1} {LazyFib 1} {LazyFib 1}+0]}**	Dette vil vise: **[1 _ 1]** 1) Eager vil kjøre som vanlig 2) LazyFib trengs ikke å kjøre, så den forblir ubundet 3) Den siste trengs for å fullføre den aritmetriske operasjonen og vil defor bli utregna!
fun {Integers} .... generer tall i lat utførelse... end **(1)** **Ints = {Integers}** **(2)** **{Browse Ints}** **(3)** **{Browse {List.take Ints 5}}**	(1) Ubundet. Integers blir ikke kjørt før Ints trenges. (2) Ints er ubundet siden den utføres med lat utførelse, så det den viser er _ (3) Tvinger Ints til å produsere de 5 første tallene,

(4) {Browse Ints} **(5) {Browse {Integers}}**	vil da vise: **[1 2 3 4 5]** (4) Siden Ints produserte de 5 første, så vil det vises **1|2|3|4|5|_** (5) Creates a new promise, and **_** is browsed.

Consumer & Producer:

- The slow consumer problem (demand driven stream pipelines)

Unntak/Exceptions:

Exceptions: Most modern programming languages provide a convenient abstraction for handling pathological situations - exceptions. Spescial syntactic constructs are used to:

- set handlers for arriving exceptions (catch, rescue, except)
- generate exceptions (throw, raise)
- preorm cleanup actons (finally, ensure)
- **The try block** contains statement which may raise an exception
- **The catch statement** is executed in case there is an exception raised in the try block
- **The finally block** is executed irrespectively of whether there is or is not an exception raised in the try block, and of whether such an exception is or is not caught in the catch block. It is typical to use this to do clean-up operations.

Extentions of the kernal language:

<statement> ::== ....
| try <statement>
catch <id> then <statement>
finally <statement> end
| raise <id> end

Semantics of the try statement:
(try <statement>1 catch <id> then <statement>2 end, E)

1. Push onto the stack the semantic statement
   (catch <id> then <statement>2 end, E)
2. Push onto the stack the semantic statement
   (<statement>1, E)

Semantics of the raise statement:
(raise <id> end, E)

1. If the semantic stack is empty, stop and report an uncaught exception
2. Else pop the first semantic statement from stack. If it is not catch statement, go to step 1.
3. The popped statement is:
   (catch <id>c then <statement>c end, Ec)
   Push onto the stack the semantic statement
   (<statement>c, Ec + {<id>c → E(<id>)})

Semantics of the catch statement:
(catch <id> then <statement> end, E)

1. Do nothing (the catch statement is equivalent to the skip statement)

Message Passing Concurrency

- **Messages can be sent:**
  - Synchroously: Waiting for a reply after each message
  - Asynchronously: One after another without waiting for the replay
- **Messages may contain:**
  - Simple or complex data (numbers, strings, records, etc.)
  - dataflow variables (unbound ones to be bound by the receiver of the message)
  - continuations (procedures to be executed later in time)
- **Message passing with Ports:**
  - A port is associated with a stream
  - A port provides an entry point for extending the associated stream
  - A port is a non-declarative object with internal mutalbale state, used to point to the unbound end og the associated stream
  - Sending a message to a port result in the stream being extended and the port's state being updated to the new unbound end.
  - A port can be used to extend the associated stream by more than one sender.
  - **{NewPort Stream}** returns a new port associated with Stream
  - **{Send Port Value}** causes Port to extend Stream with Value and a new unbound variable.
  - Ports are thread-safe
- **Port Object:** is a comination of (1) an arbitrary number of ports, and (2) a stream-processing thread (a stream handler.

```
P0 = {PortObject proc {$ Message} {Browse received(Message)} end}
{Browse P0.stream}
{Send P0.port 1}
{Send P0.port 2}
```

Non-declarative port: **declare Port Stream in** **(1) Port = {NewPort Stream}** **(2) {Send Port 1}** **(3) {Send Port 2}**	(1) Stream = _ (2) Stream = 1|_ (3) Stream = 1|2|_
Declarative port: Declare Port1 Port2 Port3 in Port1 = {NewPort Stream} (1) Stream = _ Port2 = {Send Port1 1} (2) Stream = 1|_ Port3 = {Send Port1 2} (3) Unification failure	
Thread safe ports: thread {Send Port 1} end	There is no way for these two statements to cause a unification failure. When a message is sent to a port, the port

thread {Send Port 2} end	receives the message, creates a new unbound end, updates its own state, and places the message onto the stream. It is impossible for two threads to have a port attempt to post two different messages in the same place in the stream.

