Graph-Based Intermediate Representations: An Overview and Perspectives

Peter Sovietov

About me (in compiler-related context)

- IntellaSys with Chuck Moore (author of Forth language).
  Superoptimizer and a mapper of high-level task graphs for custom multicore chips with 144 PEs.
- JBOG, Origin PC, Corsair.
  DSL compilers for FPGA soft-processors.
- RDI Kvant, MSU, MALT System.
  SW/HW co-design and C-like DSL compilers for domain-specific accelerators.
- RTU MIREA.
  PhD in compiler design. DSLs and compilers for high-level synthesis of pipelined accelerators.

IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.

IR is the heart of the compiler. What is a good IR?

- **Use of normalization.** Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction.** There is no such access if we use IRs based on lists of instructions.

IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction**. There is no such access if we use IRs based on lists of instructions.
- **Executable semantics**. Phi nodes are non-interpretable.

IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction**. There is no such access if we use IRs based on lists of instructions.
- **Executable semantics**. Phi nodes are non-interpretable.
- **Support for different models of computation, including parallel ones**. LLVM IR is RISC-based.

IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction**. There is no such access if we use IRs based on lists of instructions.
- **Executable semantics**. Phi nodes are non-interpretable.
- **Support for different models of computation, including parallel ones**. LLVM IR is RISC-based.
- **Ability to combine different transformations in a single pass**. Register allocation and instruction scheduling are best done together.

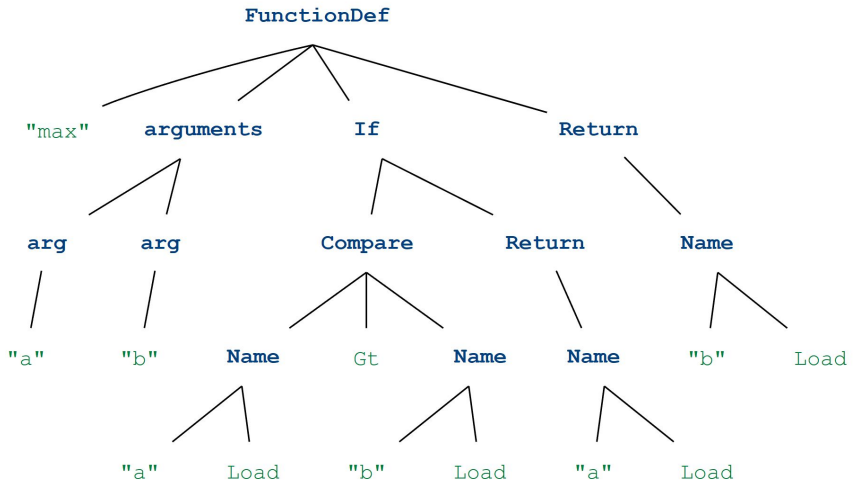
IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction**. There is no such access if we use IRs based on lists of instructions.
- **Executable semantics**. Phi nodes are non-interpretable.
- **Support for different models of computation, including parallel ones**. LLVM IR is RISC-based.
- **Ability to combine different transformations in a single pass**. Register allocation and instruction scheduling are best done together.
- **Support of both high-level and low-level constructs**.
  There is no loop construct in LLVM IR.

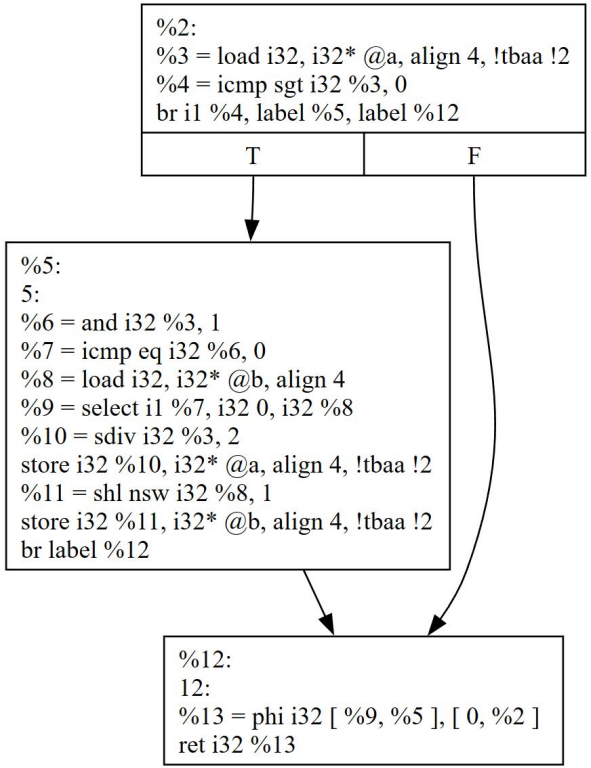
IR is the heart of the compiler. What is a good IR?

- **Use of normalization**. Functionally equal but syntactically different source fragments should be mapped to the same canonical IR code.
- **Easy access to the non-local context of any instruction**. There is no such access if we use IRs based on lists of instructions.
- **Executable semantics**. Phi nodes are non-interpretable.
- **Support for different models of computation, including parallel ones**. LLVM IR is RISC-based.
- **Ability to combine different transformations in a single pass**. Register allocation and instruction scheduling are best done together.
- **Support of both high-level and low-level constructs**.
  There is no loop construct in LLVM IR.
- **Engineering considerations**. Use of performant algorithms, small memory footprint, simple implementation.

What is graph-based IR? These are graphs too...



AST



Hybrid IR:
CFG + SSA + lists of instructions

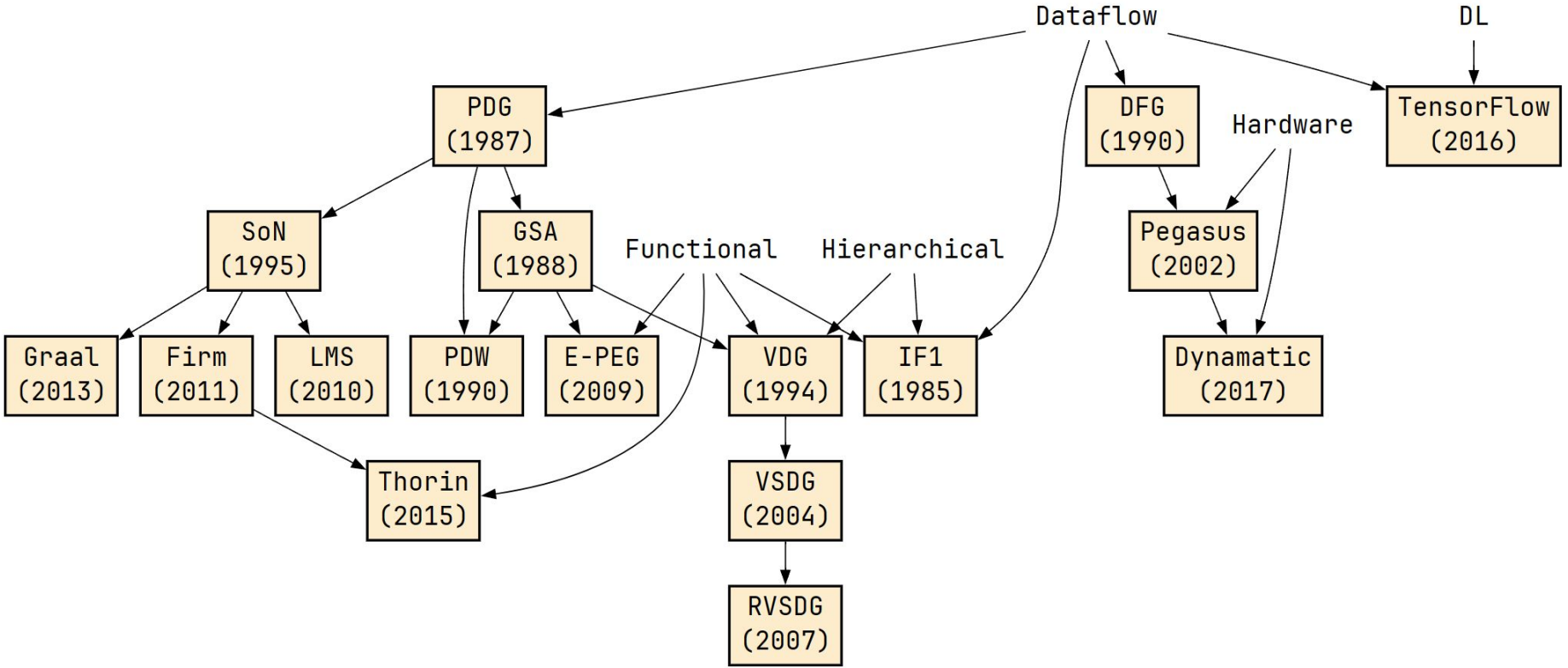
The class of graph-based IRs has many names

- Sea of nodes.
- Graph IR.
- Program dependence graph.
- **Data** dependence graph.
- **Data**-centric IR.
- **Data**flow-centric IR.
- **Data**flow IR.
- …

Graph-based IR

- Single graph structure.
- Using one of the dataflow models.
- Optimizations are done by simple algorithms on graphs.
- Optimizations without a schedule.

Development of graph-based IRs



Main uses: JIT compilers, deep learning compilers, high-level synthesis.

1. Classical dataflow models
2. Program dependence graphs
3. Gated data dependence graphs
4. Perspectives

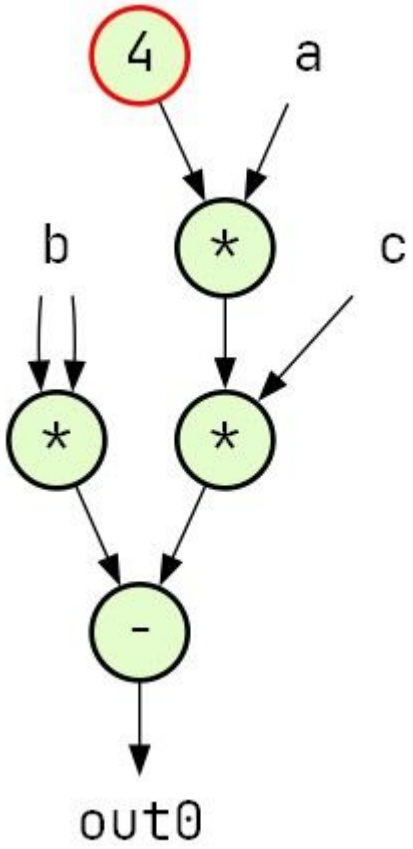
Static dataflow

This model was proposed in the 1970s by Jack Dennis.

Data is produced and consumed by nodes in the form of tokens.

Node firing rules:

1. **All inputs of the node have a token.**
2. **All consumers' inputs connected to the node's output are empty.**



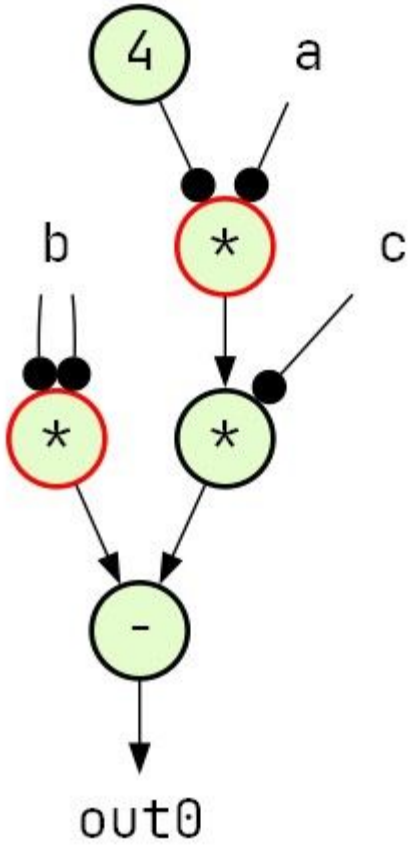
Static dataflow: example of execution (1)

This model was proposed in the 1970s by Jack Dennis.

Data is produced and consumed by nodes in the form of tokens.

Node firing rules:

1. **All inputs of the node have a token.**
2. **All consumers' inputs connected to the node's output are empty.**



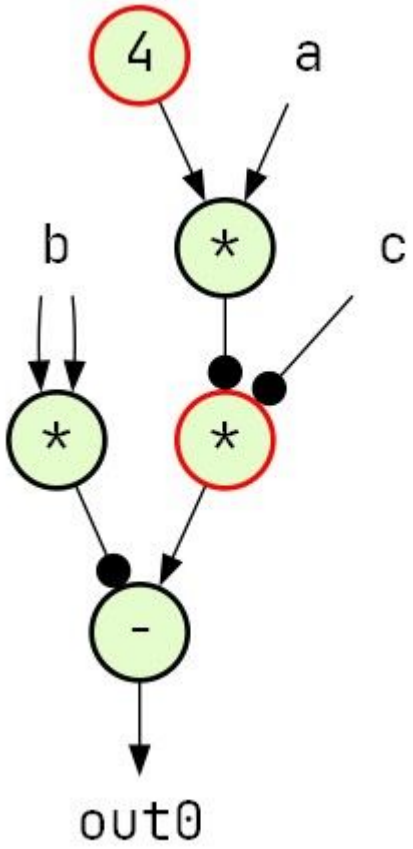
Static dataflow: example of execution (2)

This model was proposed in the 1970s by Jack Dennis.

Data is produced and consumed by nodes in the form of tokens.

Node firing rules:

1. **All inputs of the node have a token.**
2. **All consumers' inputs connected to the node's output are empty.**



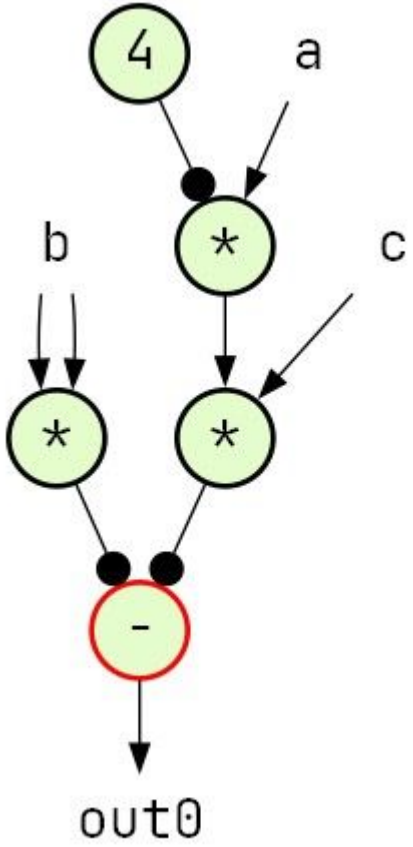
Static dataflow: example of execution (3)

This model was proposed in the 1970s by Jack Dennis.

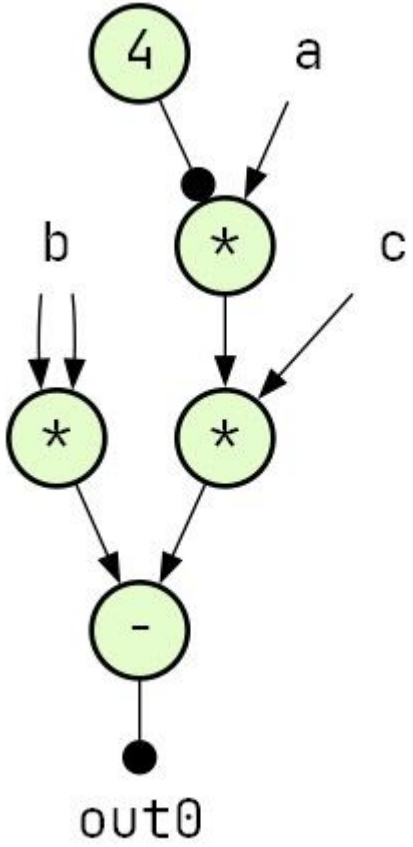
Data is produced and consumed by nodes in the form of tokens.

