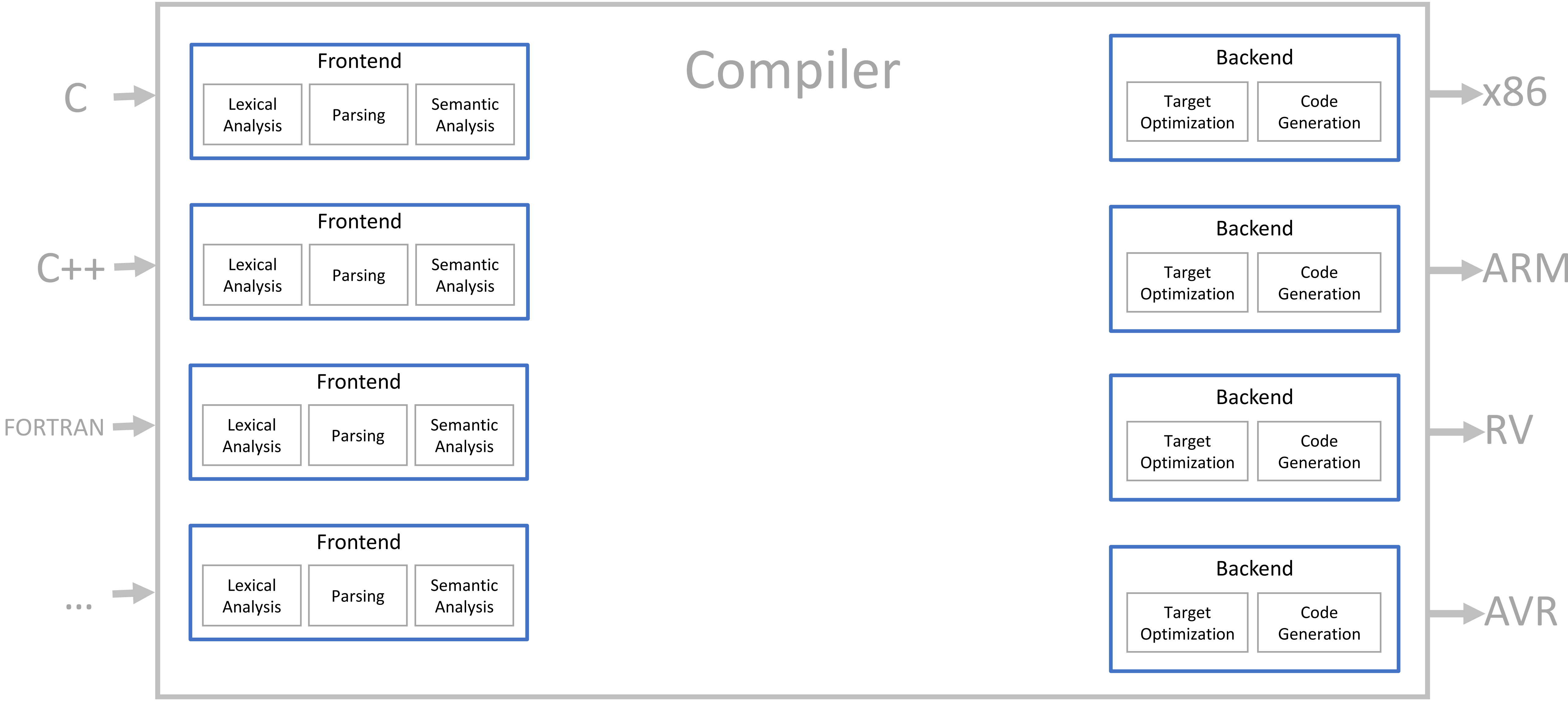
