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

Stéphane Zuckerman

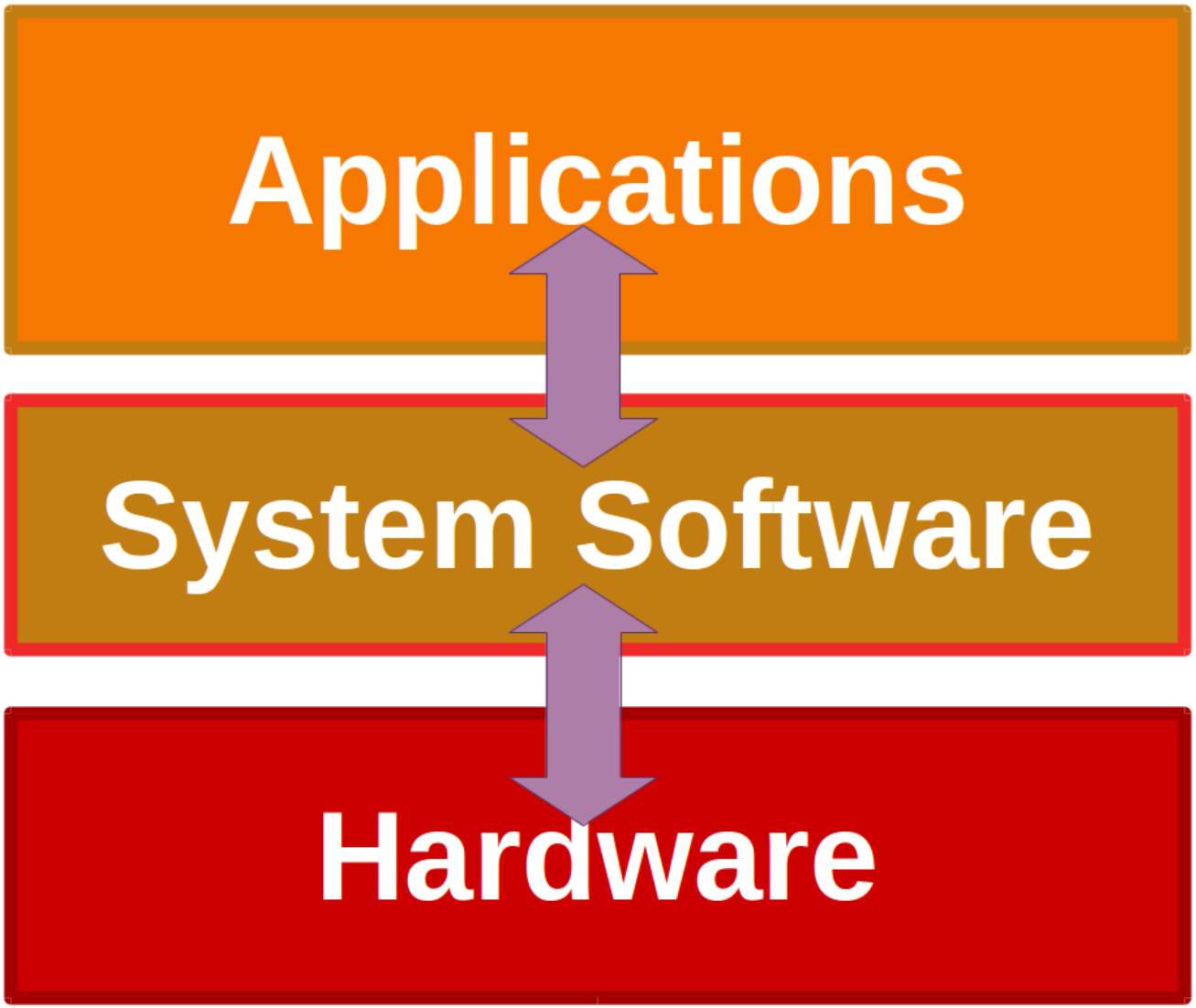
Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark,DE 19716, United States
szuckerm@udel.edu

September 8, 2015

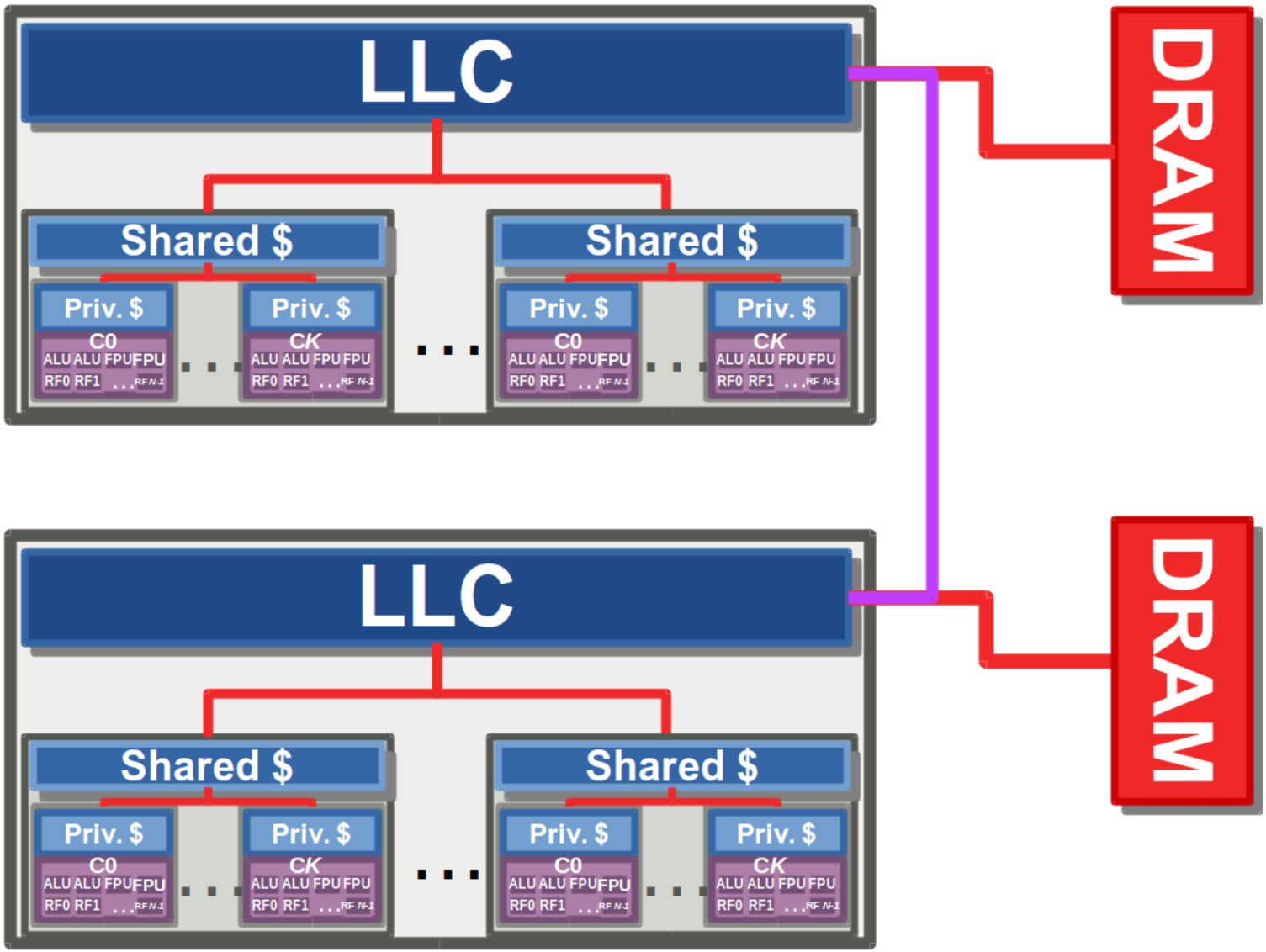
Outline

Outline

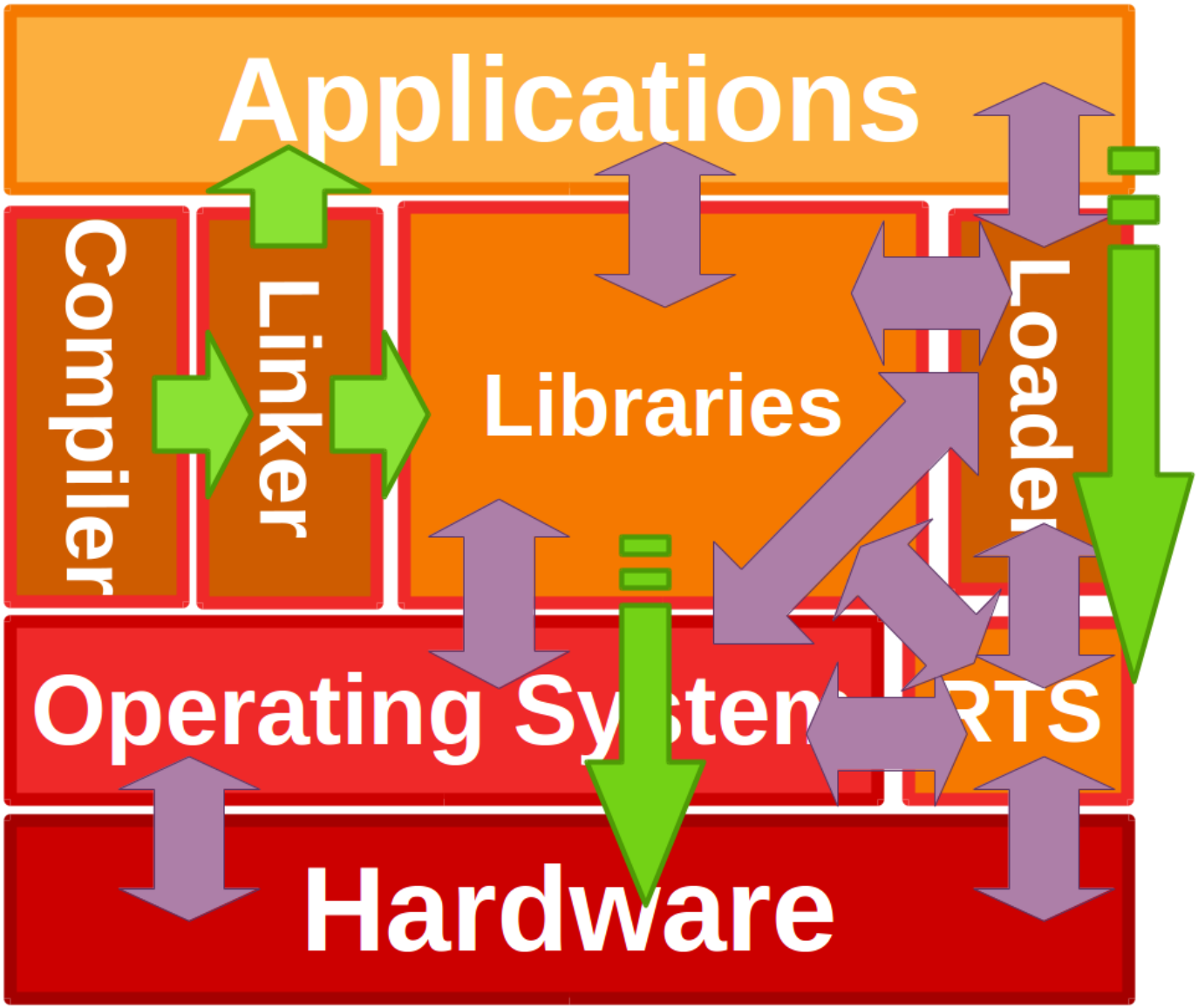
High-Level View of a Computer System



A Representative View of Current Compute Nodes' Hardware



A Representative View of Current Compute Nodes' Software Stack



Definition: Parallel Model of Computation

- Parallel models for algorithms designers
- Parallel models for system designers
  - Parallel programming models
  - Parallel execution models
  - Parallel architecture models

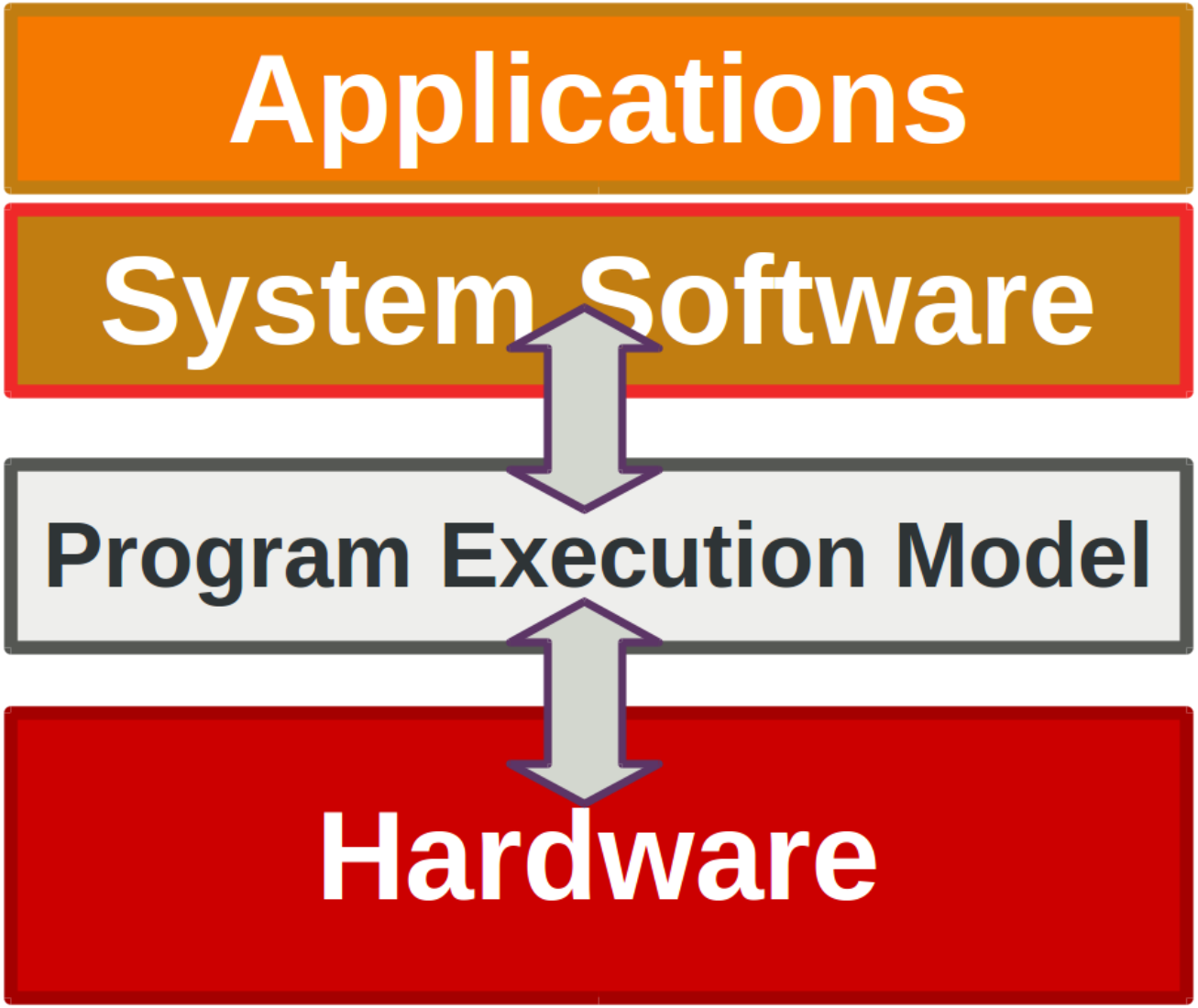
An Alternate Program Execution Model Definition

(Dr. Gao's) Definition: Program Execution Model

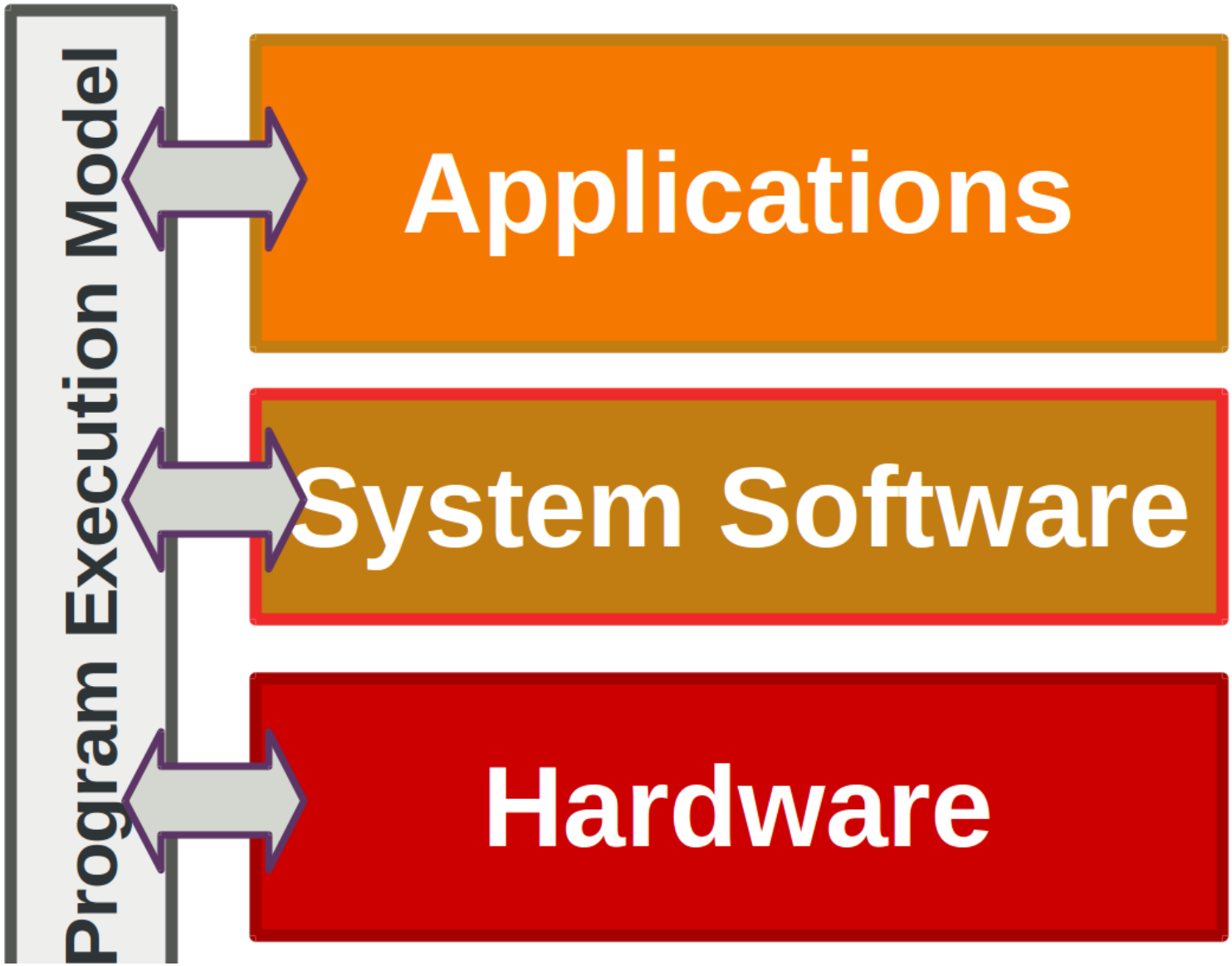
The program execution model (PXM) is the basic low-level abstraction of the underlying system architecture upon which our programming model, compilation strategy, runtime system, and other software components are developed. The PXM (and its API) serves as an interface between the architecture and the software.

Unlike an instruction set architecture (ISA) specification, which usually focuses on lower level details (such as instruction encoding and organization of registers for a specific processor), the PXM refers to machine organization at a higher level for a whole class of high-end machines as viewed by the users

Horizontal Conception of a PXM

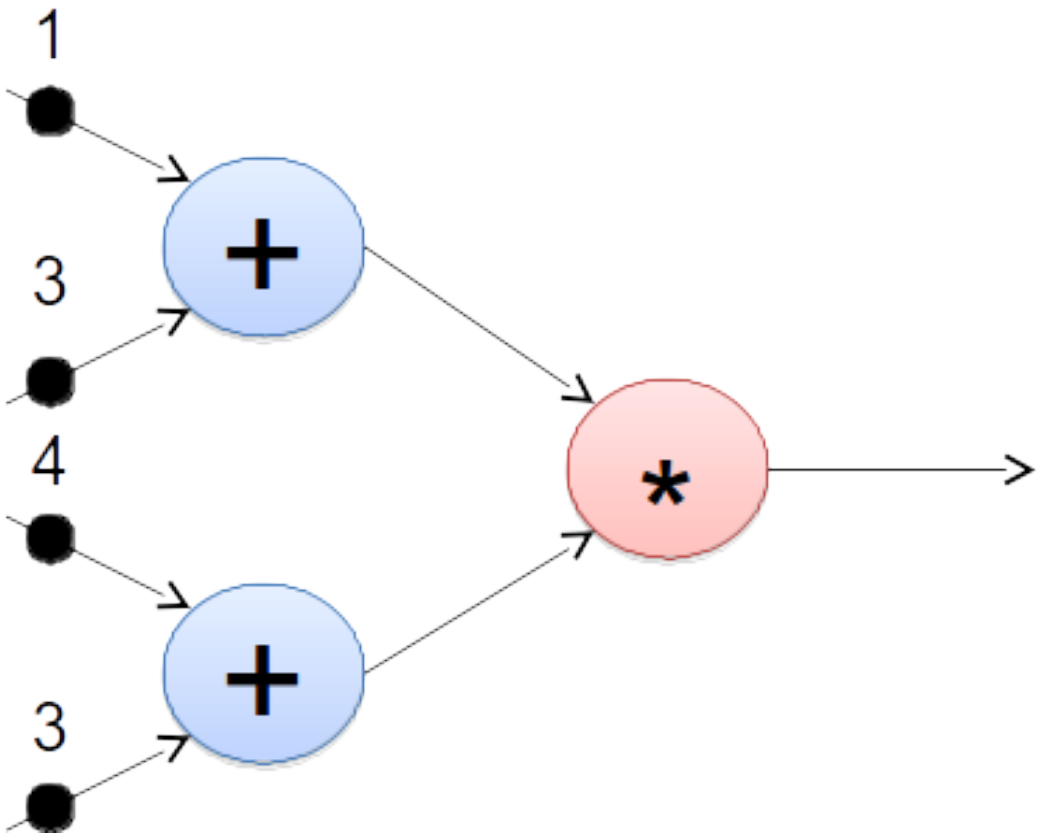


Vertical Conception of a PXM

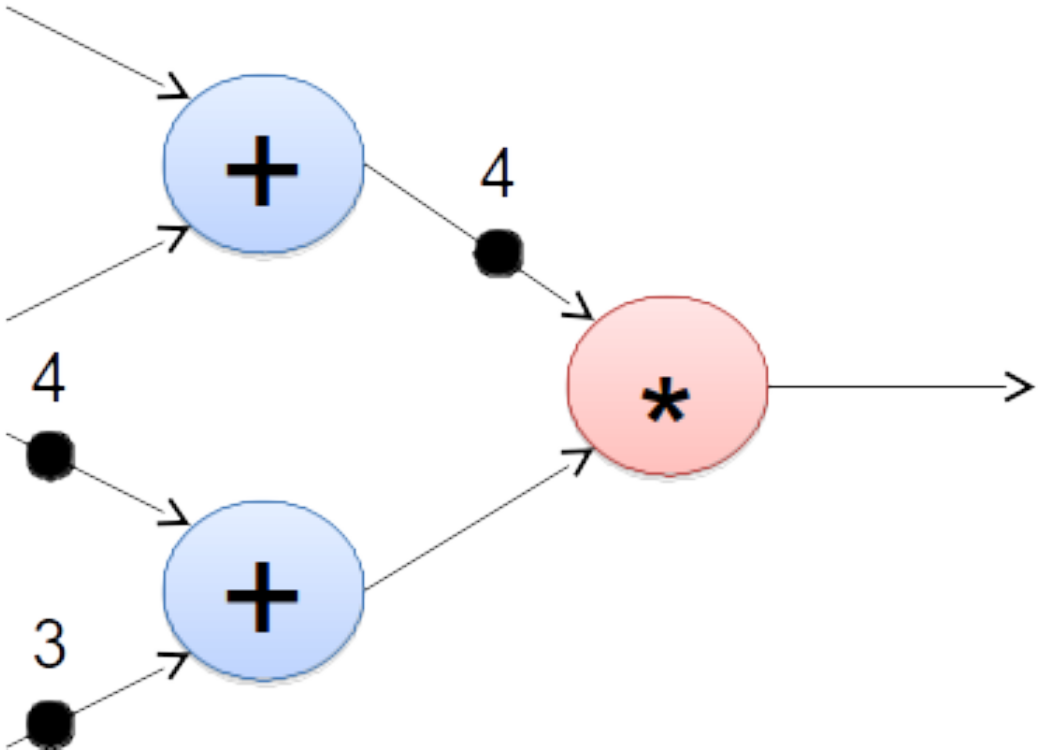


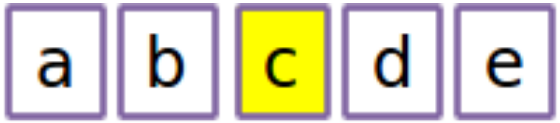
Outline

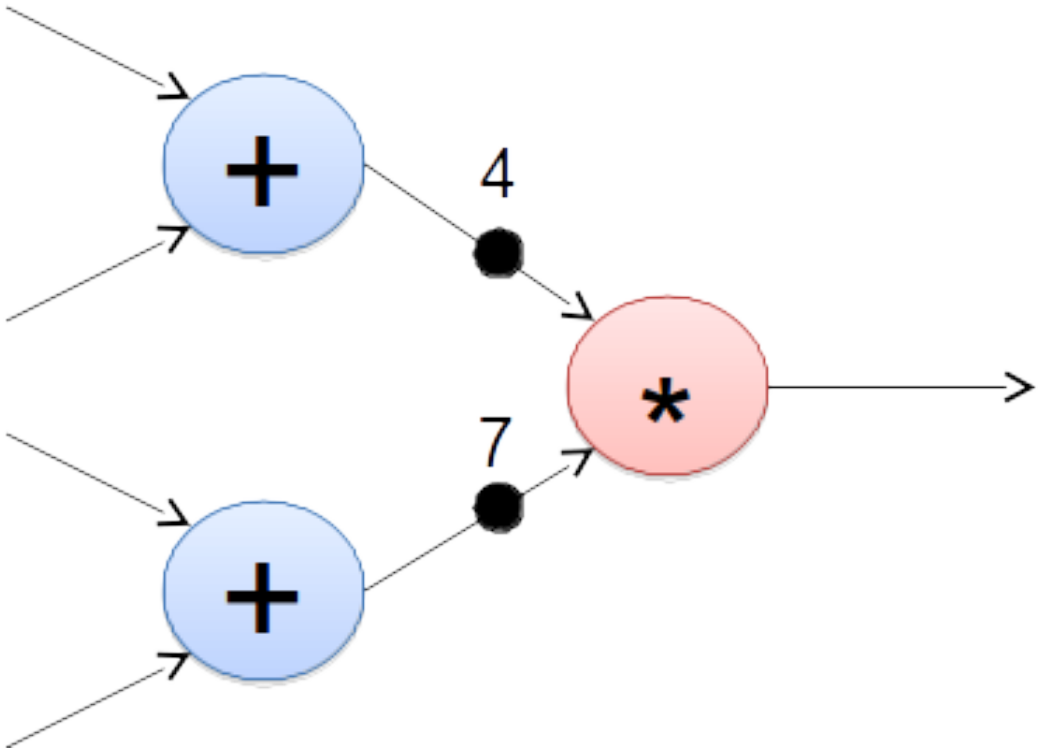

a
b
c
d
e

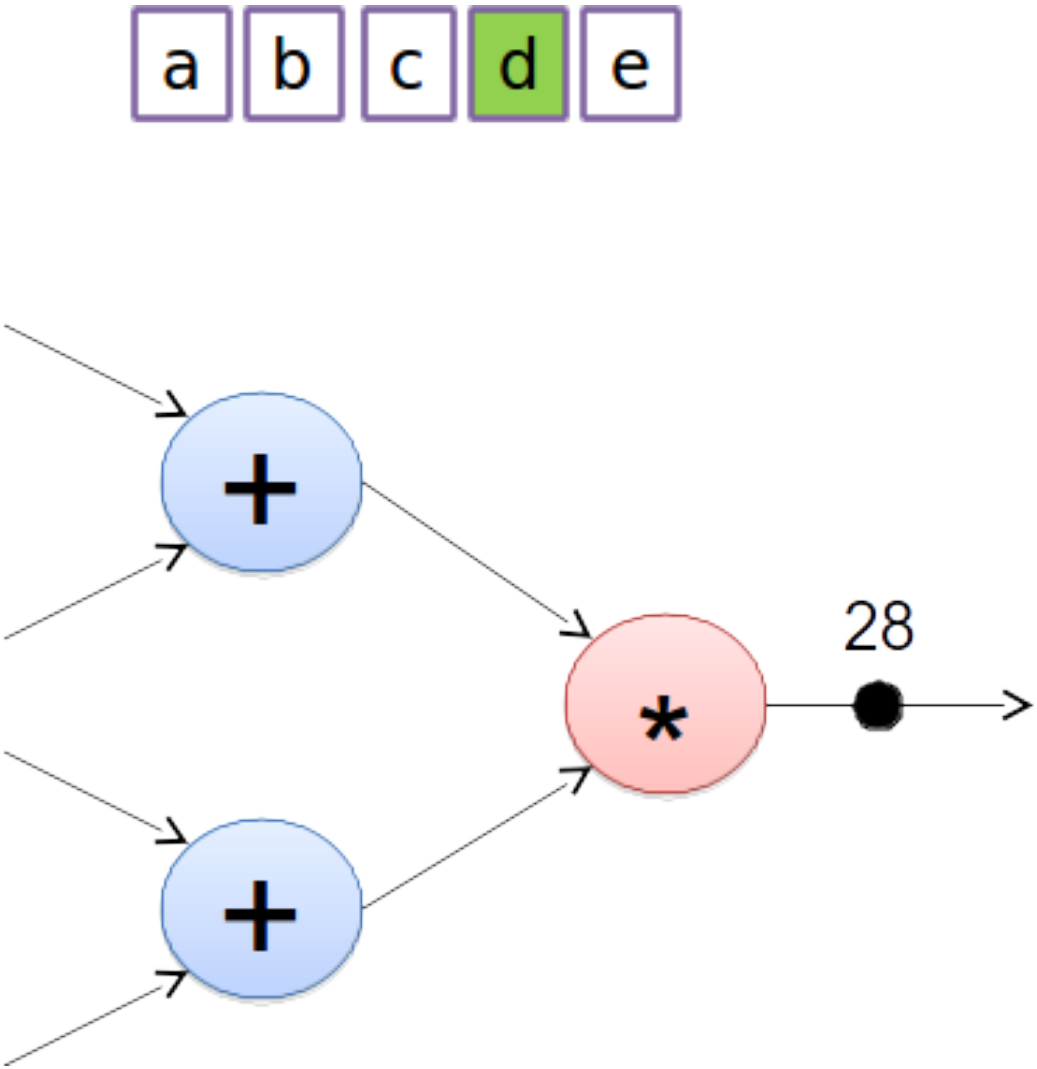

1
3
4
3
+
+
*


a
b
c
d
e

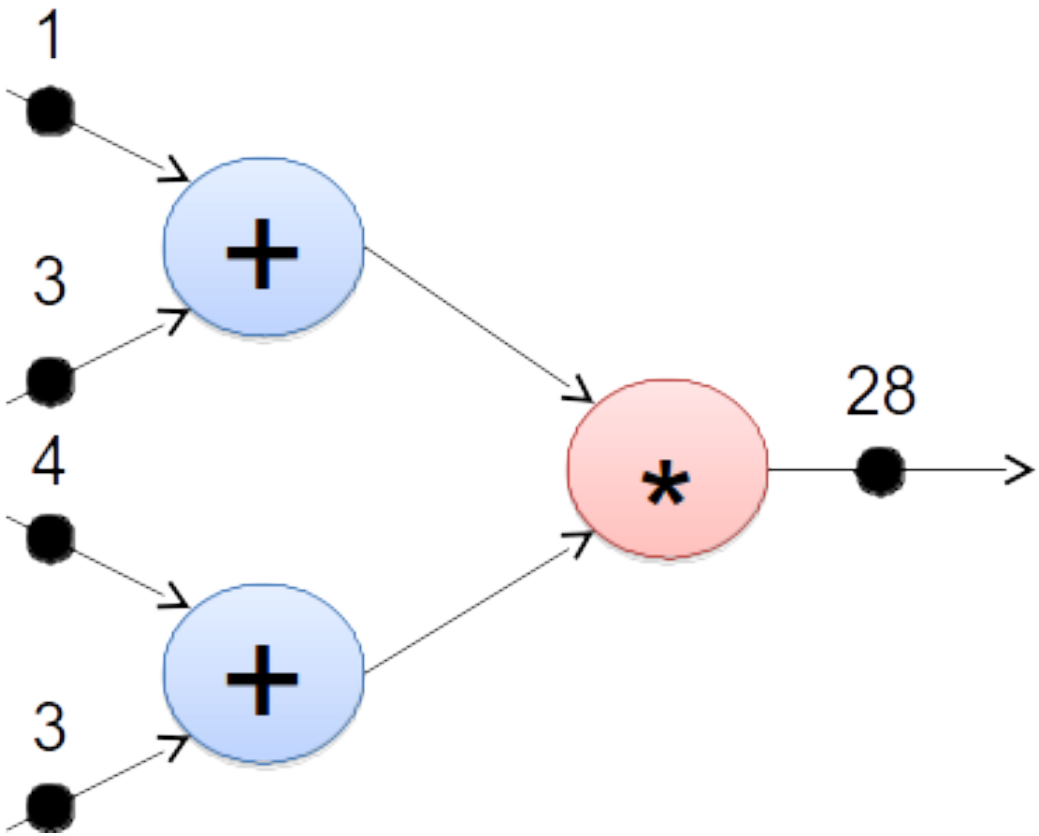

+
4
4
*
+
3


a
b
c
d
e


+
4
*
7
+


a
b
c
d
e
+
+
*
28


a
b
c
d
e


1
3
4
3
+
+
*
28

Dataflow Graphs

For J.B. Dennis, the role of Dataflow Graphs (DFGs) is two-fold:

- Serve as an intermediate-level language for high-level languages
- Serve as a machine language for parallel machine

A Brief History of Dataflow

MIT, 1974–1975

- **April 1974:** Symposium on Programming, Paris. Dennis: First Version of a Data Flow Procedure Language.
- **January 1975:** Second Annual Symposium on Computer Architecture, Houston. Dennis and Misunas: A Preliminary Architecture for a Basic Data-Flow Processor.
- **August 1975:** Sagamore Computer Conference on Parallel Processing:
  - Rumbaugh: Data Flow Languages
  - Rumbaugh: A Data Flow Multiprocessor
  - Bryant & Dennis: Packet Communication Architecture
  - Misunas: Structure Processing in a Data-Flow Computer

Eearly Roots on Dataflow Work

MIT in the 70's

- Asynchronous Digital Logic (D. E. Muller and Bartky, 1957; David E Muller, 1963)
- Control Structures for Parallel Programming (Conway, 1963; McIlroy, 1969; Dijkstra, 1971)
- Abstract Models for Concurrent Systems (Petri, 1966; Holt, 1978)
- Theory of Program Schemes (Ianov, 1958; Paterson and Hewitt, 1970)
- Structured Programming (Dahl, Dijkstra, and Hoare, 1972)
- Functional Programming (McCarthy, 1960; Landin, 1964)

Outline

Dataflow Graphs

Dataflow tokens

Values in dataflow graphs are represented as tokens. Tokens are tuples $< s, d, v >$, where s is the instruction pointer, d is the port, and v represents the actual value (data).