Node firing rules:

1. **All inputs of the node have a token.**
2. **All consumers' inputs connected to the node's output are empty.**



Static dataflow: example of execution (4)



Lots of parallelism!

In addition to the **data-driven model**, there is also a **demand-driven model**: from outputs to inputs and back.

A toy compiler: linear code to dataflow IR (1)

```
def parse_stmt(tree):
    match tree:
        case Assign([Name(name)], expr):
            env[name] = parse_expr(expr)
        case Return(value):
            add('return', parse_expr(value))
```

```
def add(*node):
    n = next(index)
    graph[n] = node
    return n
```

We use **env** and **graph** tables to get the generated code in **SSA** form;
every expression has a unique index in graph.

A toy compiler: linear code to dataflow IR (2)

```
def parse_stmt(tree):
    match tree:
        case Assign([Name(name)], expr):
            env[name] = parse_expr(expr)
        case Return(value):
            add('return', parse_expr(value))

def parse_expr(tree):
    match tree:
        case Constant(val):
            return add('const', val)
        case Name(name):
            if name not in env:
                env[name] = add('in', name)
            return env[name]
        case BinOp(x, op, y):
            a, b = parse_expr(x), parse_expr(y)
            return add(get_op_name(op), a, b)
```

```
def add(*node):
    n = next(index)
    graph[n] = node
    return n
```

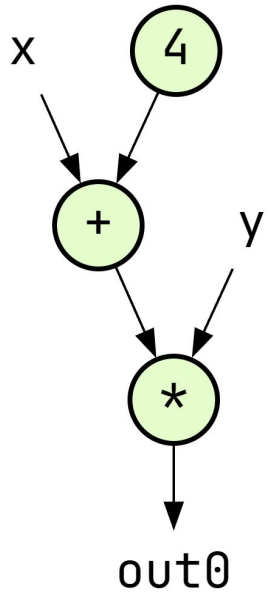
We use **env** and **graph** tables to get the generated code in **SSA** form;
every expression value has a unique index in graph.
This is **LVN** and it can be seen as a form of **symbolic execution**.

A toy compiler: linear code to dataflow IR (3)

```
def parse_stmt(tree):
    match tree:
        case Assign([Name(name)], expr):
            env[name] = parse_expr(expr)
        case Return(value):
            add('return', parse_expr(value))

def parse_expr(tree):
    match tree:
        case Constant(val):
            return add('const', val)
        case Name(name):
            if name not in env:
                env[name] = add('in', name)
            return env[name]
        case BinOp(x, op, y):
            a, b = parse_expr(x), parse_expr(y)
            return add(get_op_name(op), a, b)
```

```
def add(*node):
    n = next(index)
    graph[n] = node
    return n
```

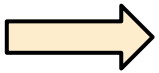
```
>>> parse('a=x; a=y*(a+4); return a')
{0: ('in', 'x'),
 1: ('in', 'y'),
 2: ('const', 4),
 3: ('+', 0, 2),
 4: ('*', 1, 3),
 5: ('return', 4)}
```

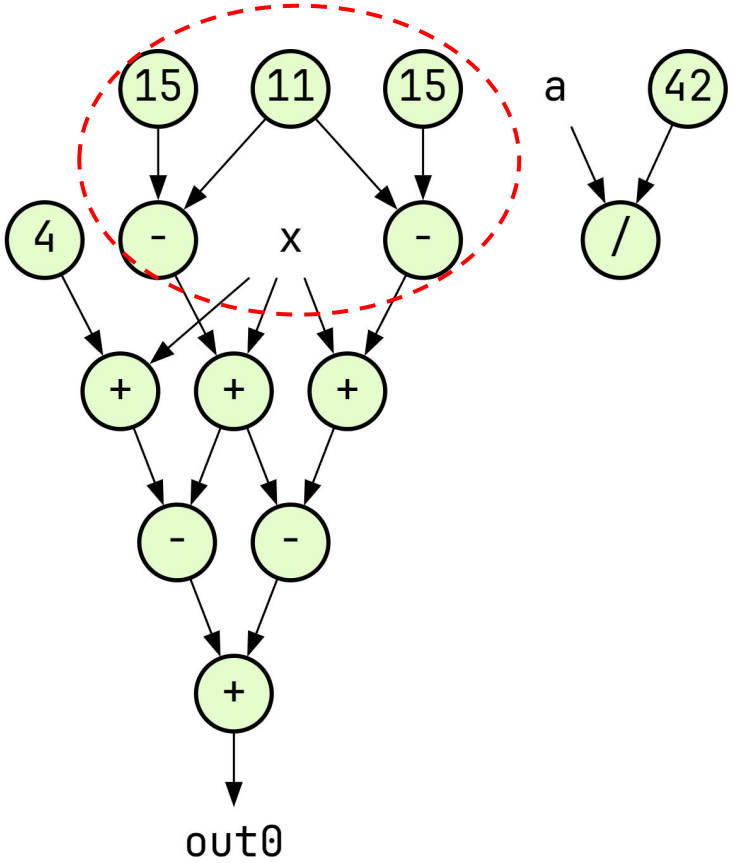


We use **env** and **graph** tables to get the generated code in **SSA** form;
every expression has a unique index in graph.
This is **LVN** and it can be seen as a form of **symbolic execution**.

A toy compiler: common subexpression elimination (1)

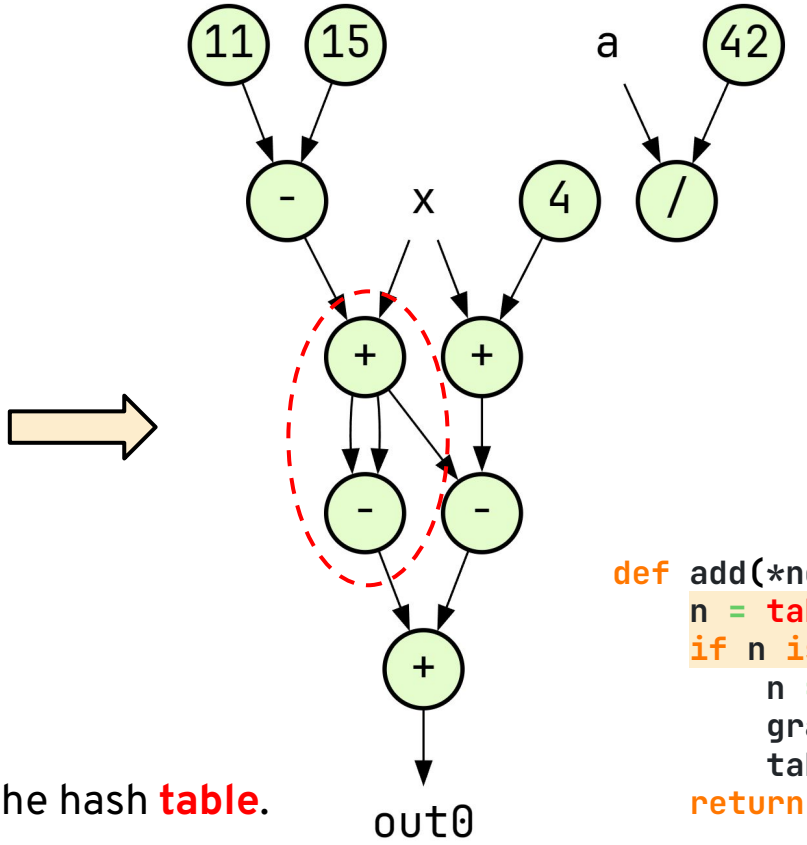
```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```





A toy compiler: common subexpression elimination (2)

```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```

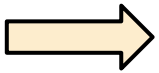


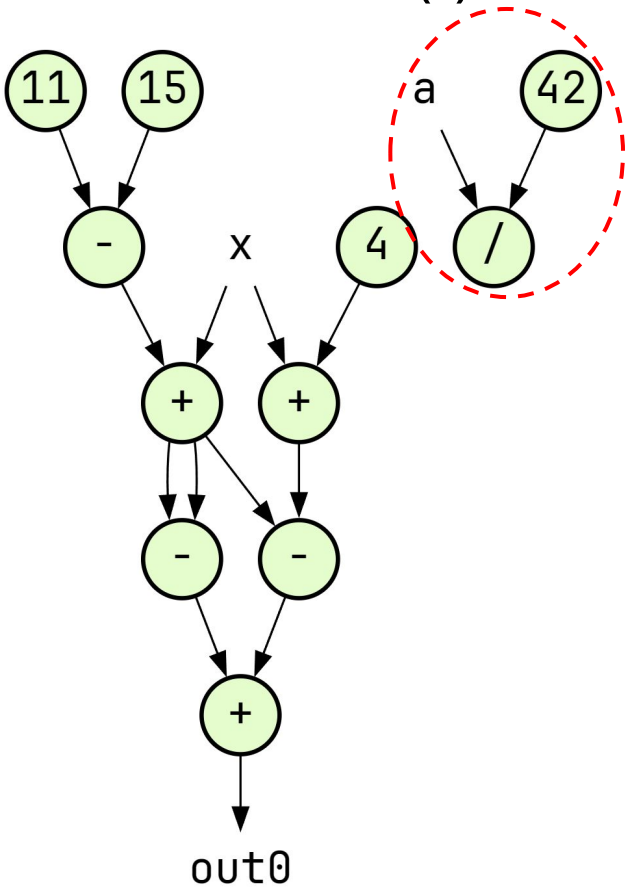

```
def add(*node):
    n = table.get(node)
    if n is None:
        n = next(index)
        graph[n] = node
        table[node] = n
    return n
```

Don't add a new node if it's already in the hash **table**.

A toy compiler: dead code elimination (1)

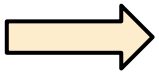
```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```

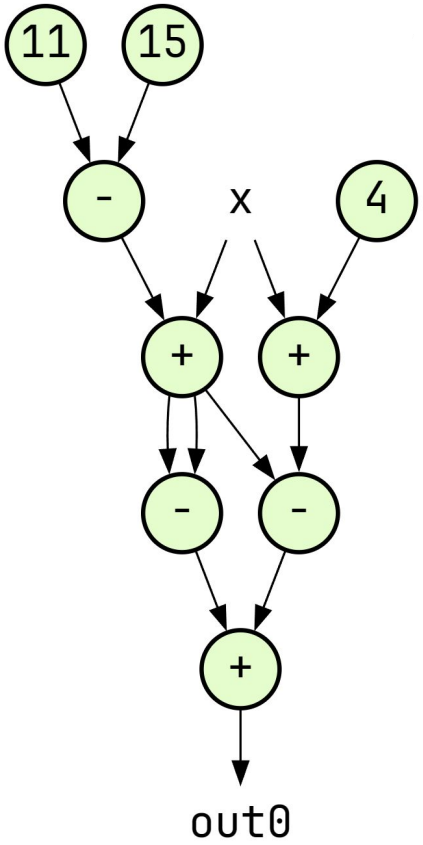




A toy compiler: dead code elimination (2)

```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```



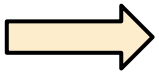


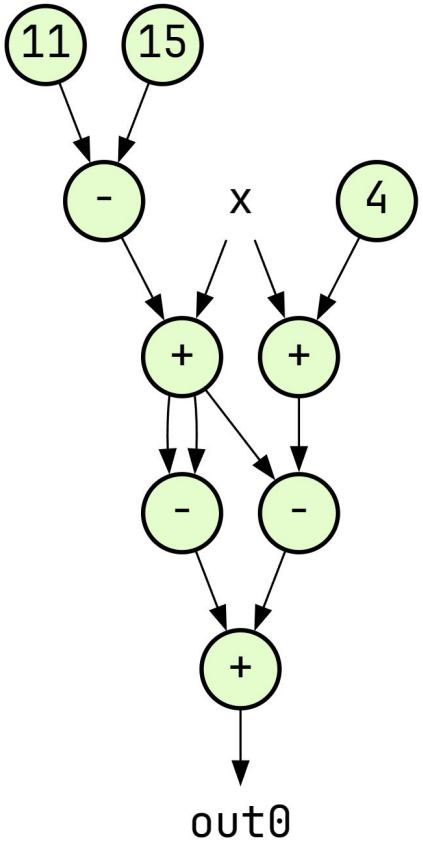

```
def dce(graph):
    scheduled = toposort(graph) # mark
    return {i: n for i, n in graph.items()
            if i in scheduled}  # sweep
```

A toy compiler: instruction scheduling

We start scheduling from the program results (in a demand-driven style).

```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```



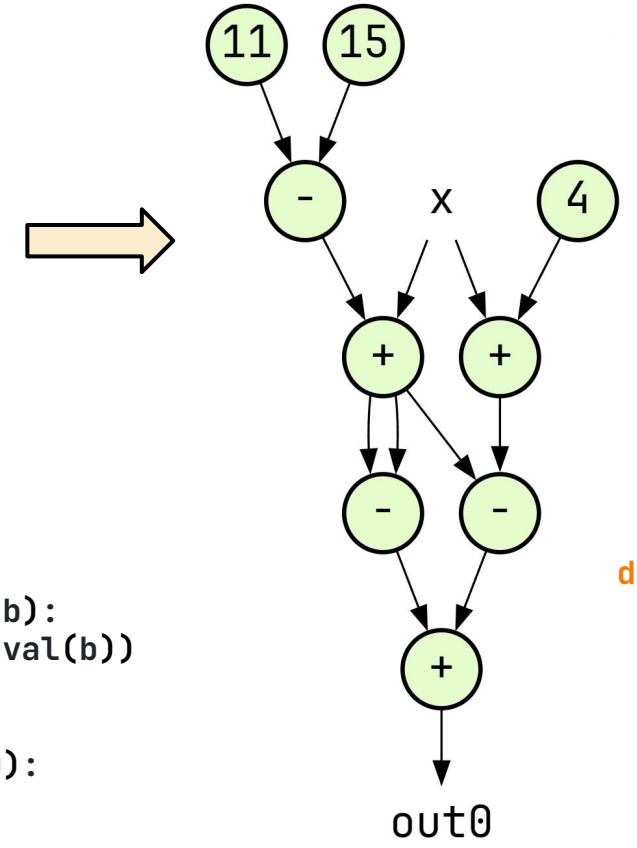



```
t1 = x
t2 = 15
t0 = 11
t3 = t2 - t0
t4 = t1 + t3
t11 = t4 - t4
t8 = 4
t9 = t1 + t8
t10 = t9 - t4
t12 = t11 + t10
return t12
```

```
def dce(graph):
    scheduled = toposort(graph) # mark
    return {i: n for i, n in graph.items()
            if i in scheduled}  # sweep
```

A toy compiler: constant folding and propagation (1)

```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```

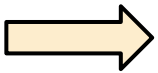


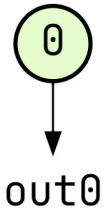

```
def fold(node):
    match node:
        case ('-', a, b) if consts(a, b):
            return ('const', val(a) - val(b))
        case ('-', a, b) if a == b:
            return ('const', 0)
        case ('-', a, b) if equal(b, 0):
            return graph[a]
    return node
```

```
def add(*node):
    node = fold(node)
    n = table.get(node)
    if n is None:
        n = next(index)
        graph[n] = node
        table[node] = n
    return n
```

A toy compiler: constant folding and propagation (2)


```
b = 11
y = x + (15 - b)
z = a / 42
a = x + 4 - y
y = y - (x + (15 - b)) + a
return y
```






```
t8 = 0
return t8
```

```
def fold(node):
    match node:
        case ('-', a, b) if consts(a, b):
            return ('const', val(a) - val(b))
        case ('-', a, b) if a == b:
            return ('const', 0)
        case ('-', a, b) if equal(b, 0):
            return graph[a]
    return node
```

```
def add(*node):
    node = fold(node)
    n = table.get(node)
    if n is None:
        n = next(index)
        graph[n] = node
        table[node] = n
    return n
```

A toy compiler: what we already got

- **Combined CSE and constant folding/propagation at the parsing time.**
- **Combined DCE and instruction scheduling.**

We got some important optimizations that are used in graph-based IRs. But at the level of a basic block.

