CS2104

HASKELL

```
error :: [Char] -> a
show :: Show a => a -> String
zip :: [a] -> [b] -> [(a, b)]

hoo :: (t1 -> t2 -> t2) -> t1 -> t2
hoo g x = g x (hoo g x)
```

PROLOG

An untyped (or dynamically-typed) language

Atoms, Terms, Variables

- Atoms are constants. They must start with a lowercase letter
- Variables denote unknown values to be computed. They must start with a uppercase letter or underscore
- Terms are used to form tree-like data structures, e.g. `cons(2,nil)`
- Can mix terms with variables, e.g. `node(X,Y)`

Relations

```
% Each predicate states a fact
% as a relation
father(john, mary).
male(john).
female(mary).

% Queries
?- father(X, mary).
X = john.
?- father(john, X), male(X).
false.
?- male(X); female(X).
X = john ;
X = mary.
```

- `,` denotes conjunction (AND)
- `;` denotes disjunction (OR)

Alternative for disjunction

```
% Disjuncion implied
par(X,Y) :- father(X,Y).
par(X,Y) :- mother(X,Y).

% Equivalent:
par(X,Y) :- father(X,Y); mother(X,Y).
```

Number of solutions

- No solutions: `false.` is shown immediately
- 1 solution: solution is shown immediately with full stop.
- Multiple solutions: one solution is shown
  - Typing `.` stops displaying other possible solns
  - Typing `;` goes to the next soln

Horn clauses

```
% Horn clause
res(X) :- father(john, X), female(X).
?- res(X).
X = mary.
```

- Can store logical formula into a single predicate

```
% Good
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

% Bad (infinite loop)
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).
```

- Can be recursive, but left recursion may result in infinite loop

Unification

- Mechanism that Prolog uses to match two terms
- Unification `t1=t2` may contain variables. The system tries to compute a substitution for the variables to make the two terms equal
- Once a variable is bound, it cannot be changed

```
a=X             % X=a
a=b             % fail, unique atoms
                % are diff
n(a,X) = n(Y,b) % X=b, Y=a
n(a,X) = n(X,b) % fail, X cannot be
                % both a and b
n(a,X) = n(X,a) % X=a
```

Algorithm

1. Given initial unification request $\Sigma_i = \Pi_1, \Sigma_2 = \Pi_2, \cdots$
2. If Σ_1 or Π_1 is a variable, add $\Sigma_1 = \Pi_1$ to the answer, and apply it as substitution to the remaining list of unifications. Go to last step
3. If the predicate name for Σ_1 and Π_1 are different, exit with failure.
4. If the number of parameters for Σ_1 and Π_1 are different, exit with failure.
5. Denote by $\Sigma_{11}, \Sigma_{12}, \cdots, \Sigma_{1k}$ the arguments of Σ_1, and similarly $\Pi_{11}, \Pi_{12}, \cdots, \Pi_{1k}$ the arguments of Π_1.
6. Set unification request to
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \cdots, \Sigma_{1k} = \Pi_{1k},$$
$$\Sigma_2 = \Pi_2, \cdots$$
7. If unification request is not empty, go to first step. Otherwise, terminate with success

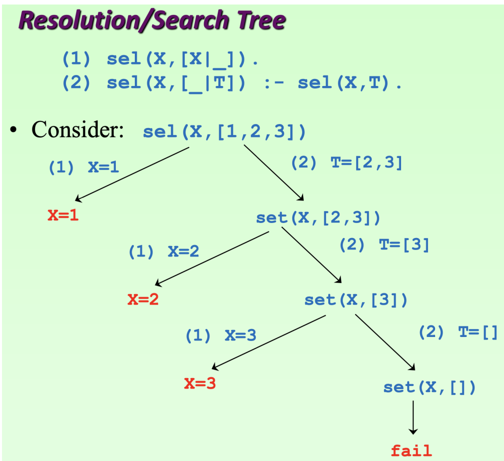
Resolution

- Process of answering a query

Algorithm

- Given query `A1, A2, ... An`
- Pick a matching rule from the program and rename its variables: `H :- B1, B2, ... Bk`
- New goal: `(H=A1), B1,B2,...,Bk, A2,...An`
- Variable bindings may be generated by the unification request `H=A1`. Add them to the answer, replacing bound variables by its substitution over the entire query
- Go back to step 1 if query is not empty. Else, return the answer