Operator Execution

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators.

Dataflow Graphs

```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```

Dataflow tokens

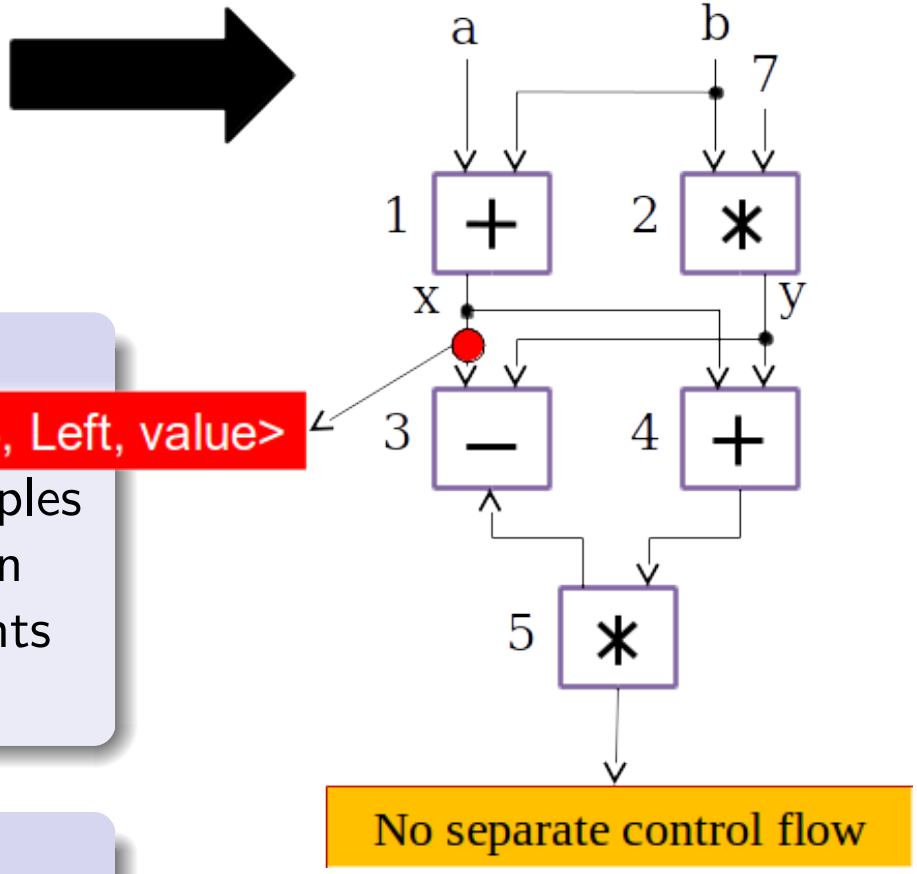
Values in dataflow graphs are represented as tokens. Tokens are tuples $< s, d, v >$, where s is the instruction pointer, d is the port, and v represents the actual value (data).

Operator Execution

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators.

Dataflow Graphs

```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



Dataflow tokens

Values in dataflow graphs are represented as tokens. Tokens are tuples $< s, d, v >$, where s is the instruction pointer, d is the port, and v represents the actual value (data).

Operator Execution

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators.

Overview of Operational Semantics

- Values are represented by tokens
- Tokens are placed (assigned) on the arcs
  - Snapshot/configuration: state
- Computation:
  - Configuration $\longrightarrow$ configuration

Details on Operational Semantics

- Tokens $\Leftrightarrow$ Data
- Assignment $\Leftrightarrow$ Placing a token on the output arc
- Snapshot/configuration: state
- Computation
  - The intermediate step between snapshots / configurations

Actors States

- An actor in a DFG is enabled if there is a token on each of its input arcs
- Any enabled actor may be fired to define the "next state" of the computation
- When an actor is fired, then:
  - All of its input tokens are removed from the input arcs
  - A token is placed on each of the actor's output arcs
- Computation $\Leftrightarrow$ Sequence of snapshots
  - Many possible sequences, as long as firing rules are obeyed
  - This is called determinacy
  - Provides "locality of effect"

General Firing Rules

- A switch actor is enabled if:
  - A token is available on its control input arc
  - A token is available on its data input arc(s)
- Firing a switch actor:
  - Removes input tokens
  - Delivers the input data value as an output token on the corresponding output arc
- A (possibly unconditional) merge actor is enabled if
  - There is a token available on any of its input arcs
  - A conditional merge need to have a control token placed on its input control arc
- An enabled (unconditional) merge actor may be fired
  - It will put (possibly non-deterministically) one of the input tokens on the output arc.

Conditional Expressions

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

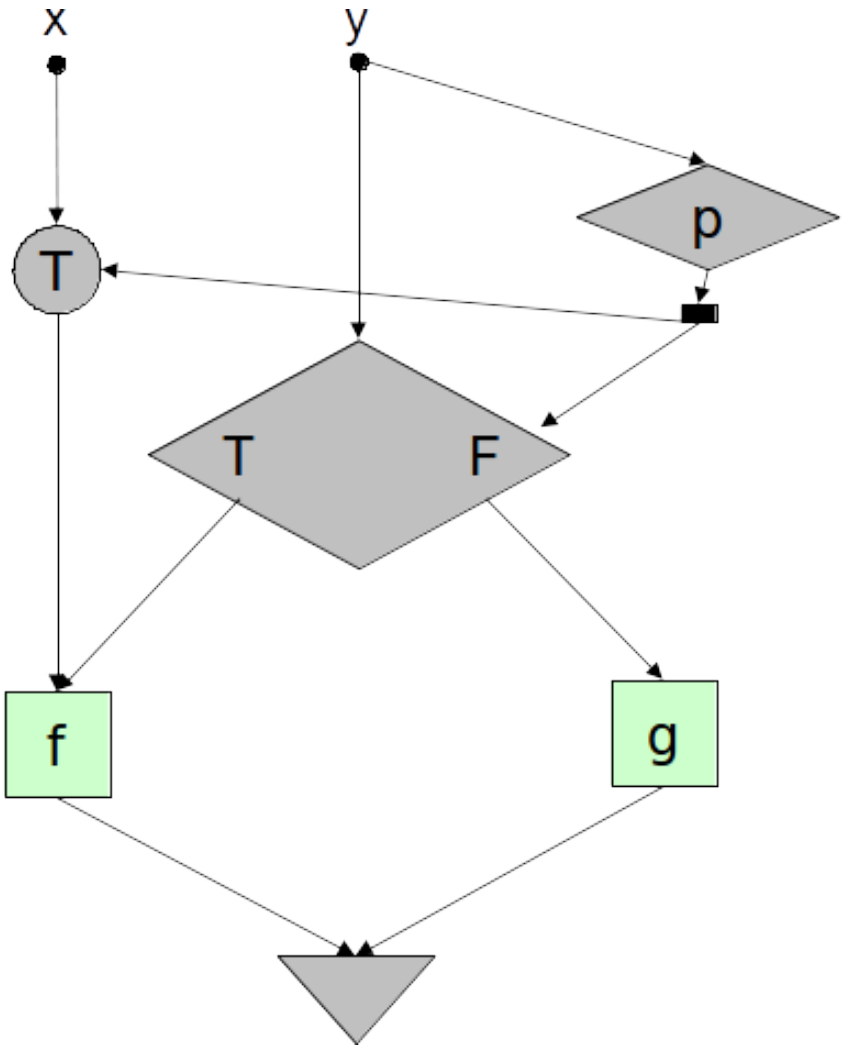


Figure: `if/else` construct

Conditional Expressions

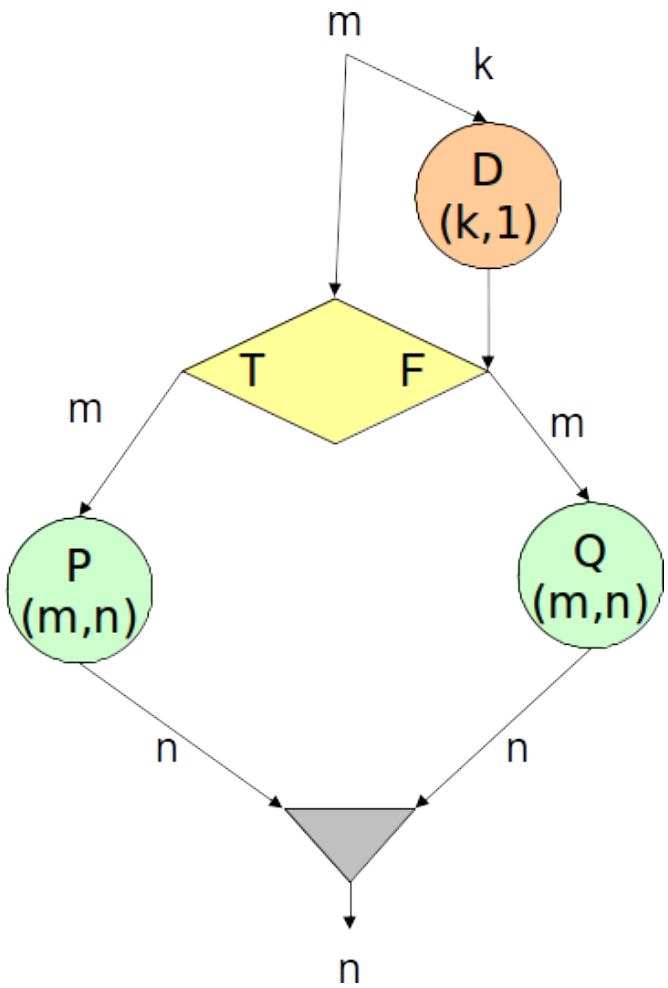


Figure: `if/else` schema

Conditional Expressions

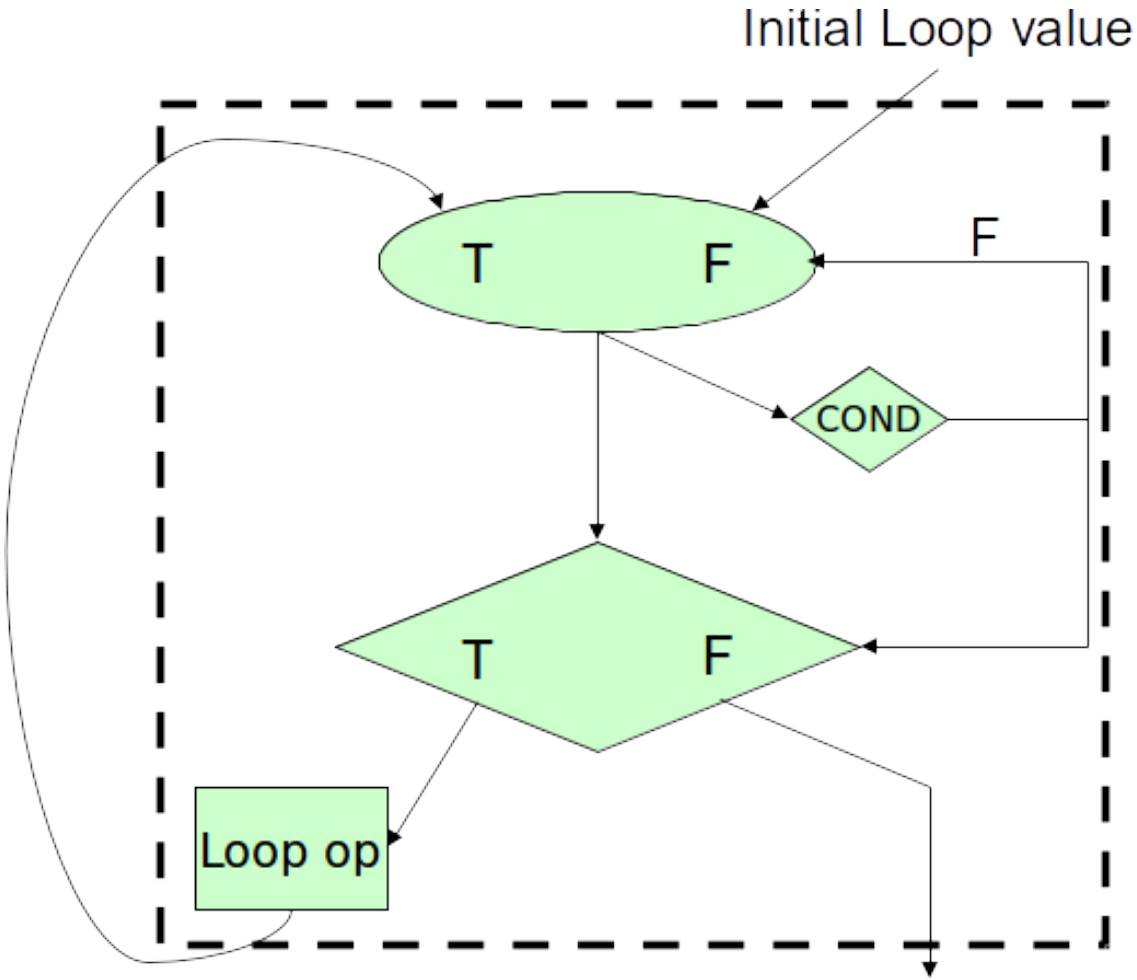


Figure: `loop` schema

Remarks on Dataflow Models

- The dataflow model of computation is fundamentally sound and simple
  - Very few other parallel MoCs can claim this
- Few dataflow architecture projects survived past the early 1990's.
  - However, dataflow principles were heavily reused in both software and hardware
- With the advent of the new multi and many core era, we have many (good!) reasons to re-examine and explore the original dataflow models
  - It never hurts to learn from the past!
- As a side-note: Jack Dennis's dataflow model was finally recognized on a world-scale level: he was awarded the John Von Neumann Medal (the IEEE equivalent to the ACM Turing Award)

Outline

Static Dataflow

- Static dataflow was the first dataflow model
- It was developed in Jack Dennis' team