The main difference between graph-based IRs is how they work outside the basic blocks.

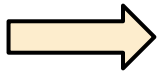
A toy compiler: memory operations (1)

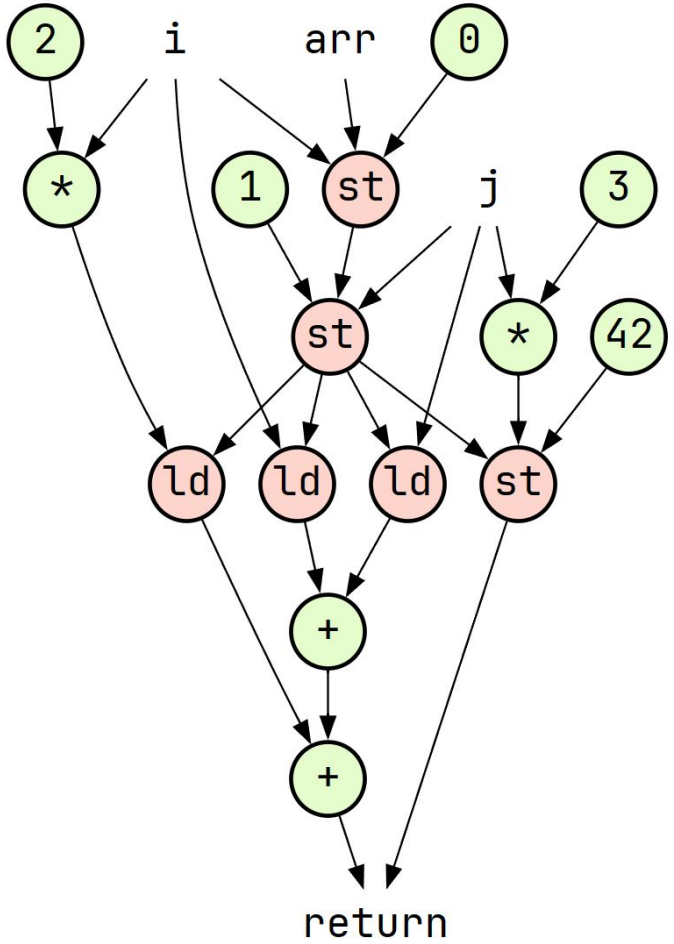
We will treat arrays as values, like in functional programming, so every store operation will update the **env** table.

```
def parse_stmt(tree):
    match tree:
        ...
        case Assign([Subscript(Name(id=name) as val, slice)], expr):
            env[name] = add('store', parse_expr(val), parse_expr(slice),
                            parse_expr(expr))
        ...

def parse_expr(tree):
    match tree:
        ...
        case Subscript(Name(id=name) as val, slice):
            return add('load', parse_expr(val), parse_expr(slice))
```

A toy compiler: memory operations (2)

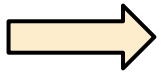
```
arr[i] = 0
arr[j] = 1
r = arr[i] + arr[j]
r = r + arr[i * 2]
arr[j * 3] = 42
return r, arr
```

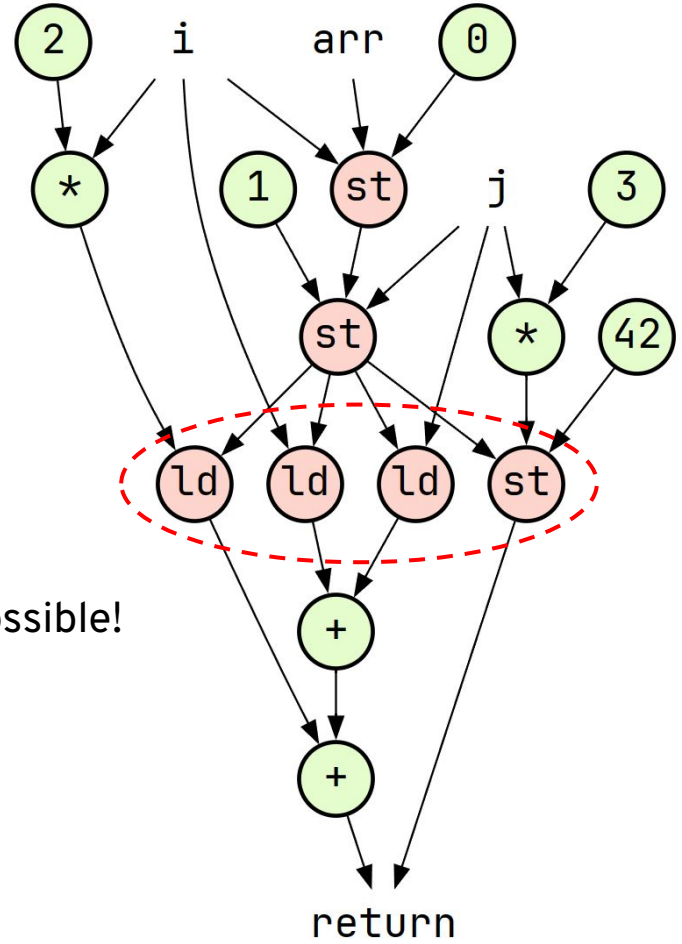




A toy compiler: memory operations (3)

```
arr[i] = 0
arr[j] = 1
r = arr[i] + arr[j]
r = r + arr[i * 2]
arr[j * 3] = 42
return r, arr
```





Incorrect ld/st ordering is possible!

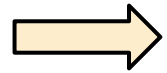
A toy compiler: memory operations (4)

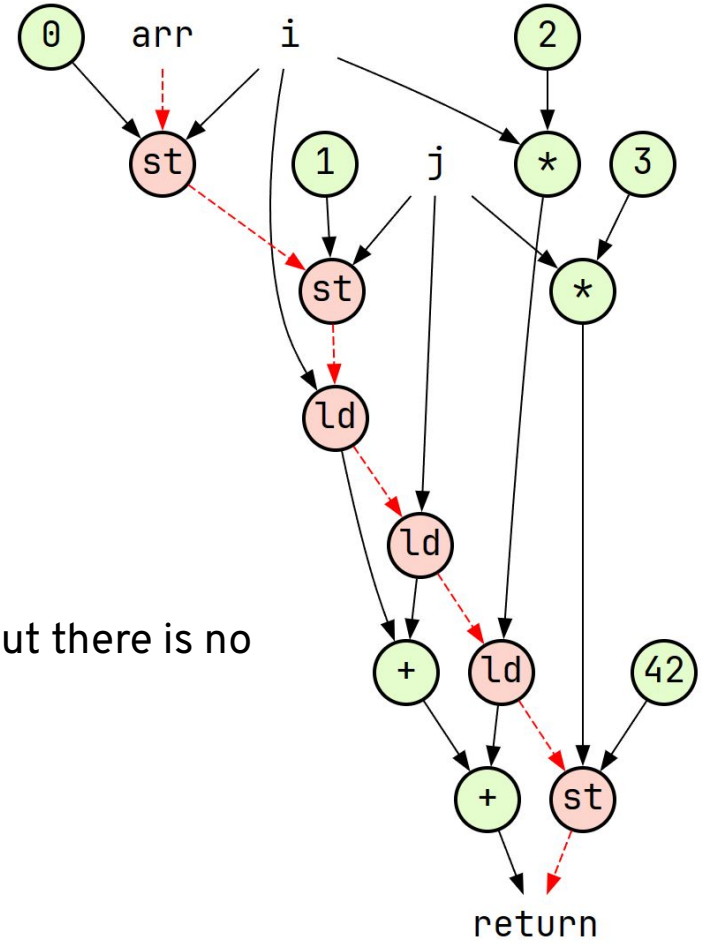
Let's update the **env** table in the load operation too.
Now we have a state value, a **memory dependency**,
that memory nodes send to each other.

```
def parse_expr(tree):
    match tree:
        ...
        case Subscript(Name(id=name) as val, slice):
            env[name] = add('load', parse_expr(val), parse_expr(slice))
            return env[name]
```

A toy compiler: memory operations (5)

```
arr[i] = 0
arr[j] = 1
r = arr[i] + arr[j]
r = r + arr[i * 2]
arr[j * 3] = 42
return r, arr
```





Now this is correct, but there is no memory parallelism!

A toy compiler: memory parallelism (1)

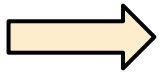
The **mem** table contains memory operations of the **same type** with a **common input memory state**.

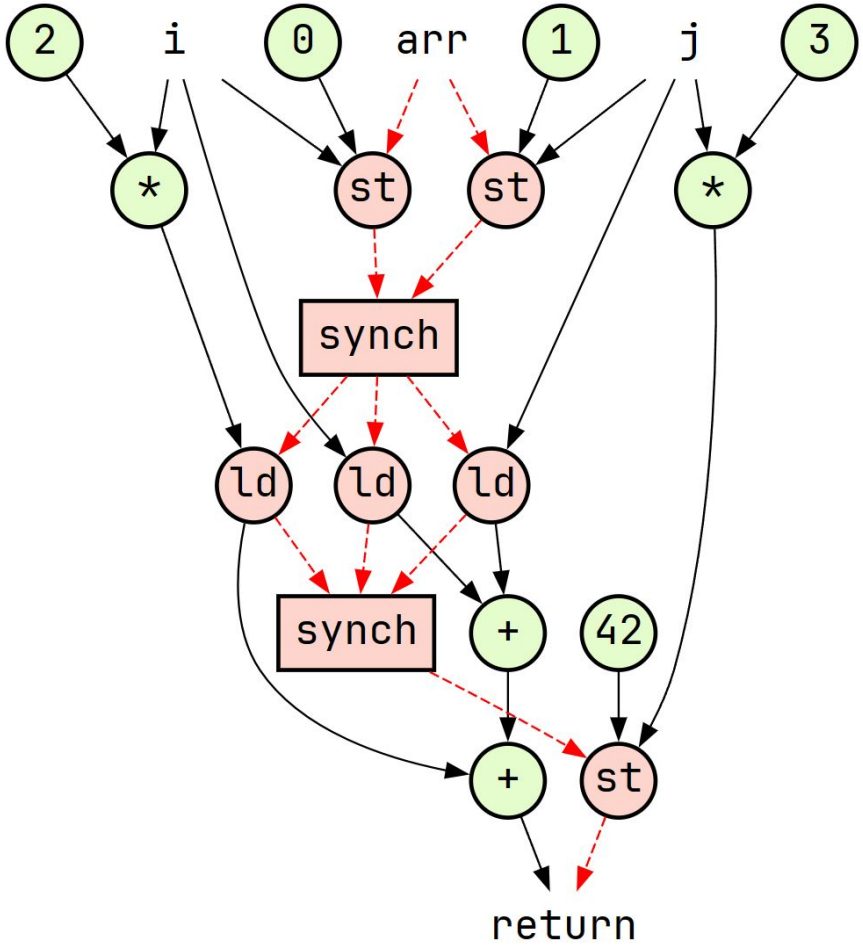
The **synch** node is used to produce a **new memory state** with inputs from **all mem table members**.

```
def synchronize(op, name):
    ...
    if not all(node_op(n) == op for n in mem[name]):
        env[name] = add_synch(name)
        mem[name] = []

def parse_expr(tree):
    match tree:
        ...
        case Subscript(Name(id=name), slice):
            synchronize('load', name)
            n = add('load', env[name], parse_expr(slice))
            mem[name].append(n)
            return n

def parse_stmt(tree):
    match tree:
        ...
        case Assign([Subscript(Name(id=name), slice)], expr):
            synchronize('store', name)
            n = add('store', env[name], parse_expr(slice),
                    parse_expr(expr))
            mem[name].append(n)
        ...
```

A toy compiler: memory parallelism (2)

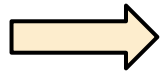
```
arr[i] = 0
arr[j] = 1
r = arr[i] + arr[j]
r = r + arr[i * 2]
arr[j * 3] = 42
return r, arr
```

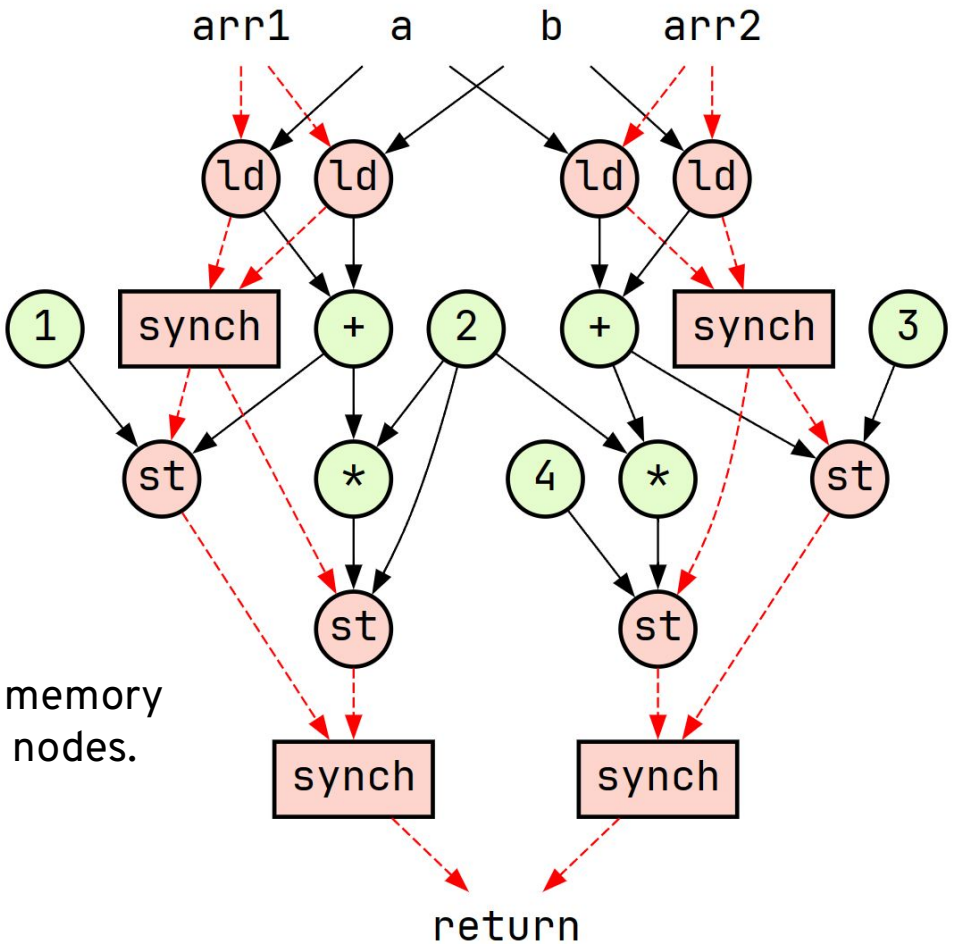




A toy compiler: memory parallelism (3)

```
x = arr1[a] + arr1[b]
y = arr2[a] + arr2[b]
arr1[x] = 1
arr1[2 * x] = 2
arr2[y] = 3
arr2[2 * y] = 4
return arr1, arr2
```



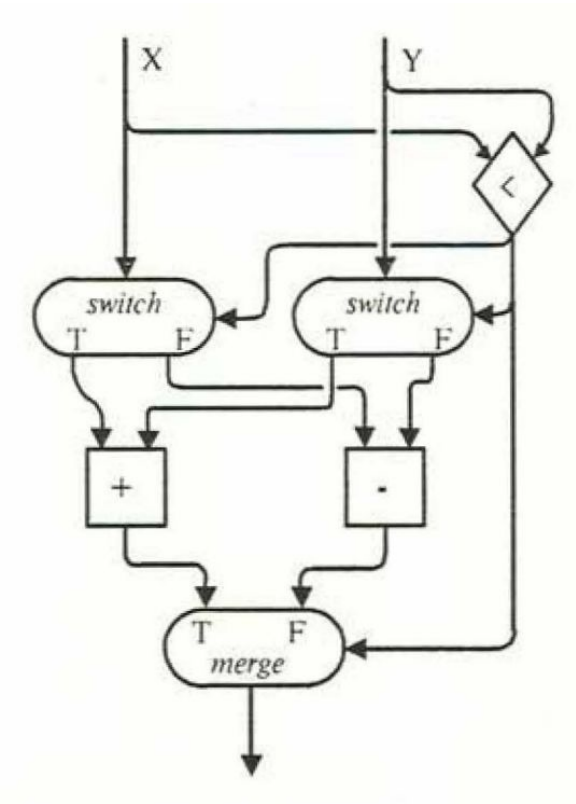


Different arrays have different memory states and use their own synch nodes.

