CPEG 852 — Advanced Topics in Computing Systems
The Dataflow Model of Computation
Dynamic Dataflow

Stéphane Zuckerman

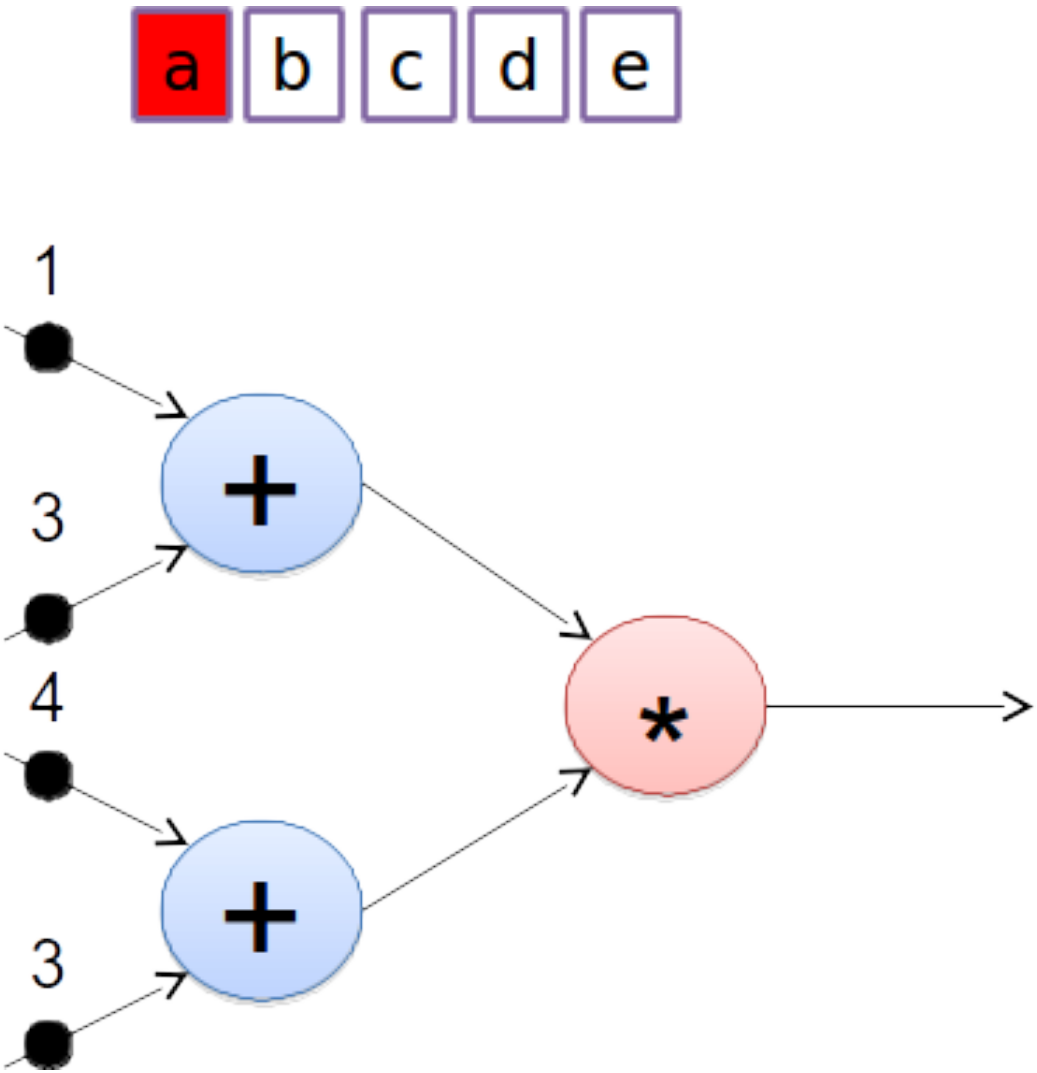
Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark,DE 19716, United States
szuckerm@udel.edu

September 8, 2015

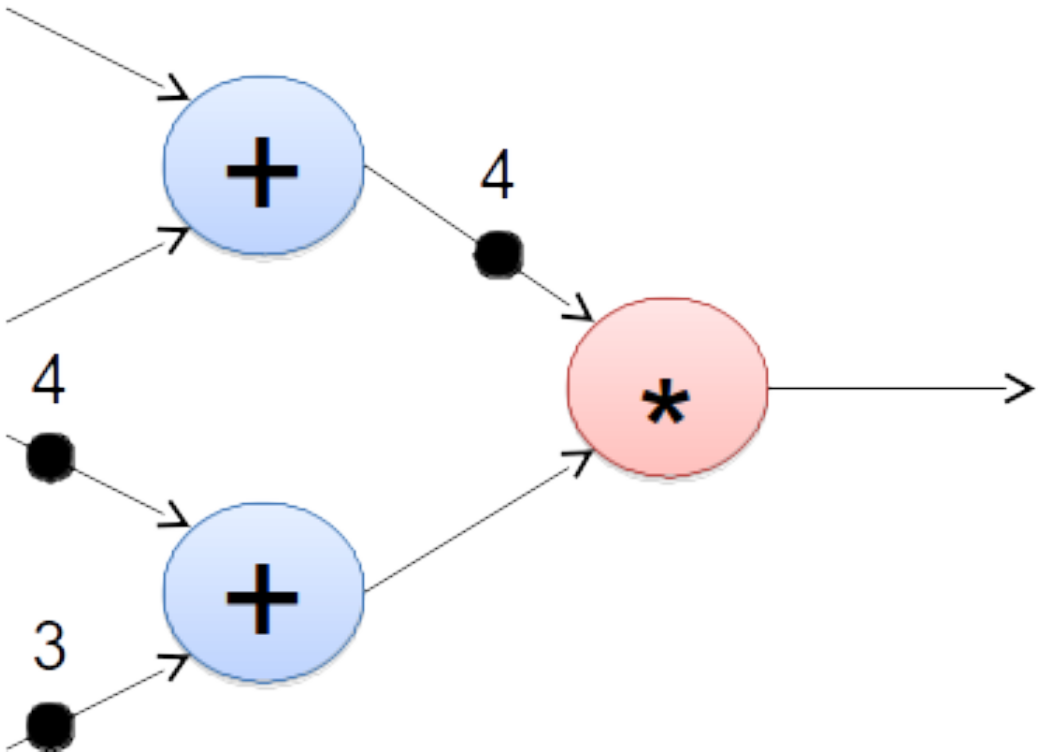
Outline

Outline

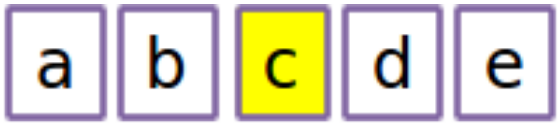
Reminder: Dataflow Model of Computation

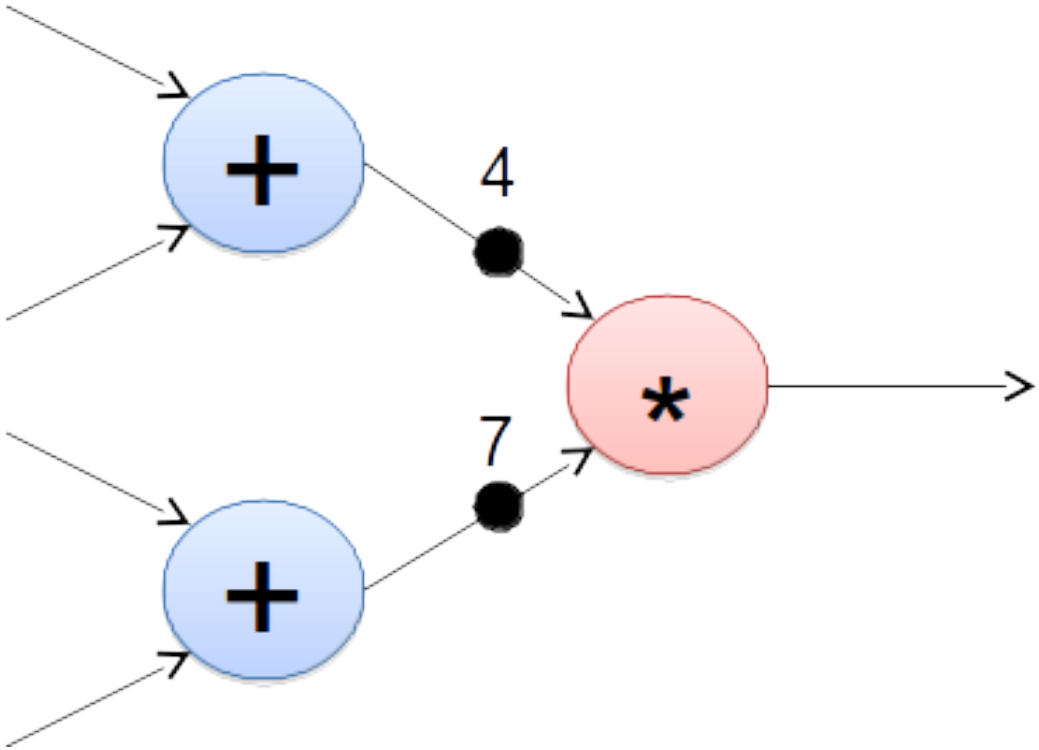


Reminder: Dataflow Model of Computation

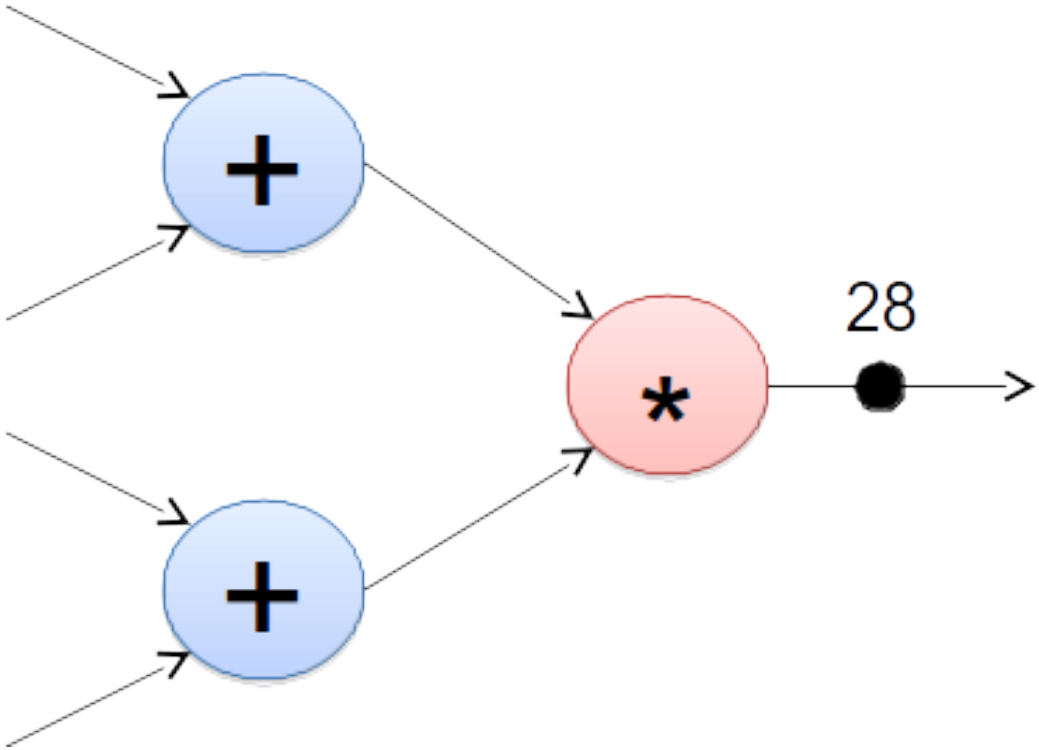


Reminder: Dataflow Model of Computation

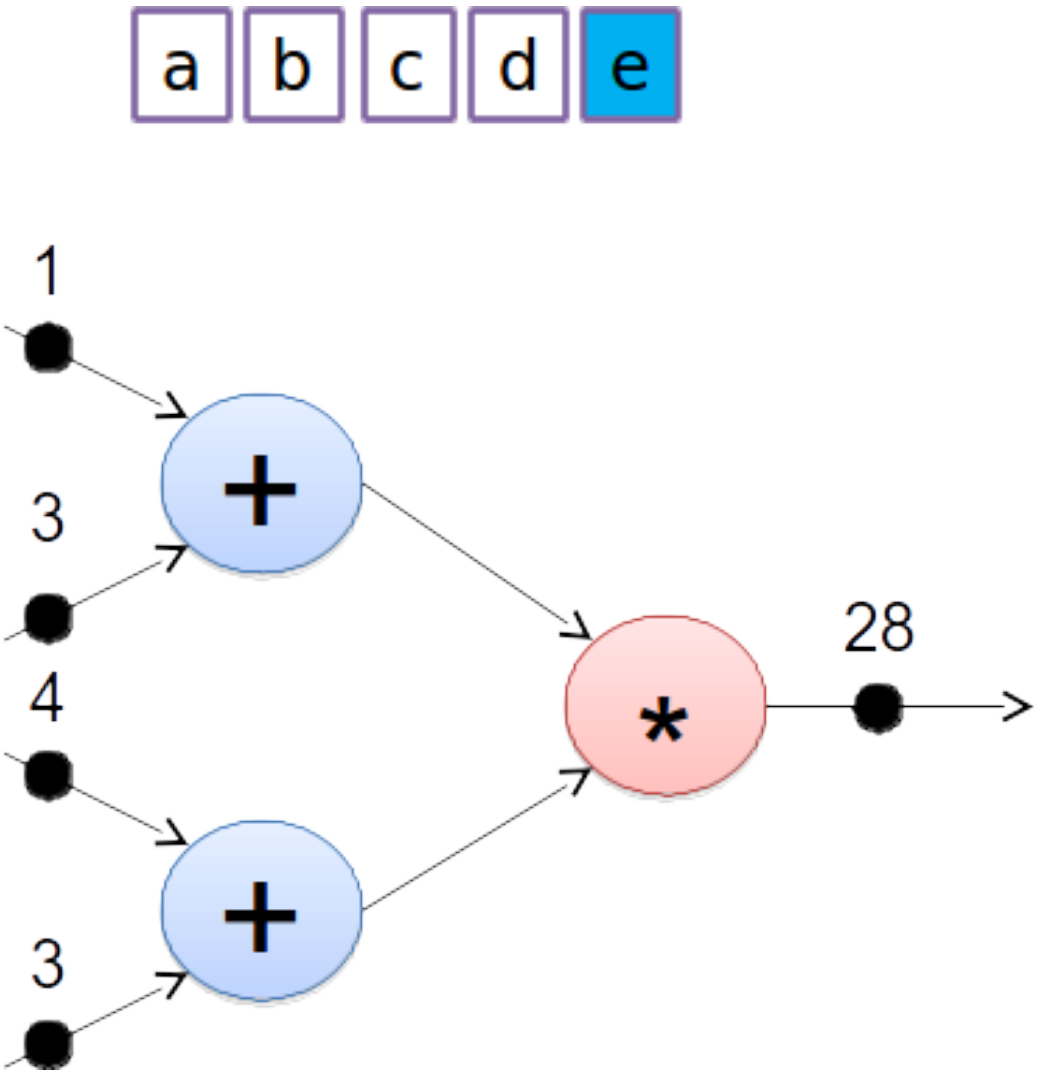




Reminder: Dataflow Model of Computation



Reminder: Dataflow Model of Computation



Conditional Expressions

```
if (p(y))
{
    f(x,y);
}
else
{
    g(y);
}
```

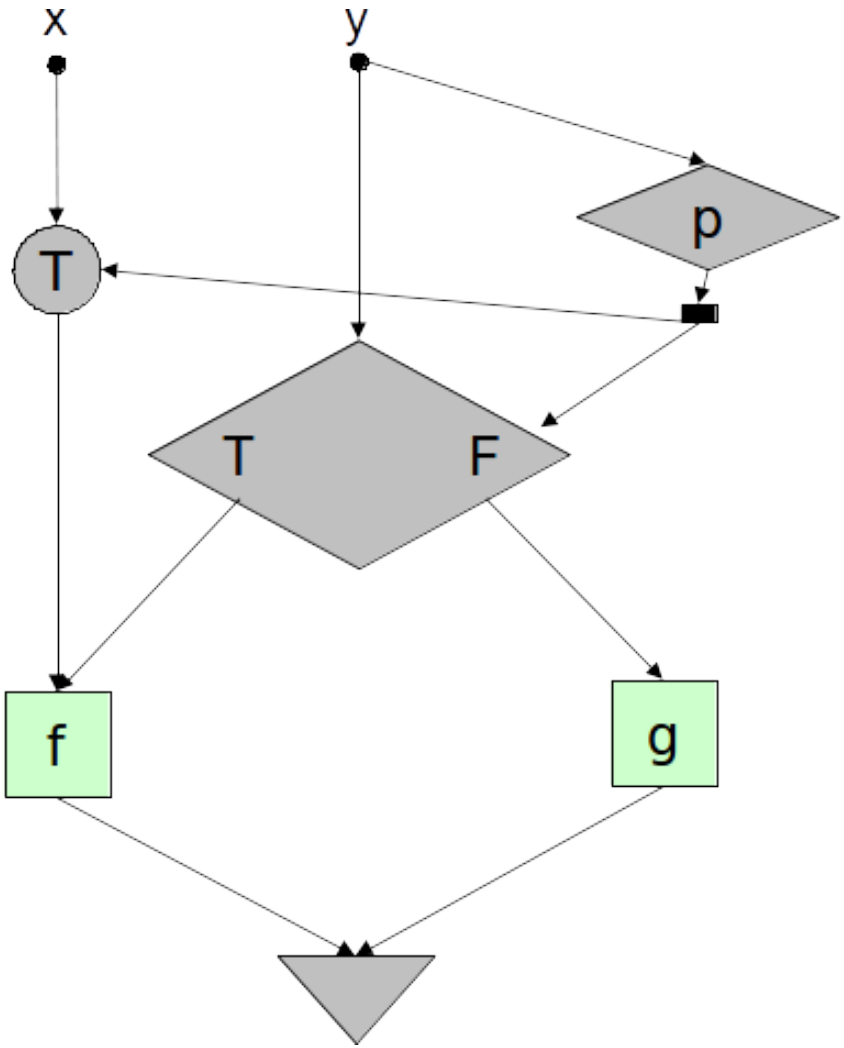


Figure : `if/else` construct

Conditional Expressions

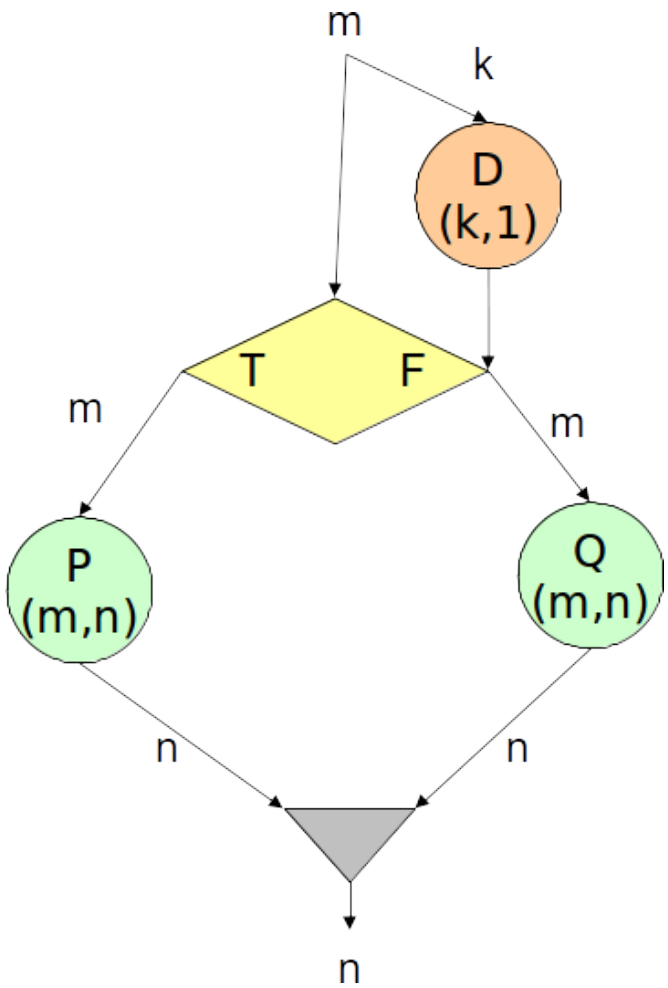


Figure : `if/else` schema

Conditional Expressions

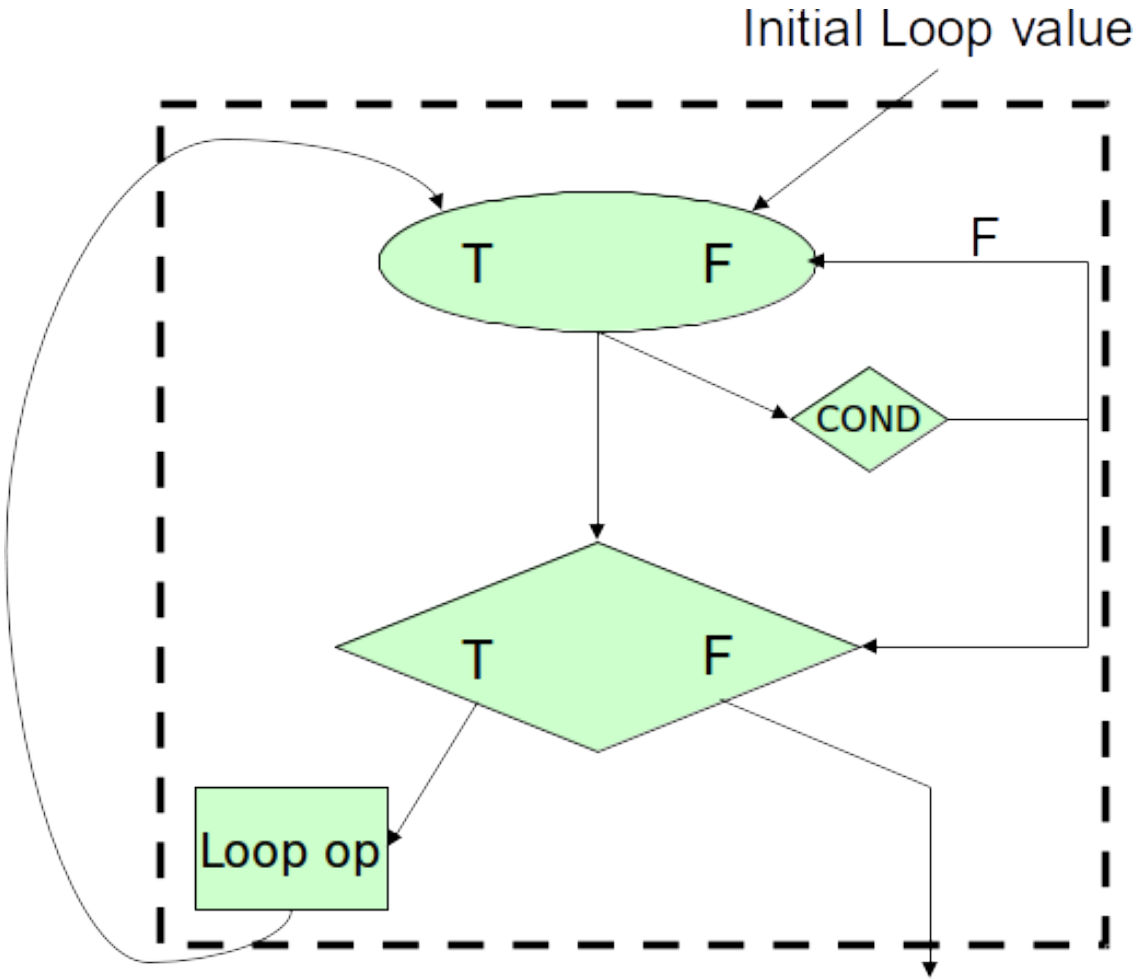


Figure : `loop` schema

UNIVERSITY OF DELAWARE

Static Dataflow's Golden Rule

. . . for any actor to be enabled, there must be no tokens on any of its output arcs. . .

Example

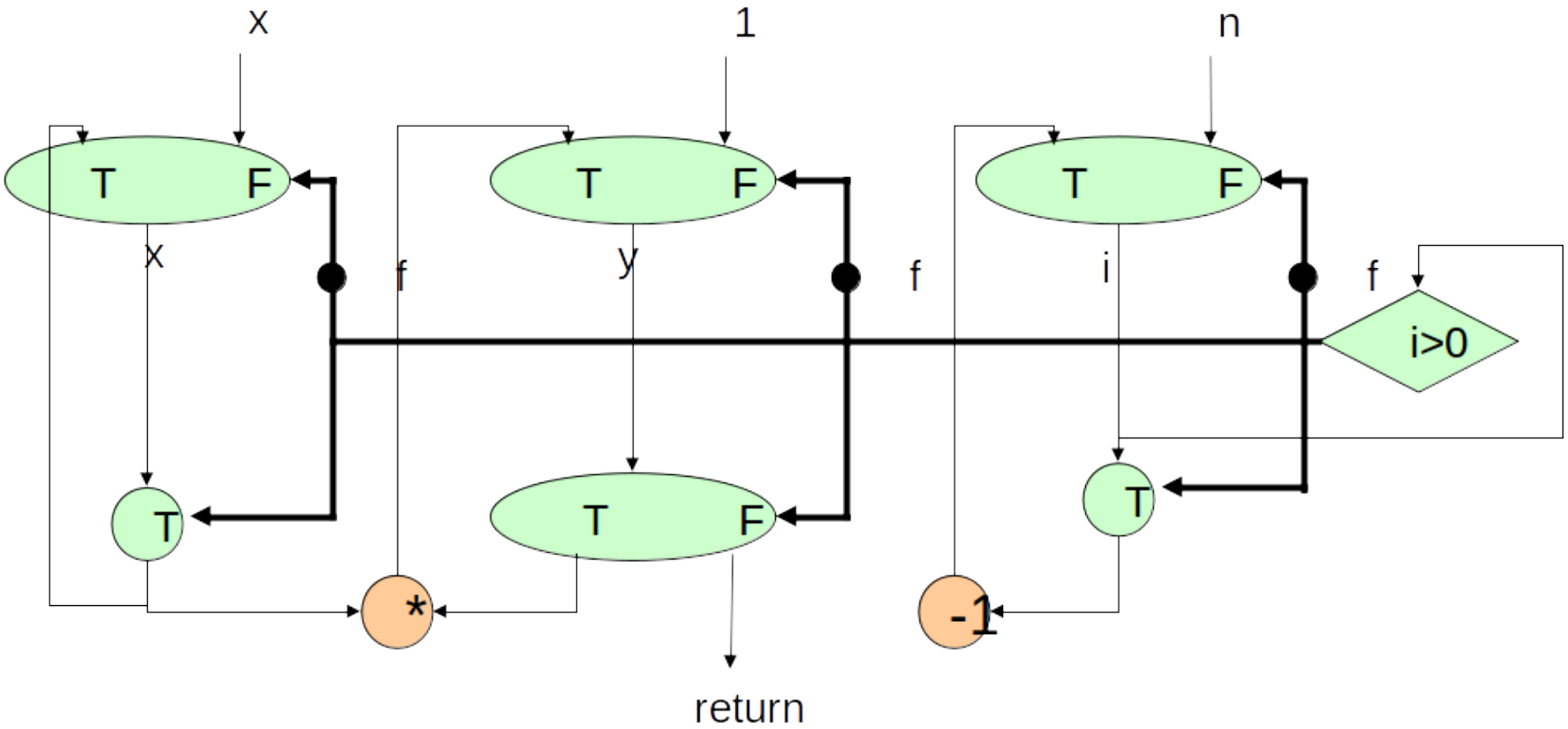
Power Function

```
long power(int x, int n)
{
    int y = 1;
    for (int i = n; i > 0; --i)
        y *= x;
    return y;
}
```

Figure : Computation: $y = x^n$

Example

Power Function – Computing $y = 2^n$

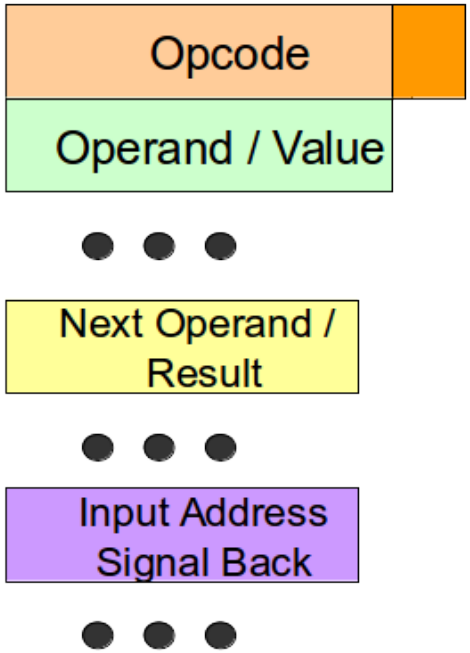


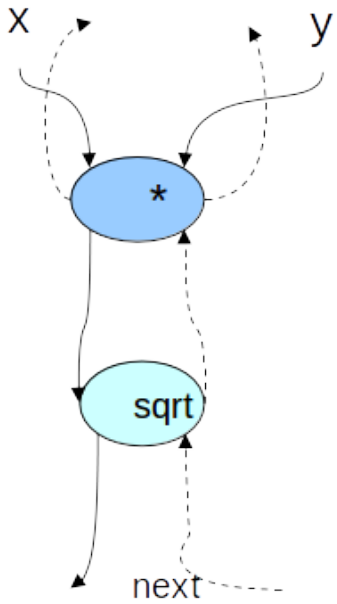
Static Dataflow

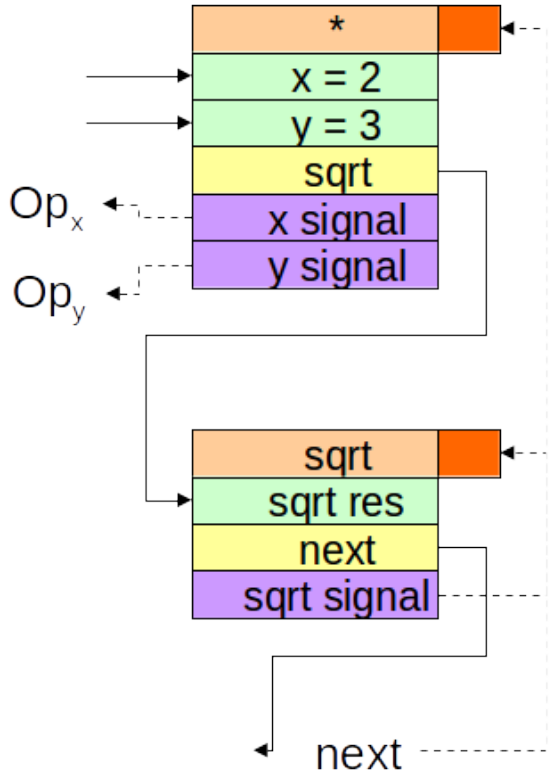
Features

- One-token-per-arc
- Deterministic merge
- Conditional/iteration construction
- Consecutive iterations of a loop can only be pipelined
- A dataflow graph $\Rightarrow$ activity templates
  - Opcode of the represented instruction
  - Operand slots for holding operand values
  - Destination address fields
- Token $\Rightarrow$ value + destination

Static Dataflow
Activity Templates







→	Token Arc
⇢	Communication Arc
Op_x / Op_y	The operation that produced x and y
next	The operation that will use the sqrt result

Recursive Program Graphs

- Extension over static dataflow concepts
- Graph must be acyclic (DAG, directed acyclic graph)
- One-token-per-arc-per-invocation
- Iteration is expressed in terms of tail recursion

