Optimizing Compilers

Prof. Andrea Marongiu
(andrea.marongiu@unimore.it)

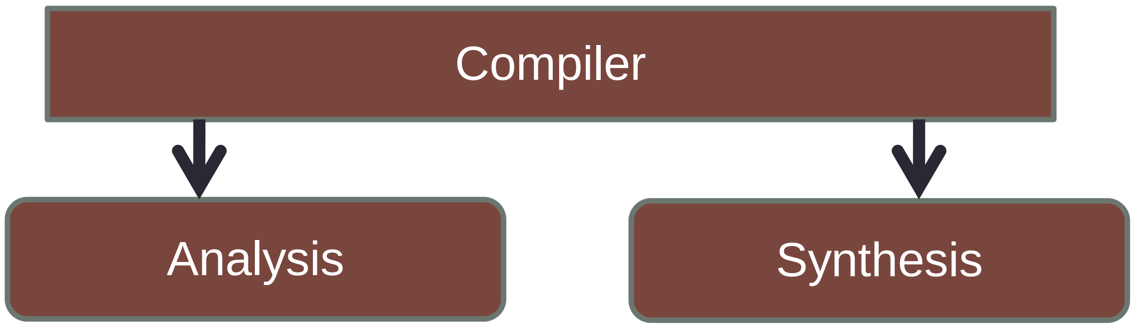
Also includes slides and contents from:
"Compiler Construction, course at University of Bern by Prof. O. Nierstrasz
"Compiler", course at University of Science and Technology of China (USTC) by Prof. Baojian Hua

Outline

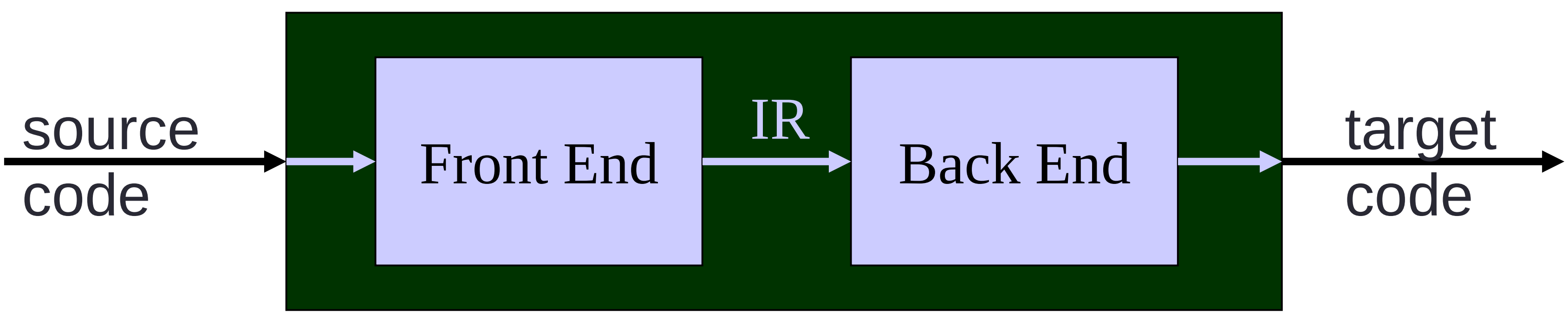
- Compiler structure
- Intermediate Representations
- Optimization

The Structure of a Compiler (1)

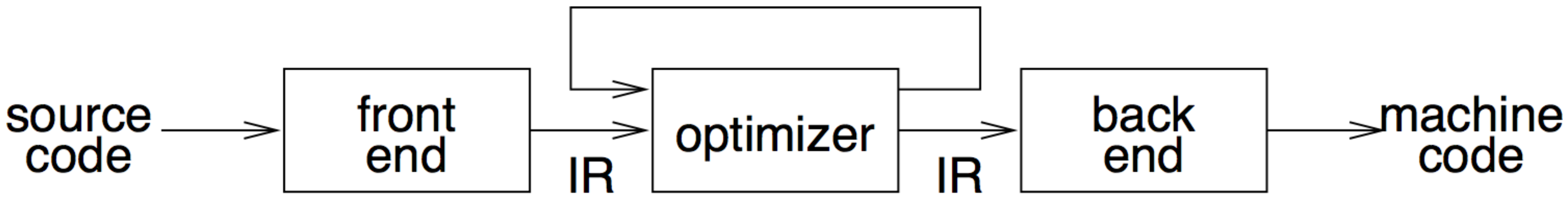
- Any compiler must perform two major tasks



  - ***Analysis*** of the source program
  - ***Synthesis*** of a machine-language program



IR scheme

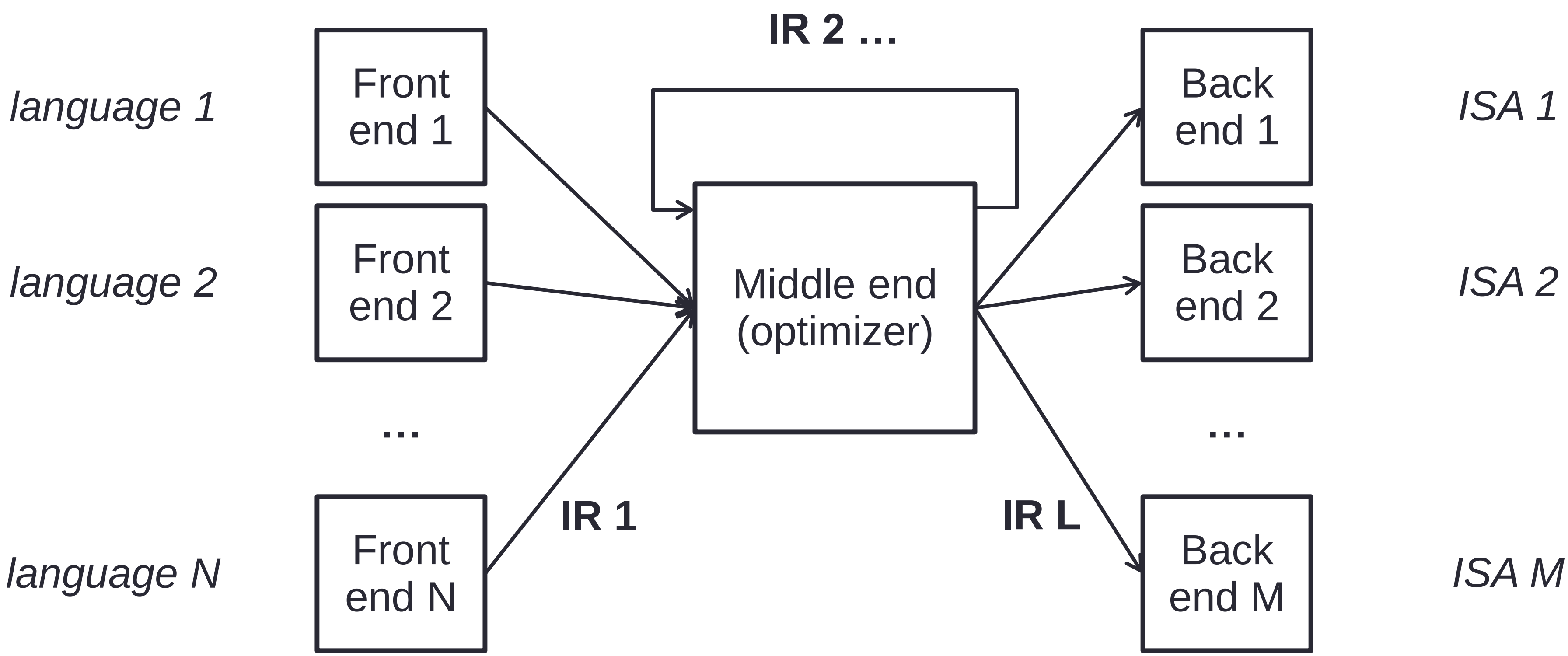


- front end produces IR
- optimizer transforms IR to more efficient program
- back end transform IR to target code

Why use intermediate representations?

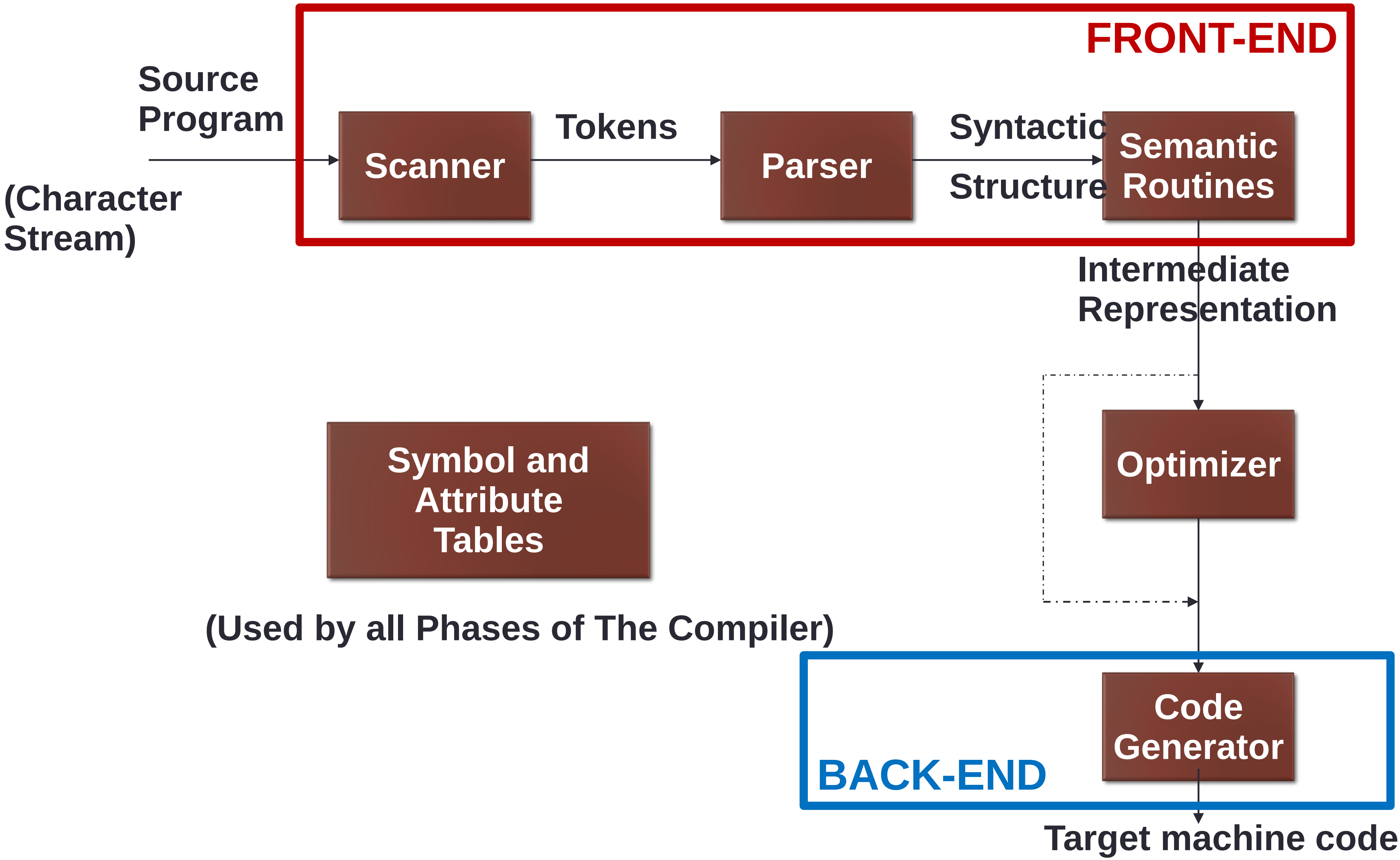
1. Software engineering principle
   - break compiler into manageable pieces
2. Simplifies retargeting to new host
   - isolates back end from front end
3. Simplifies support for multiple languages
   - different languages can share IR and back end
4. Enables machine-independent optimization
   - general techniques, multiple passes