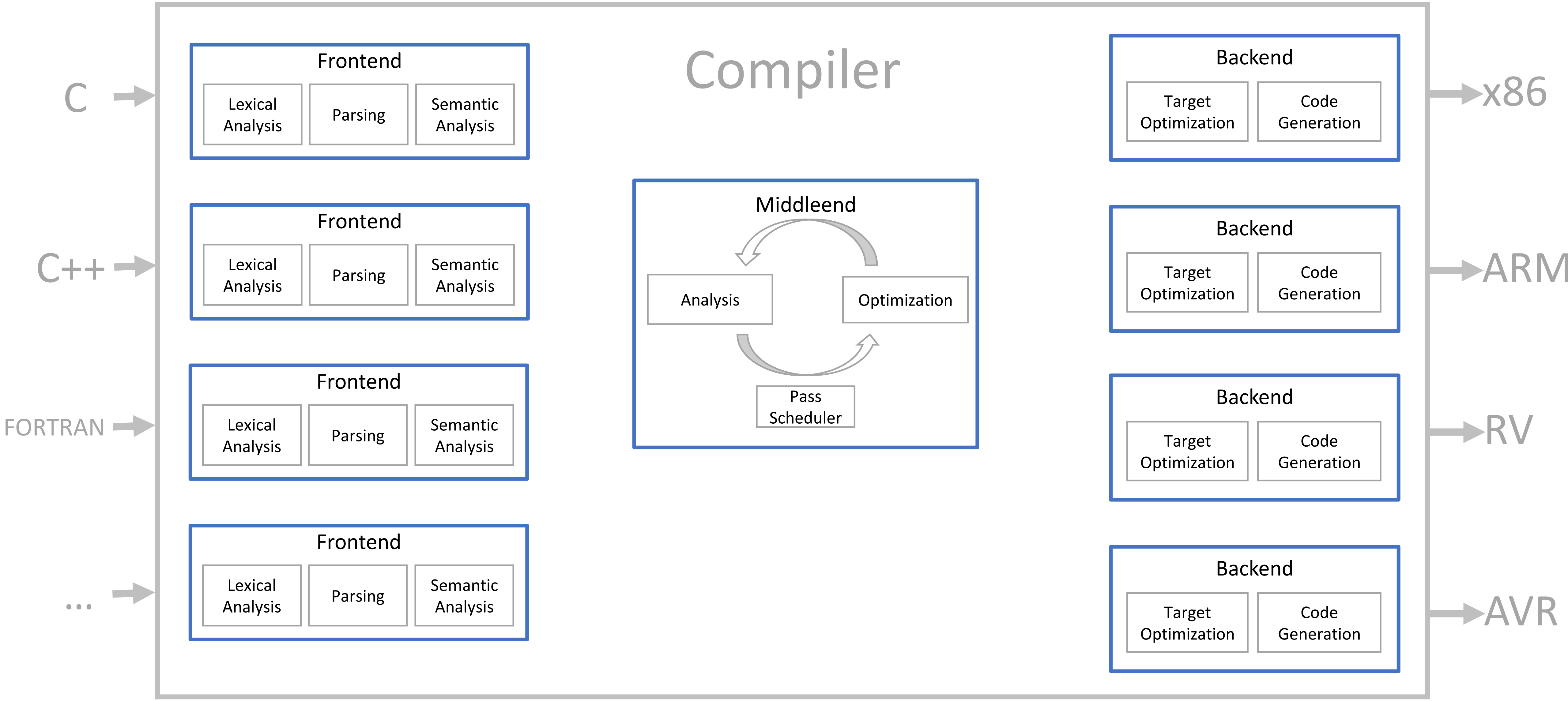
Internal Representations