Static dataflow: conditionals

Switch node has data input, two outputs and is strict only on its predicate input.
Predicate value determines the output port for the input data token.

```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



Merge node is strict only on its predicate input.
Predicate value determines the input port for the data token.
It may also be called a **multiplexer** or a **gated phi node**.

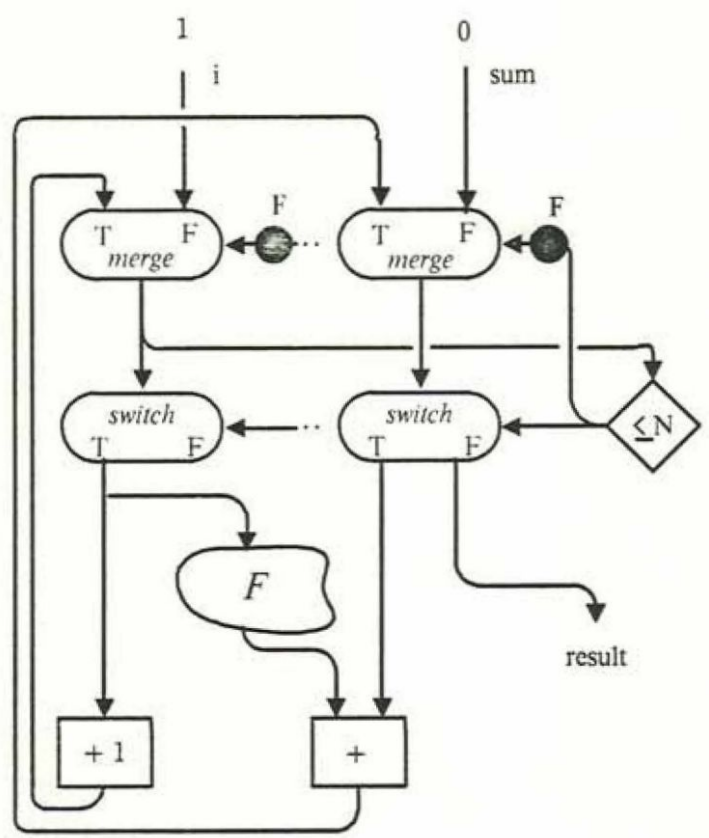
Culler, David E. "Dataflow architectures." (1986).

Static dataflow: loops

Switch node has two outputs and is strict only on its predicate input.
Predicate value determines the output port for the input data token.

```
i = 0
sum = 0
while i ⩽ N:
    sum += F(i)
    i = i + 1
return sum
```

Merge node is strict only on its predicate input.
Predicate value determines the input port for the data token.
It may also be called a **multiplexer** or a **gated phi node**.



Culler, David E. "Dataflow architectures." (1986).

A toy compiler: nodes with multiple output ports

The switch node has two outputs. With the current compiler design, it's possible to have only one output.

In the next iteration of the compiler, we define the input port as a pair **(node index, node output port)**.

An example:

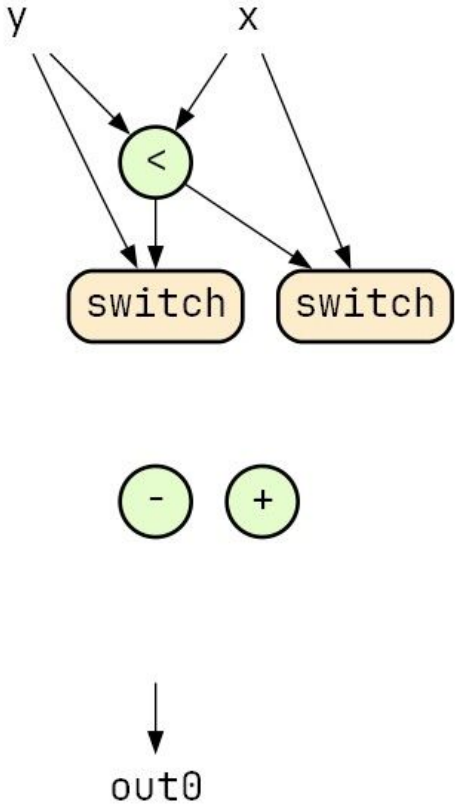
```
...
5 node(op='switch', ins=(...))
6 node(op='switch', ins=(...))
7 node(op='merge', ins=((...), (5, 0), (6, 1)))
...
```

A toy compiler: conditionals (1)

We use **ins** and **outs** tables to keep the input and output variables sets for true and false blocks while parsing.

For each input name in both blocks:

1. **Create a switch node with the external value of the name as an input.**

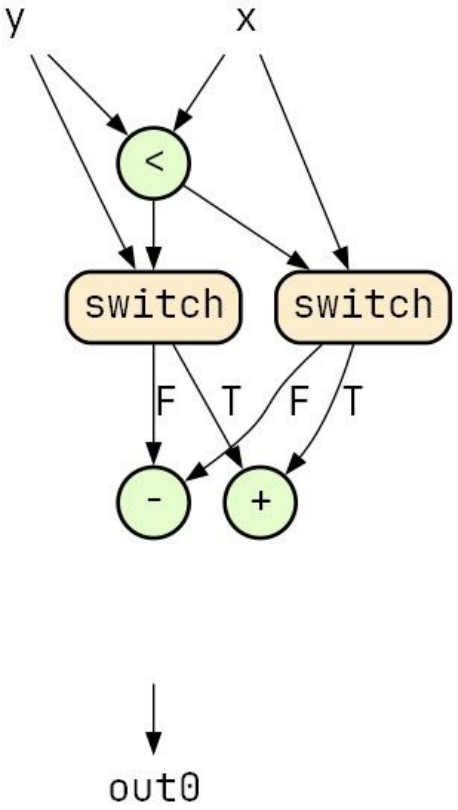


A toy compiler: conditionals (2)

We use **ins** and **outs** tables to keep the input and output variables sets for true and false blocks while parsing.

For each input name in both blocks:

1. Create a switch node with the external value of the name as an input.
2. **Connect the switch node with the block's value using the corresponding output port of the switch.**



A toy compiler: conditionals (3)

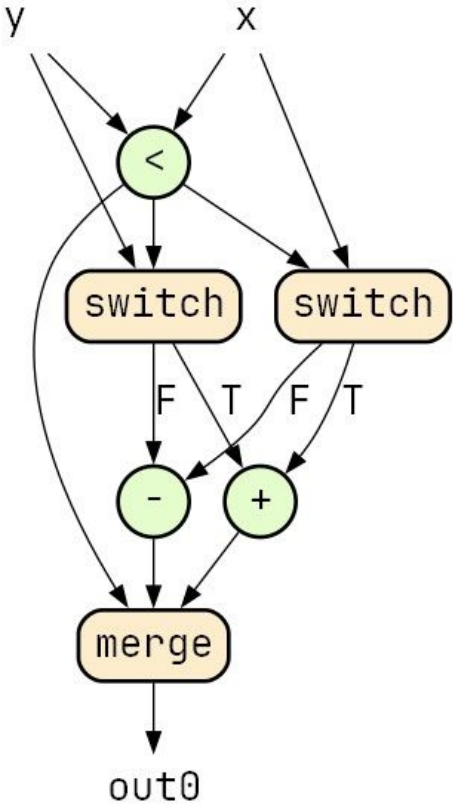
We use **ins** and **outs** tables to keep the input and output variables sets for true and false blocks while parsing.

For each input name in both blocks:

1. Create a switch node with the external value of the name as an input.
2. Connect the switch node with the block's value using the corresponding output port of the switch.

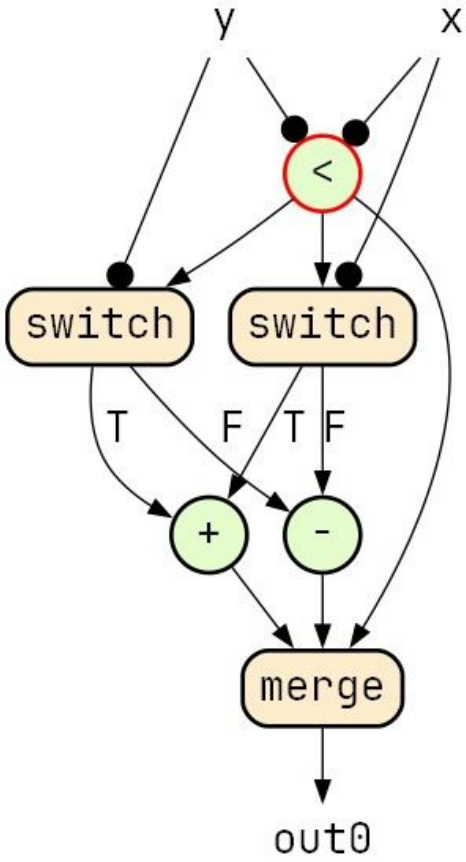
For each output name in both blocks:

1. **Compare two blocks' values, if they are not equal, create a merge node to store the combined value under the name from the external table.**



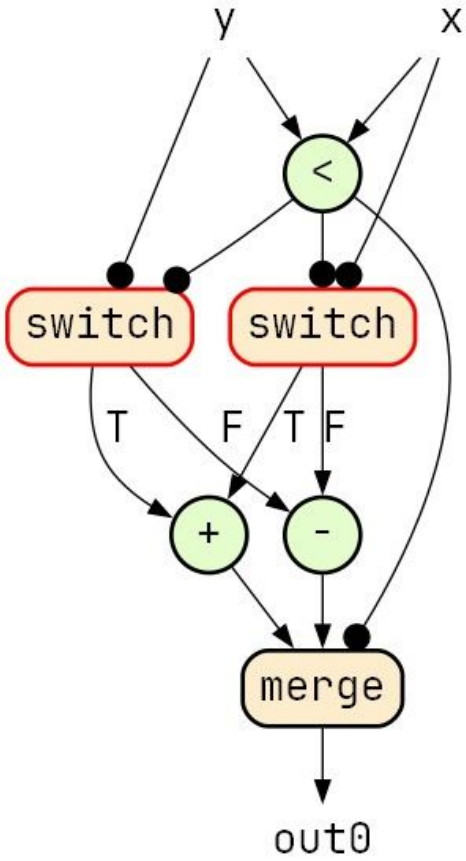
A toy compiler: conditionals, an example (1)

```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



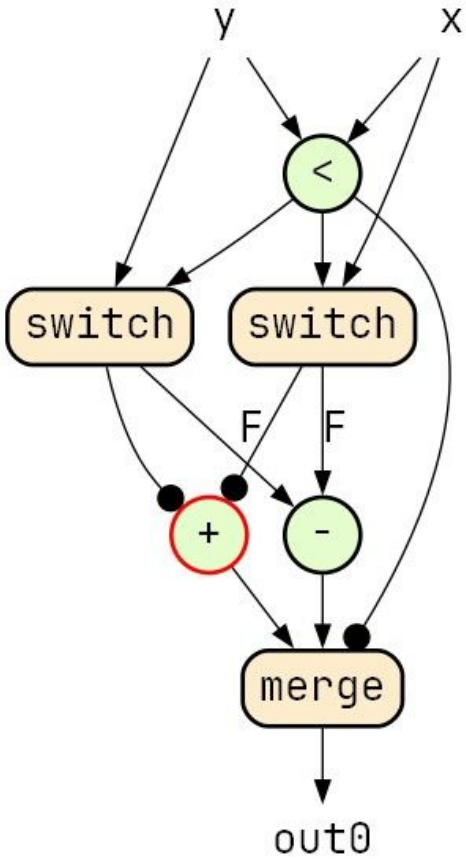
A toy compiler: conditionals, an example (2)

```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



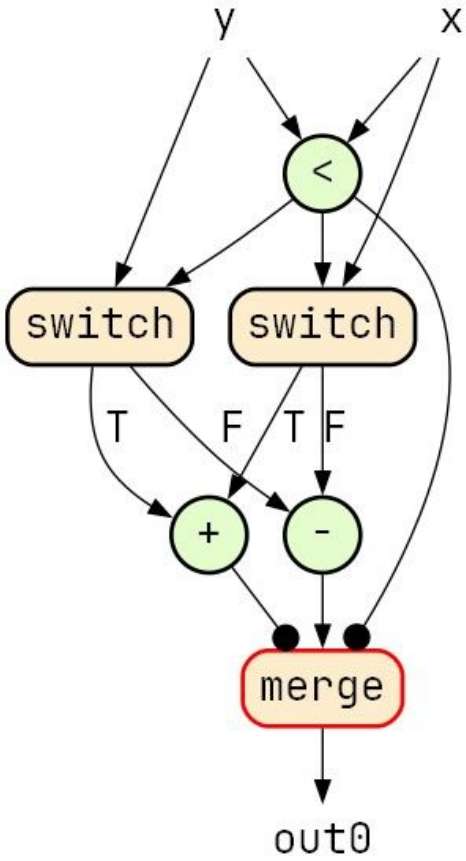
A toy compiler: conditionals, an example (3)