Extenstions to the kernal language:

Need an extension of the abstract kernel language machine → **we add a mutable store**

Semantics of the NewPort statement
({NewPort <id>1 <id>2}, E)

1. Bind E(<id>2) to a new name, n, in the single assignment store
2. If (1) successful (E(<id>2) was originally unbound)
   - create a new variable v in the store, and bind E(<id>1) to a read only-view of this variable (E(<id>1) = !!v)
   - add the pair (n, v) to the mutable store
   Otherwise (E(<id>2) was already bound), raise an error condition

Semantics of the Send statement
({Send <id>1 <id>2}, E)

1. If E(<id>1) is bound, and there is a pair (E(<id1), v) in the mutable store (means that E(<id>1) i a port., then;
   1) Create a new variable v' in the single assignment store
   2) Replace (E(<id>1),v) with (E(<id>1), v') in the mutable store
   3) Bind v to a new record '|'(E(<id>2) !!v').
2. If E(<id>1) is bound, but there is no pair (E(<id>1),v) in the mutable store (meaning that E(<id>1) is not a port), then raise error
3. If E(<id>1) is unbound, suspend the execution

Rules for memory management:

- If the mutable store contains a pair (p,v) and p is reachable, then v is reachable
- If the mutable store contains a pair (p,v) and p is not reachable, then the pair (p,v) may be removed from the mutable store.

Explicit state:

- **What is state?** According to CTMCP, a state is a sequence of values in time that contains the intermediate results of a desired computation. ("Verdien i dette øyeblikk").

- An object that has **immutable state** (its state can not change) is said to be **stateless**, or to have **implicit state.**
- An object that has **mutable state** (its state may change) is said to be **stateful**, or have **explicit state**.

- State threading: Et eksempel med en funksjon {Sum Numbers} summerer alle tallene:

- Immutable states:
  {Sum [1 2]} → {Sum [1 2] 0} → {Sum [2] 1} → {Sum nil 3} → 3
- Mutable states:
  {Sum [1 2]} → {Sum} → {Sum} → {Sum} → 3

- State Updating, Cells:

- **Cells:**
  - **Lage ny celle:** Cell = {NewCell value}
  - **Oppdatere celle:** Cell := @Cell + value
  - **Gi Celle helt ny verdi:** Cell := value (Her overskrives den gamle verdien helt)
  - **Referere/hente verdi:** @Cell
  - **Exchange:** {Exchange Cell OldContent NewContent} Unifies the variable OldContent with the original content of the cell Cell and replaces the content of Cell with NewContent.
    Dette er den egentlige syntaksen! De ovenfor er syntaktisk sukker (Gi ny verdi og oppdatere verdi)

```
fun {Sum Numbers}
  Result = {NewCell 0}
  …..
  proc {Sum}
       case ….
            Result := @Result + Number
       …..
   @Result
end
```

- Mutable vs. Immutable state

- **Mutable**
  - Because the underlying machine performs computations with mutable state
  - Because non-declarative implementations tend to be faster than declarative implementations
  - Because most algorithms are explained in terms of updating the state of a data structure
  - Because the reality is stateful and modeling with mutable state is often more approperiate
- **Immutable:**
  - Because solutions to many problems are more intuitive in the declarative form than in

the non-declarative form; the translation from an abstract algorithm to an implementation is usually easier in the declarative approach
    - Because declarative programming is safer than non-declarative programming, particularly in a concurrent environment
    - Because languages with high level of abstraction declarative techniques are often more efficient than explicit operations on mutable state.