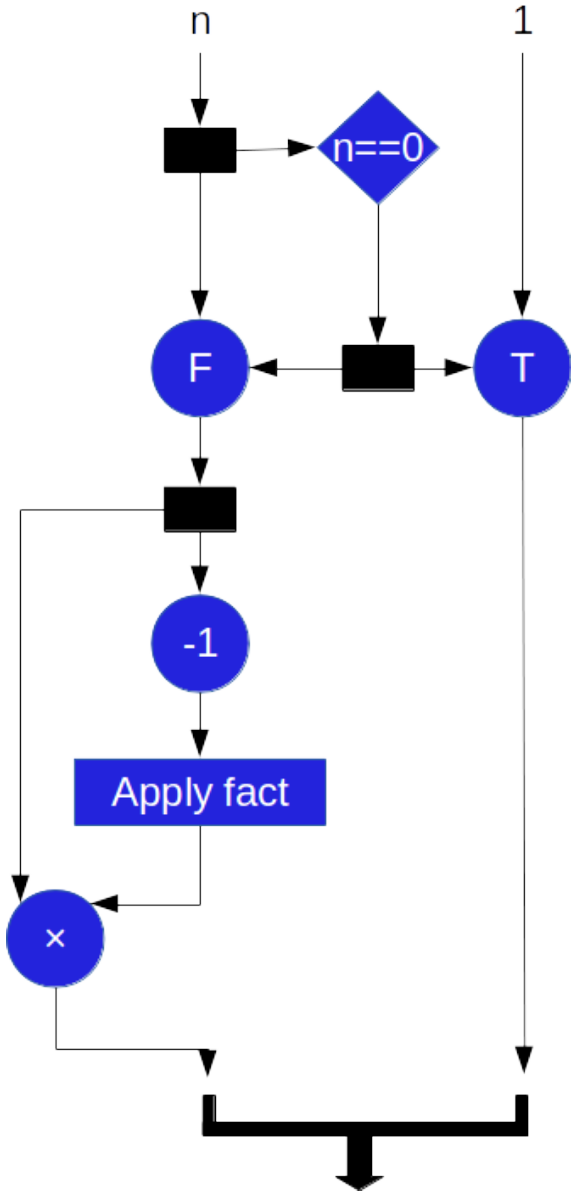
Examples of Ordinary Recursion

Factorial

```
long factorial(long n)
{
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

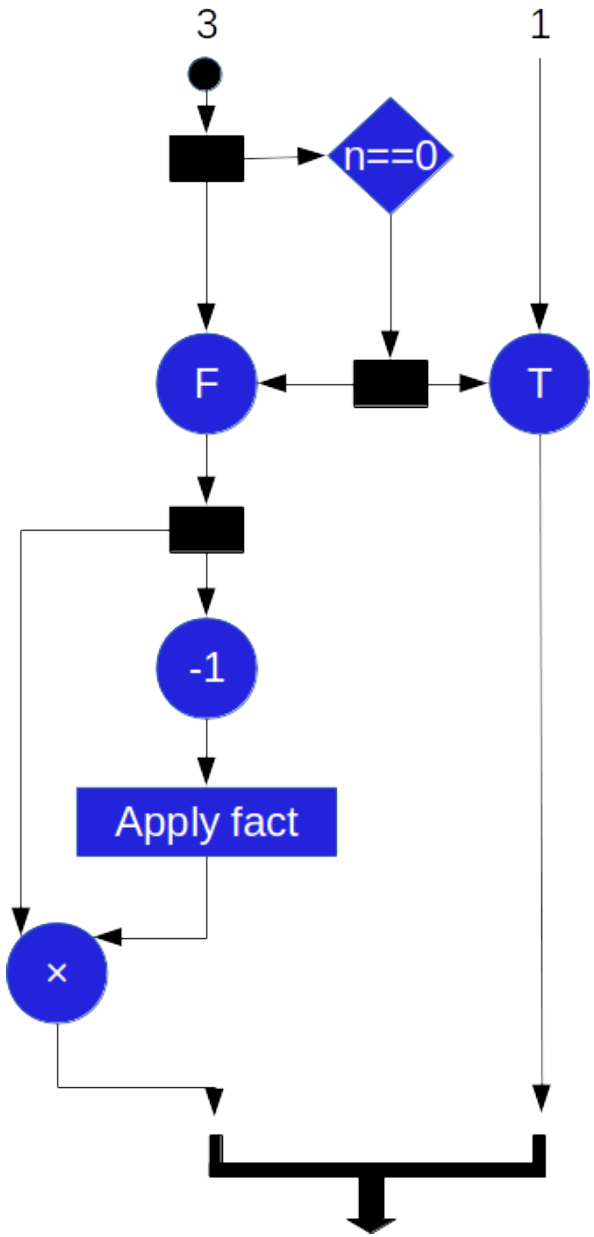
Example – Factorial

DFG for fact(3)



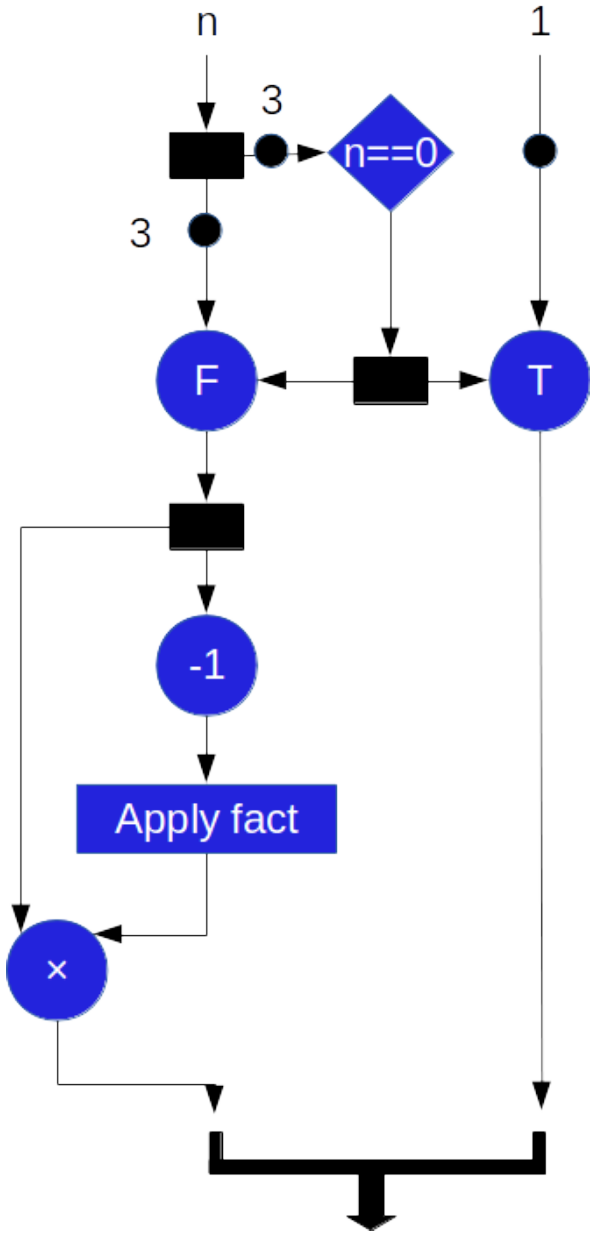
Example – Factorial

DFG for fact(3)



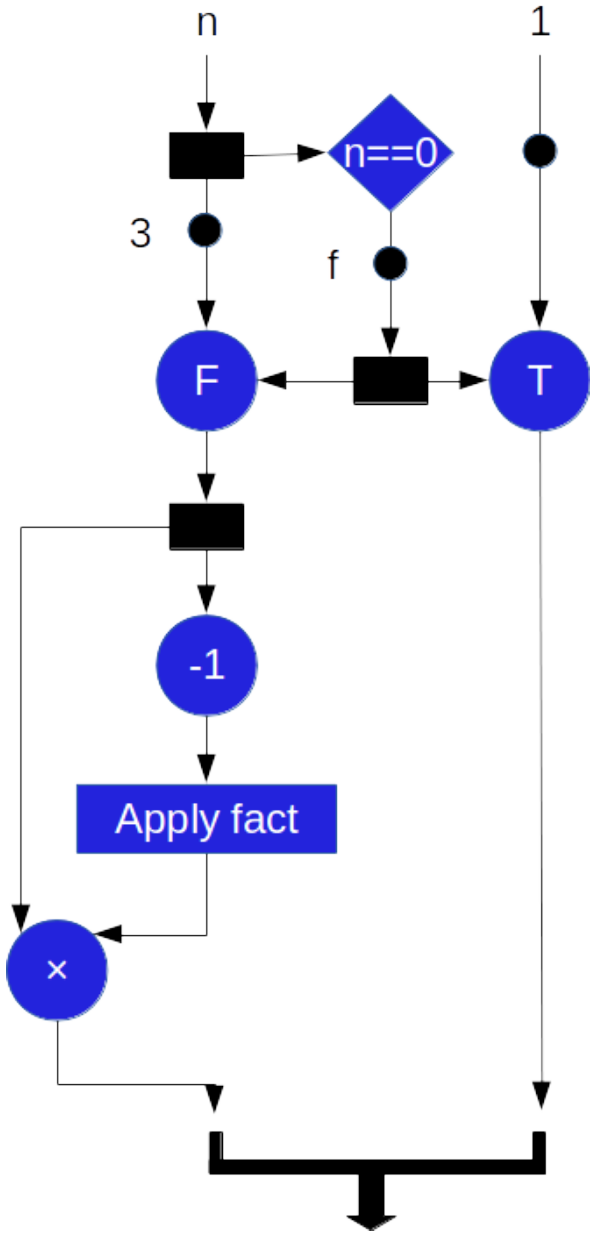
Example – Factorial

DFG for fact(3)



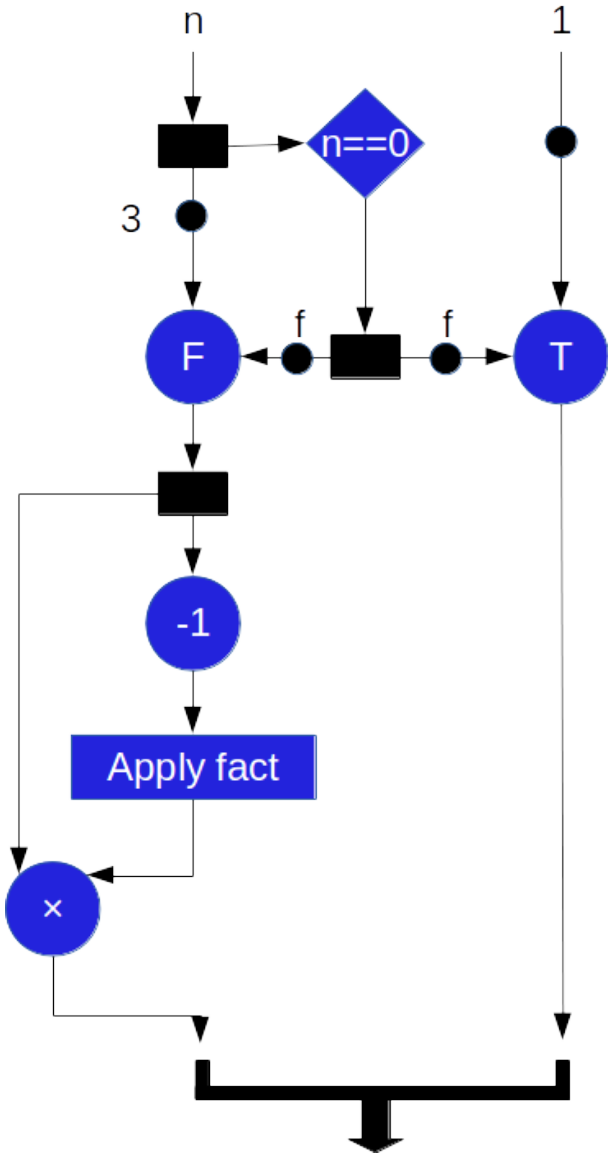
Example – Factorial

DFG for fact(3)



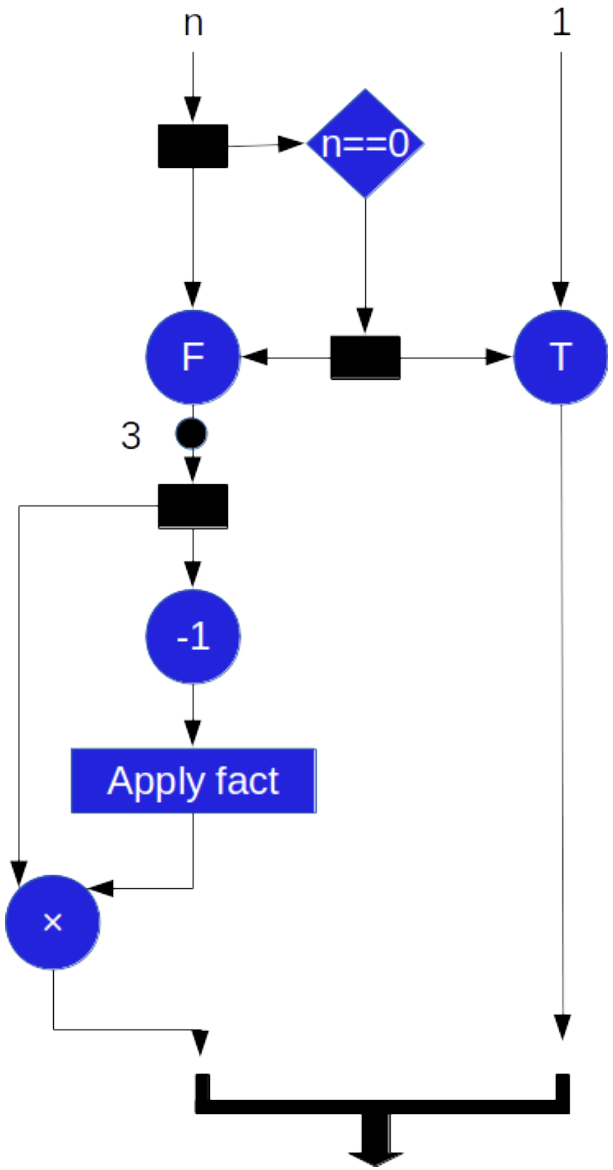
Example – Factorial

DFG for fact(3)



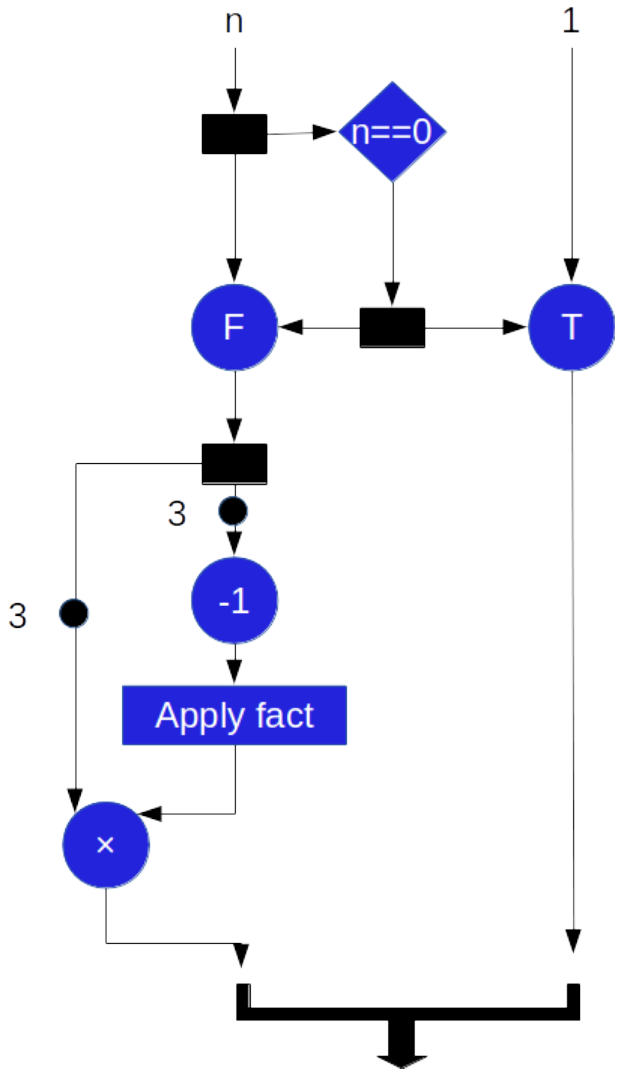
Example – Factorial

DFG for fact(3)



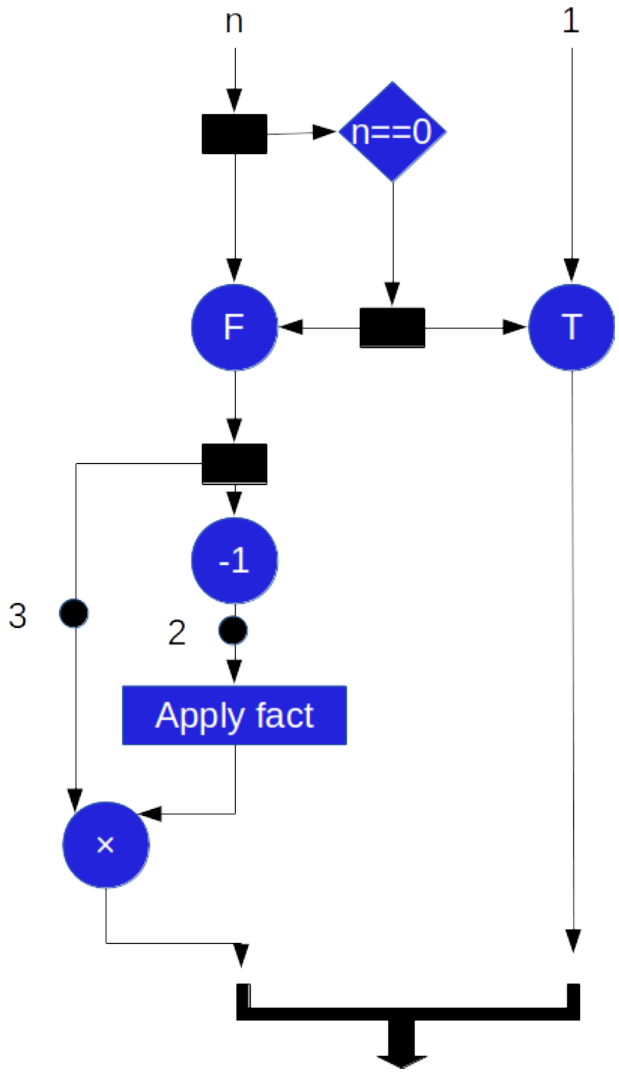
Example – Factorial

DFG for fact(3)



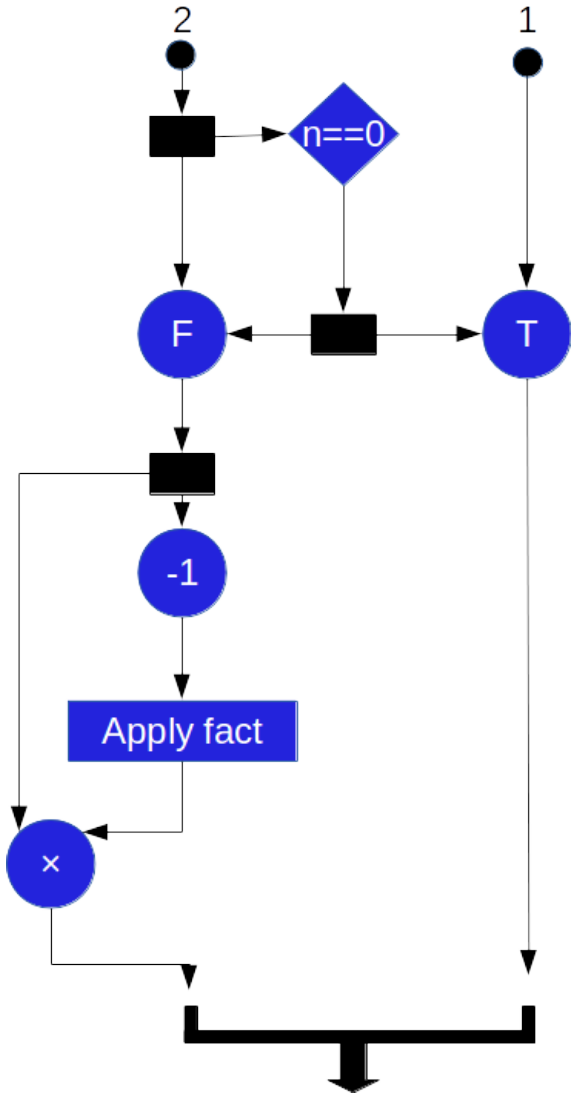
Example – Factorial

DFG for fact(3)



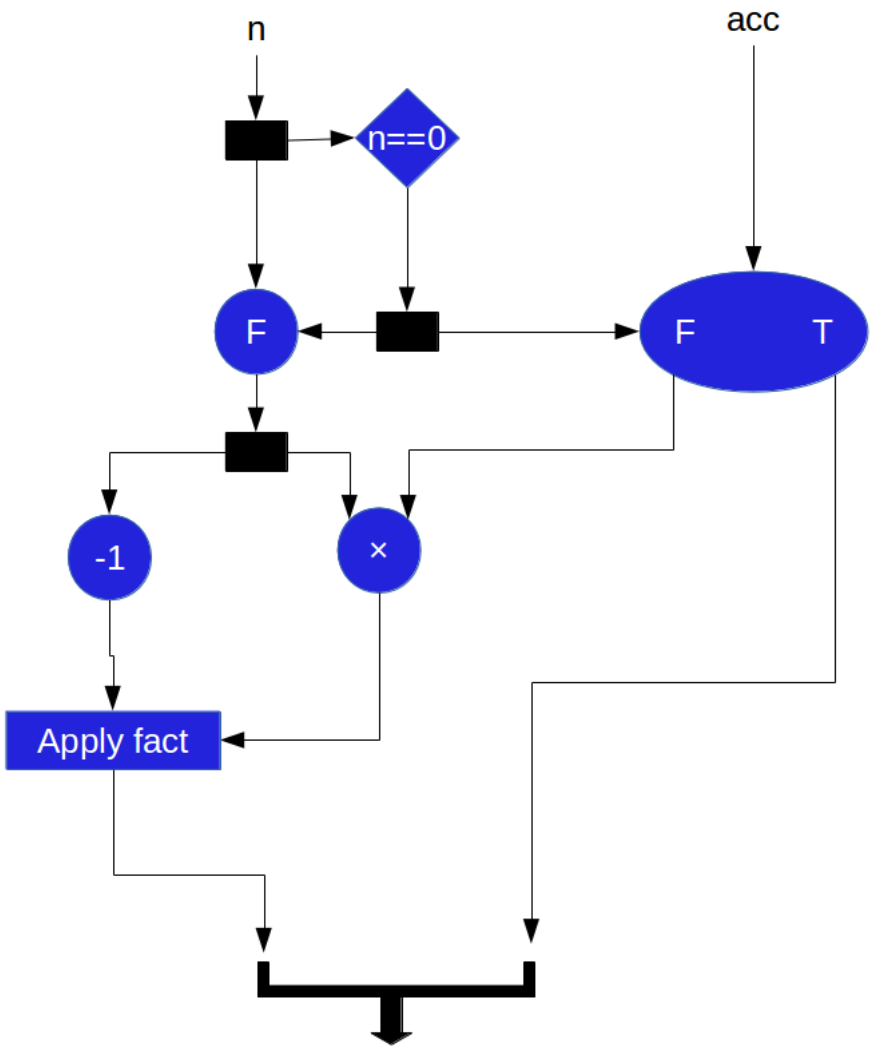
Example – Factorial

DFG for fact(3)



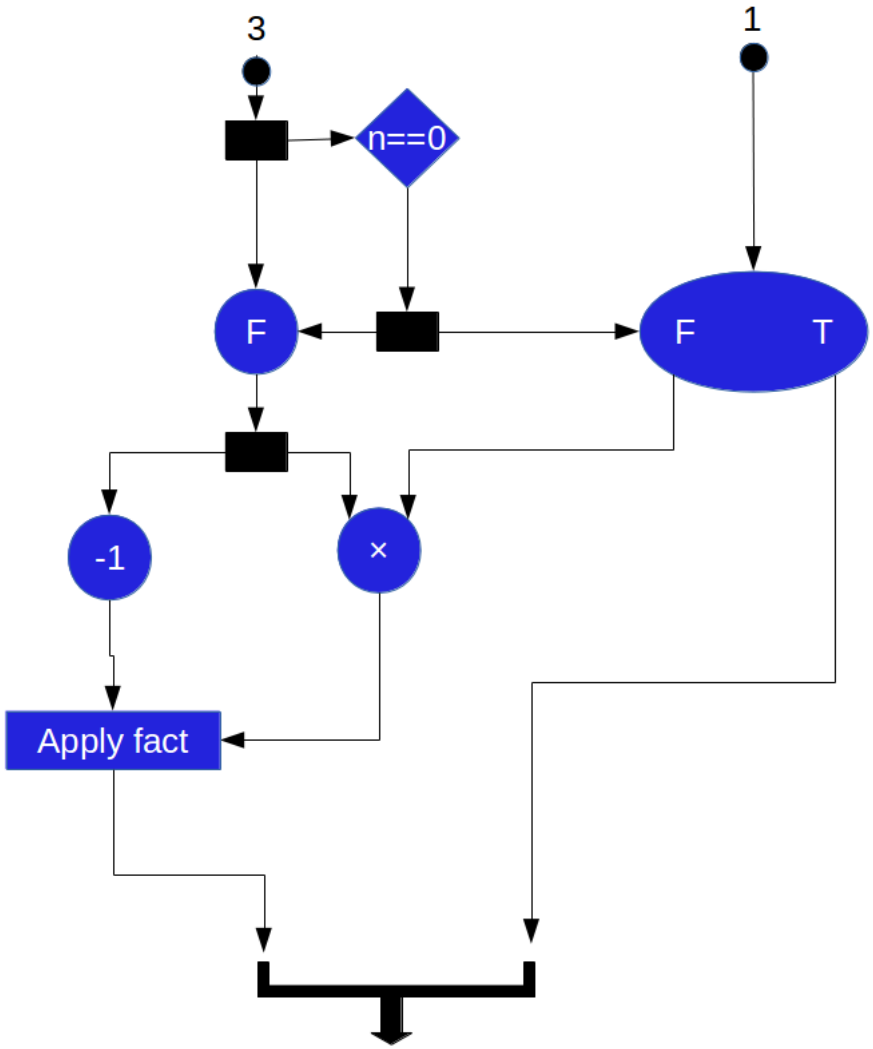
Factorial

Tail Recursive version – factorial(3)



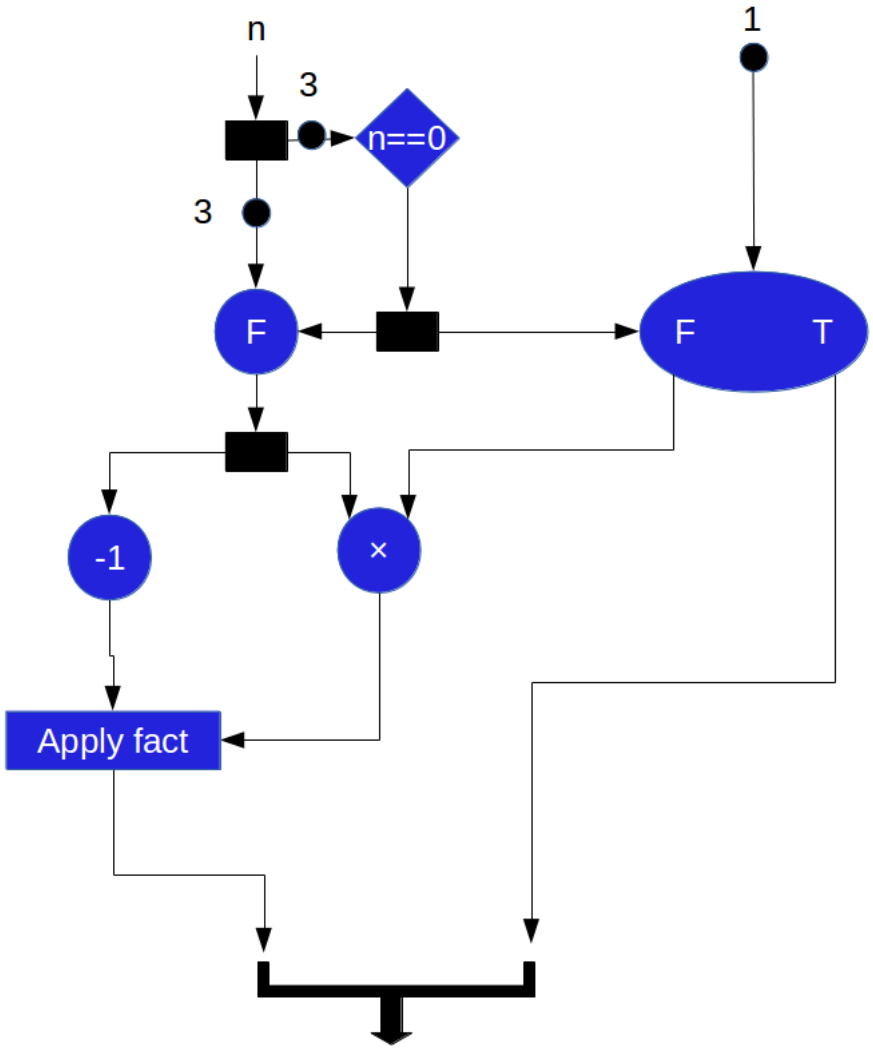
Factorial

Tail Recursive version – factorial(3)



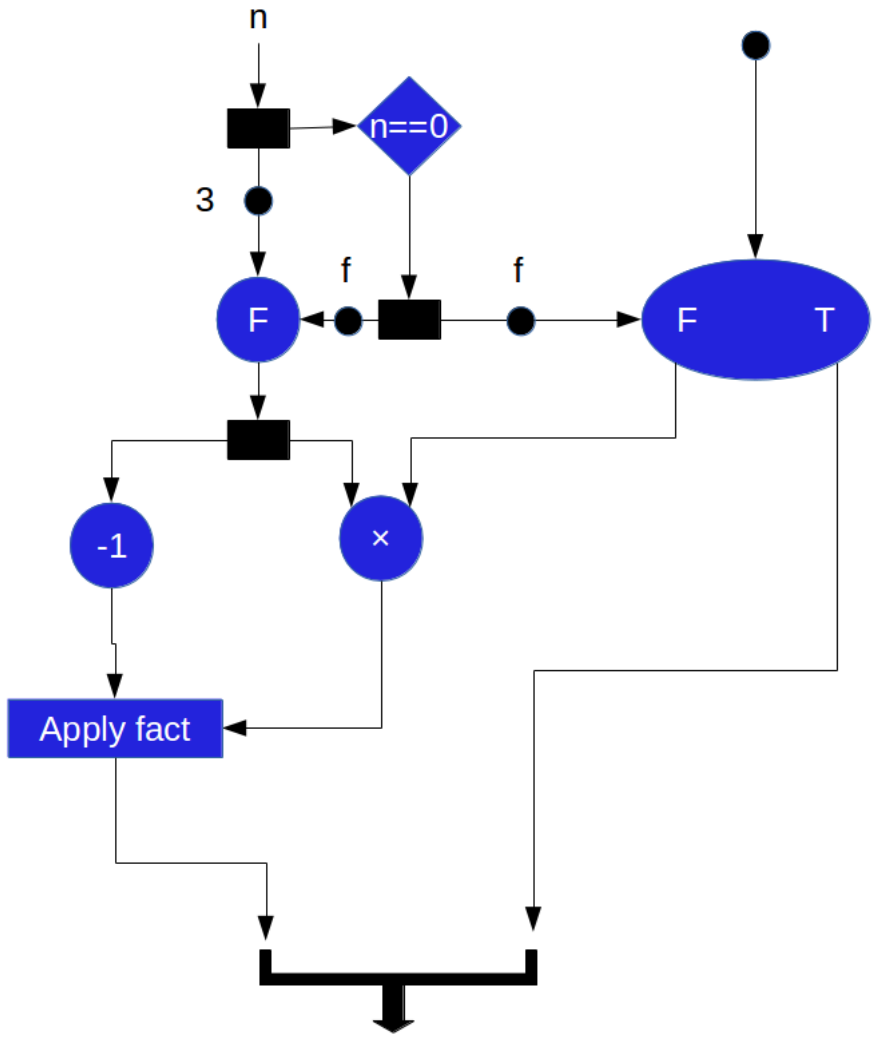
Factorial

Tail Recursive version – factorial(3)



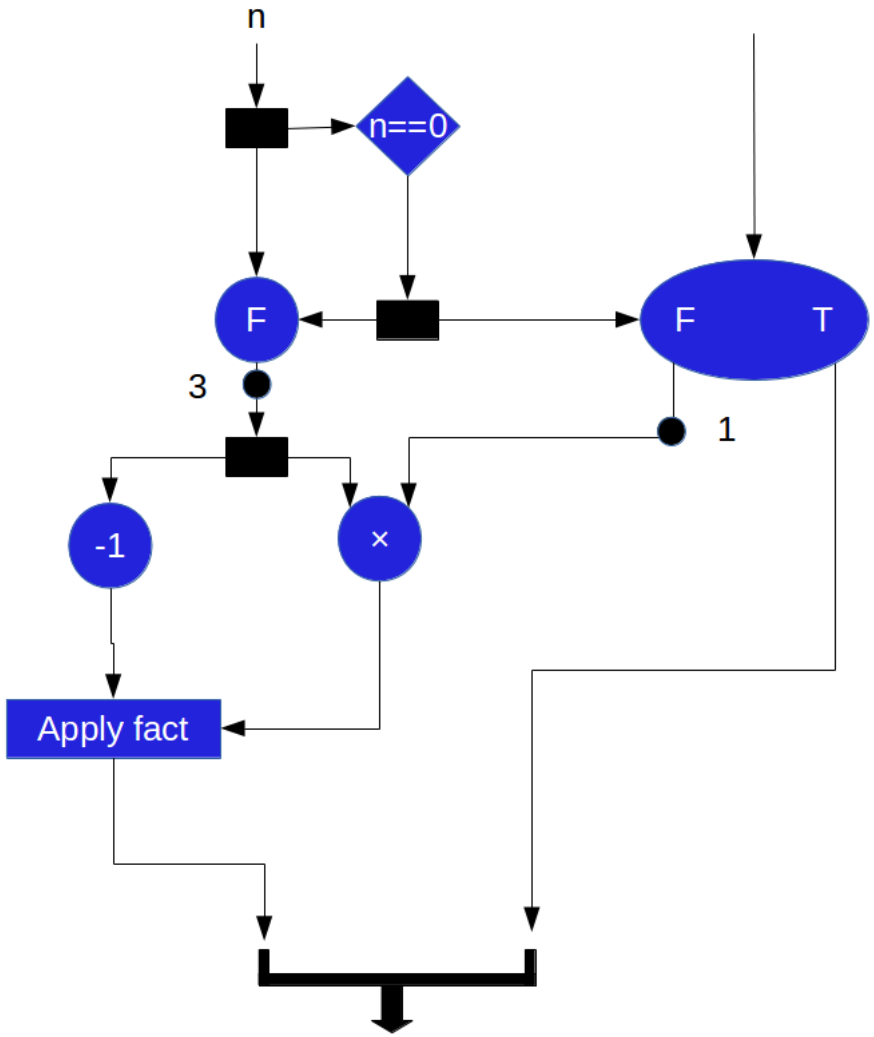
Factorial

Tail Recursive version – factorial(3)



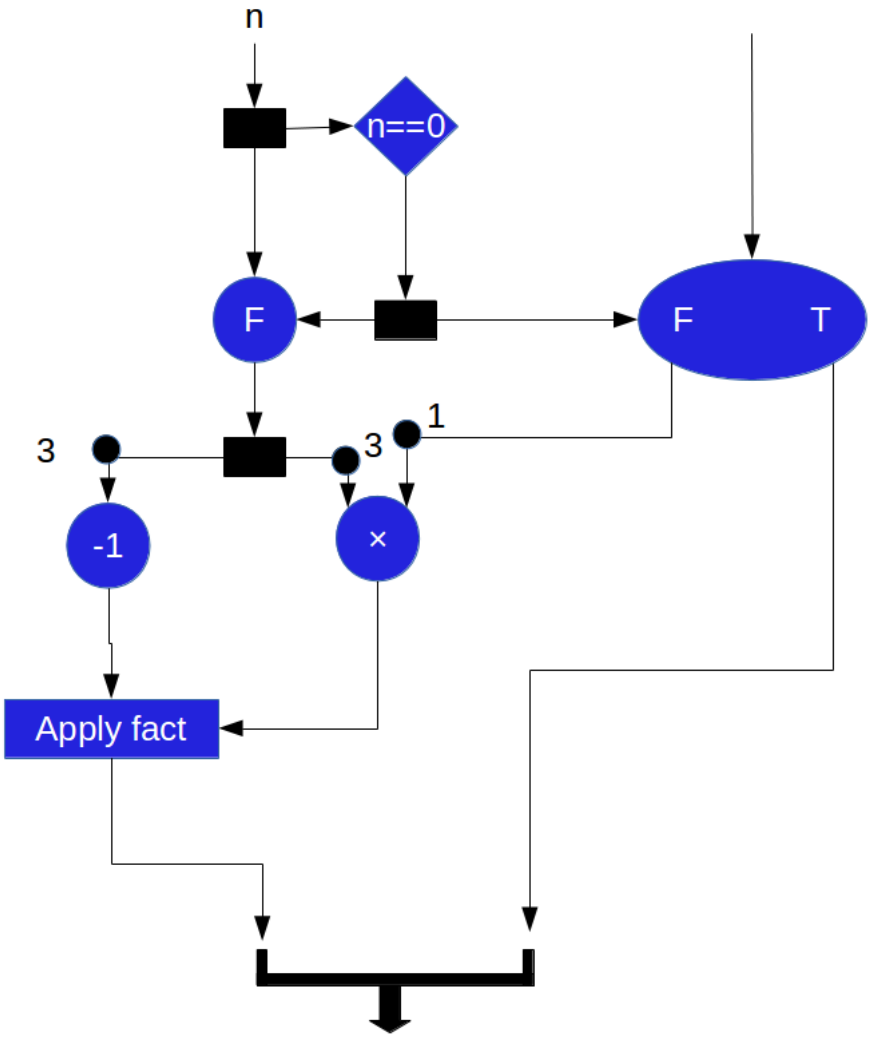
Factorial

Tail Recursive version – factorial(3)



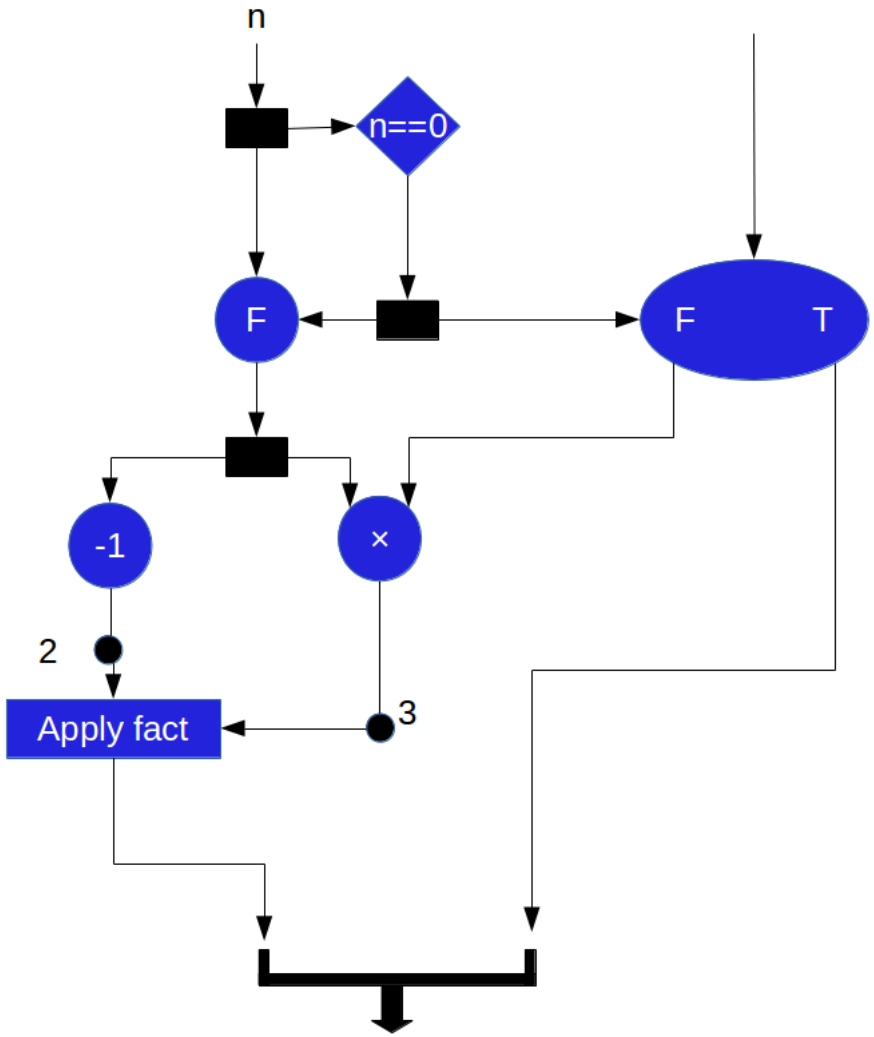
Factorial

Tail Recursive version – factorial(3)



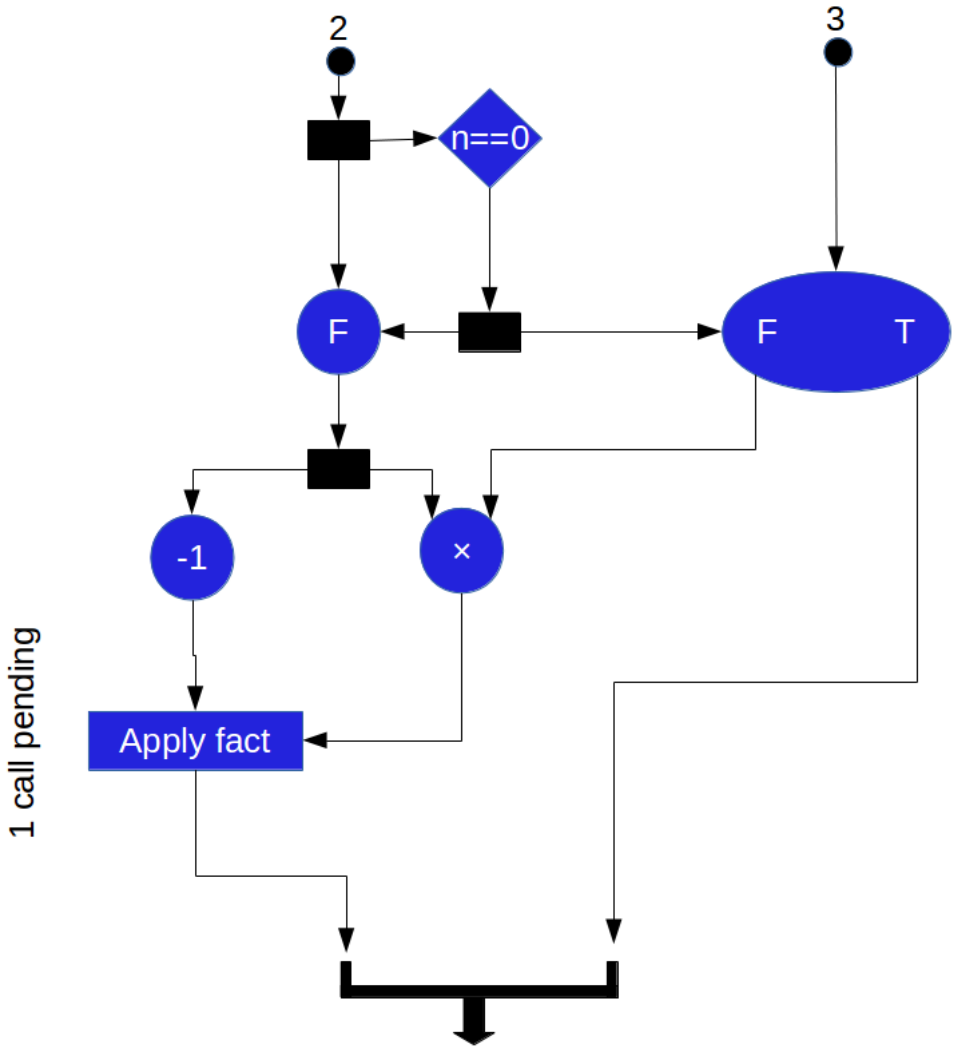
Factorial

Tail Recursive version – factorial(3)



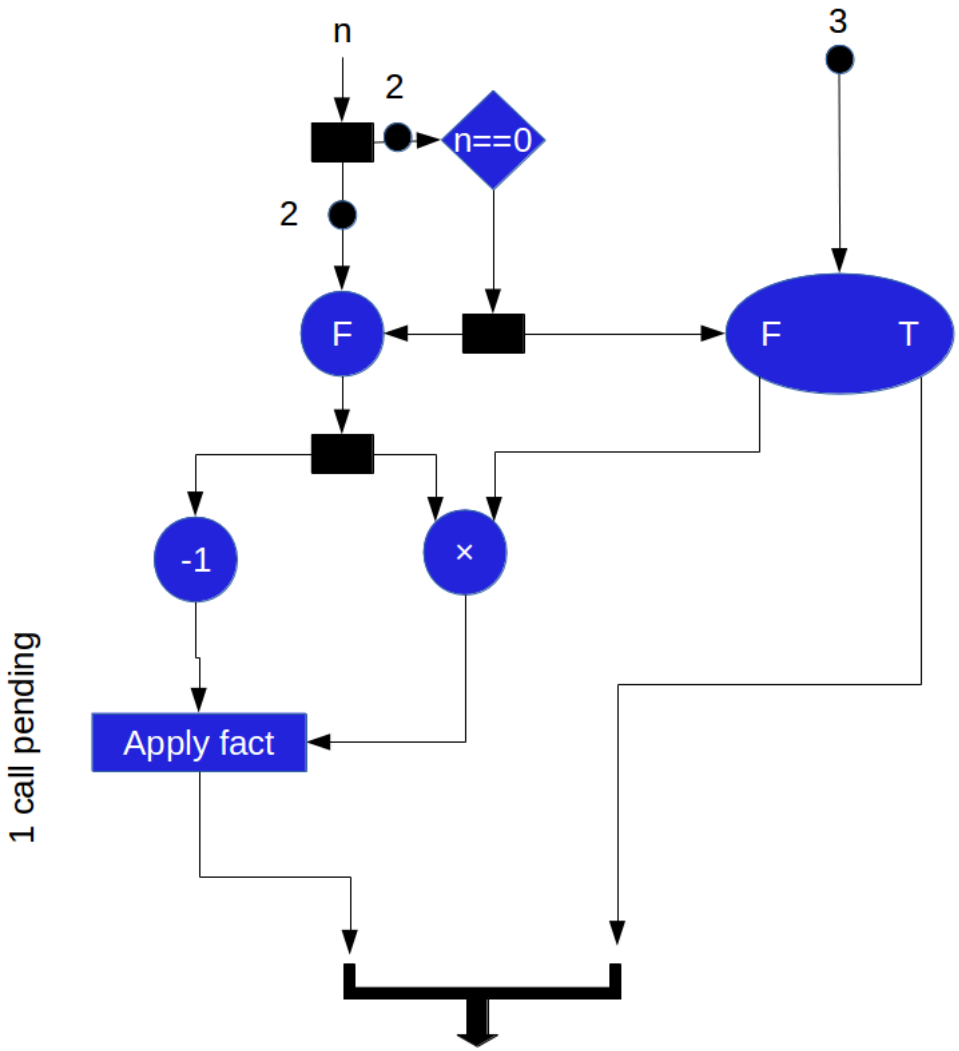
Factorial

Tail Recursive version – factorial(3)



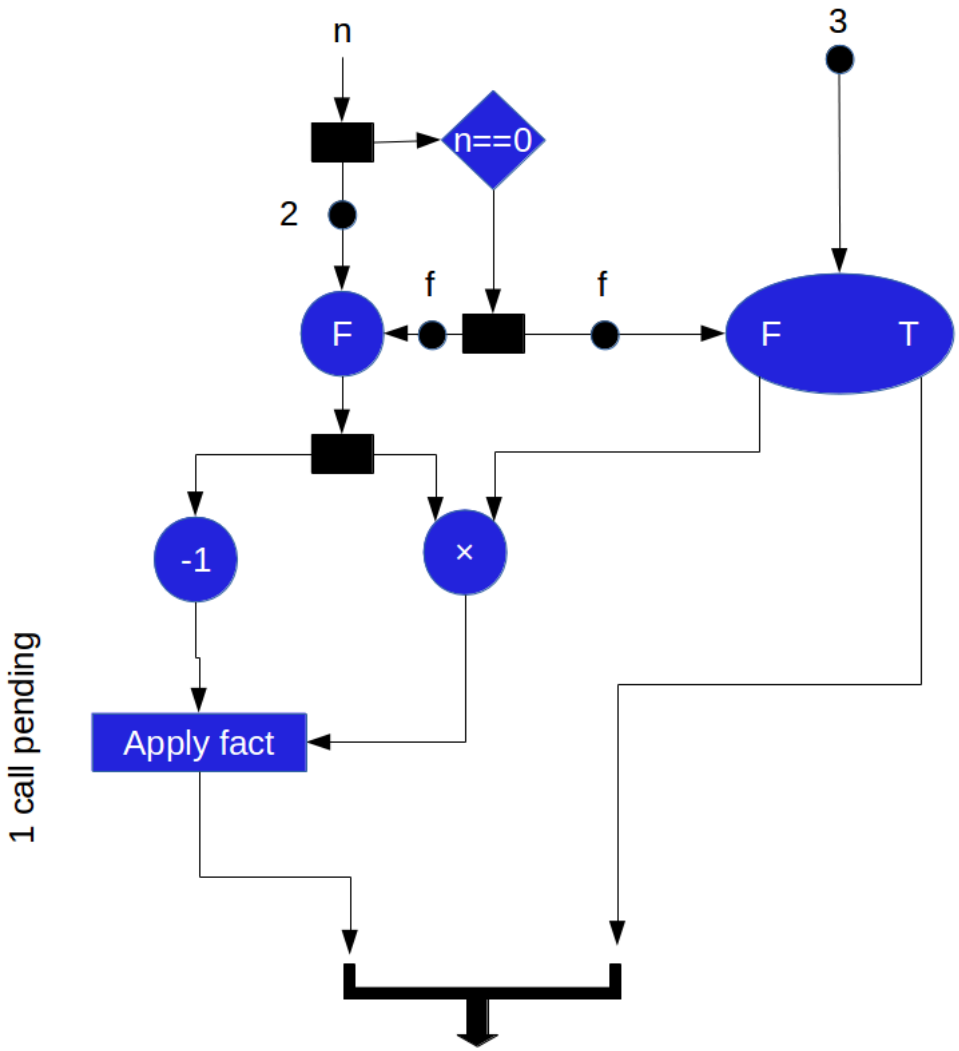
Factorial

Tail Recursive version – factorial(3)



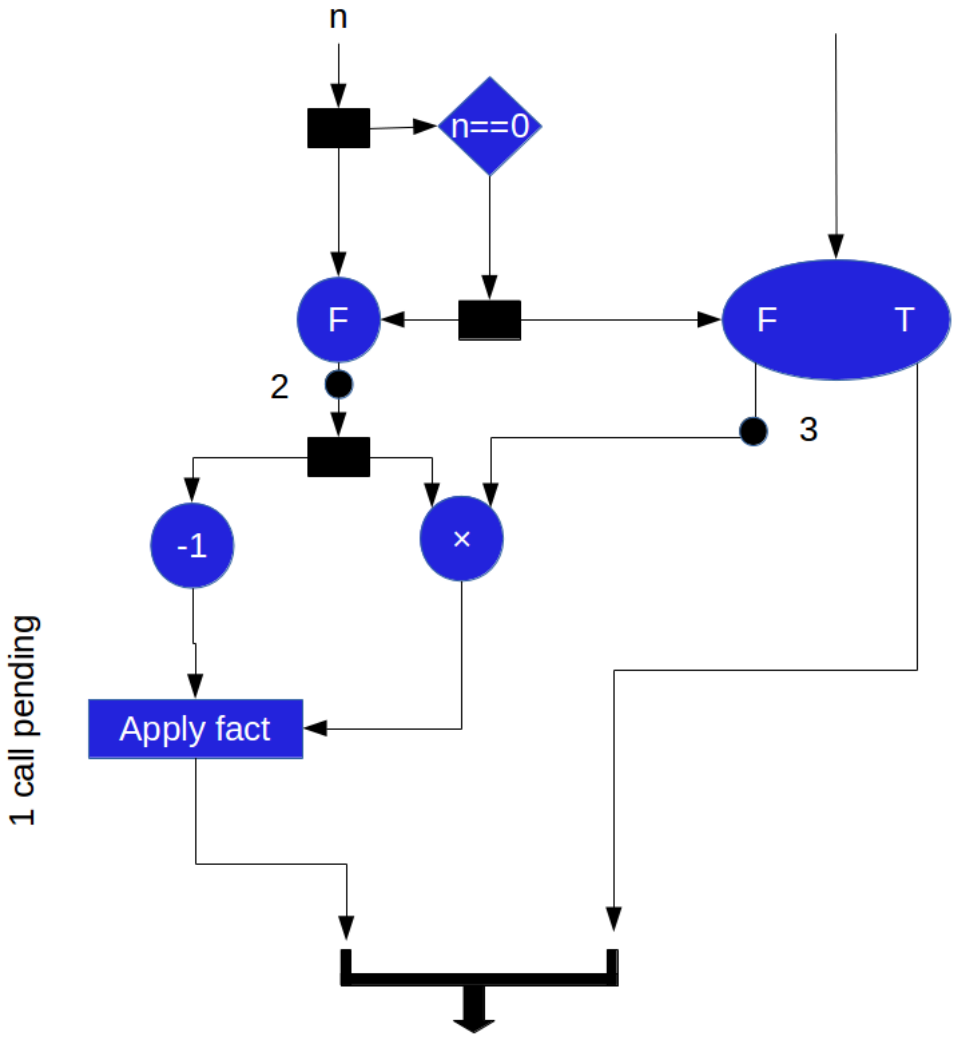
Factorial

Tail Recursive version – factorial(3)



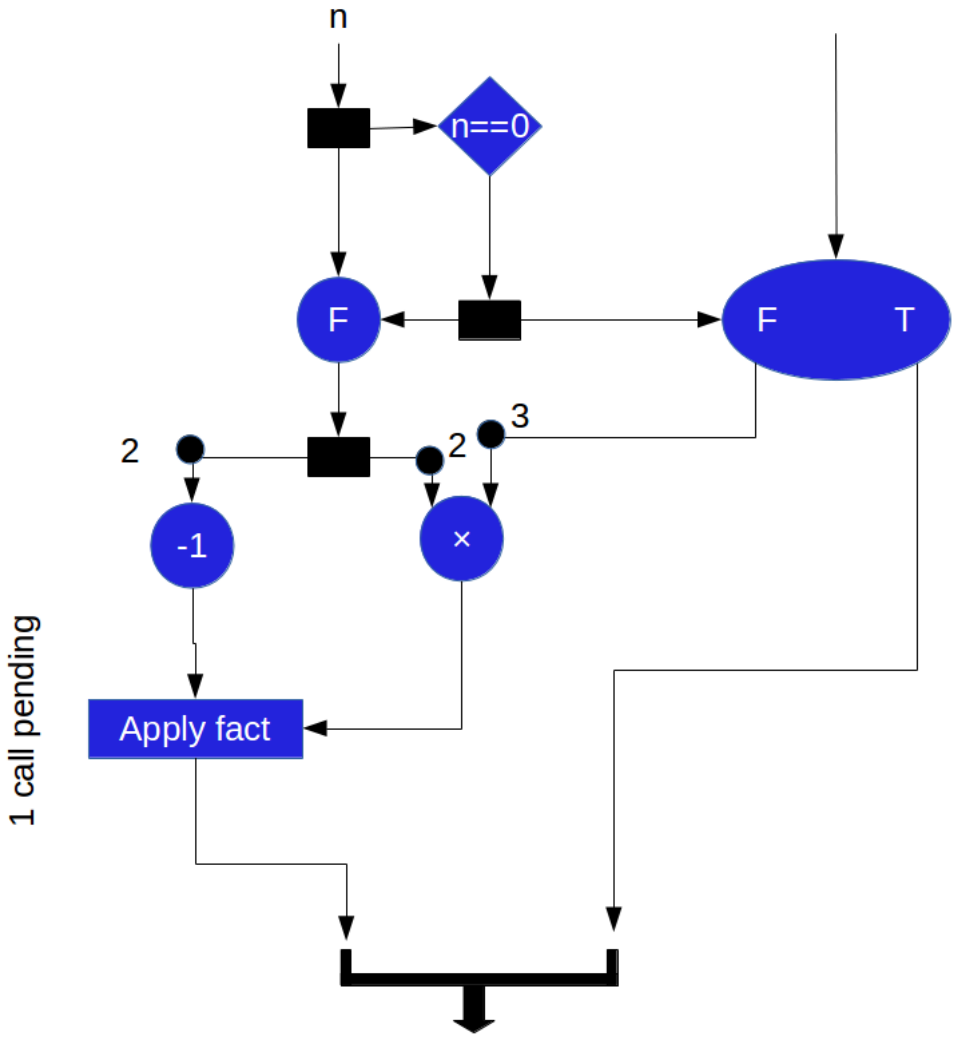
Factorial

Tail Recursive version – factorial(3)



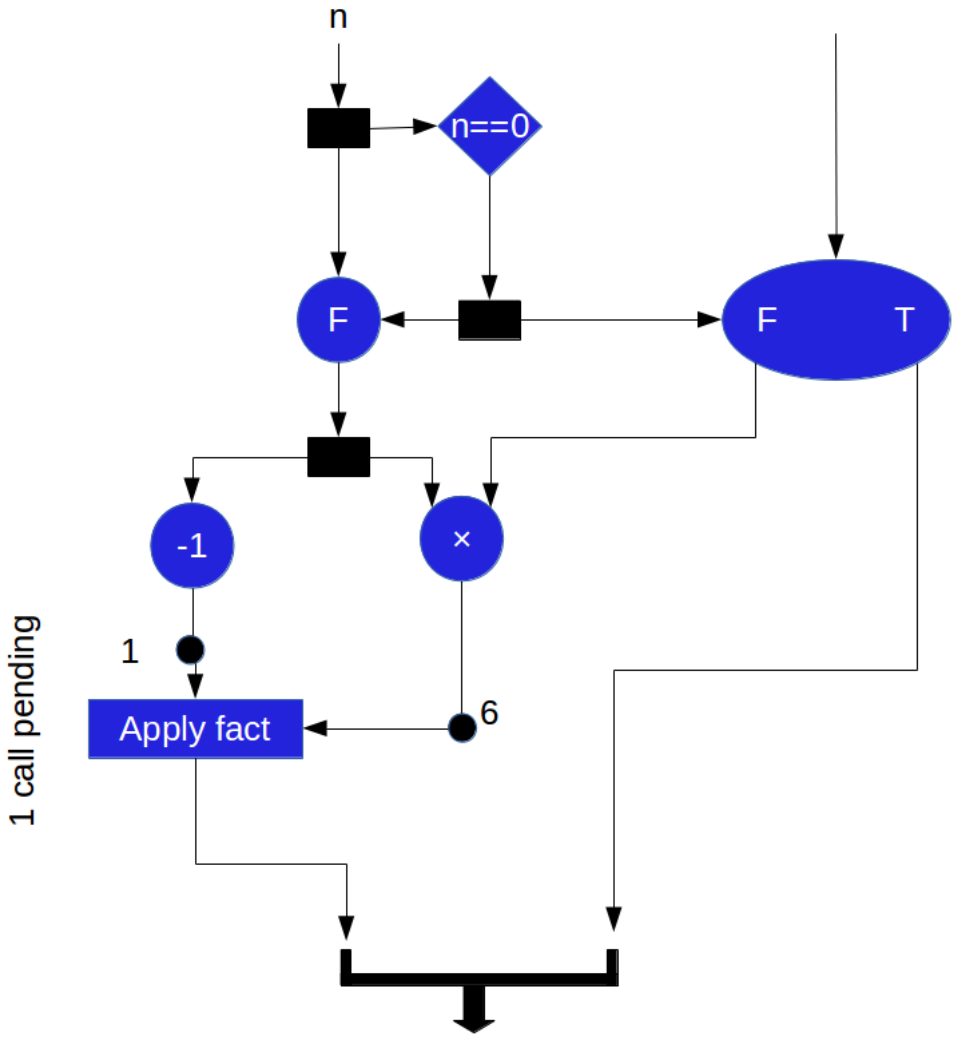
Factorial

Tail Recursive version – factorial(3)



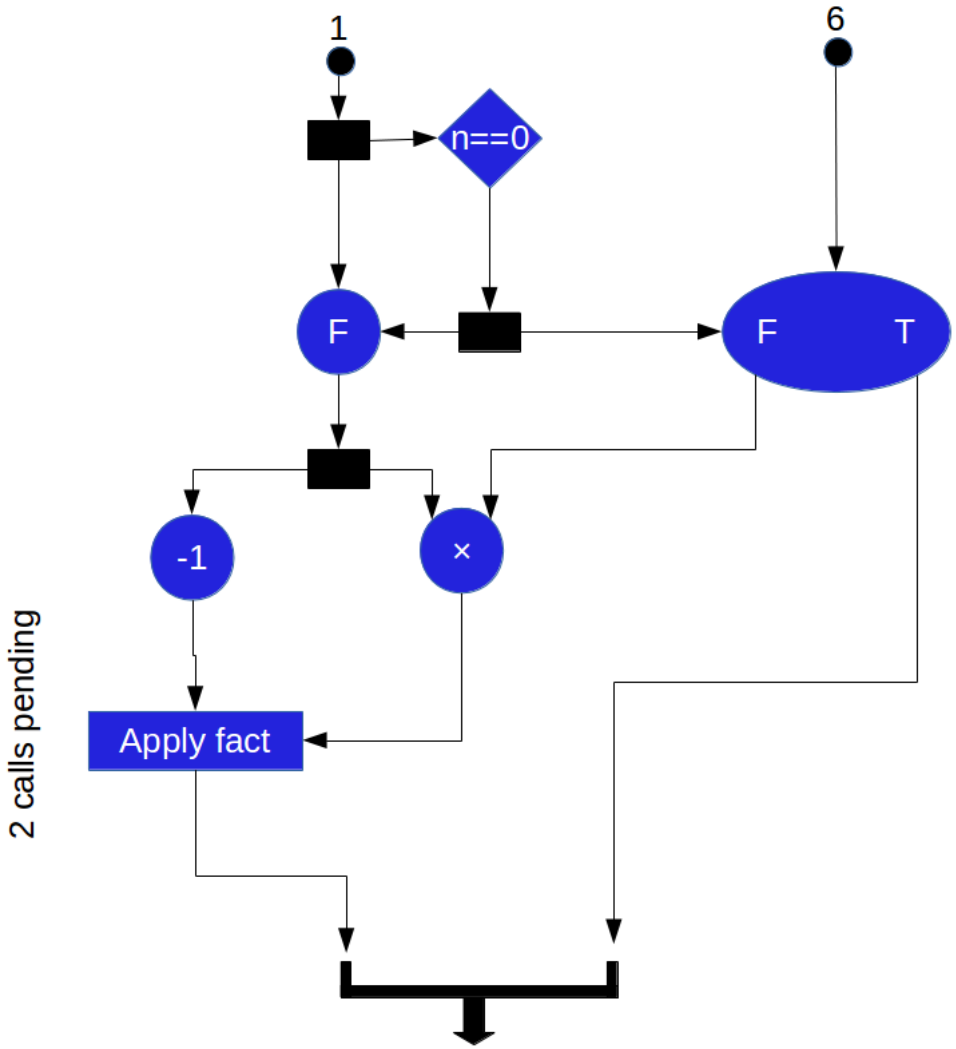
Factorial

Tail Recursive version – factorial(3)



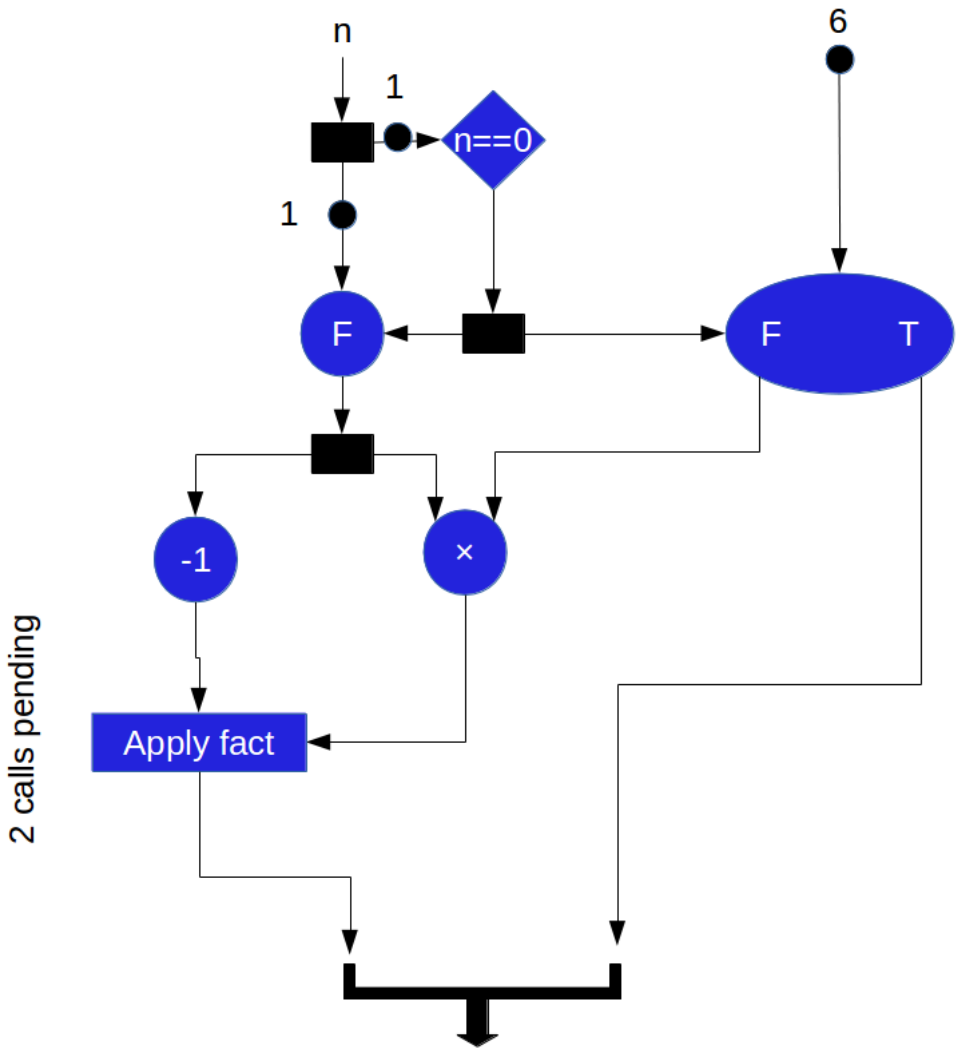
Factorial

Tail Recursive version – factorial(3)



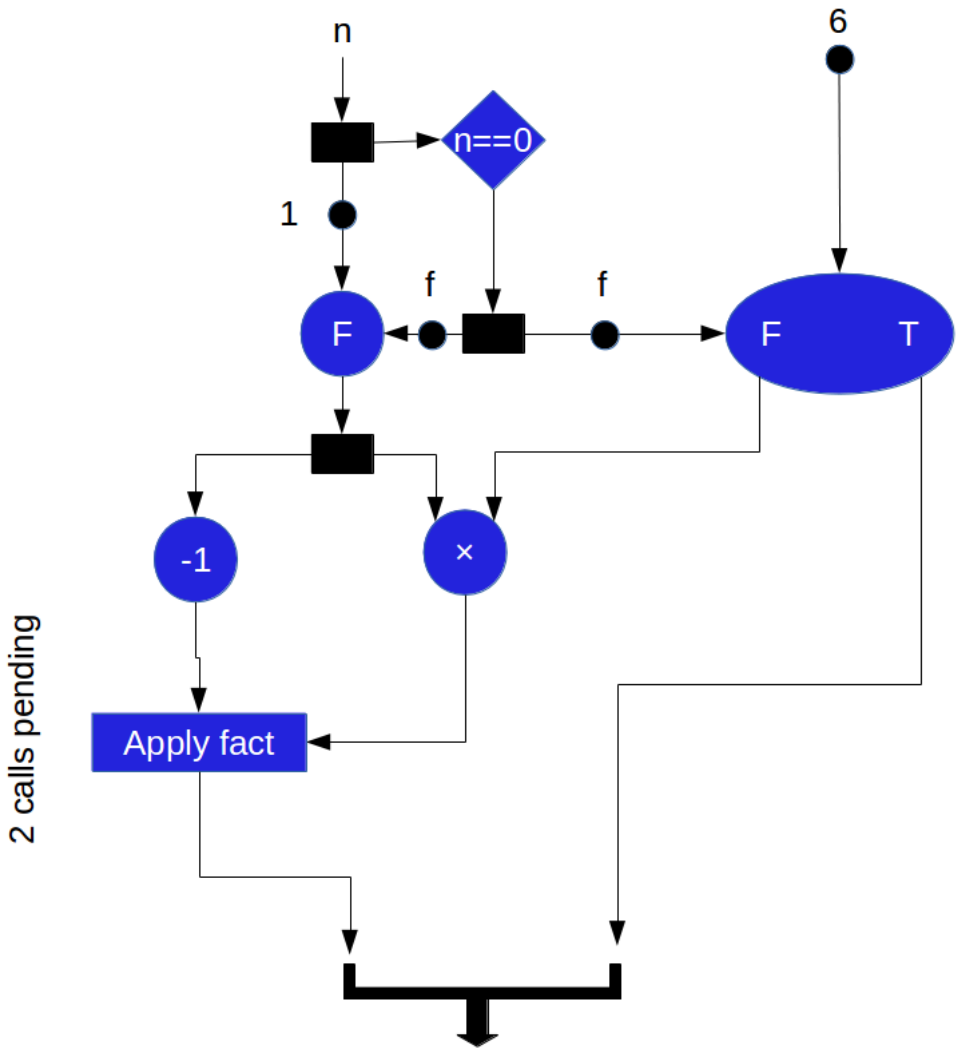
Factorial

Tail Recursive version – factorial(3)



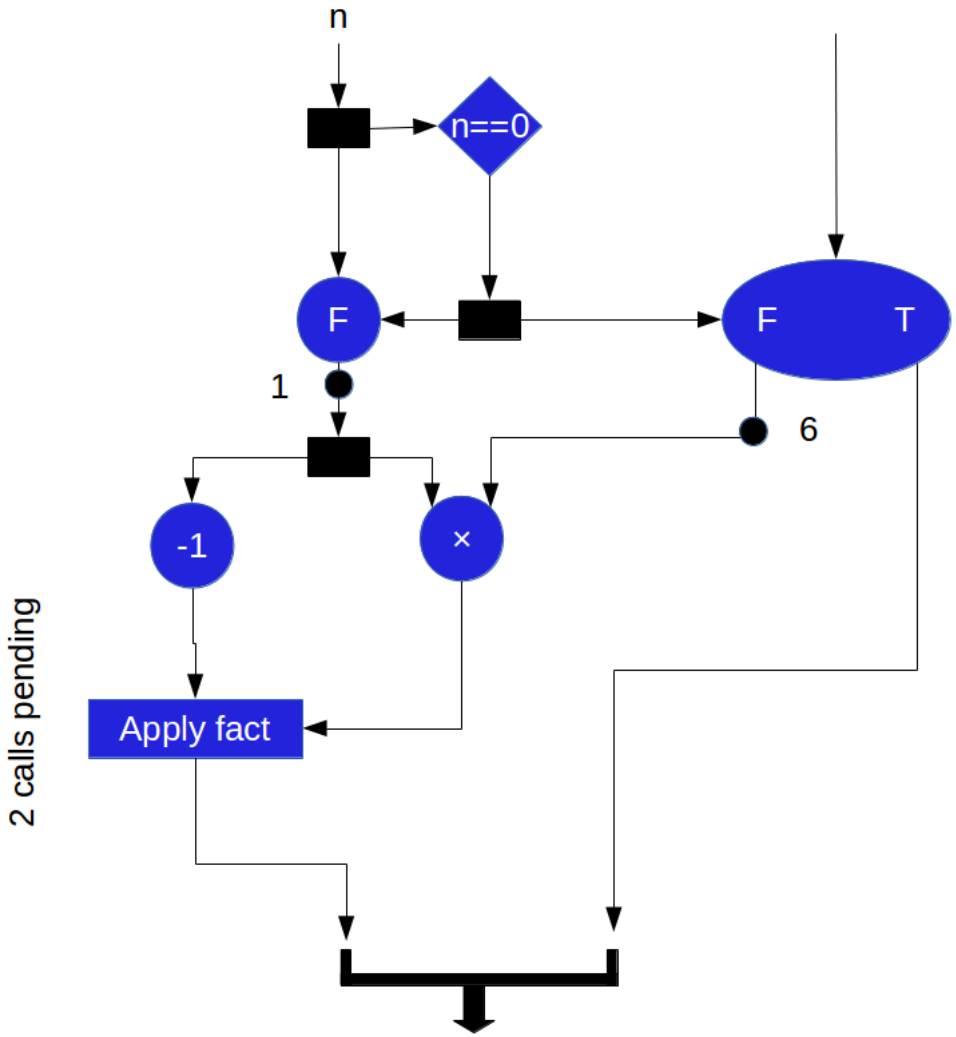
Factorial

Tail Recursive version – factorial(3)



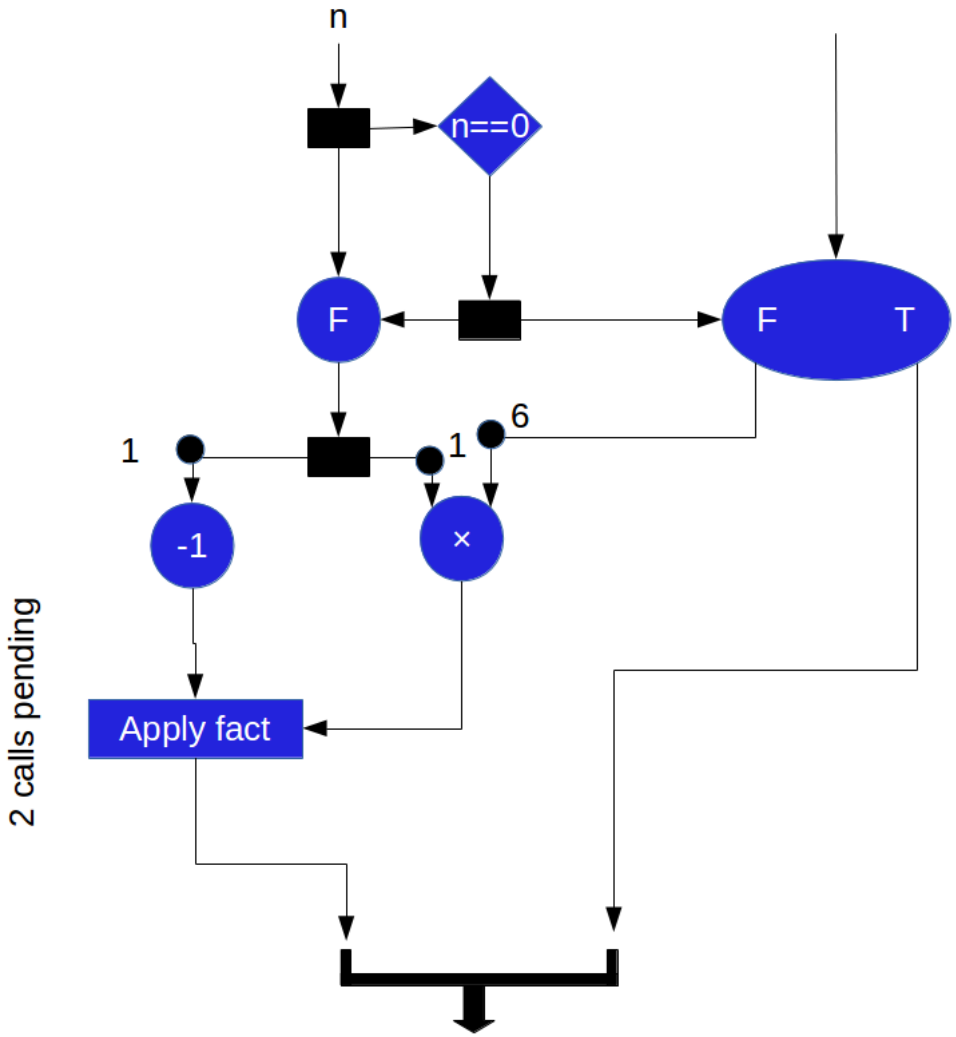
Factorial

Tail Recursive version – factorial(3)



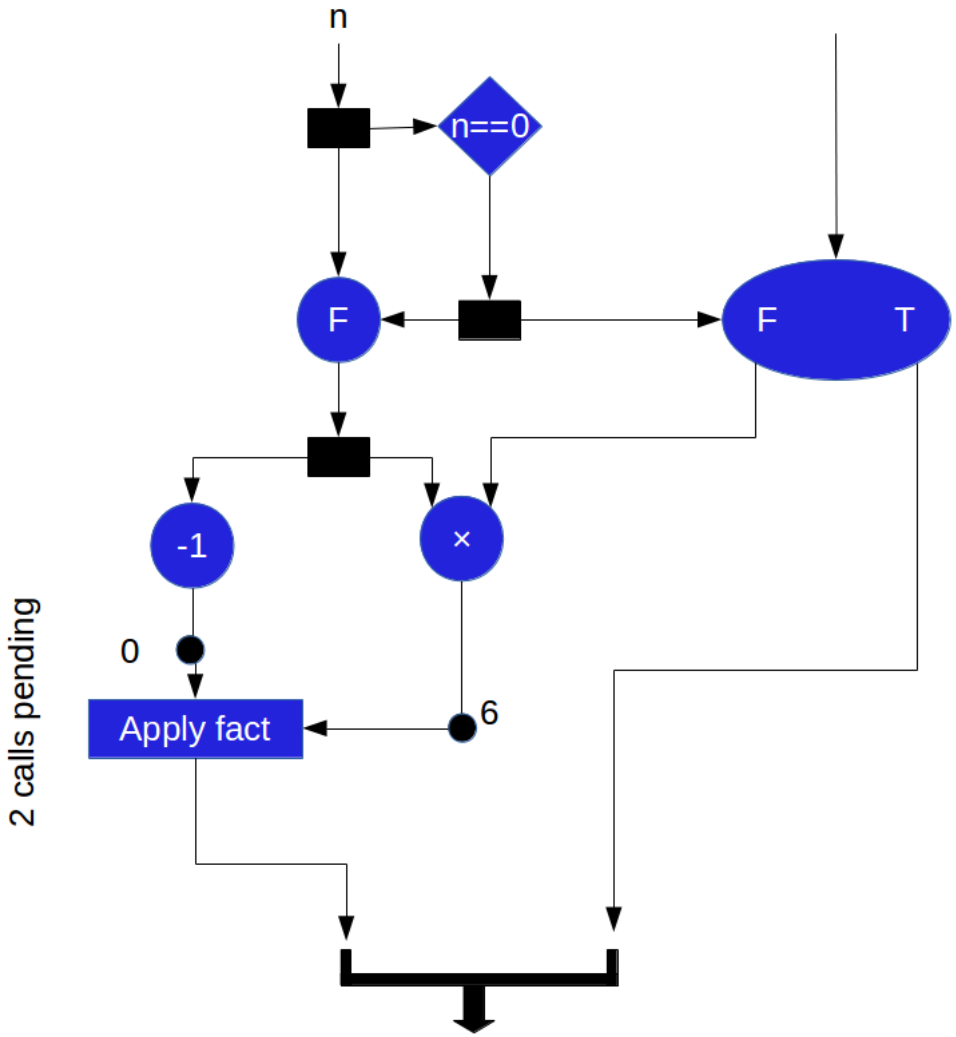