IR scheme

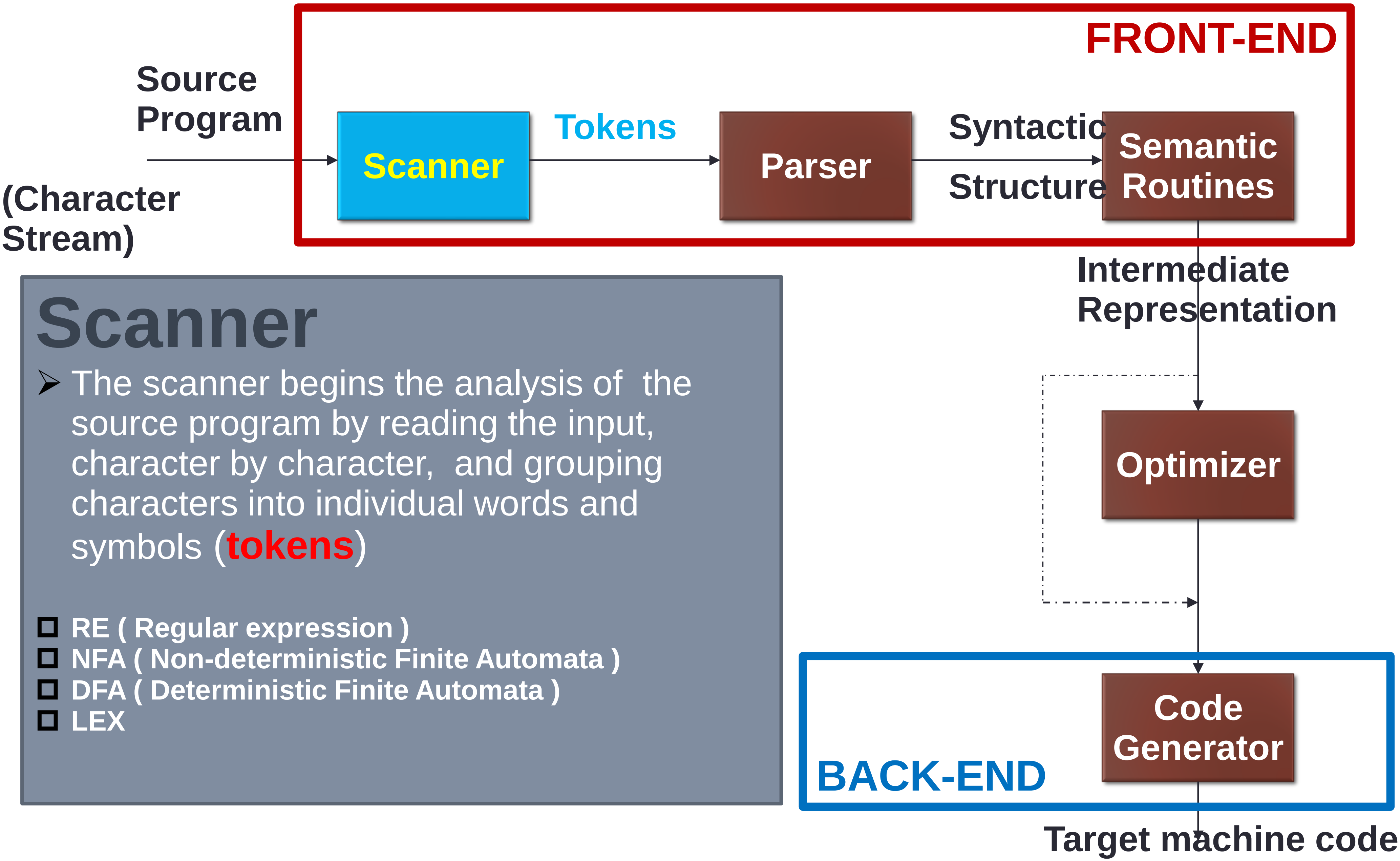


- Multiple front-ends to support different languages
- Multiple back-ends to support different target ISAs
- A common middle-end with same optimization passes
  - *Order might change*

The Structure of a Compiler (2)



The Structure of a Compiler (3)



Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (**tokens**)

- **RE (Regular expression)**
- **NFA (Non-deterministic Finite Automata)**
- **DFA (Deterministic Finite Automata)**
- **LEX**

Scanner

- Lexical Analysis
  - Recognize tokens and ignore white spaces, comments

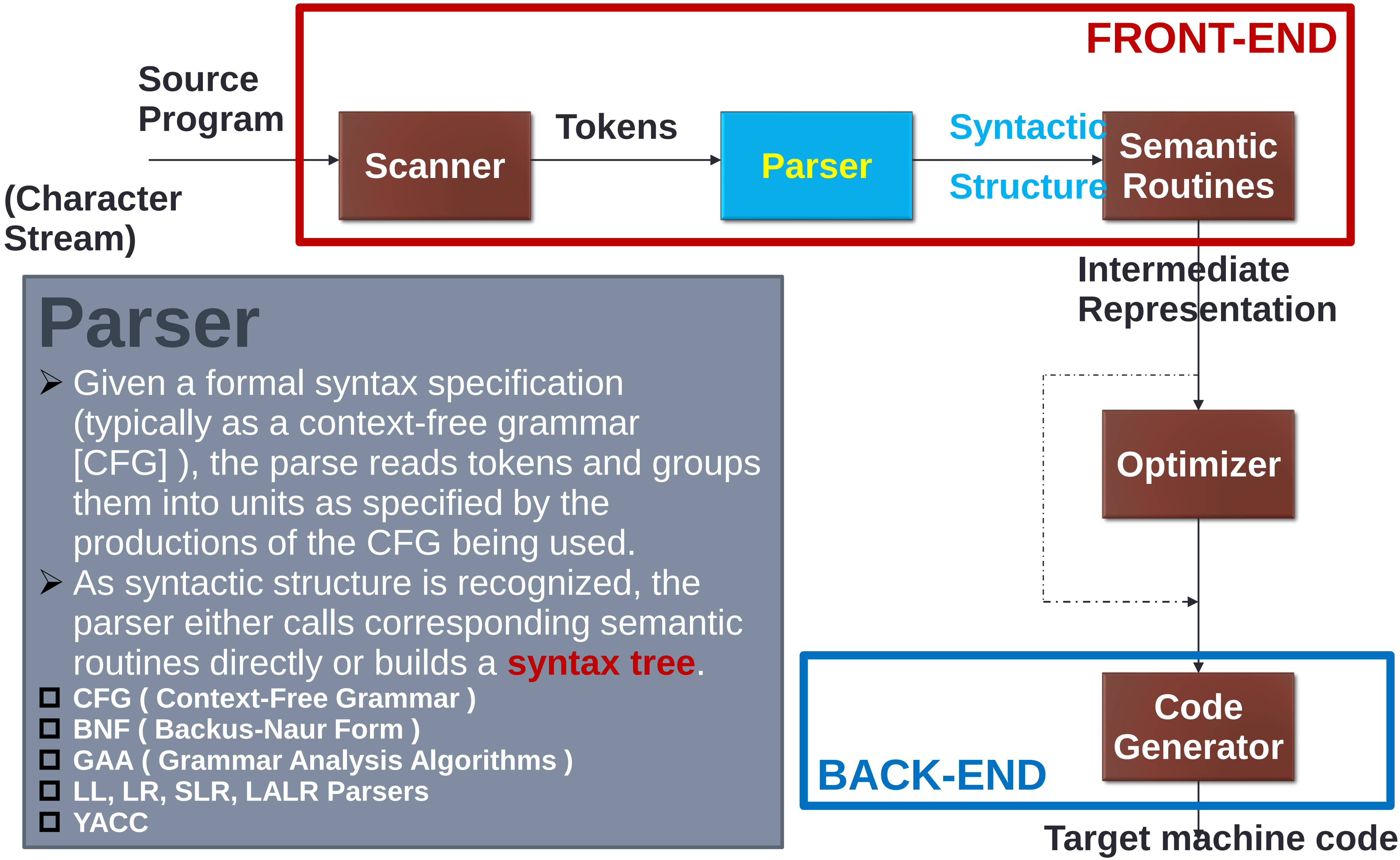
i	f		(	x	1		*	x	2	<	1	.	0	)	{

Generates token stream

if	(	x1	*	x2	<	1.0	)	{

  - Error reporting
  - Model using regular expressions
  - Recognize using Finite State Automata

The Structure of a Compiler (4)



Parser

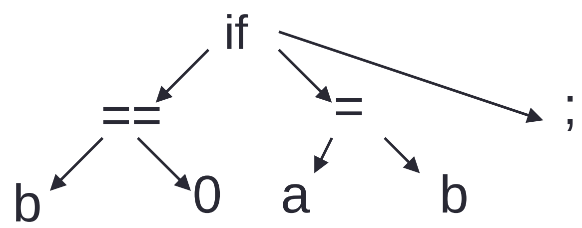
- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a **syntax tree**.

