Recursion, Mathematical Datatypes and Functional Programming

R. Sekar

(Pure) Functional Programming

- Programs consist of a set of functions
- These functions behave like as functions Mathematics.
- Variables are like variables in algebra or logic
  - They are placeholders for (input) values
  - Unlike most programming languages (e.g., Java), there is no concept of "assigning" to them or "modifying" their values.
    - The term "absence of side-effects" is used to described this property
- Data structures are based on mathematical types
  - Sets
  - Cartesian products and other *type constructors* to consruct new types from existing ones.

ML

- ML was developed initially as a "meta language" for a theorem proving system (*Logic of Computable Functions*)
  - The two main dialects, SML and CAML, have many features in common:
    - data type definition, type inference, interactive top-level, . . .
- SML syntax is closer to mathematics, so we will use it in this course.
- CAML has more features for programming (e.g., object-orientation). It is a more modern project, and hence prefer it over SML in real projects.

Installing Standard ML of New Jersey (SML/NJ)

Ubuntu/Debian Linux: Run `sudo apt install smlnj` at the command-line

Windows: Your best bet is likely to be an install within a Windows subsystem for Linux (WSL): https://docs.microsoft.com/en-us/windows/wsl/install-win10

- Within the Linux subsystem, follow the above instructions.

On a browser: You can also run SML directly on your web browser at https://www.tutorialspoint.com/execute_smlnj_online.php. No installation is required, but you have to cut/paste everything from local files on your machine (or else your programs may not be saved)

Other: Please see https://www.smlnj.org/ and especially http://www.smlnj.org/install/

Using SML

- On Linux, run `sml` on the command-line. If you are using it in a browser, you will already be at the top-level SML prompt.
- SML prompts with

  `-`

- Enter new code definitions, evaluate expressions, or issue directives at the prompt.
- Control-D to exit SML
- We will use SML interactive toplevel throughout for examples.

Run `sml` inside readline wrapper as follows:

```
rlwrap sml
```

in order to be able to go up/down and correct your mistakes.

Using SML

- Instead of typing our program at the prompt, we usually enter it into a file and execute it.
  - Type the command `use "abc"` in order to load the file abc in the current directory.
  - Contents of the file are processed as if they were directly typed in
  - Not available on the browser (no access to files).
    - You can instead cut/paste from a file stored on your computer.

Expressions

- Examples:

User Input	SML's Response
`2 * 3;`	`val it = 6 : int`

`val`	:	Indicates that the result (aka output) is a value
`it`	:	variable representing the output of the last expression
`6`	:	the number 6
`:`	:	separator between the output and its type
`int`	:	the type of output

- A few key points:
  - Use semicolons at the end of definitions. Spurious semicolons in the middle confuse SML.
  - When you type a multi-line definition, you get a secondary prompt `=` from SML.
    - When you finish the definition and type a semicolon, SML goes back to its top-level prompt `-`
    - If SML is confused and prints `=` when you think you are done with a definition, press Ctrl-C to get back to the primary prompt.

Expression Evaluation (Contd.)

More examples:

User Input	SML's Response
`2 + 3 * 4;`	`val it = 14 : int`
`~2 + 3 * 4;`	`val it = 10 : int`
`(2 + 3) * 4;`	`val it = 20 : int`
`4.4 * 2.0;`	`val it = 9.68 : real`

`2 + 2.2;`	`stdIn:15.1-15.8 Error: operator and operand don't agree [overload conflict]`
	`operator domain: [+ ty] * [+ ty]`
	`operand: [+ ty] * real`
	`in expression:`
	`2 + 2.2`

Operators

Operators	Types

`+`
`-`
`*`
	Integer or real arithmetic

`true` `false`	Boolean constants
`div`	Integer division
`/`	Real number division

`andalso`
`orelse`
`not`
	Boolean operations

Value definitions

- Syntax: `val` ⟨*name*⟩ `=` ⟨*expression*⟩ `;`
- Examples:

User Input	**SML's Response**
`val x = 1;`	`val x = 1:  int`
`val y = x + 1;`	`val y = 2 :  int`
`val x = x + 1;`	`val x = 3 :  int`
`val z = "SML rocks!";`	`val z = "SML rocks!":  string`