Resolution tree example



Lists

Denoted by square brackets with elements separated by comma: `[mary, [], n(A), [1,2,3], X]`

Append

```
% Params may be input/output
append([],Y,Y).
append([X|Xs],Y,[X|Rs]) :- append(Xs,Y,Rs).

% Join lists, Z as output
?- append([1,2],[3],Z).
Z = [1,2,3].

% Compute difference, Y as output
?- append([1,2],Y,[1,2,3]).
Y = [3].

% Split list
?- append(X,Y,[1,2]).
X=[], Y=[1,2];
X=[1], Y=[2];
X=[1,2], Y=[];

% Prefix a list
?- append(X,_,[1,2]).
X=[]; X=[1]; X=[1,2]; X=[1,2,3];
```

- Can represent the backtracking performed by resolution using a search tree
- One outgoing edge per predicate clause
- Answers are found at child nodes. All answers are provided using `;` (OR) to represent multiple possibilities

Reverse

```
rev([],[]).
rev([X|Xs],Y):- rev(Xs,Y2),append(Y2,[X],Y).
?- rev([1,2,3],Y).
Y = [3,2,1].
?- rev(X,[1,2,3]).
X = [3,2,1].
```

- Note: this implementation of reverse results in unterminated search after finding the first (only reasonable) answer.

```
?- rev(X,X).
X = [] ;
X = [_] ;
X = [_A, _A] ;
X = [_A, _, _A] ;
...
```

Select (list membership)

```
sel(X,[X|_]).
sel(X,[_|T]) :- sel(X,T).

% Fails immediately since both clauses
% are not applicable
?- sel(X,[]).
false.

?- sel(3,[1,3,5]).
true.
?- sel(X,[1,3,5]).
X=1; X=3; X=5.
```

Arithmetic

is-Operator

```
X is 3+4    % X=7
7 is 3+4    % succeed
7 is X+4    % fail, RHS must be
            % fully instantiated
```

Comparison

- Relations `>`, `<`, `>=`, `=<`, `==`, `=:=` compares two arithmetic expressions that must be fully instantiated
- Operator `=` is for term unification

Factorial

```
fact(0,1).
fact(N,R) :- N>0, M is N-1, fact(M, R1), R
    is N*R1.

?- fact(5,R).
R = 120.
?- fact(X,120).
% ERROR: >/2: Arguments are not sufficiently
% instantiated. Matches with second clause,
% then N>0 cannot be evaluated.
```

Negation

- Prolog is based on closed world assumption. Instantiation of predicates are true. If the instantiation does not exist, then it is assumed to be false.

```
father(john,mary).

?- father(john,kerry).
false.
```

- If we have the clause `male(X):- not(female(X)).`, then need only specify facts on female
- Above clause says if a person cannot be proven to be female, then assume the person is male

Cut

- Cut operator `!`
- Used to stop backtracking

```
teach(dr_X, compsci).
teach(dr_X, math).
teach(dr_X, physics).
teach(dr_Y, chemistry).
study(alice, chemistry).
study(bob, math).
study(charlie, physics).

% All students taught by dr_X
?- teach(dr_X, Subj), study(Stud, Subj).
Subj = math, Stud = bob;
Subj = physics, Stud = charlie;

% teach first matches on compsci
% cut prevents further backtracking
?- teach(dr_X, Subj), !, study(Stud, Subj).
false.
```

Impure Prolog

- `atom(X)`, `var(X)`, `integer(X)`
- `write(X)` outputs the binding of X

Constraint Solving

- `use_module(library(clpfd)).`
- Regular prolog comparison (e.g. `A > B`) requires both A and B to be instantiated to evaluate
- Constraint solving uses `A #> B`, obtaining `B #=< A + -1`

```
?- X #> 3.
X in 4..sup.
?- X #\= 10.
X in inf..9\/11..sup.
?- 3*X #= 9.
X=3.
?- X*X #= 9.
X=3\/-3.
```