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

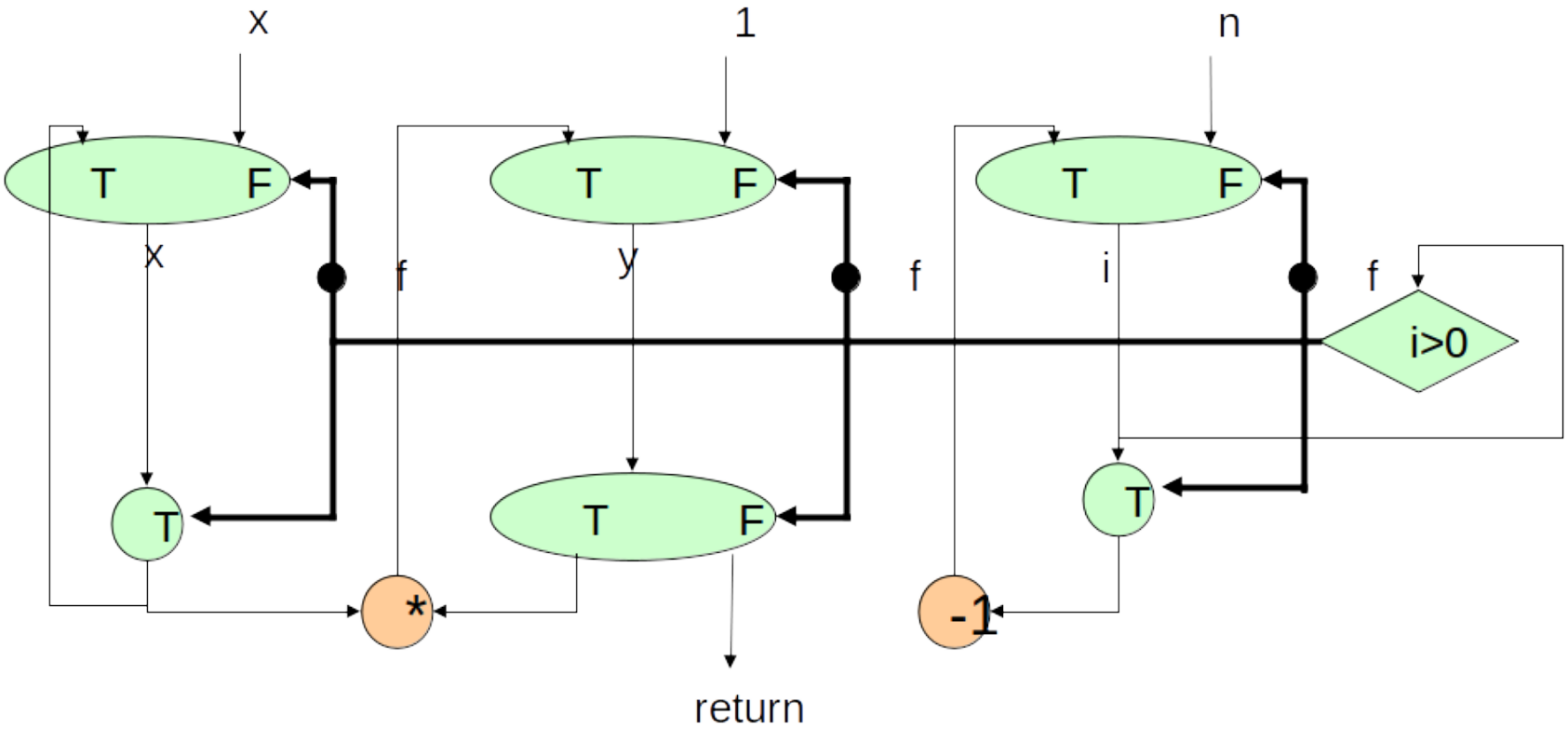
Power Function

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

Figure: Computation: $y = x^n$

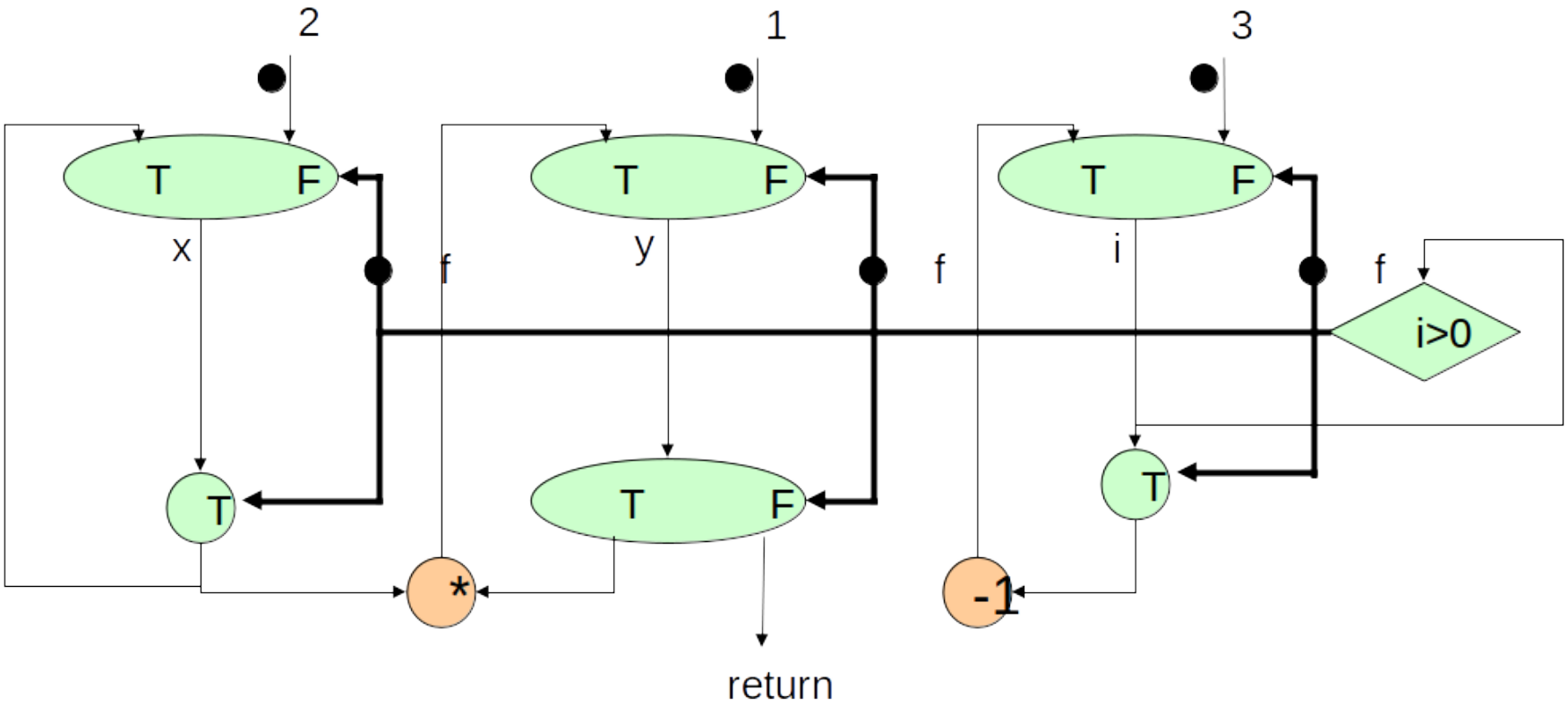
Example

Power Function – Computing $y = 2^3$



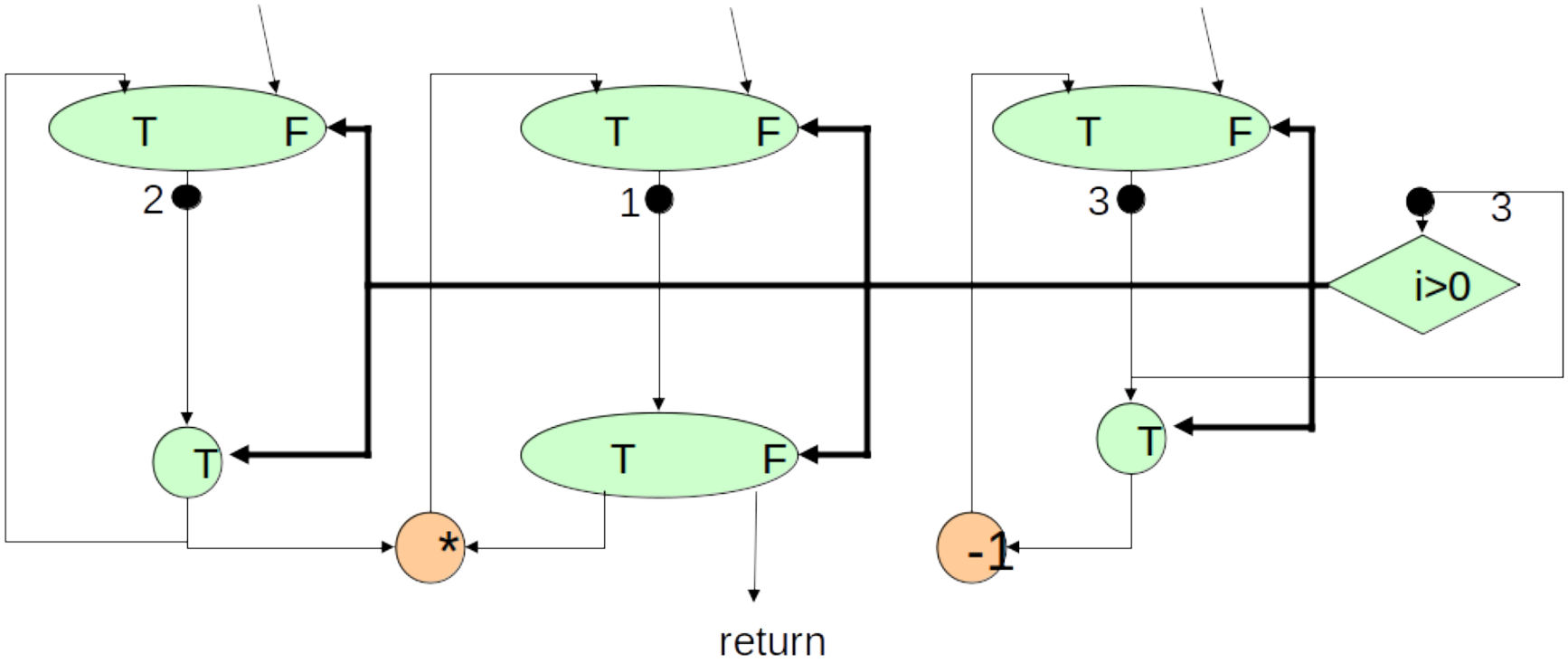
Example

Power Function – Computing $y = 2^3$



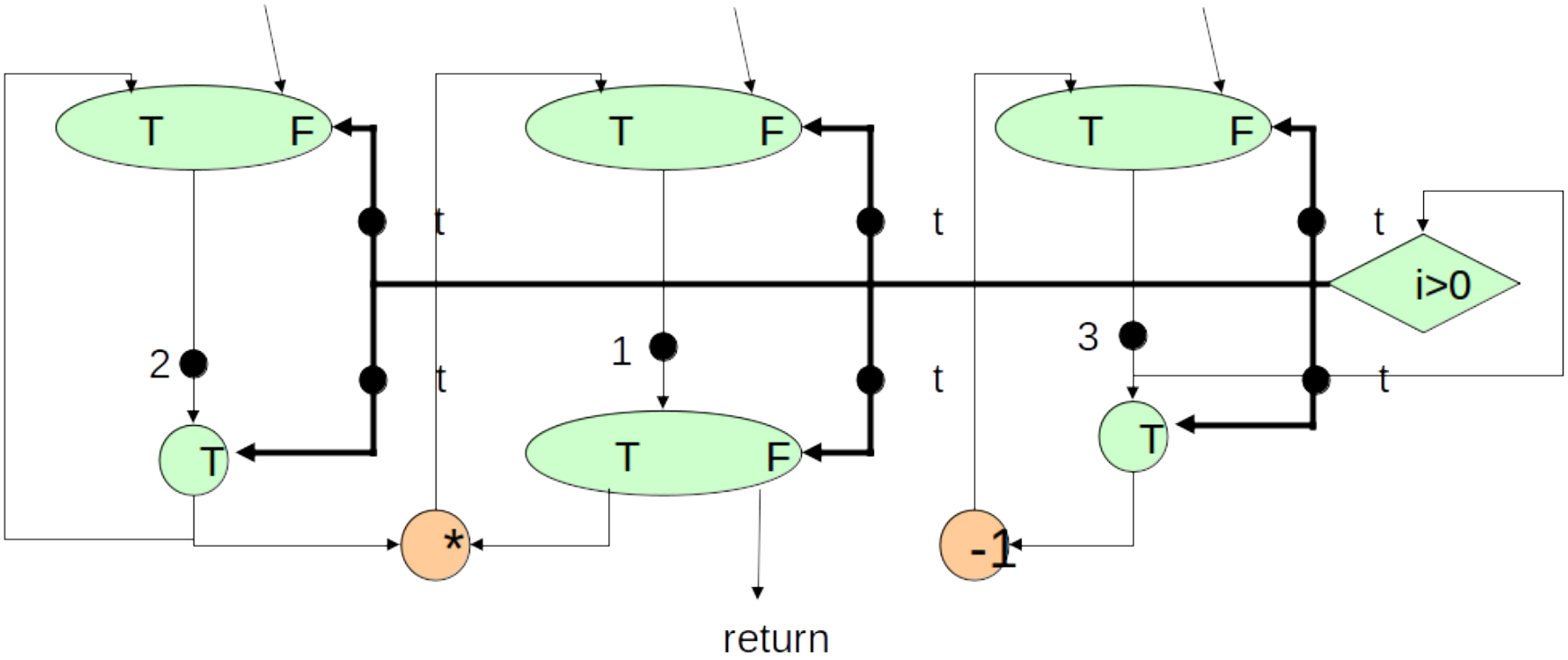
Example

Power Function – Computing $y = 2^3$



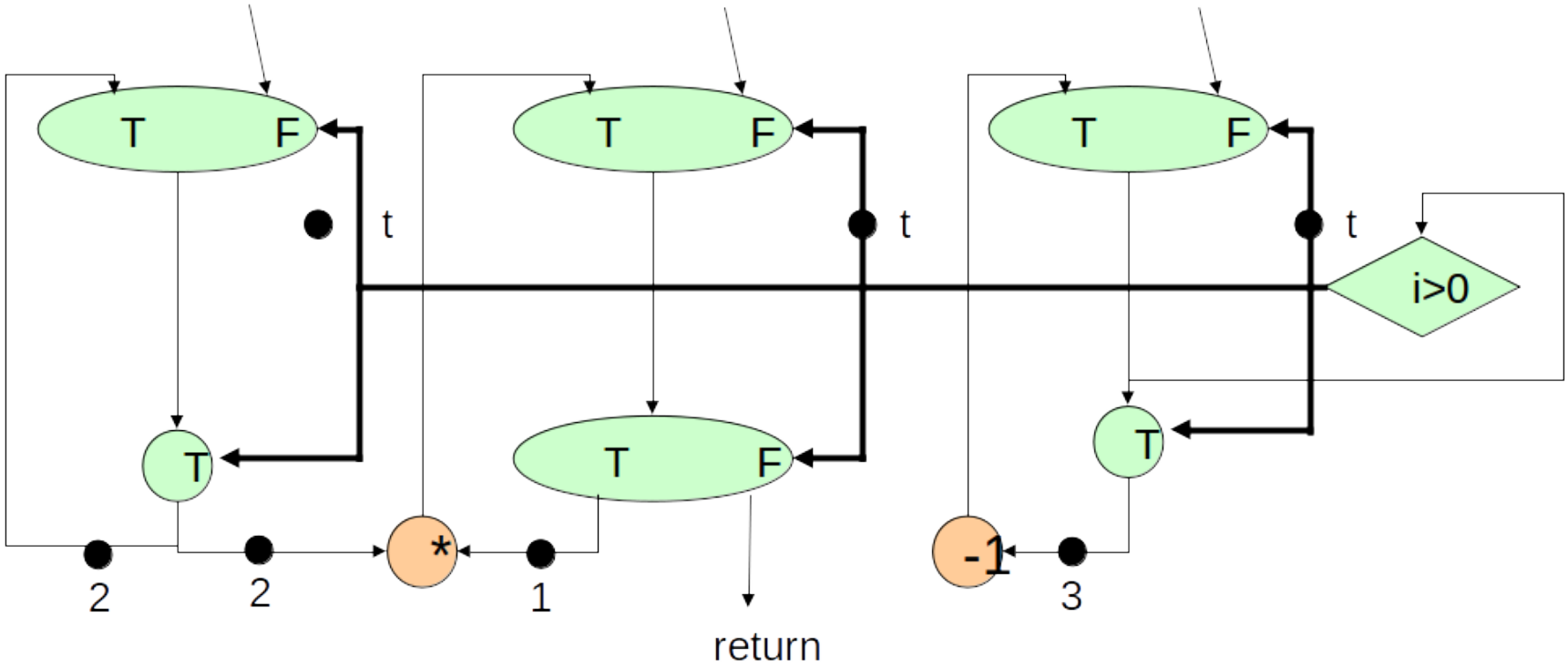
Example

Power Function – Computing $y = 2^3$



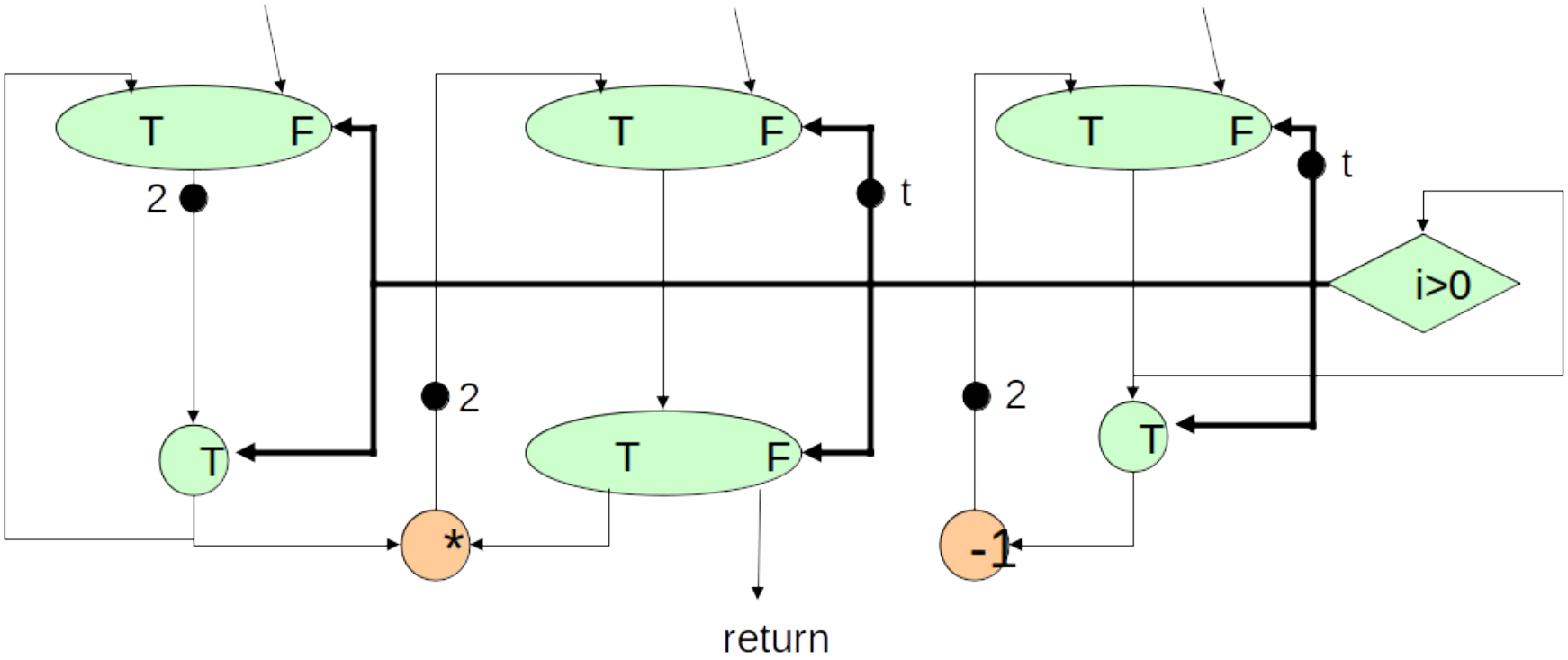
Example

Power Function – Computing $y = 2^3$



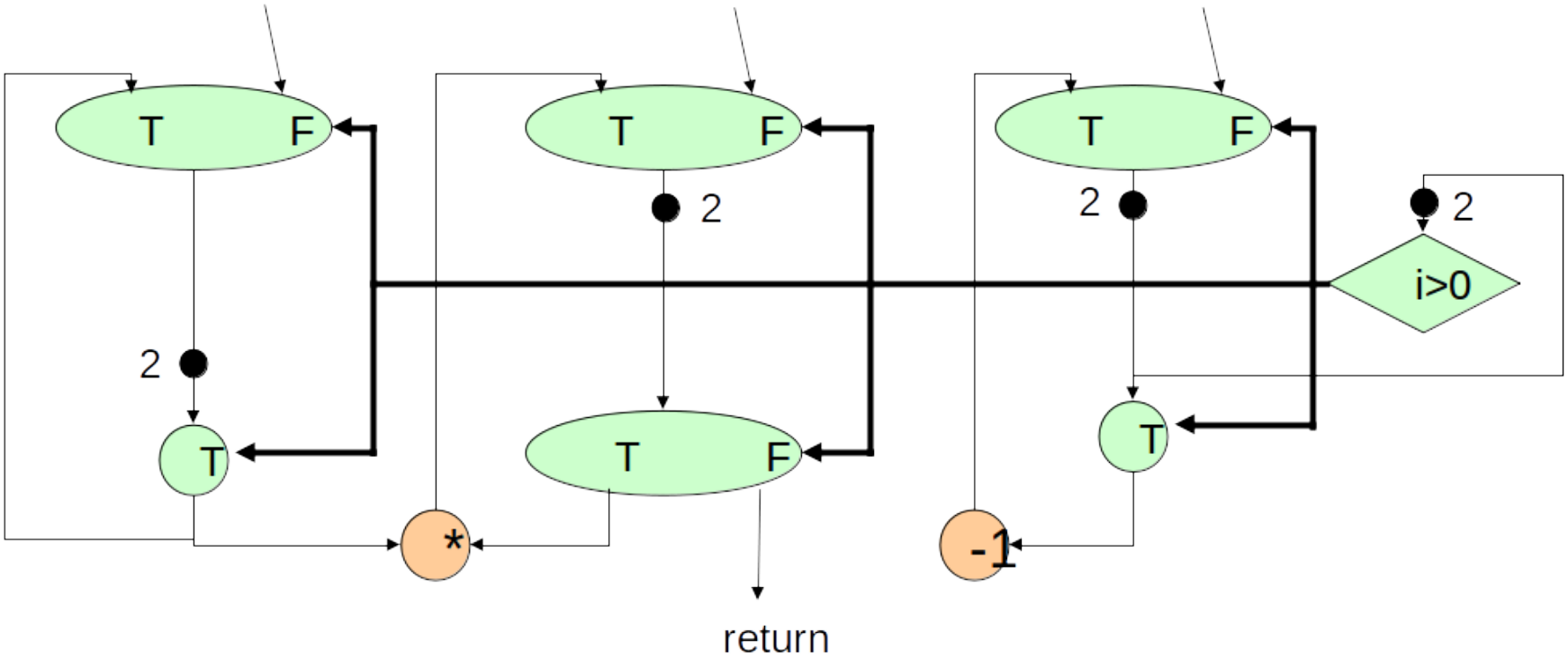
Example

Power Function – Computing $y = 2^3$



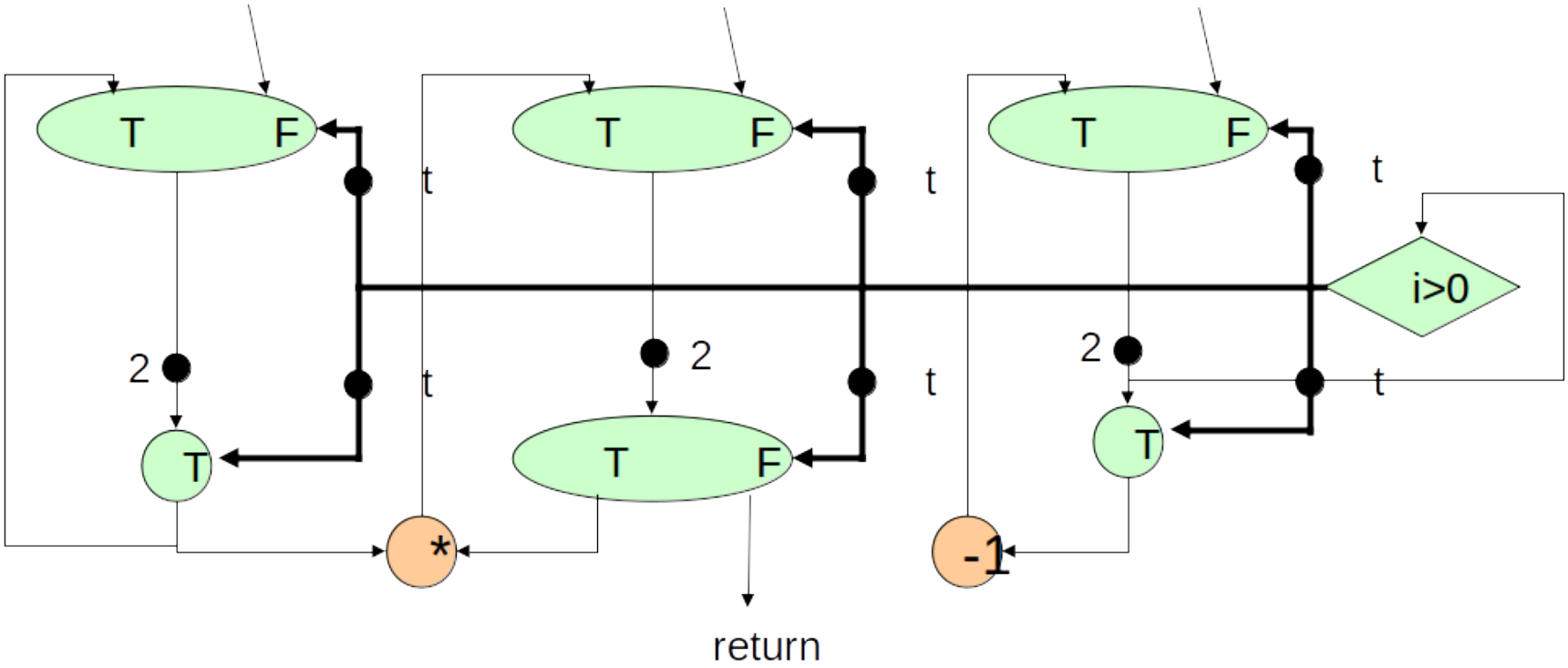
Example

Power Function – Computing $y = 2^3$



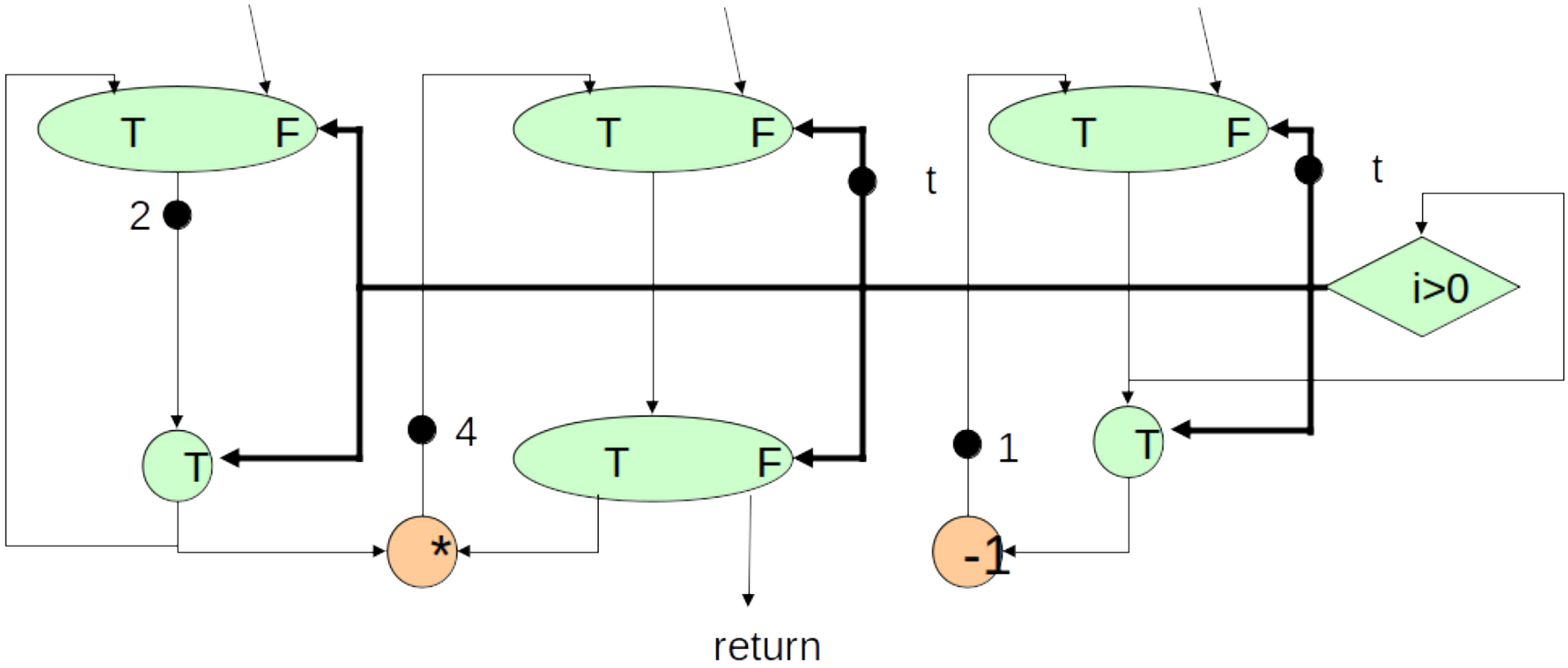
Example

Power Function – Computing $y = 2^3$



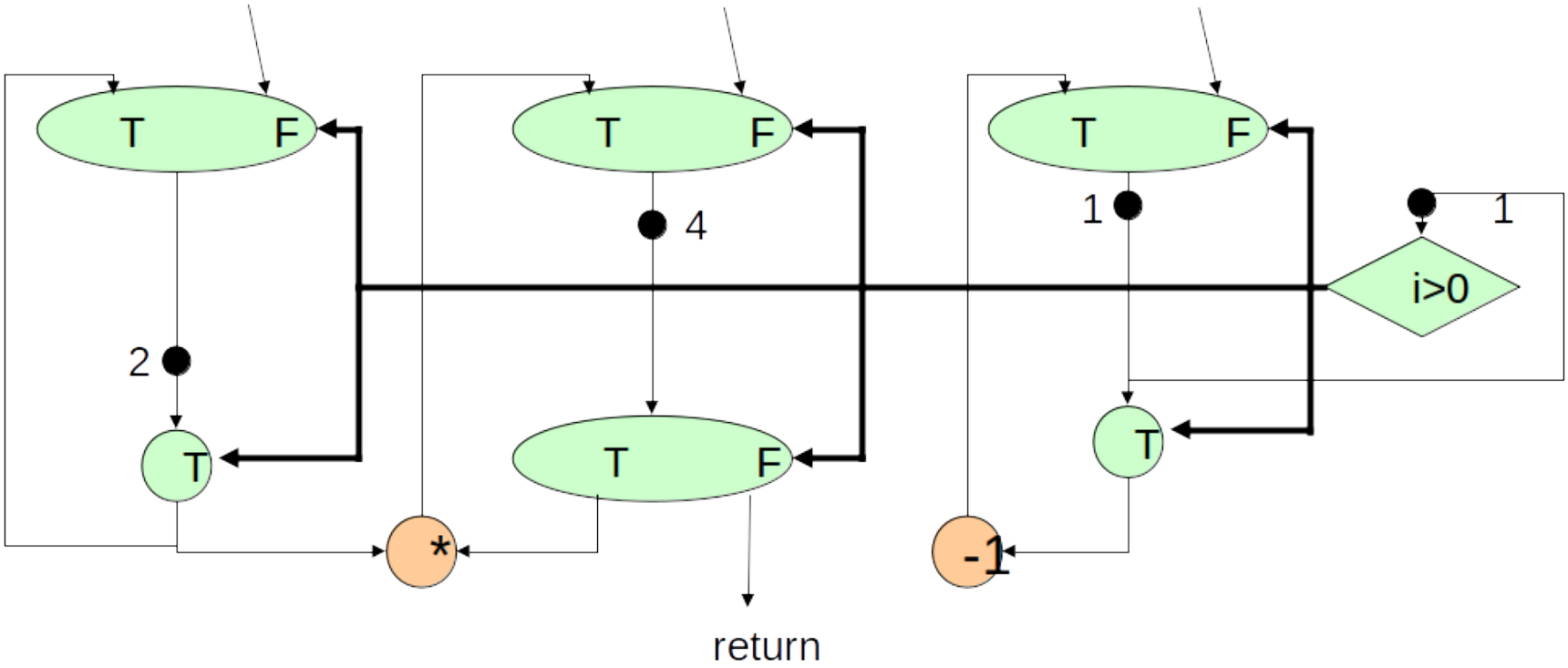
Example

Power Function – Computing $y = 2^3$



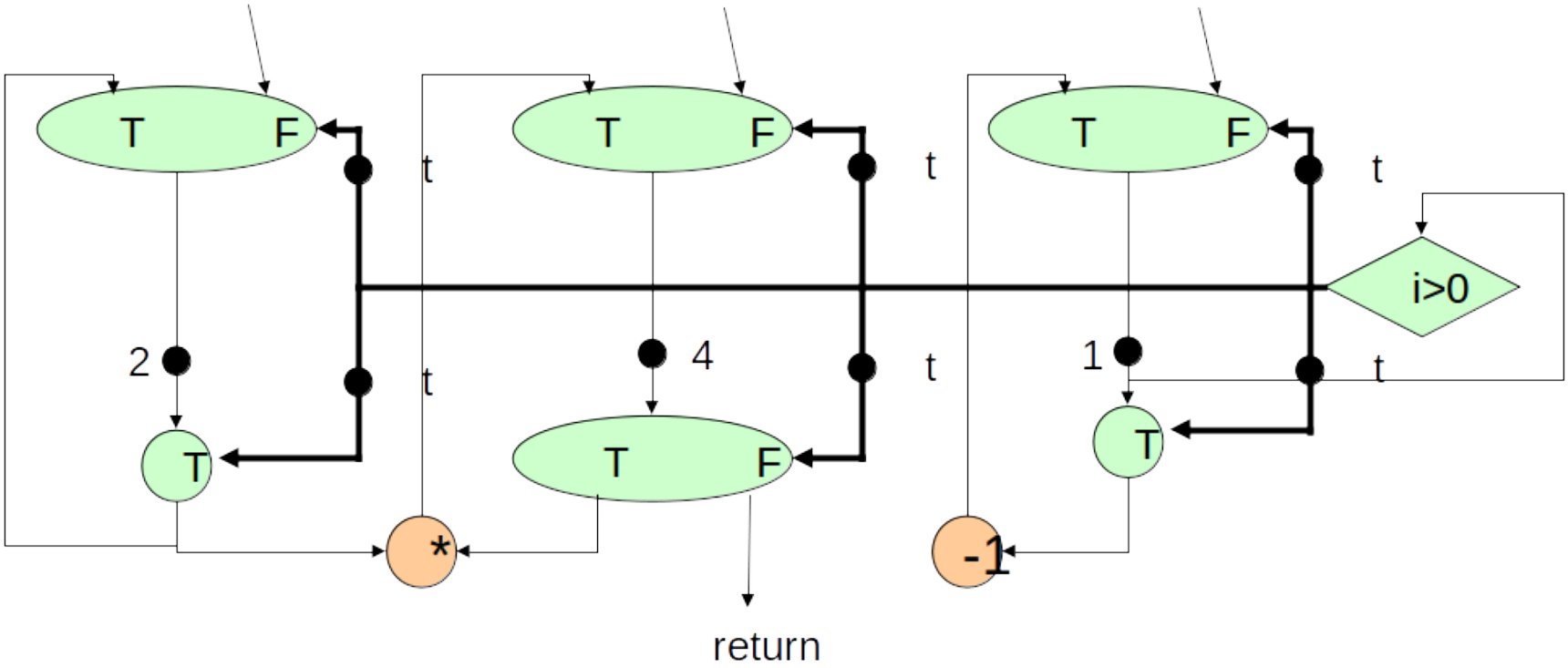
Example

Power Function – Computing $y = 2^3$



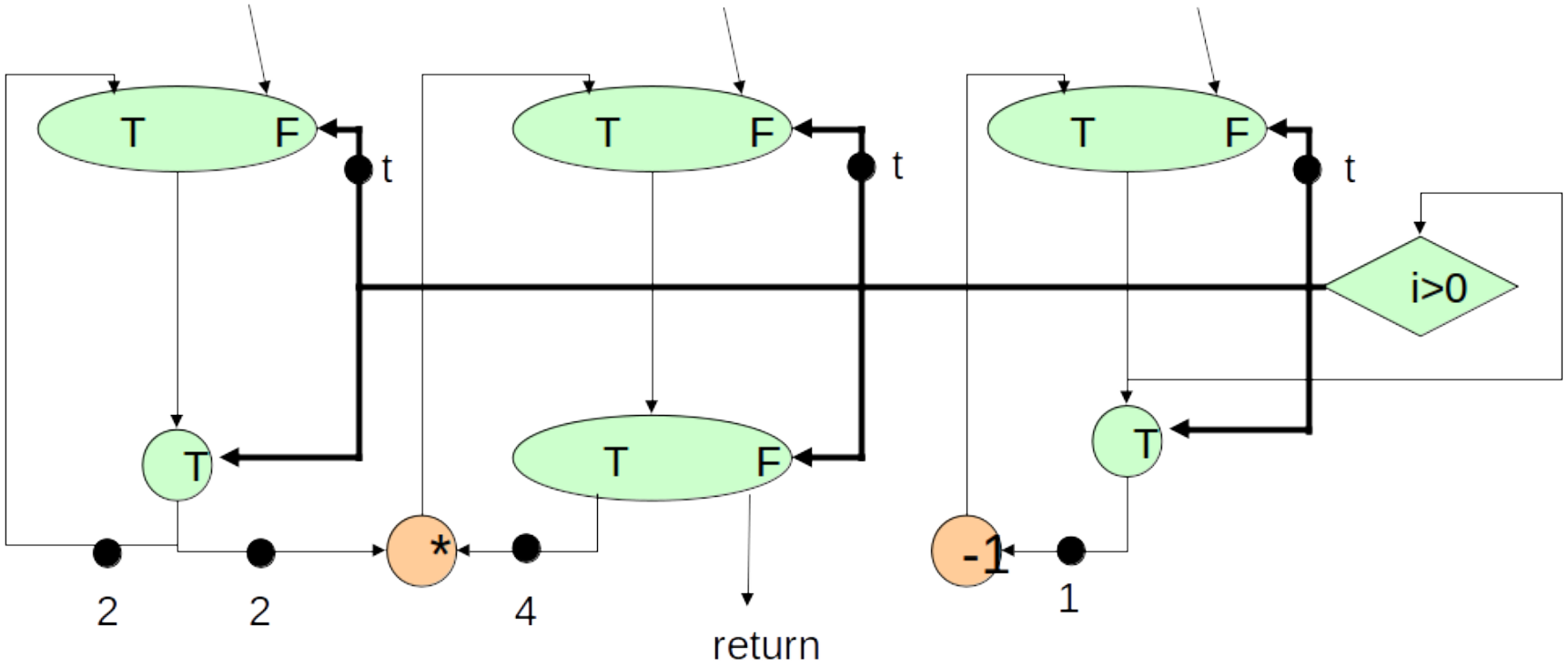
Example

Power Function – Computing $y = 2^3$



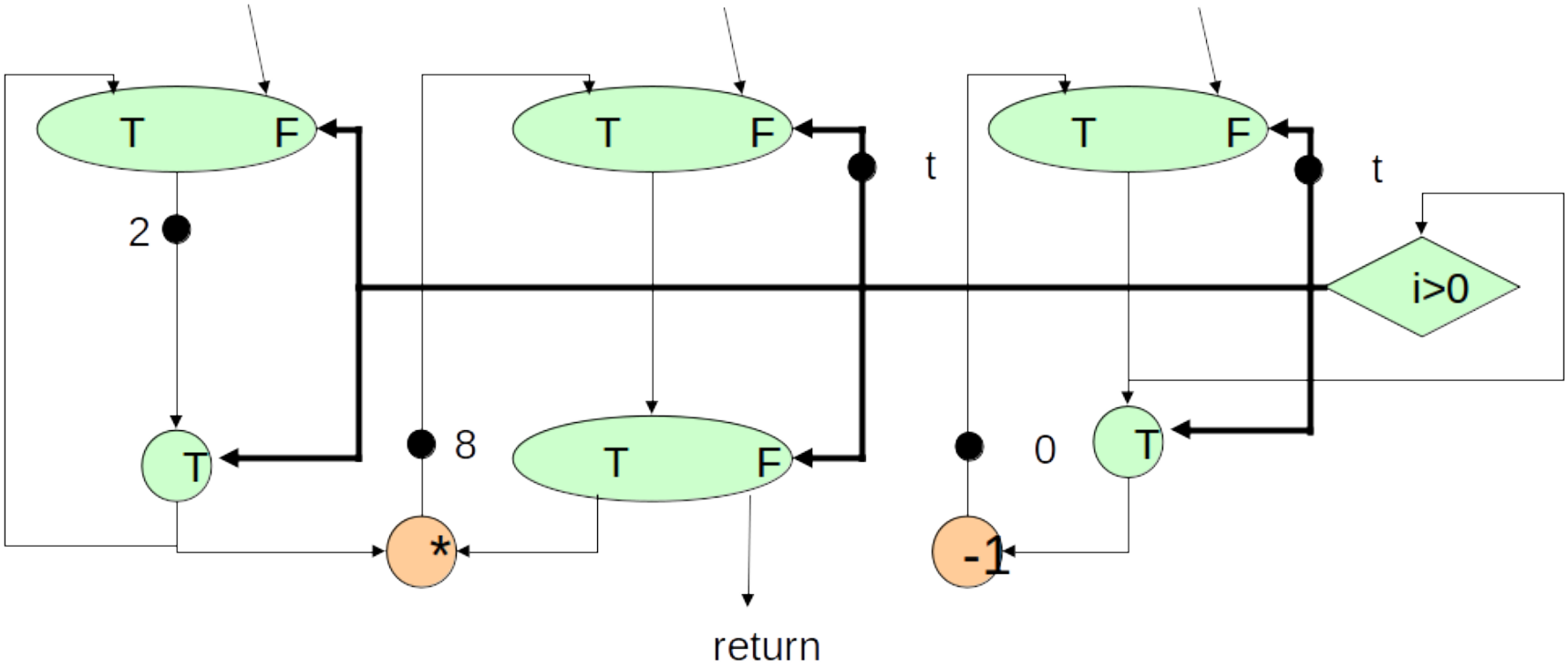
Example

Power Function – Computing $y = 2^3$



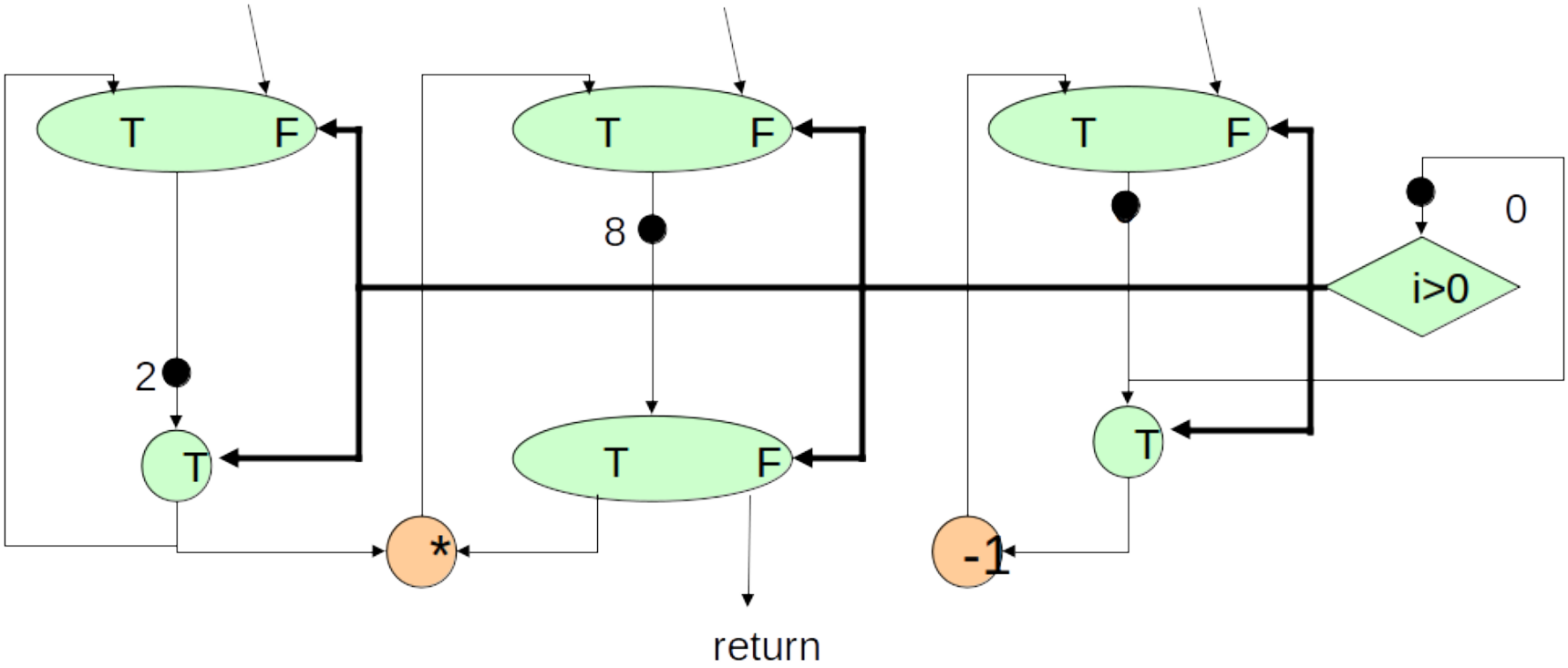
Example

Power Function – Computing $y = 2^3$



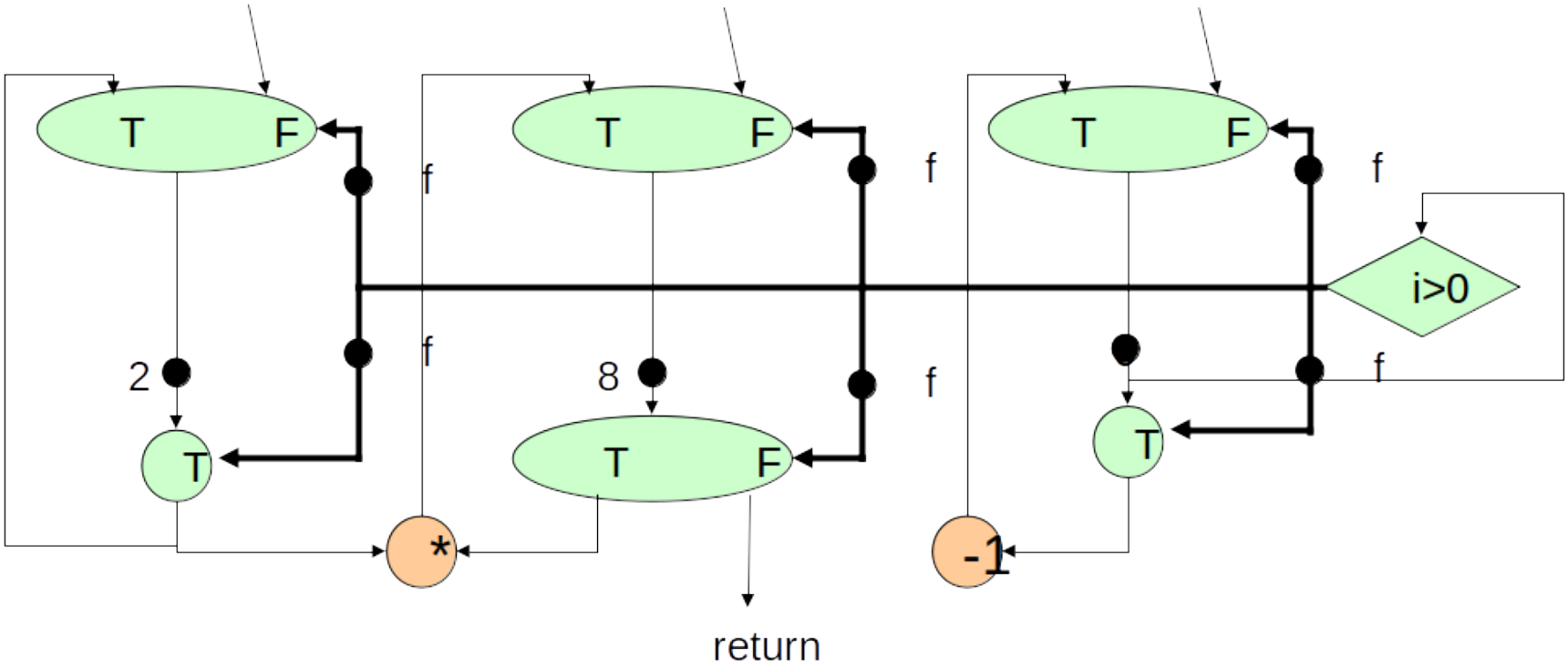
Example

Power Function – Computing $y = 2^3$



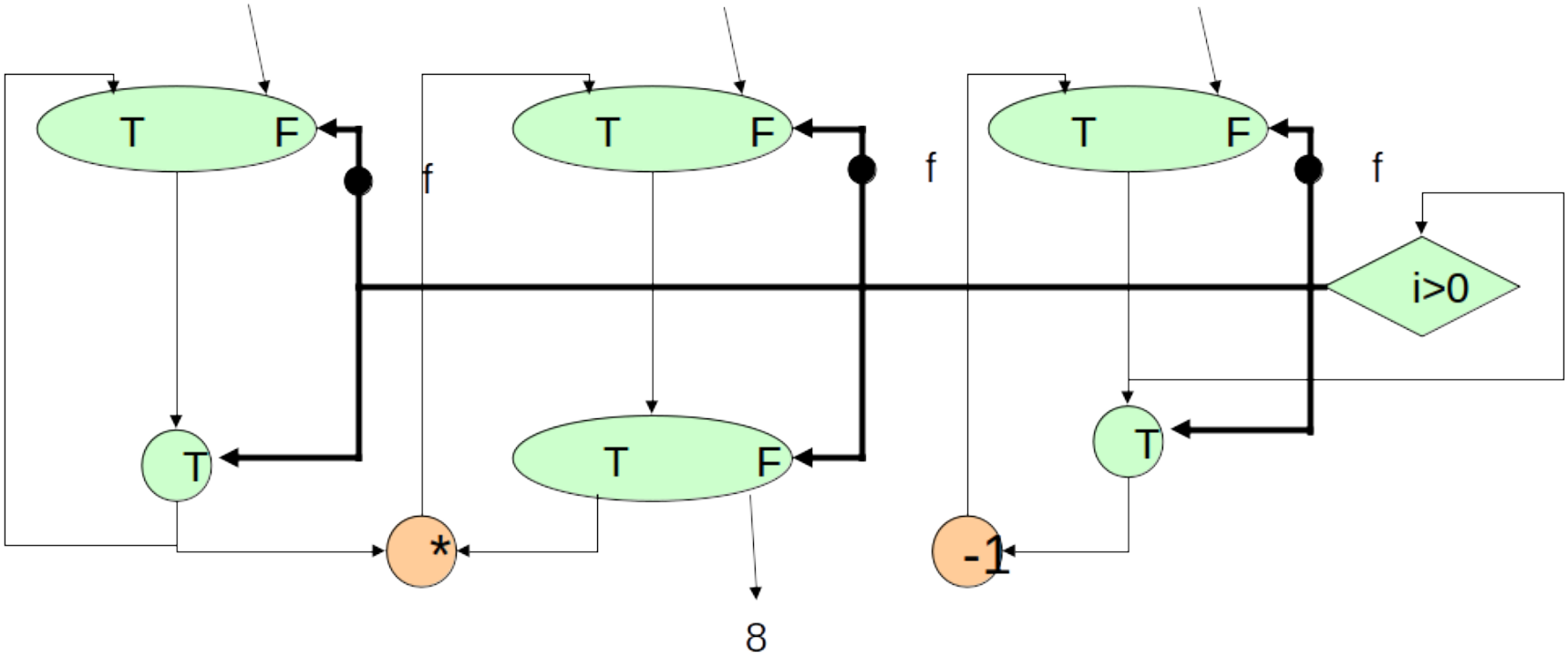
Example

Power Function – Computing $y = 2^3$



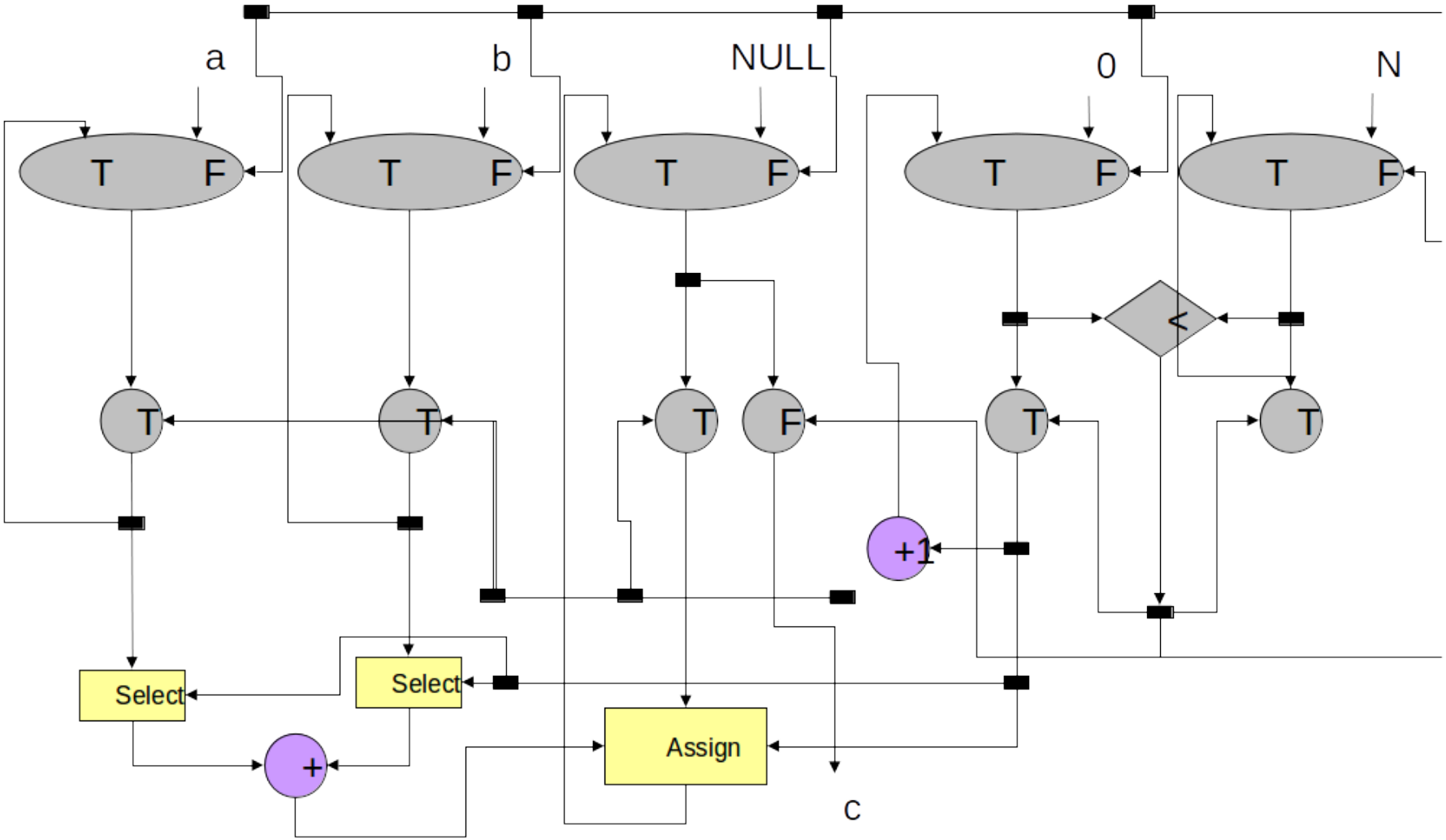
Example

Power Function – Computing $y = 2^3$



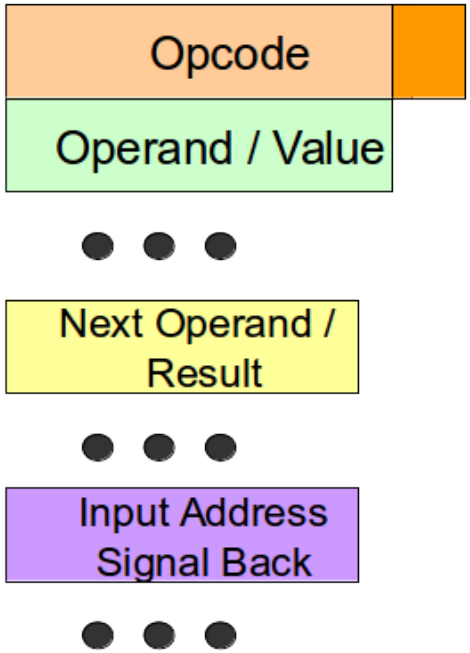
Example

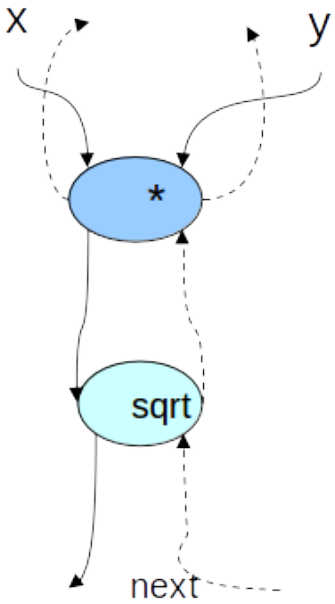