- **CFG (Context-Free Grammar)**
- **BNF (Backus-Naur Form)**
- **GAA (Grammar Analysis Algorithms)**
- **LL, LR, SLR, LALR Parsers**
- **YACC**

Parser

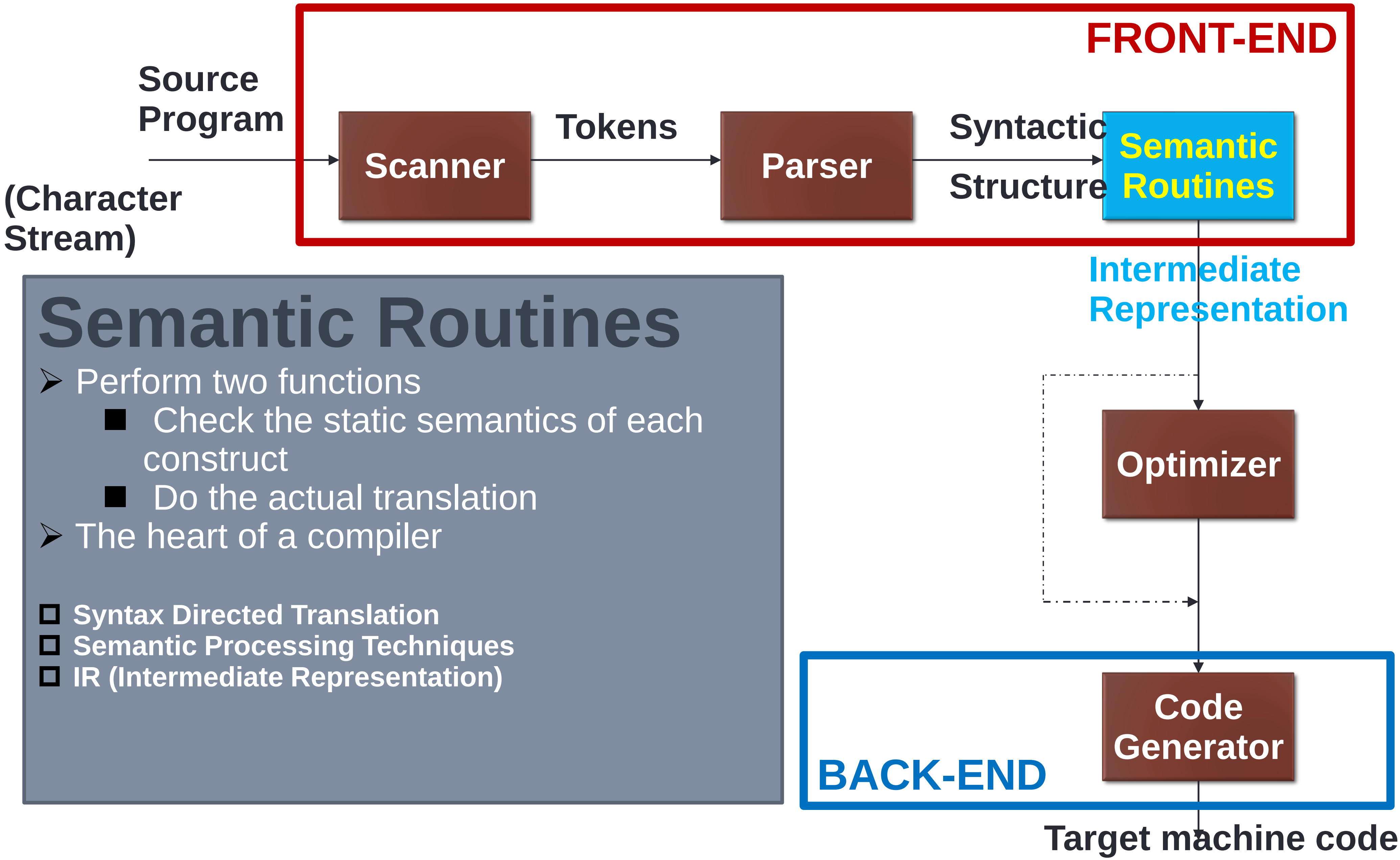
- Check syntax and construct abstract syntax tree

if	(	b	==	0	)	a	=	b	;



- Error reporting and recovery
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

The Structure of a Compiler (5)



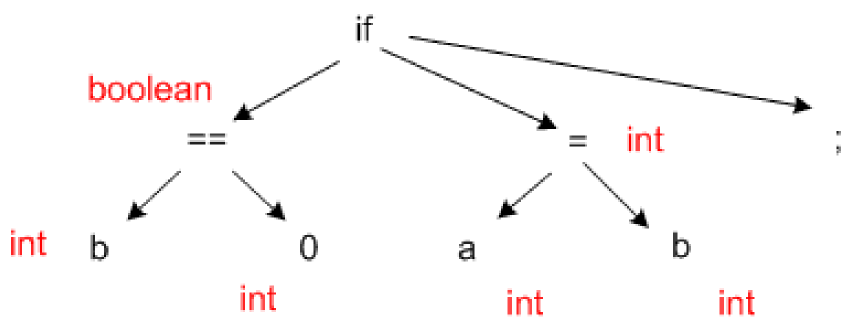
Semantic Routines

- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
- The heart of a compiler

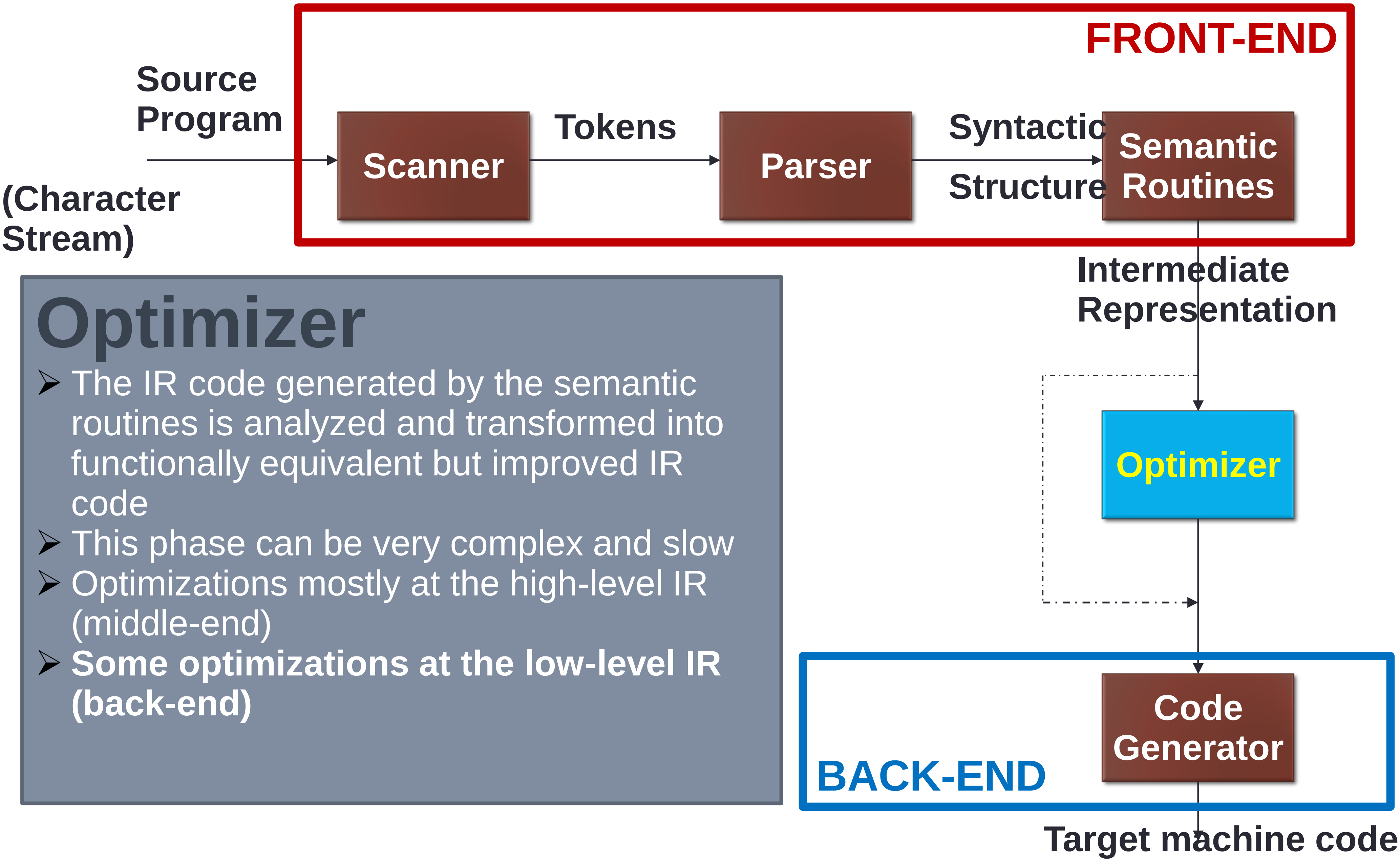
- **Syntax Directed Translation**
- **Semantic Processing Techniques**
- **IR (Intermediate Representation)**

Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate overloaded operators
- Type coercion
- Static checking
  - Type checking
  - Control flow checking
  - Unique ness checking
  - Name checks