```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



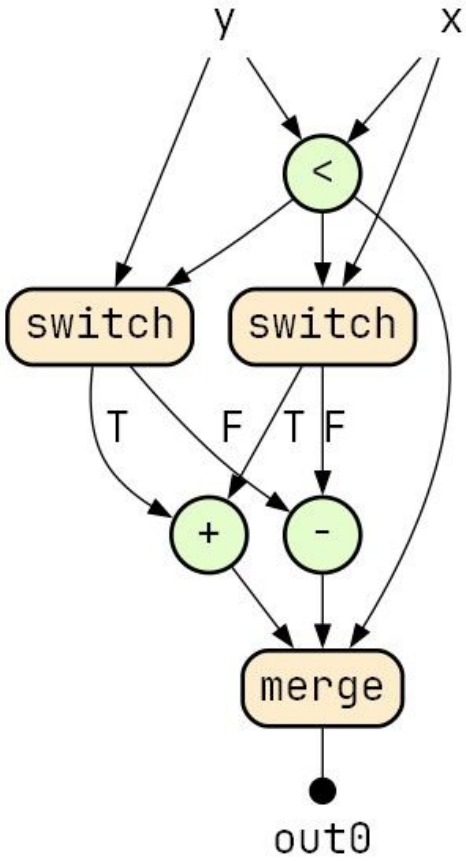
A toy compiler: conditionals, an example (4)

```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



A toy compiler: conditionals, an example (5)

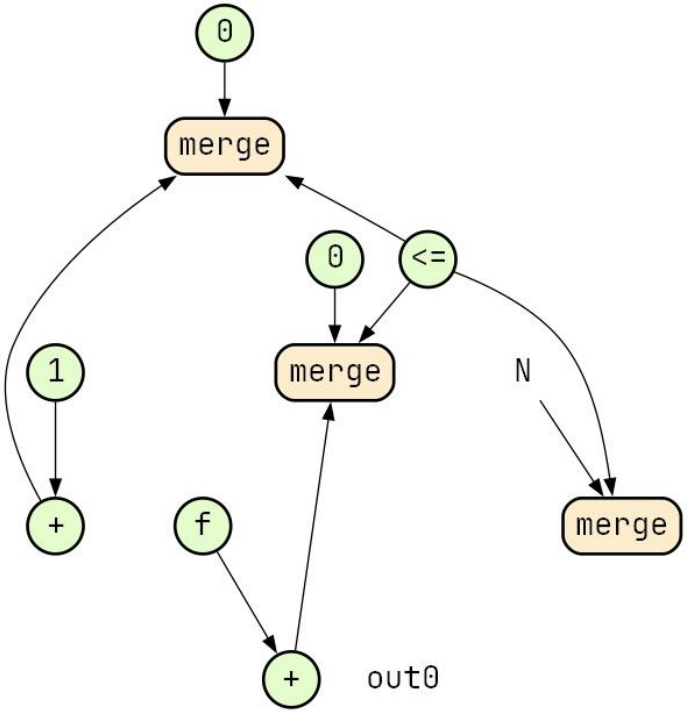
```
if x < y:
    r = x + y
else:
    r = x - y
return r
```



A toy compiler: loops (1)

For each input and output name in the loop body:

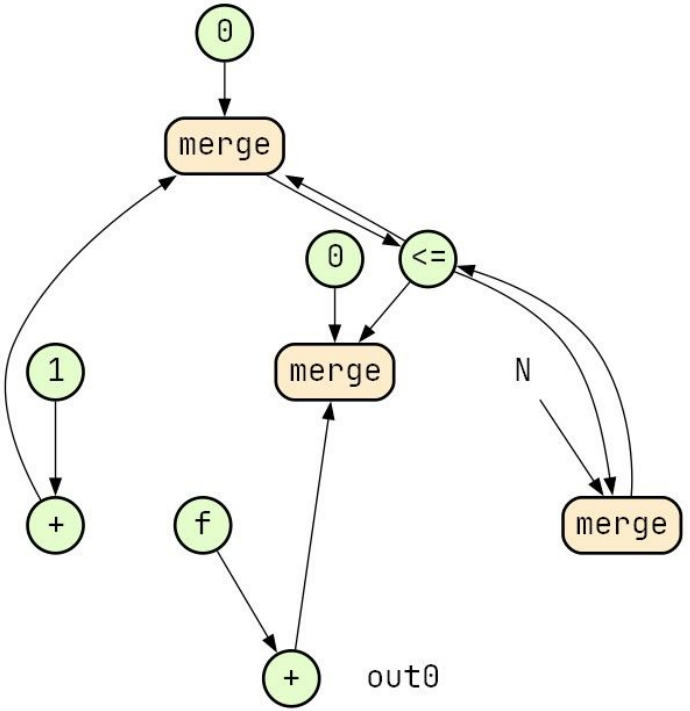
1. **Create a merge node to combine a loop body output value with its external value. Connect the predicate output value with the merge input.**



A toy compiler: loops (2)

For each input and output name in the loop body:

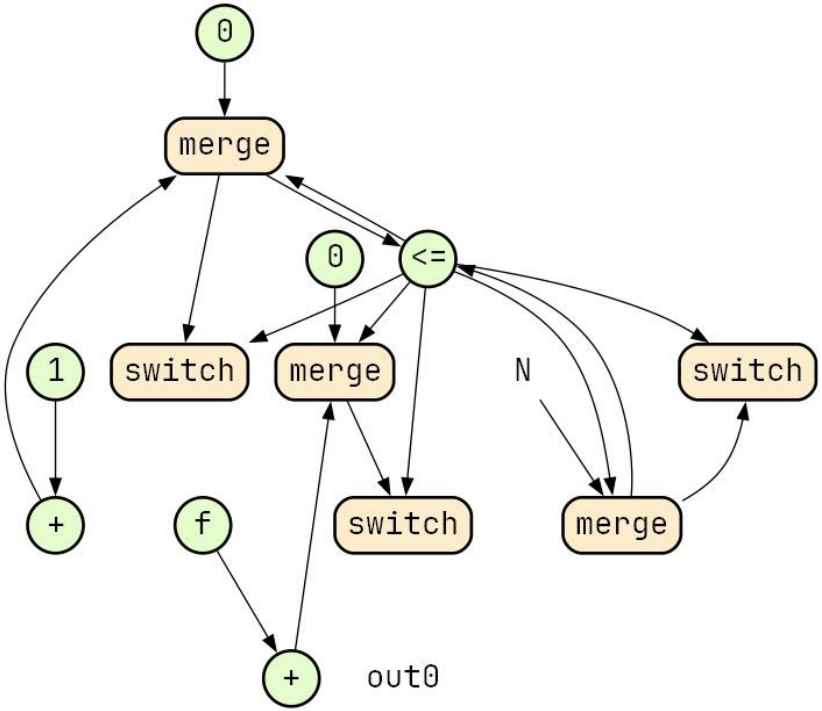
1. Create a merge node to combine a loop body output value with its external value. Connect the predicate output value with the merge input.
2. **Connect the merge node output with a predicate input value.**



A toy compiler: loops (3)

For each input and output name in the loop body:

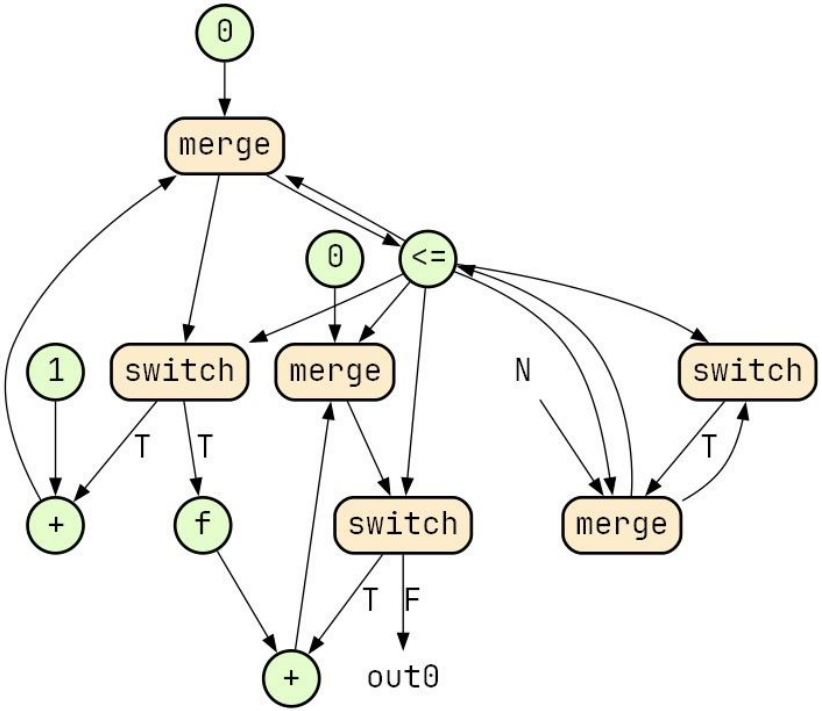
1. Create a merge node to combine a loop body output value with its external value. Connect the predicate output value with the merge input.
2. Connect the merge node output with a predicate input value.
3. **Create a switch node with inputs from the predicate output value and the merge node output.**



A toy compiler: loops (4)

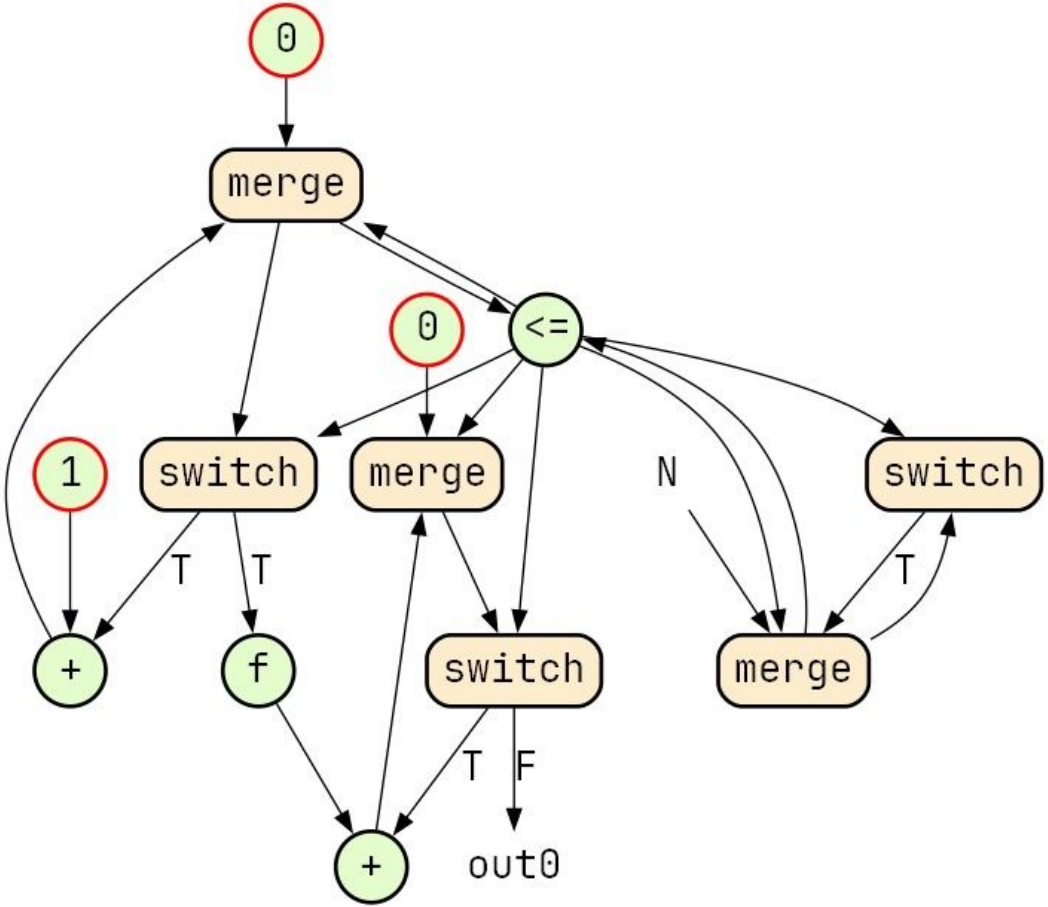
For each input and output name in the loop body:

1. Create a merge node to combine a loop body output value with its external value. Connect the predicate output value with the merge input.
2. Connect the merge node output with a predicate input value.
3. Create a switch node with inputs from the predicate output value and the merge node output.
4. **Connect the output T of the switch node to the loop body input value and store the value on the output F under the name from the external table.**



A toy compiler: problem with merge (1)

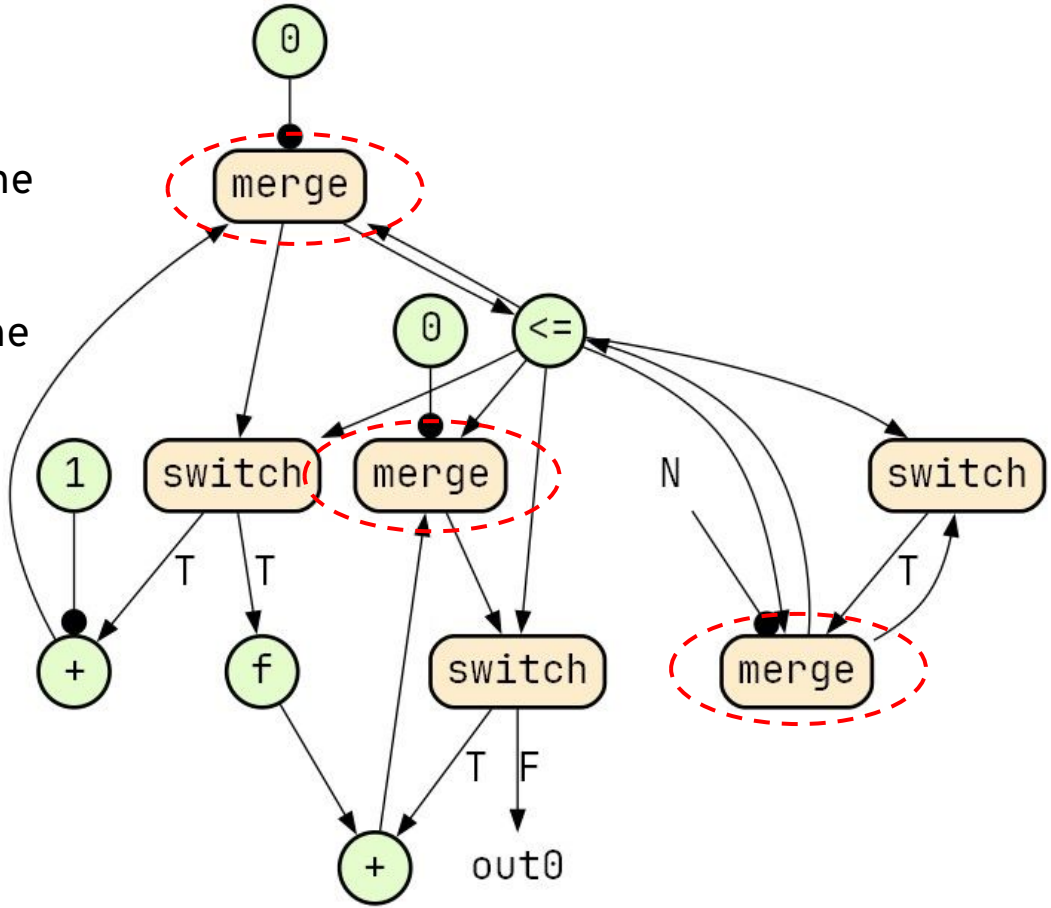
```
i = 0
sum = 0
while i ⩽ N:
    sum += f(i)
    i = i + 1
return sum
```



A toy compiler: problem with merge (2)

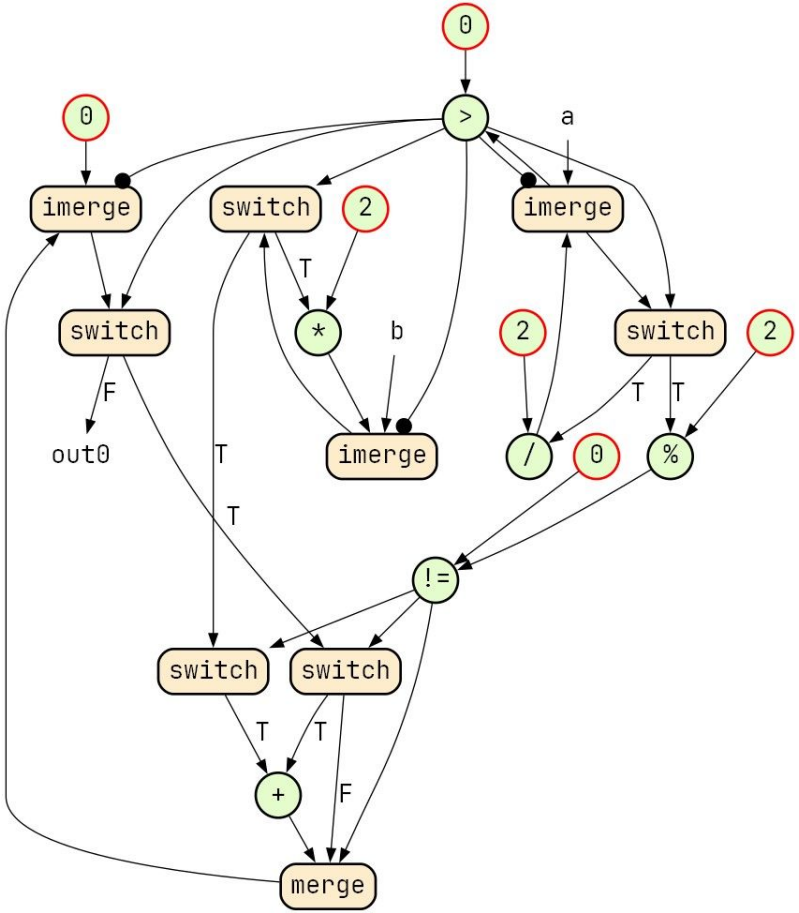
The merge node **infinitely waits** for the predicate value, to send initial value into the loop body.
We will use **imerge** node that unconditionally reads the initial value at the start of the program's execution.