Vector Addition: $\vec{C}_N = \vec{A}_N + \vec{B}_N$

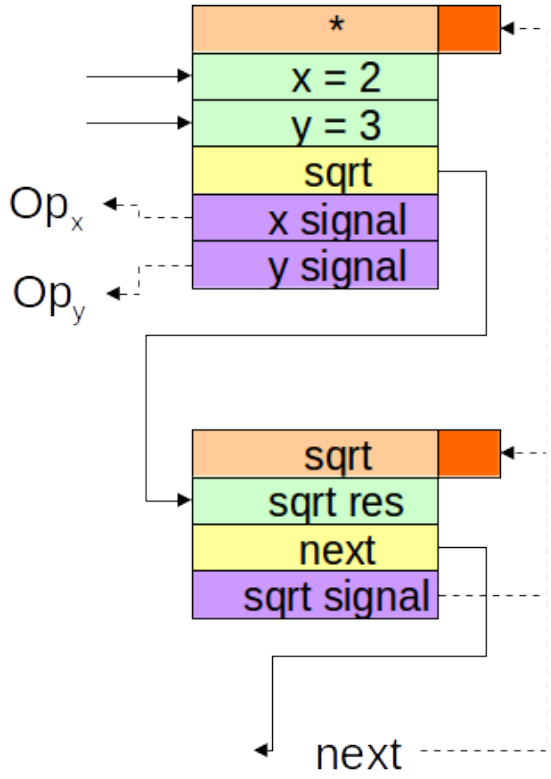


- One-token-per-arc
- Deterministic merge
- Conditional/iteration construction
- Consecutive iterations of a loop can only be pipelined
- A dataflow graph $\Rightarrow$ activity templates
  - Opcode of the represented instruction
  - Operand slots for holding operand values
  - Destination address fields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

Static Dataflow
Activity Templates



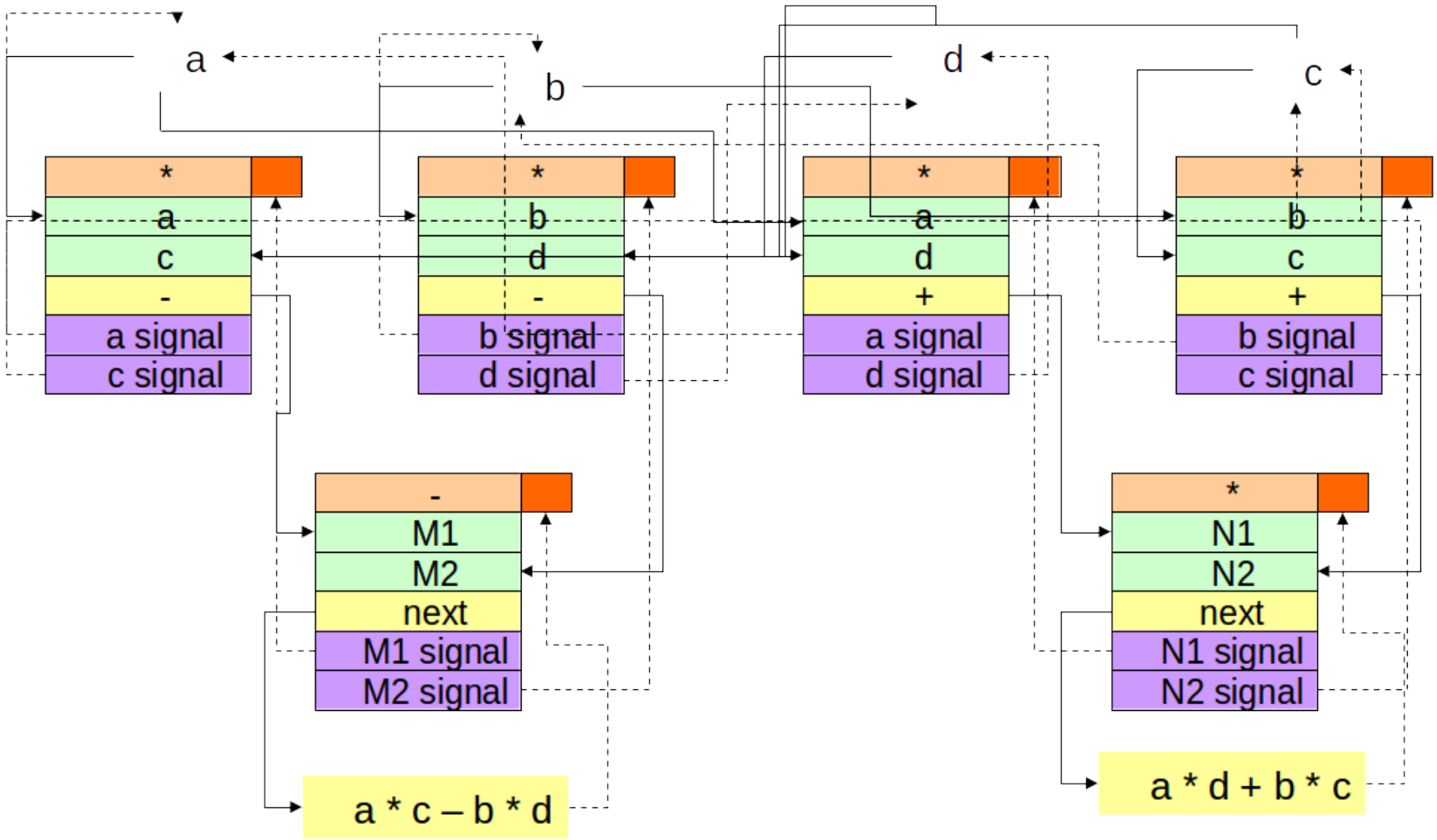




→	Token Arc
⇢	Communication Arc
Op_x / Op_y	The operation that produced x and y
next	The operation that will use the sqrt result

Static Dataflow

Activity Templates



The static dataflow model has the advantage of being very simple. However, it is also fairly limited:

- Due to acknowledgment tokens, the token traffic is doubled
- Lack of support for programming constructs that are essential to modern programming languages
- No procedure calls
- No recursion

Outline

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);
}
```

Examples of Tail Recursion

Factorial

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

long factorial(long n)
{
    return fact_tail_rec(n,1);
}
```

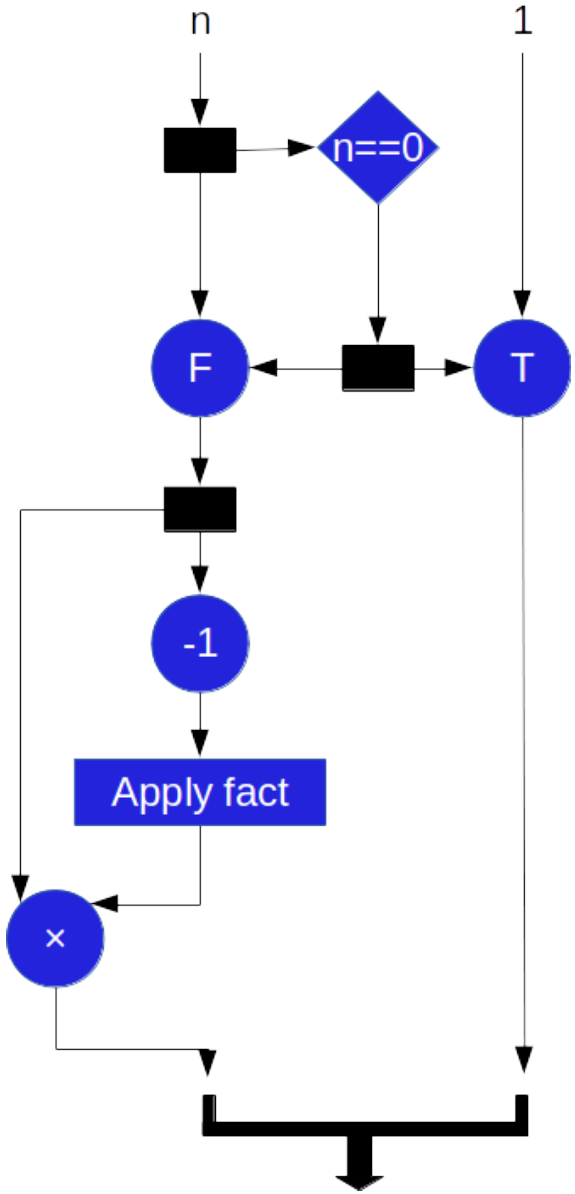
Example – Factorial

Hand Simulation: fact(3)

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

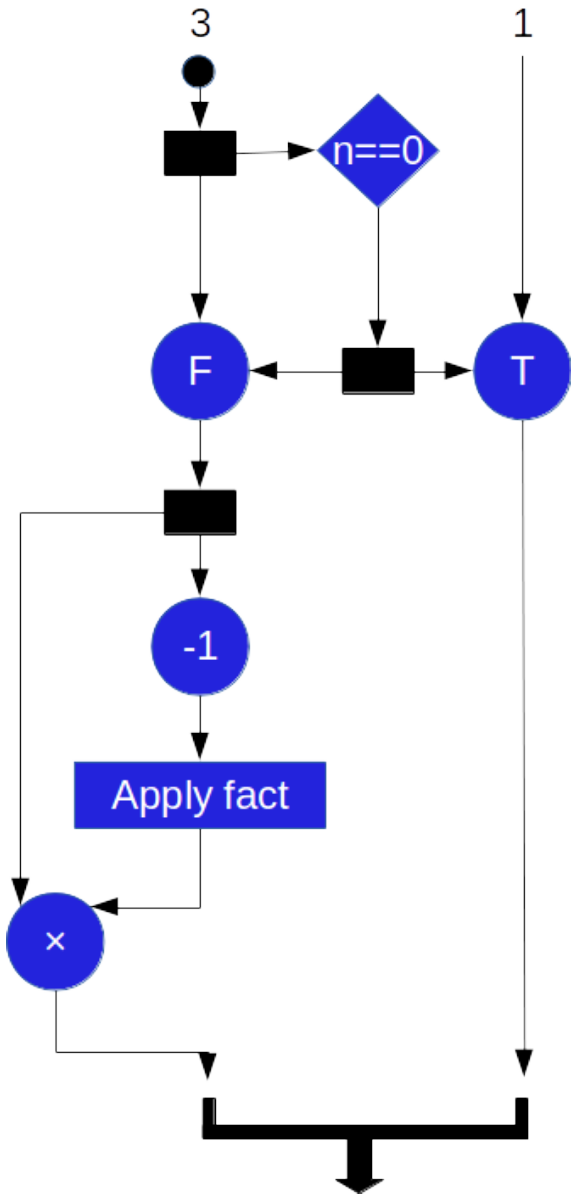
Example – Factorial

DFG for fact(3)



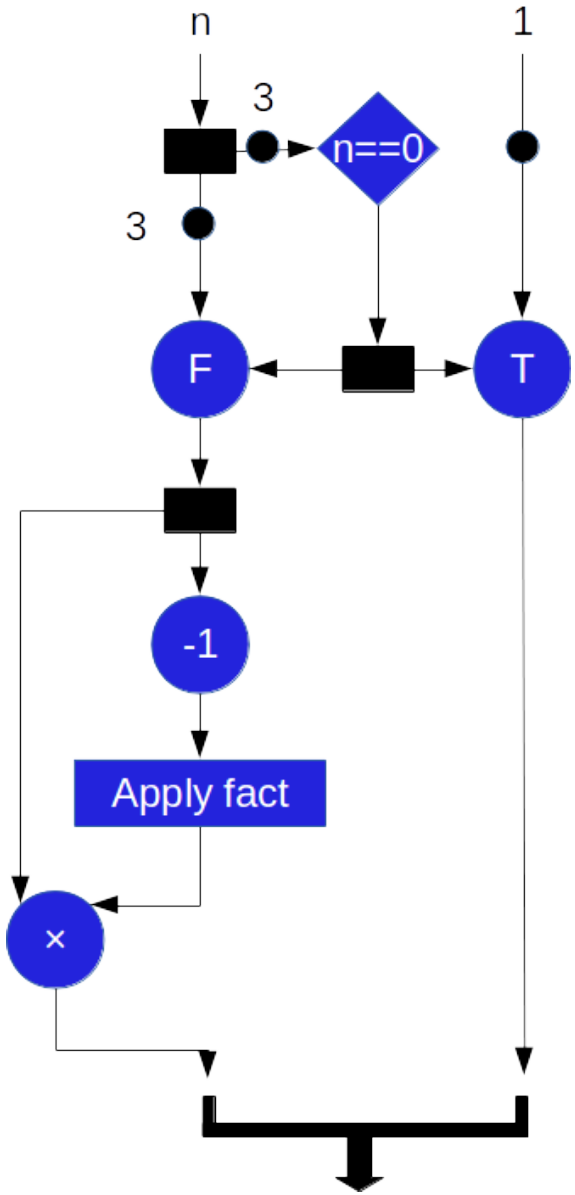
Example – Factorial

DFG for fact(3)



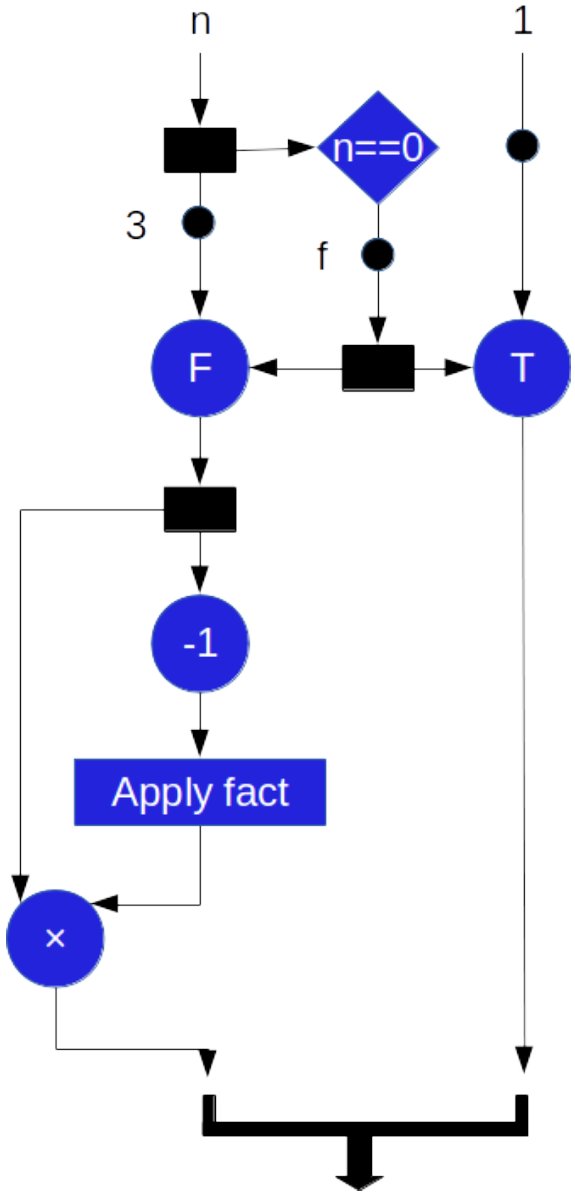
Example – Factorial

DFG for fact(3)



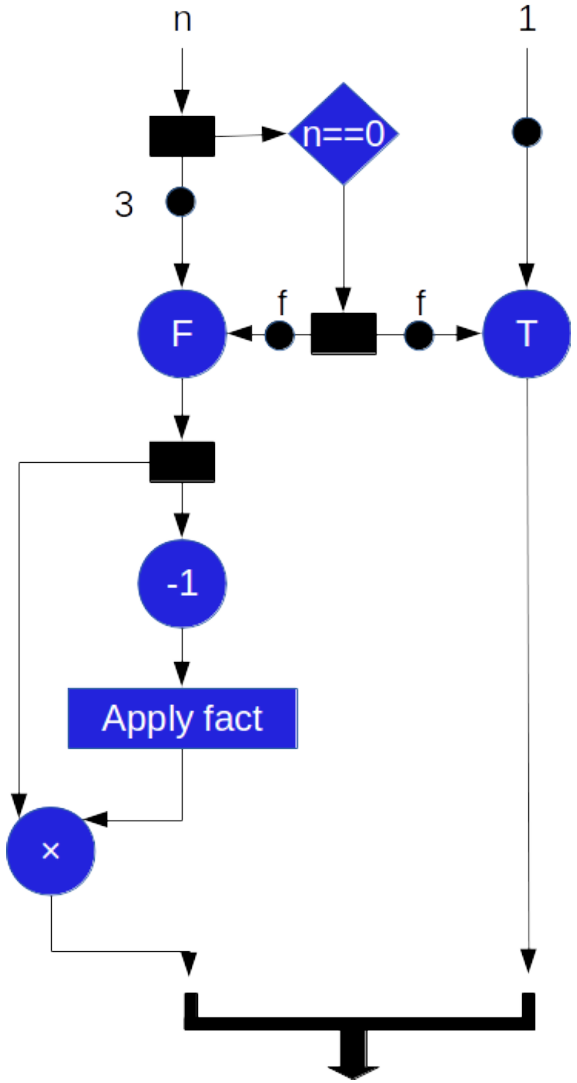
Example – Factorial

DFG for fact(3)