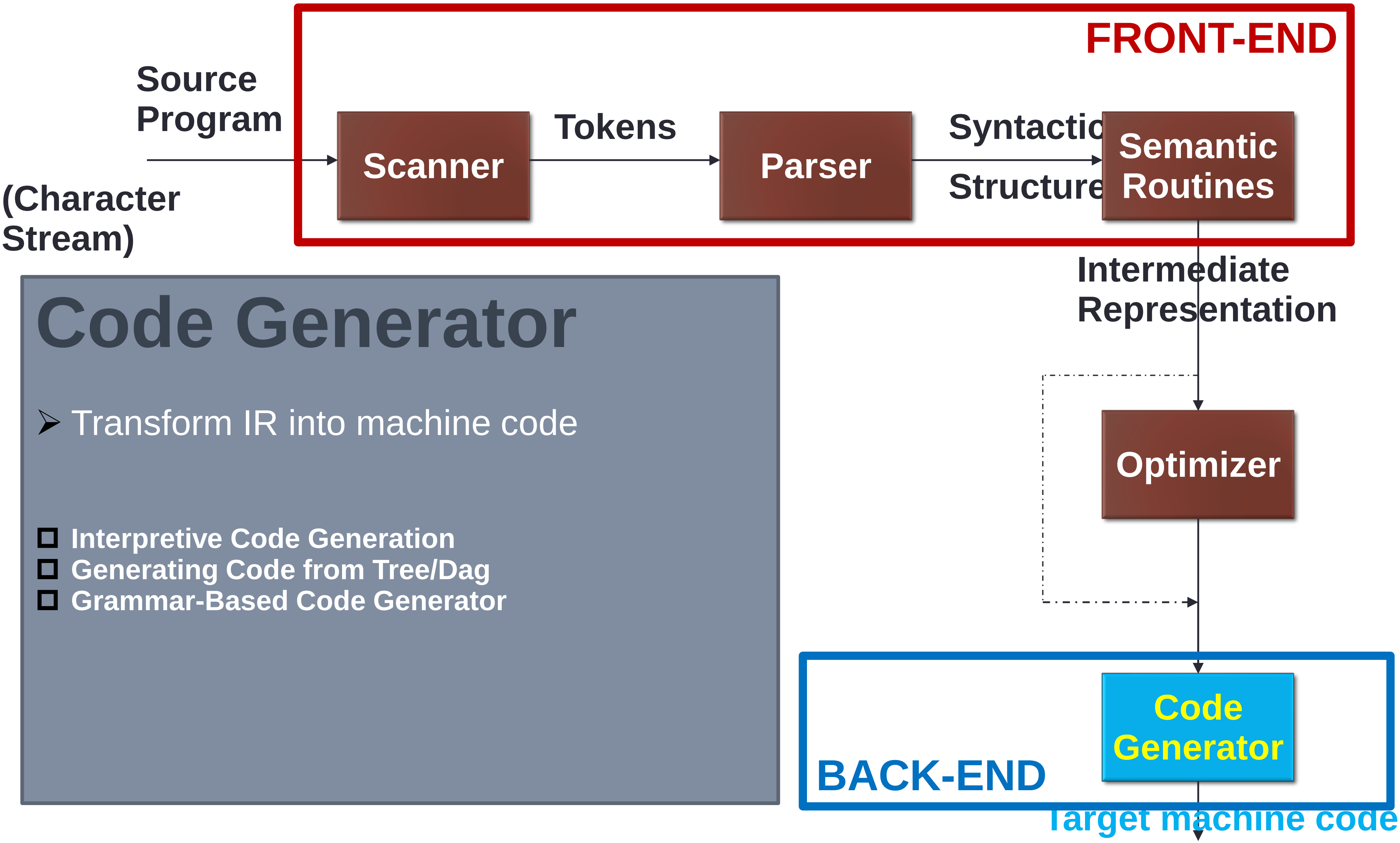
The Structure of a Compiler (6)



Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Optimizations mostly at the high-level IR (middle-end)
- **Some optimizations at the low-level IR (back-end)**

The Structure of a Compiler (7)

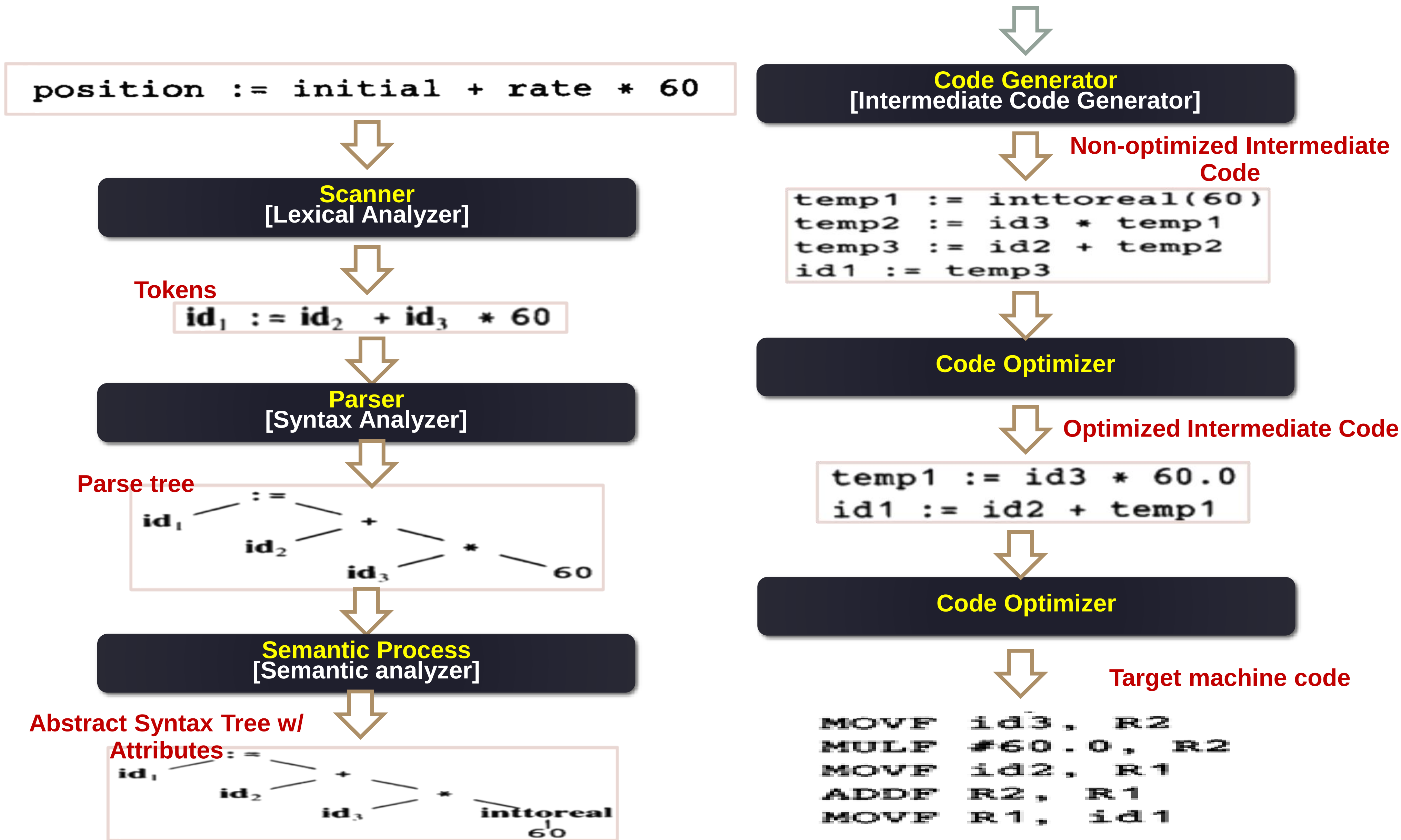


Code Generator

- Transform IR into machine code

- Interpretive Code Generation
- Generating Code from Tree/Dag
- Grammar-Based Code Generator

The Structure of a Compiler (8)



Outline

- Compiler structure
- Intermediate Representations
- Optimization

Kinds of IR

- Abstract syntax trees (AST)
- Linear operator form of tree (e.g., postfix notation)
- Directed acyclic graphs (DAG)
- Control flow graphs (CFG)
- Program dependence graphs (PDG)
- Static single assignment form (SSA)
- 3-address code
- Hybrid combinations

Categories of IR

- Structural
  - → Graphically oriented
  - → Heavily used in source-to-source translators
  - → Tend to be large

Examples:
Trees, DAGs

- Linear
  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange

Examples:
3 address code
Stack machine code

- Hybrid
  - → Combination of graphs and linear code
  - → Example: control-flow graph

Example:
Control-flow graph

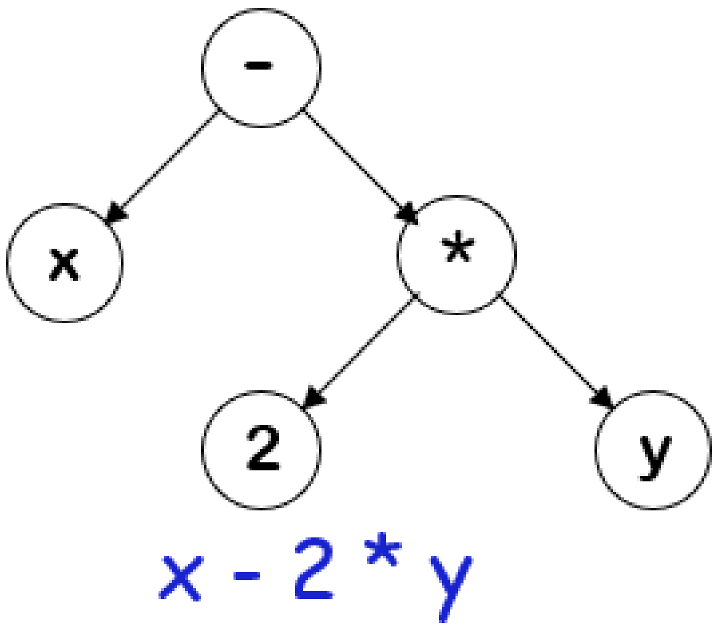
Important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Level of abstraction
- Freedom of expression (!)
- Size of typical procedure
- Original or derivative

Subtle design decisions in the IR can have far-reaching effects on the speed and effectiveness of the compiler!
➔ *Degree of exposed detail can be crucial*

Abstract syntax tree

An AST is a parse tree with nodes for most non-terminals removed.



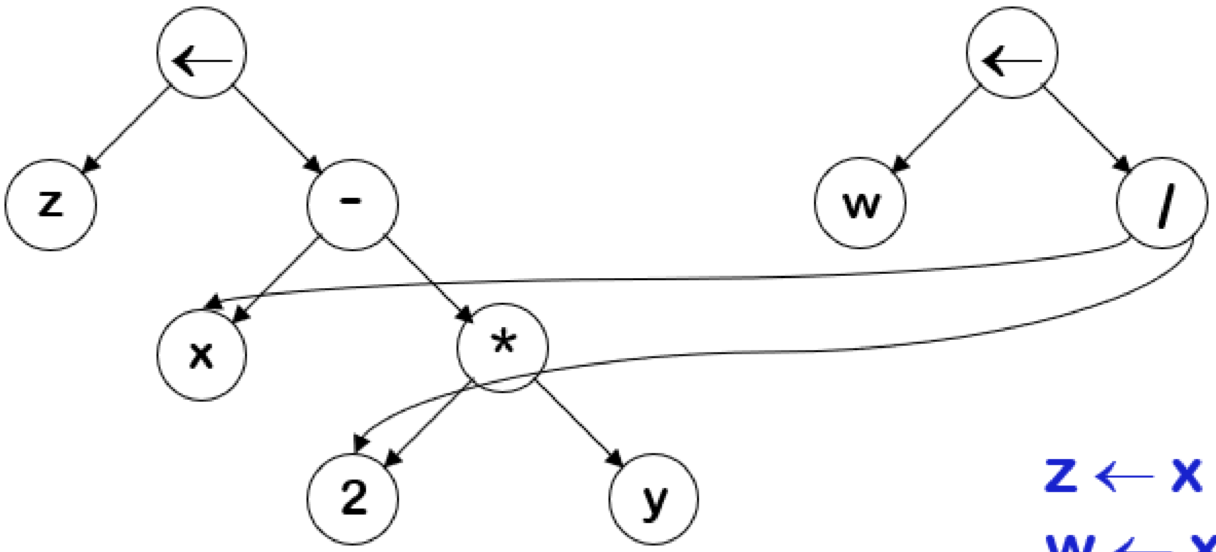
Since the program is already parsed, non-terminals needed to establish precedence and associativity can be collapsed!