Functions

- Syntax: `fun` ⟨*name*⟩ (⟨*arguments*⟩) `=` ⟨*expression*⟩ `;`
- Examples:

User Input	**SML's Response**
`fun f(x) = 1;`	`val f = fn : 'a -> int`
`fun g(x) = x;`	`val g = fn : 'a -> 'a`
`fun inc(x) = x + 1;`	`val inc = fn : int -> int`
`fun sum(x,y) = x+y;`	`val sum = fn : int * int -> int`

`fun max(x, y) =`
`  if x < y`
`    then y`
`    else x;`
	`val max = fn : int * int -> int`

`max(2,3);`	`val it = 3 : int`

Let Statements

- We often want to break down a complex expression into a series of steps to make it easier to understand

  let ⟨*definition*⟩ in ⟨*expression*⟩ ;

- A let can define one or more values (left), functions (right), or a mix of them.

```
- fun f(x, y) = let
                  val x2 = x*x
                  val y2 = y*y
                in
                  x2+y2
                end;
```

```
- fun f(x, y) = let
                  fun sq(z) = z*z
                in
                  sq(x)+sq(y)
                end;
```

Strings and Printing in SML

- The built-in function `print` prints strings (and only strings)

```
-print;
val it = fn : string -> unit
```

- Use predefined functions `Int.toString` and `Real.toString` to covert `int`'s and `real`'s to strings.
- Use string concatenation operator ^ to construct complex strings.

```
- print("I know 10 * 20 is " ^ Int.toString(10*20) ^ "\n");
I know 10 * 20 is 200
val it = () : unit
```

Recursion

```
fun g(0) = 1                 (* Base Case 1 *)
|   g(1) = 1                 (* Base Case 2 *)
|   g(n) = g(n-1)+g(n-2);    (* Recursive Case *)

fun f(0) = 1                 (* Base Case *)
|   f(n) = 2*f(n-1);         (* Recursive Case *)

fun h(1) = 1                 (* Base Case *)
|   h(n) = 2*h(n div 2);     (* Recursive Case *)
```

More recursion

```
fun f(0) = 1
|   f(n) = n * f(n-1);
```

More recursion

```
fun f(0) = 1
|   f(n) = n * f(n-1);

fun euclid(0, b) = b
|   euclid(a, b) = euclid(b mod a, a);
```

More recursion

```
fun f(0) = 1
|   f(n) = n * f(n-1);

fun euclid(0, b) = b
|   euclid(a, b) = euclid(b mod a, a);
fun gcd(a,b) = if a < b then euclid(a, b) else euclid(b,a);
```

(Mathematical) Data Types in SML

- *Pre-defined Sets*
  - Built-in Sets: $\mathtt{int} \subset \mathbb{N}$, $\mathtt{real} \subset \mathbb{R}$, `bool`, `string`

(Mathematical) Data Types in SML

- *Pre-defined Sets*
  - Built-in Sets: `int` $\subset \mathbb{N}$, `real` $\subset \mathbb{R}$, `bool`, `string`
  - Programmer-defined sets ("enumerated types")

```
datatype Suit = Spades|Clubs|Diamonds|Hearts;
(* the type corresponding to the set {Spades, Clubs, Diamonds, Hearts} *)

datatype Rank = One|Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King|Ace;
```

(Mathematical) Data Types in SML

- *Pre-defined Sets*
  - Built-in Sets: `int` $\subset \mathbb{N}$, `real` $\subset \mathbb{R}$, `bool`, `string`
  - Programmer-defined sets ("enumerated types")

```
datatype Suit = Spades|Clubs|Diamonds|Hearts;
(* the type corresponding to the set {Spades, Clubs, Diamonds, Hearts} *)
```

```
datatype Rank = One|Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King|Ace;
```

- *Tuples:* Cartesian Products of Sets

```
type Card = Suit * Rank; (* Type corresponding to the set Suit × Rank *)
```

```
type PokerHand = Card * Card * Card * Card * Card;
(* A hand with five cards.  Note that this is not a set:  the order matters.*)
```