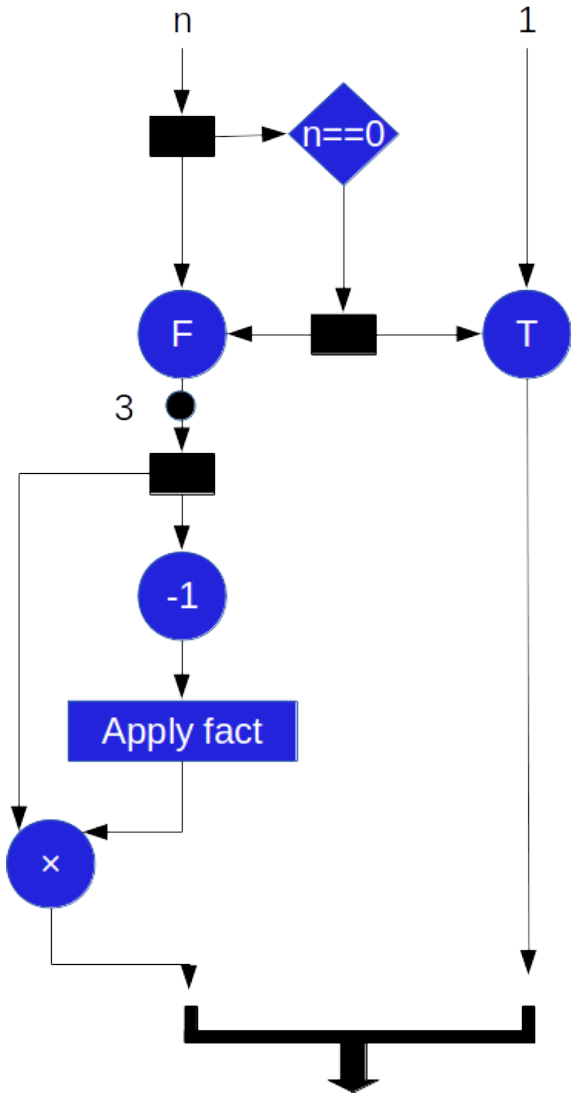
Example – Factorial

DFG for fact(3)



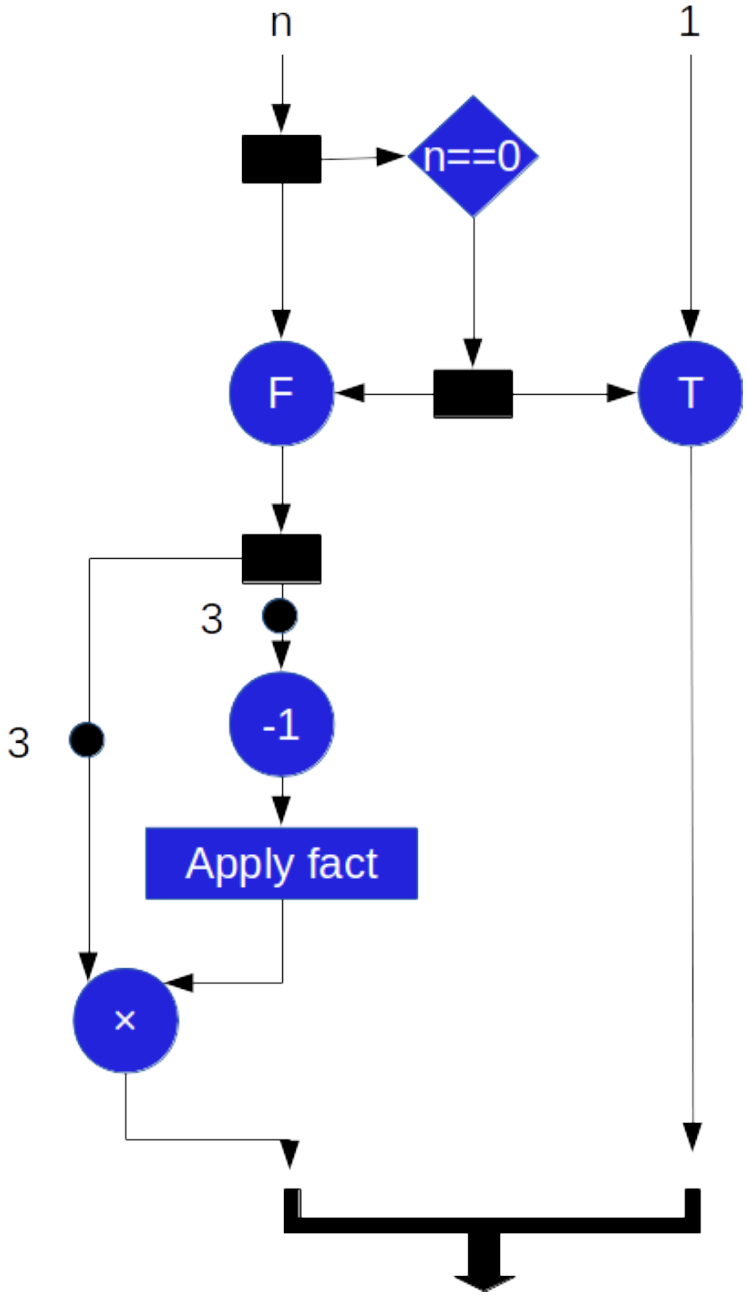
Example – Factorial

DFG for fact(3)



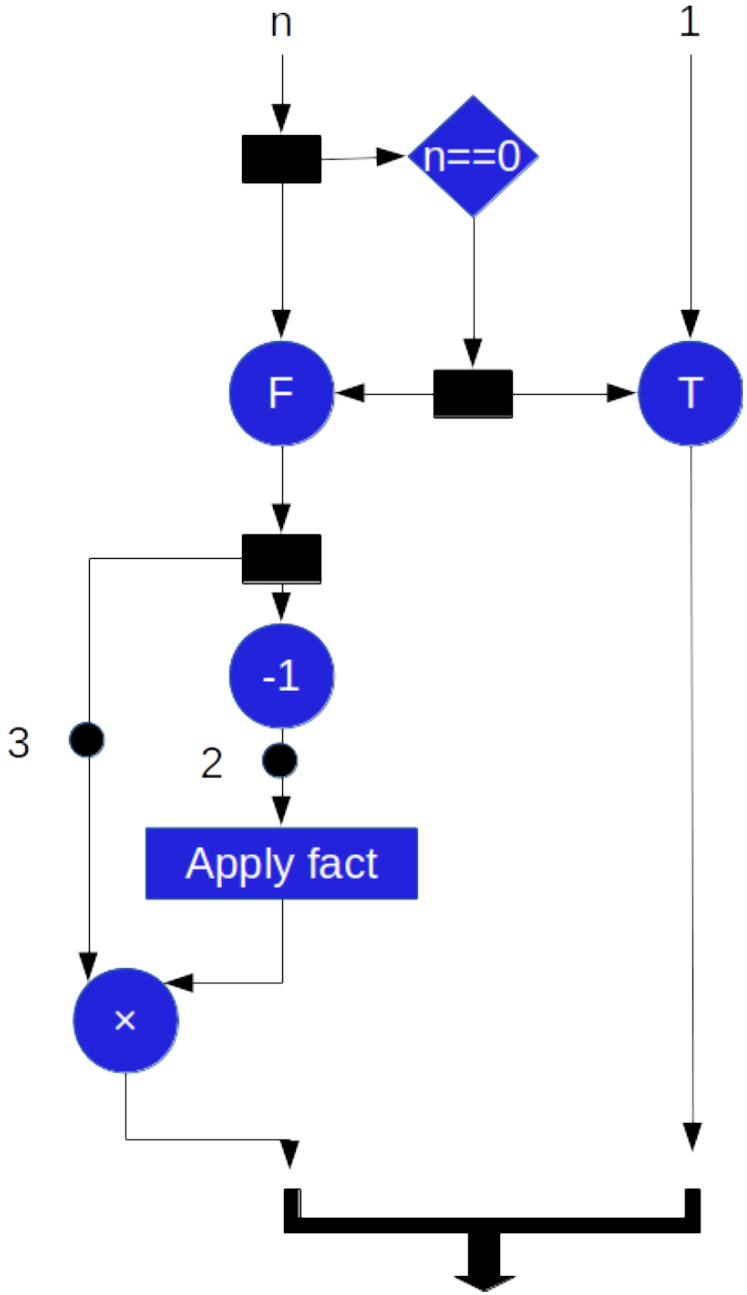
Example – Factorial

DFG for fact(3)



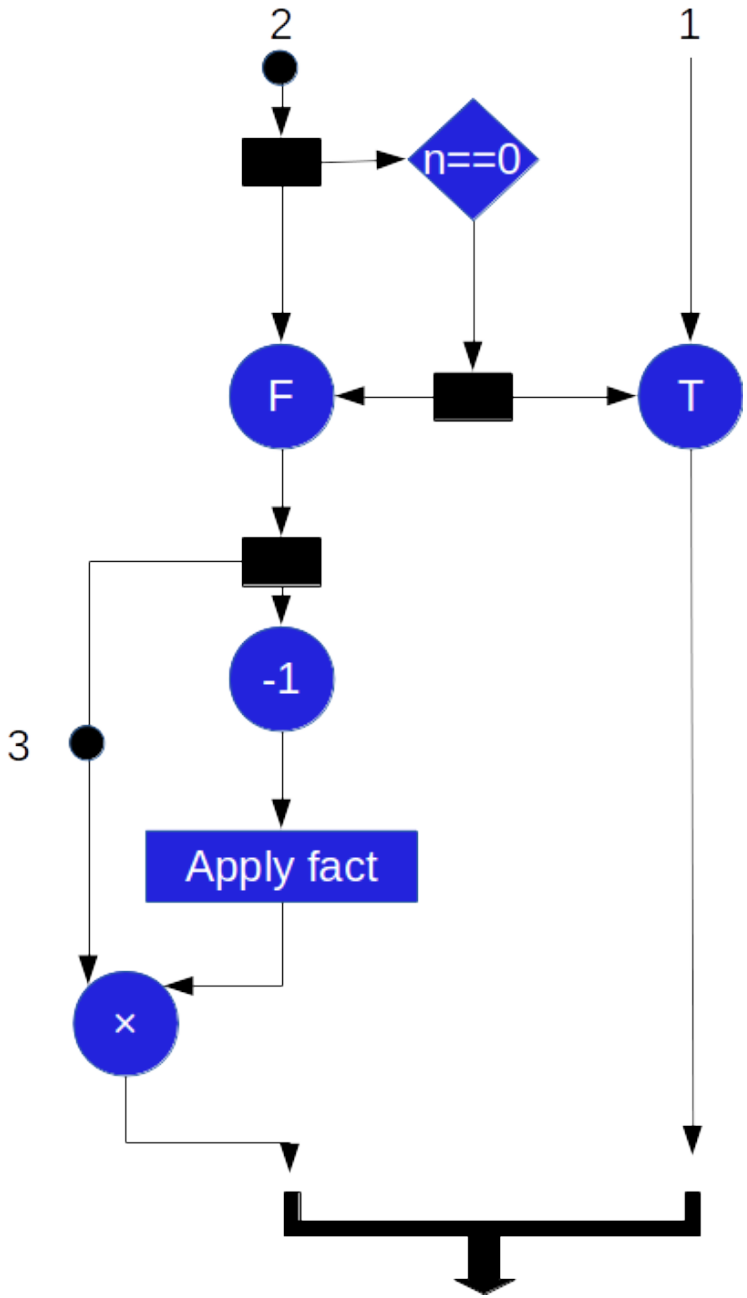
Example – Factorial

DFG for fact(3)



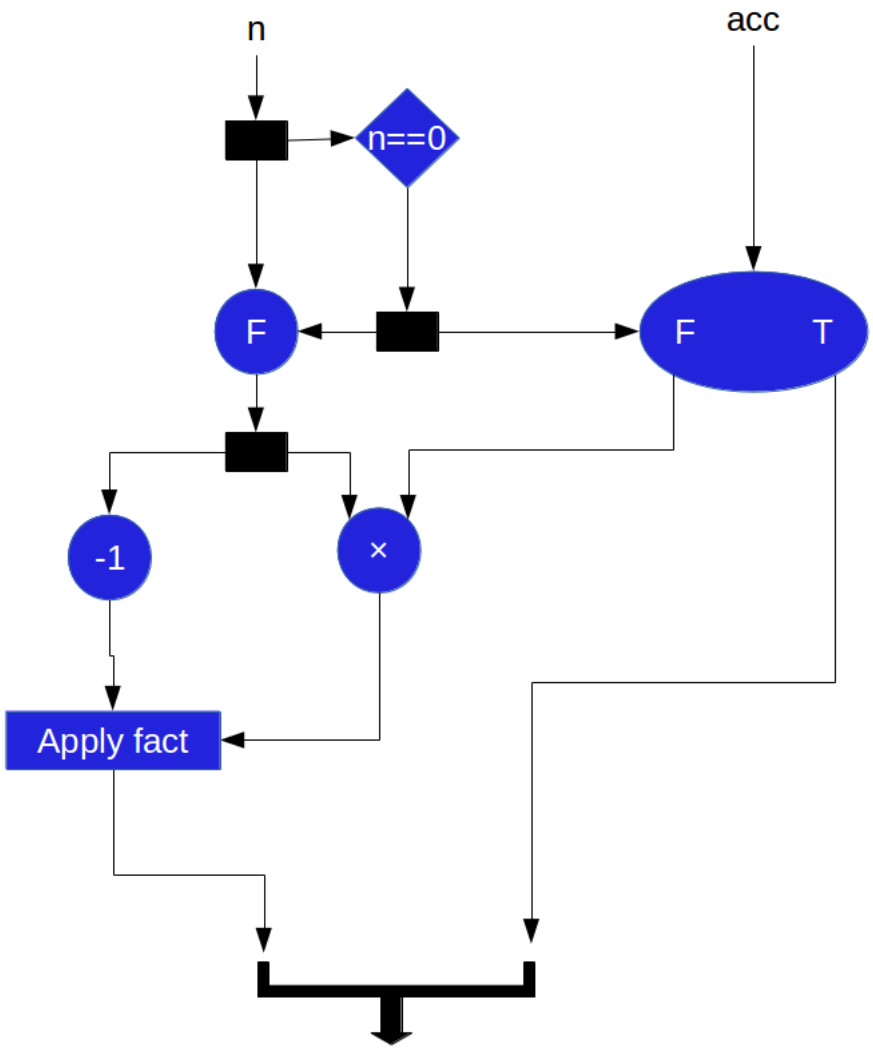
Example – Factorial

DFG for fact(3)



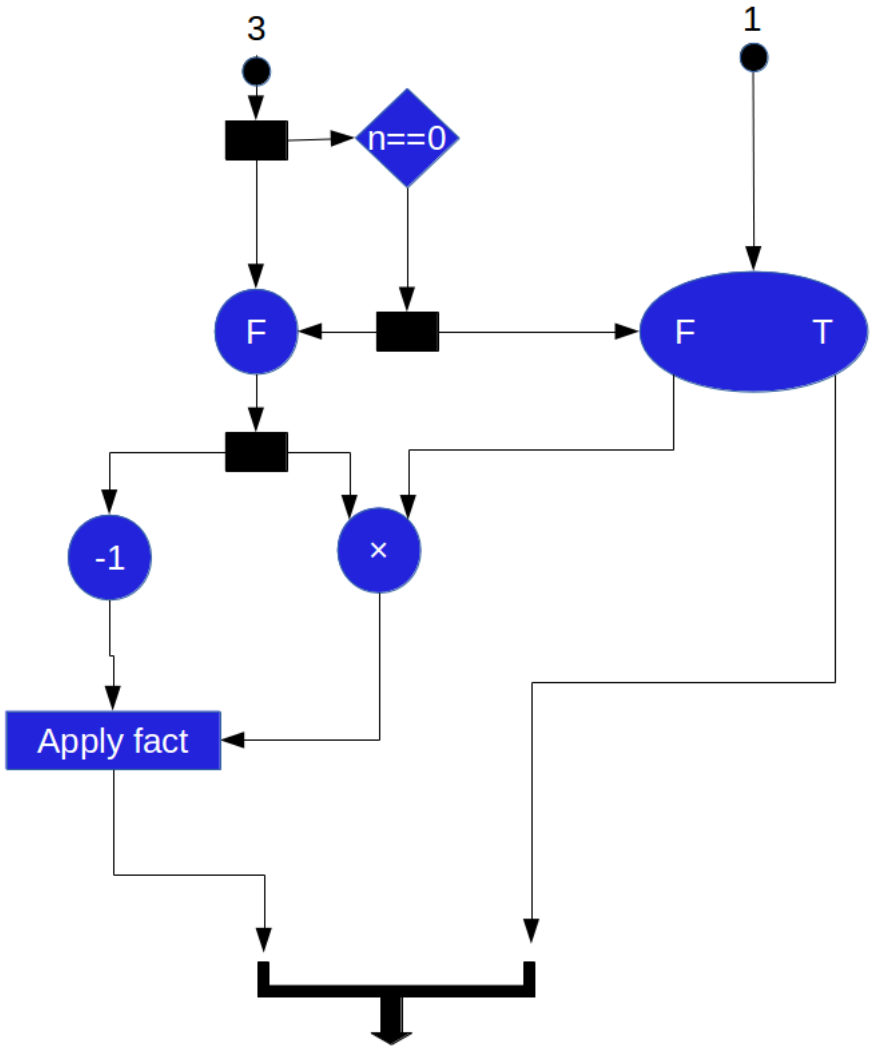
Factorial

Tail Recursive version – factorial(3)



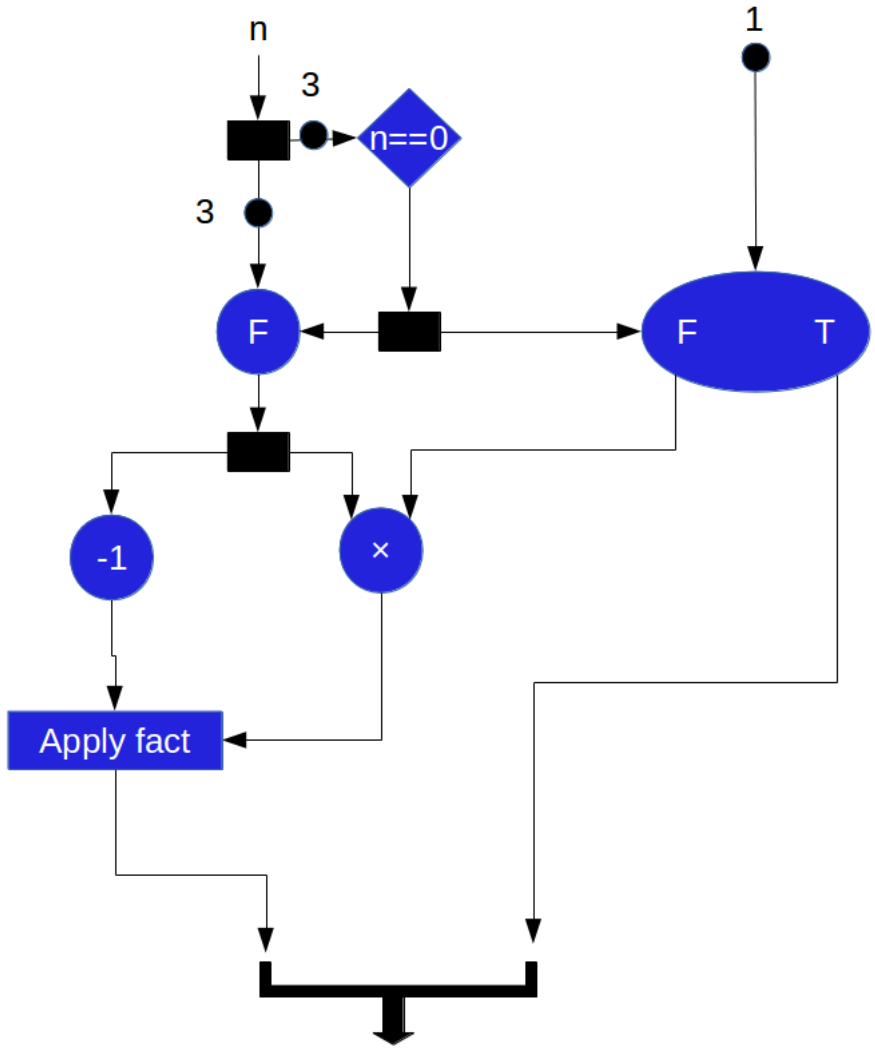
Factorial

Tail Recursive version – factorial(3)



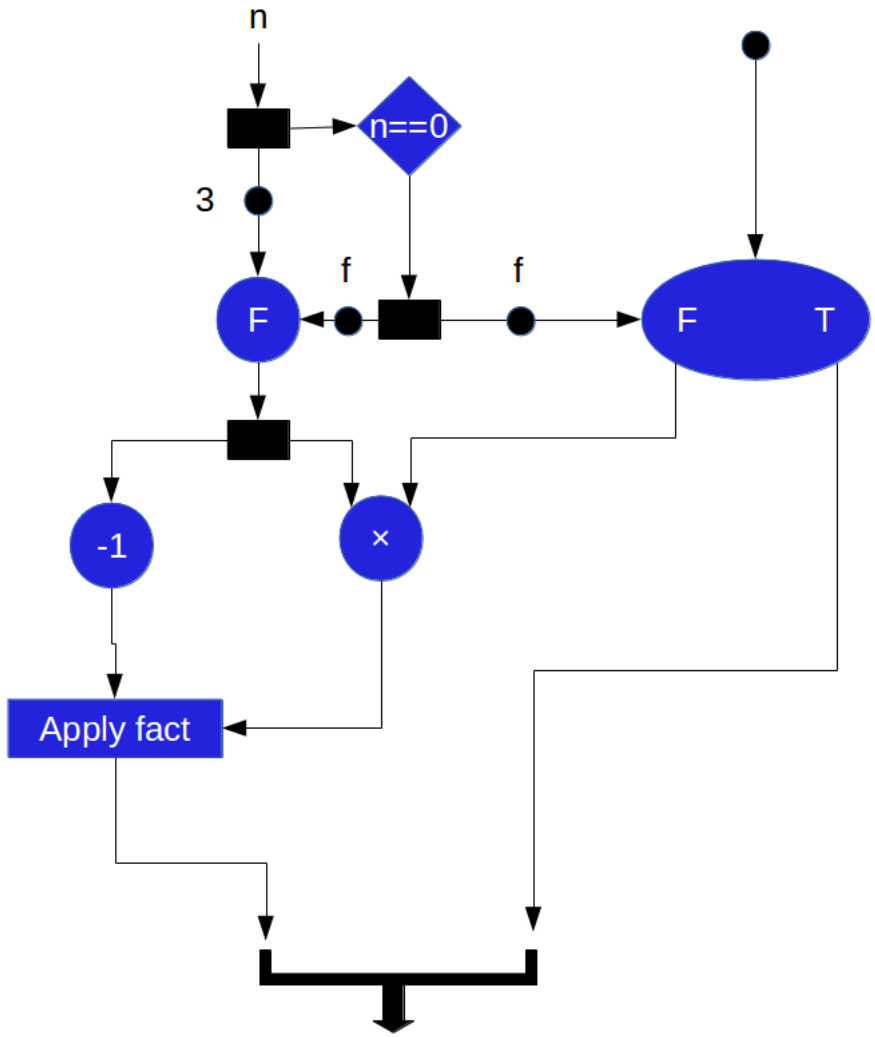
Factorial

Tail Recursive version – factorial(3)



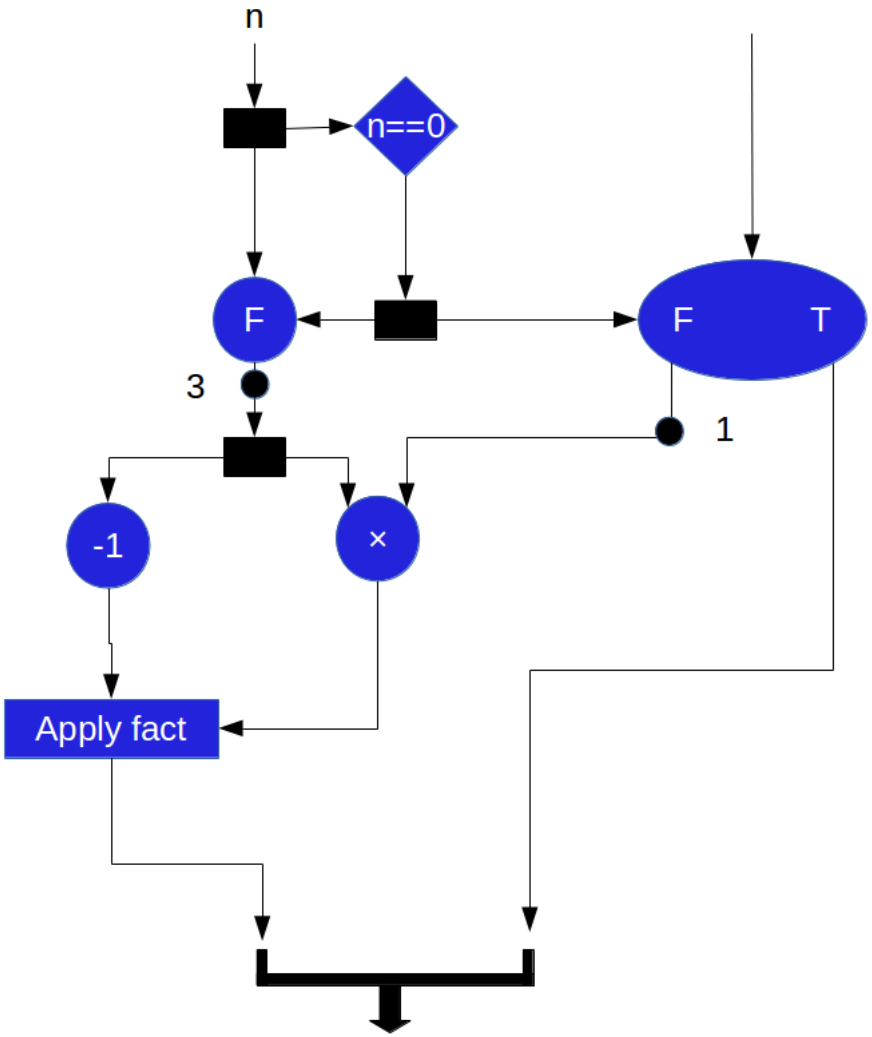
Factorial

Tail Recursive version – factorial(3)



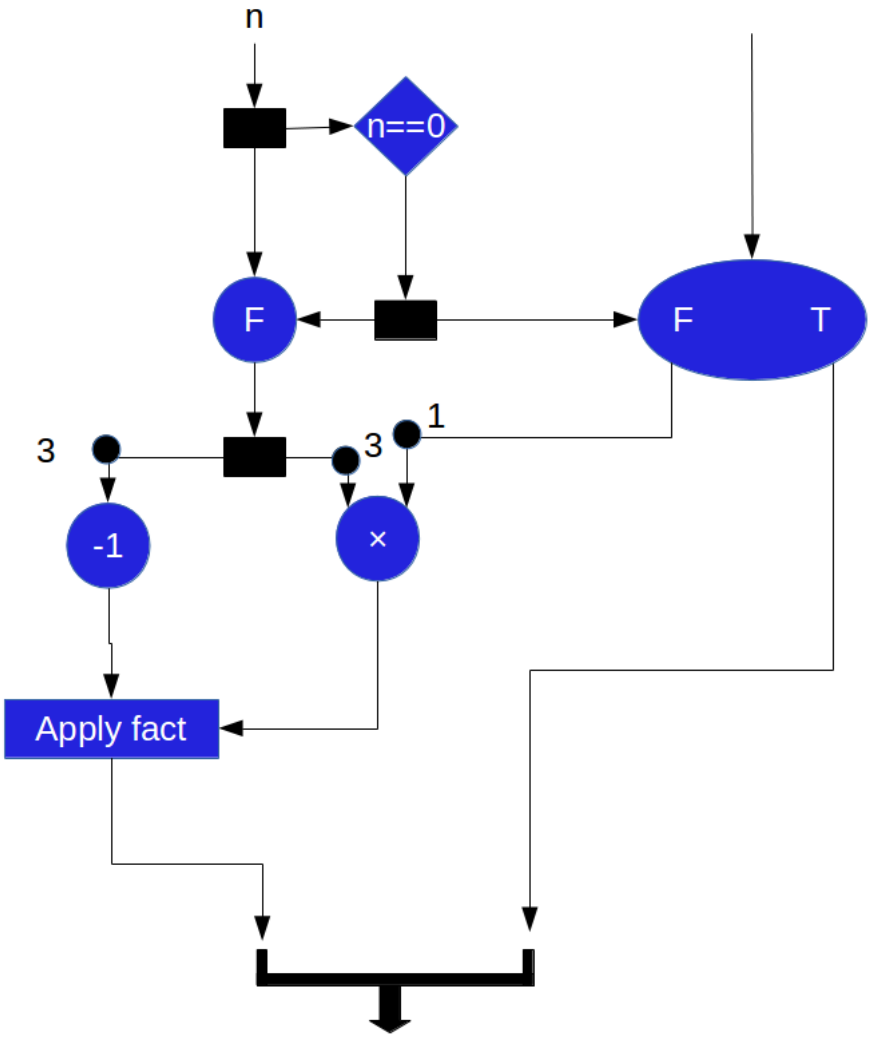
Factorial

Tail Recursive version – factorial(3)



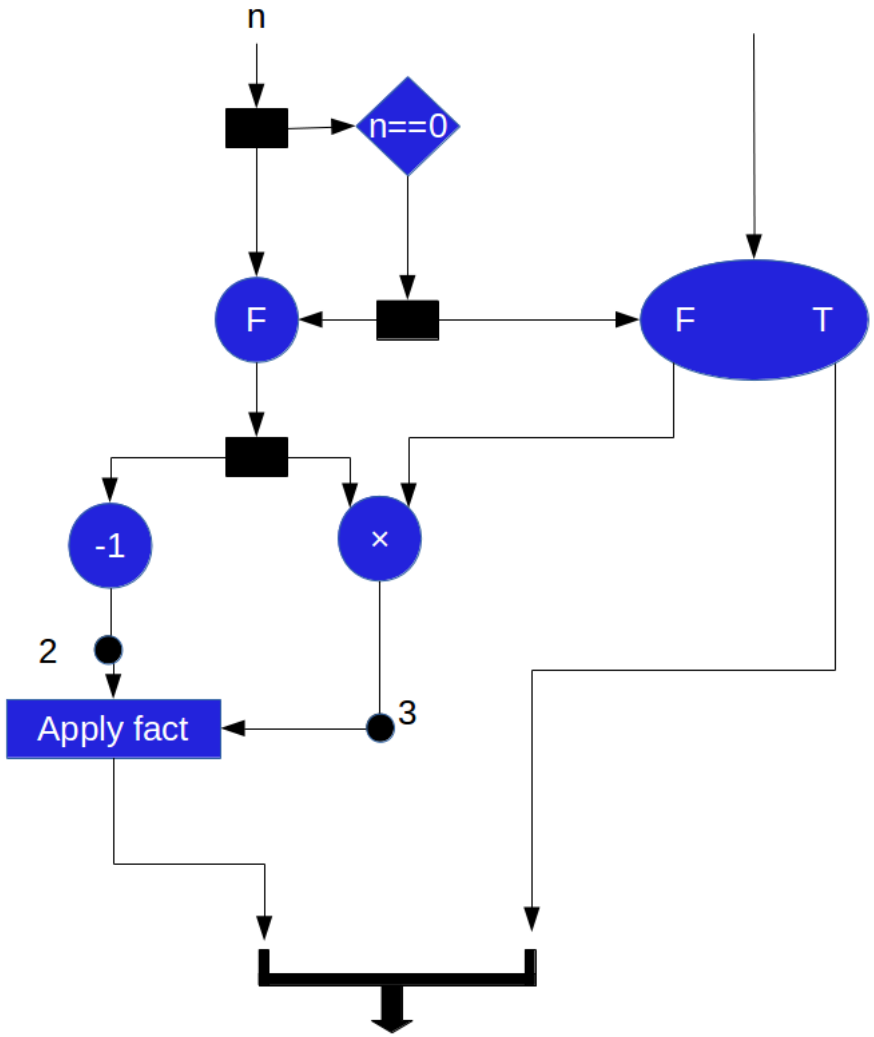
Factorial

Tail Recursive version – factorial(3)