NI(E)-GEN, Spring 2021

https://courses.fit.cvut.cz/NI-GEN


Compiler
C
C++
FORTRAN
...
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
x86
ARM
RV
AVR


Compiler
C
C++
FORTRAN
...
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Middleend
Analysis
Optimization
Pass Scheduler
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
Backend
Target Optimization
Code Generation
x86
ARM
RV
AVR

Intermediate Representation

- source and target independent representation
  - how to represent machine, control flow, operations, types, …
- easy to process
  - create, read, write, traverse, transfer
  - structured vs flat
- easy to optimize
  - preserve enough semantics for high level optimizations
  - allow target-independent low level optimizations
- often multiple IRs in a single compiler

AST

- close to the source language, usually expresses all semantics
- variables, source language typesystem

- far from the target language
- trivial to translate to (parser)
- simple to translate from (syntax driven code generation)

- memory inefficient, poor locality (in naïve form)
- good for local optimalizations only (intra-expression, CP, CSE, …)

Directed Acyclic Ggraphs (DAG)

- variant of expression trees that captures common subexpressions
- smaller footprint than AST
- and allows compiler to easily optimize identical values and computations within single expression

Creating DAGs

- when creating node or leaf, check if identical node exists
- existing nodes stored in a table
  - leaf – value
  - node – lhs id, rhs id
- moving from leaves to nodes, a common subexpression will be correctly matched

$$(a + 2) * (a + 2)$$

Stack Based IR

- no registers, arguments to operations placed on stack
- compact representation
- simple translation
- control flow usually translated with jumps

- often used for bytecodes (JVM, CIL, etc.)
- sometimes, even target architectures are stack based

$$(a + 2) * (a + 2)$$

```
PUSH a
PUSH 2
ADD
PUSH a
PUSH 2
ADD
MUL
```

$$(a + 2) * (a + 2)$$

```
PUSH a
PUSH 2
ADD
PUSH a
PUSH 2
ADD
MUL
```

```
PUSH a
PUSH 2
ADD
DUP
MUL
```

Stack Based IR

- harder to optimize
- even local optimizations are non-trivial and require extra stack manipulation
- code movement hard (code expects stack layout)

$$(a + 2) * (a - 2) * (a + 2)$$

```
PUSH a
PUSH 2
ADD
PUSH a
PUSH 2
SUB
MUL
PUSH a
PUSH 2
ADD
MUL
```

Register Machine

- operates on registers, but the number of registers is usually not bounded
- registers in this setting almost like variables

- usually lower level (i.e. different type system)
- control flow almost exclusively as jumps
- high-level assembler

- good for low level optimizations

Three Address Code

- register machine IRs are usually normalized, such as the 3AC
- each operation is expressed as

```
result = src1 op src2
```

- i.e. instruction (the operation) and three addresses (result, src1 and src2)
- the addresses are either names (symbols), constants, or compiler generated temporaries

3AC

- arithmetics
- copy (assignment)
- pointers, dereferencing, etc.
- function calls
- jumps (conditional and unconditional)

(a + 2) * (a - 2)*(a + 2)

```
t1 = ADD a 2
t2 = SUB a 2
t3 = MUL t1 t2
t3 = MUL t3 t1
```

3AC Representation

- quadruples

```
(a + 2) * (a - 2)*(a + 2)
```

```
t1 = ADD a 2
t2 = SUB a 2
t3 = MUL t1 t2
t3 = MUL t3 t1
```

Operation	Arg 1	Arg 2	Target
ADD	a	2	t1
SUB	a	2	t2
MUL	t1	t2	t3
MUL	t3	t1	t3

3AC Representation

- quadruples

- triples
  - the index of the instruction itself is a temporary that others may reference

$$(a + 2) * (a - 2)*(a + 2)$$

```
t1 = ADD a 2
t2 = SUB a 2
t3 = MUL t1 t2
t4 = MUL t3 t1
```

Operation	Arg 1	Arg 2
ADD	a	2
SUB	a	2
MUL	(0)	(1)
MUL	(2)	(0)

3AC Representation

- quadruples

- triples
  - the index of the instruction itself is a temporary that others may reference

- indirect triples
  - like tripes, but extra table that determines the instruction order so that they can be easily moved

SSA

- property of representation, such as 3AC, where each address is assigned only once and before any use

- really good for many optimizations
  - faster analysis algorithms
  - simpler code movement

$$(a + 2) * (a - 2) * (a + 2)$$

```
t1 = ADD a 2
t2 = SUB a 2
t3 = MUL t1 t2
t4 = MUL t3 t1
```

SSA

- property of representation, such as 3AC, where each address is assigned only once and before any use