Factorial

- Constraint solving allows both params as input

```
cfact(0,1).
cfact(N,R) :- N #> 0, M #= N-1,
      R #= N*R1, cfact(M, R1).
?- cfact(5,R).
R=120.
?- cfact(N,120).
N=5.
```

Puzzle Solving

```
:- use_module(library(clpfd)).

puzzle([S,E,N,D]+[M,O,R,E]=[M,O,N,E,Y]) :-
  Vars = [S,E,N,D,M,O,R,Y],
  Vars ins 0..9,
  all_different(Vars),
            S*1000 + E*100 + N*10 + D +
            M*1000 + O*100 + R*10 + E #=
  M*10000 + O*1000 + N*100 + E*10 + Y,
  M #\= 0, S #\= 0.

?- puzzle(As+Bs=Cs),label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2];
```

- Without label, we have a partially solved answer
- Label tells the CLP to find solutions for the given variables

SCALA

Misc

Pure vs imperative

- Pure functions are: Easier to reason/debug | Less error prone | Easily composable
- Imperative features are still needed
- Use pure functions where possible, and imperative features where necessary

Mutable class

```
class Point(xc: Int, yc: Int) extends .. {
  var x: Int = xc
  var y: Int = yc
  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
  }
  override def toString(): String
    = "(" + x + ", " + y + ")";
}
```

Modifiers

- `val` for constant. Allows getter
- `var` for mutability. Allows getter, setter
- `private` disallows getter, setter
- `protected` allows access only via base and subclasses

Mutable reference

- Essence of mutation can be captured by a polymorphic `Ref` class with mutation:

```
class Ref[A](v:A):
  var vl = v
  def get: A =
    vl
  def update(nv:A): Unit =
    vl = nv
end Ref
```

Loops

For loop

```
// Prints 0 to 3 (inclusive)
for (i <- 0 to 3)
  println(s"i = $i")
// Prints 3 to 0 (inclusive)
for (i <- 3 to 0 by -1)
  println(s"i = $i")
// Prints 3 to 0 (inclusive)
for (i <- (0 to 3).reverse)
  println(s"i = $i")
```

While loop

```
// Prints 3 to 0 (inclusive)
var i = 3
while (i >= 0) {
  println(s"i = $i")
  i = i-1
}
```

Sequence comprehension

```
// Vector((1,2), (1,3), (2,3))
val f_lst =
  for (i <- 1 to 3; j <- 1 to 3 if i<j)
    yield (i,j)
```

List iterator

```
for (name <- lst) {
  ..code..
}
lst.forEach((name: String) => ..code..)
```

Hash table/map

- Implements generic dictionary

```
class HashMap[K, V] extends AbstractMap[K,
    V]
// Initialization with expected load factor
new HashMap(initialCapacity: Int,
    loadFactor: Double)
```

```
// Operations
tbl.+=(k,v)  // to add (k,v) to tbl
tbl.get(k)   // return binding of k
tbl.remove(k) // remove binding of k
```

- Can be used for memoization
- If hash map is localized in the function, then when viewed from outside it appears as a pure function

```
// fib with memoization
def fib_memo(n: Int): Int = {
  val tbl = new HashMap[Int, Int]()
  def aux(n: Int): Int = {
    if (n <= 1) 1
    else {
      val r = tbl.get(n)
      r match {
        case None => {
          val ans = aux(n-1) + aux(n-2)
          tbl.+=((n, ans))
          ans
        }
        case Some(ans) => ans
      }
    }
  }
  aux(n)
}
```

SCALA OOP

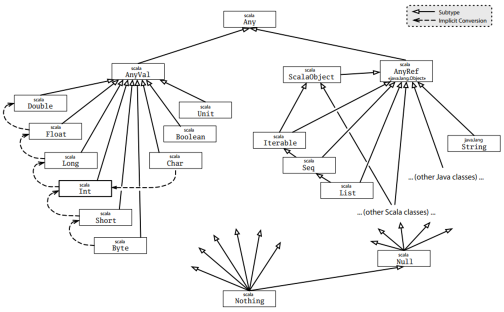
OOP Principles

- **Abstraction** allows implementation details to be hidden via private fields and methods
- **Encapsulation** binds data fields and methods together
- **Inheritance** is facilitated by sub-class mechanism with inherited fields and methods
- **Polymorphism** is supported by type variables and class hierarchy