- **Syntactic sugar for cells:**
  - Old = @Cell is syntactic sugar for {Exchange Cell Old Old}
  - Cell := New is syntactic sugar for {Exchange Cell _ New}
  - Old = Cell := New is syntactic sugar for {Exchange Cell Old New}

- Equality and Identity:
  - **Idenity:** An object is identical to itself, and no two distinct objects are identical
  - **Equality:** Two objects can be equal if they are composed of the same or equal elements

- Aliasing: An alias is a variable that has exactly the same content as another variable. An alias is an identifier (a variable name) that maps to the same variable as another identifier, which it is an alias for.

- **Argument passing (parameter passing):** An argument passing mechanism is a strategy for how arguments in a procedure call are received in the procedure body.
  - **Parameters** (formal parameters), variables (or identifiers) included in a procedure declaration
  - **Arguments** (actual parameters), variables (or values) included in a procedure call.
  - **Call by value:**
  - **Call by reference:** the formal parameter (the identifier) is an alias to the actual argument
  - **Call by result**
  - **Call by need**
  - **Cal by name**
  - **Default arguments**

C1 = {NewCell 1} C2 = {NewCell 1} C1 = C2 %Ooops, failure {Browse C1 == C2} %Prints false @C1 = @C2 % succeeds	Det er forskjell på verdier og objekter!! C1 og C2 er to helt forskjellige objekter. @C1 og @C2 er bare en ref. til verdiene av "objektet".

<Statement> ::== …..
| {NewCell <id> <id>}
| {Exchange <id> <id> <id>}

Semantics of NewCell Statements
({NewCell <id>1 <id>2}, E)

1. Create a fresh name n
2. Bind E(<id>2) to n in the single assignment store
3. Ass to the mutable store the pair (E(<id>2), E(<id1))

Semantic of Exchange statements
({Exchange <id>1 <id>2 <id>3}, E)

1. If E(<id>1) is a cell, i.e., E(<id>1) is bound to a name n and there is in the mutable store then:
   - Unify E(<id>2) with v
   - Replace, in the mutable store, the pair (E(<id>1), v) with the pair (E(<id>1),E(<id>3))
2. If E(<id>1) is unbound, suspend the computation
3. Otherwise (E(<id>1) is bound, but not a cell) raise an exception.

Memory management:
Two additional memory management rules are needed for garbage collection:

- If the mutable store contains a pair (v1, v2) and v1 is reachable, then v2 is reachable.
- If the mutable store contains a pair (v1, v2) and v1 is not reachable, then the pair (v1, v2) can be removed from the mutable store.

- Memoization: Is a technique that allows, certain circumstances, to avoid repetitive computations specified by multiple occurences of the same expression

- Is typically used in functional programming
- A momoized function remembers the results of computations for arguments with which it has been called
- When a memoized function is applied to arguments it has already been applied to earlier, it simply returns the precomputed result instead of computing it a new.
- Memoization may improve the performance of a program, if the computation is costly and the function is repetively called with the same arguments
- Memoization may worsen the performance of a program, if the computation is cheap and the function is seldom called with the same arguments more than once.

Relational Programming

- What is rational programming?: is an approach to programming where procedures are throuht of as relations between values, and where no clear distinction between input and output is made.

- We have used functions to compute and return values, and proceudres to perform side effects
- In declarative model, a function always returns the same output given the same input
- In a non-declarative model, a function may return different values on different calls with the same input, but the function still works one way
- In relation programming, procedures are used to assert or query relations between arguments.

Div definisjoner:

- Functions and procedures:

- Functions are linguistic abstractions over procedures
- Any function definition and application can be translated into a corresponding procedure definition and application
- A function of n arguments is a procedure of n+1 arguments - a functions return value is bound ti an additional, implicit argument (?out).
- A procedure may have more than one 'result' argument, but only one of them is implicit.
- A function can also have more than one 'result' argument, but only one of them is implicit
- For convenience, 'result' arguments may be marked with the prefix '?'
- Since functions are a linguistic abstraction over procedures, it should be possible to use them interchangeably.
- Procedure (and function) value expression result in unique objects, just like calls to NewName. proc{$} skip end == proc {$} skip end % false!