(Mathematical) Data Types in SML

- *Pre-defined Sets*
  - Built-in Sets: `int` $\subset \mathbb{N}$, `real` $\subset \mathbb{R}$, `bool`, `string`
  - Programmer-defined sets ("enumerated types")

```
datatype Suit = Spades|Clubs|Diamonds|Hearts;
(* the type corresponding to the set {Spades, Clubs, Diamonds, Hearts} *)

datatype Rank = One|Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King|Ace;
```

- *Tuples:* Cartesian Products of Sets

```
type Card = Suit * Rank; (* Type corresponding to the set Suit × Rank *)

type PokerHand = Card * Card * Card * Card * Card;
(* A hand with five cards.  Note that this is not a set:  the order matters.*)
```

- *Sequences:* A `list` in SML represents the type $S^* = \bigcup_{i=0}^{\infty} S^i$ for any set S.

```
type Hand = Card list;
```

List Examples in SML

User Input	SML's Response
`[1];`	`val it = [1] : int list`
`[4.1,2.7,3.1];`	`val it = [4.1, 2.7, 3.1] : real list`

`[4.1, 2];`	`Error: operator and operand don't agree [overload conflict]`
	`operator domain: real * real list`
	`operand: real * [int ty] list`
	`in expression:`
	`4.1 :: 2 :: nil`

`[[1,2],[4,8,16]];`	`val it = [[1,2], [4,8,16]] : int list list`
`1::2::[]`	`val it = [1, 2] : int list`

Tuple Examples in SML

```
(2,"Andrew") : int * string
(true,3.5,"x") : bool * real * string
((4,2),(7,3)) : (int * int) * (int * int)
[(2,3),(2,2),(9,1)] : (int * int) list
```

- Lists are homogeneous: All elements must have the same type.
- But tuples are heteregeneous: components can be of different types.

Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
  - For integers, the base case corresponds to $n = 0$
  - For lists, the base case typically corresponds to the empty list

Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
  - For integers, the base case corresponds to $n = 0$
  - For lists, the base case typically corresponds to the empty list
- We use *pattern-matching* to distinguish between cases
  - On the surface, this looks very similar to what we have seen before

Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
    - For integers, the base case corresponds to $n = 0$
    - For lists, the base case typically corresponds to the empty list
- We use *pattern-matching* to distinguish between cases
    - On the surface, this looks very similar to what we have seen before
- Note that `nil` and `[]` both denote empty lists

Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
  - For integers, the base case corresponds to $n = 0$
  - For lists, the base case typically corresponds to the empty list
- We use *pattern-matching* to distinguish between cases
  - On the surface, this looks very similar to what we have seen before
- Note that `nil` and `[]` both denote empty lists
- Non-empty lists are of the form `h::t` where `h` denotes the first element of the list ("head") and `t` denotes the rest of the list ("tail")
  - For a list of type `'a list`, the head has the type `'a` while the tail has the type `'a list`

Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
  - For integers, the base case corresponds to $n = 0$
  - For lists, the base case typically corresponds to the empty list
- We use *pattern-matching* to distinguish between cases
  - On the surface, this looks very similar to what we have seen before
- Note that `nil` and `[]` both denote empty lists
- Non-empty lists are of the form `h::t` where `h` denotes the first element of the list ("head") and `t` denotes the rest of the list ("tail")
  - For a list of type `'a list`, the head has the type `'a` while the tail has the type `'a list`
- List constants are typically specified without using `::`, e.g., `[1, 2, 3, 4]`
  - But you can use `::` if you wanted: `1::2::3::4::nil`
  - You can also mix and match the `::` and `[]` notations, e.g., `1::2::[3,4]` or `1::2::[]`

Example List Functions

```
fun length(nil) = 0
| length(x::xs) = 1 + length(xs);
```

Example List Functions

```
fun length(nil) = 0
| length(x::xs) = 1 + length(xs);
```

```
fun append(nil, y) = y
| append(x::xs, y) = x::append(xs, y);
```

Example List Functions

```
fun length(nil) = 0
| length(x::xs) = 1 + length(xs);

fun append(nil, y) = y
| append(x::xs, y) = x::append(xs, y);

fun reverse(nil) = nil
| reverse(x::xs) = append(reverse(xs), [x]);
```

Example List Functions

```
fun length(nil) = 0
| length(x::xs) = 1 + length(xs);
```

```
fun append(nil, y) = y
| append(x::xs, y) = x::append(xs, y);
```

```
fun reverse(nil) = nil
| reverse(x::xs) = append(reverse(xs), [x]);
```

There is an infix operator @ for append: e.g., `[1,2] @ [3,4]` yields `[1,2,3,4]`.