A linear operator form of this tree (postfix) would be:

```
x 2 y * -
```

Directed acyclic graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



$z \leftarrow x - 2 * y$
$w \leftarrow x / 2$

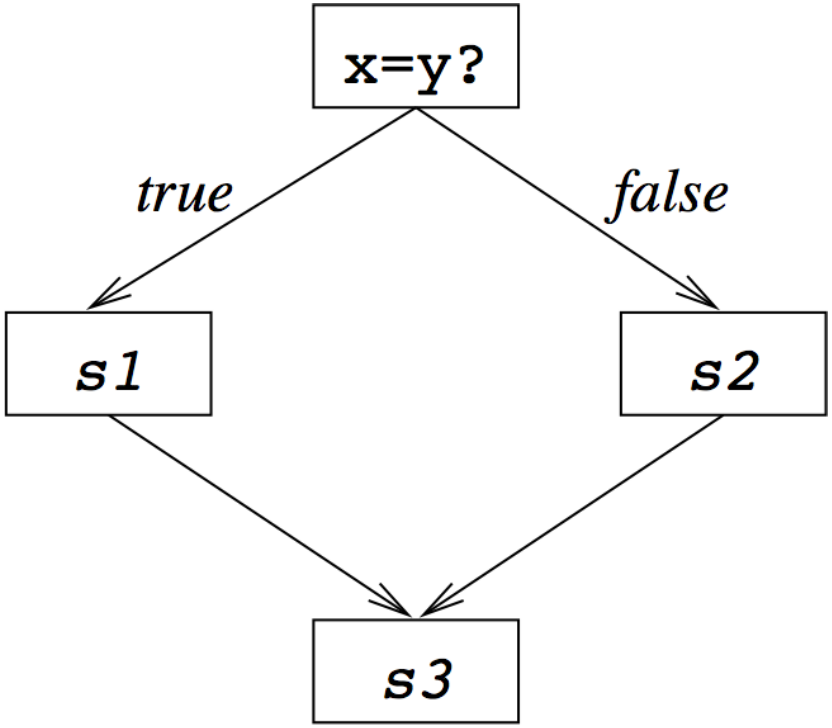
- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

Control flow graph

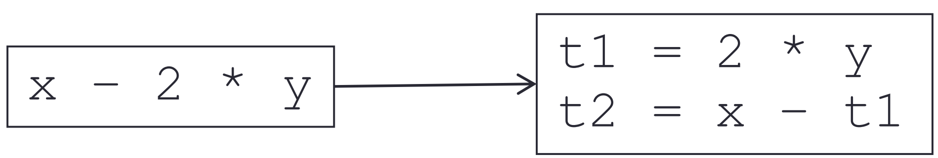
- A CFG models *transfer of control* in a program
  - nodes are *basic blocks* (straight-line blocks of code)
  - edges represent *control flow* (loops, if/else, goto …)

```
if x = y then
   S1
else
   S2
end
S3
```



3-address code

- Statements take the form: `x = y op z`
  - single operator and at most three names



> Advantages:

— compact form

— names for intermediate values

Typical 3-address codes

<table>
<tr><td rowspan="4">assignments</td><td><code>x = y op z</code></td></tr>
<tr><td><code>x = op y</code></td></tr>
<tr><td><code>x = y[i]</code></td></tr>
<tr><td><code>x = y</code></td></tr>
<tr><td>branches</td><td><code>goto L</code></td></tr>
<tr><td>conditional branches</td><td><code>if x relop y goto L</code></td></tr>
<tr><td>procedure calls</td><td><code>param x</code>
<code>param y</code>
<code>call p</code></td></tr>
<tr><td>address and pointer assignments</td><td><code>x = &y</code>
<code>*y = z</code></td></tr>
</table>

3-address code — two variants

Quadruples

`x - 2 * y`

(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

- simple record structure
- easy to reorder
- explicit names

Triples

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

- table index is implicit name
- only 3 fields
- harder to reorder

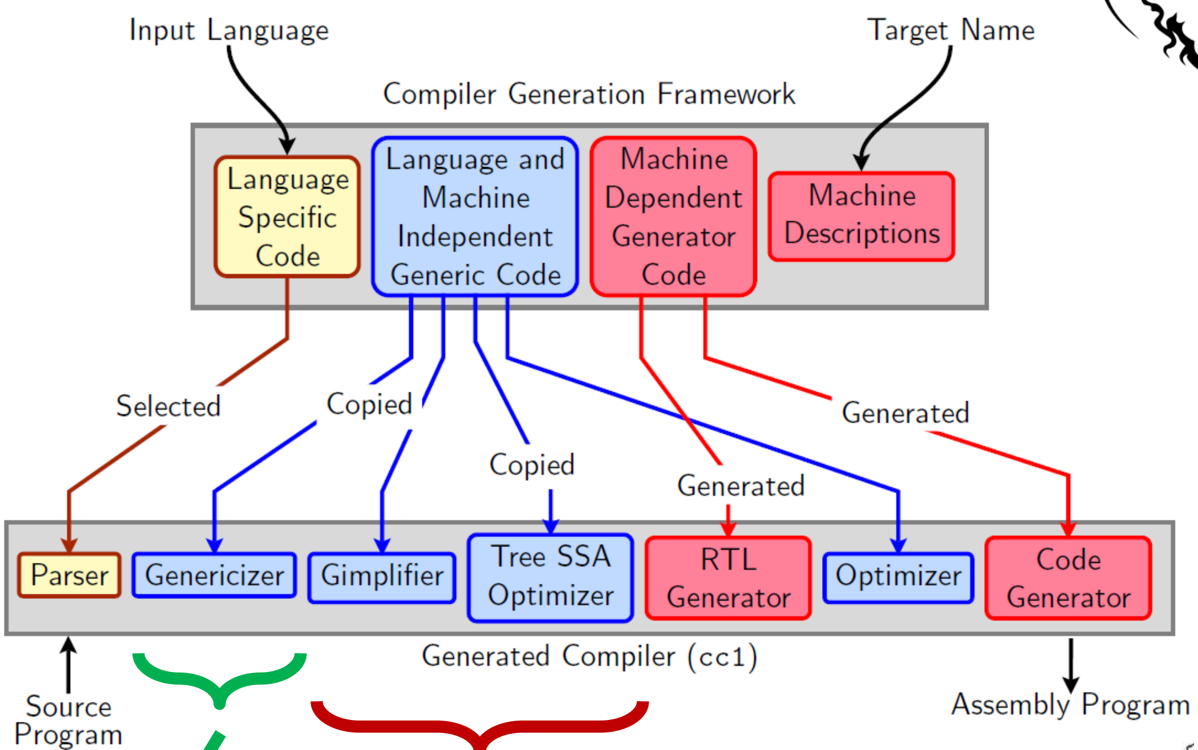
IR choices

- Other hybrids exist
  - combinations of graphs and linear codes
  - CFG with 3-address code for basic blocks
- Many variants used in practice
  - no widespread agreement
  - compilers may need several different IRs!

- Advice:
  - choose IR with right level of detail
  - keep manipulation costs in mind

The GNU Compiler Collection: gcc

GCC has three IRs



GENERIC
basically an AST

GIMPLE
3AC, SSA

RTL
simple (Lisp-inspired), low-level

The GNU Compiler Collection: gcc

GCC has three IRs

Previously, the only common IR was RTL (Register Transfer Language)

- *Drawbacks of RTL for high-level optimizations :*

- RTL is a low-level IR, works well for optimizations close to machine (e.g., register allocation)
- Some high level information is difficult to extract from RTL (e.g. array references, data types etc.)
- Optimizations involving such higher level information are difficult to do using RTL.
- Introduces stack too soon, even if later optimizations don't demand it.

The GNU Compiler Collection: gcc

GCC has three IRs

Why not ASTs for optimization?

ASTs contain detailed function information but are not suitable for optimization because

- Lack of a common representation
- No single AST shared by all front-ends
- So each language would have to have a different implementation of the same optimizations
- Difficult to maintain and upgrade so many optimization frameworks
- Structural Complexity
- Lots of complexity due to the syntactic constructs of each language

The GNU Compiler Collection: gcc

GCC has three IRs

Result: **GIMPLE**

The Goals of GIMPLE are

- Lower control flow
  - Program = sequenced statements + unrestricted jump
- Simplify expressions
  - Typically: two operand assignments!
- Simplify scope
  - move local scope to block begin, including temporaries

Notice

- Lowered control flow ! nearer to register machines + Easier SSA!