Factorial

Tail Recursive version – factorial(3)



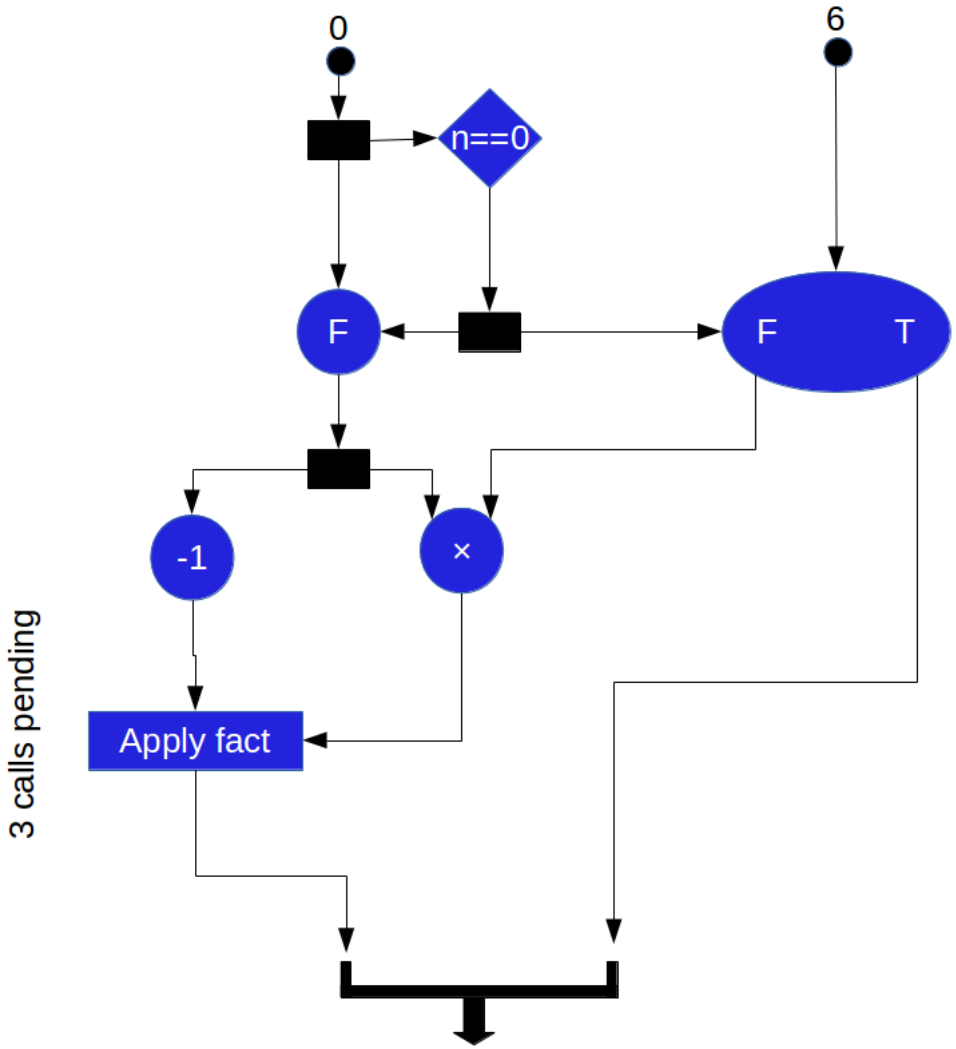
Factorial

Tail Recursive version – factorial(3)



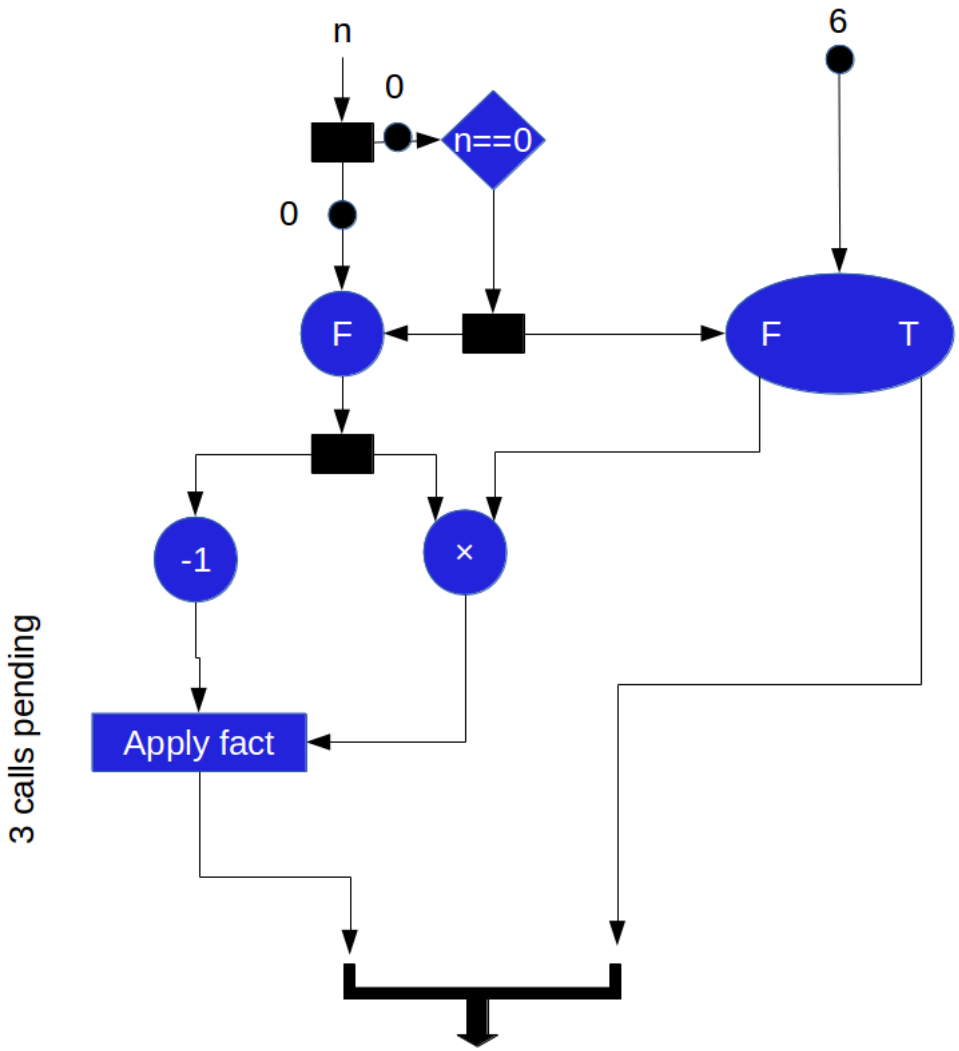
Factorial

Tail Recursive version – factorial(3)



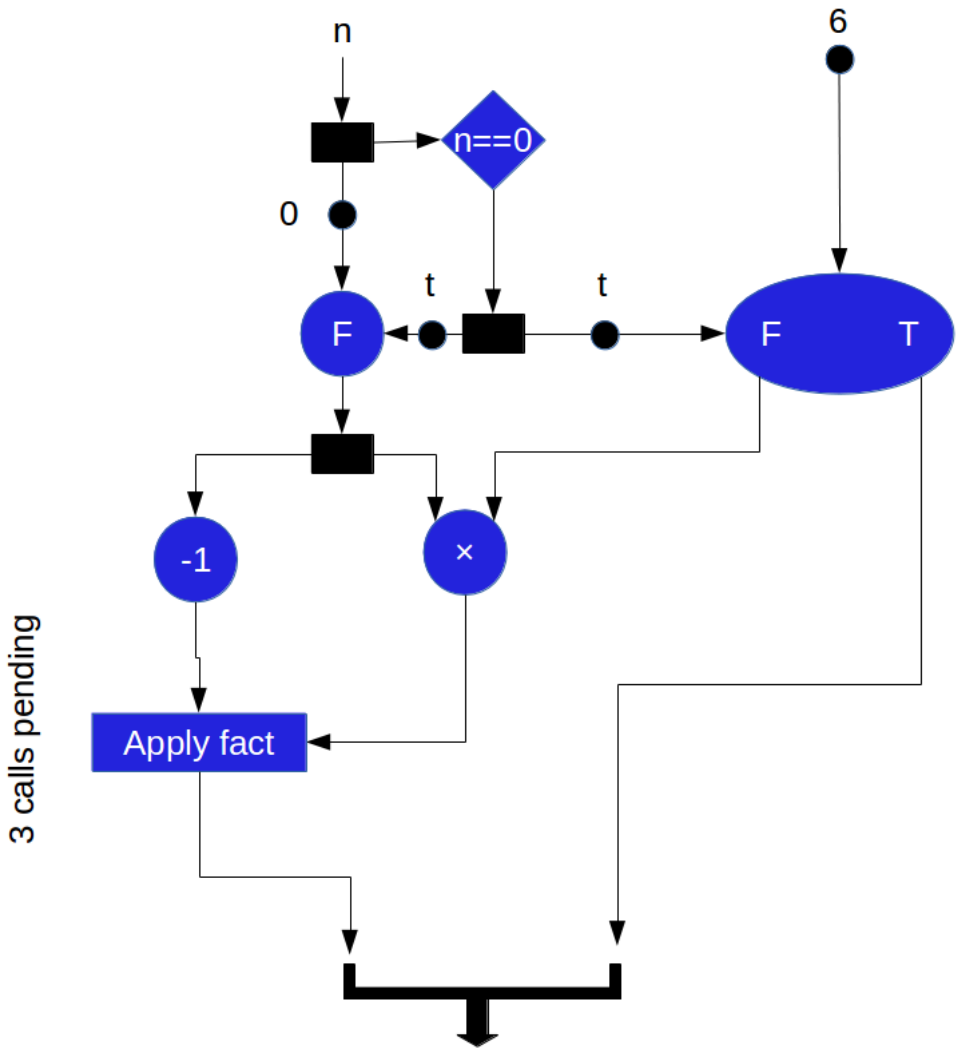
Factorial

Tail Recursive version – factorial(3)



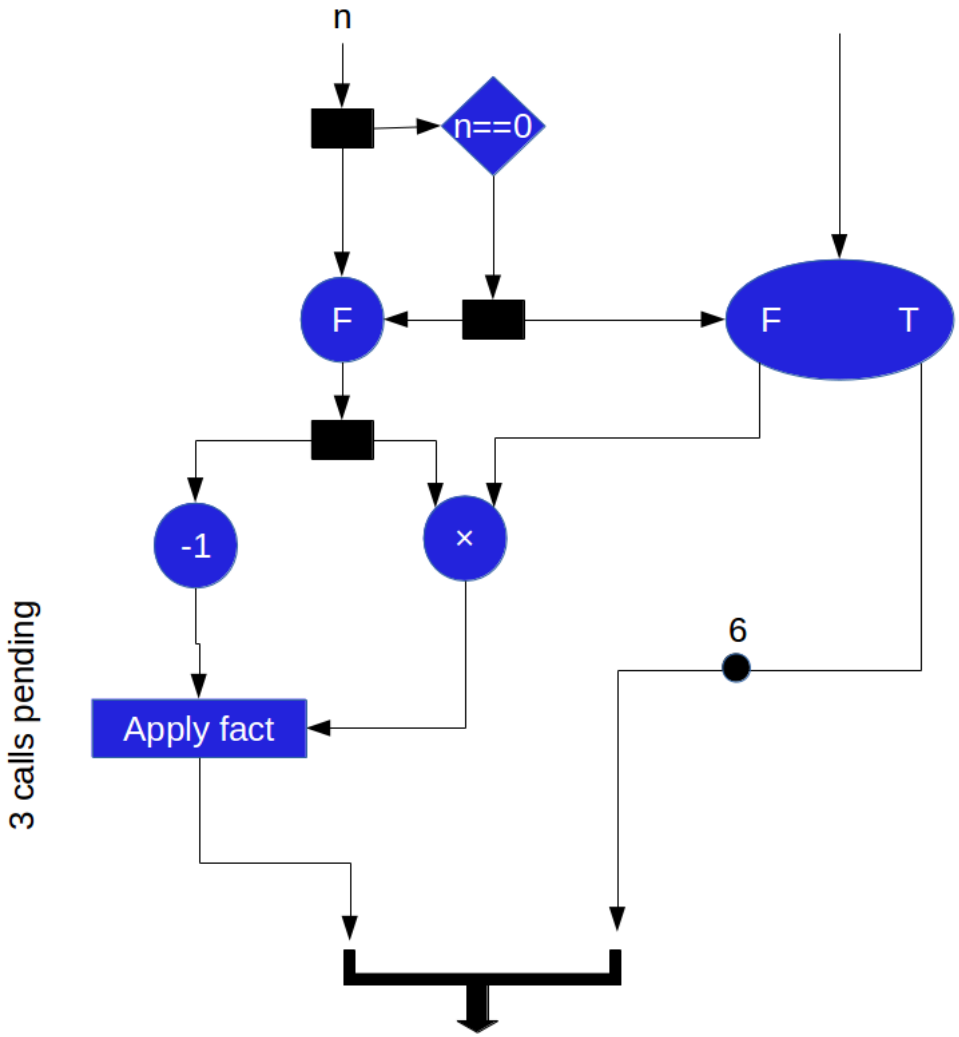
Factorial

Tail Recursive version – factorial(3)



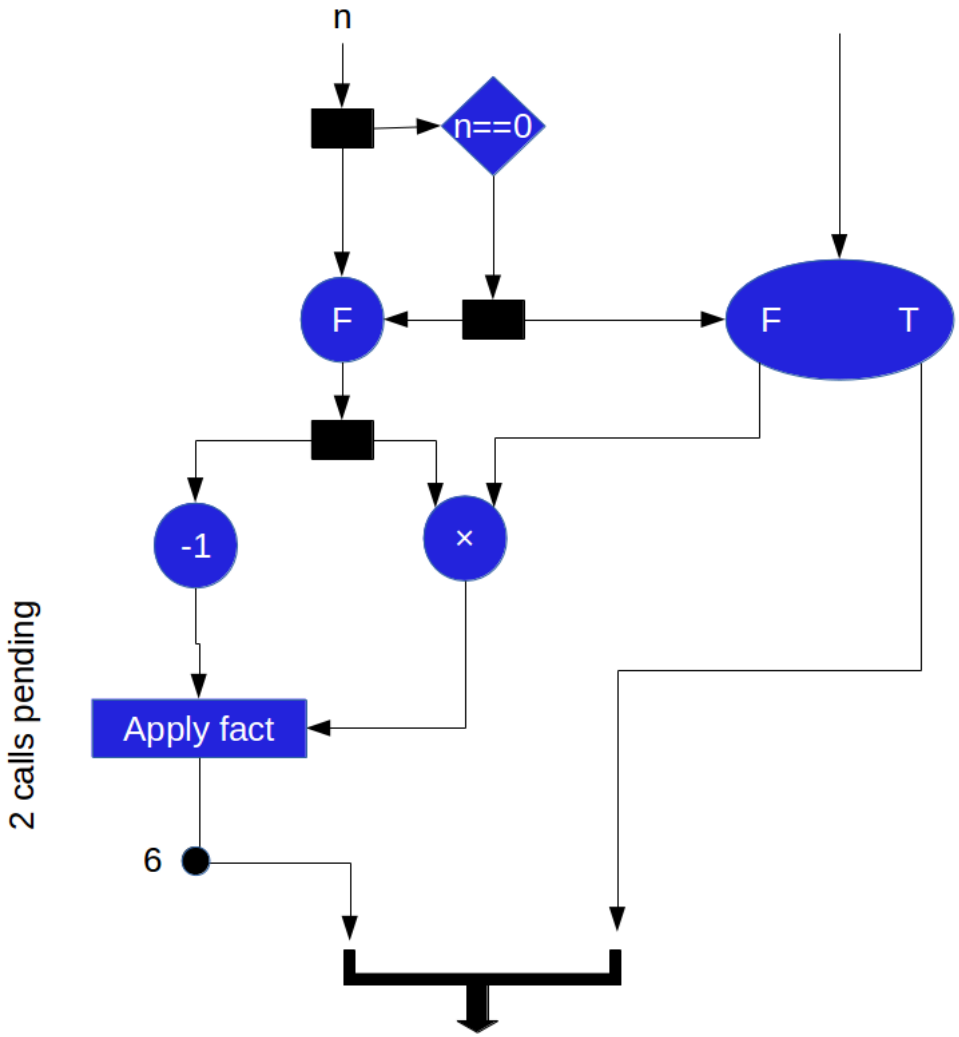
Factorial

Tail Recursive version – factorial(3)



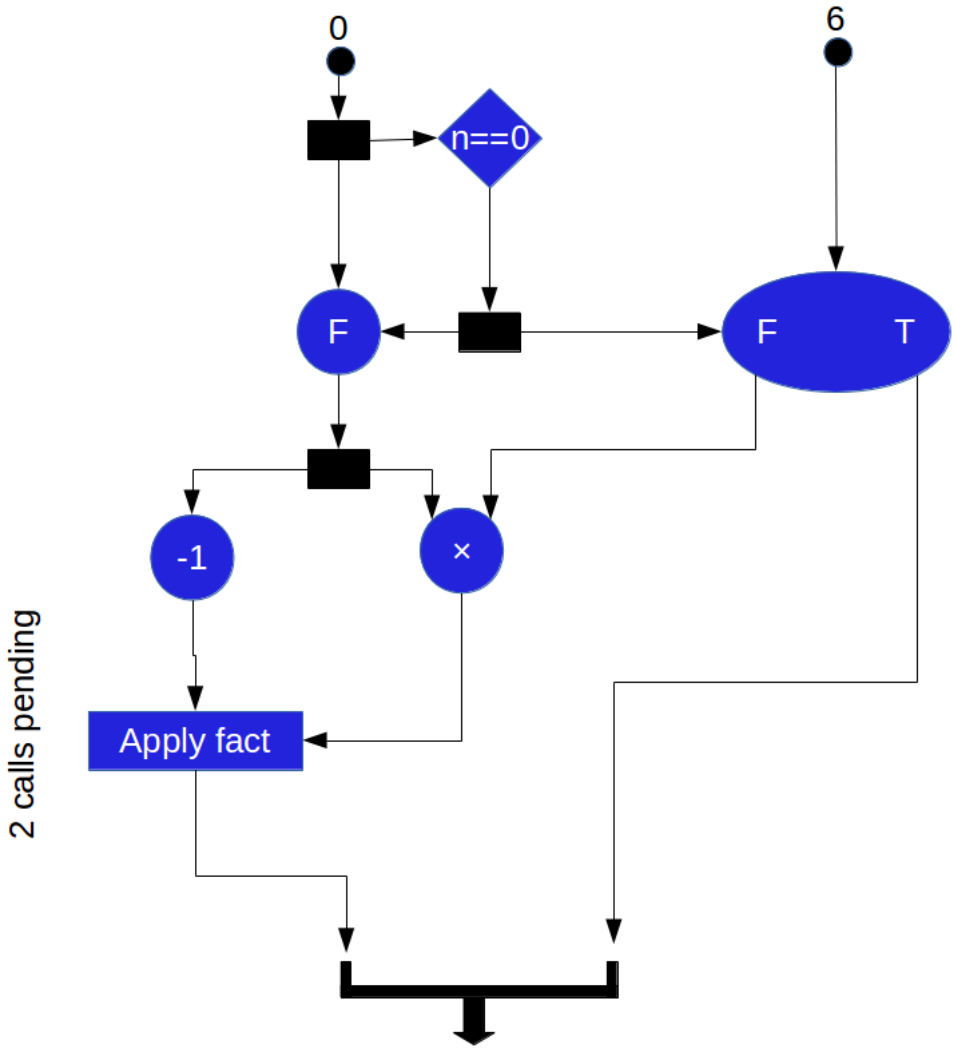
Factorial

Tail Recursive version – factorial(3)



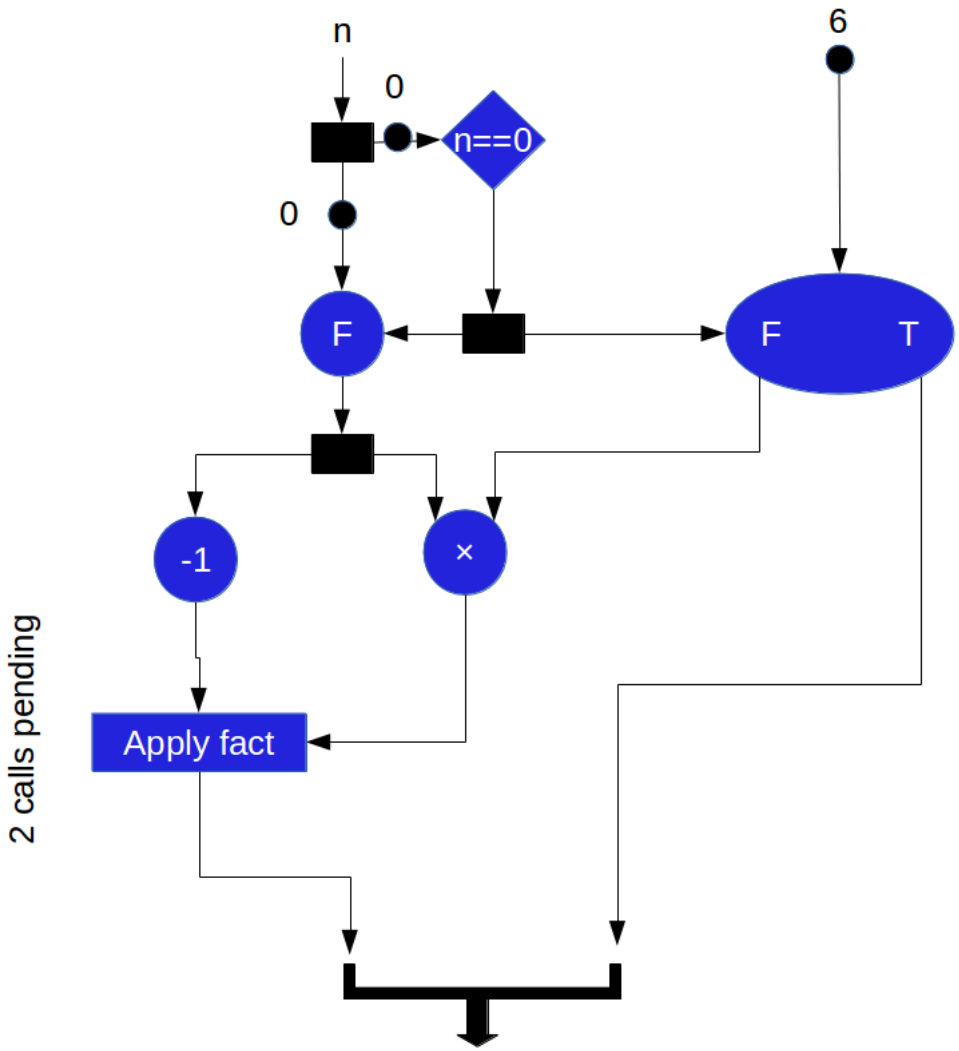
Factorial

Tail Recursive version – factorial(3)



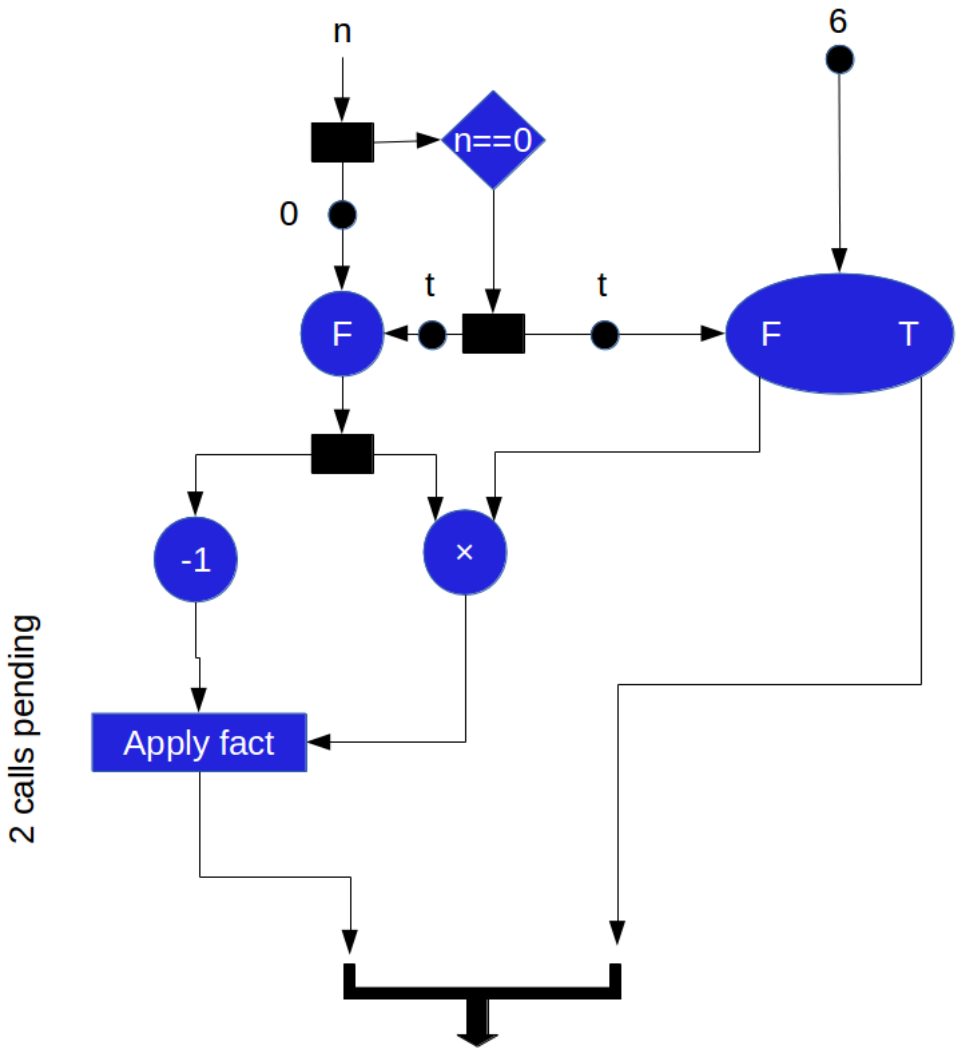
Factorial

Tail Recursive version – factorial(3)



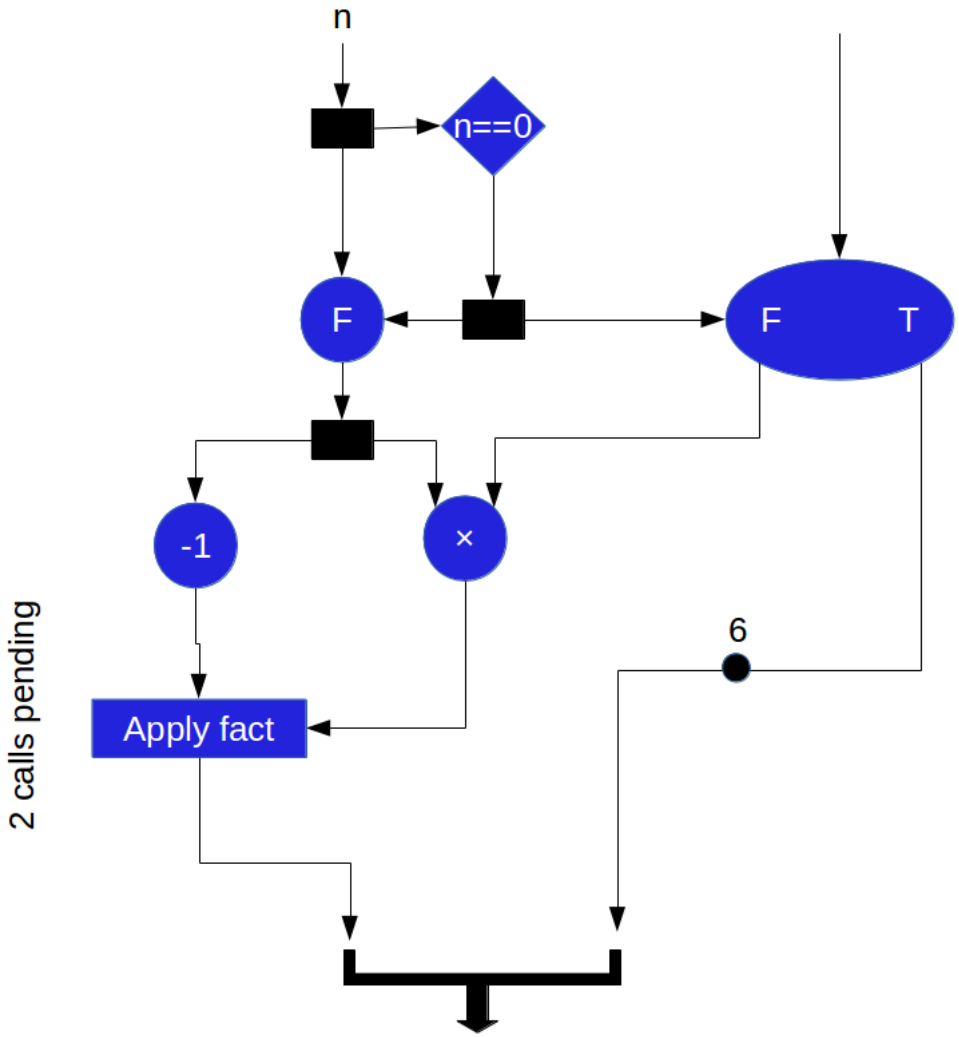
Factorial

Tail Recursive version – factorial(3)



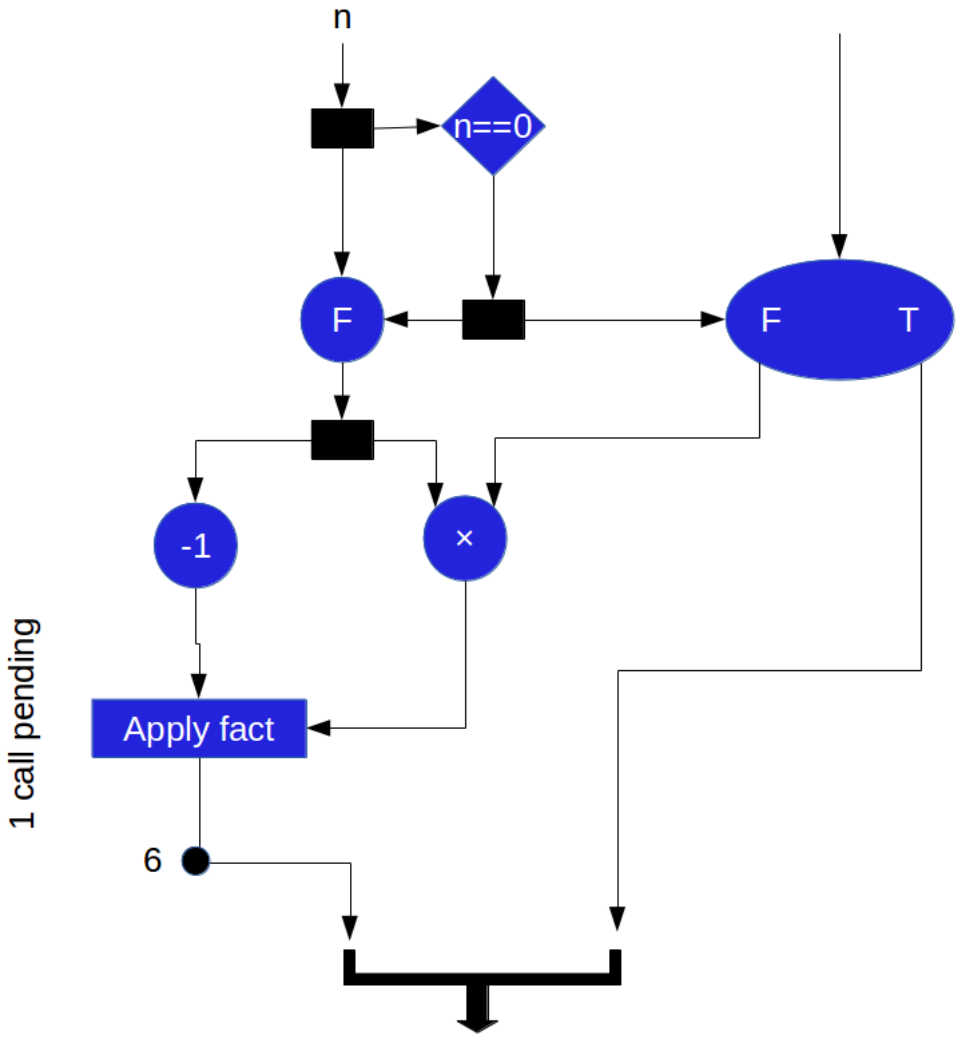
Factorial

Tail Recursive version – factorial(3)



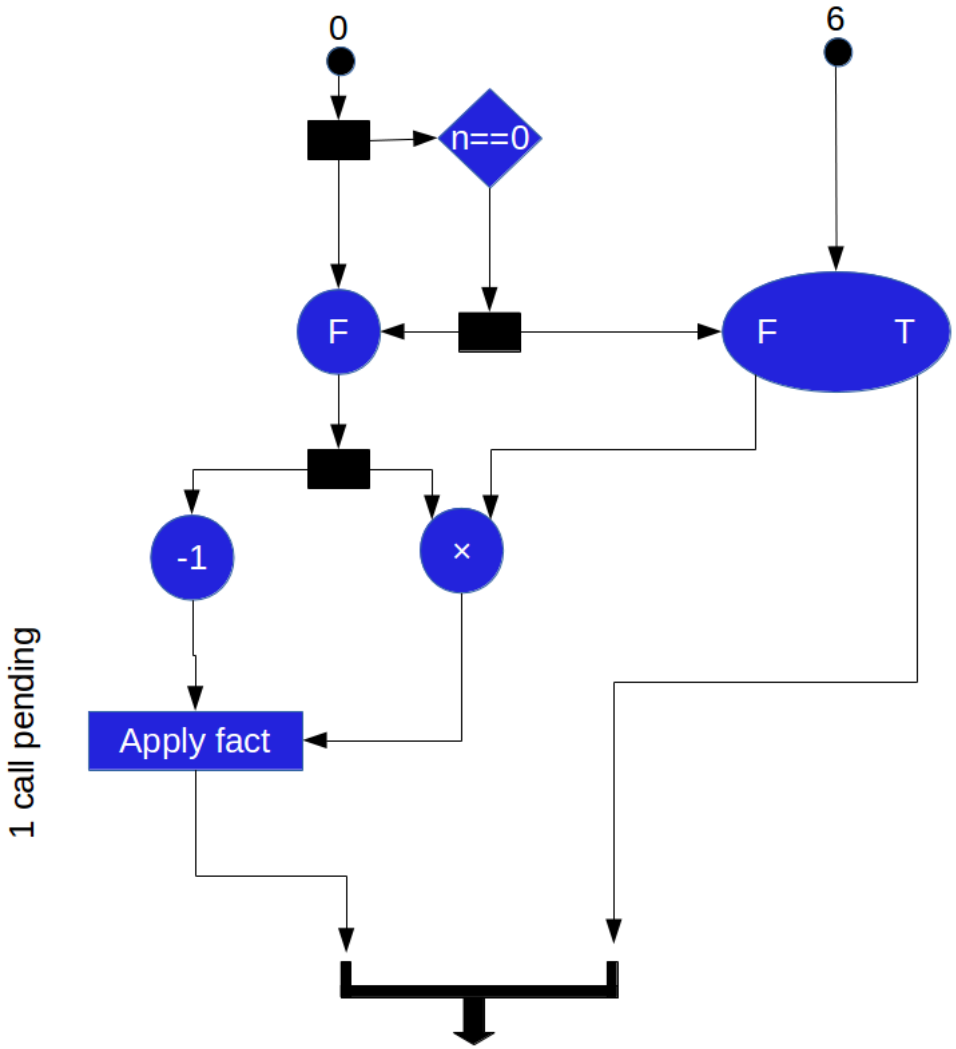
Factorial

Tail Recursive version – factorial(3)



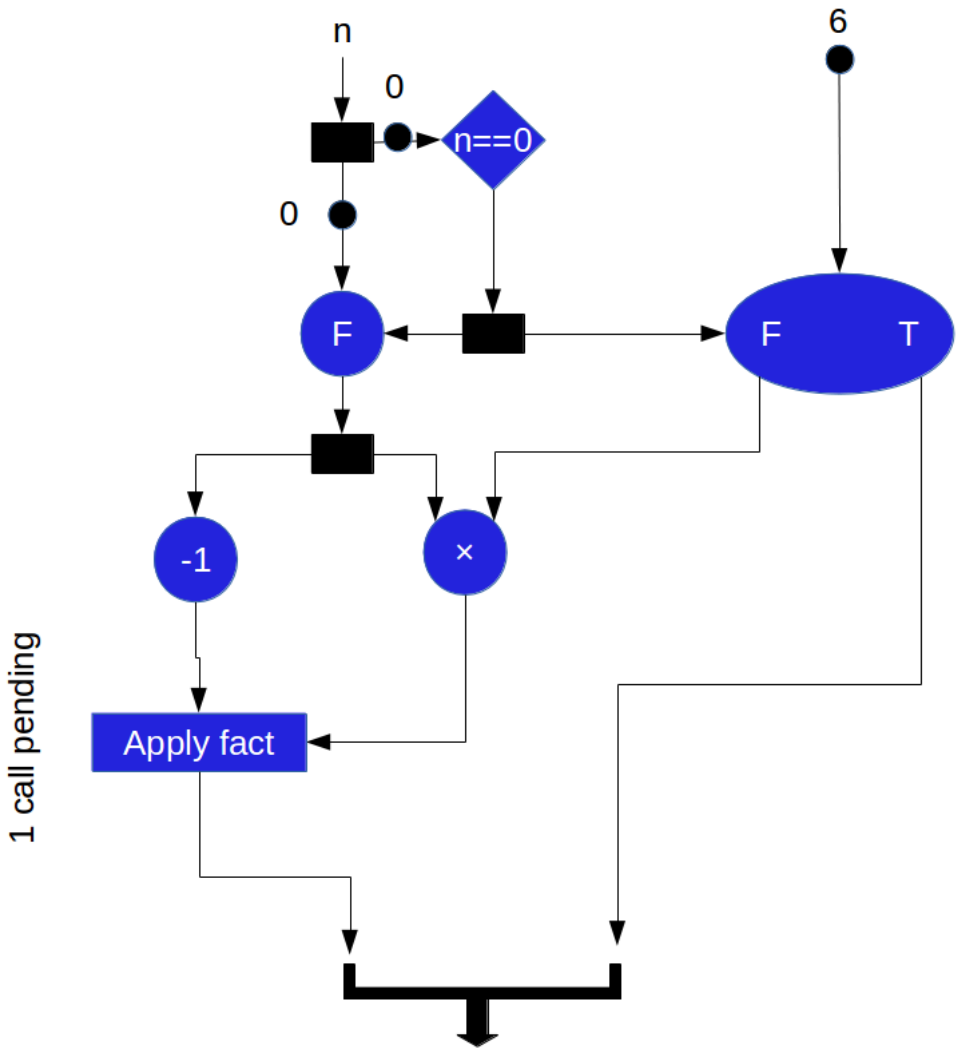
Factorial

Tail Recursive version – factorial(3)



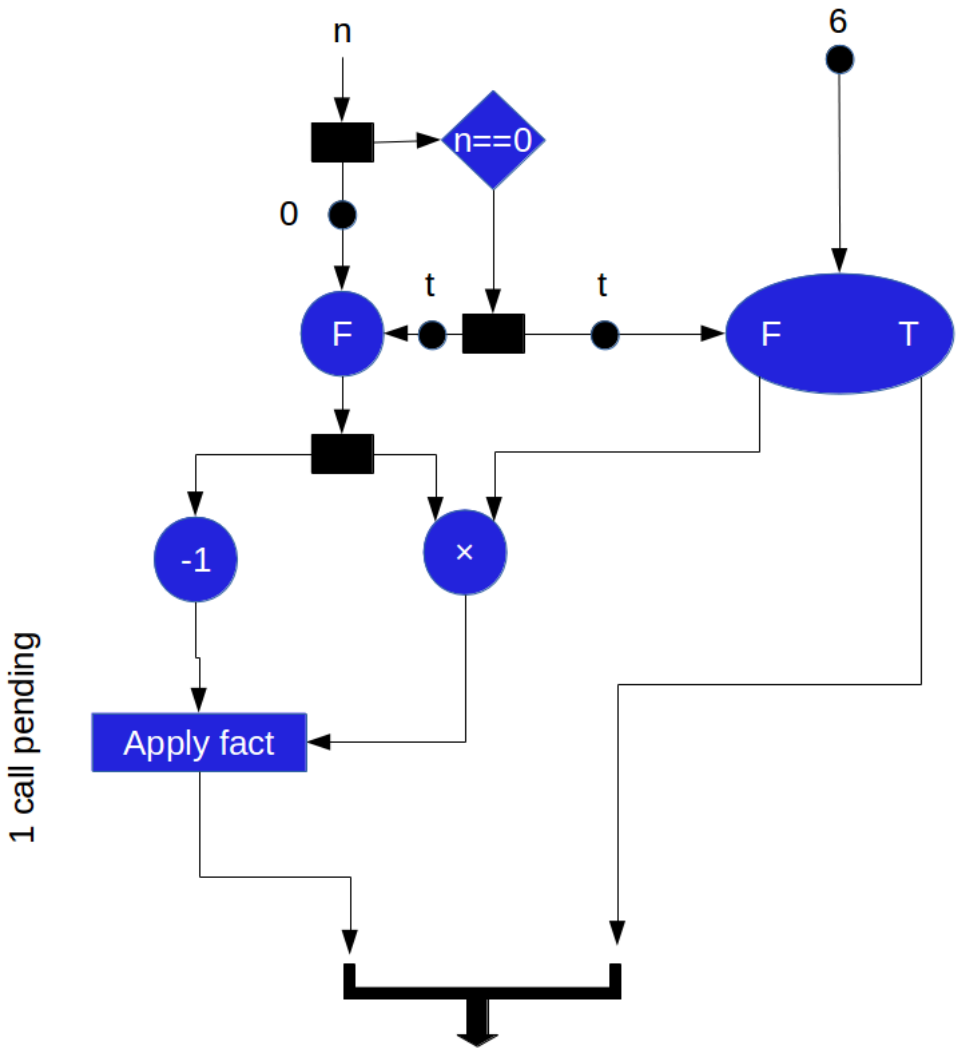
Factorial

Tail Recursive version – factorial(3)



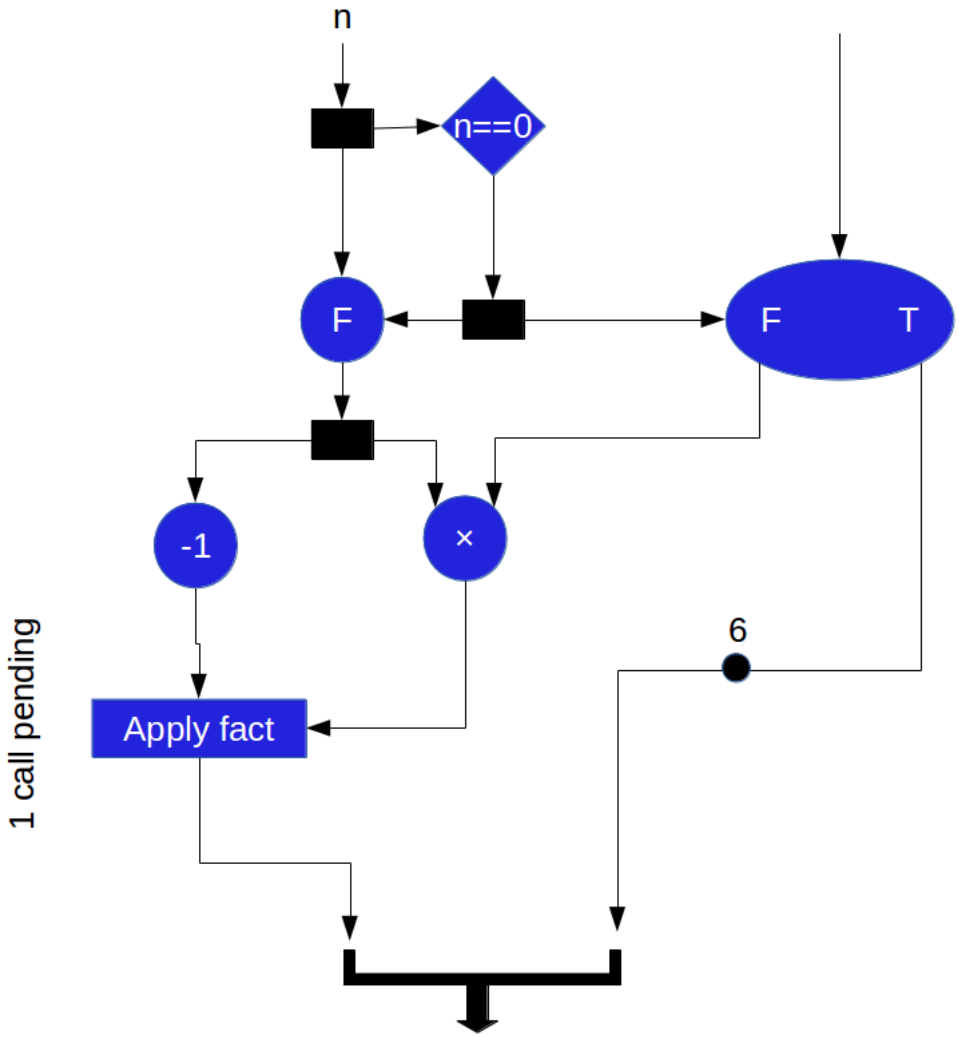
Factorial

Tail Recursive version – factorial(3)



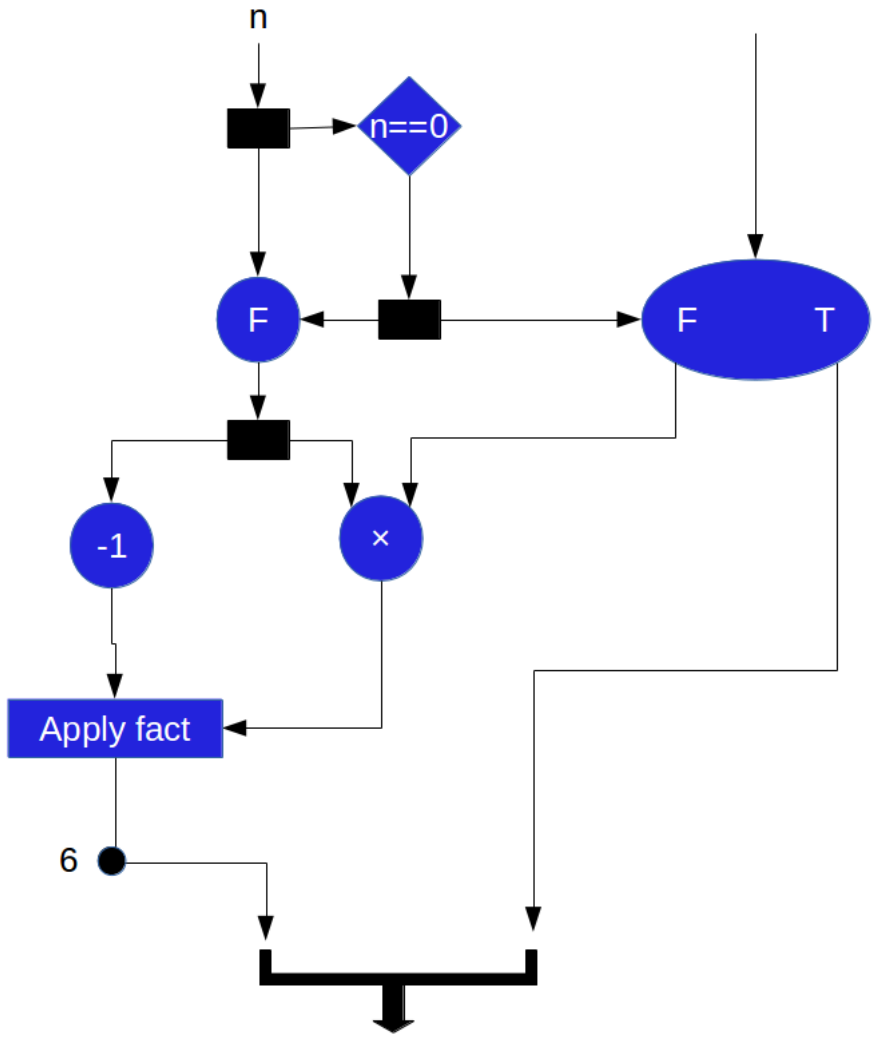
Factorial

Tail Recursive version – factorial(3)



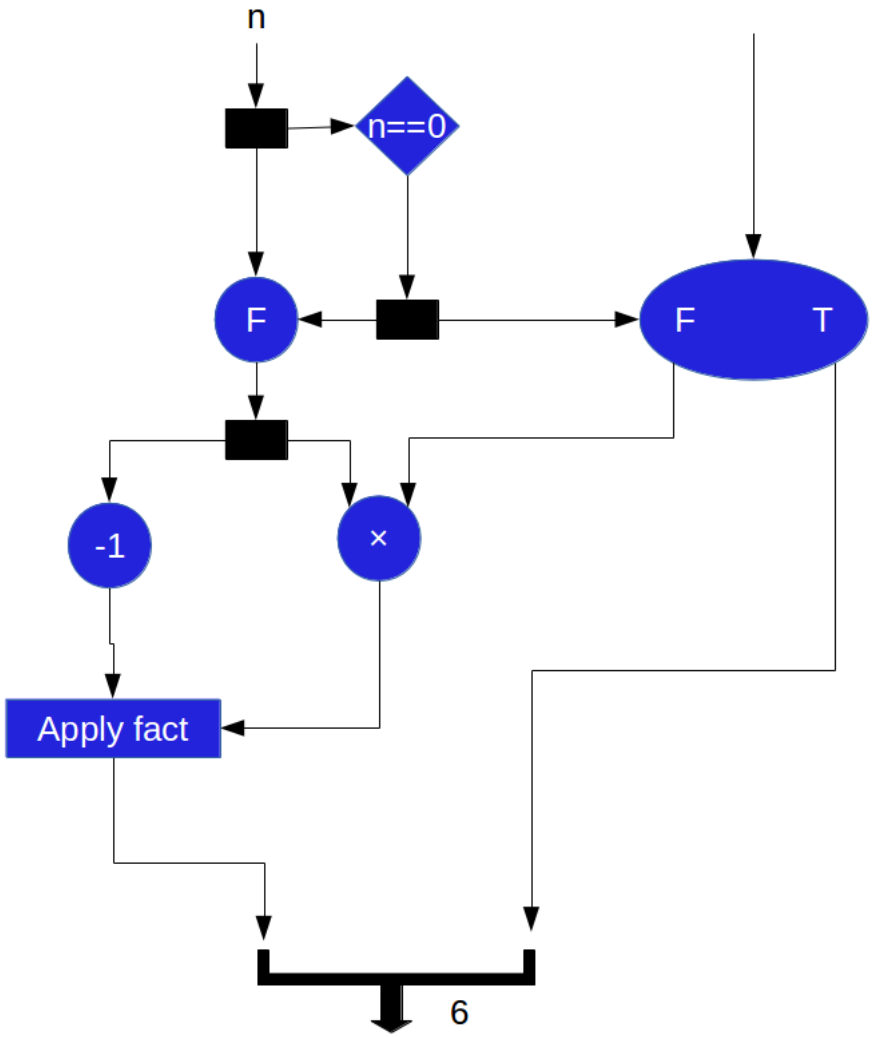
Factorial

Tail Recursive version – factorial(3)



Factorial

Tail Recursive version – factorial(3)



Outline

Static Dataflow

- Static dataflow allows for loops to be created
- Simple rules: One-token-per-arc
- But static dataflow has many limitations:
  - A static dataflow actor cannot *fire* until all its output arcs are empty.
  - Cannot handle procedure calls
  - Cannot handle tail recursions
  - Cannot handle arbitrary recursions

Recursive Program Graphs

- Recursive Program Graphs extend static dataflow relax its constraints
- Some features:
  - One-token-per-arc-per-invocation (also known as one-token-per-link-in-lifetime)
  - No deterministic merge needed
  - Graph must be acyclic (DAG, directed acyclic graph)
  - Procedure calls are allowed through the use of the apply special actor
  - Tail-recursive calls are allowed
    - In fact, iterative behaviors are expressed through tail recursion
    - Recursion is expressed by runtime copying
    - Use of tags
  - No token matching is needed