Scala vs Java

- No static fields/methods. Workaround: use a singleton class with keyword `object` instead of `class`
- No primitive types

Class hierarchy



- Every user class
  - is indirectly a subclass of `scala.AnyRef`
  - implicitly extends trait `scala.ScalaObject`

Product types

Includes tuples and records; Similar to conjunction $a \wedge b$

```
class Pair[A,B] = (a, b)
```

Sum types

Includes ordinals and general algebraic data types; Similar to disjunction $a \vee b$

```
abstract class Either[A,B]
case class Left[A] extend Either[A,B]
case class Right[B] extend Either[A,B]
```

Type inference

- Types can either be declared or inferred
- Can infer type of
  - variable (through its initialization)
  - results of non-recursive method
  - type instantiation of polymorphic methods
- May fail

```
class MyPair[A, B](x: A, y: B);
object InferenceTest3 extends Application {
  def id[T](x: T) = x
  // type: MyPair[Int, String]
  val p = new MyPair(1, "scala")
  val q = id(1) // type: Int
}

// Explicit instantiation
val x: MyPair[Int, String] = new
    MyPair[Int, String](1, "scala")
val y: Int = id[Int](1)

// Failure
// Compilation error: Overloaded or
    recursive method fac needs return type
object Test {
  def fac(n: Int) = if (n == 0) 1 else n *
      fac(n - 1)
}
```

Runtime type representation

- Use `classOf[T]` to get string representation of a type
- Use `var.getClass()` to get the representation of runtime type for object

Abstract classes

- Provides a common definition of a base class that multiple derived classes can share
- Supports generic classes
- Can have deferred/abstract type, deferred value definition

```
abstract class Buffer[T] {
  val element: T
}
abstract class SeqBuffer[U,T<:Seq[U]]
    extends Buffer[T] {
  def length = element.length
}
```

Traits

- Can have fields, methods
- May have default impl for some methods
- Do not have constructor parameters
- Can be extended by other traits, abstract classes, concrete classes, and case classes
- More expressive than Java interfaces

```
// Similar to Eq type class in Haskell
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean =
      !isSimilar(x)
}
```

Mixins

- Classes can only have one superclass, but have many mixins
- Use mixins to derive from multiple classes

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}
// A mixin
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit)
    { while (hasNext) f(next) }
}
// A concrete class
class StringIterator(s: String) extends
    AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1;
      ch }
}
// A mixin class composition
object StringIteratorTest {
  def main(args: Array[String]) {
  class Iter extends
      StringIterator(args(0)) with
      RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

Polymorphism

Method polymorphism

- Same method name takes different forms
- **Static polymorphism**: Overloading
- **Dynamic polymorphism**: Overriding

Type polymorphism

- **Parametric polymorphism**: Function can take different types
- **Sub-class polymorphism**: A list with type `List[A]` contains elems that are sub-classes of `A`
- Note: Scala 2 `val` cannot be polymorphic, but Scala 3 `val` can

Abstract type definition

```
// Java vs Scala
? extends T = +T
? super T = -T
T = T
? = _
```

Variances

- `<:` is read "is a subtype of"
- Subclass $\implies$ subtype

$$\frac{T1 \text{ is subclass of } T2}{T1 <: T2}$$

- Covariance subtyping for read-only (OUT) structures, e.g. `List[Int]` <: `List[Object]`

$$\frac{T2 <: T4}{List[+T2] <: List[+T4]}$$

- Invariance subtyping for mutable (IN/OUT) structures, e.g. `Array[Int] <: Array[Int]`

$$\frac{T2 = T4}{Array[T2] = Array[T4]}$$

- Contravariance subtyping for write-only (IN) structures, e.g. `Printer[Object] <: Printer[Int]`

$$\frac{T2 <: T4}{Printer[-T4] <: Printer[-T2]}$$

- Function subtyping

$$\frac{I2 <: I1 \qquad O1 <: O2}{Func[-I1, +O1] <: Func[-I2, +O2]}$$

Packages

- Defines a set of member classes, objects, and packages
- Private: visible only for other members of the package
- Protected: accessible from all code inside the package