The GNU Compiler Collection: gcc

Example: C code (test.c)

```
int test1()
{
    int a;
    if (a)
    {
        int b;
        b = 2 + a + b;
    }
    return 0;
}
```

The GNU Compiler Collection: gcc

Example: GIMPLE code

```
int test1 ()
{
   int b;
   int a;
   int D.1196;
   int D.1195;

   # BLOCK 2
   # PRED: ENTRY (fallthru)
   if (a != 0)
          goto <bb 3>;
   else
          goto <bb 4>;
   # SUCC: 3 (true) 4 (false)

   # BLOCK 3
   # PRED: 2 (true)
   D.1195 = a + 2;
   b = D.1195 + b;
   # SUCC: 4 (fallthru)

   # BLOCK 4
   # PRED: 2 (false) 3 (fallthru)
   D.1196 = 0;
   # SUCC: 5 (fallthru)

   # BLOCK 5
   # PRED: 4 (fallthru)
   return D.1196;
   # SUCC: EXIT
}
```

NOTE: The CFG is encoded in GIMPLE

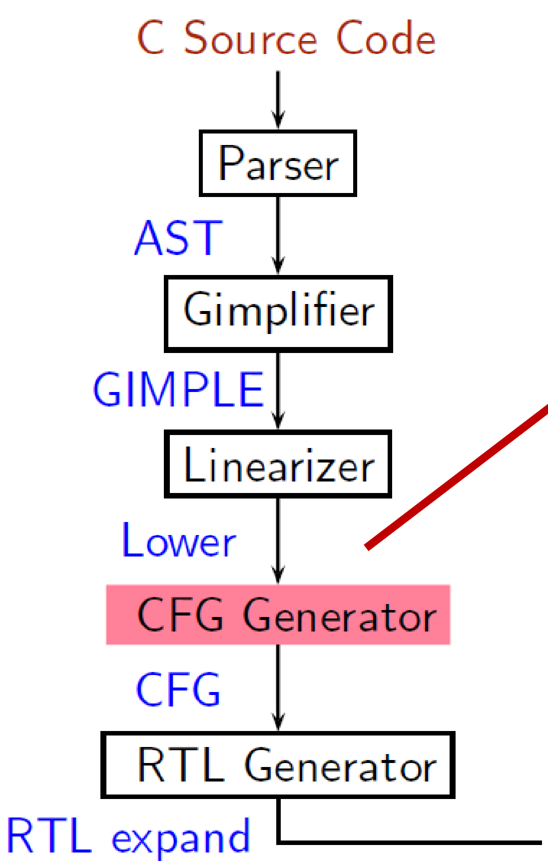
The GNU Compiler Collection: gcc

Example: GIMPLE code

```
main ()
{
  int b;
  int a;
  int D.1196;
  int D.1195;

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  if (a != 0)
      goto <bb 3>;
  else
      goto <bb 4>;
  # SUCC: 3 (true) 4 (false)

  # BLOCK 3
  # PRED: 2 (true)
  D.1195 = a + 2;
  b = D.1195 + b;
  # SUCC: 4 (fallthru)

  # BLOCK 4
  # PRED: 2 (false) 3 (fallthru)
  D.1196 = 0;
  # SUCC: 5 (fallthru)

  # BLOCK 5
  # PRED: 4 (fallthru)
  return D.1196;
  # SUCC: EXIT
}
```

The GNU Compiler Collection: gcc

Important phases of GCC (test.c)



Gimple

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
{
        goto <D.1196>;
}
else
{
        goto <D.1198>;
}
<D.1198>:;
```

```
void test2()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
        a = a+b+c;
}
```

- Three-address representation breaks expressions down into tuples of no more than 3 operands
- Temporaries are introduced to hold intermediate values needed to compute complex expressions.
- Control structures are explicated into conditional jumps.

The GNU Compiler Collection: gcc

Phases of GCC - dumping

To see output after each pass use the option
-fdump-<ir>-<pass>

- <ir>
  - -tree-<pass>
    - gimple
    - original
    - cfg etc.
    - Use -all to see all dumps
  - -rtl-<pass>
    - expand
    - greg
    - vreg etc
    - Use -all to see all dumps

Example: gcc -fdump-tree-all -fdump-rtl-all test.c

The GNU Compiler Collection: gcc

Phases of GCC - dumping

```
gcc -c -fdump-tree-all test.c
```

```
-rw-rw-r--  1 hero-vm hero-vm    998 May 23 17:07 test.c
-rw-rw-r--  1 hero-vm hero-vm 806079 May 23 17:08 test.c.001t.tu
-rw-rw-r--  1 hero-vm hero-vm      0 May 23 17:08 test.c.002t.class
-rw-rw-r--  1 hero-vm hero-vm    885 May 23 17:08 test.c.003t.original
-rw-rw-r--  1 hero-vm hero-vm    732 May 23 17:08 test.c.004t.gimple
-rw-rw-r--  1 hero-vm hero-vm    991 May 23 17:08 test.c.006t.omplower
-rw-rw-r--  1 hero-vm hero-vm   1028 May 23 17:08 test.c.007t.lower
-rw-rw-r--  1 hero-vm hero-vm   1028 May 23 17:08 test.c.010t.eh
-rw-rw-r--  1 hero-vm hero-vm   1590 May 23 17:08 test.c.011t.cfg
-rw-rw-r--  1 hero-vm hero-vm    995 May 23 17:08 test.c.012t.ompexp
-rw-rw-r--  1 hero-vm hero-vm    995 May 23 17:08 test.c.017t.fixup_cfg1
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.018t.ssa
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.025t.fixup_cfg3
-rw-rw-r--  1 hero-vm hero-vm   2287 May 23 17:08 test.c.026t.inline_param1
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.027t.einline
-rw-rw-r--  1 hero-vm hero-vm      0 May 23 17:08 test.c.042t.profile_estimate
-rw-rw-r--  1 hero-vm hero-vm   1198 May 23 17:08 test.c.045t.release_ssa
-rw-rw-r--  1 hero-vm hero-vm   2287 May 23 17:08 test.c.046t.inline_param2
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.068t.fixup_cfg4
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.183t.veclower
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.184t.cplxlower0
-rw-rw-r--  1 hero-vm hero-vm   1075 May 23 17:08 test.c.191t.optimized
-rw-rw-r--  1 hero-vm hero-vm      0 May 23 17:08 test.c.271t.statistics
```

Parsed C code

Low GIMPLE created

CFG created

Let's ignore these for now…

Outline

- Compiler structure
- Intermediate Representations
- Optimization

Optimization: The Idea

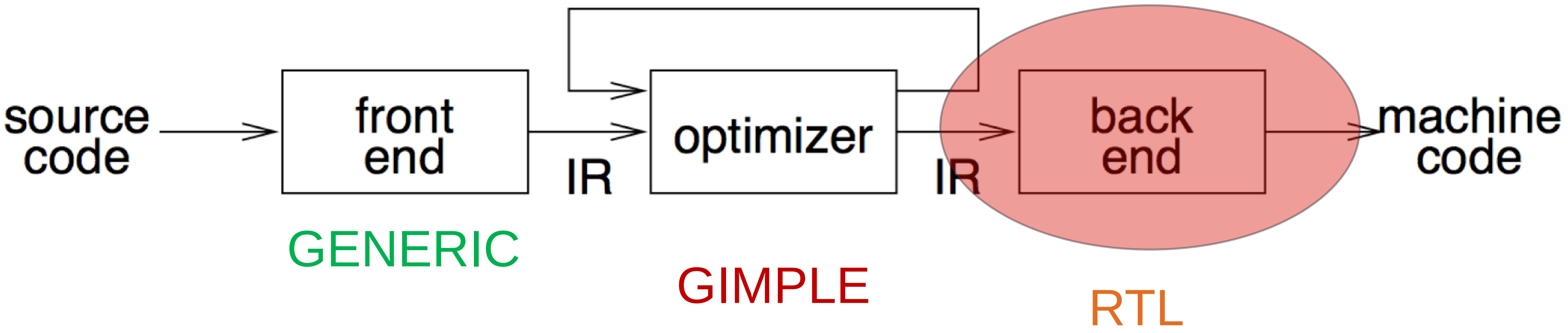
- Transform the program to improve efficiency

- **Performance**: faster execution
- **Size**: smaller executable, smaller memory footprint

Tradeoffs: 1) **Performance** vs**. Size**

2) **Compilation speed** and **memory**

Optimizations in the Backend



- Register Allocation
- Instruction Selection
- Peep-hole Optimization

Register Allocation

- Processor has only finite amount of registers
  - Can be very small (x86)

- Temporary variables
  - non-overlapping temporaries can share one register
- Passing arguments via registers

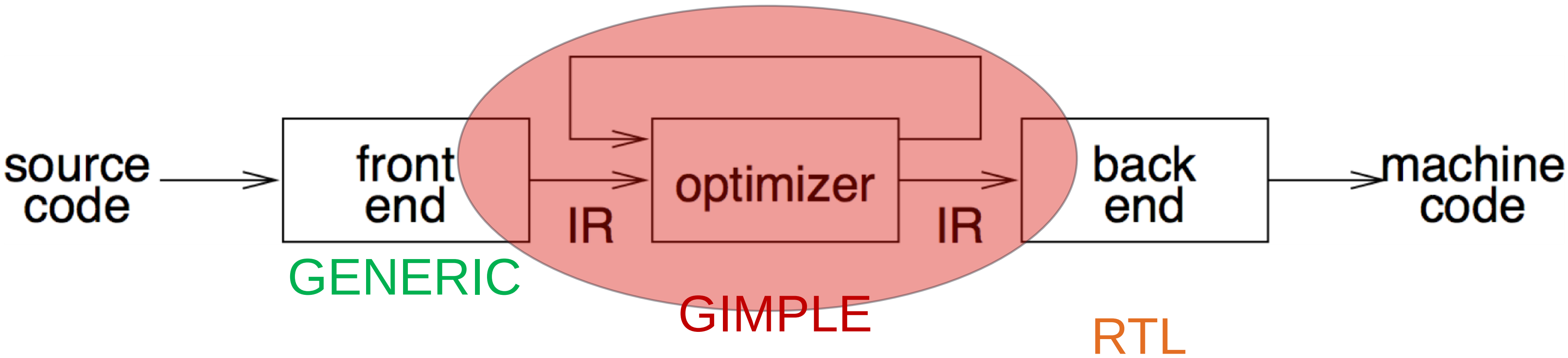
- Optimizing register allocation very important for good performance
  - Especially on x86

Instruction Selection

- For every expression, there are many ways to realize them for a processor
- Example: Multiplication*2 can be done by bit-shift

Instruction selection is a form of optimization

Optimization in the middle-end



OUR MAIN FOCUS

Examples of optimization: Constant Folding

- Evaluate constant expressions at compile time
- Only possible when side-effect freeness guaranteed

```
c:= 1 + 3    →    c:= 4

true not     →    false
```

Caveat: Floats — implementation could be different between machines!

Examples of optimization: Constant Propagation

- Variables that have constant value, e.g. b := 3
  - Later uses of b can be replaced by the constant
  - If no change of b between!