Sieve of Eratosthenes

- Start with a list of natural numbers > 1

Sieve of Eratosthenes

- Start with a list of natural numbers > 1
- Take the first number that has not been crossed out, and output it
  - It is a prime number

Sieve of Eratosthenes

- Start with a list of natural numbers > 1
- Take the first number that has not been crossed out, and output it
  - It is a prime number
- Cross out all multiples of this number

Sieve of Eratosthenes

- Start with a list of natural numbers > 1
- Take the first number that has not been crossed out, and output it
  - It is a prime number
- Cross out all multiples of this number
- Repeat until the list is exhausted.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80

Sieve of Eratosthenes: Approach

1. Define a function `gen` to generate a list of integers
2. Define a helper function `elim` that takes a list and a number as arguments, and deletes all multiples of the number from the list
3. Define the `sieve` function that
   - takes the current list as input,
   - moves the first element to a list of prime numbers
   - uses `elim` to eliminate multiples of this number from the list,
   - and finally, repeats this until done
4. Define a top-level function that generates a list and then invokes `sieve` on it.

SML Code for Sieve of Eratosthenes

```
fun gen(0, m) = [] (* 1st arg is number of elems, m is the next elem *)
| gen(n, m) = m::gen(n-1, m+1);

fun elim([], n) = []
| elim(x::xs, n) = if (x mod n = 0) then elim(xs, n) else x::elim(xs, n);

fun sieve([], primes) = primes
| sieve(n::ns, primes) = sieve(elim(ns, n), n::primes);

fun dosieve(n) = sieve(gen(n-1, 2), []);
```

Sets

- Lists are *not* sets: they represent sequences.
  - But we can represent sets using lists: keep list elements in a sorted order, and eliminate duplicates.
- If we start with lists that satisfy these properties, then we can implement set operations easily using list operations

Set Operations in SML

- Start with the definition of Cartesian product

$$X \times Y = \{(x, y) | x \in X \wedge y \in Y\}$$

- To express it in SML, break into two steps:
  - First iterate through the set Y:

```
fun cart1(x, []) = []
| cart1(x, y::ys) = (x, y)::cart1(x, ys);
```

  - then iterate through the set X

```
fun cart([], ys) = []
| cart(x::xs, ys) = cart1(x, ys) @ cart(xs, ys);
```

Set Operations in SML

$$X \cup Y = \{a | a \in X \vee a \in B\}$$

In the SML code below, we iterate simultaneously through X and Y. We have some extra work over the above definition so that we maintain elements in sorted order.

```
fun union([], ys) = ys
| union(xs, []) = xs
| union(x::xs, y::ys) = if (x = y)
                           then x::union(xs, ys)
                        else if (x < y)
                           then x::union(xs, y::ys)
                        else y::union(x::xs, ys);
```

Set Operations in SML: Exercises

- Membership
- Intersection

$$X \cap Y = \{a | a \in X \wedge a \in Y\}$$

- Subset

$$X \subseteq Y \text{ iff } X \cup Y = Y$$

- Difference

$$X - Y = \{a | a \in X \wedge a \notin Y\}$$

- Complement
  - Test your implementation by checking De Morgran's laws

Relations

- Recall that a binary relation $R : X \longrightarrow Y$ is a subset of $X \times Y$. For instance, the following list represents a binary relation from $\mathbb{N}$ to $\mathbb{N}$:

$$[(1,3),(2,4),(1,4),(3,5)]$$

```
fun getSupport([]) = []
| getSupport((x,y)::xys) = union([x], getSupport(xys));

fun getRange([]) = []
| getRange((x,y)::xys) = union([y], getRange(xys));

fun isTotal(Domain, R) = (Domain = getSupport(R));

fun isOnto(Codomain, R) = (Codomain = getRange(R));
```