Procedure = proc {$?X} X=1 end
{Browse{Procedure}}
	Function = fun{$} 1 end
	local Result in
	  {Function Result}
	  {Browse Result}
	end

- Akkumulere
- Explicit State
- Beregningsmodell/Kjernespråk
- Parameteroverføring: (Call by: reference, variable, value, value-result, name and need)
- Porter og strømmer
- Nedstigningsparsing/Recursive decent parsing

-Procedural abstraction: Ant piece of code can be wrapped into a procedure , and later reused by simply invoking (calling, applying) the procedure

- Programmable with observational declarativeness: declarative programming in a language that does not guarantee declarativity (e.g., Java).

- Programmable with definitional declarativeness: programming in a language that guarantees declarativeness.

- Genericity: A procedure can be generalized by abstracting some part of the code in its body into an argument. The abstracted functionality can be supplied, in different versions, by calling the procedure with different values for the extra argument.

- Instantiation: Instead of producing direct results, a generic procedure can return more specific procedures that, when called with approperiate arguments, compute the final result. The spesific procedures 'instantiate a generic abstraction.

- Embedding: Procedures can be stored in data structures, which make it possible, to call a number of

procedures in an iterative loop rather than manually.

- Deterministisk

- its behaviour is consistent
- its behaviour is predictable

- Recursion: is a fundamental concept in mathematics, and in computer science - in particular, in functional programming.

- A recursive procedure is one that calls itself within its own body.
- A recursive computation is one in which a recursive procedure is called repetitively so that the result of each call depends on the result of the subsequent calls.
- Recursive computations need appropriate stopping conditions!!
- tail-recursion?
- halvrekursivt?

- Higher order programming (HOP):

- Mapping is a HOP technique where we apply a function to all elements of a list.
  Map is a function that takes two arguments: a list and a one argument function, and returns a list of the reuslts of application of the function to every element of the input list.
  {Map [1 2 3] fun {$ Item} Item+1 end} → [2 3 4]
- HOP is a programming paradigm where procedures (functions) are first-class objects, they can be:
  - Passed as argument in proceedure and function calls
  - returned as values from function calls
  - embedded as elements in datastructures

- **Expression and statement:**

- Expression: is a syntactic construct witch specifies an operation that evaluates to a value
- Statement: is a syntactic construct which specifies an operation that does not evaluate to a value.

Expressions:	Statements:
● '2 + 2' evaluates to 4 ● 'A==B' evaluates to true of false ● 'proc {$ X} X+1 end' evaluates to a procedure ● '{fun {$ X} X+1 end 1} evaluates to the result of an application of an anonymous function to the value 1, which is 2.	● 'X=1' unifies X with 1 ● '{Browse X+1}' displays the value of X+1 ● 'skip' causes the execution to proceed without any action, a 'no-operation' statement. ● '{proc {$} {Browse done} end}' applies a procedure of no arguments.

Diverse kode og eksamenseksempler:

Midterm 2011, task 3:
Interpreting the following expression in Mozart: **{Browse local Y in local X=[1 2] in X.1=Y end end}**
This will display: 1

Midterm 2012, task 3:
Interpreting the following expression in Mozart: **{Browse local Y in local X=[1 2] in X.1=Y.1 end end}**
This will display: will display nothing, since it will suspend forever.
X.1=Y.1 ? Y er ikke en liste enda, så det kan man vel ikke skrive!

Midterm 2011, task4:
When is a grammar ambiguous? Give an example grammar:
A grammar is ambigous when:

1. Has more than 1 parse tree
2. When it has more than one left derivation

Ambigous grammar:
<S> ::== S+S | S-S |<digit>

Midterm 2011, task 8,9:

```
declare fun {Dobop L}
   case L of _|T then
      1 + {Dobop T}
   else 0 end
end
Declare X = 1|2|3|nil
{Browse {Dobop 1|2|3|X}}
```

→ Will show 6
→ If we change X to X = 1|2|3|_, then the computation freezes.

Midterm 2010, task 22:
Which are other representation of [a b c]?

1. (a|b|c|_)#_
2. [a b c] | nil
3. [a b c nil]
4. '|' (1:a 2:'|'(b '|' (c nil)))
5. '|'(a '|' (b '|' (c nil)))
6. c|b|a| nil
7. a|b|c
8. (a|b|c|X)#X