- But: still not completely general-purpose:
  - Cannot handle arbitrary recursions
  - Cannot handle mutual function function calls (*e.g.*, function g calls function `f`, and vice-versa)

Dynamic Dataflow Model of Computation

Dynamic Dataflow solves the remaining problems recursive program graphs couldn't:

- There is no limitation on the number of tokens per arc anymore
- Tokens are tagged with a "color"
  - This implies new firing rules – we'll examine them later
- Resulted in the MIT Tagged Token dataflow computer

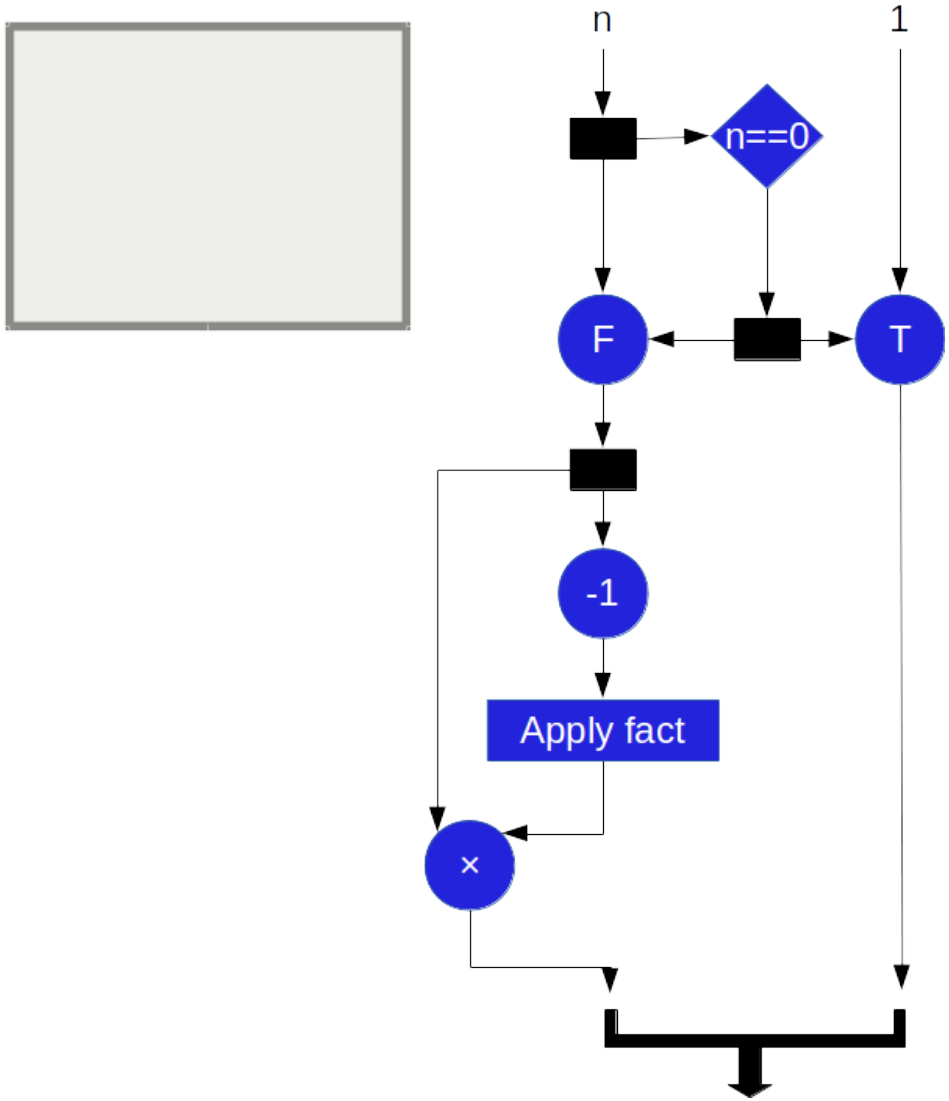
Token $= \Rightarrow$ Tag + Value: $[v, < u, s >, d]$

- **v** – Value
- **u** – Activation instance
- **s** – Destination actor
- **d** – Operand slot

- Loops & Function Calls
  - Should be executed in parallel as instances of the same graph
- Arc $\Rightarrow$ A container with different tokens that have different tags ("colors")
  - Stored in a "token store"
- A node can fire as soon as all tokens with identical tags ("colors") are presented in its input arcs

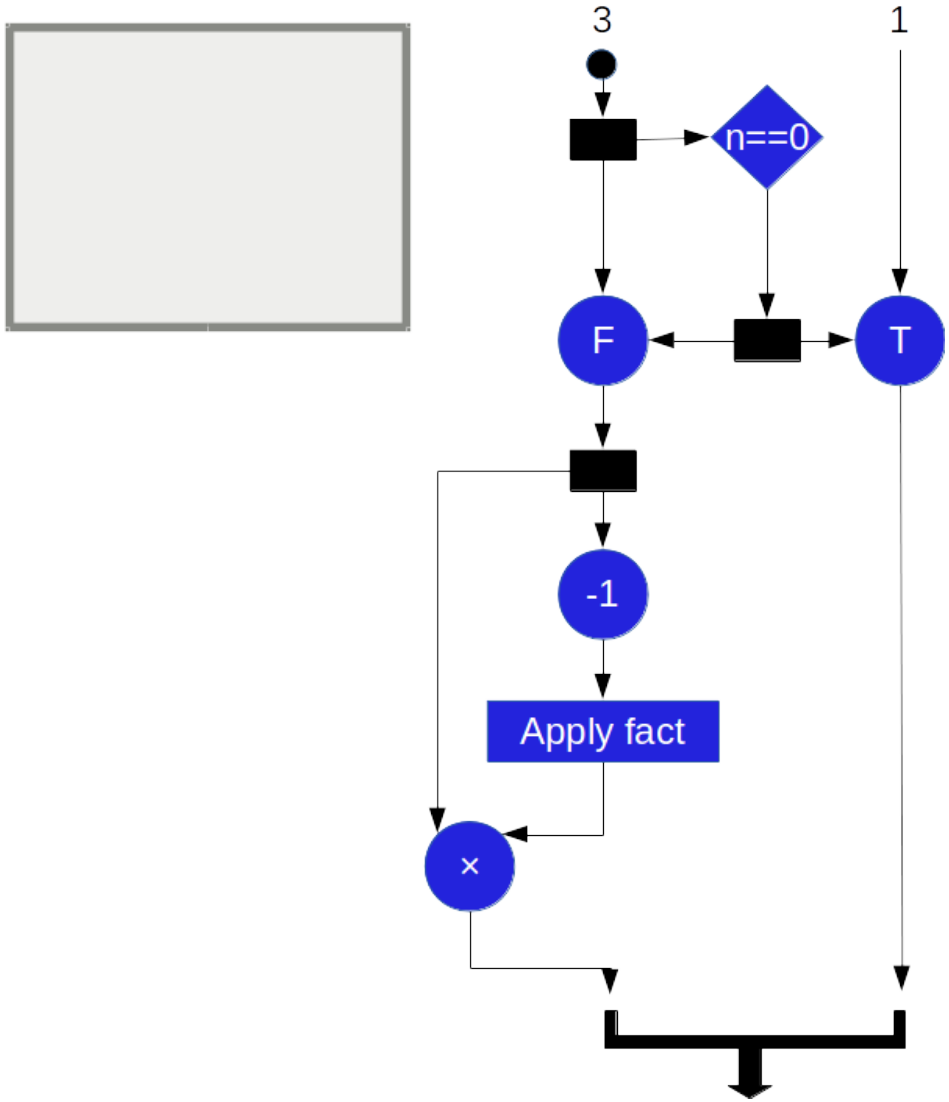
Revisiting Ordinary Recursions

Factorial



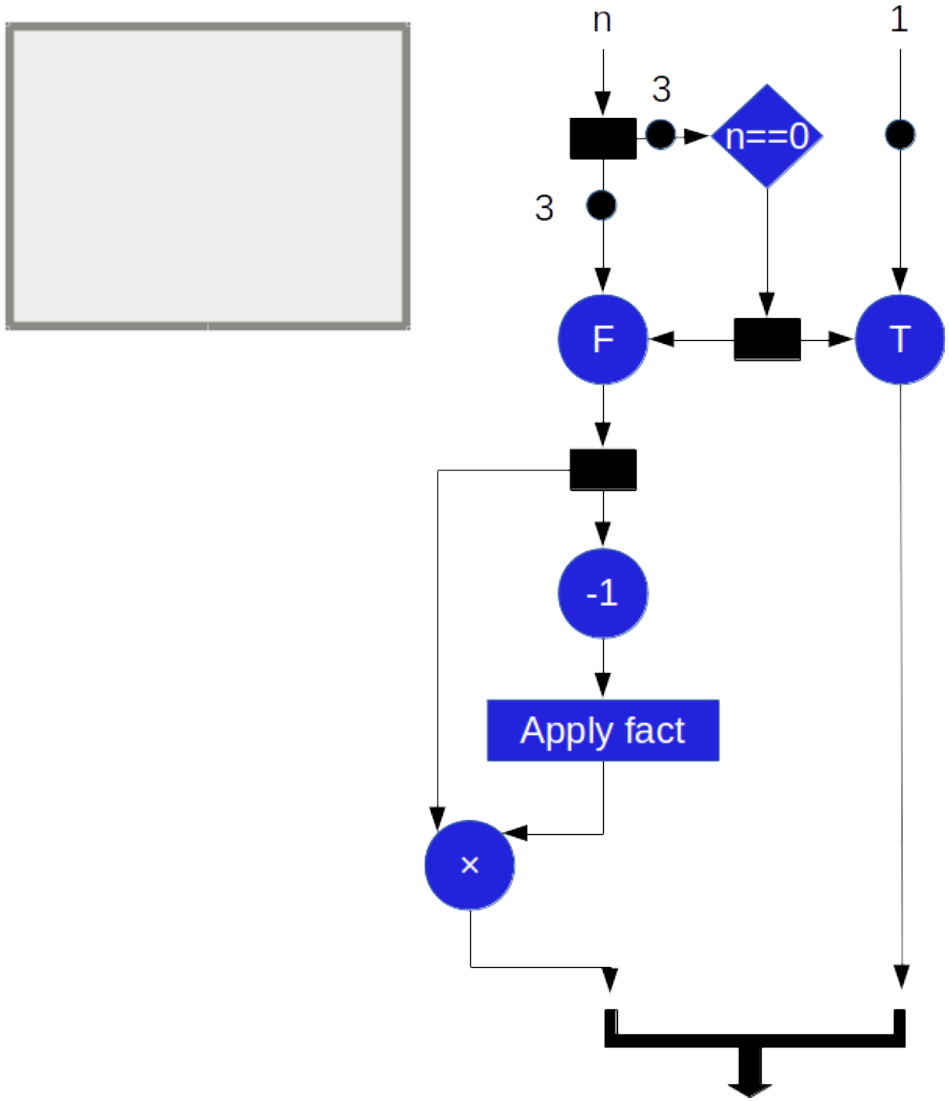
Revisiting Ordinary Recursions

Factorial



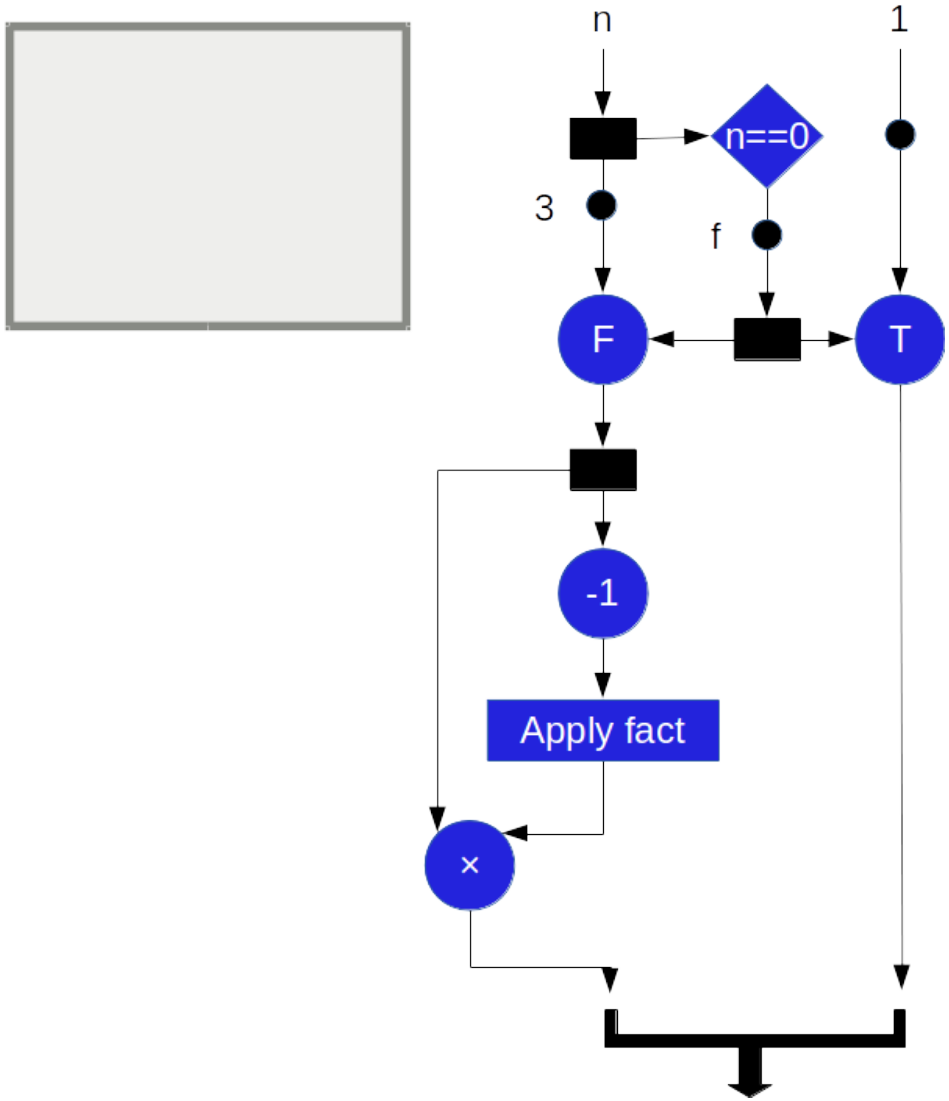
Revisiting Ordinary Recursions

Factorial



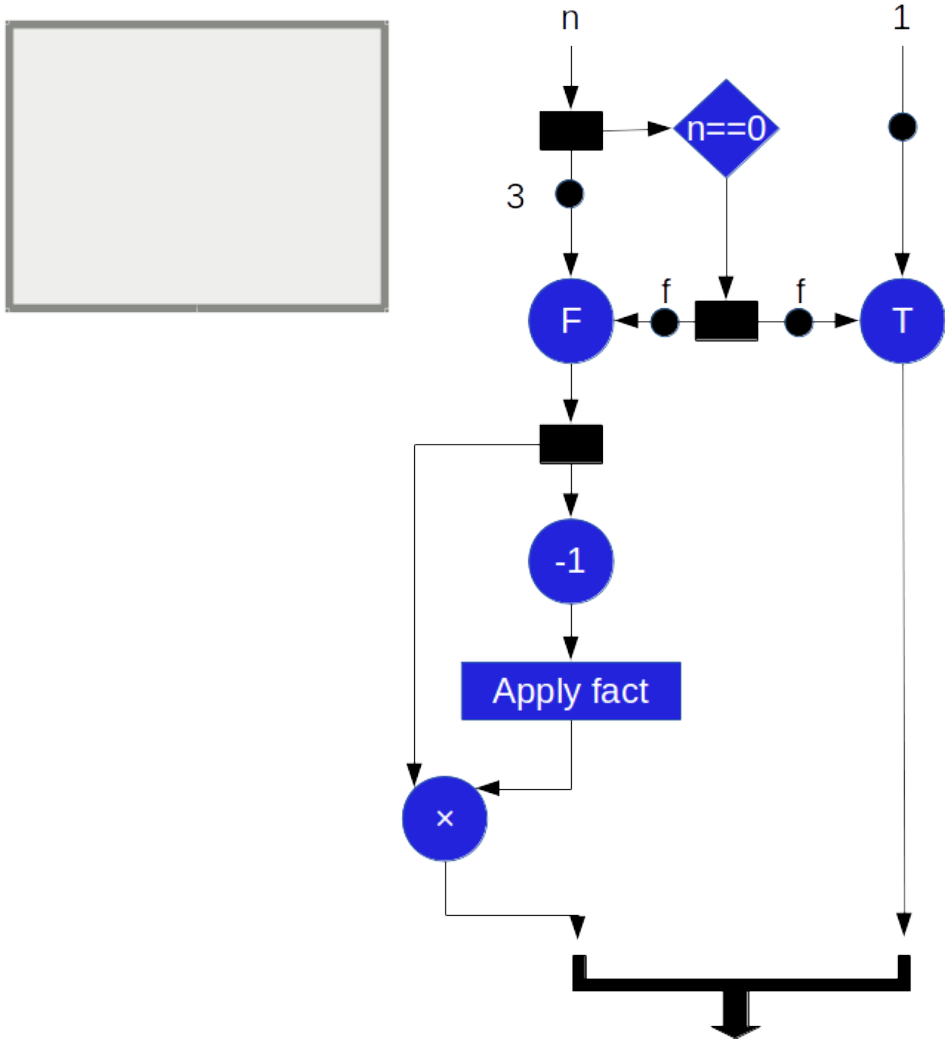
Revisiting Ordinary Recursions

Factorial



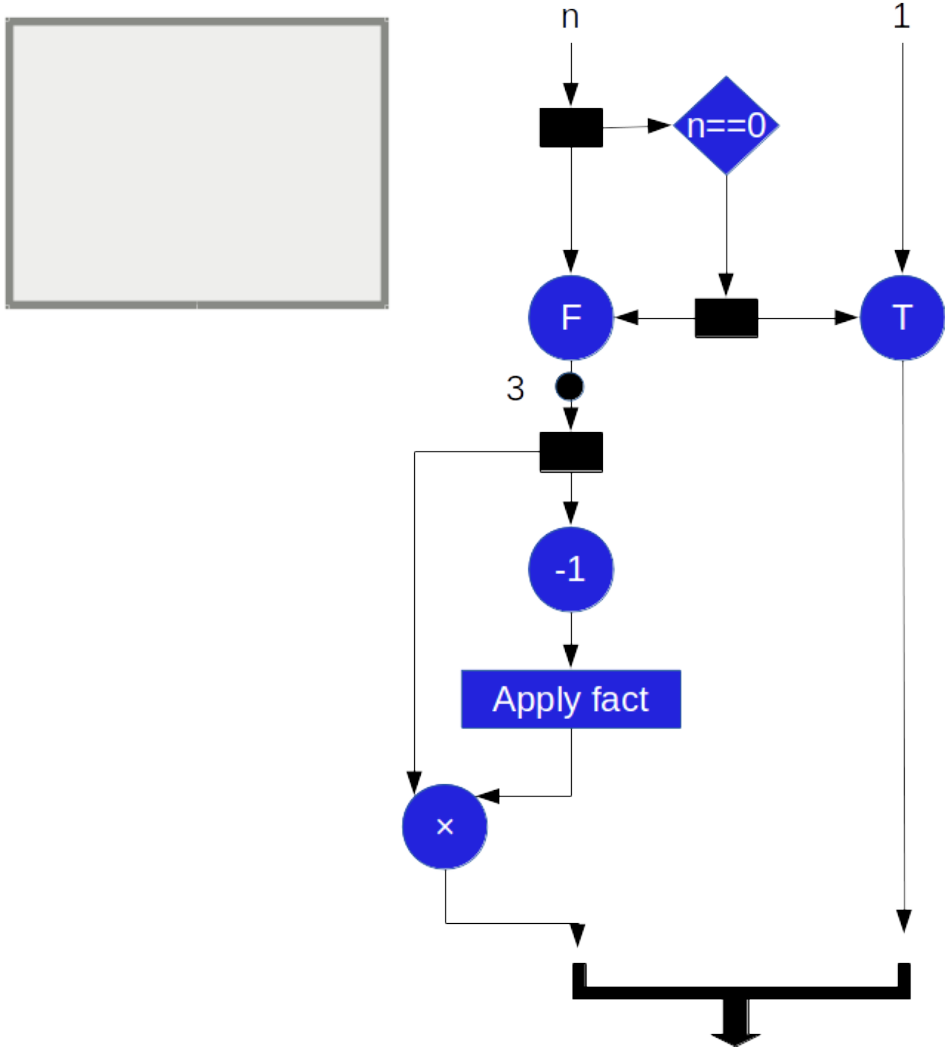
Revisiting Ordinary Recursions

Factorial



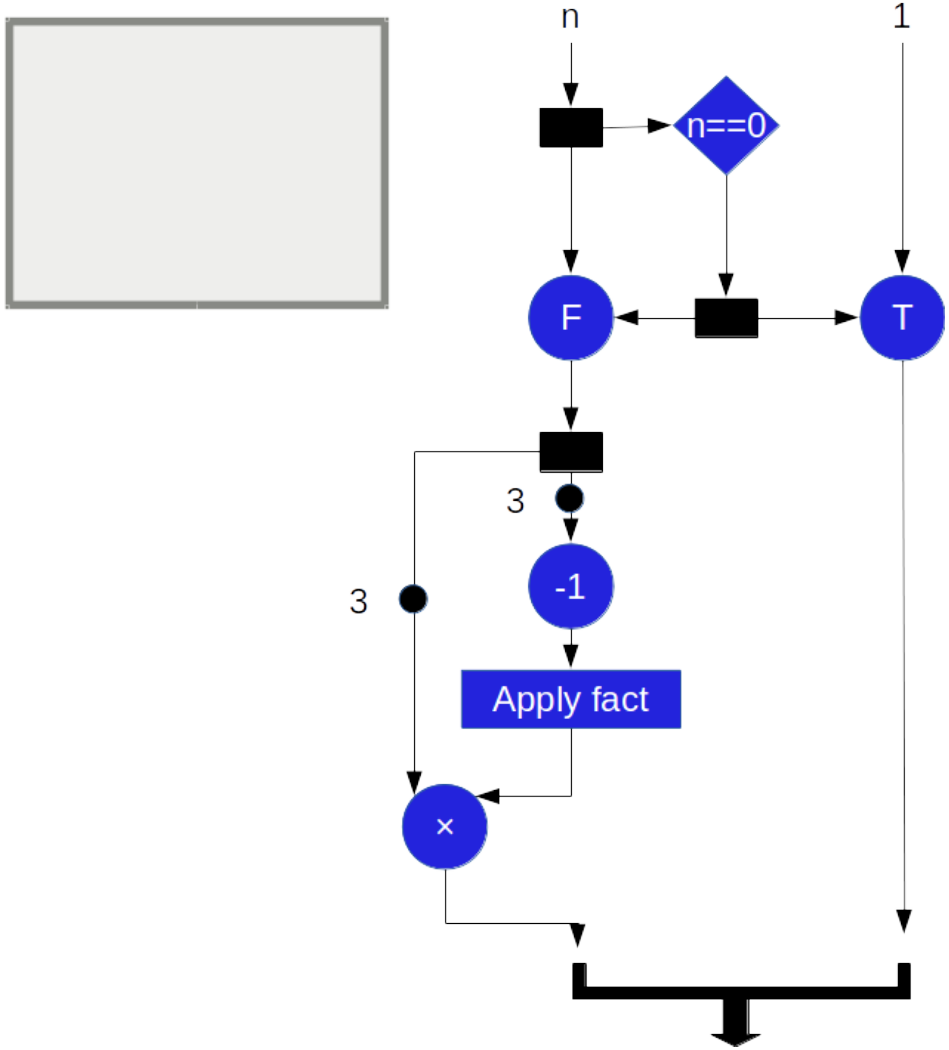
Revisiting Ordinary Recursions

Factorial



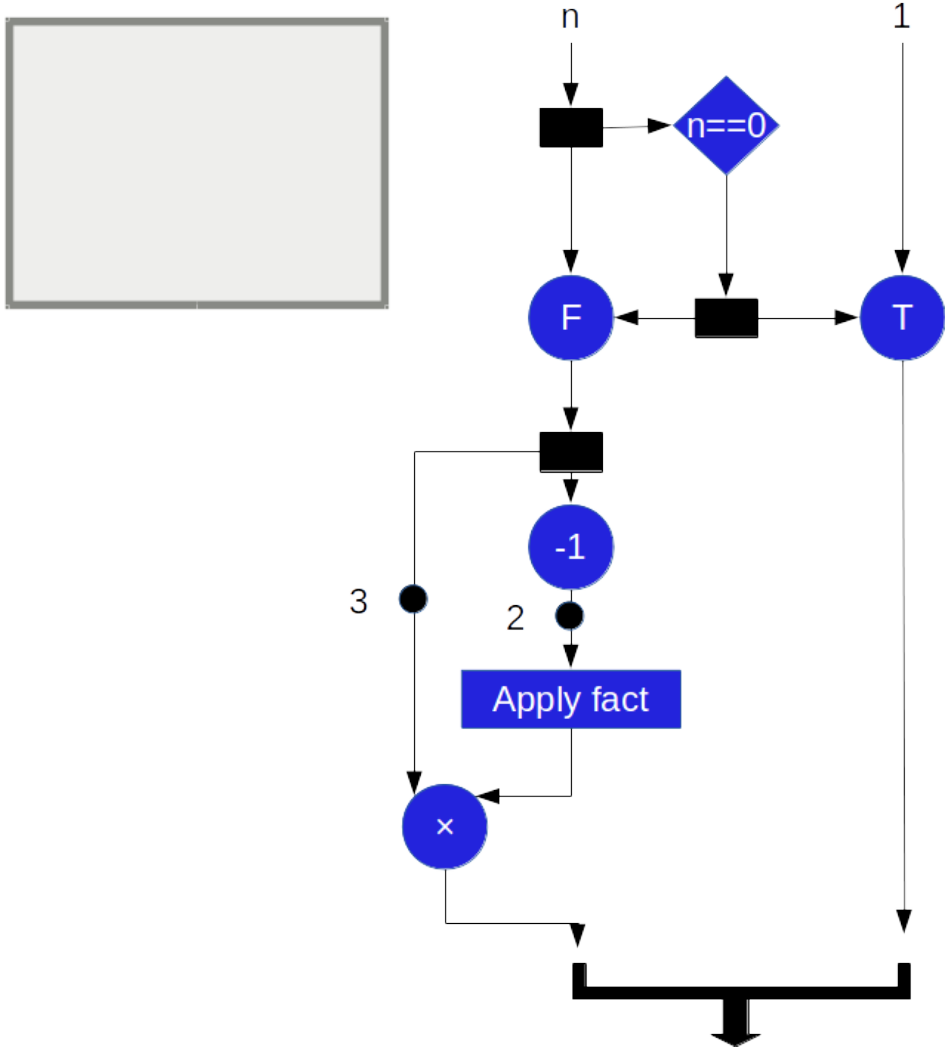
Revisiting Ordinary Recursions

Factorial



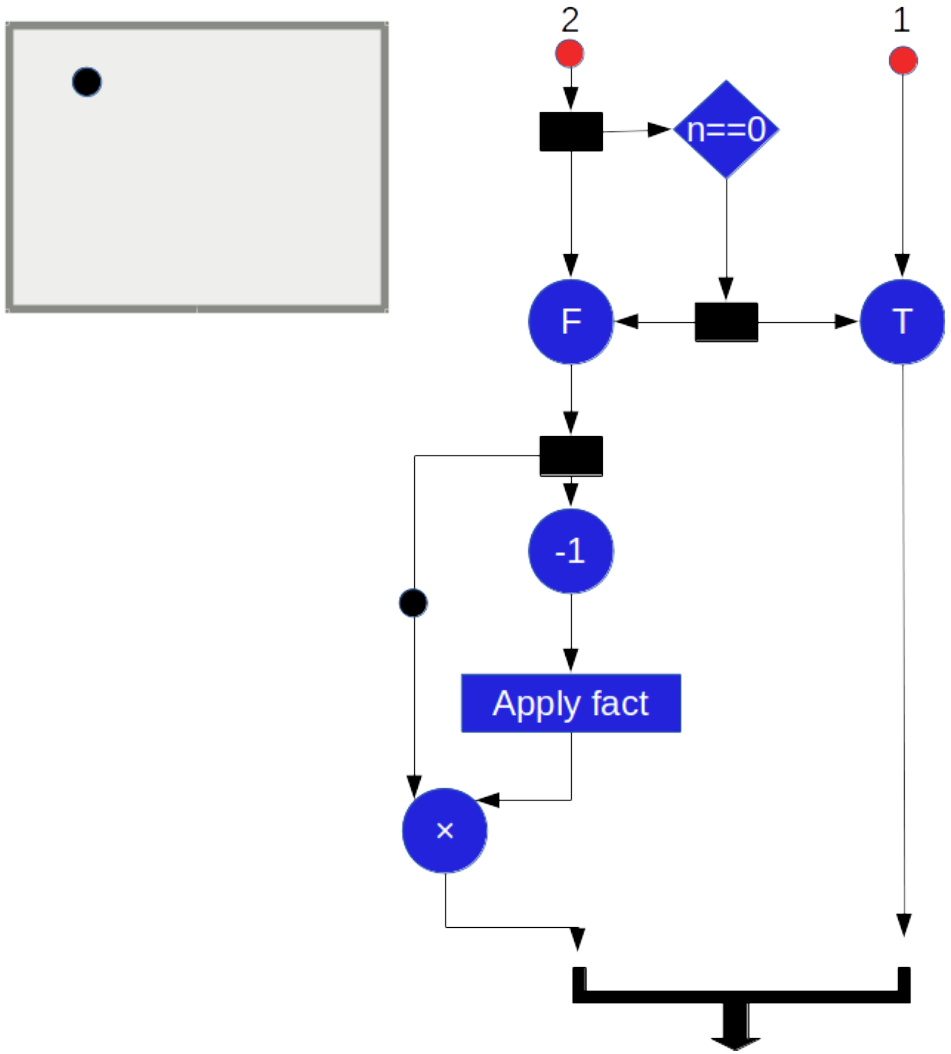
Revisiting Ordinary Recursions

Factorial



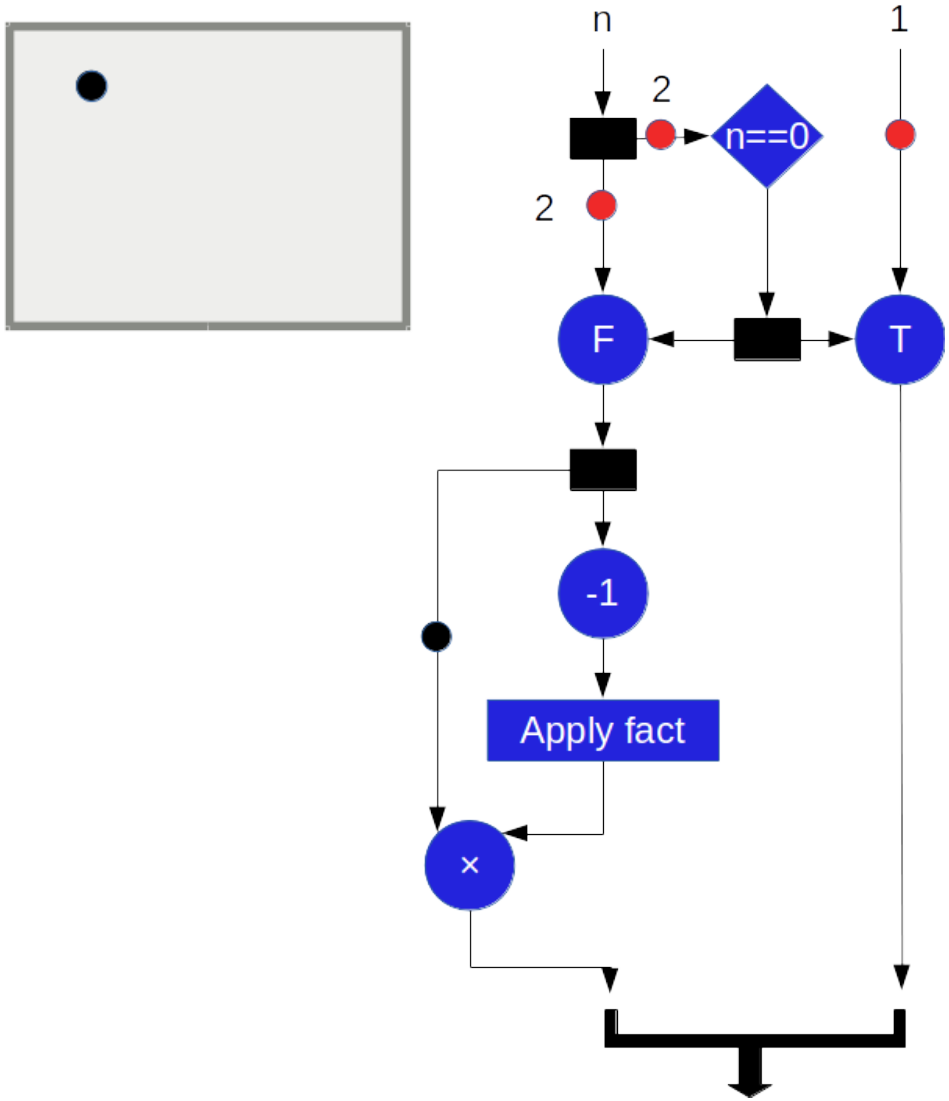
Revisiting Ordinary Recursions

Factorial



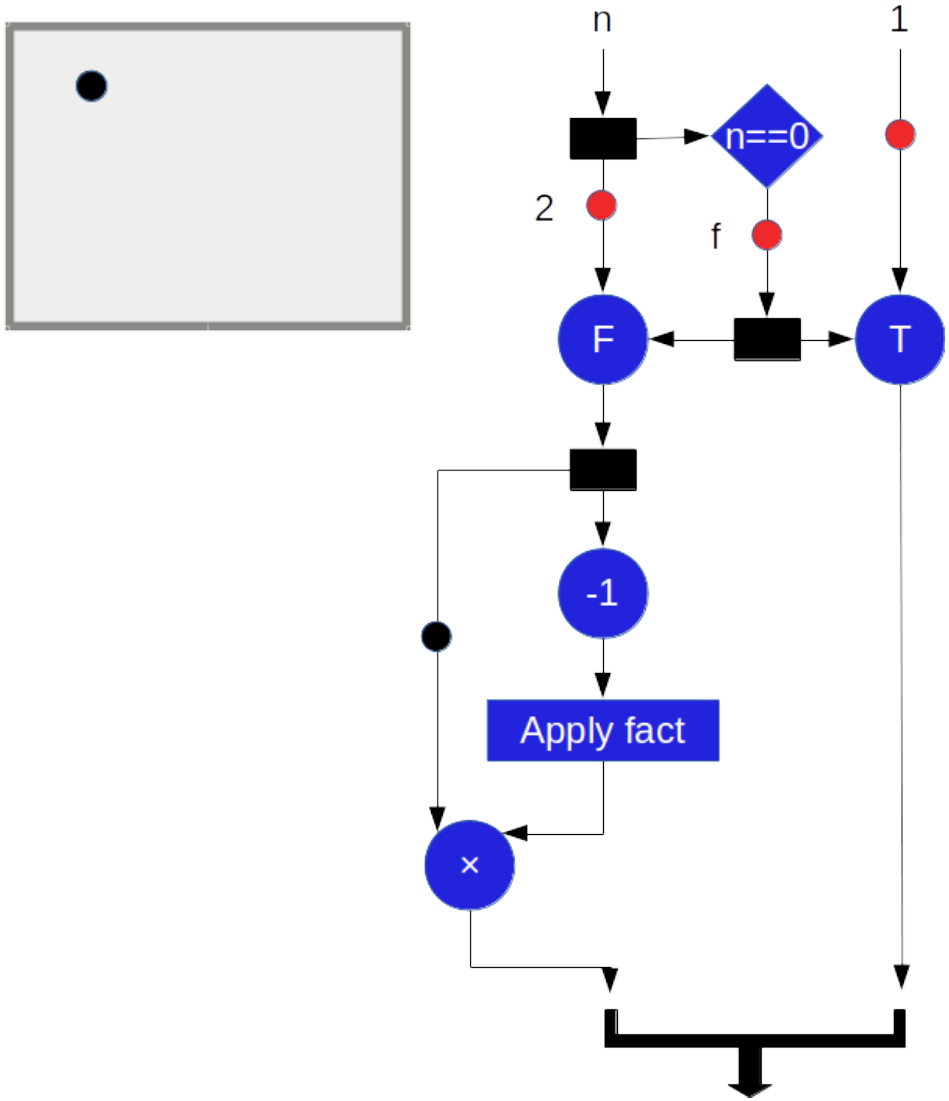
Revisiting Ordinary Recursions

Factorial



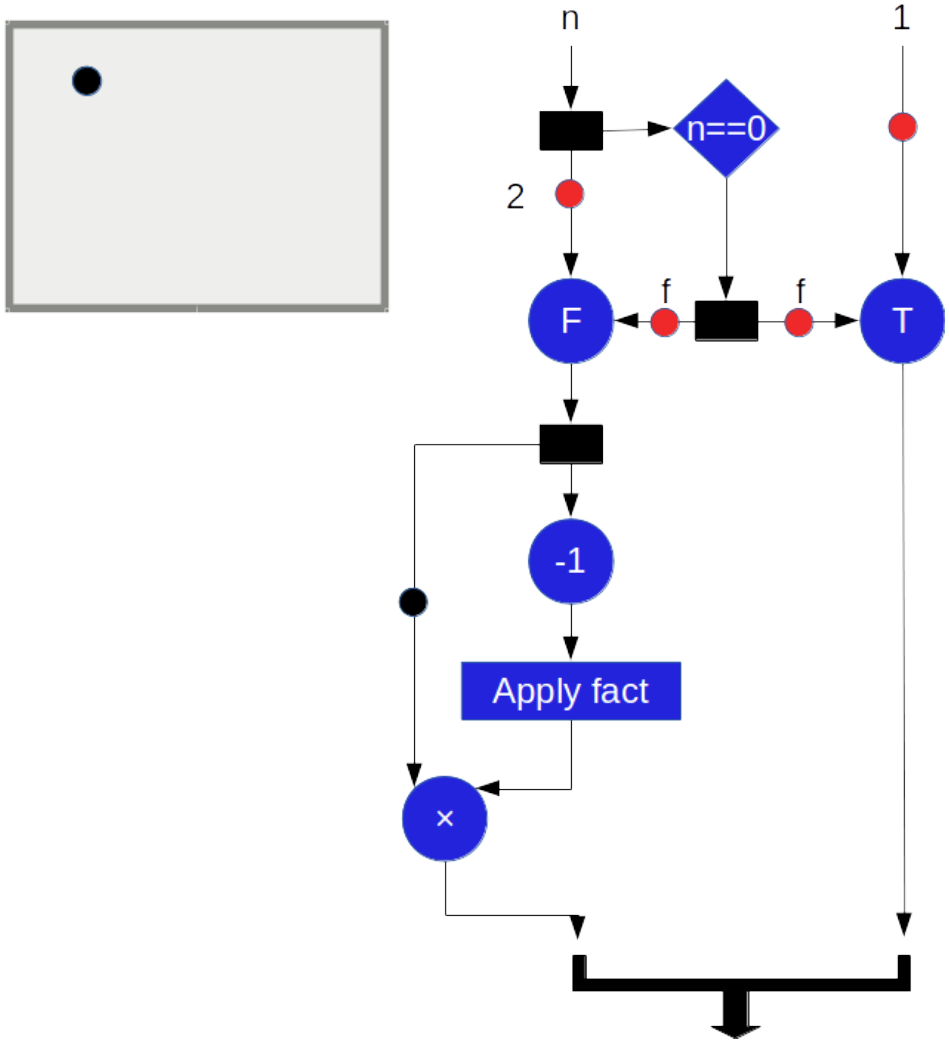
Revisiting Ordinary Recursions

Factorial



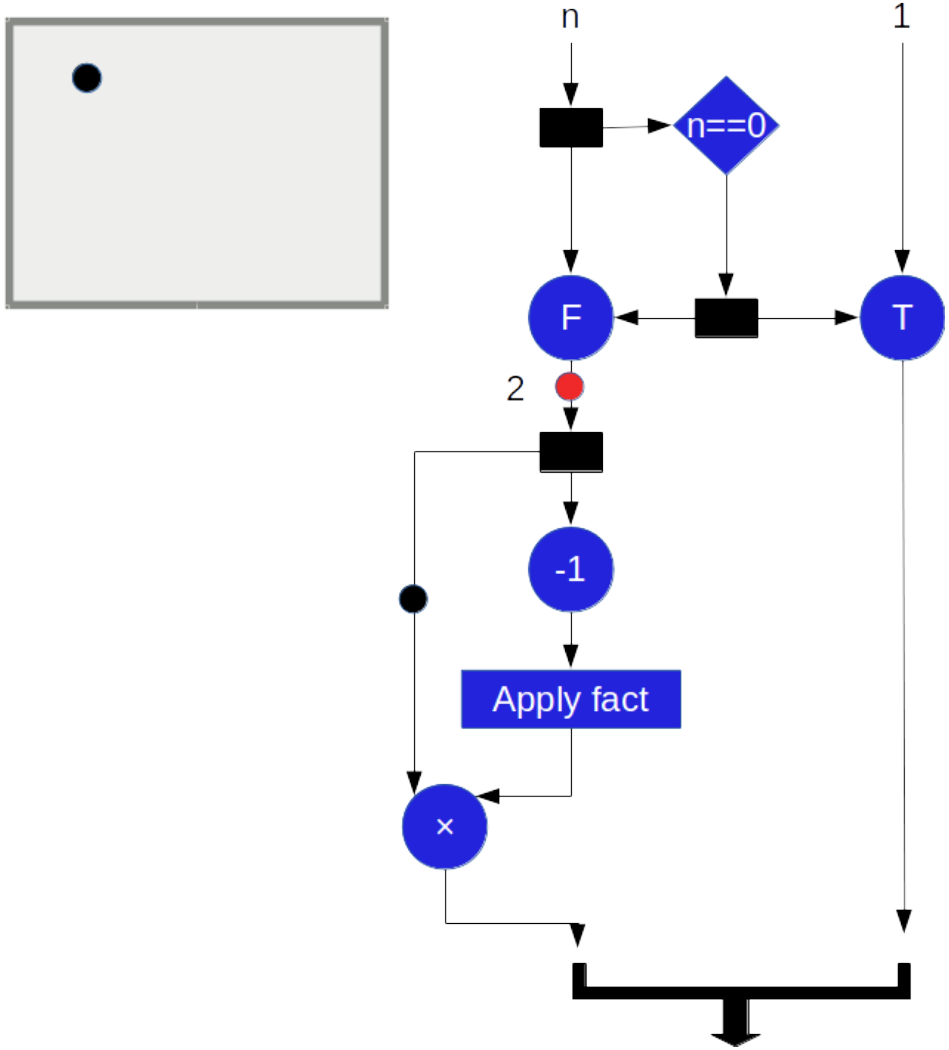
Revisiting Ordinary Recursions

Factorial



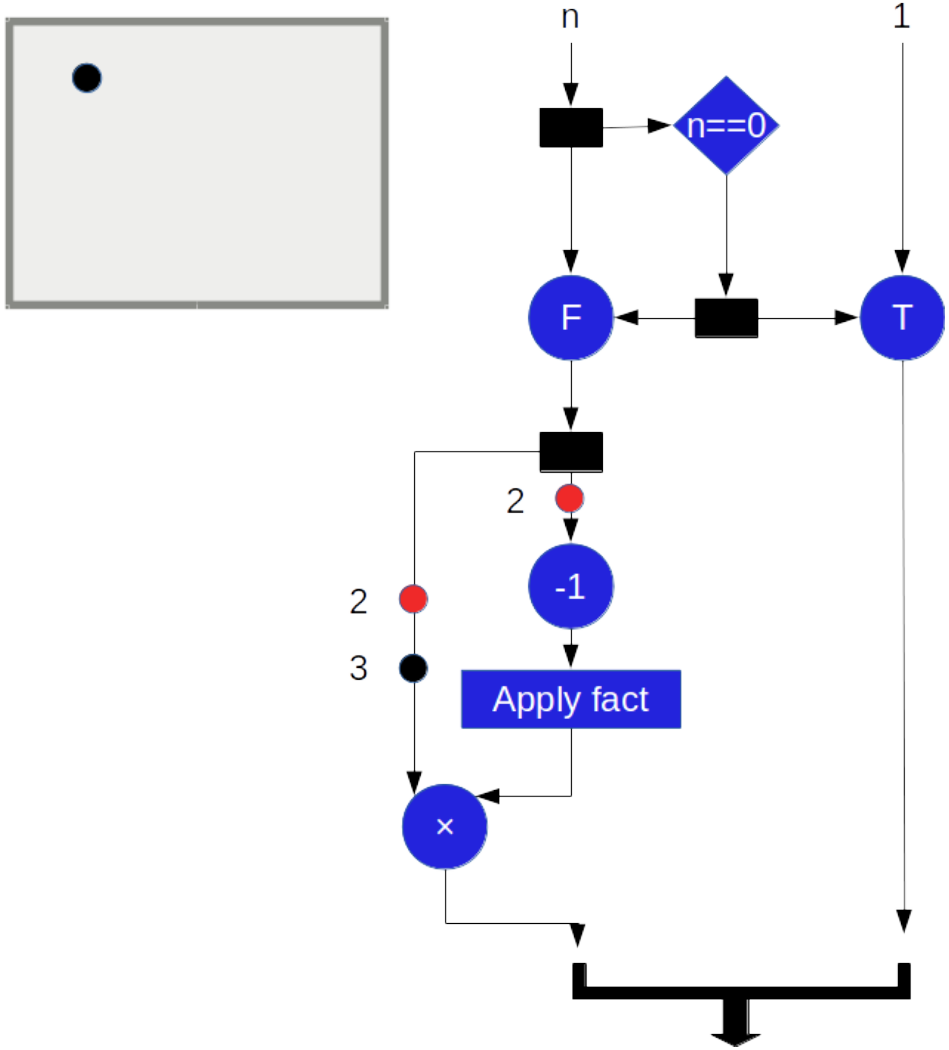
Revisiting Ordinary Recursions

Factorial



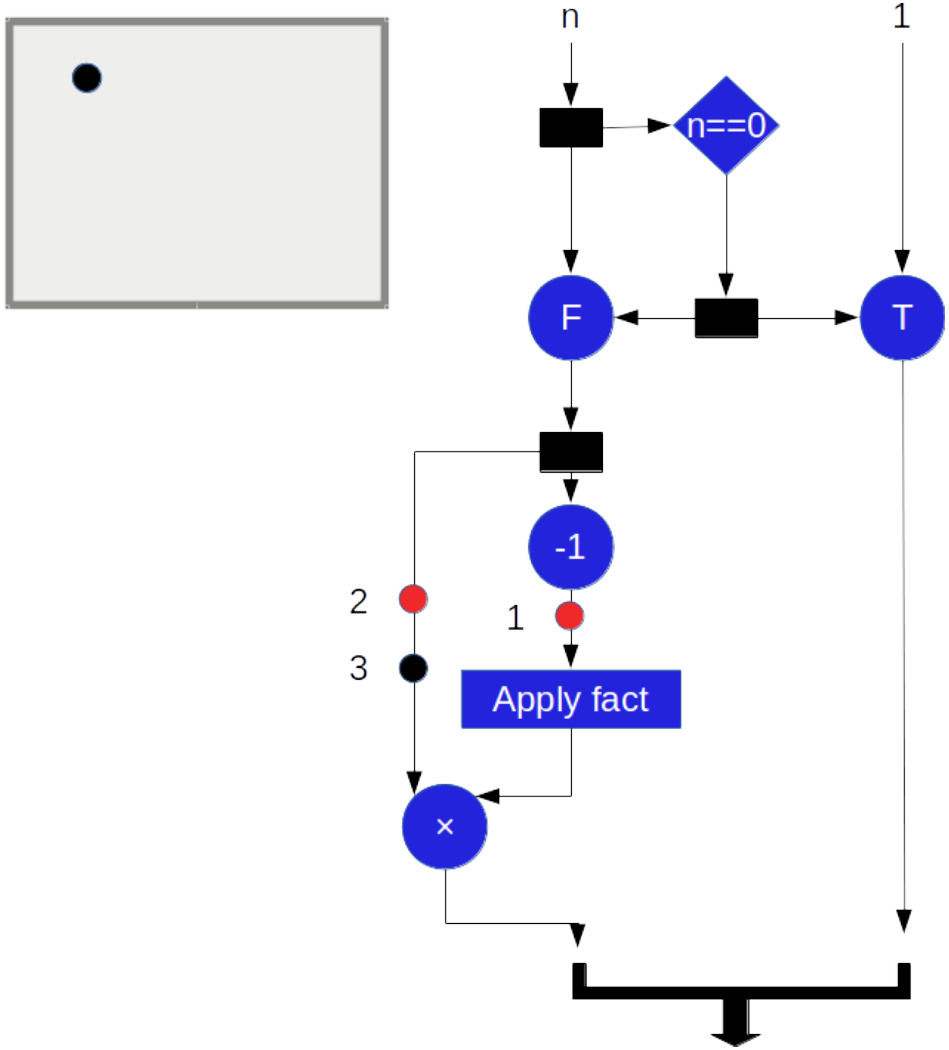
Revisiting Ordinary Recursions

Factorial



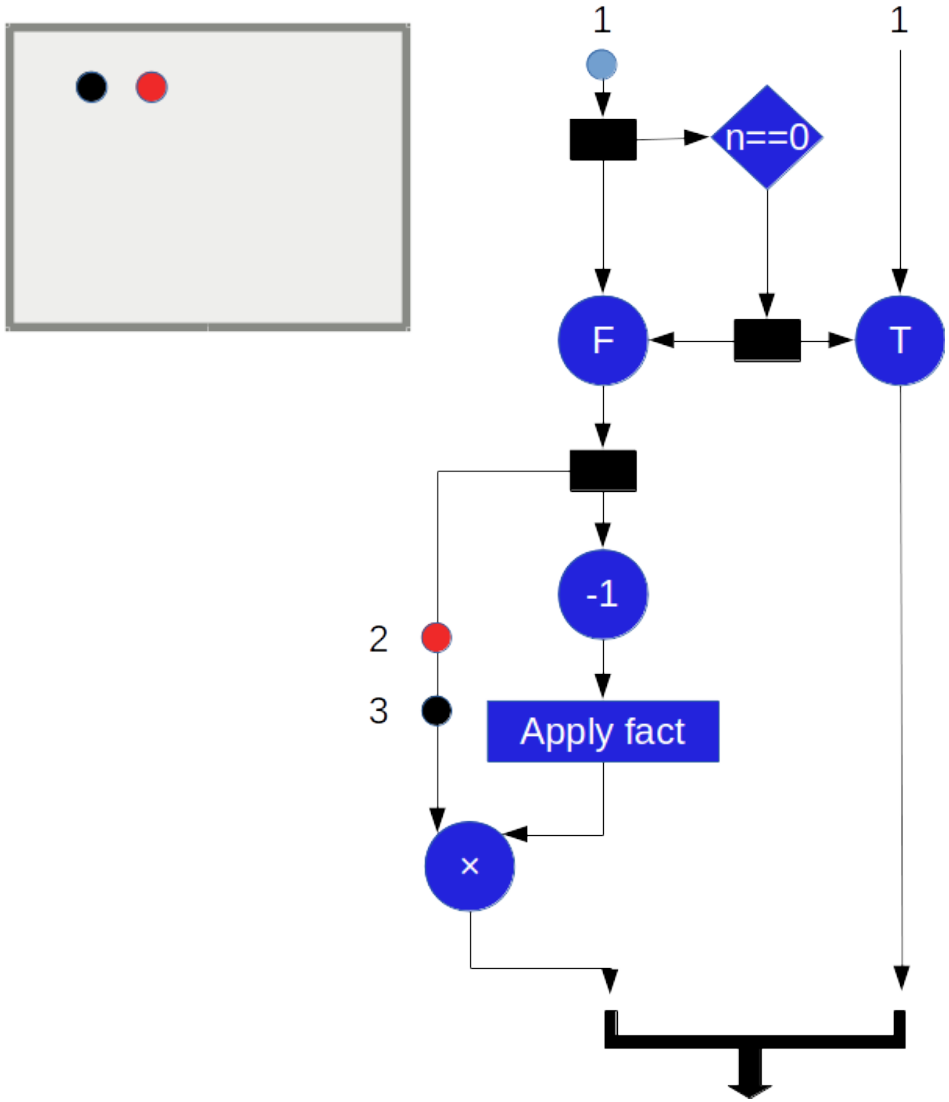
Revisiting Ordinary Recursions

Factorial



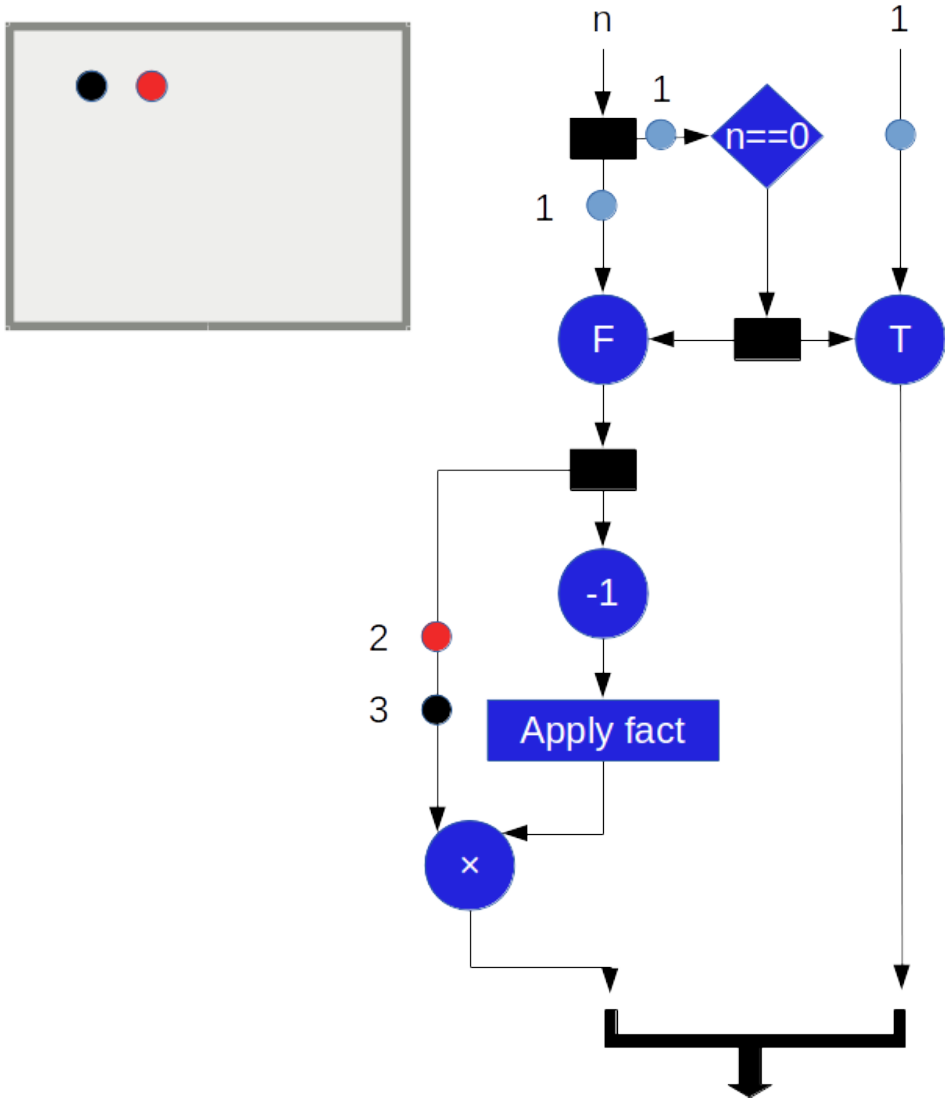
Revisiting Ordinary Recursions

Factorial



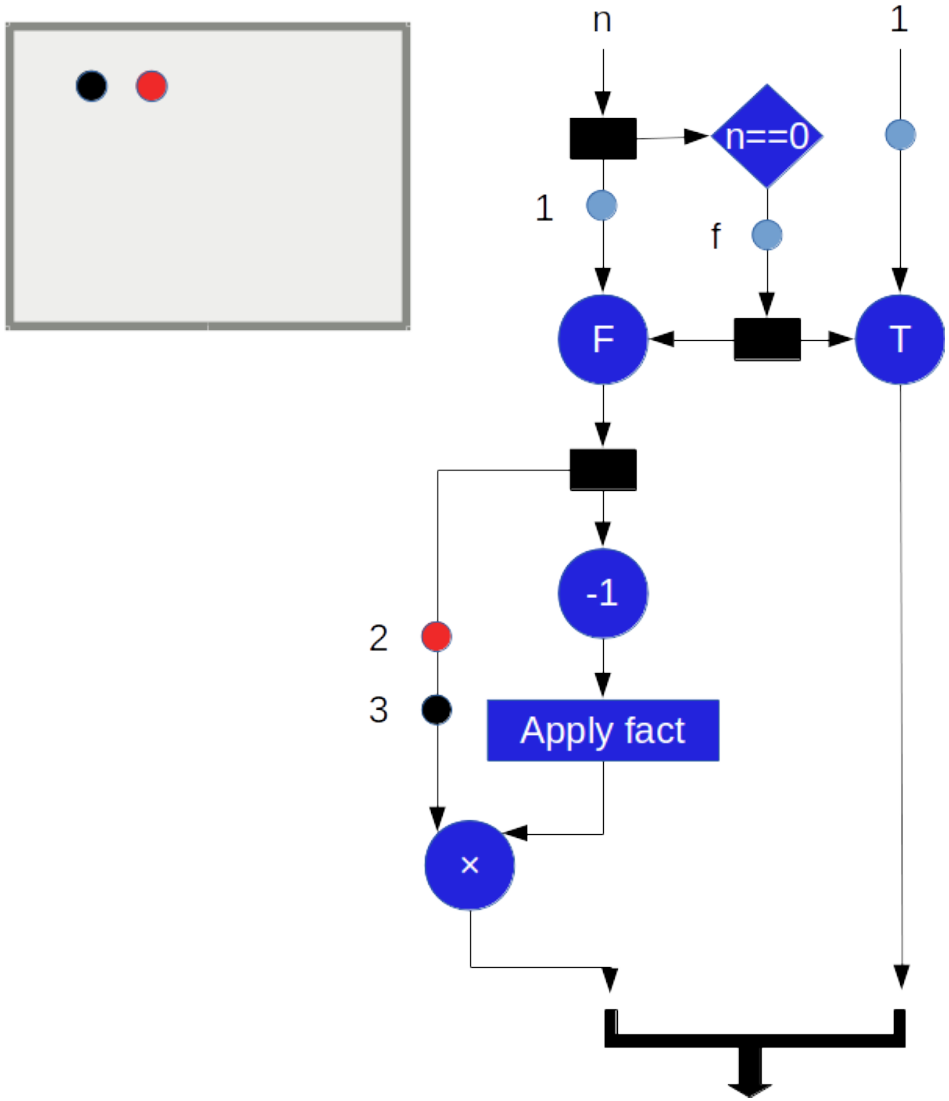
Revisiting Ordinary Recursions

Factorial



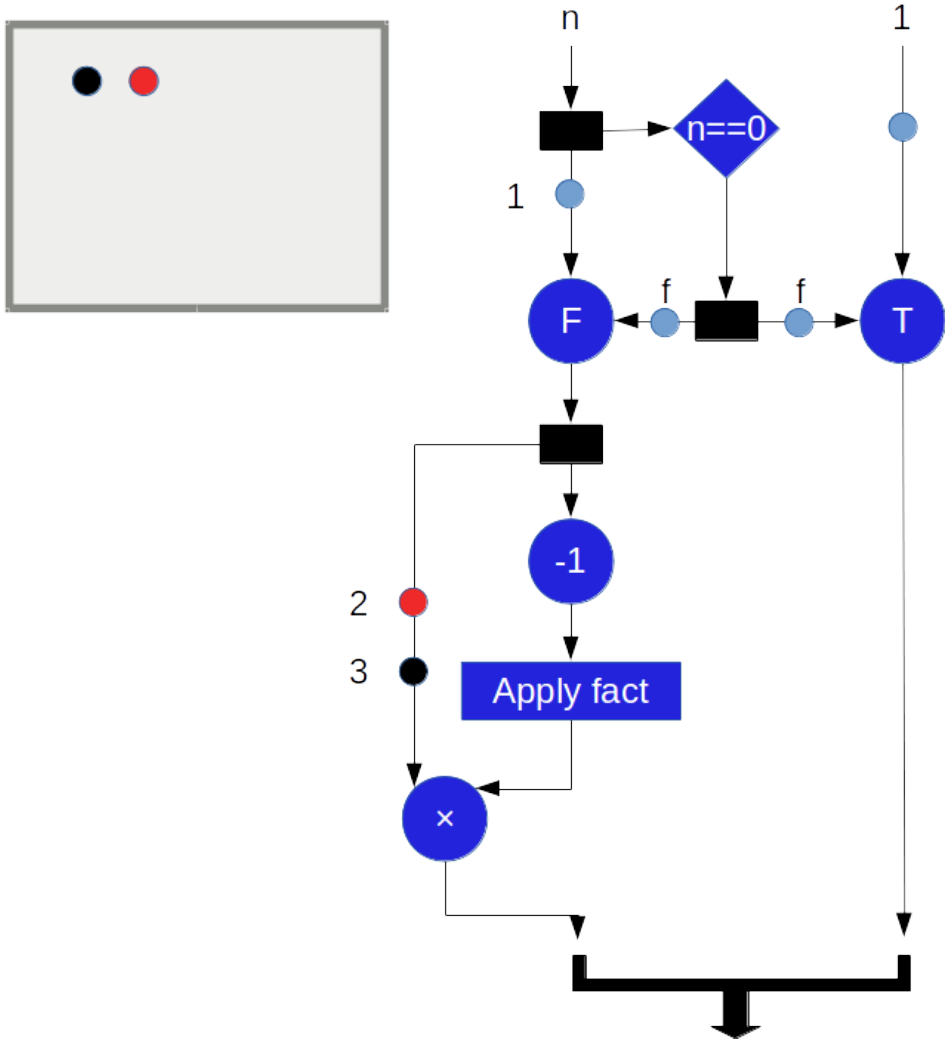