Relations

```
fun getArrowTails([]) = []
| getArrowTails((x,y)::xys) = x::getArrowTails(xys);

fun getArrowHeads([]) = []
| getArrowHeads((x,y)::xys) = y::getArrowHeads(xys);

fun isFun(R) = (getSupport(R) = getArrowTails(R));

fun isOneToOne(R) = (getRange(R) = getArrowHeads(R));

fun isBijection(X, Y, R) = isFun(R) andalso isTotal(X, R)
andalso isOneToOne(R) andalso isOnto(Y, R);
```

Relations: Exercises

- Composition of relations $R : X \longrightarrow Y$ and $S : Y \longrightarrow Z$

$$R \circ S = \{(x, z)|\ \exists y\ (x, y) \in R \wedge (y, z) \in S\}$$

- Compute R^n, where $R : X \longrightarrow X$ is a binary relation on X.
- Check for reachability in a graph
- Check if a given walk (or path or circuit) is valid
- Compute paths in a graph

End of Slides Covered in Class

The remaining slides are included for those that are interested in learning more about SML. Students are not expected to review them for the Final exam.

Generating Combinations (Subsets of Given Size)

- Generate subsets of size n that include the first element
  - Combine first element with any $n-1$-member subset of the remaining elements
- Generate subsets of size n that *don't* include the first element
  - In this case, we are generating n-member subsets of the remaining elements

```
fun prefixAll([], x) = [] (* put x at the beginning of all lists in the first arg *)
|   prefixAll(xs::xss, x) = (x::xs)::prefixAll(xss, x);

fun subsets(xs, 0) = [[]] (* All sets have ONE subset of size 0 *)
|   subsets([], n) = []   (* An empty set has NO subsets of size > 0 *)
|   subsets(x::xs, n) = prefixAll(subsets(xs, n-1), x) @ subsets(xs, n);

fun genlist(0, s) = []
|   genlist(r, s) = s::genlist(r-1, s+1);

fun choose(m, n) = List.length(subsets(genlist(m, 1), n));
```

Generating Permutations

- Generate all permutations of all but the first element
- Insert the first element at every possible position in each of these permutations.

```
fun insertAt(0, xs, y) = y::xs (* Insert y into xs at position given by 1st argument *)
|   insertAt(n, x::xs, y) = x::insertAt(n-1, xs, y);

fun insAtAll(xs, y) = (* Returns |xs|+1 lists: ith list is xs with y inserted at ith pos *)
    let fun doInsAtAll(0, xs, y) = [insertAt(0, xs, y)]
        |   doInsAtAll(n, xs, y) = insertAt(n, xs, y)::doInsAtAll(n-1, xs, y)
    in doInsAtAll(List.length(xs), xs, y) end;

fun insAtAllIntoAll([], x) = [] (* insert x into all possible pos in all lists in the 1st arg *)
|   insAtAllIntoAll(zs::zss, x) = insAtAll(zs, x) @ insAtAllIntoAll(zss,x);

fun permute([]) = [[]] (* Not [] but [[]]: there is one permutation of empty list *)
|   permute(x::xs) = insAtAllIntoAll(permute(xs), x);
```

Counting: n of m Books, Omit k after a selected book

- An analytical solution to this problem required a new way of thinking, but its code is not far from the $\binom{m}{n}$ problem
  - Just discard k elements in one of the two recursive cases
- The program is simple because it is brute-force:
  - Recursive calls implicitly construct the tree diagram, and then we simply count the leaves
- Somtimes, it is easier to apply conditions in a post-processing phase.

```
fun dropN(0, xs) = xs
|   dropN(n, []) = []
|   dropN(n, x::xs) = dropN(n-1, xs)

fun books(xs, 0, k) = [[]] (* All sets have ONE subset of size 0 *)
|   books([], n, k) = []   (* An empty set has NO books of size > 0 *)
|   books(x::xs, n, k) =
       prefixAll(books(dropN(k, xs), n-1, k), x) @ books(xs, n, k);

fun countBooks(m, n, k) = List.length(books(genlist(m, 1), n, k));
```

Permutations, Combinations and Counting: Exercises

- Generate the Powerset of a given set
- Generate all possible words from a given string, also count their number.
- How many numbers between 1 to 1M contain the digit 3?
- In how many ways can we pick n fruits subject to the following constraints?
  - The number of apples must be even.
  - The number of bananas must be a multiple of 5.
  - There can be at most four oranges.
  - There can be at most one pear.
- How many 5-card hands have a jack, queen, or king, but not all of them?