```
i = 0
sum = 0
while i ⩽ N:
    sum += f(i)
    i = i + 1
return sum
```



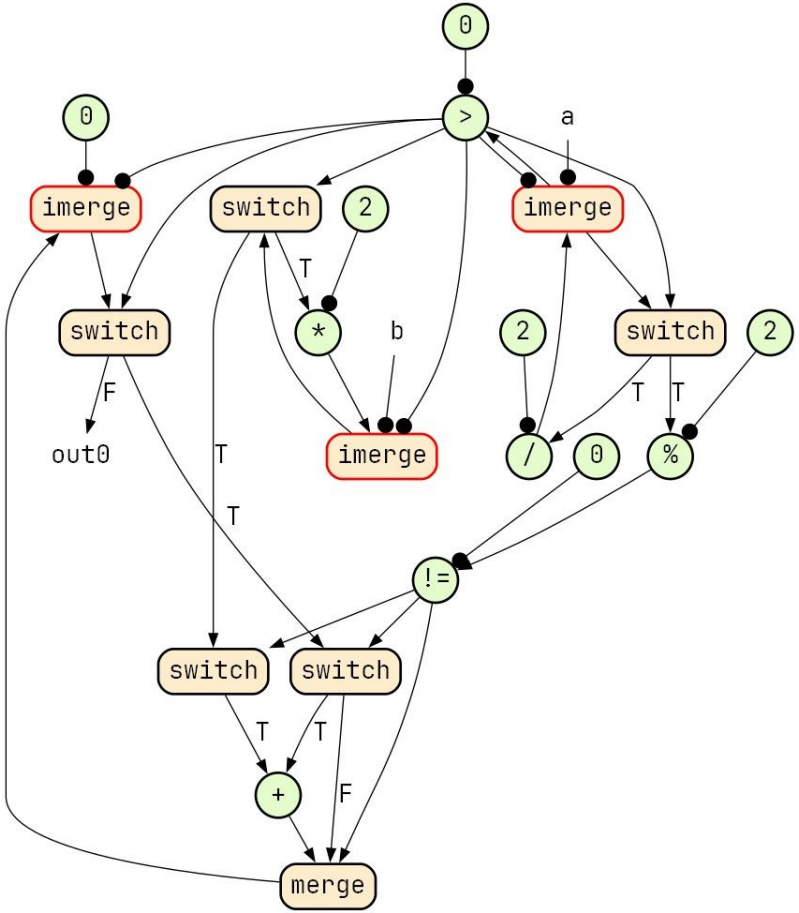
A toy compiler: imerge example (1)

```
r = 0
while a > 0:
    if a % 2 =/= 0:
        r = r + b
    a = a / 2
    b = b * 2
return r
```



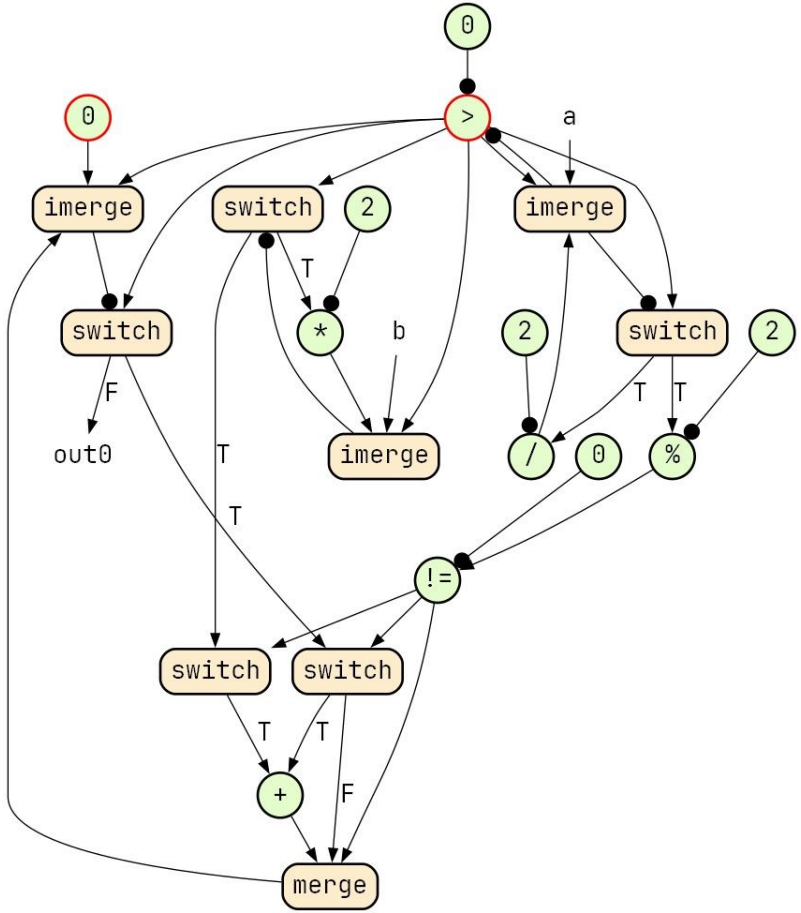
A toy compiler: imerge example (2)

```
r = 0
while a > 0:
    if a % 2 =/= 0:
        r = r + b
    a = a / 2
    b = b * 2
return r
```



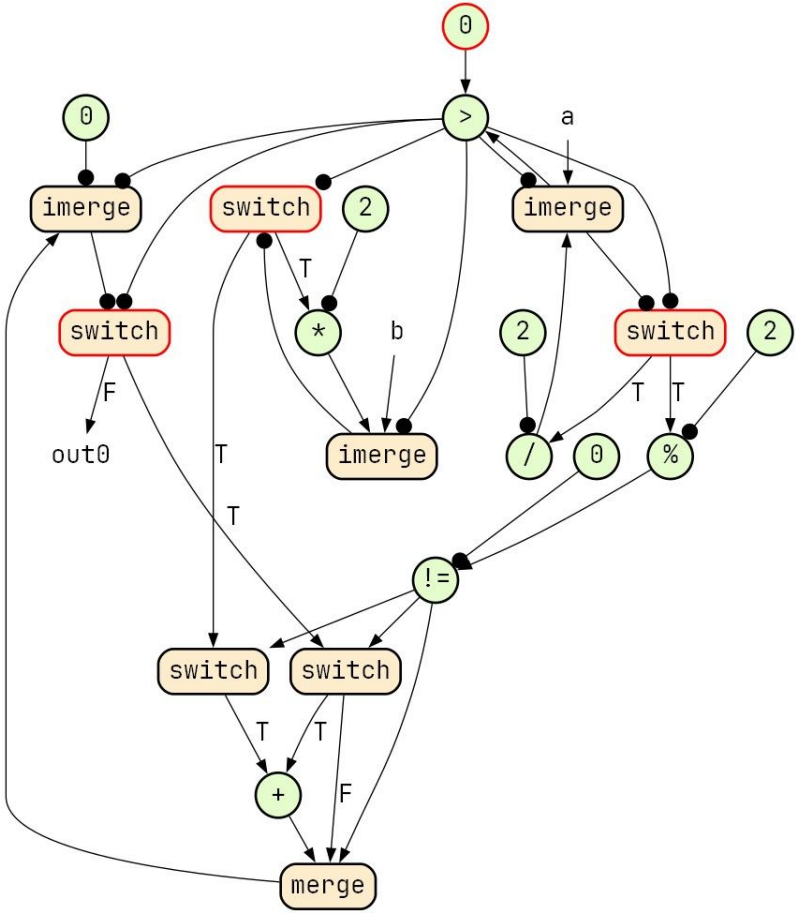
A toy compiler: imerge example (3)