Revisiting Ordinary Recursions

Factorial



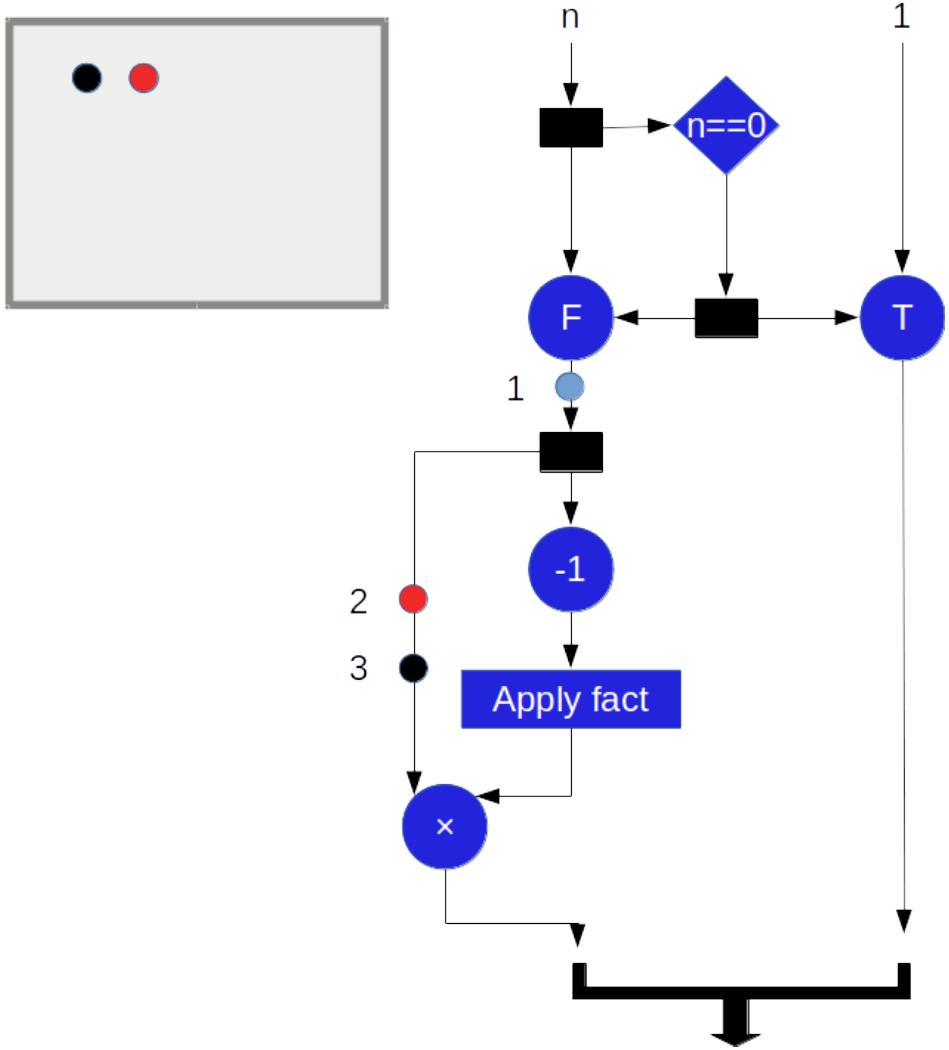
Revisiting Ordinary Recursions

Factorial



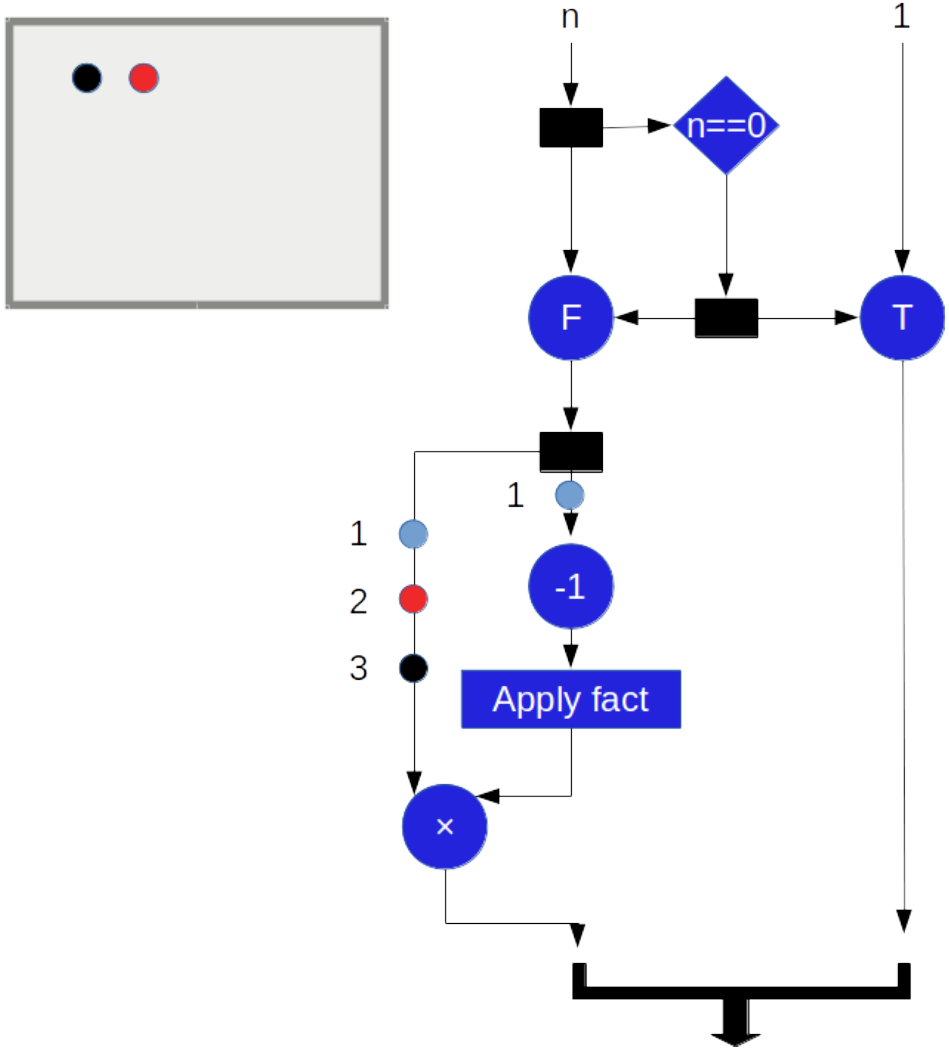
Revisiting Ordinary Recursions

Factorial



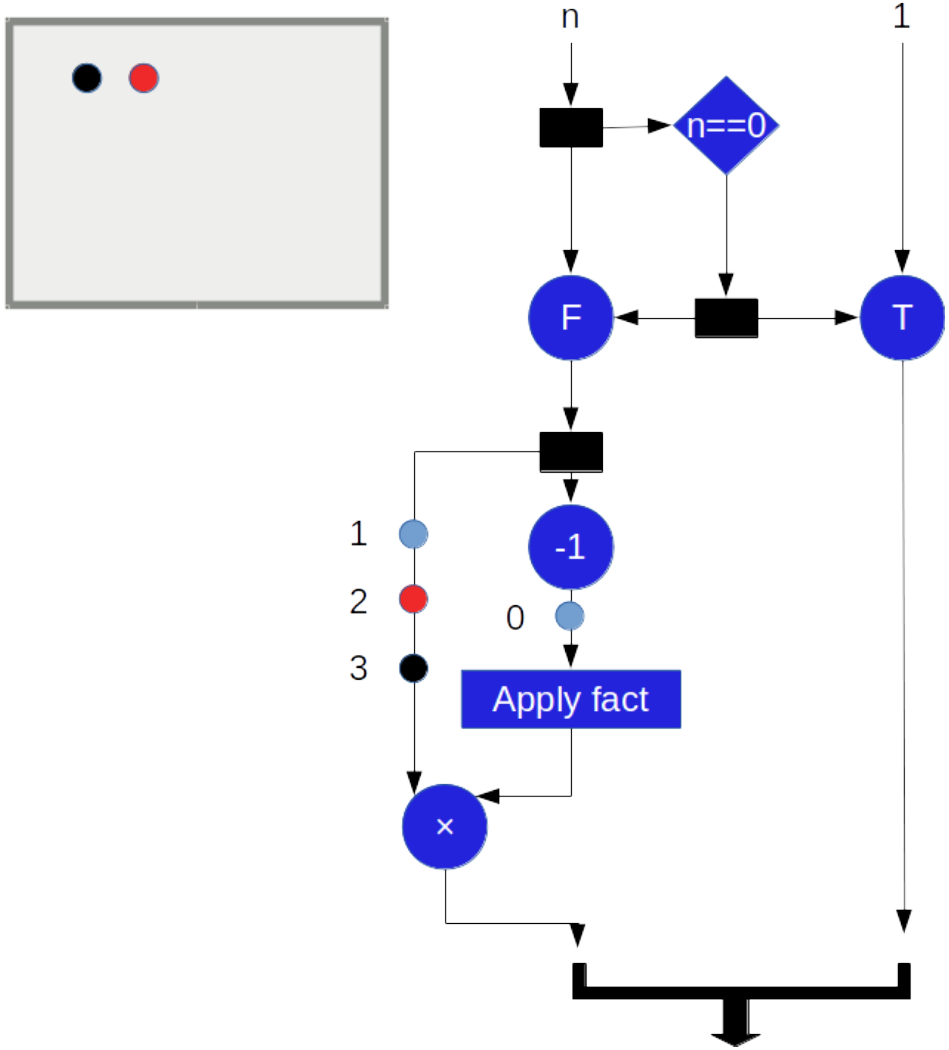
Revisiting Ordinary Recursions

Factorial



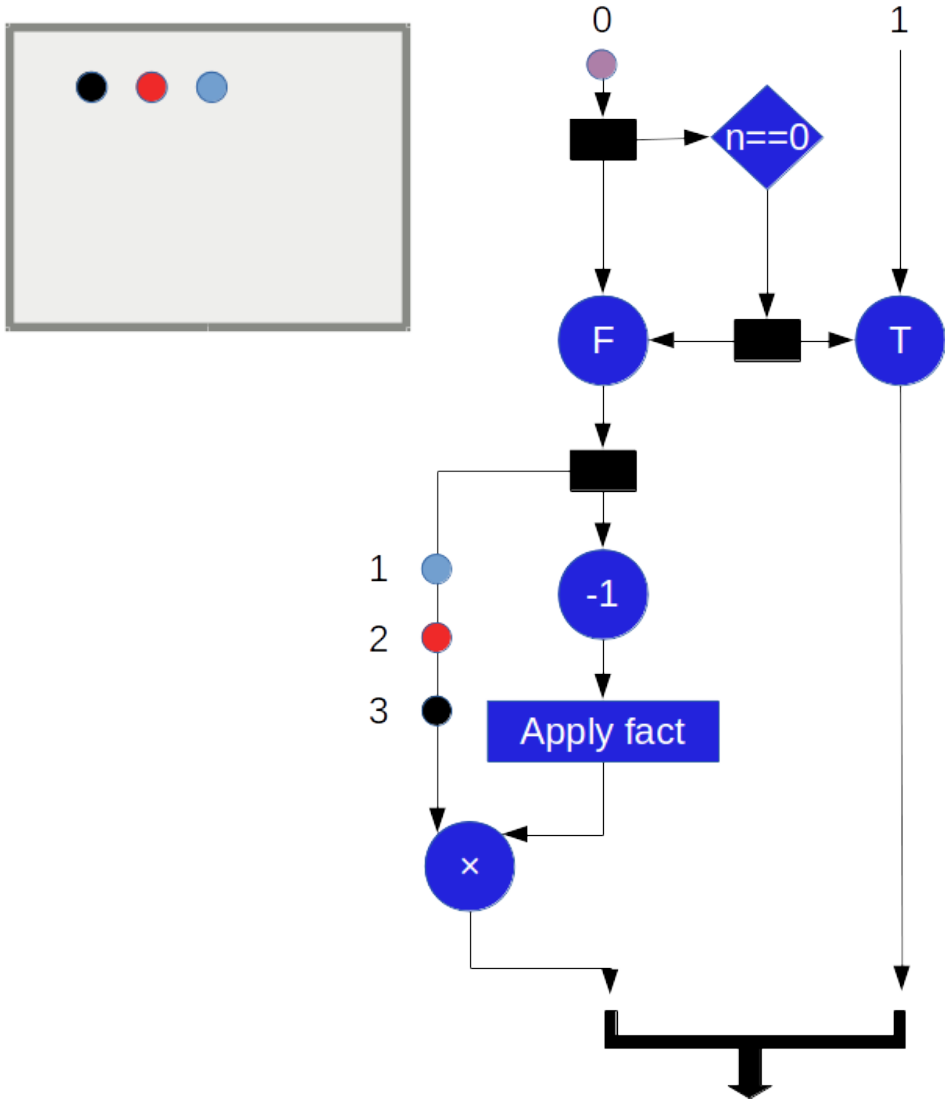
Revisiting Ordinary Recursions

Factorial



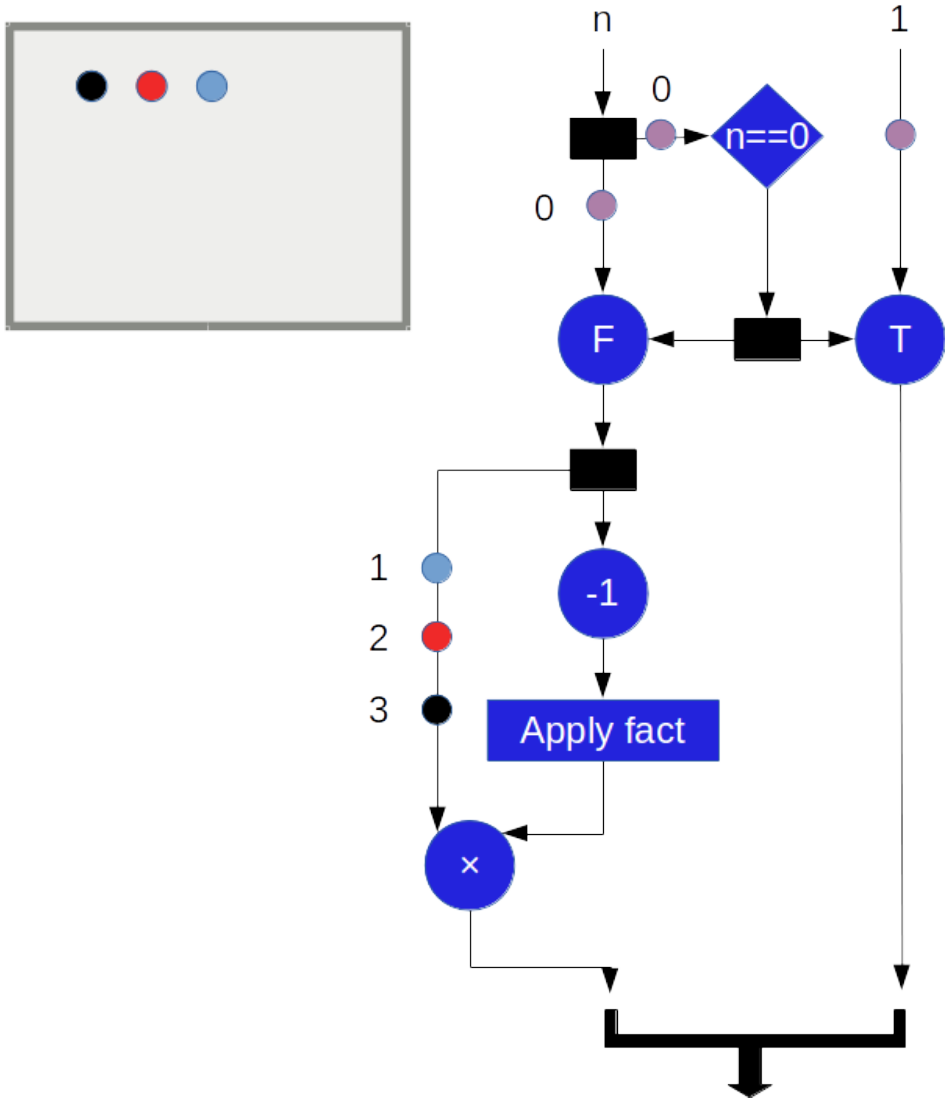
Revisiting Ordinary Recursions

Factorial



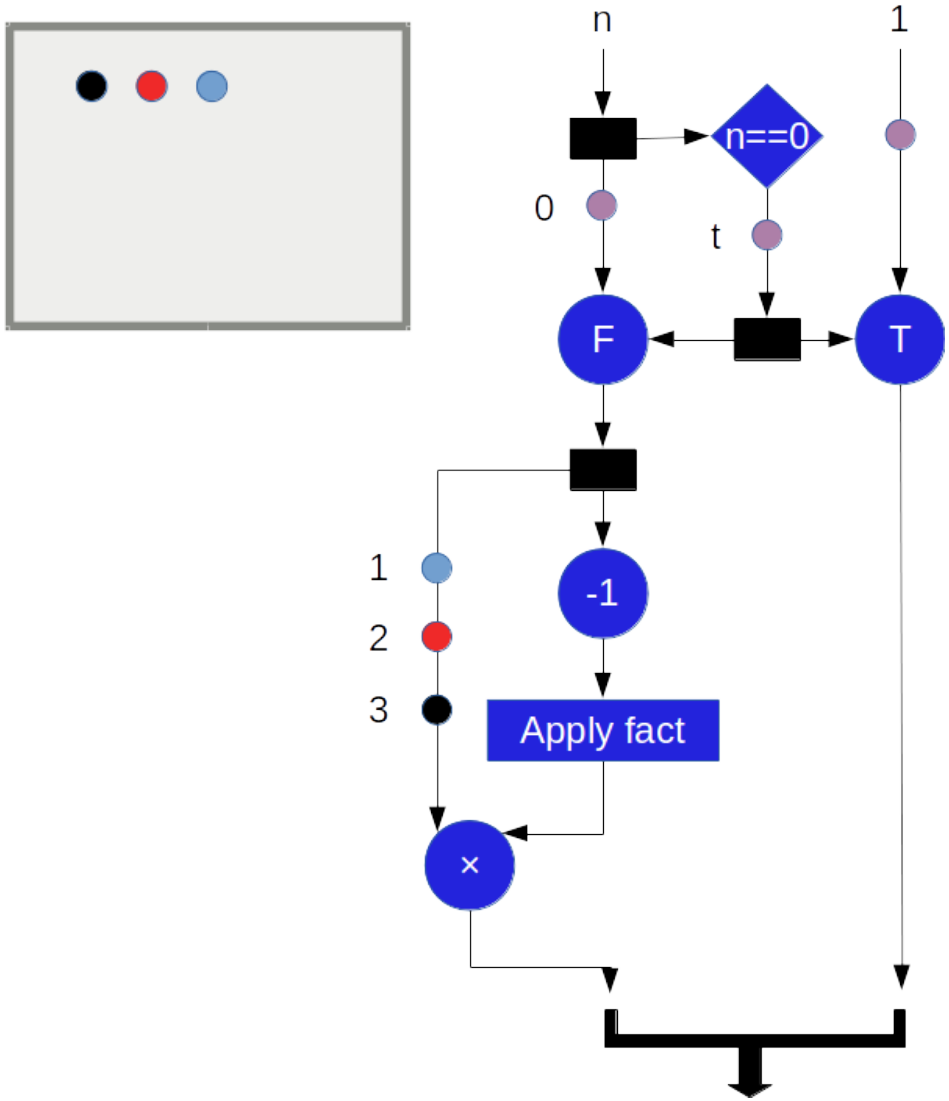
Revisiting Ordinary Recursions

Factorial



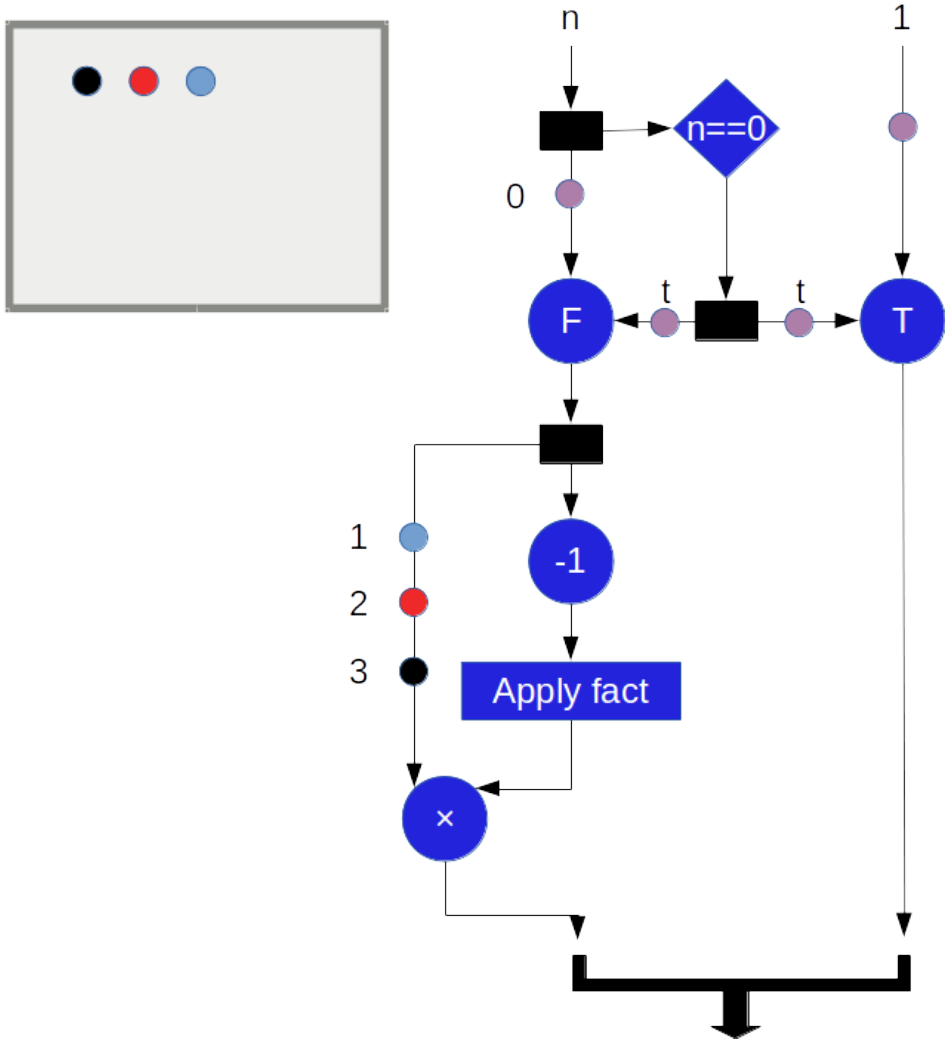
Revisiting Ordinary Recursions

Factorial



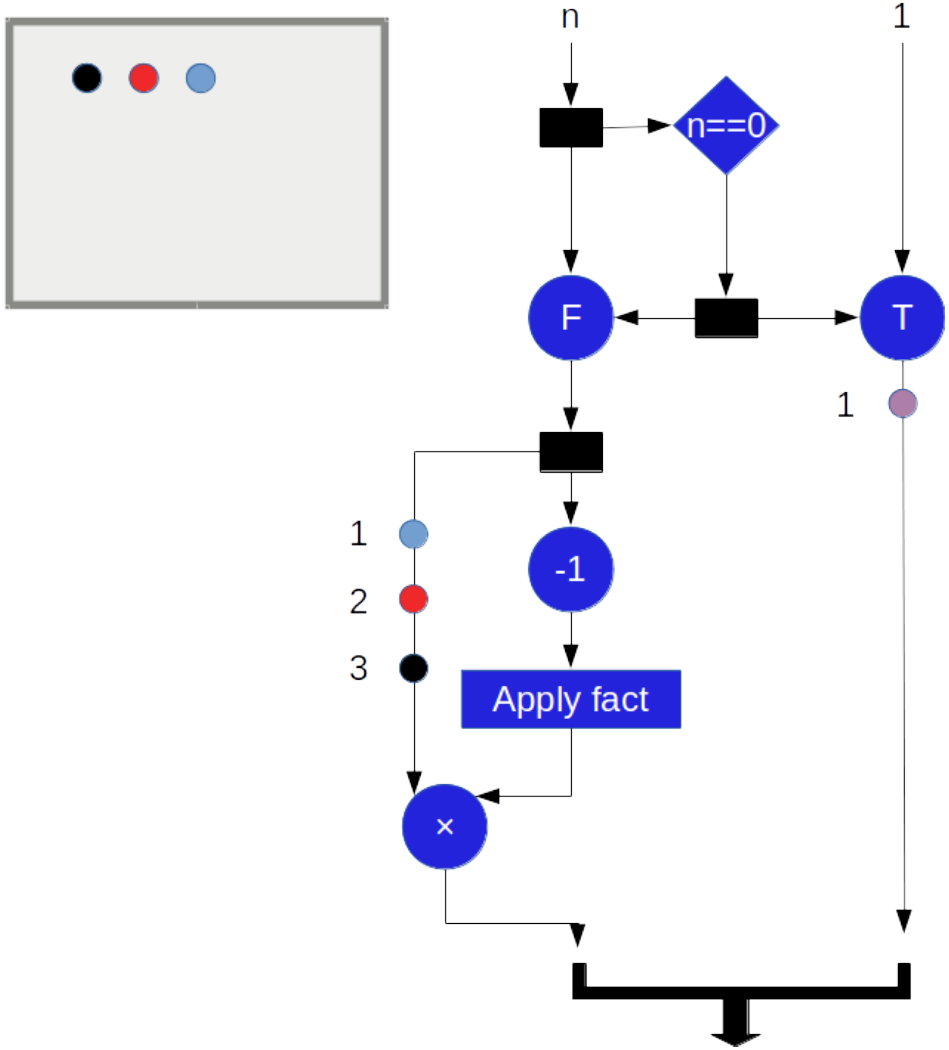
Revisiting Ordinary Recursions

Factorial



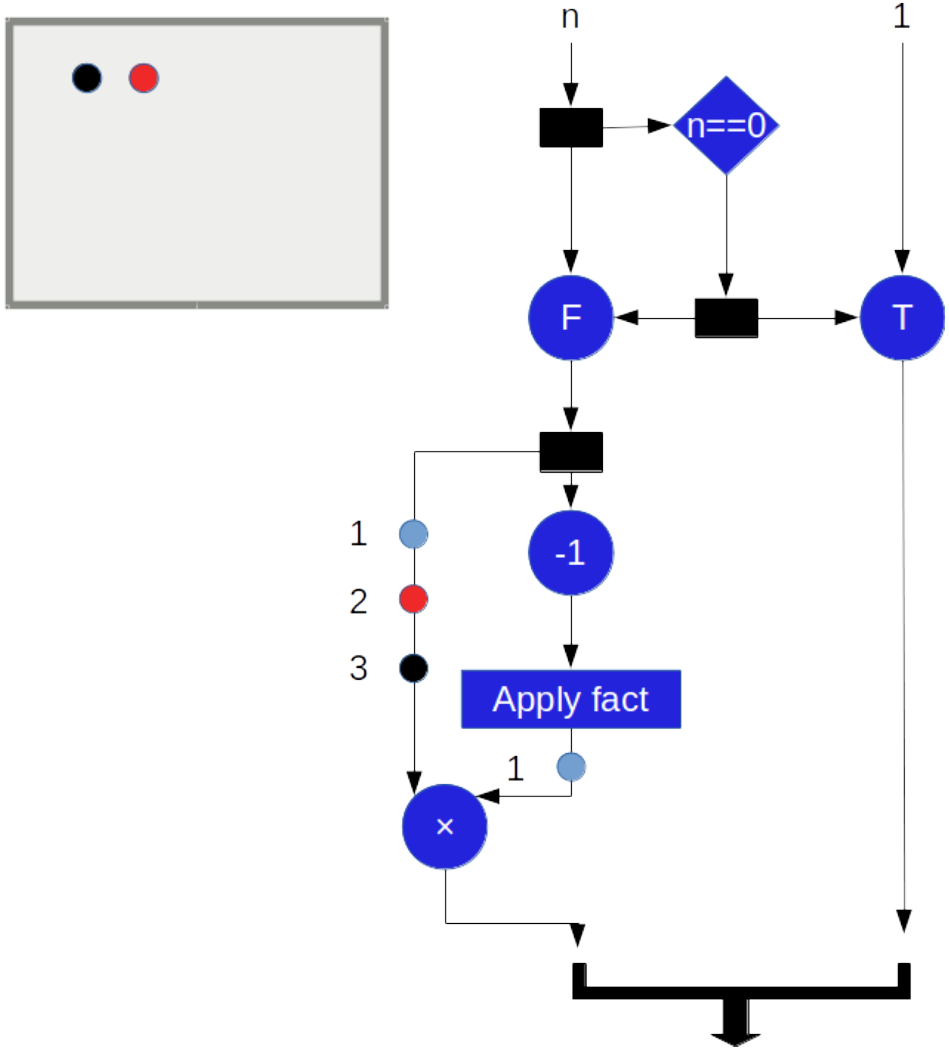
Revisiting Ordinary Recursions

Factorial



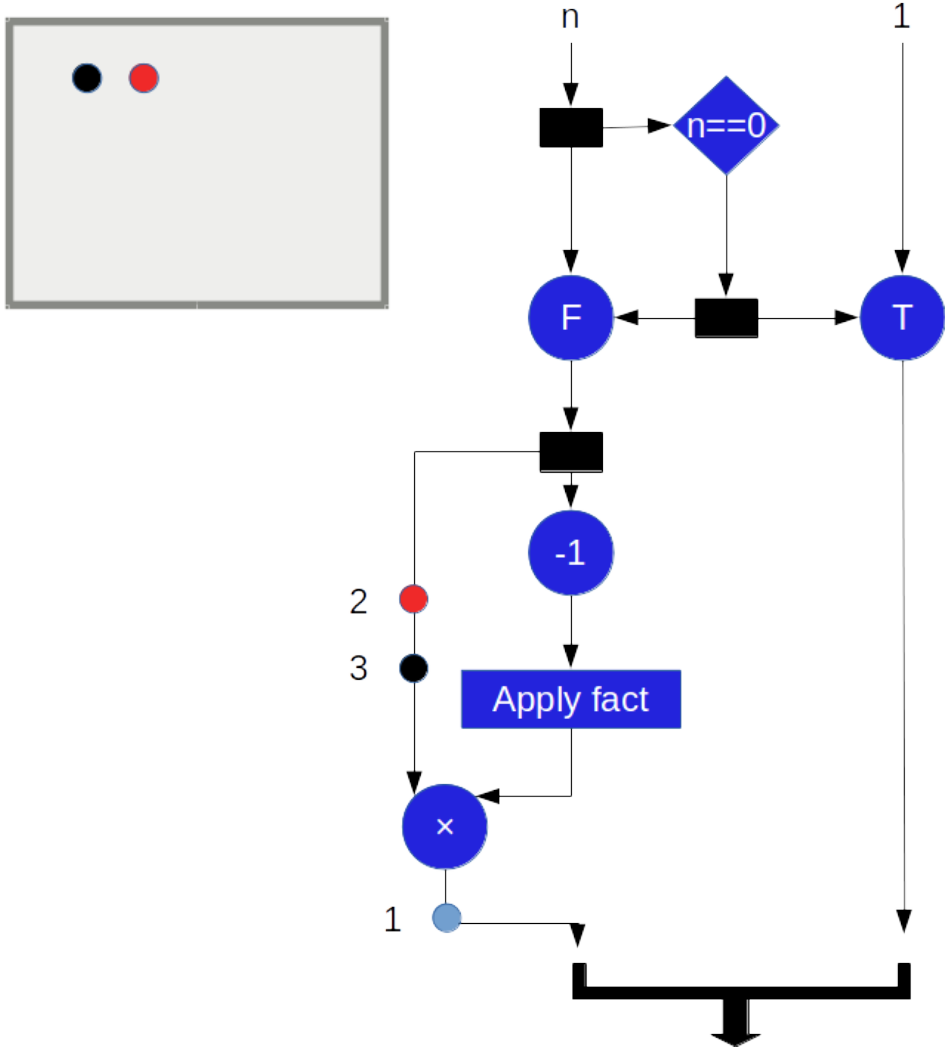
Revisiting Ordinary Recursions

Factorial



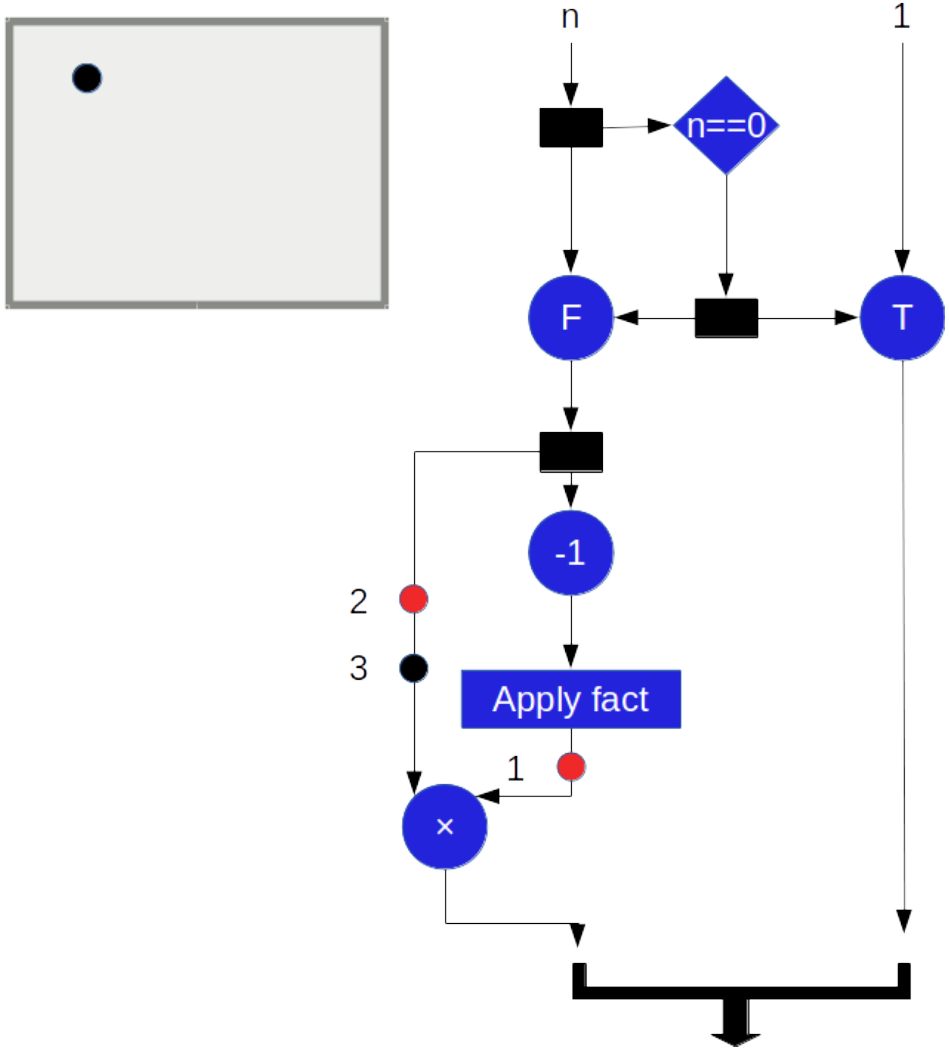
Revisiting Ordinary Recursions

Factorial



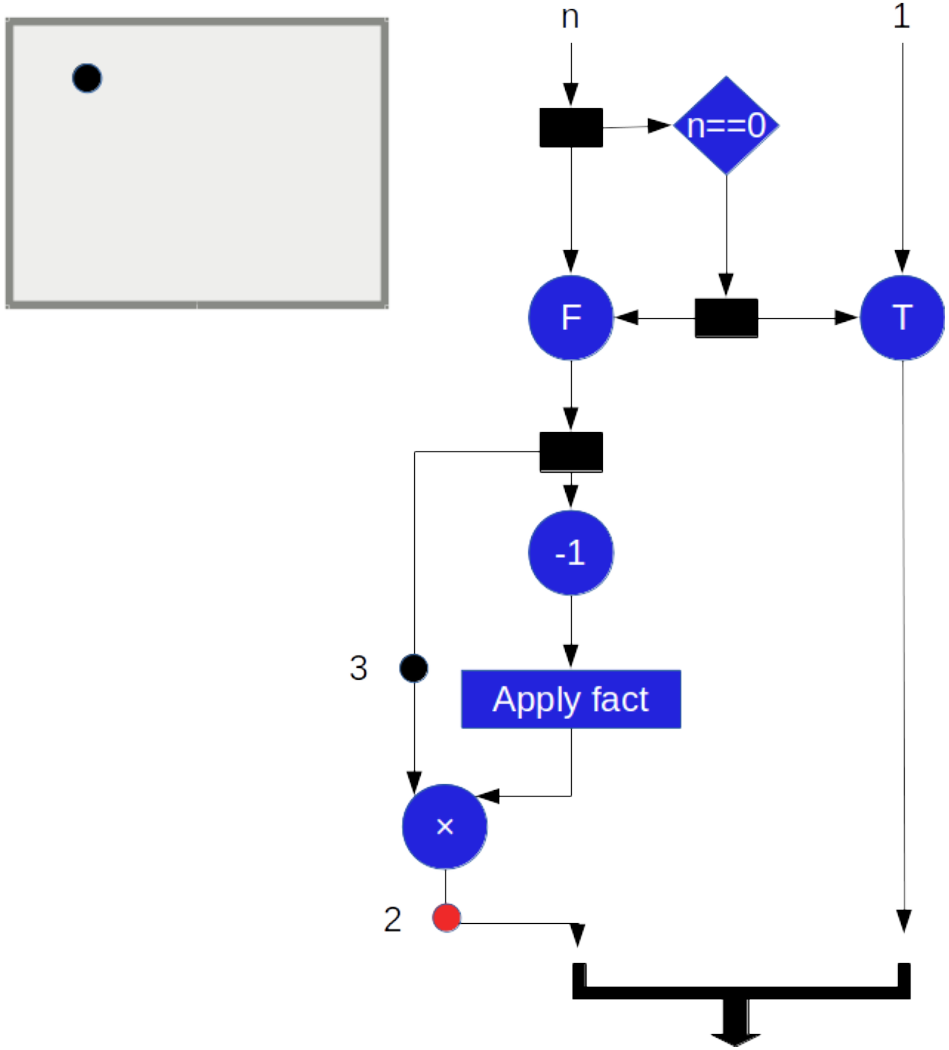
Revisiting Ordinary Recursions

Factorial



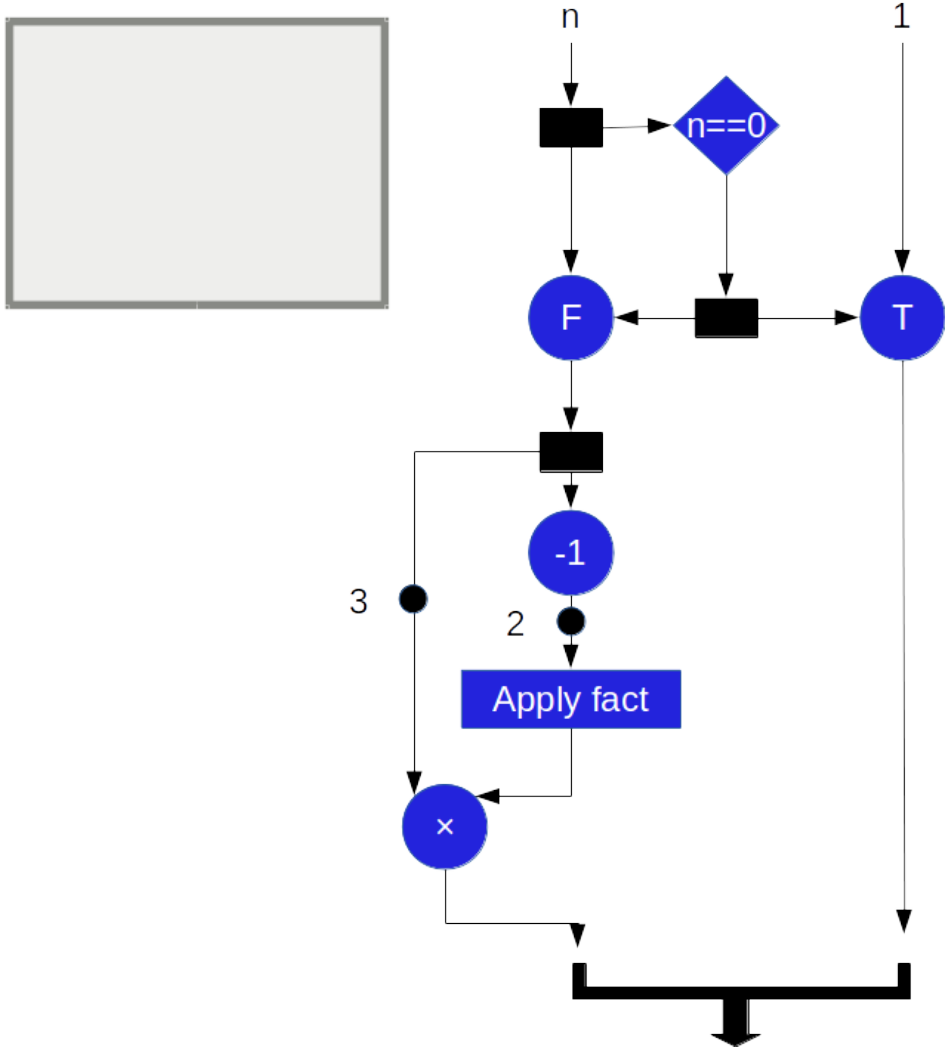
Revisiting Ordinary Recursions

Factorial



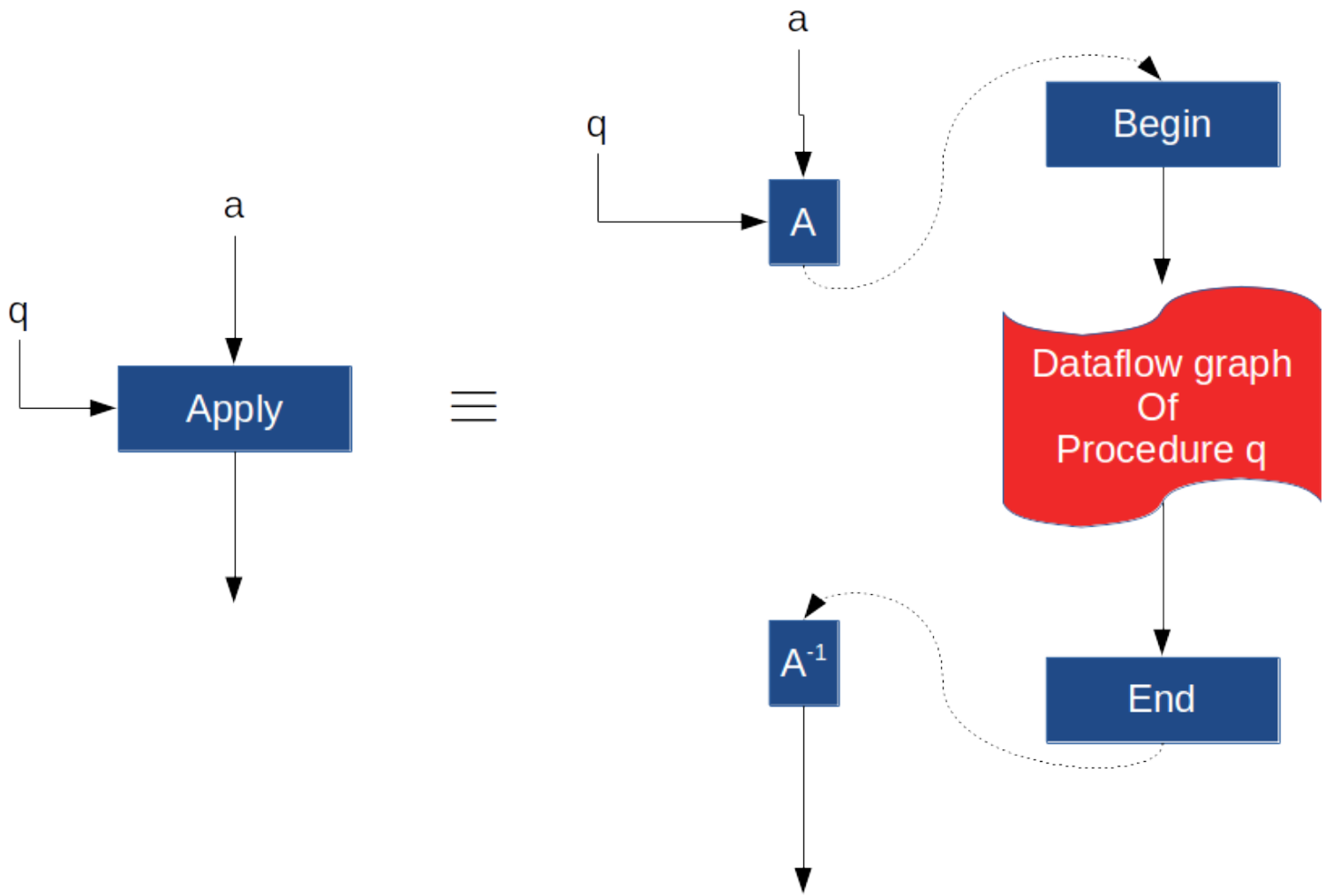
Revisiting Ordinary Recursions

Factorial



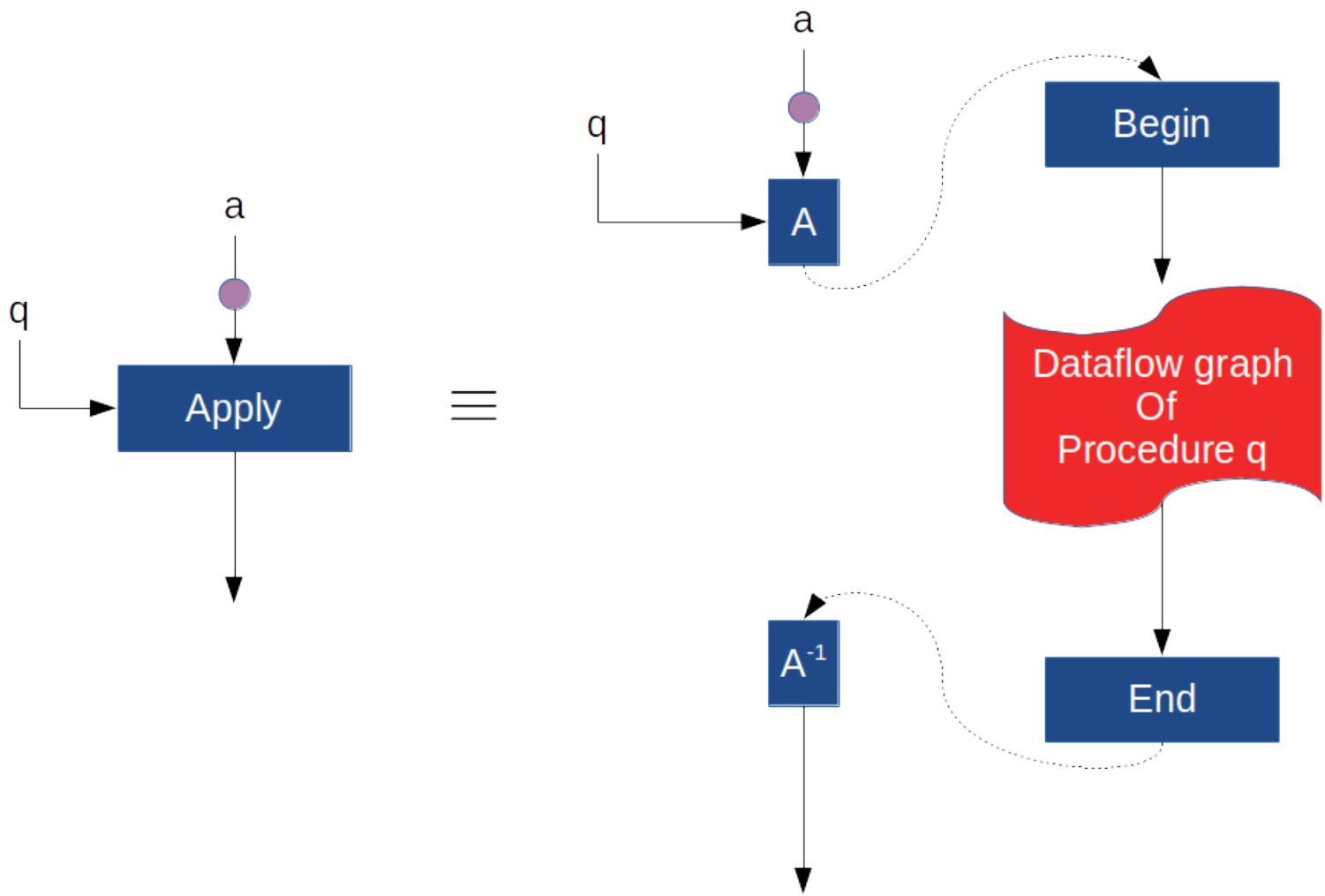
Dynamic Dataflow

The Apply Operator



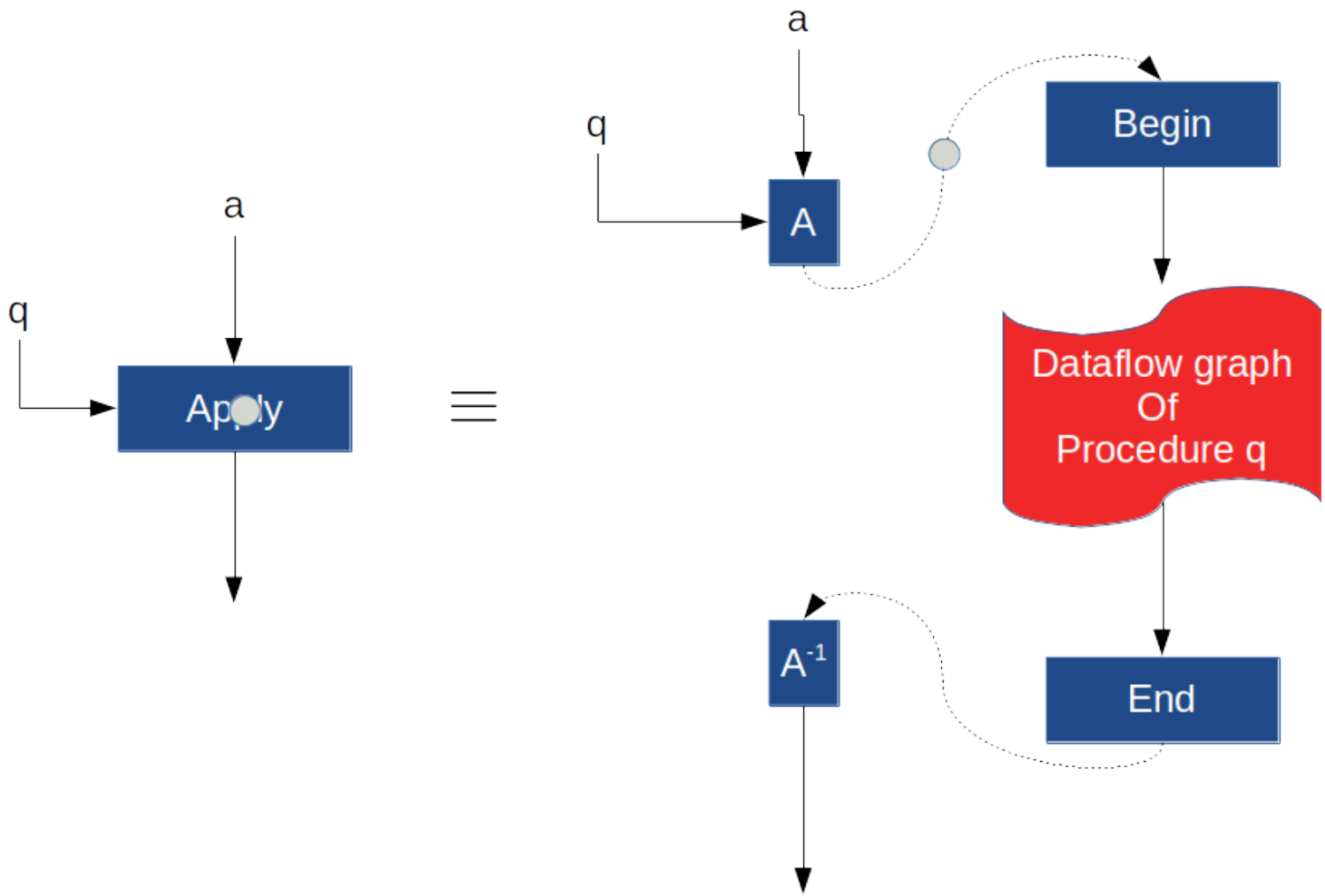
Dynamic Dataflow

The Apply Operator



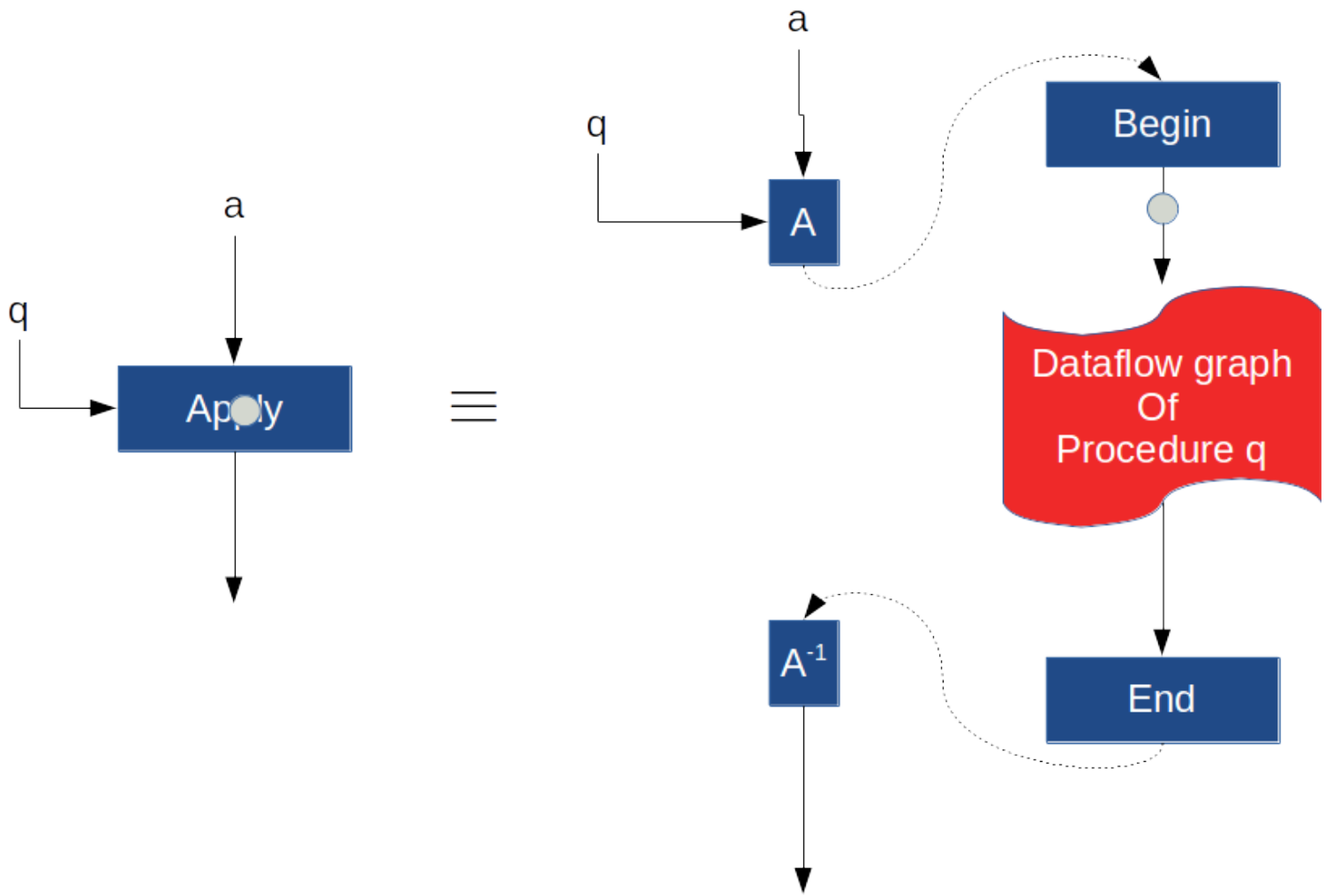
Dynamic Dataflow

The Apply Operator



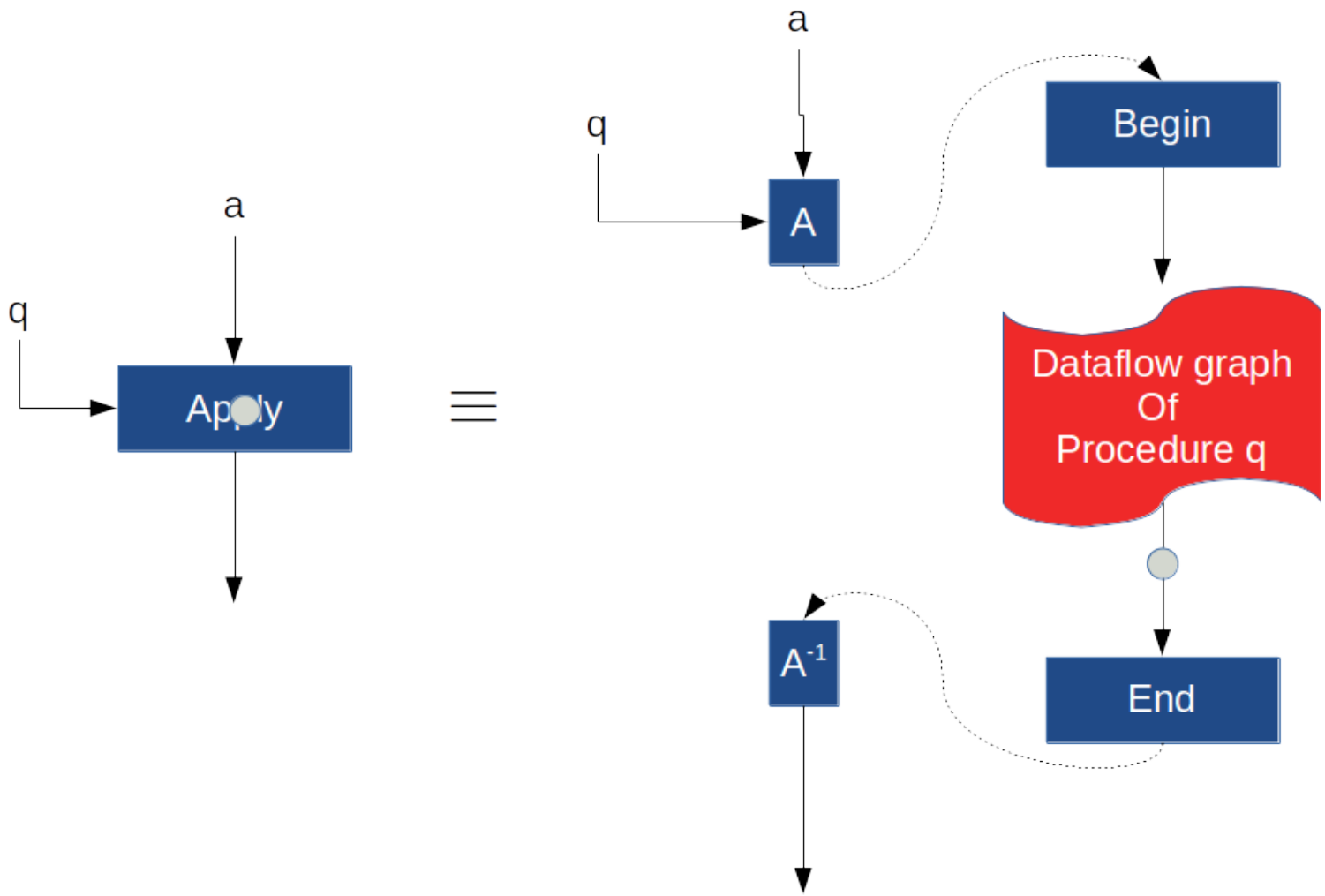
Dynamic Dataflow

The Apply Operator



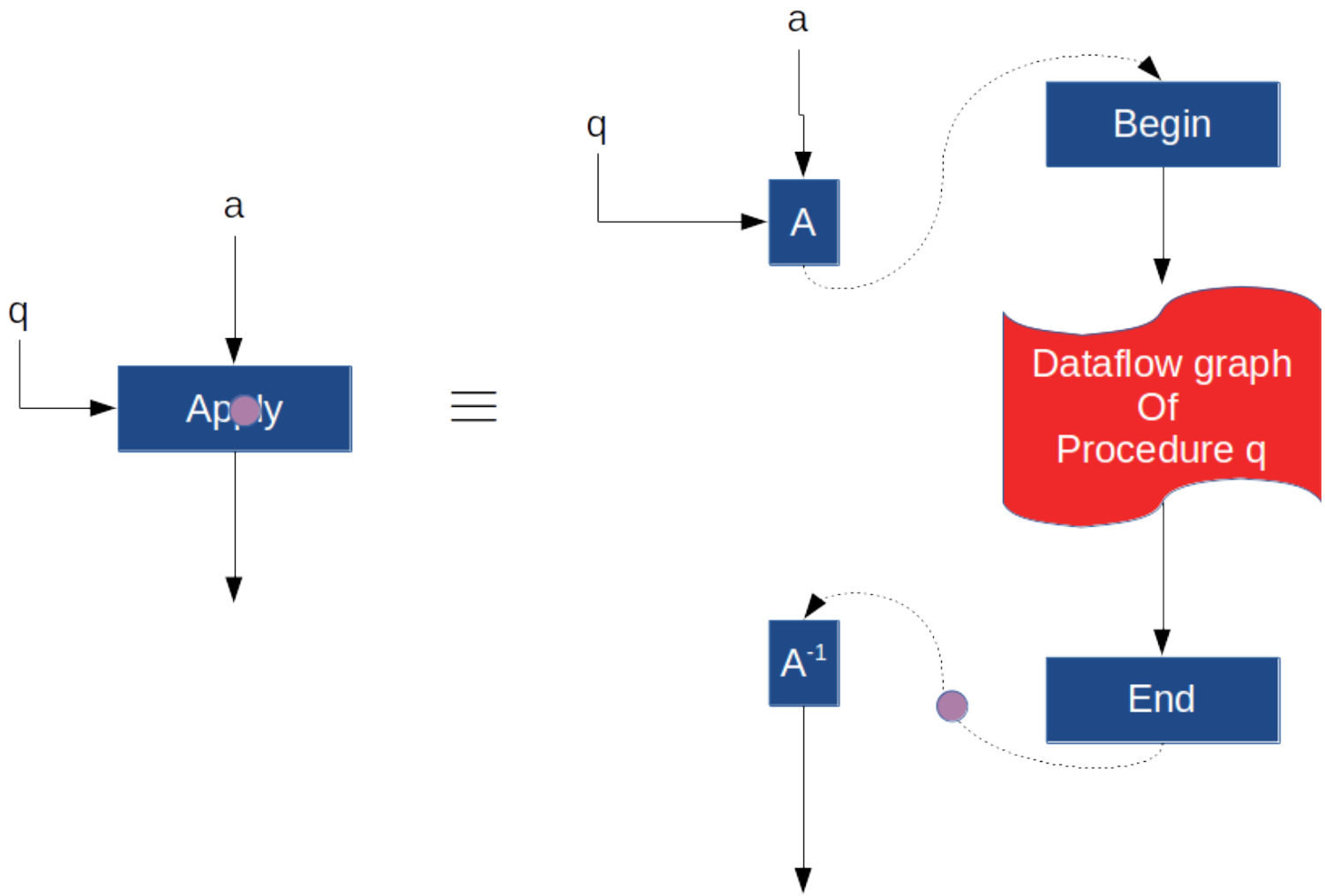
Dynamic Dataflow

The Apply Operator



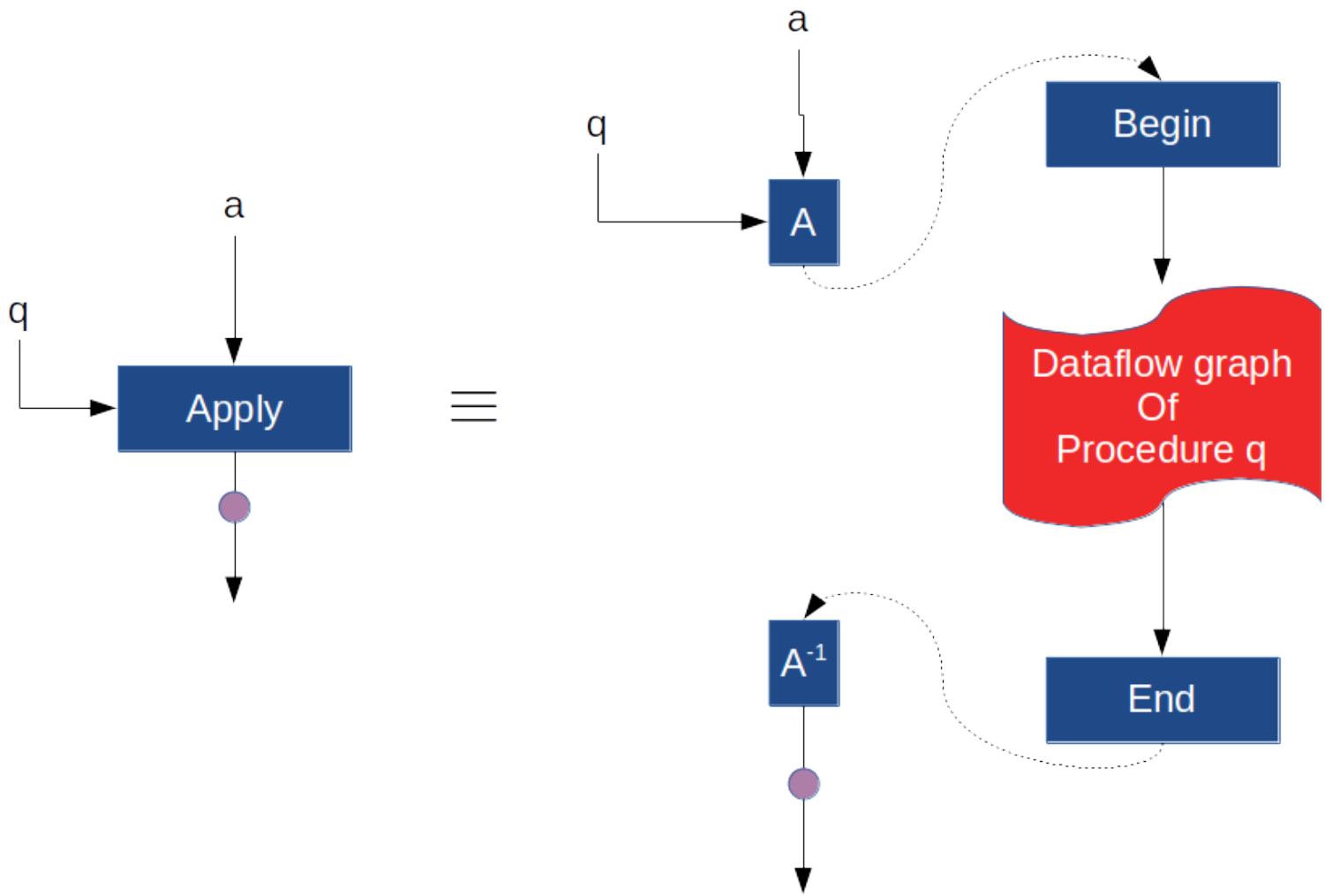
Dynamic Dataflow

The Apply Operator



Dynamic Dataflow

The Apply Operator



Dynamic Dataflow

Features

- Advantages:
  - "Most parallel model out there"
    - You can be as parallel as Dynamic Dataflow, but not more
  - Can handle arbitrary recursion
- Shortcomings:
  - Needs to implement a token tag matching unit
  - Ideally: use of associative memory (*i.e.*, use of key-value pairs)
  - . . . But it is not cost effective
    - Instead, hashing can be used.

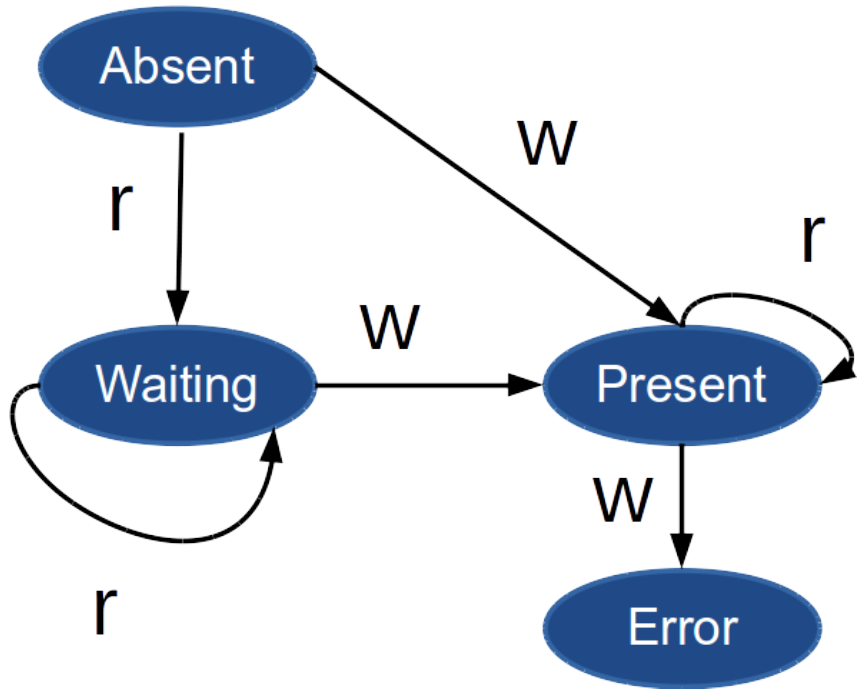
Contrary to the von Neumann model, dataflow models try to avoid side effects. I-Structures were designed to allow memory exploitation by dataflow programs.

- Special memory which follows a single assignment rule
  - Each access to the data structure "consumes" it entirely