Recursive Data Types

- *Constructors* are operators that construct new data from existing data. They:
  - combine multiple data elements into one
  - attach a “tag” to the combination
    - Tags are used to distinguish base and inductive cases

```
datatype iTree = Leaf of int (* Base case *)
               | BNode of int * iTree * iTree (* Inductive case 1 *)
               | SNode of int * iTree         (* Inductive case 2 *)
```

Tree	Denoted by
1	`BNode(1,`
↙ ↘	`BNode(2,`
2 3	`Leaf(4),`
↙ ↘ ↘	`Leaf(5))`
4 5 6	`SNode(3, Leaf(6)))`

Lists are Recursive Types!

- They could have been defined as:

```
datatype 'a list = nil                (* Base case *)
                 | ::  of 'a * 'a list (* Recursive case *)
```

- Starting from an arbitrary type *A*, we are constructing a recursive type for representing a sequence of (0 or more) *A*'s
- In SML, names prefixed with single quotes are used to refer to *type variables* that can stand for an arbitrary type such as *A*
- Because SML predefines this particular recursive type, we are able to use infix notation for `::` rather than the usual prefix notation for type constructors.

```
- nil;
val it = [] : 'a list
- 1::2::nil;
val it = [1,2] : int list
```

More on Binary Trees

- Complete or Full Trees: every internal node has two children
- Perfectly balanced: all root-to-leaf paths have the same length
  - What is the number of nodes in a perfectly balanced binary tree of height h?
  - What fraction of nodes are leaved?
- Balanced: At every node, the height of left and right children differ by at most one.

Functions on Binary Trees: height, leaves, nodes

Binary Trees and Searching

- All nodes in the left subtree are less than the value at the node
- All nodes in the right subtree are greater than the value at the node
- Search is efficient: takes $O(\log n)$ time, where n is the number of nodes in the tree.

Tree Traversals

- Prefix traversal: visit node before either children
- Postfix traversal: visit both children before the node
- Infix traversal: visit left child, then node, then right child

Propositions

```
datatype PropFormula = T
                     | F
                     | VAR of int
                     | NOT of PropFormula
                     | AND of PropFormula*PropFormula
                     | OR of PropFormula*PropFormula
                     | IMPL of PropFormula*PropFormula;
```

Example: $x_1 \wedge x_2 \rightarrow \neg x_3$ becomes `IMPL(AND(VAR(1),VAR(2)), NOT(VAR(3)))`

```
fun find([], y) = false
|   find(x::xs, y) = (x=y) orelse find(xs, y);
fun eval(T, asg) = true
|   eval(F, asg) = false
|   eval(VAR(x), asg) = find(asg, x)
|   eval(NOT(f), asg) = not(eval(f, asg))
|   eval(AND(f1, f2), asg) = eval(f1, asg) andalso eval(f2, asg)
|   eval(OR(f1, f2), asg) = eval(f1, asg) orelse eval(f2, asg)
|   eval(IMPL(f1, f2), asg) = not(eval(f1, asg)) orelse eval(f2, asg);
```

Predicates

```
datatype PredFormula = PRED of int * int list
                     | NOT of PredFormula
                     | AND of PredFormula*PredFormula
                     | OR of PredFormula*PredFormula
                     | IMPL of PredFormula*PredFormula
                     | EXISTS of int*PredFormula
                     | FORALL of int*PredFormula
```

Example:

$$\forall x_1\ (p_1(x_1) \to \exists x_2\ p_1(x_2) \land p_2(x_1, x_2))$$

becomes

```
FORALL(1,
       IMPL(PRED(1, [1]),
               EXISTS(2,
                      AND(PRED(1, [2]),
                          PRED(2, [1,2]))
               )
           )
      )
```