```
r = 0
while a > 0:
    if a % 2 =/= 0:
        r = r + b
    a = a / 2
    b = b * 2
return r
```



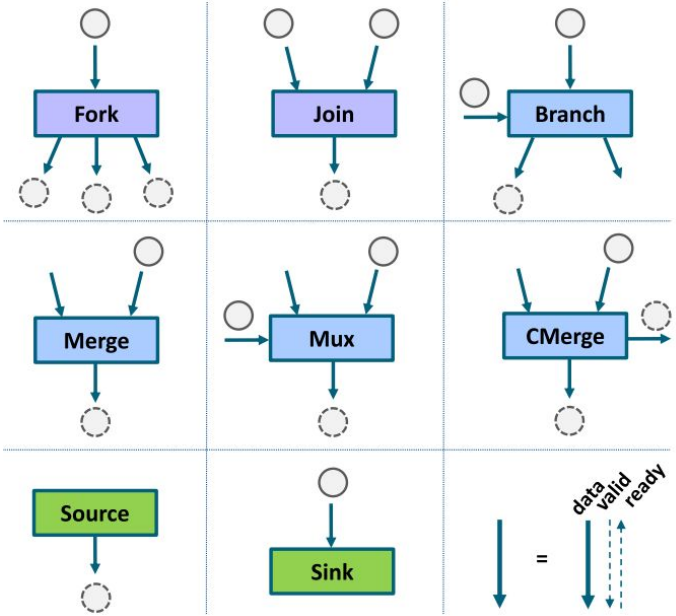
A toy compiler: imerge example (and so on...)

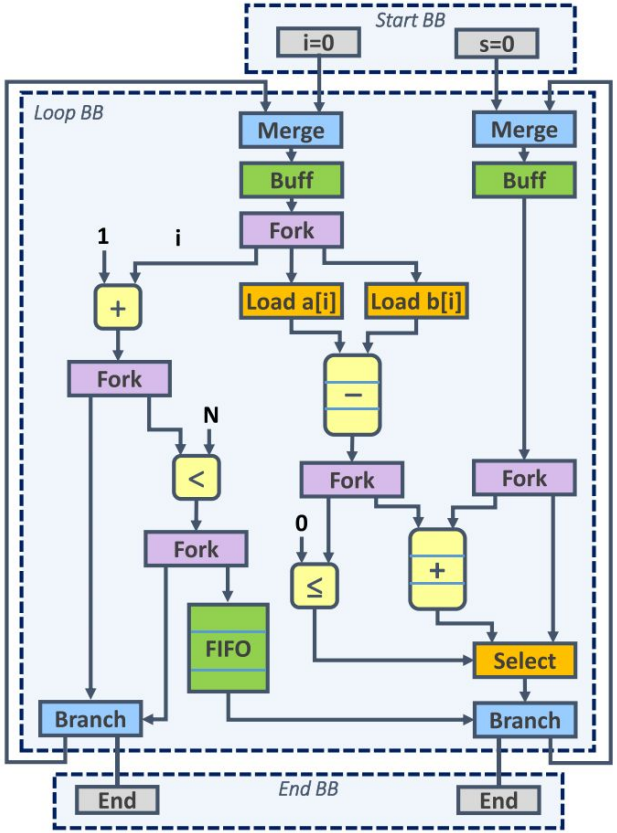
```
r = 0
while a > 0:
    if a % 2 =/= 0:
        r = r + b
    a = a / 2
    b = b * 2
return r
```



Classical dataflow IRs today: Dynamatic

An HLS tool to compile C/C++ code into dynamically scheduled FPGA circuits.

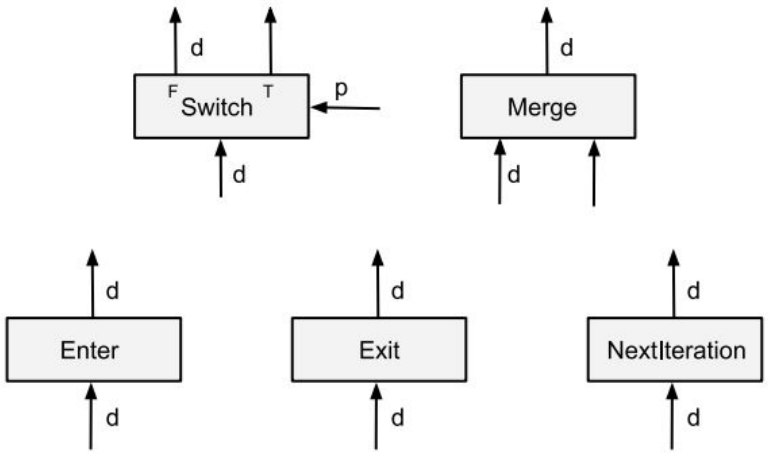


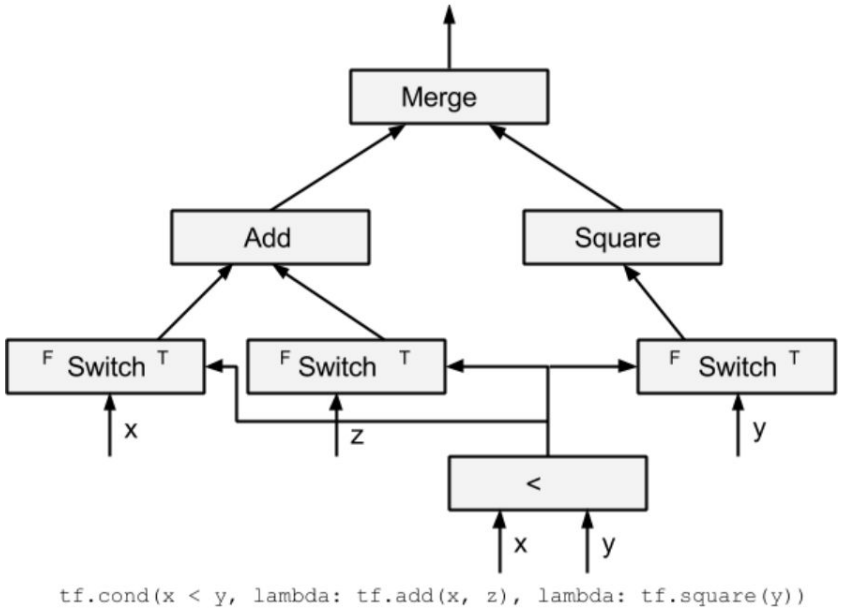


Josipović, Lana, Andrea Guerrieri, and Paolo Ienne. "From C/C++ code to high-performance dataflow circuits." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.7 (2021): 2142-2155.

Classical dataflow IRs today: TensorFlow

A compiler of machine learning models into dynamic dataflow representation.





Authors, TensorFlow. "Implementation of Control Flow in TensorFlow." *Nov* 1 (2016): 18.

Classical dataflow IR vs. modern CPUs

Modern processors use **dataflow execution** model on the **microarchitectural level**. But the CPUs still use **ISAs** based on the **sequential model**!

All modern and advanced **compilers** convert source code through various stages and representation **into an internal data-flow representation** [...]
The compiler backend converts that back to an imperative representation, i.e. machine code. [...]
A **modern CPU** uses super-scalar and out-of-order execution. So the first thing it has to do, is to perform data-flow analysis on the machine code to turn that **back into an (implicit) data-flow representation**! Otherwise the CPU cannot analyze the dependencies between instructions. Sounds wasteful? Oh, yes, it is.

Mike Pall, the LuaJIT author

https://www.freelists.org/post/luajit/Ramblings-on-languages-and-architectures-was-Re-any-benefit-to-throwing-off-lua51-constraints

Dataflow-based “LLVM IR” from the past: IF1

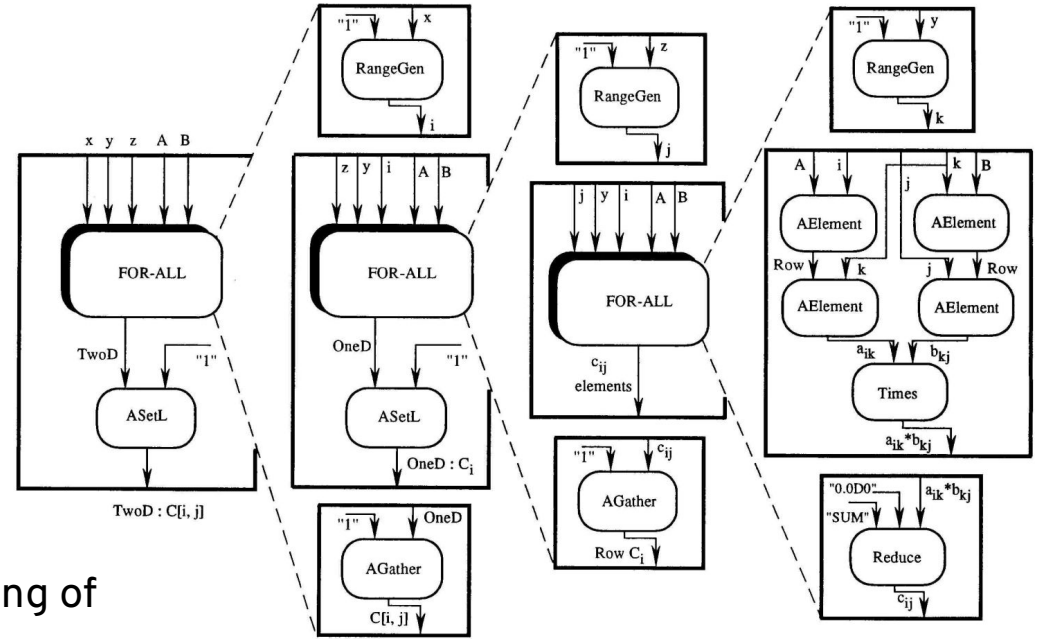
Originally designed for the SISAL single assignment (functional) language (an alternative to Fortran in HPC).

Uses a **hierarchical and acyclic graph** of simple and complex nodes.

Complex nodes:

- If-else.
- Case.
- While.
- Do-while.
- For-all.

Global CSE based on isomorphism checking of subgraphs of complex nodes.



Raisinghani, Manoj H. "Allocation of SISAL program graphs to processors using BLAS." (1994).

1. Classical dataflow models
2. Program dependence graphs
3. Gated data dependence graphs
4. Perspectives

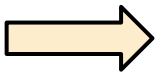
Control dependence graph (CDG)

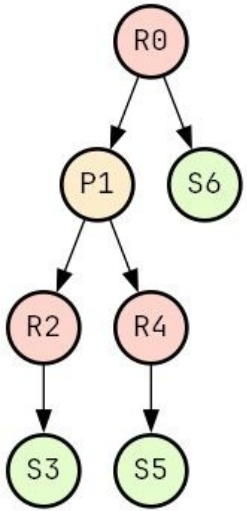
Control dependence means that execution of one node in the graph conditionally depends upon the execution of another node in the graph.

Region nodes summarize the control dependencies of a group of statements.

CDG doesn't have a sequential nature of CFG. If two nodes are not control-dependent on each other, we can try to execute them in any order.

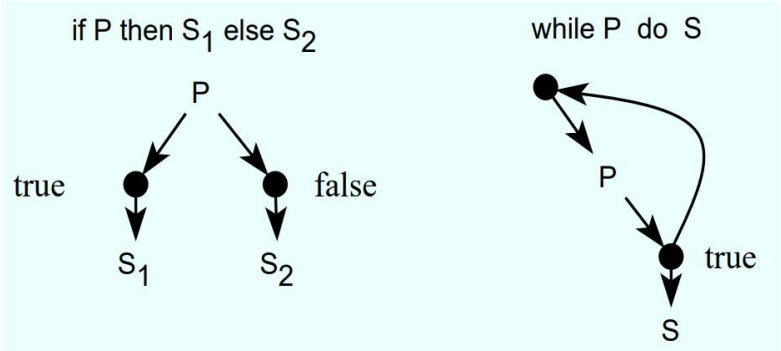
```
if x < y:      (P1)
    r = x + y (S3)
else:
    r = x - y (S5)
return r       (S6)
```





Building CDG

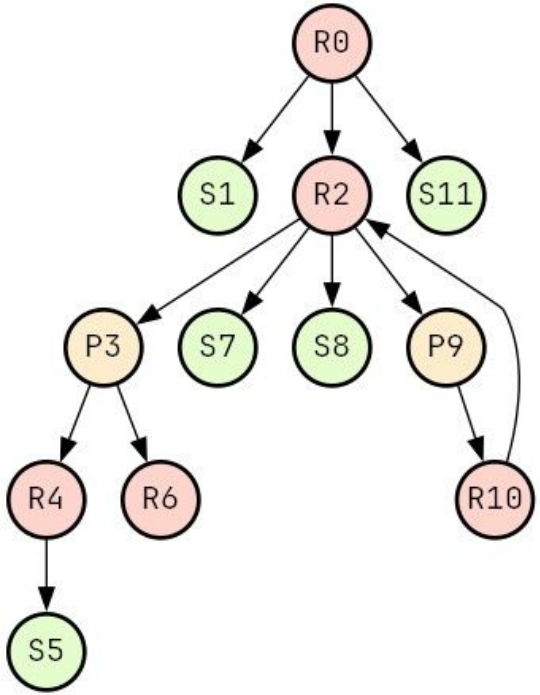
```
def parse_stmt(region, tree):
    match tree:
        case Assign() | Return():
            add('S', region)
        case If(_, true, false):
            test = add('P', region)
            parse_block(add('R', test), true)
            parse_block(add('R', test), false)
        case While(_, body):
            header = add('R', region)
            parse_block(header, body)
            add_edge(header, add('R', add('P', header)))
```



Ballance, Robert A., and Arthur B. Maccabe. *Program dependence graphs for the rest of us.* Department of Computer Science, College of Engineering, University of New Mexico, 1992.

Building CDG: an example

```
r = 0                (S1)
while a > 0:         (P9)
    if a % 2 ≠ 0:    (P3)
        r = r + b    (S5)
    a = a / 2        (S7)
    b = b * 2        (S8)
return r             (S11)
```



Program dependence graph (PDG)

PDG = **control dependence graph** + **data dependence graph**.
But data dependencies are not in SSA form!

What the authors of PDG say about classical dataflow IRs:

> Unnecessary statement orderings are eliminated in data flow machine graphs, exposing low-level parallelism. Yet, due to the distribution of control operators throughout the data dependence edges, **both data and control become too fragmented for the convenient application of conventional optimizations.**

The Program Dependence Graph and Its Use in Optimization

JEANNE FERRANTE
IBM T. J. Watson Research Center
KARL J. OTTENSTEIN
Michigan Technological University
and
JOE D. WARREN
Rice University

In this paper we present an intermediate program representation, called the *program dependence graph* (*PDG*), that makes explicit both the data and control dependences for each operation in a program. Data dependences have been used to represent only the relevant data flow relationships of a program. Control dependences are introduced to analogously represent only the essential control flow relationships of a program. Control dependences are derived from the usual control flow graph. Many traditional optimizations operate more efficiently on the PDG. Since dependences in the PDG connect computationally related parts of the program, a single walk of these dependences is sufficient to perform many optimizations. The PDG allows transformations such as vectorization, that previously required special treatment of control dependence, to be performed in a manner that is uniform for both control and data dependences. Program transformations that require interaction of the two dependence types can also be easily handled with our representation. As an example, an incremental approach to modifying data dependences resulting from branch deletion or loop unrolling is introduced. The PDG supports incremental optimization, permitting transformations to be triggered by one another and applied only to affected dependences.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Data flow, dependence analysis, intermediate program representation, internal program form, vectorization, parallelism, node splitting, code motion, loop fusion, slicing, debugging, incremental data flow analysis, branch deletion, loop unrolling

1. INTRODUCTION

This paper introduces a program representation, called the *Program Dependence Graph* or *PDG*, that provides a unifying framework in which previous work in program optimization may be applied. We present a new incremental data flow algorithm that operates directly on the PDG. This algorithm is important not

3737 citations!

Sea of nodes (SoN): Most popular graph-based IR

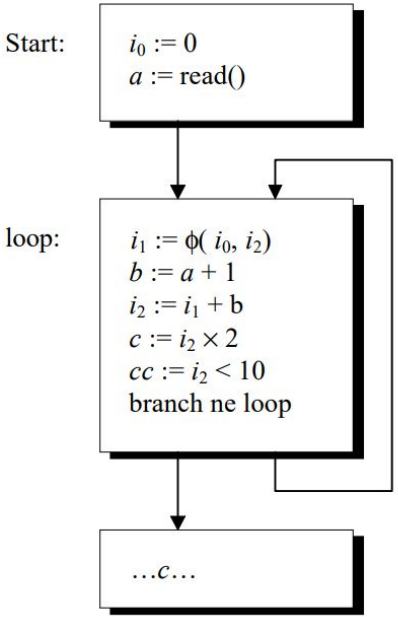
SoN = simplified **PDG** + **SSA** graph.

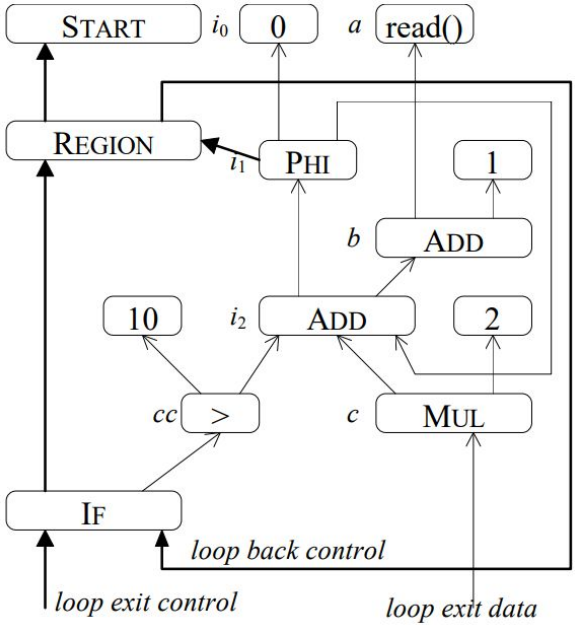
Direct correspondence to CFG:

- **START** and **REGION** nodes start a basic block.
- **IF**, **JUMP,** and **STOP** nodes end a basic block.

It has a sequential **executable model** on the CFG level:
a single control token moves from node to node by control edges.

Phi nodes are executable in SoN!



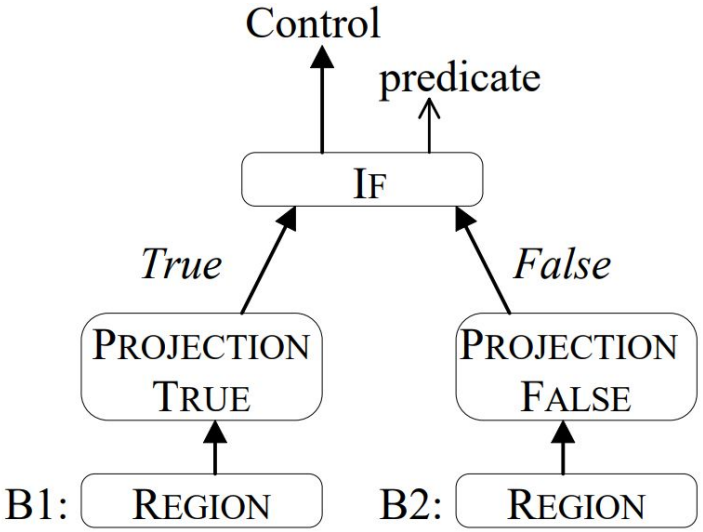


Click, Cliff, and Keith D. Cooper. "Combining analyses, combining optimizations." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.2 (1995): 181-196.

SoN: multiple output ports

In SoN, the nodes with multiple output ports return a **tuple of results**.

Then the **PROJECTION** node is used to select one of the needed results.



SoN: optimizations

Removing the schedule from the machine independent optimizations allows stronger optimizations using simpler algorithms.

Cliff Click, the SoN author

GVN in SoN can be done using a simple LVN pass **without checking the control dependencies**.
It works thanks to the CFG building pass, which also does LICM and other code motion optimizations.

The advantage of building a PDG is the discovery of more congruences. The disadvantage is that serialization is more difficult.

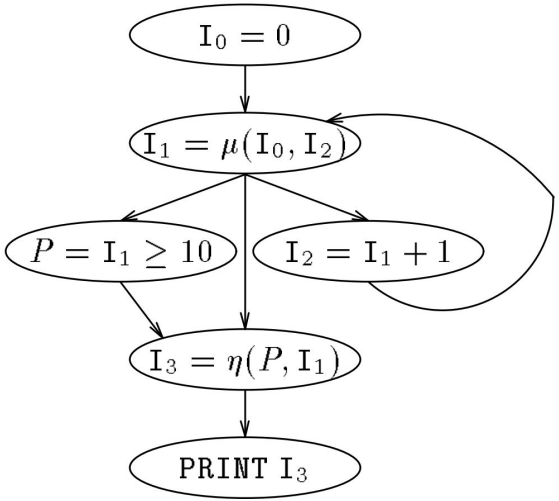
Cliff Click, the SoN author

1. Classical dataflow models
2. Program dependence graphs
3. Gated data dependence graphs
4. Perspectives

Gated SSA (GSA)

GSA is an SSA graph with added **gating phi-functions**.
GSA has **demand-driven semantics**.

- The **γ-node**(P, true, false) is the same as a merge node.
- The **μ-node**(init, iter) at the loop header produces an infinite sequence of values, starting from init.
- The **η-node**(P, final) at the loop exit returns the final values.

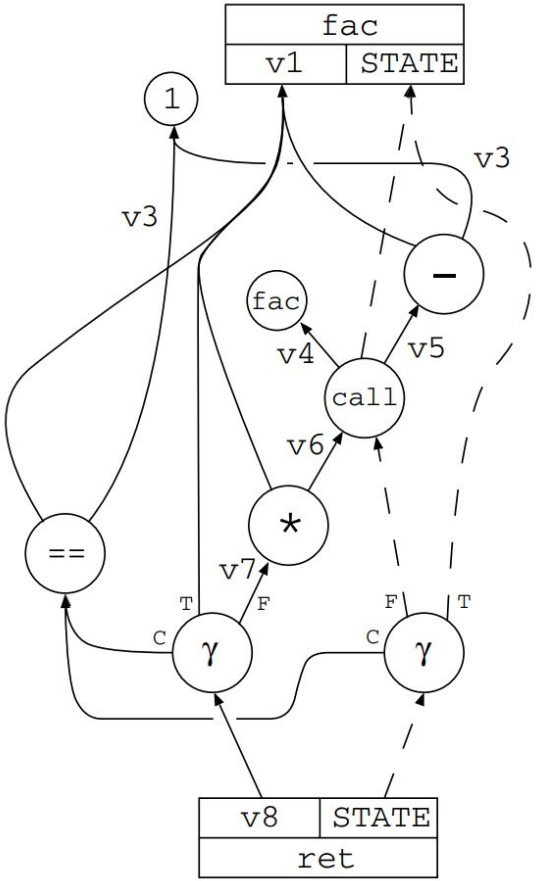


Havlak, Paul. "Construction of thinned gated single-assignment form." *International Workshop on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993.

Value State Dependence Graph (VSDG)

- VSDG is based on GSA and has two types of edges: **value dependencies and state dependencies**.
- Building CFG is done by adding enough of **serializing state edges**.
- VSDG was used for **combined register allocation and code motion**.