```
b := 3
c := 1 + b
d := b + c
```

→

```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Examples of optimization: Copy Propagation

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + x
d := x + c
```

```
x := y
c := 1 + y
d := y + c
```

Analysis needed, as y and x can be assigned more than once!

Examples of optimization: Algebraic Simplifications

- Use algebraic properties to simplify expressions

`-(-i)` → `i`

`b or: true` → `true`

Important to simplify code for later optimizations

Examples of optimization: Strength Reduction

- Replace expensive operations with simpler ones
- Example: Multiplications replaced by additions

```
y := x * 2
```

```
y := x + x
```

Peephole optimizations are often strength reductions

Examples of optimization: Dead Code

- Remove *unnecessary* code
  - e.g. variables assigned but never read

```
b := 3
c := 1 + 3
d := 3 + c
```

→

```
c := 1 + 3
d := 3 + c
```

> Remove code never reached

```
if (false)
 {a := 5}
```

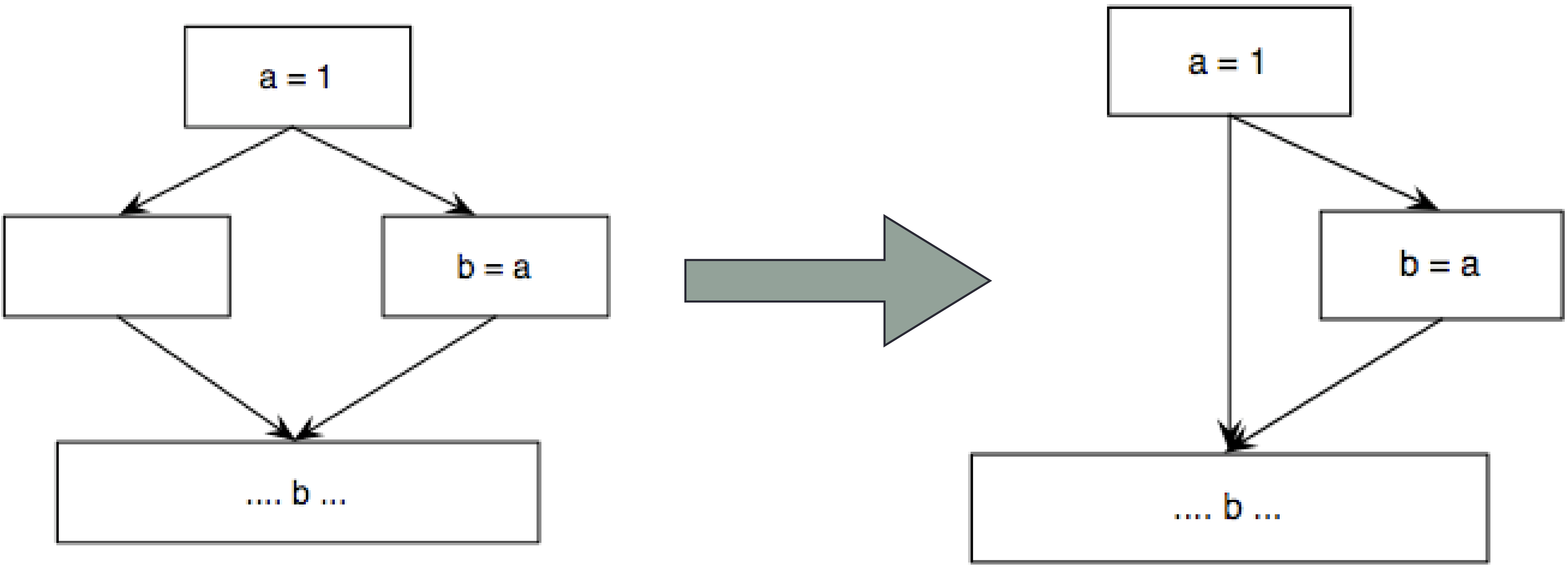
→

```
if (false) {}
```

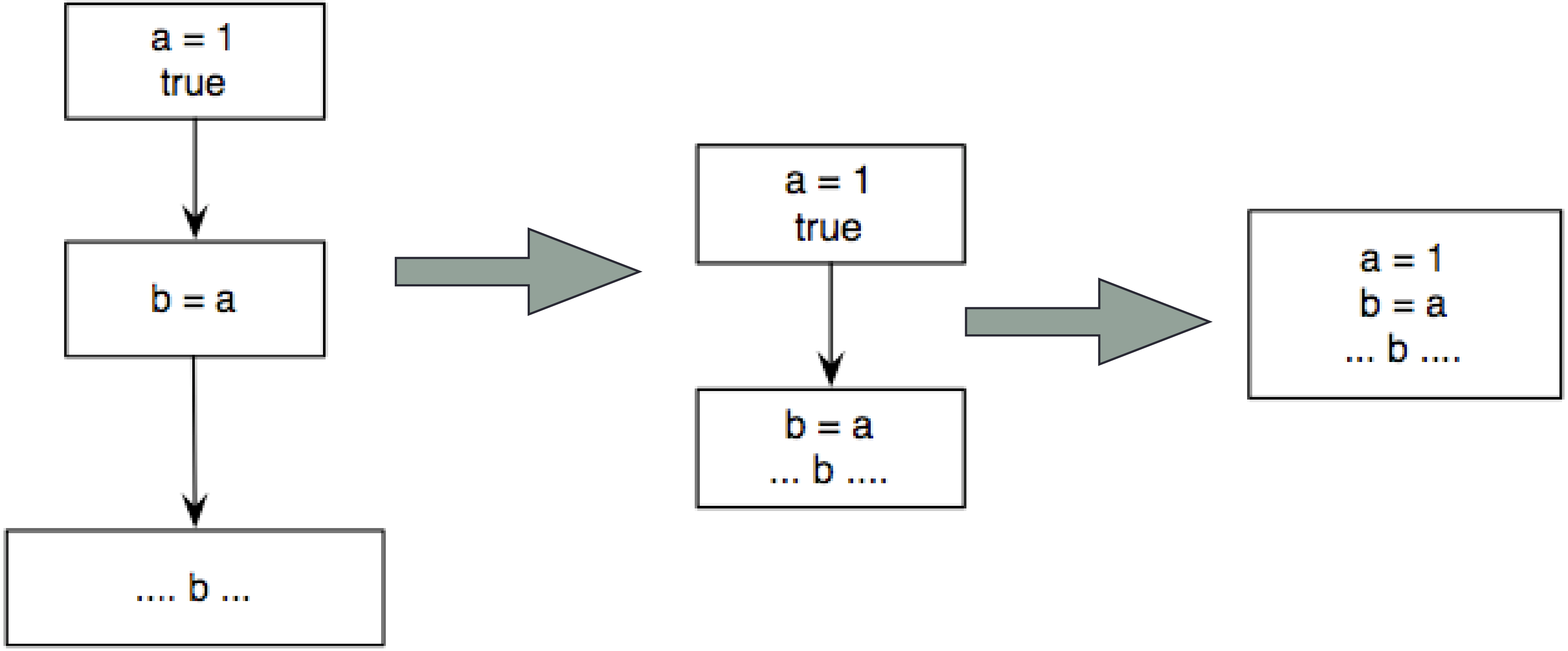
Examples of optimization: Simplify Structure

- Similar to dead code: Simplify CFG Structure
- Optimizations will degenerate CFG
- Needs to be cleaned to simplify further optimization!

Examples of optimization: Delete Empty Basic Blocks



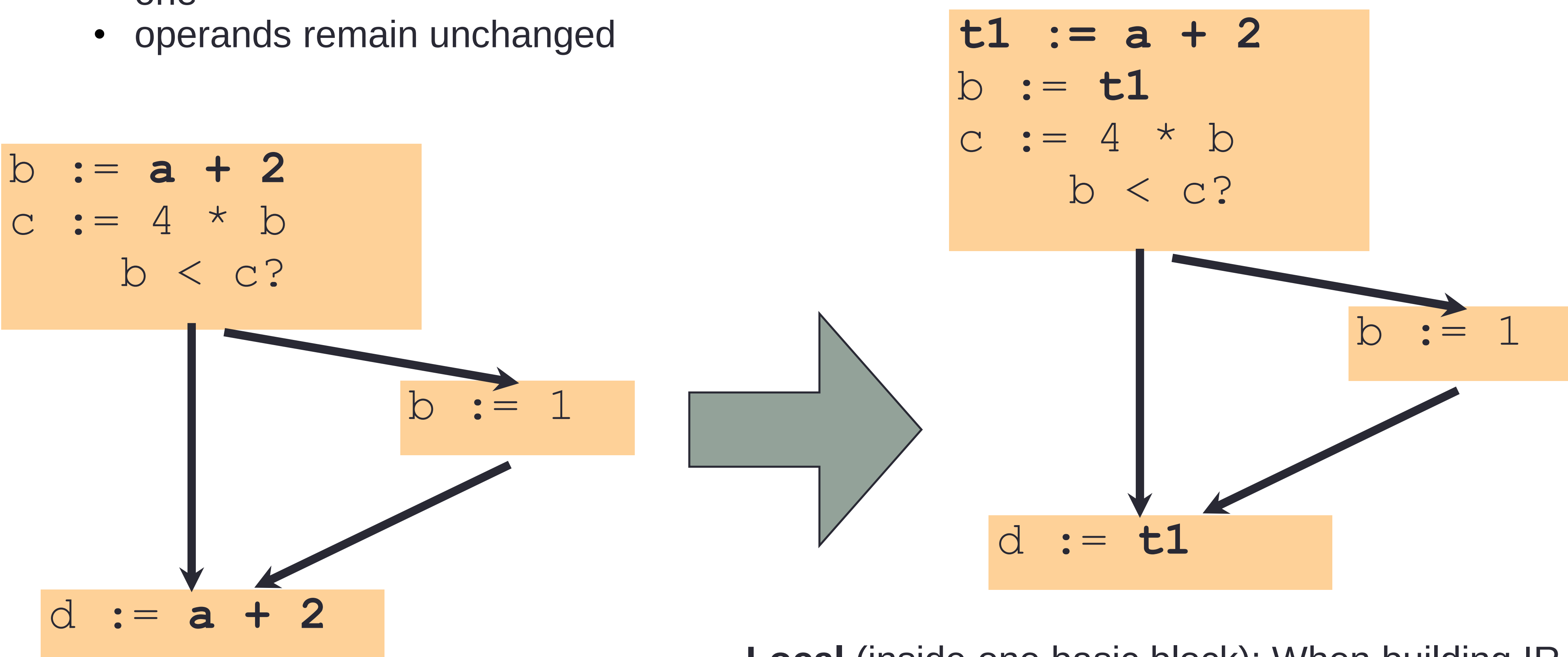
Examples of optimization: Fuse Basic Blocks



Examples of optimization: Common Subexpression Elimination (CSE)

Common Subexpression:

- There is another occurrence of the expression whose evaluation always precedes this one
- operands remain unchanged



Local (inside one basic block): When building IR
Global (complete flow-graph)

SSA form

- Static Single Assignment Form
  - Encodes information about data and control flow
  - Two constraints:
    - Each definition has a unique name
    - Each use refers to a single definition
      - all uses reached by a definition are renamed accordingly
  - Advantages:
    - Simplifies data flow analysis & several optimizations
    - SSA size is linear to program size
    - Eliminates certain dependences (write-after-read, write-after-write)
  - Example:

x := 5
x := x +1
y := x * 2

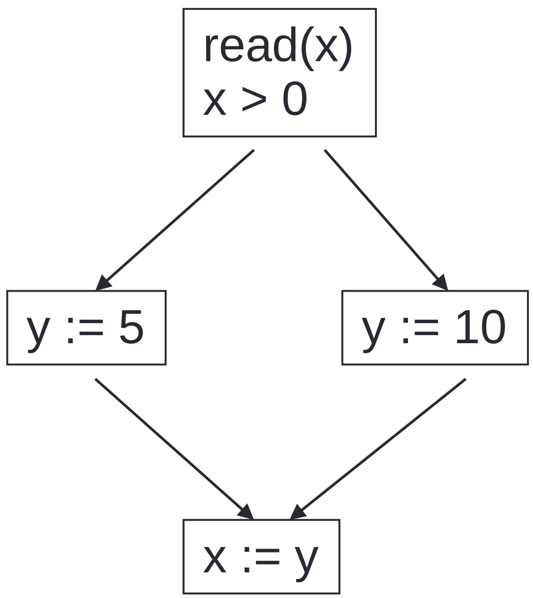
➡

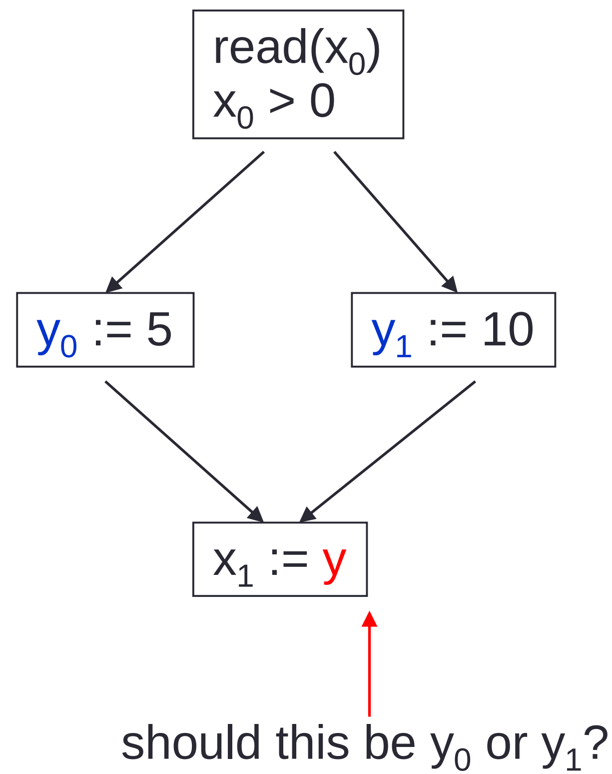
x_0 := 5
x_1 := x_0 +1
y_0 := x_1 * 2

SSA form

- Consider a situation where two control-flow paths merge (e.g. due to a loop, or an if-statement)

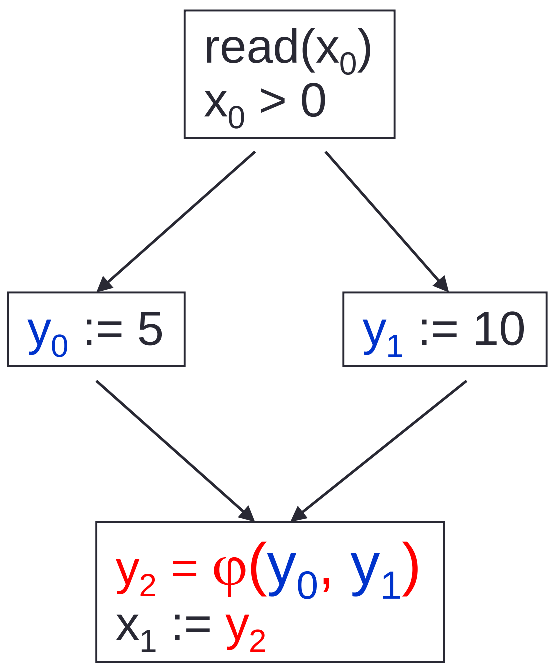
```
read(x)
if ( x > 0)
    y:= 5
else
    y:=10
x := y
```



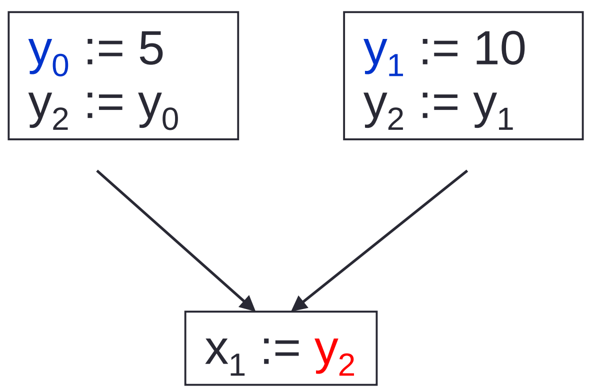


SSA form

- The compiler inserts special join functions (called φ-functions) at points where different control flow paths meet.



φ is not an executable function!
If we do need to generate executable code from this form, we insert appropriate copy statements in the predecessors:



SSA Optimizations

- SSA: Static Single Assignment Form
- **Definition**: Every variable is only assigned once
- Simplifies analysis and optimization (in many cases)

Properties

- Definitions of variables (assignments) have a list of all uses
- Variable uses (reads) point to the one definition
- CFG of Basic Blocks

Examples: Optimization on SSA

- We take three simple ones:
  - Constant Propagation
  - Copy Propagation
  - Simple Dead Code Elimination

Recall: Constant Propagation

- Variables that have constant value, e.g. b := 3
  - Later uses of b can be replaced by the constant
  - If no change of b between!