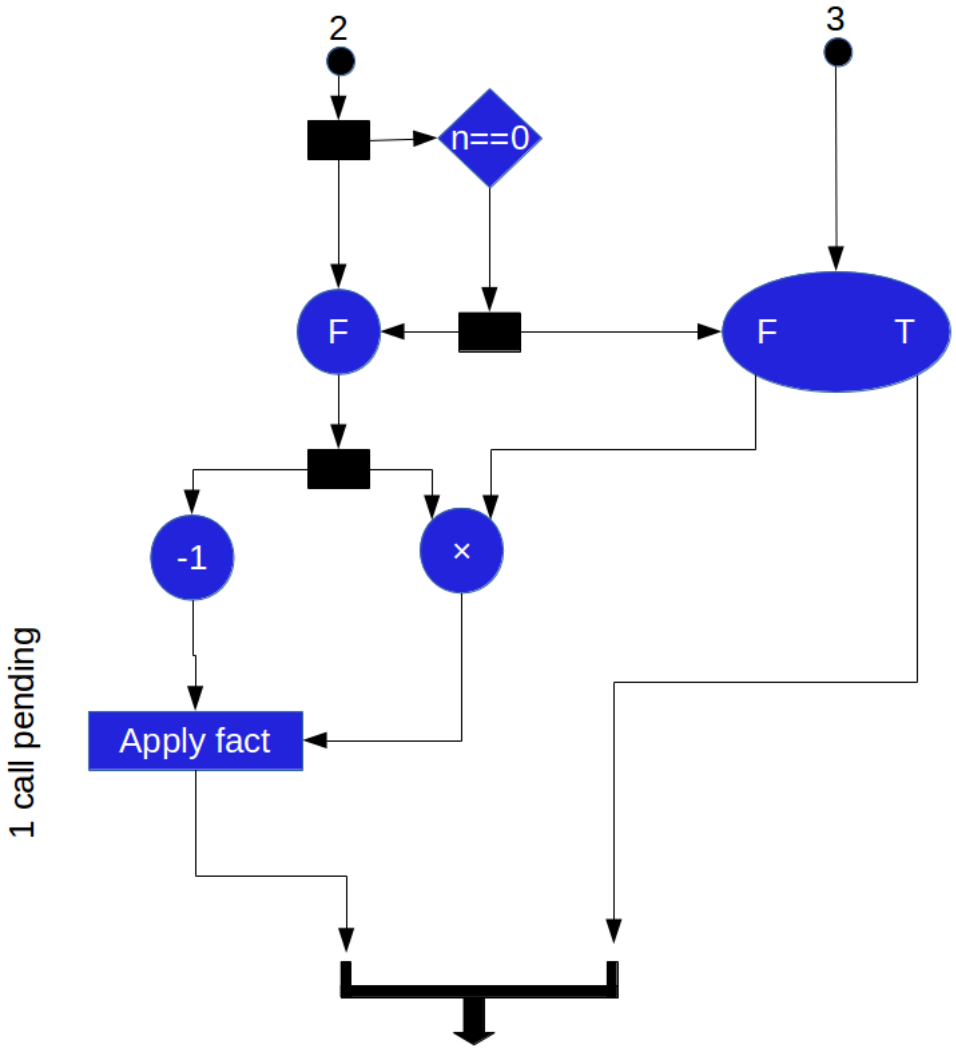
Factorial

Tail Recursive version – factorial(3)



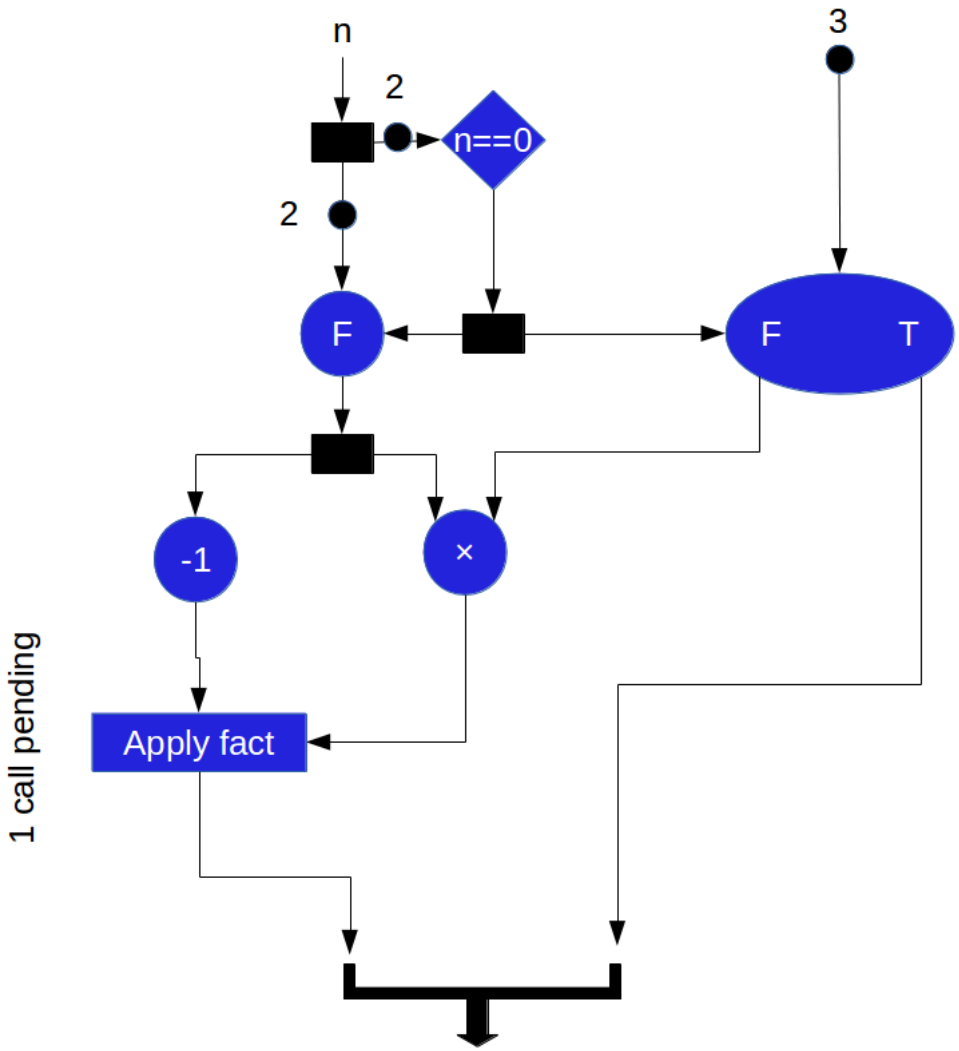
Factorial

Tail Recursive version – factorial(3)



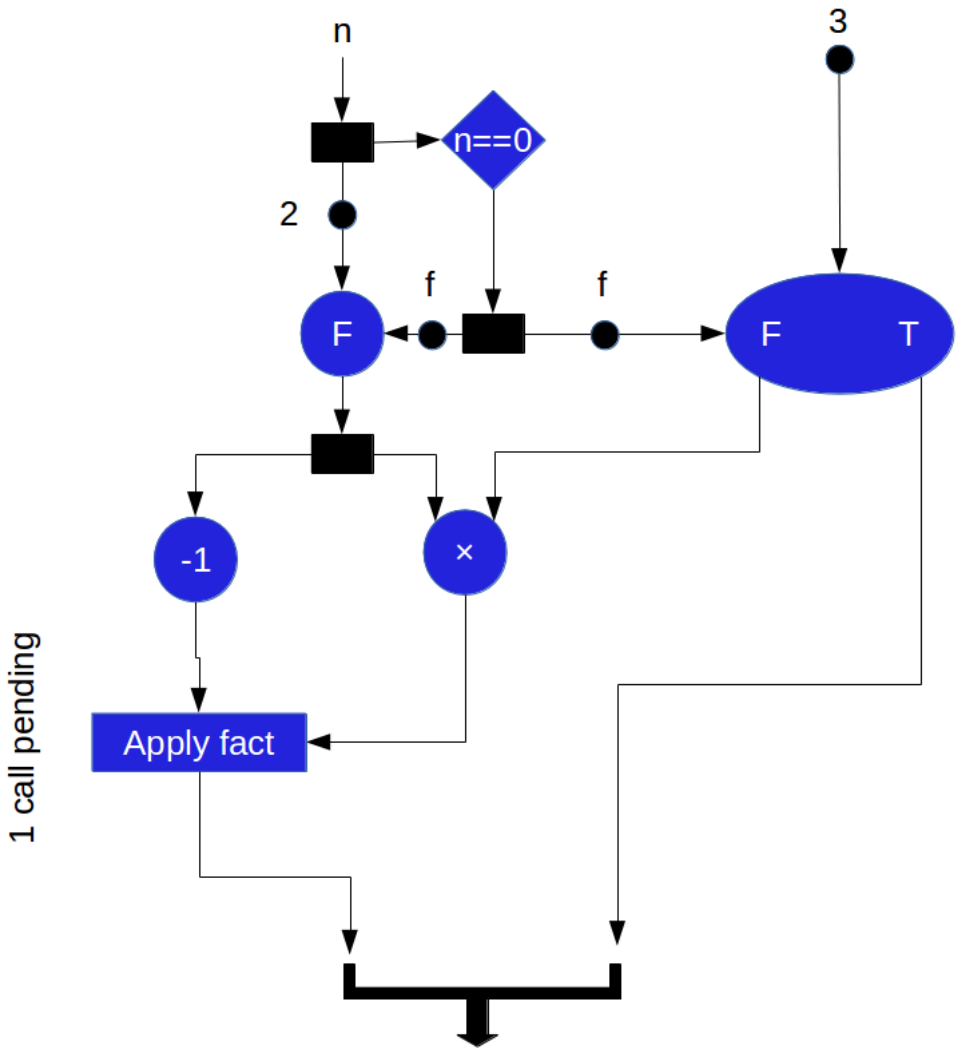
Factorial

Tail Recursive version – factorial(3)



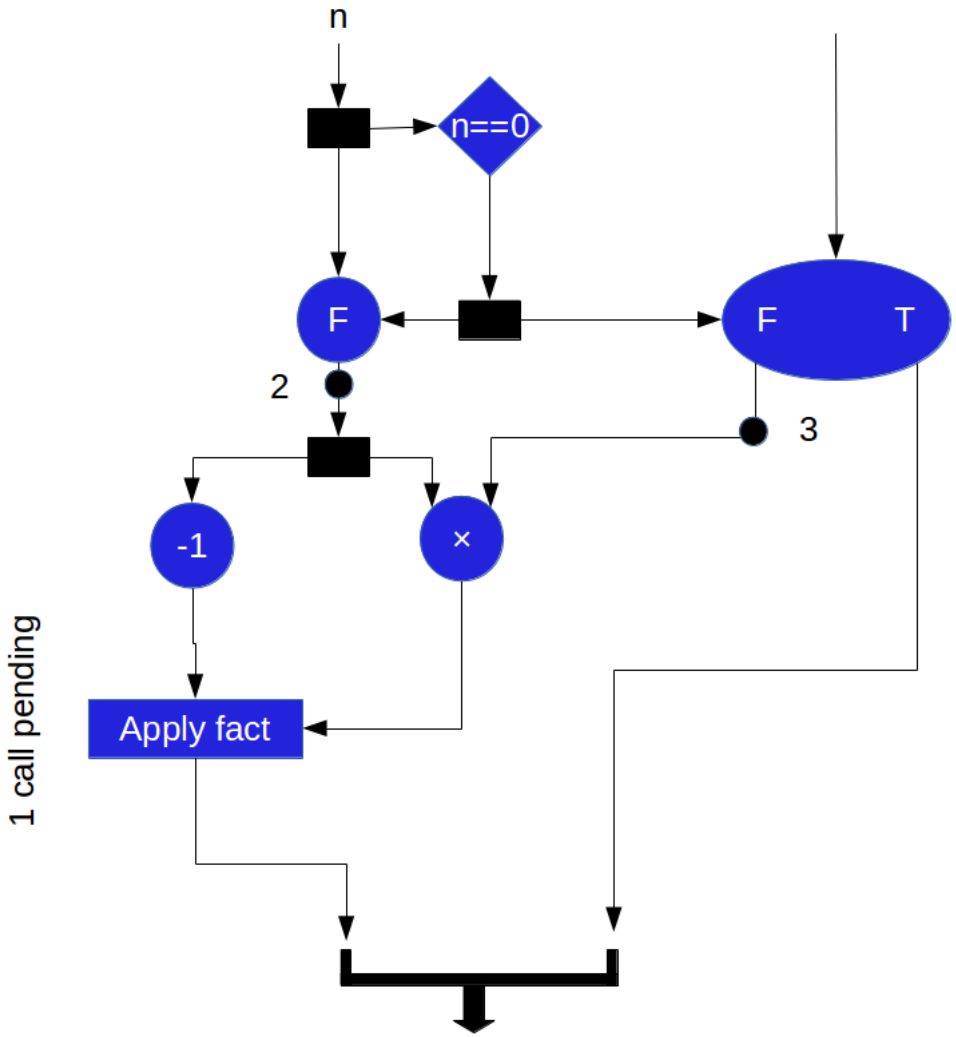
Factorial

Tail Recursive version – factorial(3)



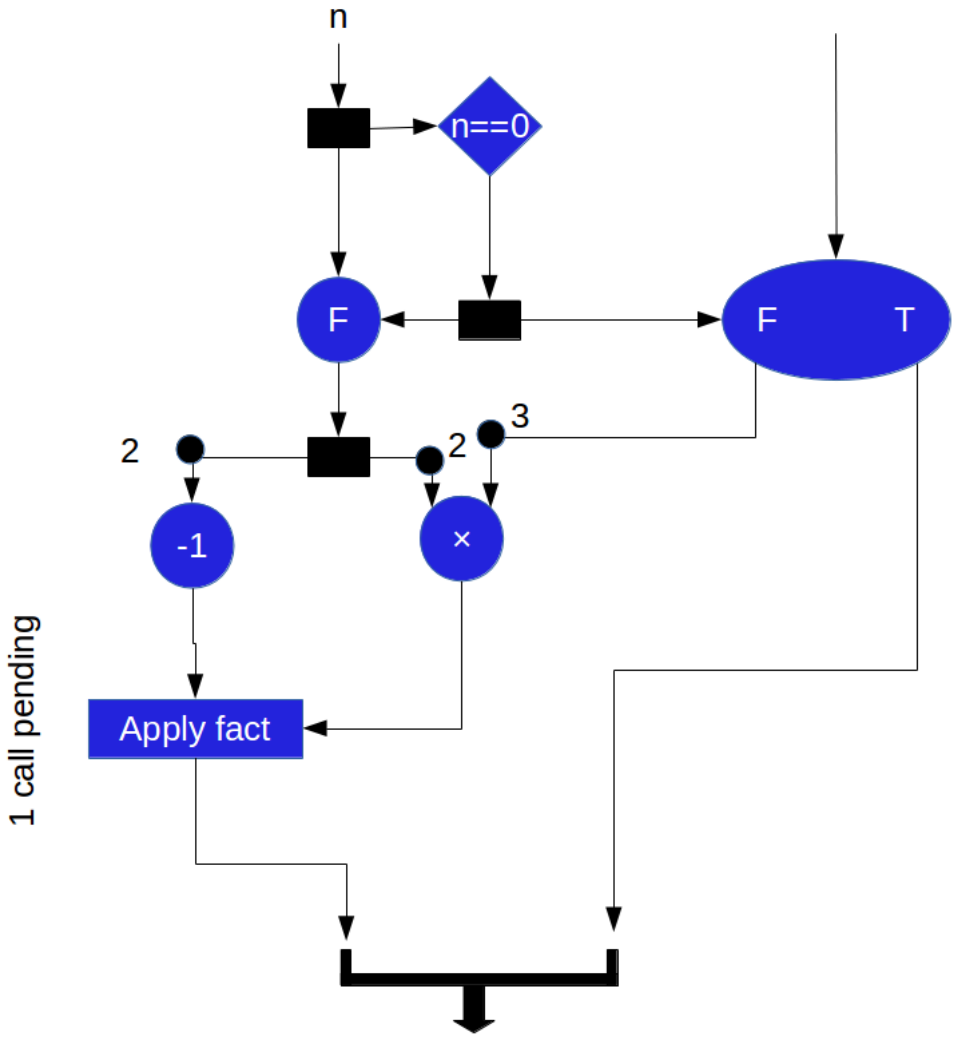
Factorial

Tail Recursive version – factorial(3)



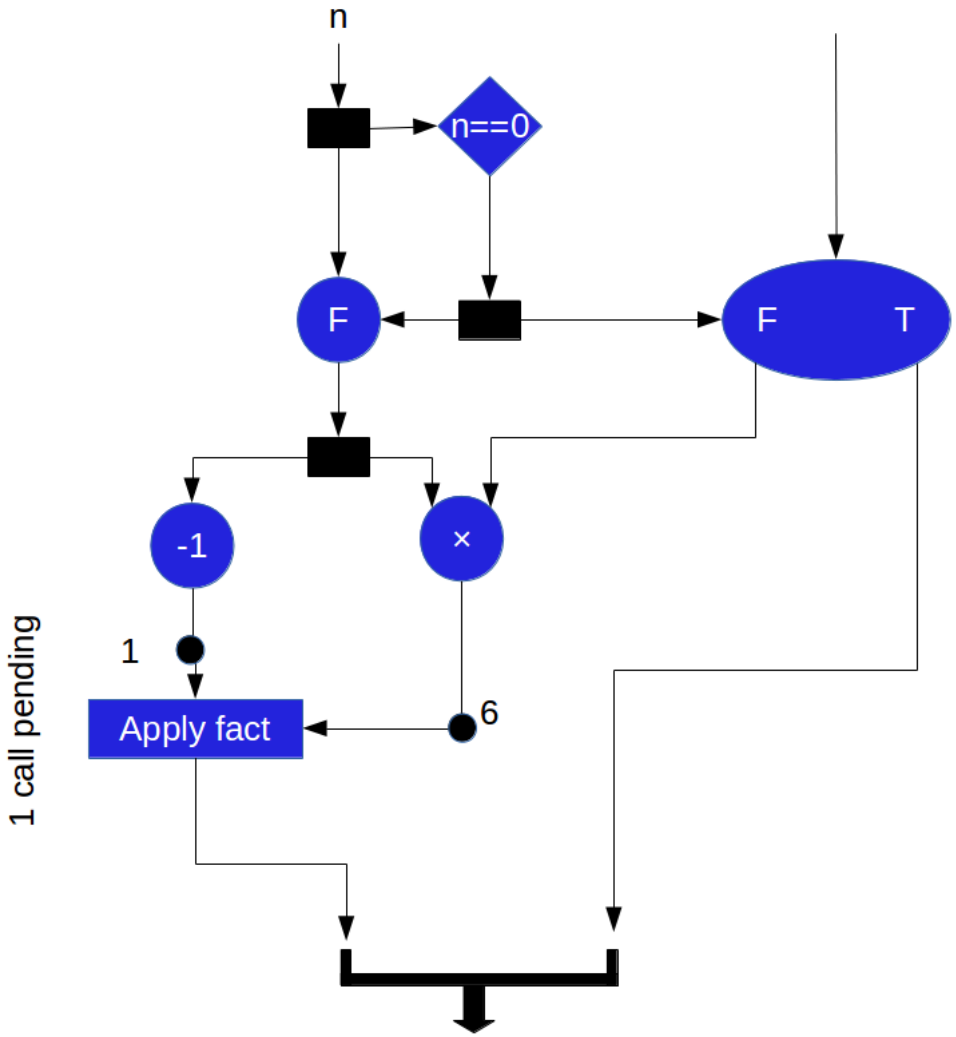
Factorial

Tail Recursive version – factorial(3)



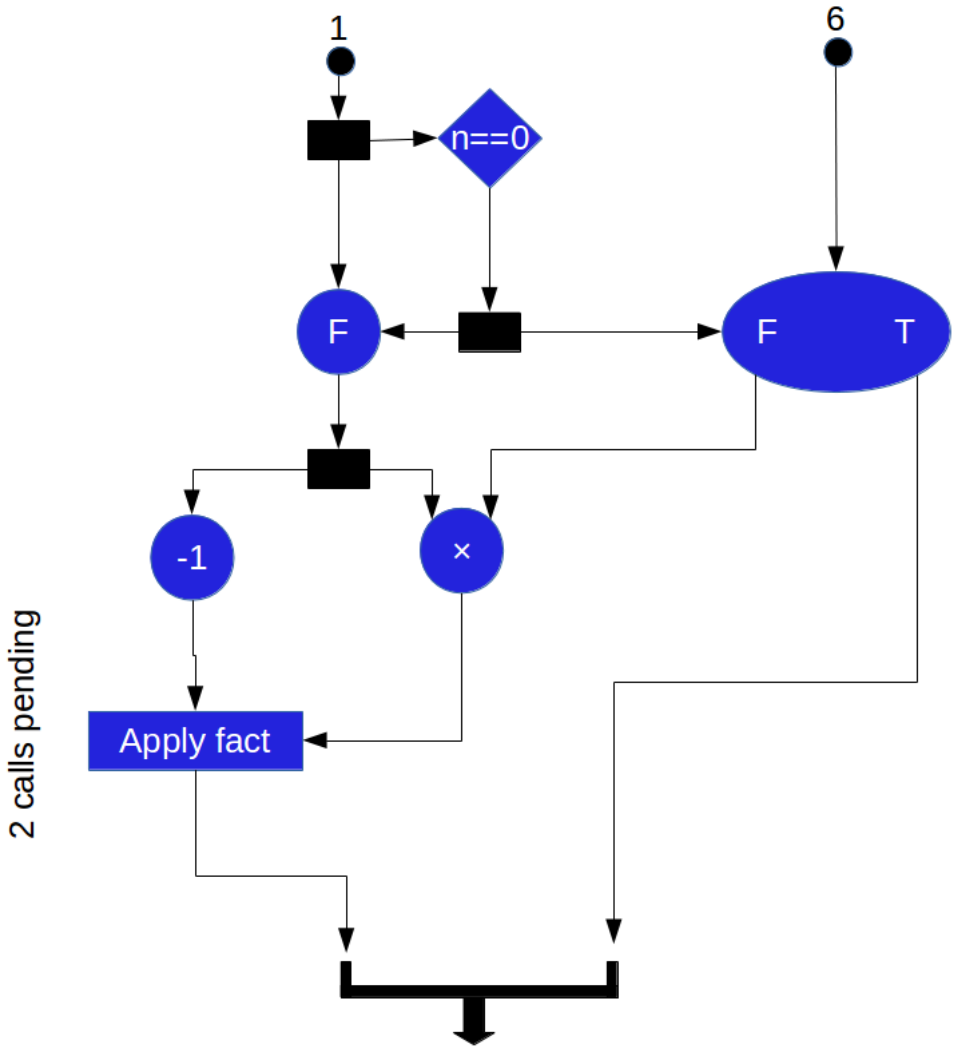
Factorial

Tail Recursive version – factorial(3)



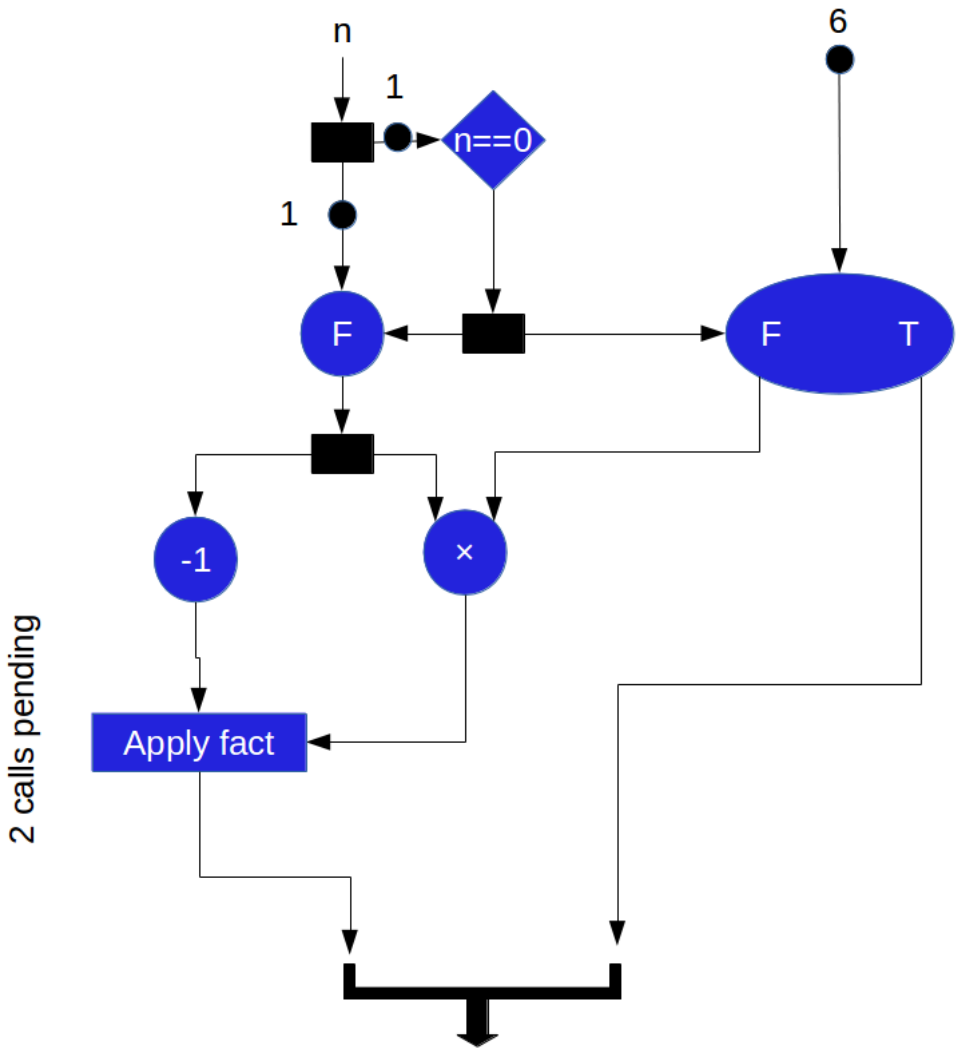
Factorial

Tail Recursive version – factorial(3)



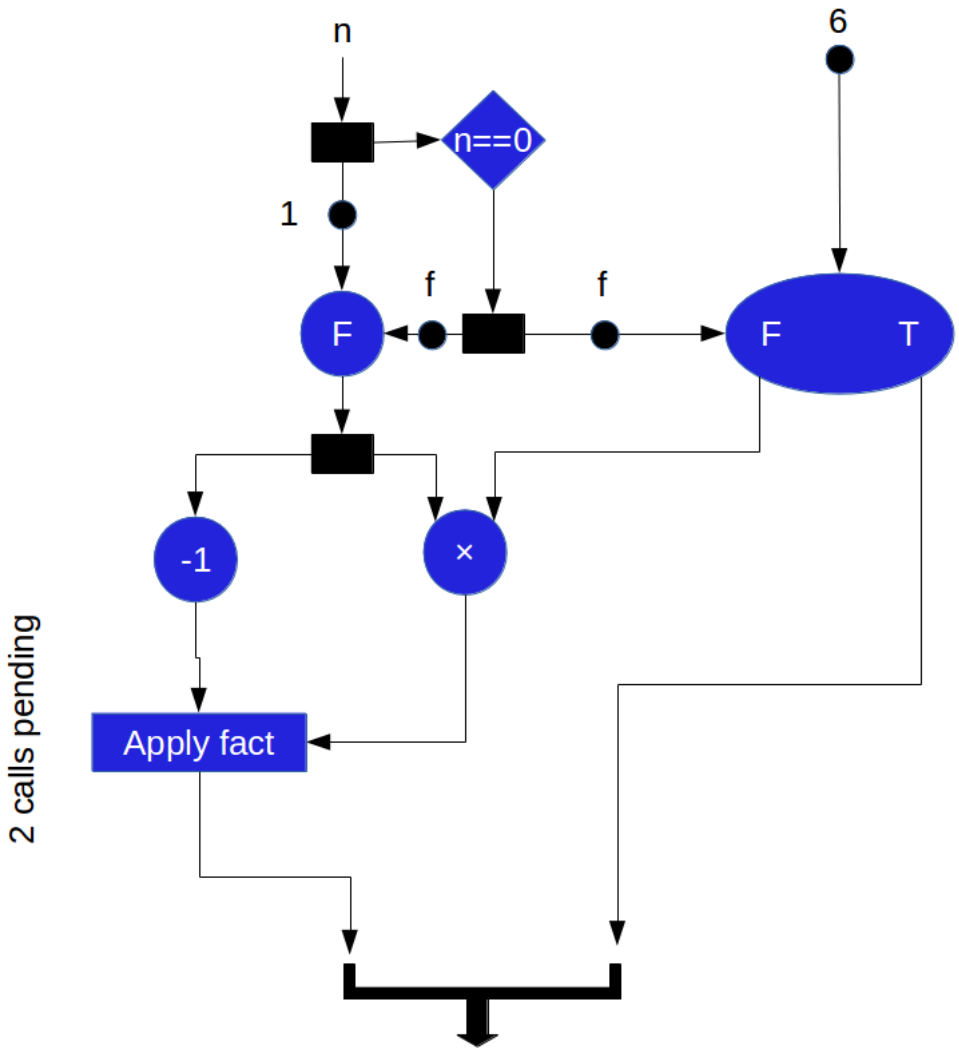
Factorial

Tail Recursive version – factorial(3)