- Basic concept of the I-Structure:
  - Only consume the entry on a write
  - An I-Structure is a data repository which obeys the single assignment rule
  - An I-Structure entry is written once, but read many times
- A data structure has a nil (undefined) value on a write
  - Once written, the data becomes immutable
  - As long as the data has not be written to the structure, all reads are deferred

- This allows a program to use a data structure before it is completely defined
- Data structures can be incrementally created or read

Status

- Present: The element can be read but not written
- Absent: The element has been attempted to be read but the element has not been written yet (initial state)
- Waiting: At least one read request has been deferred
- Error: an element was written more than once, an or out-of-range memory access happened



Elementary Operations

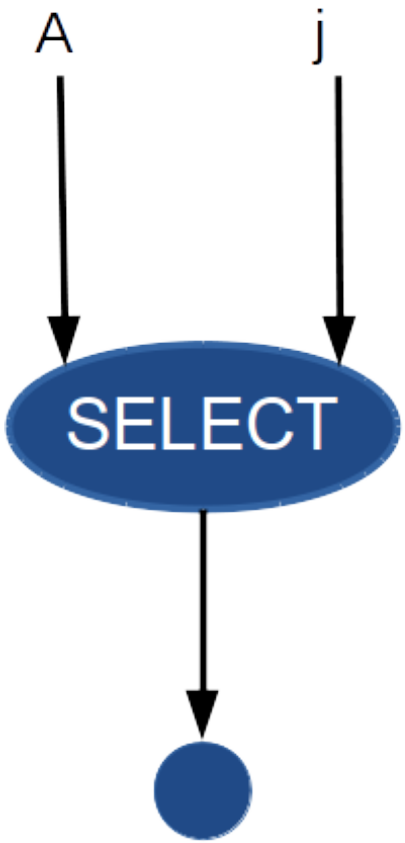
- `allocate`: reserves space for the new I-Structure
- `I_fetch`: get the value of the new I-Structure (deferred)
- `I_store`: writes a value into the specified I-Structure element

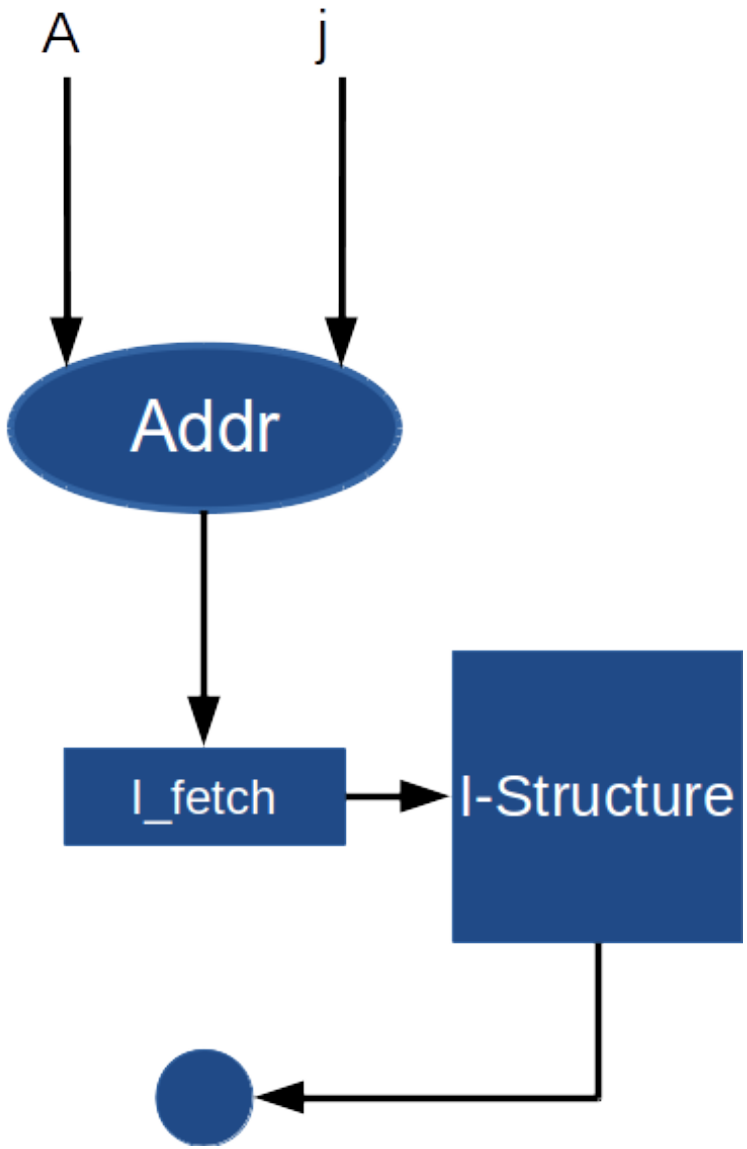
Node Construction

- `Select` — think `...= A[i]`
- `Assign` — think `A[i] = ...`

Node Construction

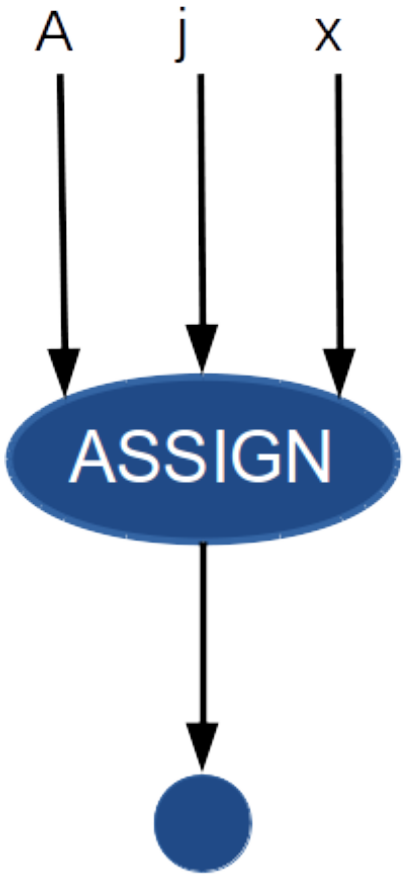
Select

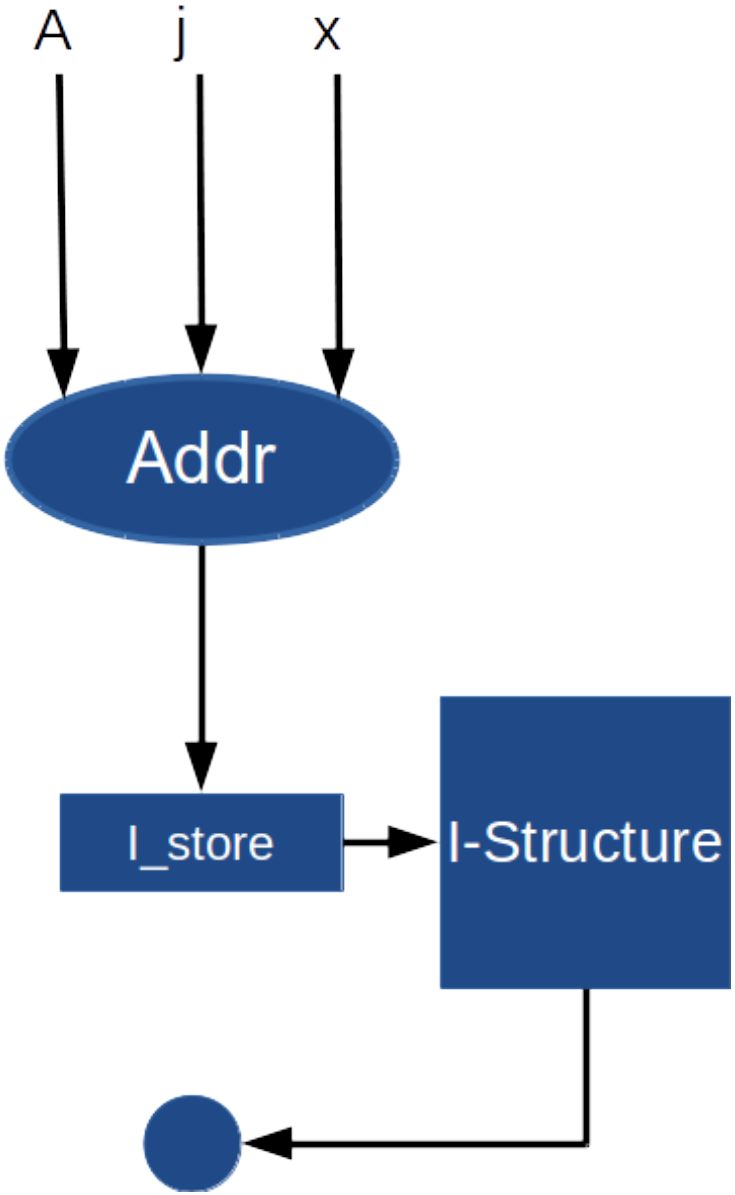




Node Construction

Assign





Determinacy, Determinism, and Indeterminacy

- An execution is deterministic when
  - For a given set of inputs, the computation always produces the same set of outputs
  - The resulting computation issues operations exactly in the same order each time it is executing.
- An execution is determinate when
  - For a given set of inputs, the computation always produces the same set of outputs
  - The order of individual (sets of) instructions does not matter for the end result
- An execution is indeterminate or non-determinate when
  - There is no guarantee that a given set of inputs fed to the computation will always yield the same outputs if we fire the computation with the same inputs.

- Not history-sensitive (*i.e.*, doesn't depend on time)
- Clear & simple semantics which lead to determinacy
- Maximal parallelism can be reached
  - It actually can be a problem...

A Word on Eager vs. Lazy Expression Evaluation

Definition: Eager Evaluation

A system of expression evaluation is said to be *eager* when all its input arguments are evaluated *before* they are being used.

Definition: Lazy Evaluation