- really good for many optimizations
  - faster analysis algorithms
  - simpler code movement
  - encodes reaching definitions automatically

if (a) b = 6; else b = 7;

```
    CMP a
    IF_TRUE l1 l2
l1: b = 6
    JMP l3
l2: b = 7
    JMP l3
l3:
```

$$\mathtt{if\ (a)\ b_1 = 6;\ else\ b_2 = 7;}$$

```
     CMP a
     IF_TRUE l1 l2
l1:  b₁ = 6
     JMP l3
l2:  b₂ = 7
     JMP l3
l3:
```

```
if (a) b = 6; else b = 7; b = b + 1;
```

```
    CMP a
    IF_TRUE l1 l2
l1: b = 6
    JMP l3
l2: b = 7
    JMP l3
l3: b = b + 1
```

$$\text{if (a) } b_1 = 6\text{; else } b_2 = 7\text{; } b_3 = b_? + 1\text{;}$$

```
     CMP a
     IF_TRUE l1 l2
l1:  b₁ = 6
     JMP l3
l2:  b₂ = 7
     JMP l3
l3:  b₃ = b? + 1
```

$$\texttt{if (a) } b_1 = 6\texttt{; else } b_2 = 7\texttt{; } b_4 = b_? + 1\texttt{;}$$

```
     CMP a
     IF_TRUE l1 l2
l1:  b₁ = 6
     JMP l3
l2:  b₂ = 7
     JMP l3
l3:  b₃ = Φ(b₁,b₂)
     b₄ = b₃ + 1
```

Φ (Phi) Nodes

- chooses the correct previous value based on where control flow is came from
- pseudo-instruction with no runtime counterpart
  - when executed, all Bs share are one variable and the the control flow taken modifies the value alone

Converting to SSA

- in general, it is non-trivial to determine where and for which variables to put Φ nodes (consider loops, jumps, etc.)

- efficient solution for minimal SSA exists

Converting to SSA - Dominance

- instruction A strictly dominates different instruction B if
  - all control flow to B must first pass through A

- instruction A dominates B if
  - A strictly dominates B
  - or A == B

Converting to SSA – Dominance Frontier

- instruction B is in the dominance frontier of instruction A if:
  - A does not strictly dominate B, but dominates some predecessor of B

- in English: if A assigns and strictly dominates B, then B will see A’s value, the dominance frontiers are earliest nodes from A to B where other control flow paths are joining

Converting to SSA

- in general, it is non-trivial to determine where and for which variables to put Φ nodes (consider loops, jumps, etc.)

- efficient solution for minimal SSA exists
- place Φ nodes for live variables at dominance frontiers

Converting from SSA

Converting from SSA

- converting optimized SSA to non-SSA representation also not trivial
- optimizations can prevent conversion just by merging values

```
x = 3; y = 3; x = 4; y = y + 2;
```

$x_0 = 3$

$y_0 = 3$

$x_1 = 4$

$y_1 = y_0 + 2$

```
x = 3; y = 3; x = 4; y = y + 2;
```

$x_0 = 3$	$x_0 = 3$
$y_0 = 3$	$x_1 = 4$
$x_1 = 4$	$y_1 = x_0 + 2$
$y_1 = y_0 + 2$	