Imports

- Implicit imports: packages `java.lang`, `scala`, the object `scala.Predef`

```
// all members of p
// (analogous to import p.* in Java).
import p._
// the member x of p.
import p.x
// the member x of p renamed as a.
import p.{x => a}
// the members x and y of p.
import p.{x, y}
// the member z of p2, itself member of p1.
import p1.p2.z
```

SCALA FP

- Every function is an object
- Functions are first-class values. Can be
  - Passed as argument
  - Returned as result
  - Stored in data structures
- Anonymous functions allowed

```
object XXX extends App {
  def inc (x:int) : int = x+1
}
// Anonymous functions
(x:Int) => x+1
new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
```

Handling errors

Option type

```
enum Option[+T]:
  case None
  case Some(x:T)

def divide(x:Float,y:Float): Option[Float] {
  if (y==0) None
  else Some(x/y)
}
```

Either type

```
enum Either[+T,+S]:
  case Left(x:T)
  case Right(x:S)

def divide(x:Float,y:Float):
    Either[String,Float] {
  if (y==0) Left("cannot div by 0")
  else Right (x/y)
}
```

Exception handling

- Not referentially transparent

```
class DivideByZero extends RuntimeException

def divide(x:Float,y:Float):Float {
  if (y==0) throw new DivideByZero
  else (x/y)
}
```

Case classes

- Allow constructor parameters to be used in pattern-matching
- `new` keyword not required

```
// Untyped lambda calculus
abstract class Term
  case class Var(name: String) extends Term
  case class Fun(arg: String, body: Term)
      extends Term
  case class App(f: Term, v: Term) extends
      Term

val x = Var("x")
Console.println(x.name)
```

Pattern matching

- First-match policy
- `match` keyword allows pattern matching function to be applied to an object
- Can match against different types
- `_` indicates the default catch all

```
object MatchTest1 extends Application {
  def matchTest(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
```

```
    case _ => "many"
  }
  println(matchTest(3))
}
```

Lists

```
val fruit = List("apple", "orange", "pear")
val f2="apple" :: "orange" :: "pear" :: Nil

fruit.head // apple
fruit.tail // List(orange, pear)
fruit.isEmpty // false

val List(a,b,c) = fruit
a: String = apple
b: String = orange
c: String = pear

val a :: b :: rest = fruit
a: String = apple
b: String = orange
rest: List[String] = List(pear)
```

List operations

```
def length[T](xs:List[T]):Int =
  xs match {
    case List() => 0
    case x :: xs1 => 1+length(xs1)
  }
def append[T](xs: List[T], ys: List[T]):
    List[T] =
  xs match {
    case List()  => ys
    case x :: xs1 => xs :: append(xs1, ys)
  }
def sum(xs: List[Int]): Int = (0 /: xs) (_
    + _)
def prod(xs: List[Int]): Int = (1 /: xs) (_
    * _)
```

Higher-order functions

- Functions can be used as parameters, results, inside data structure

```
class Dec(left: String, right: String) {
  def layout[A](x: A) = left + x.toString()
      + right
}
object FunTest extends Application {
  def apply(f: Int => String, v: Int) = f(v)
  val decorator = new Dec("[", "]")
  // Function as argument
  println(apply(decorator.layout, 7))
}
```

Placeholder

- `_ + 1` equivalent to `(x: Int)=> x+1`
- `(_:Int)+ (_:Int)` equivalent to `(x:Int, y:Int)=>x+y`

Shorthand for types

Shorthand	Longer variant
`Int => Int`	`Function1[Int, Int]`
`(Int, Int) => String`	`Function2[Int, Int, String]`
`() => String`	`Function0[String]`

Operators

- Method with one param can be used in infix form
- Method with no params can be used in postfix form

```
class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x)
      that else this
  def or(that: MyBool): MyBool = if (x)
      this else that
  def negate: MyBool = new MyBool(!x)
}

// Infix vs traditional
def not(x: MyBool) = x negate; // Semicolon
    required
def not(x: MyBool) = x.negate; // Semicolon
    required here

// Postfix vs traditional
def xor(x: MyBool, y: MyBool) = (x or y)
    and not(x and y)
def xor(x: MyBool, y: MyBool) =
    x.or(y).and(x.and(y).negate)
```