A system of expression evaluation is said to be *lazy* when its input arguments are evaluated only at the time they are needed (*i.e.*, when the system needs to read or maybe write a value).

A Word On Strictness I

Intuitive Definition

A mechanism is strict if it requires all its arguments to be evaluated from the innermost expression toward its outermost expression. As a result, if an inconsistent or error state is encountered somewhere during the *reduction* of an expression, it will propagate the error state to the outer expressions.

Formal Definition

- A function f is strict if $f \perp \rightarrow \perp$
- you can read this as "a function applied to an invalid expression results in (or is reduced to) an invalid result"

A Word On Strictness II

Definition: Non-Strict Expression Reduction

- A function f is *non-strict* if $f \perp \rightarrow v, v \neq \perp$
- Non-strict functions evaluate expressions from the outermost level down to the innermost ones
  - By doing so, they may end up simplifying expressions which could yield errors, but which will never be reached.
  - This feature inherently allows for more "loose" expression evaluation, and, as a result, for more parallelism

Using a Strict Apply Actor

- Provides a lot of parallelism
- Proposes a call-by-value type of semantics
- But: we lose the "substitution" property (also called "referential transparency")
  - In other words: we risk losing determinacy
  - To avoid this, a "strictness analysis" must be performed (at compile time)

Referential Transparency I

More Formal Definitions

Wikipedia's

An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referential opacity.

Dr. Gao's

. . . The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. . .

. . . Moreover, the value of an expression is, within certain limits, the same when ever it occurs. . .

Referential Transparency and the Substitution Property

Let f, g be two procedures/functions such that f appears as an application inside of g; let g' be the procedure obtained from g by substituting the text of f in place of the application;
In languages which are "referentially transparent", the specification of the function for g will not depend on any terminating property of f, or $g{=}g'$

Strictness, Eagerness, Non-Strictness, and Laziness I

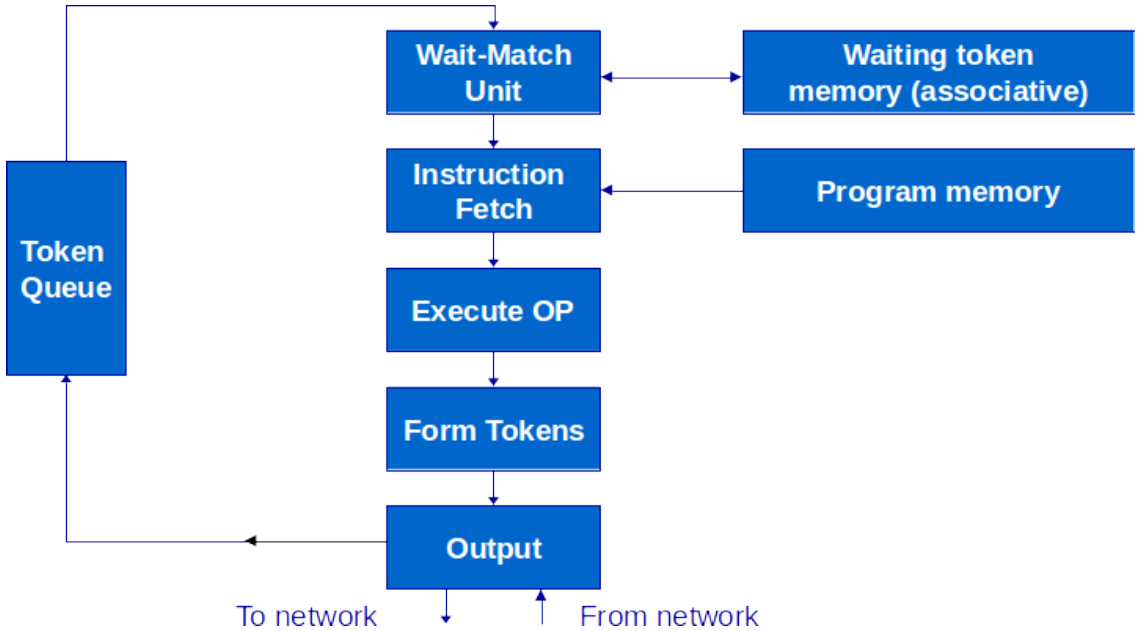
- There is very often a confusion between
  - eager expression evaluation and strict expression reduction
  - lazy expression evaluation and non-strict expression reduction