$$x = 3;\ y = 3;\ x = 4;\ y = y + 2;$$

$x_0 = 3$
$y_0 = 3$
$x_1 = 4$
$y_1 = y_0 + 2$

$x_0 = 3$
$x_1 = 4$
$y_1 = x_0 + 2$

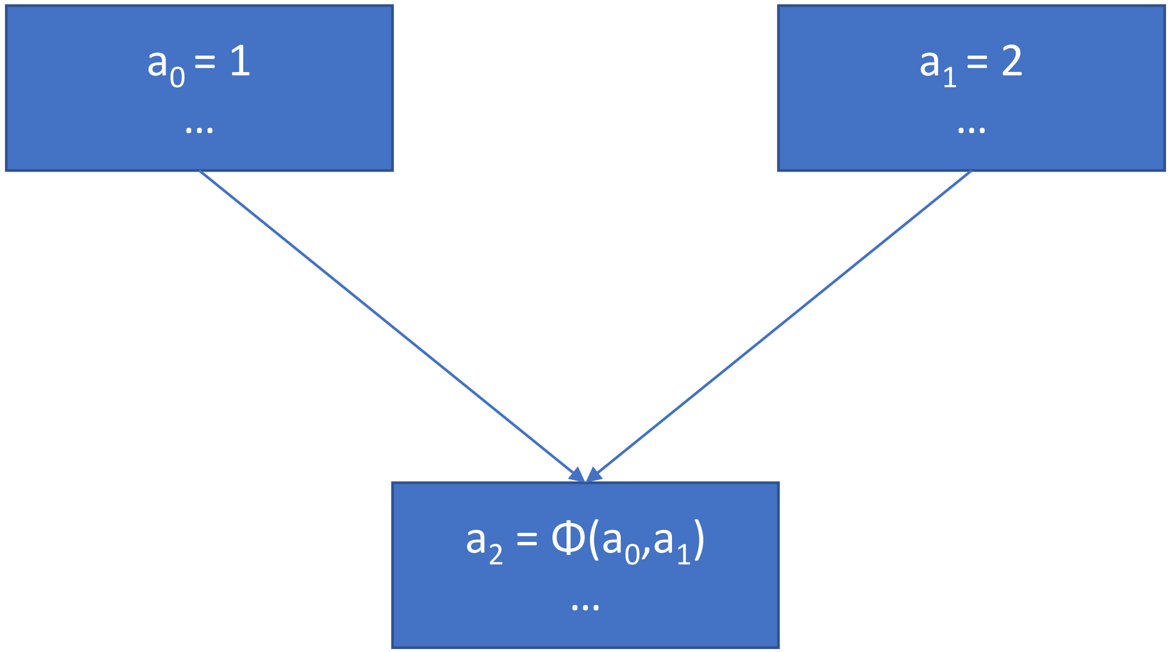
x = 3
x = 4
y = x + 2 // 4 + 2!

Converting from SSA

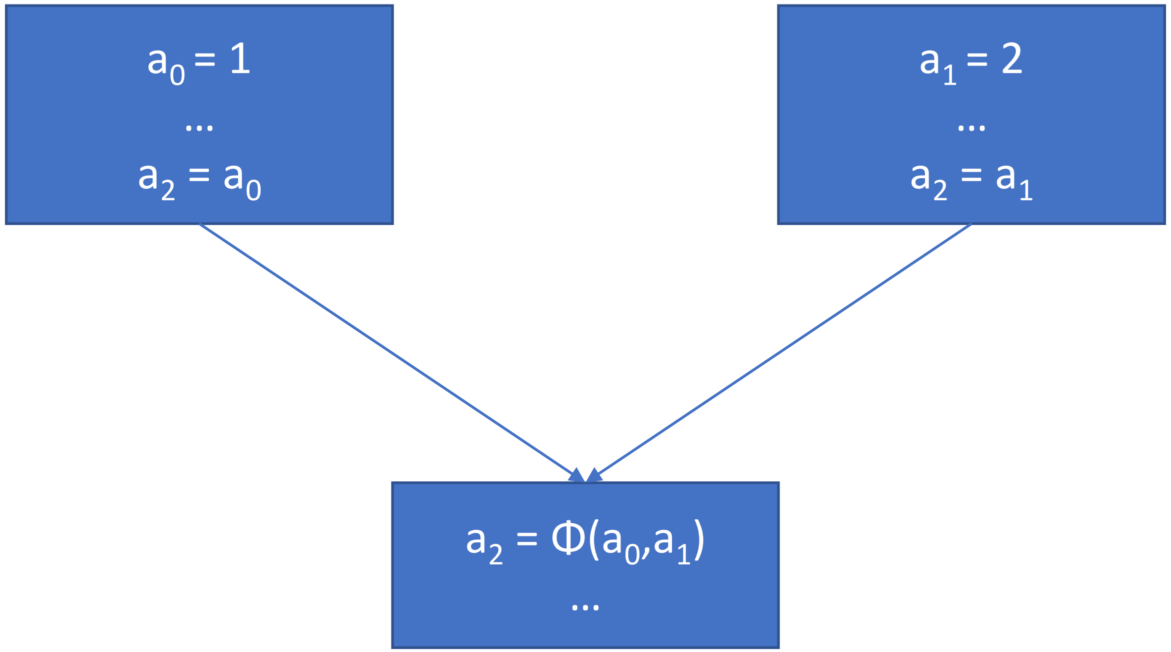
- converting optimized SSA to non-SSA representation also not trivial