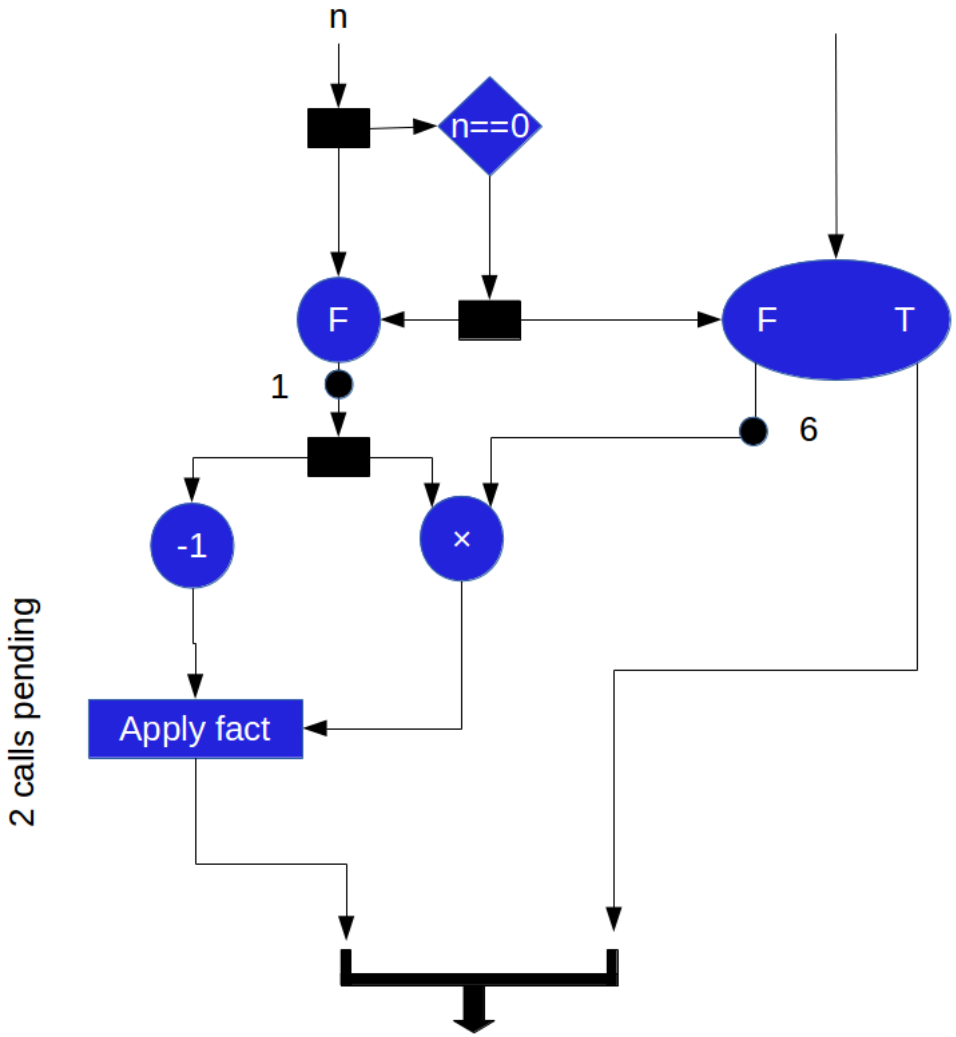
Factorial

Tail Recursive version – factorial(3)



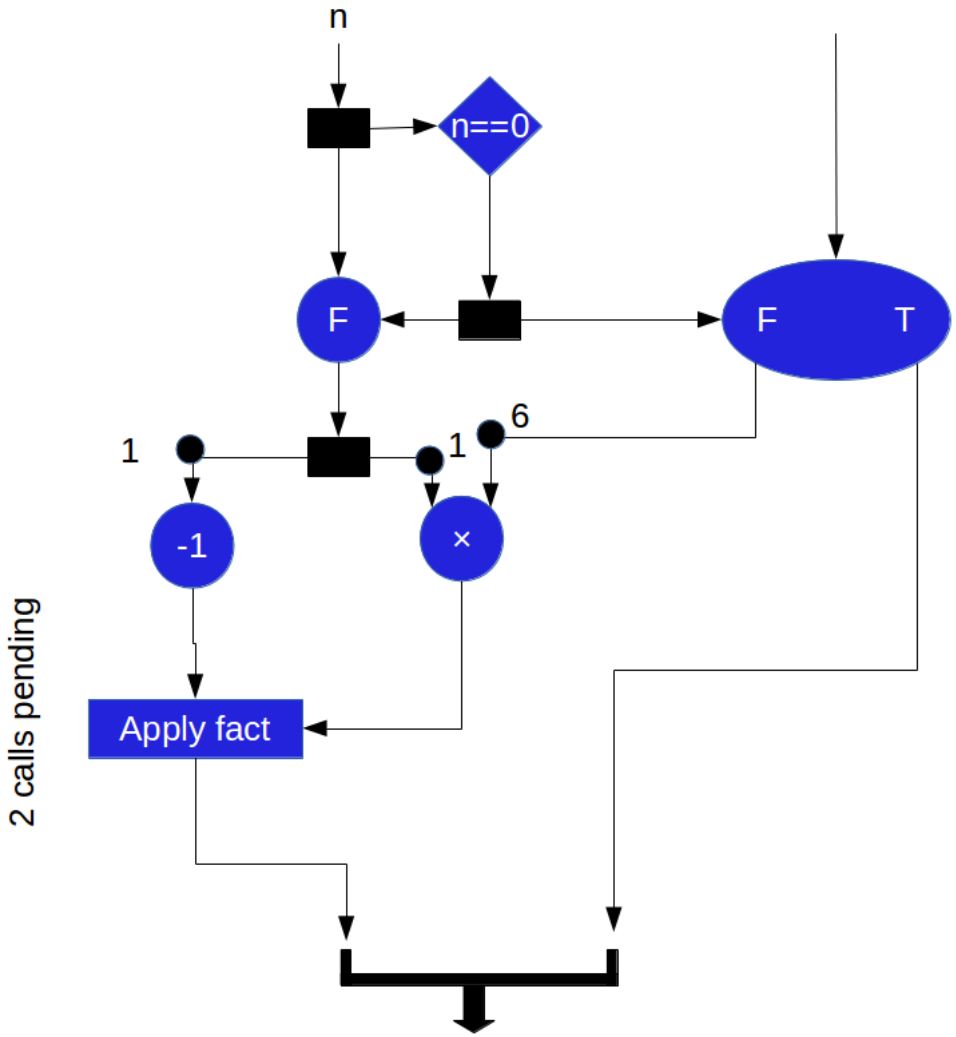
Factorial

Tail Recursive version – factorial(3)



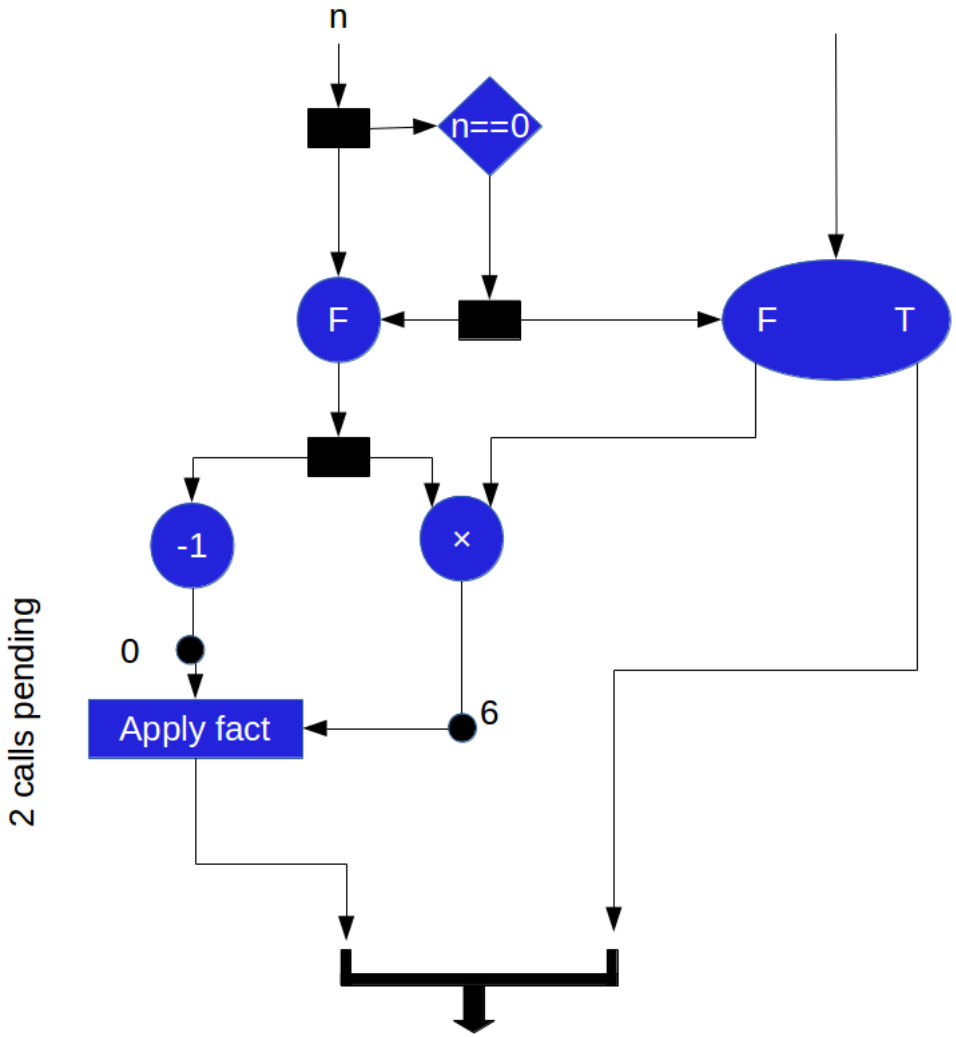
Factorial

Tail Recursive version – factorial(3)



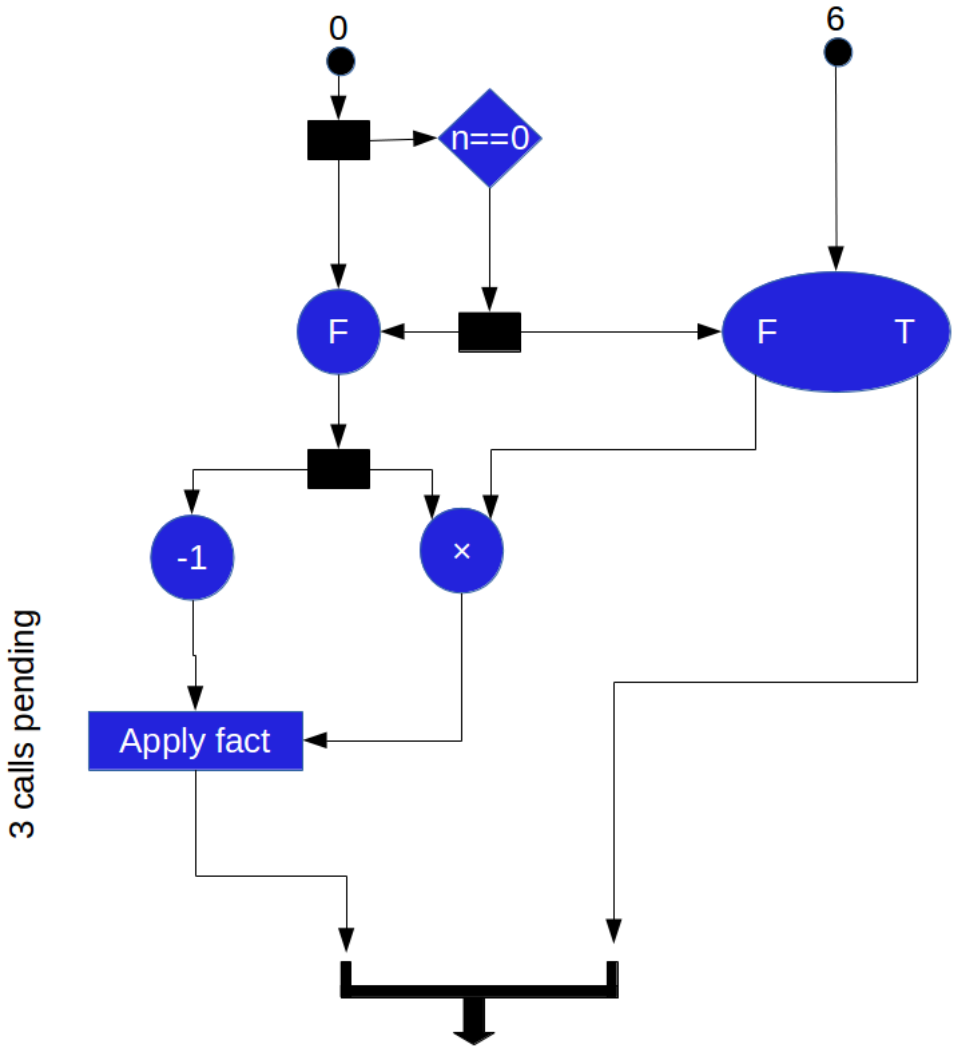
Factorial

Tail Recursive version – factorial(3)



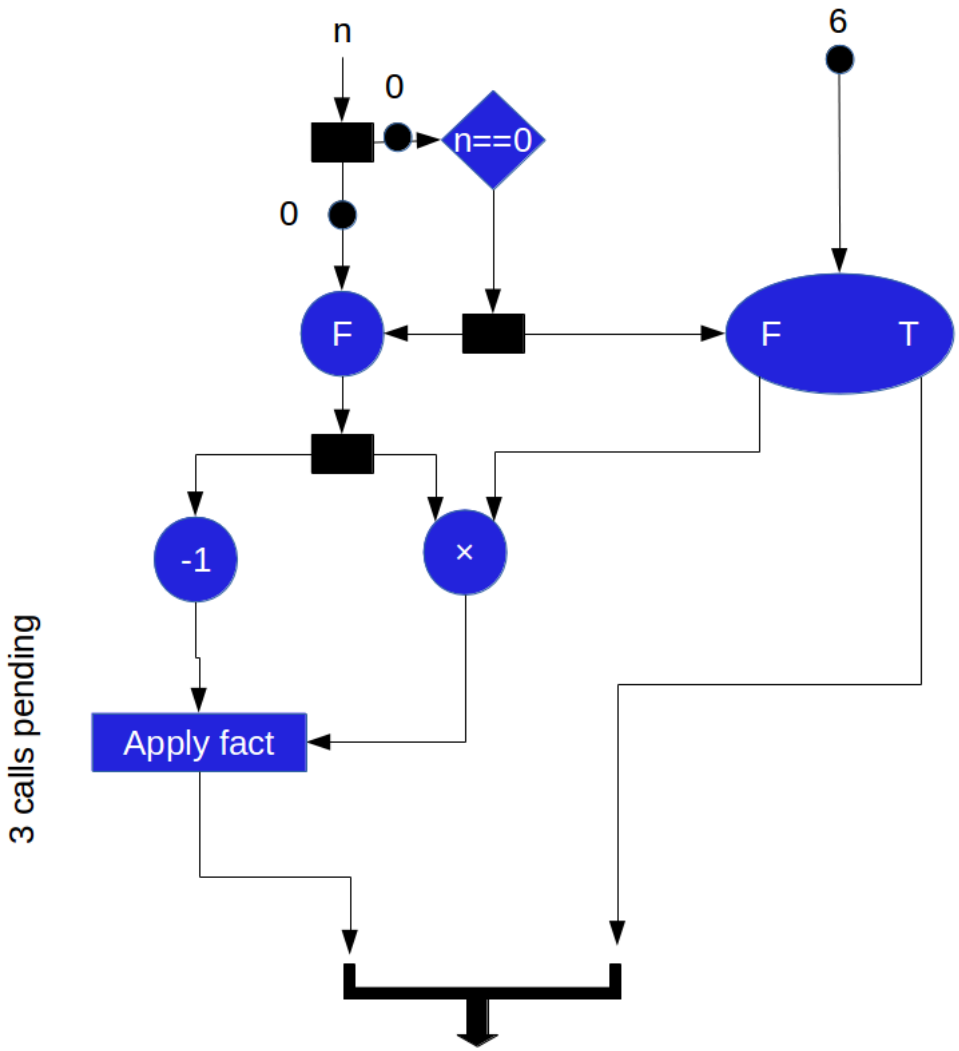
Factorial

Tail Recursive version – factorial(3)



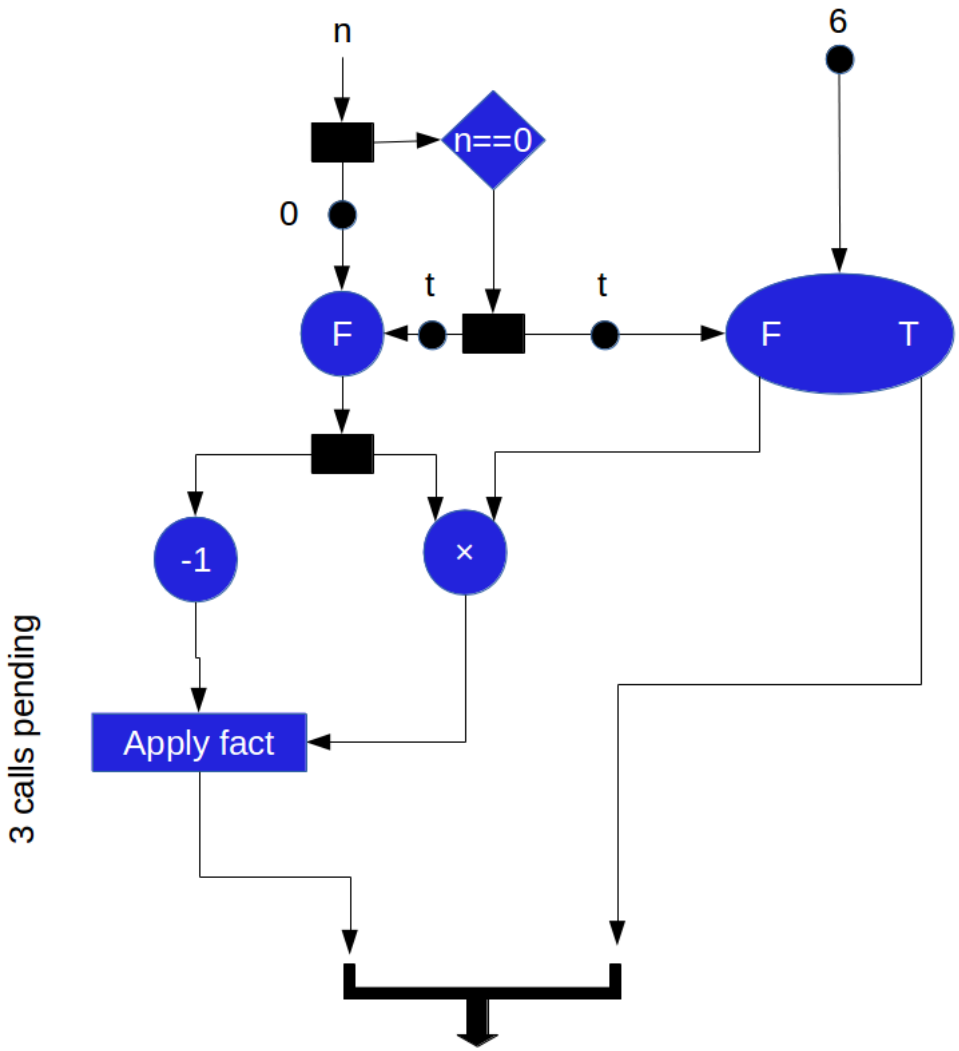
Factorial

Tail Recursive version – factorial(3)



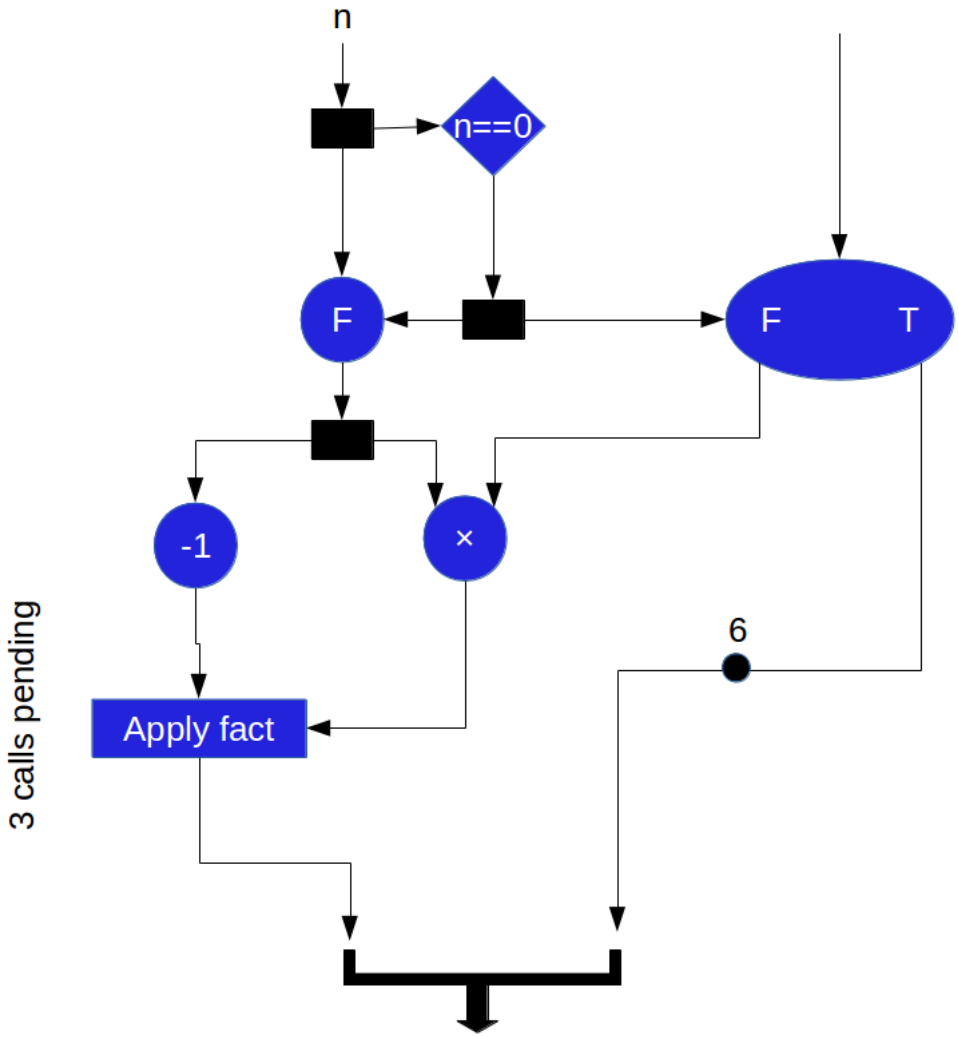
Factorial

Tail Recursive version – factorial(3)



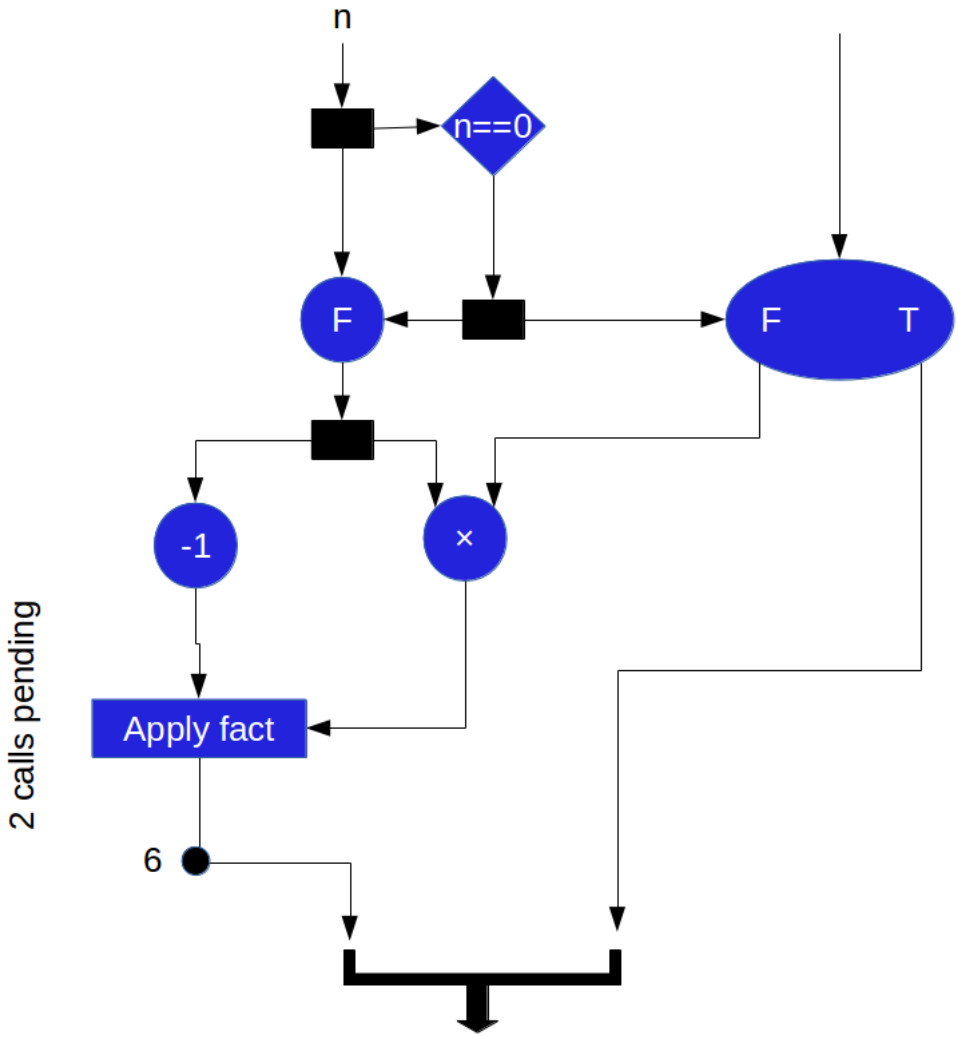
Factorial

Tail Recursive version – factorial(3)



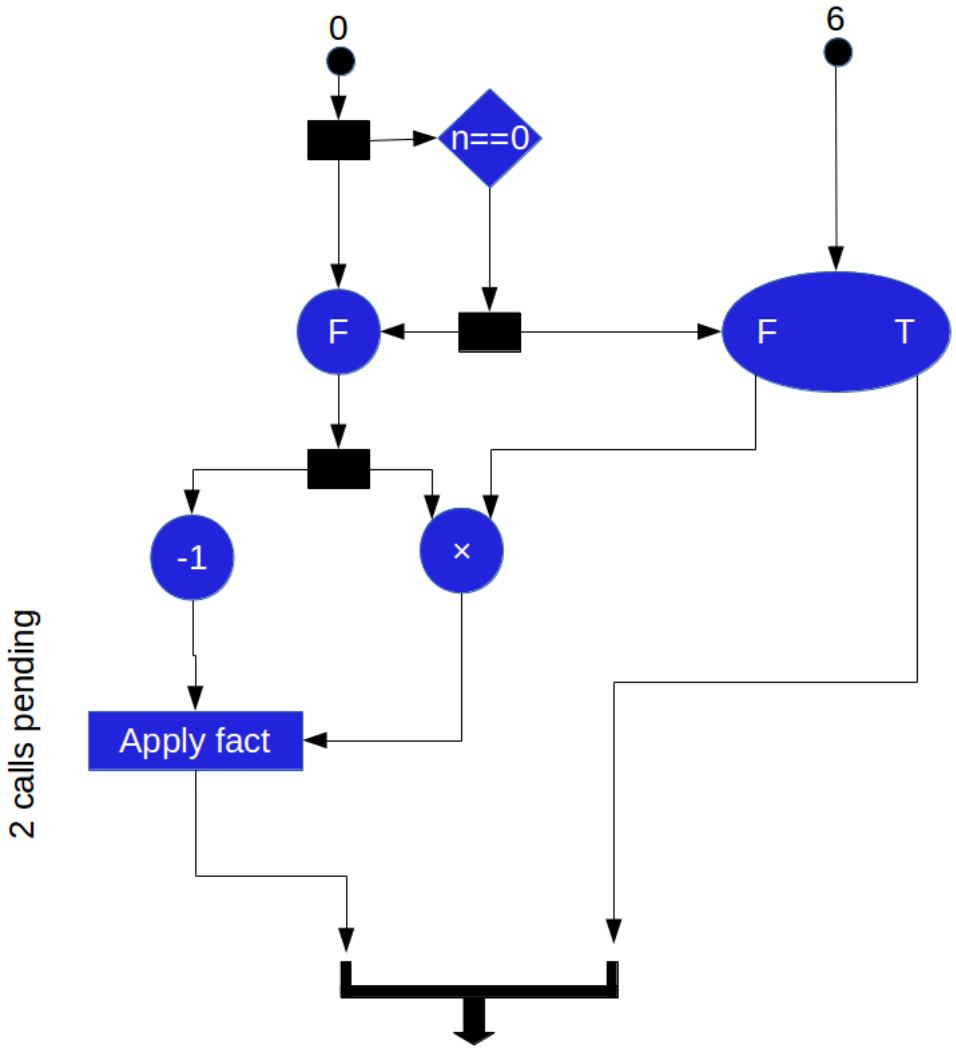
Factorial

Tail Recursive version – factorial(3)



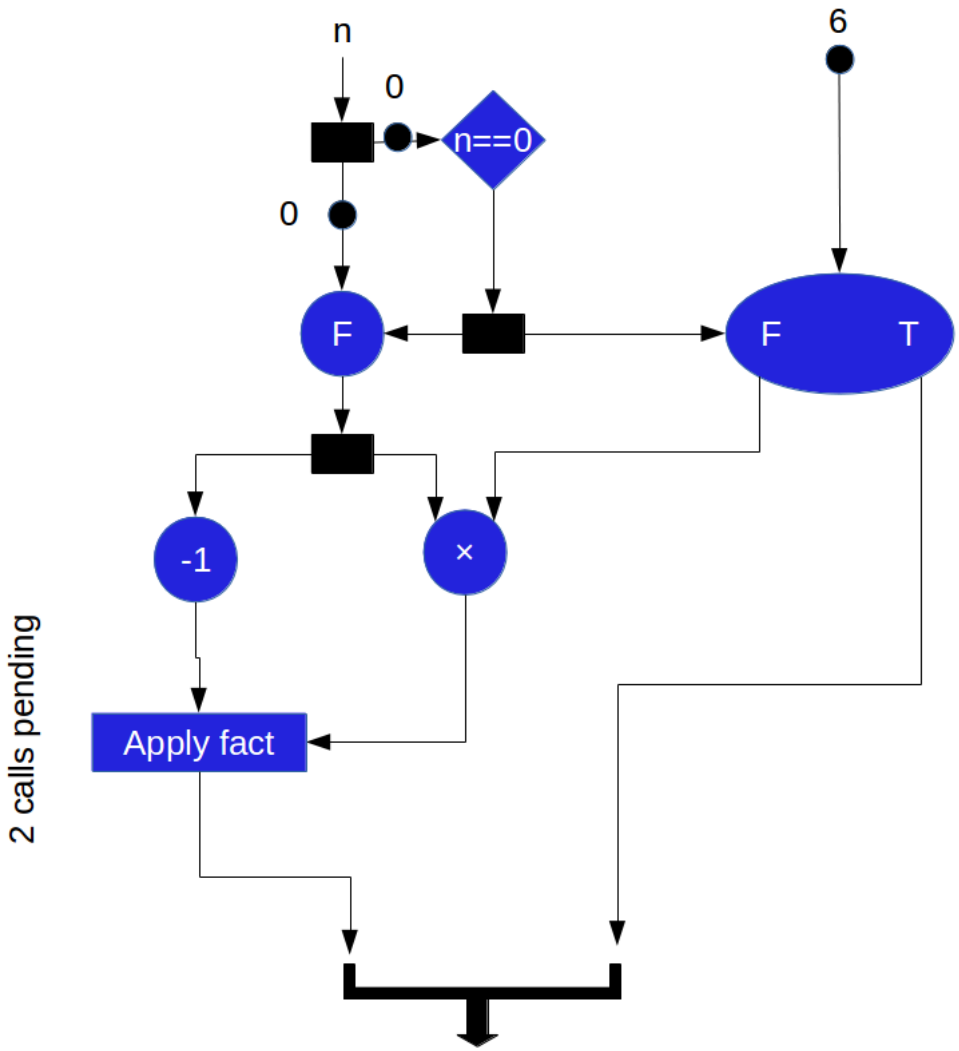
Factorial

Tail Recursive version – factorial(3)