```
int fac( int n) {
  int result;
  if (n == 1)
    result = n;
  else
    result= n * fac(n - 1);
  return result;
}
```

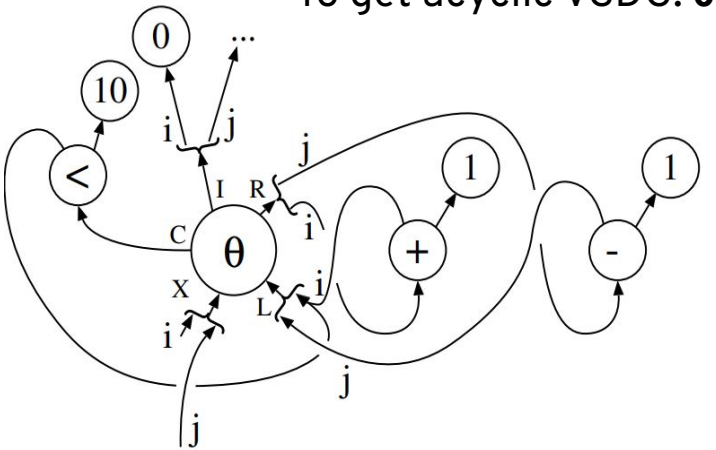


Johnson, Neil E. *Code size optimization for embedded processors.* No. UCAM-CL-TR-607. University of Cambridge, Computer Laboratory, 2004.

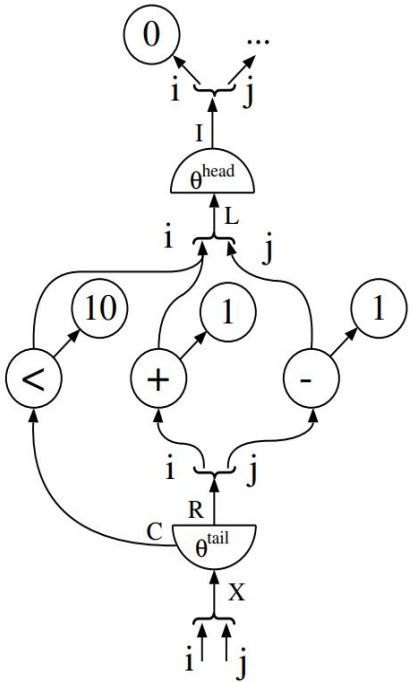
VSDG: loops

- A **θ-node(C, I, R, L, X)** sets its internal value to **initial value I**. Then, while **condition value C** holds true, sets **L** to the current **internal value** and updates the internal value with the **repeat value R**.
- When C evaluates to false loop computation stops and the internal value is **returned** through the **X port**.

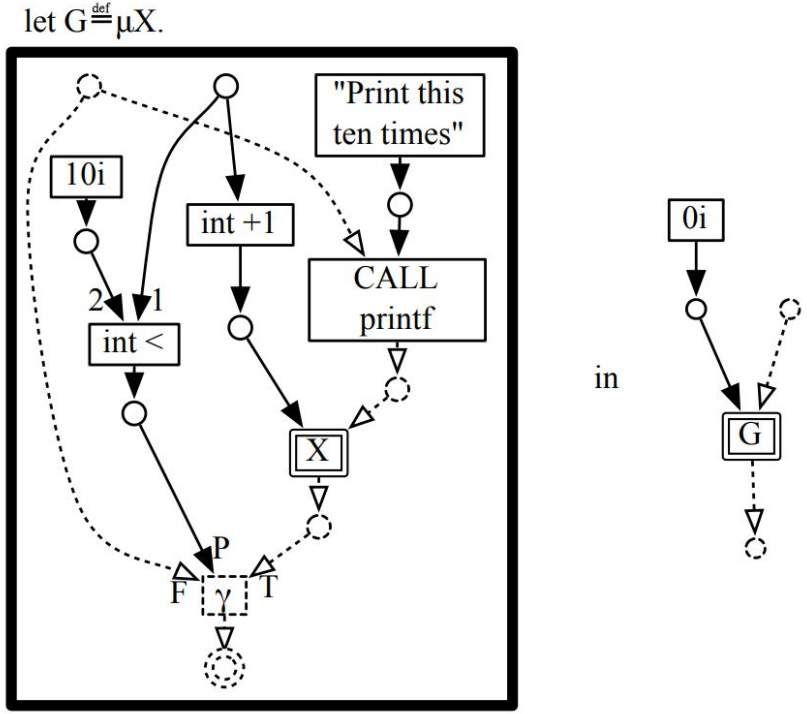
To get acyclic VSDG: **θ-head**(I, L) and **θ-tail**(R, X, C).




```
j = ...
for(i = 0; i < 10; ++i)
  --j;

... = j;
```



VSDG: yet another loop definition



Hierarchical graph and tail recursion.

Lawrence, Alan C. *Optimizing compilation with the value state dependence graph*. No. UCAM-CL-TR-705. University of Cambridge, Computer Laboratory, 2007.

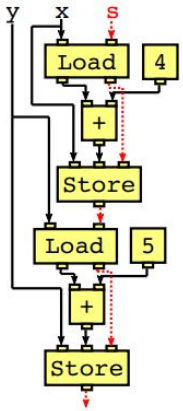
Regionalized Value State Dependence Graph (RVSDG)

RVSDG — **acyclic hierarchical graph** consisting of nested subgraphs called **regions**.

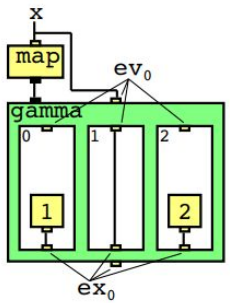
Lots of complex nodes, containing regions:

1. If-else (gamma).
2. Do-while (theta).
3. Function (lambda).
4. Global variable (delta).
5. Mutally recursive environment (phi).
6. Translation unit (omega).

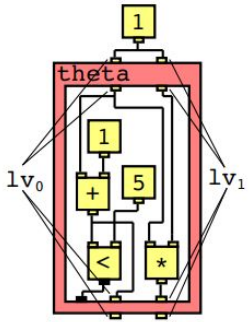
No interesting optimizations yet.
And isn't it too similar to IF1?




```
*x += 4;
*y += 5;
```




```
switch(x){
  case 0: y=1; break;
  case 1: y=x; break;
  default: y=2; break;
}
```




```
int r=1, n=1;
do {
  r=n*r;
  n++;
} while(n<5);
```

Reissmann, Nico, et al. "RVSDG: An intermediate representation for optimizing compilers." *ACM Transactions on Embedded Computing Systems (TECS)* 19.6 (2020): 1-28.

1. Classical dataflow models
2. Program dependence graphs
3. Gated data dependence graphs
4. Perspectives

Declarative graph rewriting

Graph rewriting rules can be very complex and error-prone. So it's important to have a **DSL for a declarative description of the rules**.

A fragment of imperative rule definition from TensorFlow graph optimizer:

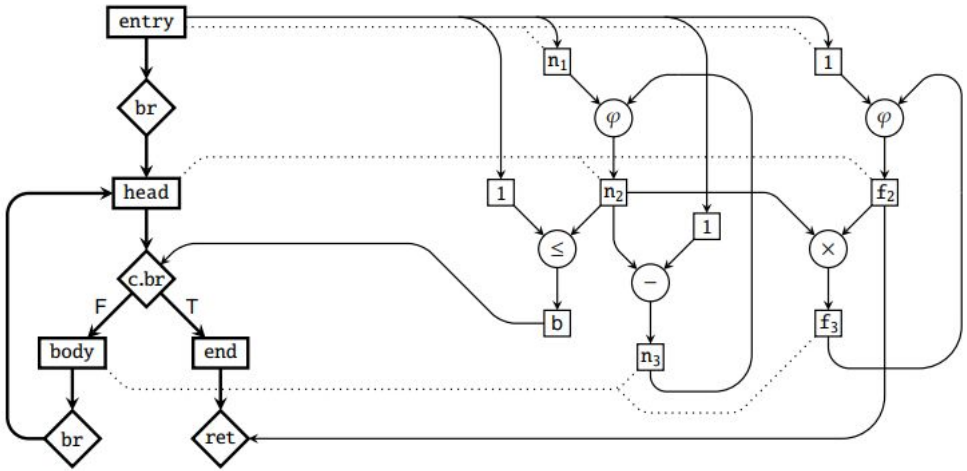
```
...
if (IsNeg(*y)) {
      // a - (-b) = a + b or  a + (-b) = a - b
      ForwardControlDependencies(node, {y});
      ctx().node_map->UpdateInput(node->name(), node->input(1), y->input(0));
      node->set_op(IsAdd(*node) ? "Sub" : "AddV2");
      node->set_input(1, y->input(0));
      updated = true;
    } else if (IsAdd(*node) && IsNeg(*x)) {
      // (-a) + b = b - a
      ForwardControlDependencies(node, {x});
      ctx().node_map->UpdateInput(node->name(), node->input(0), x->input(0));
      node->set_op("Sub");
      ...
```

https://github.com/tensorflow/tensorflow/blob/893aa7518fe3175739ac1ba70d7355a0b091115c/tensorflow/core/grappler/optimizers/arithmetic_optimizer.cc

Global graph rewriting

It's important for algorithm-level transformations and custom hardware support to **rewrite graphs at the function level**, using any type of graph, not just trees or single output DAGs, as patterns.

The authors of **Complete and Practical Universal Instruction Selection** (2017) use SoN-like IR, constraint programming and VF2 algorithm for subgraph isomorphism testing.

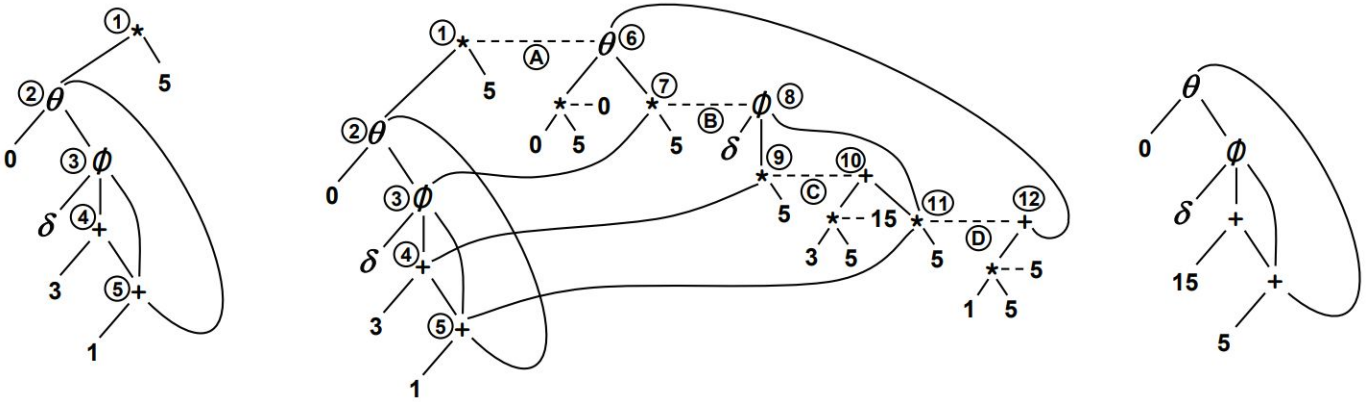


Blindell, Gabriel Hjort, et al. "Complete and practical universal instruction selection." ACM Transactions on Embedded Computing Systems (TECS) 16.5s (2017): 1-18.

Global equality saturation

Each optimization rule **only adds information** to the program graph **in the form of axioms denoting alternative rewrites** without having to prematurely decide on an order. Then the best result is found using some cost model once all the information is known.

E-PEG (program expression graph, 2009) is a graph-based IR for global equality saturation. E-PEG is based on **gated SSA** (θ-node below represent a loop header).



Tate, Ross, et al. "Equality saturation: a new approach to optimization." *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009.

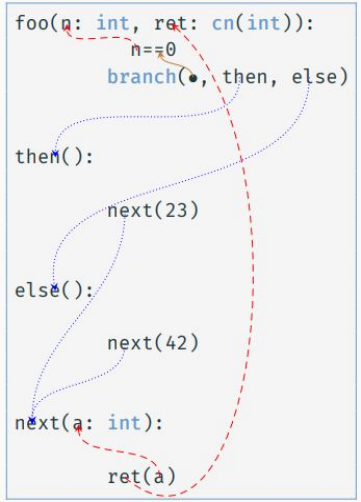
Partial evaluation (PE)

Partial evaluation allows the transformation of a program written in **general and abstract form** into its **specialized and optimized version**.
It is especially good for DSL compilers.

Graph-based IRs are good for PE because they may have an executable model of the whole program, and at any transformation step it's possible to know all needed dependencies.

- **LMS** (2010), (Lightweight Modular Staging) framework. Uses SoN-like IR.
- **Graal** IR (2013). Uses SoN IR.
- **Thorin** IR (2015), a part of AnyDSL project. Uses SoN-lke IR with continuation-passing style.

```
int foo(int n) {
    int a;
    if (n==0) {
        a = 23;
    } else {
        a = 42;
    }

    return a;
}
```



Rompf, Tiark. "Reflections on LMS: exploring front-end alternatives." Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala. 2016.
Duboscq, Gilles, et al. "Graal IR: An extensible declarative intermediate representation." Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop. 2013.
Leißa, Roland, Marcel Köster, and Sebastian Hack. "A graph-based higher-order intermediate representation." 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2015.

Other perspective ideas

- Combining **global instruction scheduling** with other code generation phases.
- Combining graph-based IR with a more detailed loop representation based on a **polyhedral model**.
- Adding **powerful type systems** to graph-based IR.
  The authors of **Graph IRs for Impure Higher-Order Languages** (2023) use reachability types (which combine ideas from ownership types and separation logic) and an effect system.

BRAČEVAC, OLIVER, et al. "Graph IRs for Impure Higher-Order Languages." (2023).

MLIR is starting to support graph-based IRs

Graph regions have recently been added to MLIR. CIRCT and TensorFlow dialects already use them.

See **Representing Concurrency with Graph Regions in MLIR** talk.

But it looks like there is **no support for hierarchical graphs** yet:

> Currently graph regions are arbitrarily **limited to a single basic block**, although there is no particular semantic reason for this limitation. This limitation has been added to make it easier to stabilize the pass infrastructure and commonly used passes for processing graph regions to properly handle feedback loops. Multi-block regions may be allowed in the future if use cases that require it arise.

https://mlir.llvm.org/docs/LangRef/#graph-regions

THANK YOU!

Email: peter.sovietov@gmail.com
Telegram: @true_grue