Currying

```
def modN(n:Int)(x:Int) = ((x % n) == 0)
modN : Int => (Int => Boolean)
```

Laziness

- Default strategy is strict evaluation
- Exceptions
  - `e1 && e2` - if e1 is False, then e2 not evaluated
  - `if (e1) e2 else e3` - either e2 or e3 not evaluated

Non-strict parameters

```
// () optional
def if2[A] (cond:Boolean,f1:()=>A,f2:=>A):
    A = {
  if (cond) f1()
  else f2()
}
```

Expression

- To make an expression lazy and memoized, use `lazy` keyword

Implicits

- Implicit conversions are applied when an object of wrong type is used

```
// Type mismatch
scala> val i: Int = 3.5
scala> implicit def doubleToInt(x: Double)
    = x.toInt
// OK
scala> val i: Int = 3.5
val i: Int = 3
scala> val i: Int = 3.6
val i: Int = 3
```

- Function parameters may be implicit

```
val x = 10
implicit val y: Int = 3
// If this is uncommented there will be an
// error - Ambiguous given instances: both
// value y and value z match type Int of
// parameter m of method mult
// implicit val z: Int = 4

def mult(implicit m: Int) = x * m

// Implicit parameter will be passed here
val result = mult

// It will print 30 as a result
println(result)
```

- Implicit overloading allows instances of abstract classes
- Can support generic methods

```
object ImplicitTest extends Application {
  implicit object StringMonoid extends
      Monoid[String] {
    def add(x: String, y: String): String =
        x concat y
    def unit: String = ""
  }
  implicit object IntMonoid extends
      Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}

// Generic sum method
def sum[A](xs: List[A])(implicit m:
    Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))

// Infer type
println(sum(List(1, 2, 3)))
println(sum(List("a", "b", "c")))
// Or pass in
println(sum(List(1, 2, 3))(IntMonoid))
println(sum(List("a", "b",
    "c"))(StringMonoid))
```

MONADIC PARSING

Generic parser

```
enum Token = ...
type Tokens = [Token]
//              Initial    Remaining
type Parser[A] = Tokens => (Tokens, A)

// Support error handling
type ParserE[A] = Tokens => Either[String,
    (Tokens, A)]
```

Text.Parsec combinators

Combinator	Description
`p1 ~~ p2`	sequencing: must match p1, followed by p2
`p1 <|> p2`	alternation: match either p1 or p2, preference given to p1
`p1 <?> st`	match p1 or show st as error message
`try p1`	does not consume input if it fails
`choice [p1,p2,...]`	applies alternation sequentially
`many p1`	repeating zero or more times
`many1 p1`	repeating one or more times
`skipMany p1`	like many, but skips its result
`between open p1 close`	parses open, p1, then close
`parseMap f p1`	applies f to the parser's result
`p1 *> p2`	like sequencing, but ignore left result
`p1 <* p2`	like sequencing, but ignore right result

Arithmetic Expression Parser

Right-recursive rule

- Supports right associativity of operators

```
// In Scala
def expr: Parser[Any] = term ~rep("+" ~
    term | "-" ~term)
def term: Parser[Any] = factor ~rep("*" ~
    factor | "/" ~factor)
def factor: Parser[Any] = wholeNumber | "("
    ~> expr <~ ")"
```

Left-recursive rule

- Left recursion works well only if we have a non-deterministic parser, which terminates whenever there is a base case
- Solution: use repetition construct of Extended BNF form

```
<expr> ::= <term> | <expr> ("+"|"-") <term>
<term> ::= <fac> | <term> ("*"|"/") <fac>
<fac>  ::= <id>  | <constant> | "(" <expr>
    ")"

// Avoiding left recursion
<expr> ::= <term> { ("+"|"-") <term> }
<term> ::= <fac> { ("*"|"/") <fac> }
<fac>  ::= <constant> | "(" <expr> ")"
```

Expr AST Type

```
enum Expr:
  case Const(i:Int)
  case Plus(e1:Expr,e2:Expr)
  case Minus(e1:Expr,e2:Expr)
  case Mult(e1:Expr,e2:Expr)
  case Div(e1:Expr,e2:Expr)
end Expr
```