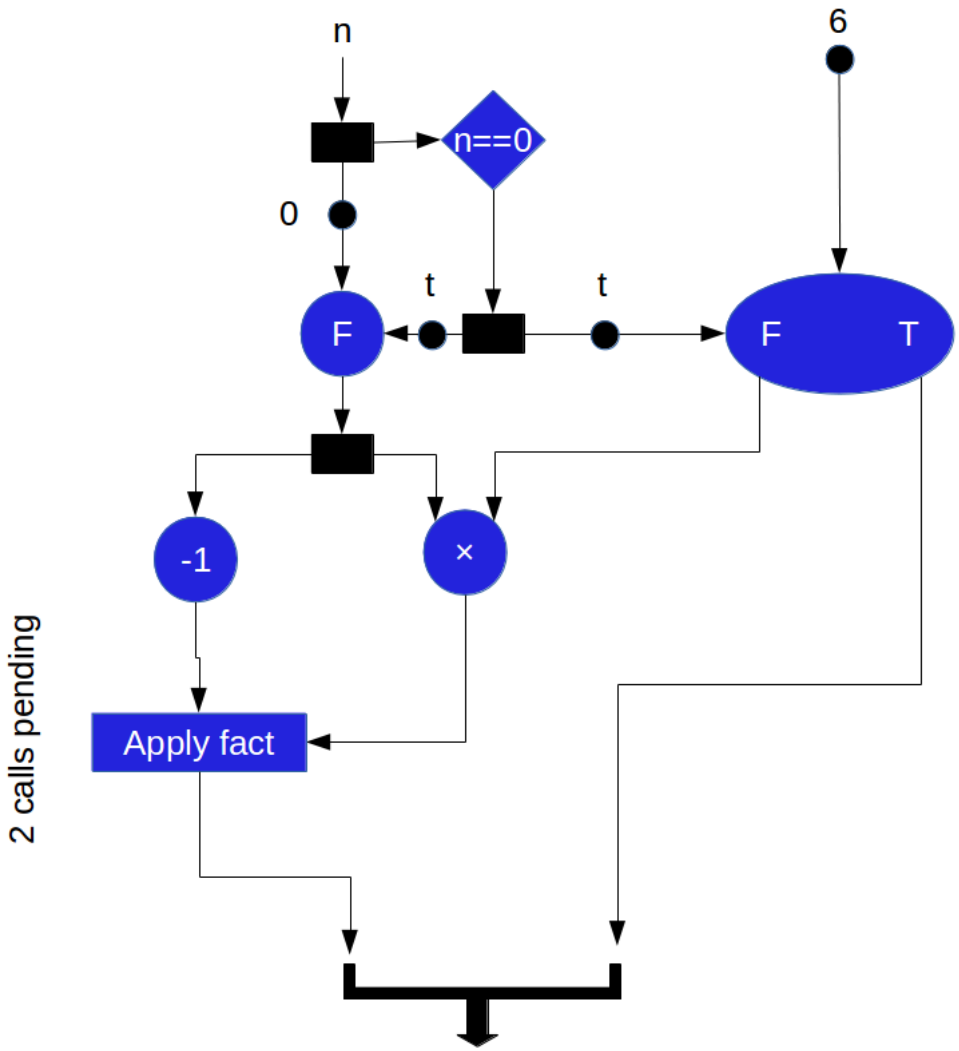
Factorial

Tail Recursive version – factorial(3)



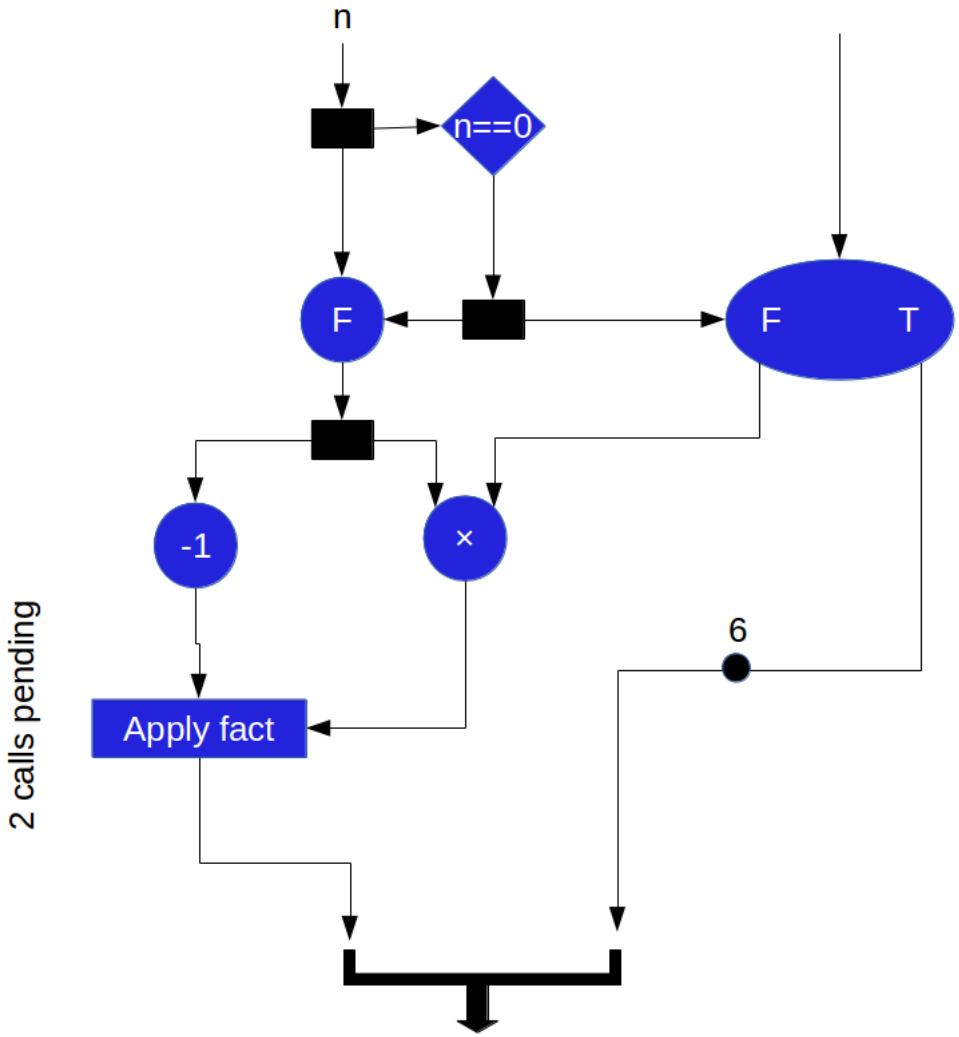
Factorial

Tail Recursive version – factorial(3)



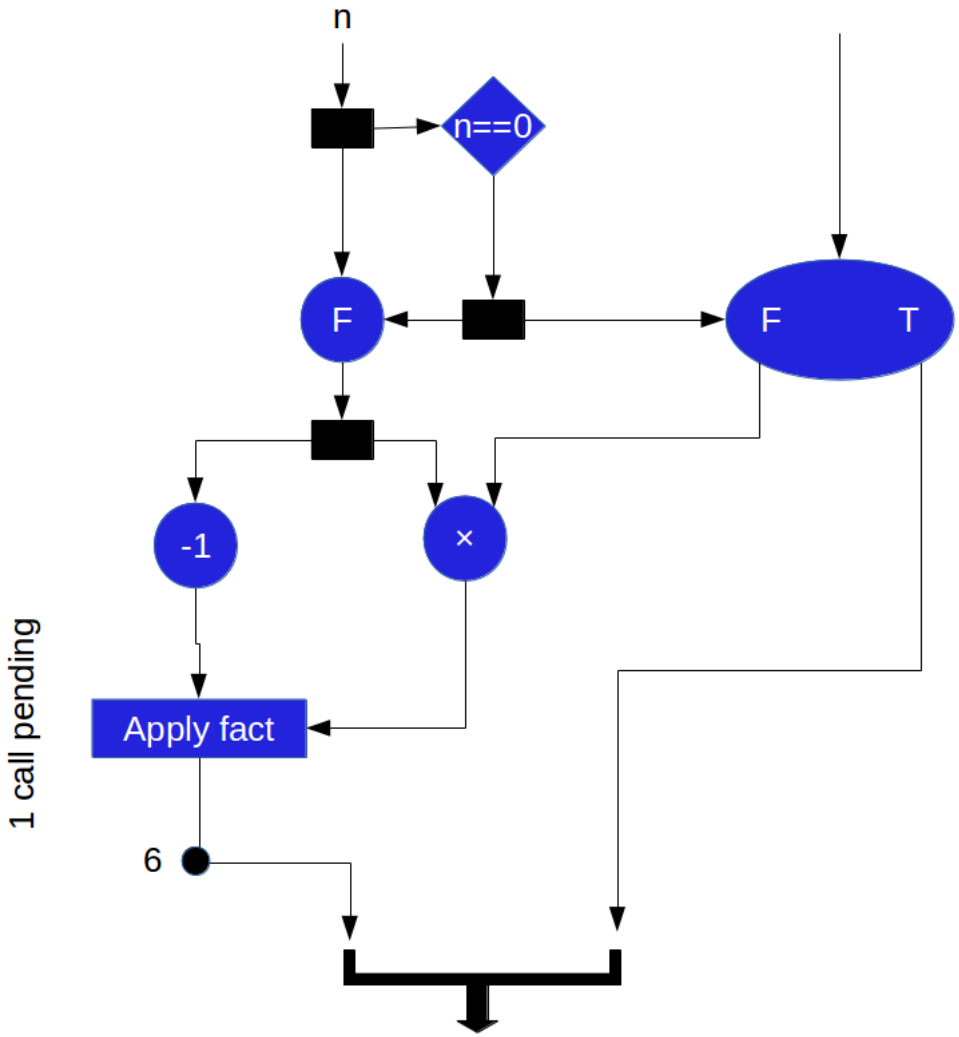
Factorial

Tail Recursive version – factorial(3)



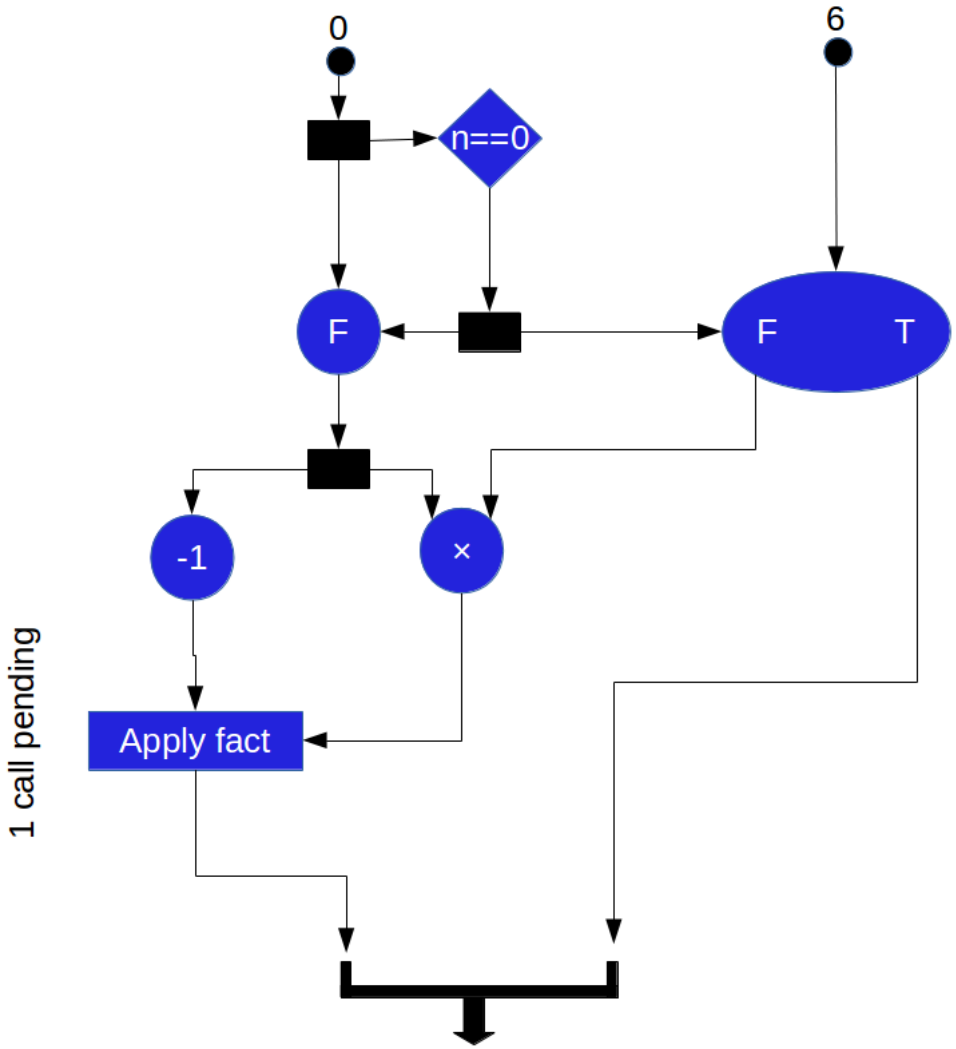
Factorial

Tail Recursive version – factorial(3)



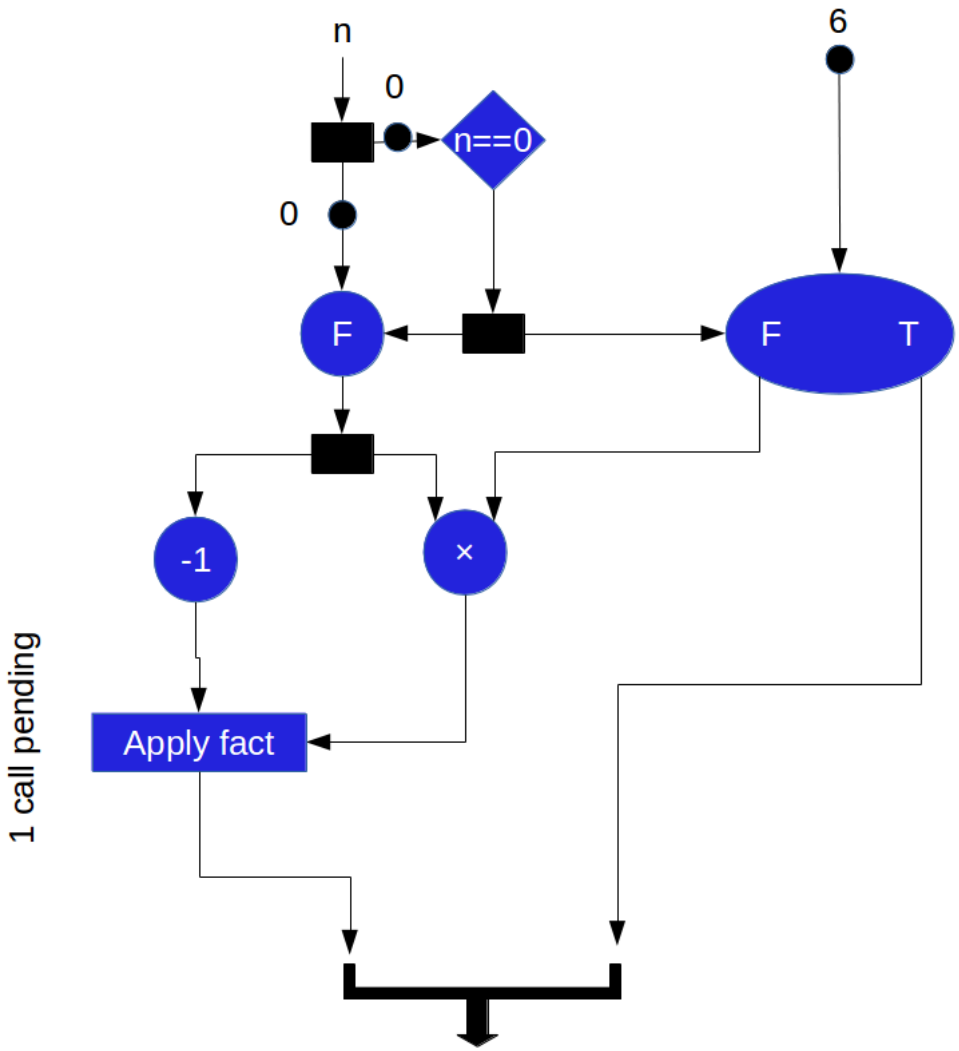
Factorial

Tail Recursive version – factorial(3)



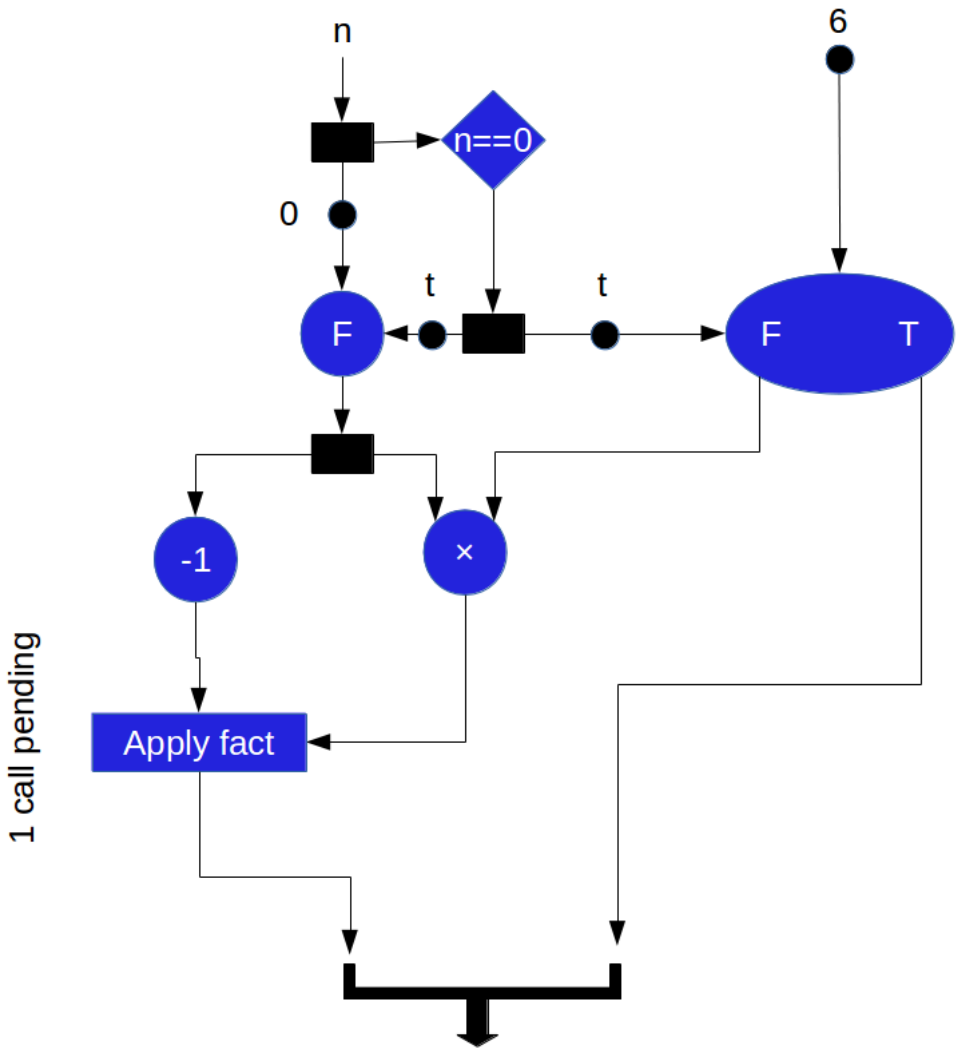
Factorial

Tail Recursive version – factorial(3)



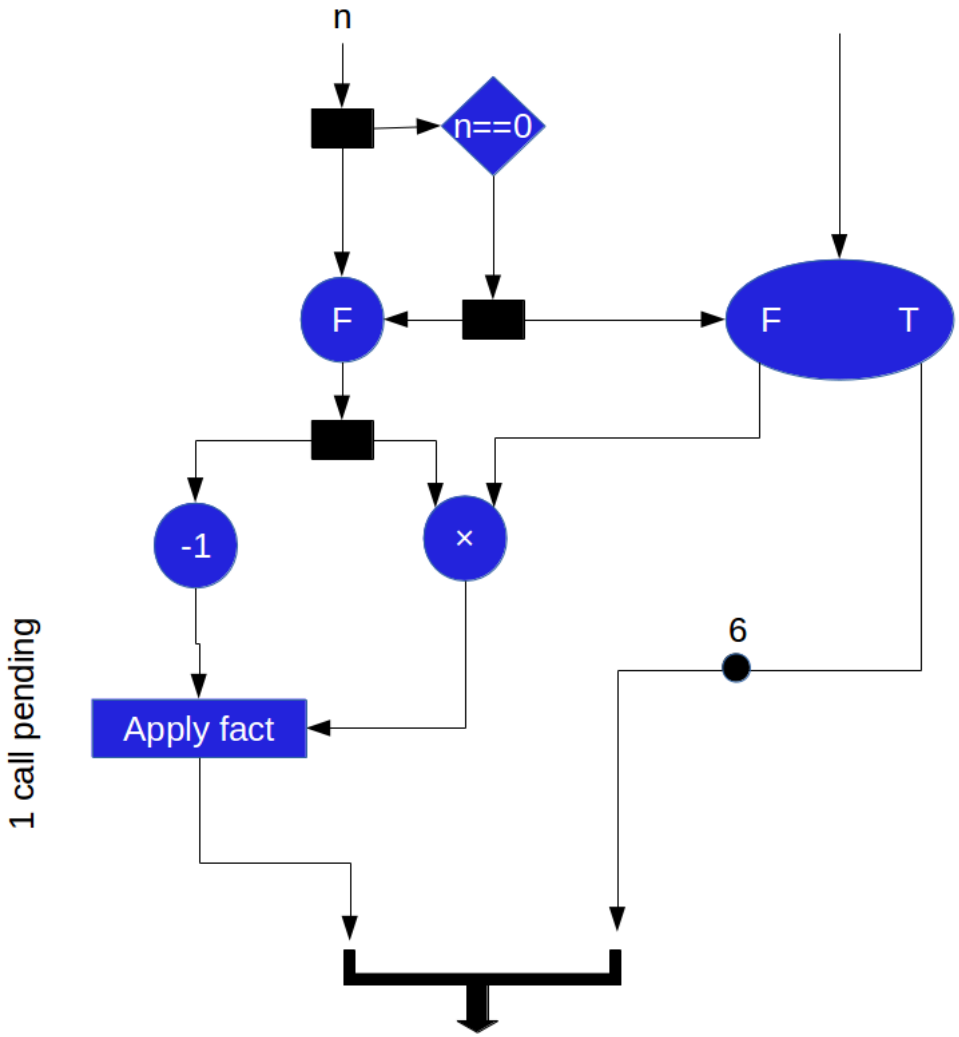
Factorial

Tail Recursive version – factorial(3)



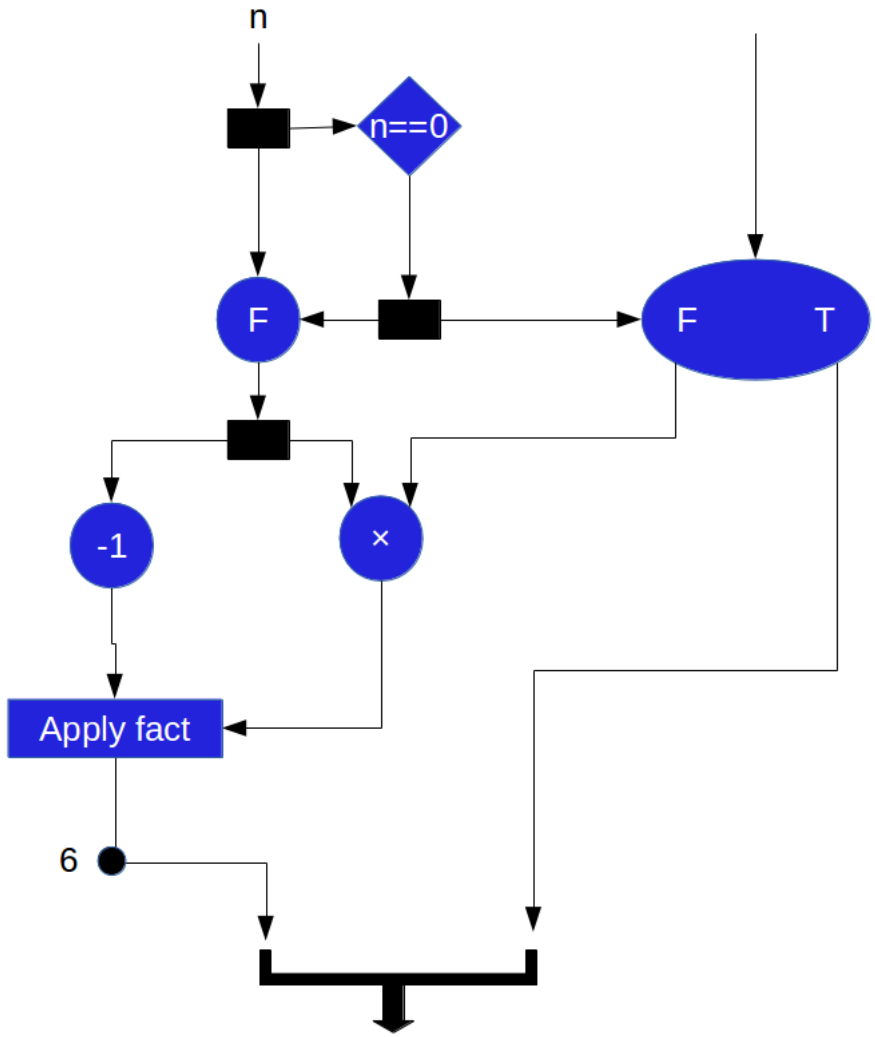
Factorial

Tail Recursive version – factorial(3)



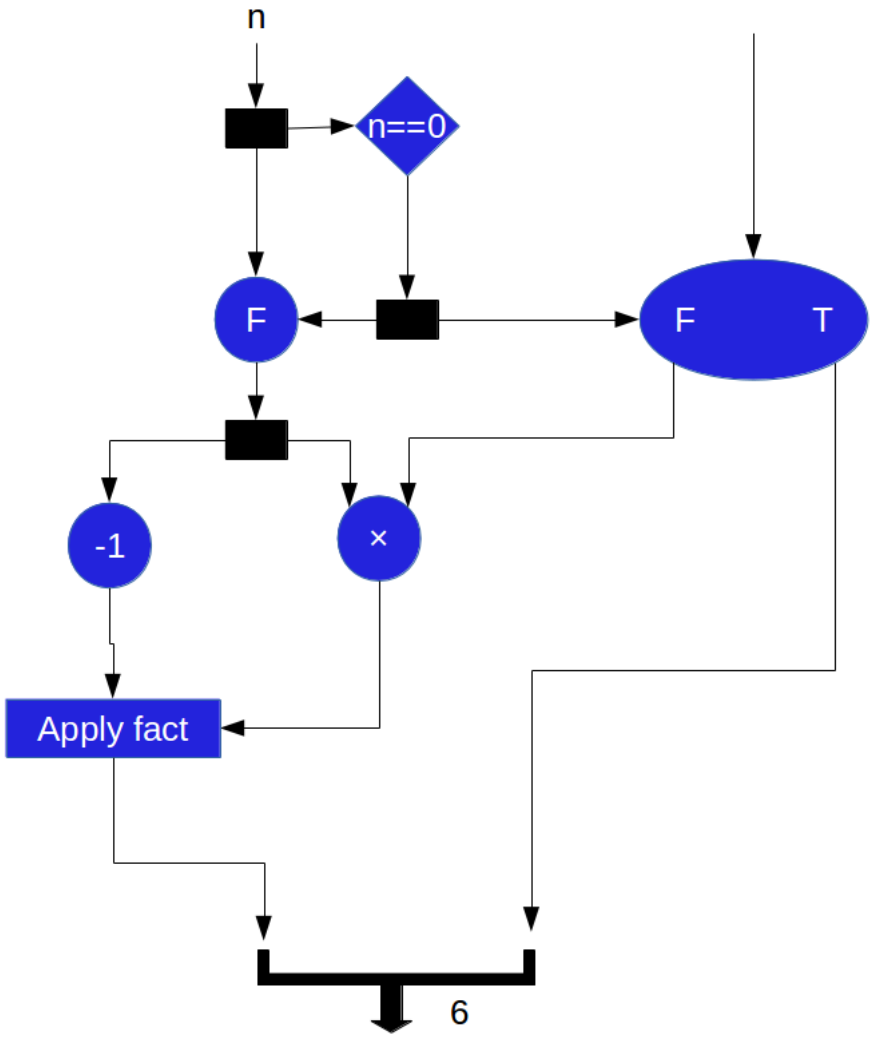
Factorial

Tail Recursive version – factorial(3)



Factorial

Tail Recursive version – factorial(3)



Recursive Program Graph

Features

- Acyclic
- One-token-per-link-in-lifetime
- Tags
- No deterministic merge needed
- Recursion is expressed by runtime copying

Outline

Homework

- *factorial* function:
  - Convert the recursive form of *factorial* into an iterative form
  - Create/draw the corresponding static dataflow graph
- *power* function:
  - Convert the iterative form of the *power* function into its recursive form
  - Create/draw its recursive program graph

References I

Conway, Melvin E. (1963). "A Multiprocessor System Design". In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. AFIPS '63 (Fall). Las Vegas, Nevada: ACM, pp. 139–146. DOI: 10.1145/1463822.1463838. URL: http://doi.acm.org/10.1145/1463822.1463838.

Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, eds. (1972). *Structured Programming*. London, UK, UK: Academic Press Ltd. ISBN: 0-12-200550-3.

Dijkstra, E. W. (1971). "Hierarchical Ordering of Sequential Processes". In: *Acta Inf.* 1.2, pp. 115–138. ISSN: 0001-5903. DOI: 10.1007/BF00289519. URL: http://dx.doi.org/10.1007/BF00289519.

Holt, Richard C (1978). *Structured concurrent programming with operating systems applications*. Vol. 2937. Reading, Mass.; Don Mills, Ont.: Addison-Wesley Publishing Company.

References II

Ianov, Iu I. (1958). "On the Equivalence and Transformation of Program Schemes". In: *Commun. ACM* 1.10, pp. 8–12. ISSN: 0001-0782. DOI: 10.1145/368924.368930. URL: http://doi.acm.org/10.1145/368924.368930.

Landin, Peter J (1964). "The mechanical evaluation of expressions". In: *The Computer Journal* 6.4, pp. 308–320.

McCarthy, John (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4, pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: http://doi.acm.org/10.1145/367177.367199.

McIlroy, M. D. (1969). "Alternatives to Extensible Languages". In: *SIGPLAN Not.* 4.8, pp. 50–52. ISSN: 0362-1340. DOI: 10.1145/1115858.1115870. URL: http://doi.acm.org/10.1145/1115858.1115870.

References III

- Muller, D. E. and W. Scott Bartky (1957). *A Theory of Asynchronous Circuits*. Report 78. University of Illinois, Graduate College, Digital Computer Laboratory.
- Muller, David E (1963). "Asynchronous logics and application to information processing". In: *Switching Theory in Space Technology*, pp. 289–297.
- Paterson, Michael S. and Carl E. Hewitt (1970). "Record of the Project MAC Conference on Concurrent Systems and Parallel Computation". In: ed. by Jack B. Dennis. New York, NY, USA: ACM. Chap. Comparative Schematology, pp. 119–127. DOI: 10.1145/1344551.1344563. URL: http://doi.acm.org/10.1145/1344551.1344563.
- Petri, Carl Adam (1966). "Communication with automata". eng. PhD thesis. Universität Hamburg.