- optimizations can prevent conversion just by merging values
  - use copies in predecessors to phi nodes and keep variables

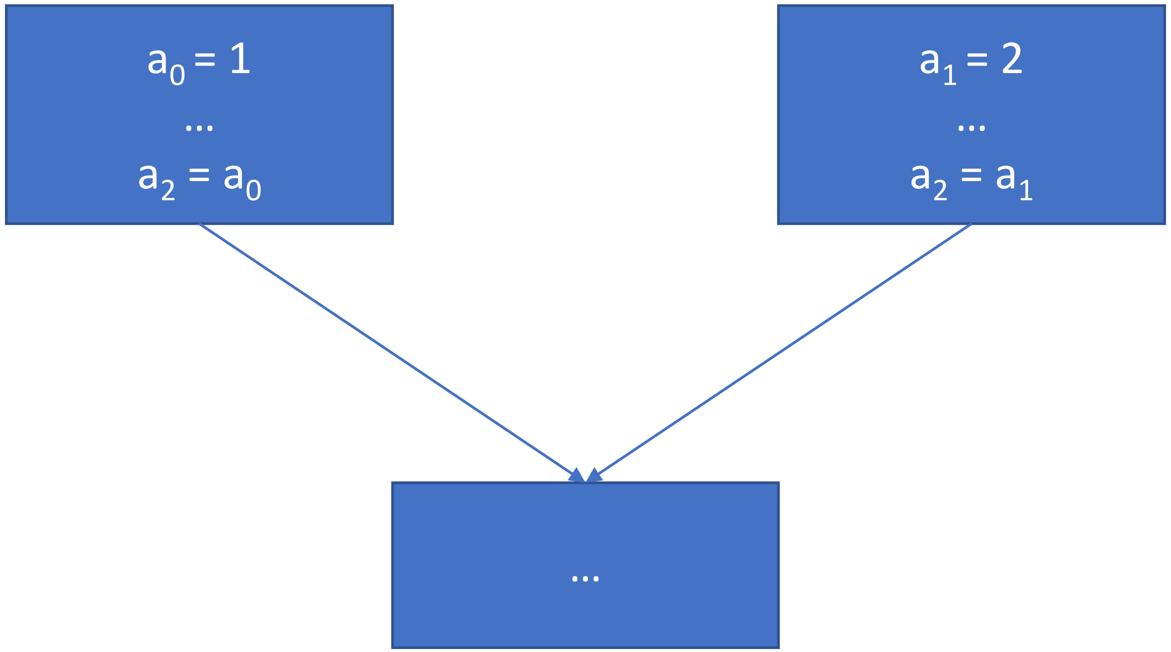
Predecessor Copies



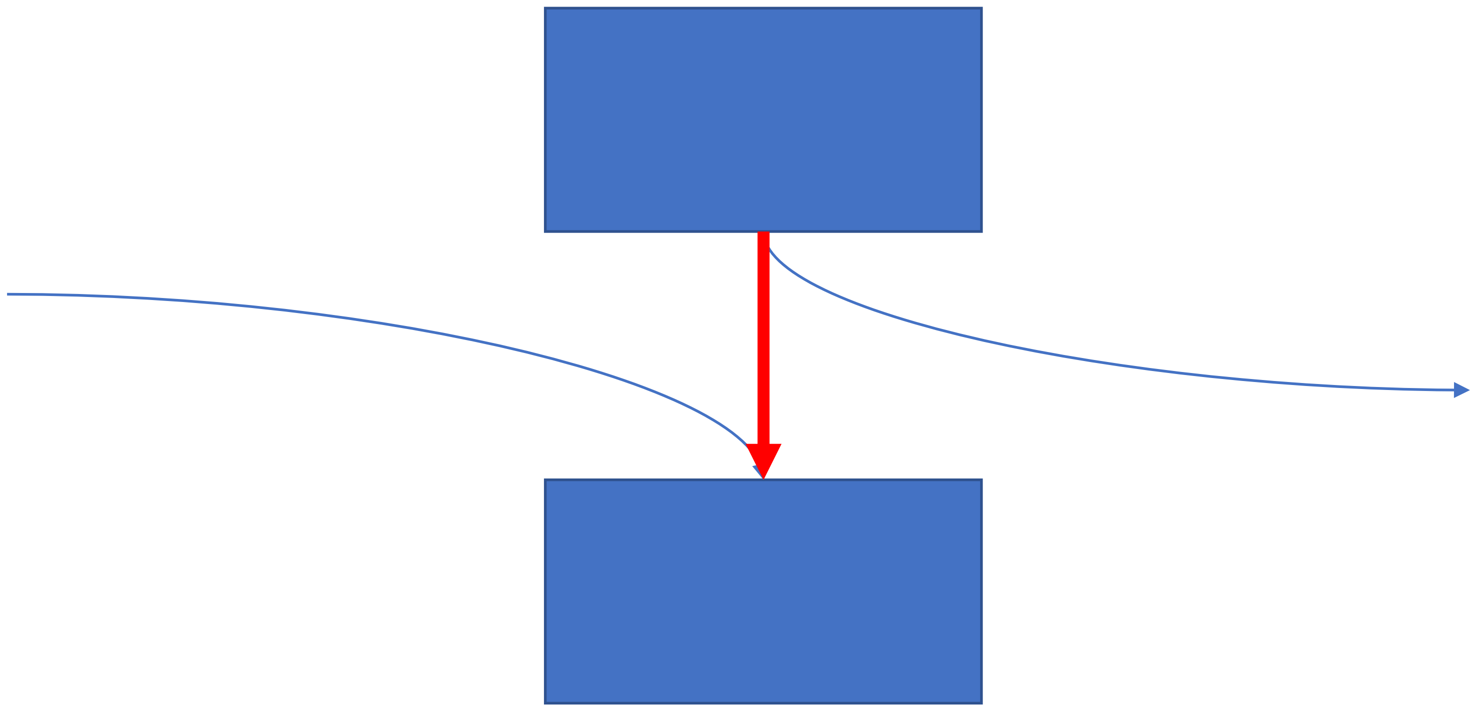
Predecessor Copies



Predecessor Copies

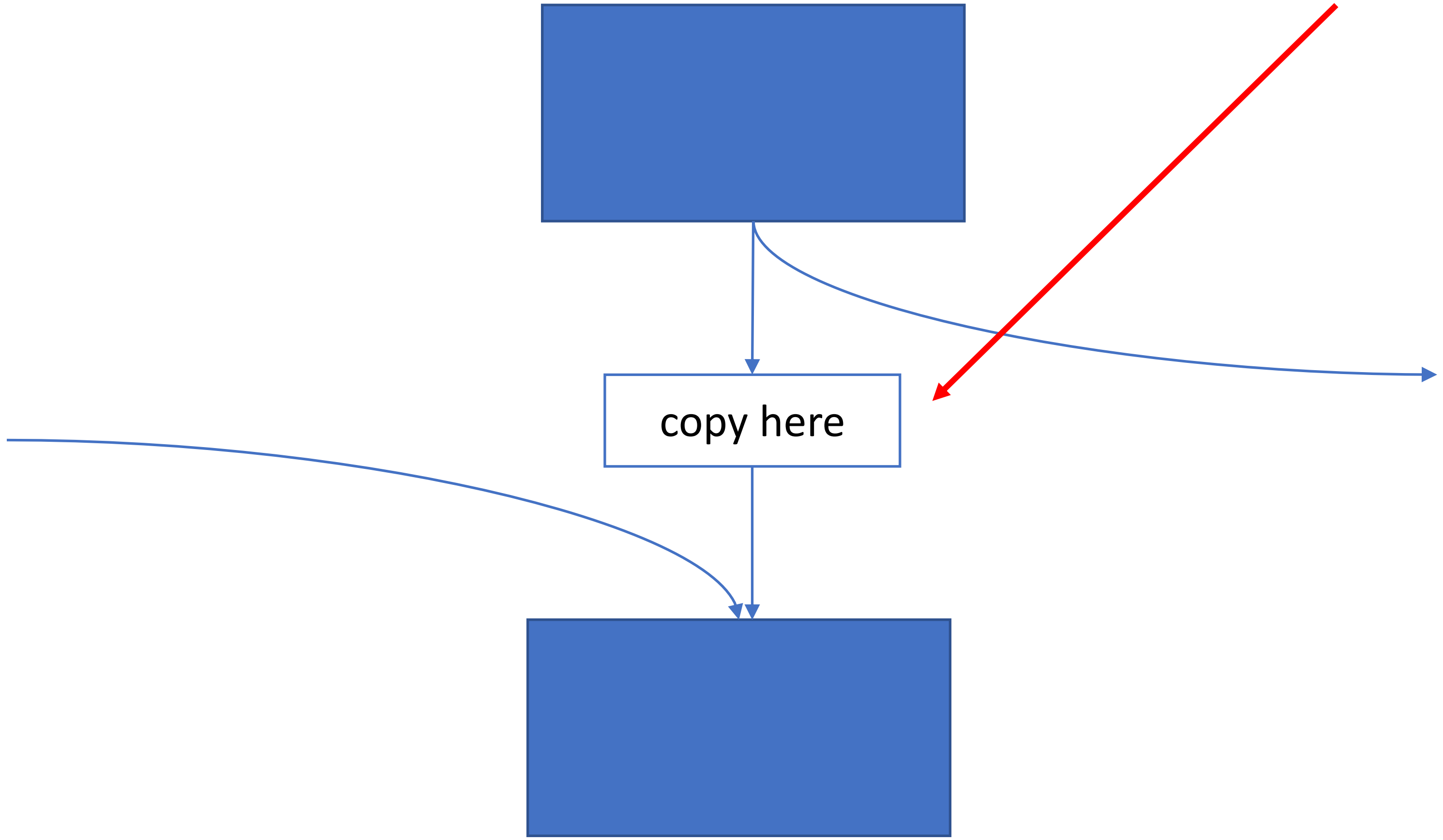


Splitting Critical Edges



Splitting Critical Edges

costly extra jump in tight loops



Swap Problem

- multiple phi nodes “execute” all in parallel
- placing copies means sequential execution, which may cause problems

SSA

- powerful representation with huge benefits for optimizing compilers
- hard to work with manually
- complex to transform to/from
- state of the art in ahead-of-time compilers (such as LLVM), little use in JITs

Sea Of Nodes

- designed by Cliff Click, first used in Java HotSpot compiler
- memory efficient representation combining IR, CFG and DDG in a single graph
- facilitates late instruction scheduling and code movement
- used in HotSpot/Graal

more about Sea of Nodes later in JITs

Non-Imperative Languages

- Continuation Passing Style
  - each function gets extra argument which is f(x) that provides the code to be executed with the result of the function, a continuation
  - similar properties to SSA, including being hard to work with

- ANF (A-Normal Form)
  - all arguments to function calls must be trivial (i.e. constants, or let-bound variables)
  - simplifies many operations, not as complex as CPS

see NI-MPJ

Further Reading

https://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf - Swift SIL design considerations

https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/ - WebFIT FTL LLVM Backend

https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/ - and how it failed