- Eager/lazy evaluation refers to operational behavior, *i.e.*, *how* the program "reads" (evaluates) an expression.
- Strict/Non-strict refers to semantics, *i.e.*, the mathematical *meaning* of a given expression
  - Which is why say that we "reduce" expressions (which from a mathematical point of view, boils down to meaning "evaluate").
- In practice, non-strict and lazy can often be used interchangeably, but beware!
- There is a class of expression reduction schemes called lenient executions
  - Lenient executions are non-strict

Strictness, Eagerness, Non-Strictness, and Laziness II

- Lenient executions are not eager : arguments are not necessarily *all* evaluated before evaluating an expression,
- Lenient executions are not lazy: arguments may be evaluated before they are needed

Dynamic Dataflow Architecture

The MIT Tagged Token Dataflow Architecture Processor

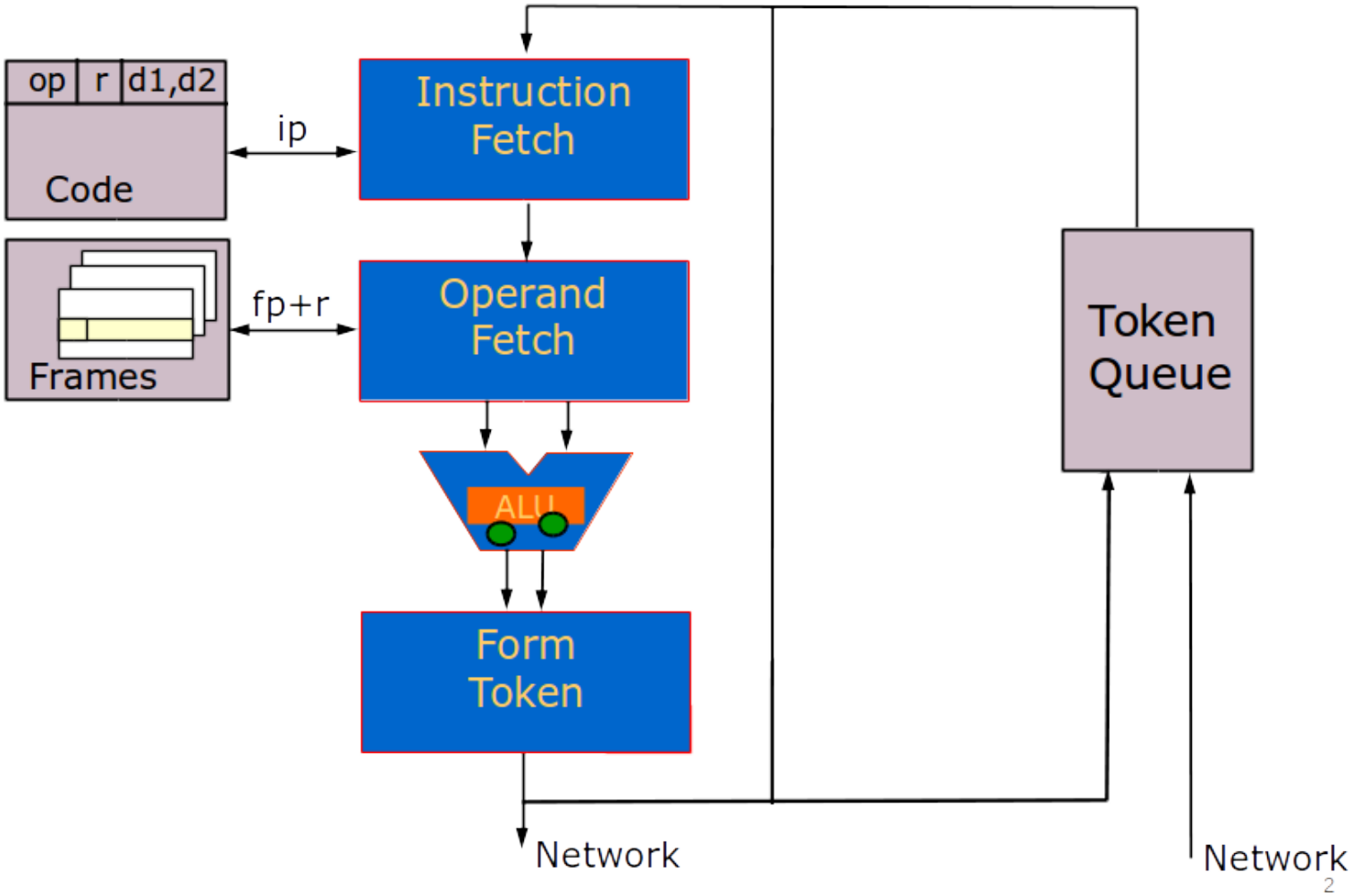


Wait-Match Unit:

- Tokens for unary operations: go straight through
- Tokens for binary operations: try to match incoming token and a waiting token with same instruction address and context id
  - Success: both token proceed forward
  - Failure: incoming token *rightarrow* waiting token memory, bubble (no-op) forwarded

Dynamic Dataflow Architecture

The Monsoon Dataflow Processor



Outline

Definition: Fibonacci Sequence

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = U_{n-1} + U_{n-2} \end{cases} , \forall n \in \mathbb{N}$$

- Let the definition of the *Fibonacci* function using a pseudo-C syntax be: `unsigned long fib(unsigned long n);`
- Write the pseudo-code for the recursive version of `fib`
- Write the pseudo-code for the tail recursive version of `fib`
- Write the pseudo-code for the iterative (loop) version of `fib`

Definition: Fibonacci Sequence

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = U_{n-1} + U_{n-2} \end{cases} , \forall n \in \mathbb{N}$$

- Let the definition of the *Fibonacci* function using a pseudo-C syntax be: `unsigned long fib(unsigned long n);`
- Write the pseudo-code for the recursive version of `fib`
- Write the pseudo-code for the tail recursive version of `fib`
- Write the pseudo-code for the iterative (loop) version of `fib`
- Push tokens!
  - Select one of the three versions you wrote for `fib`, and draw its corresponding dataflow graph
    - If it is the ordinary recursive version, use dynamic dataflow
    - If it is the tail-recursive version, use recursive program graphs
    - If it is the iterative version, use static dataflow

References I