```
b := 3
c := 1 + b
d := b + c
```

→

```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Now the two **uses** of b refer to different **definitions**

```
b := 3
c := 1 + b
b := 4
d := b + c
```

→

```
b := 3
c := 1 + 3
b := 4
d := 4 + c
```

Constant Propagation and SSA

- Variables are assigned once
- We know that we can replace all uses by the constant!

```
b1 := 3
c1 := 1 + b1
d1 := b1 + c1
```

→

```
b1 := 3
c1 := 1 + 3
d1 := 3 + c1
```

With SSA names **uses** are easily associated to their **definitions**

```
b1 := 3
c1 := 1 + b1
b2 := 4
d1 := b2 + c1
```

→

```
b1 := 3
c1 := 1 + 3
b2 := 4
d1 := 4 + c1
```

Recall: Copy Propagation

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + x
d := x + c
```

```
x := y
c := 1 + y
d := y + c
```

Analysis needed, as y and x can be assigned more than once!

Copy Propagation and SSA

- for a statement x1 := y1
- replace later uses of x1 with y1

```
x1 := y1
c1 := 1 + x1
d1 := x1 + c1
```

```
x1 := y1
c1 := 1 + y1
d1 := y1 + c1
```

Dead Code Elimination and SSA

- Variable is *live* if the list of uses is not empty.

- Dead definitions can be deleted
  - (If there is no side-effect)

Let’s try it on **gcc**

First step is to build the SSA form

```
int func ()
{
  int i=1, j=1, k=0;

  while (k < 200)
    if (j < 20)
    {
      j = i;
      k++;
    }
    else
    {
      j = k;
      k += 2;
    }

    return j;
}
```

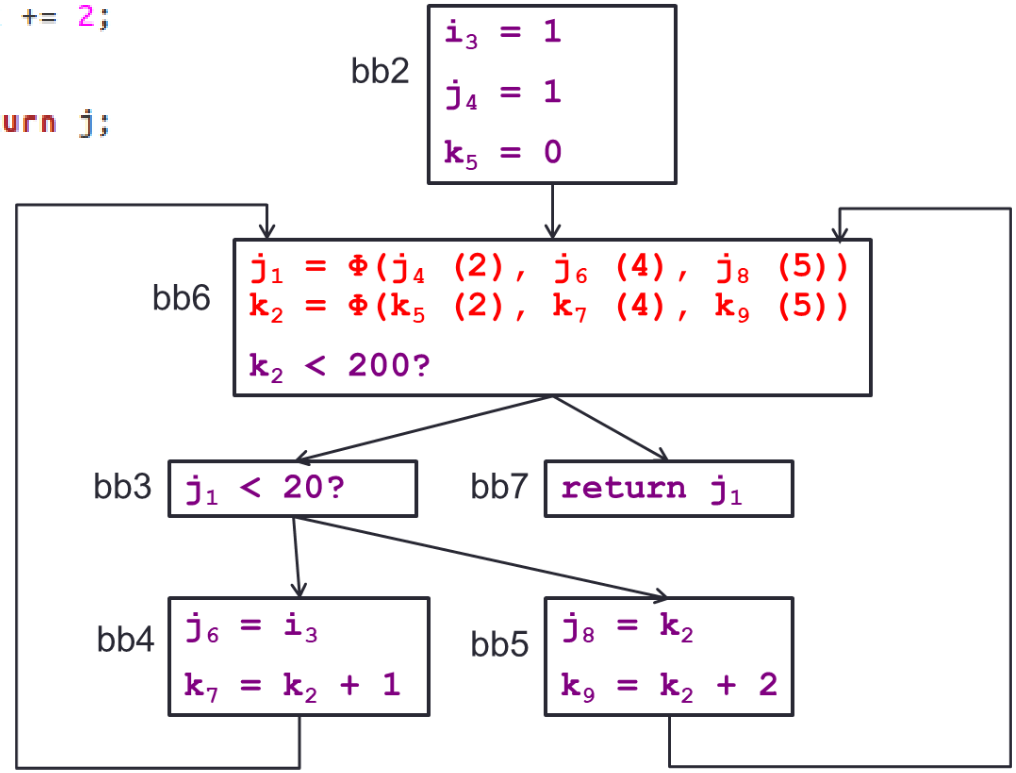
This happens at pass **pass_build_ssa** (*)

```
gcc -c -fdump-tree-ssa func.c
```

(*) Source code is in <HERO_SDK>/hero-gcc-toolchain/src/riscv-gcc/gcc/tree-into-ssa.c

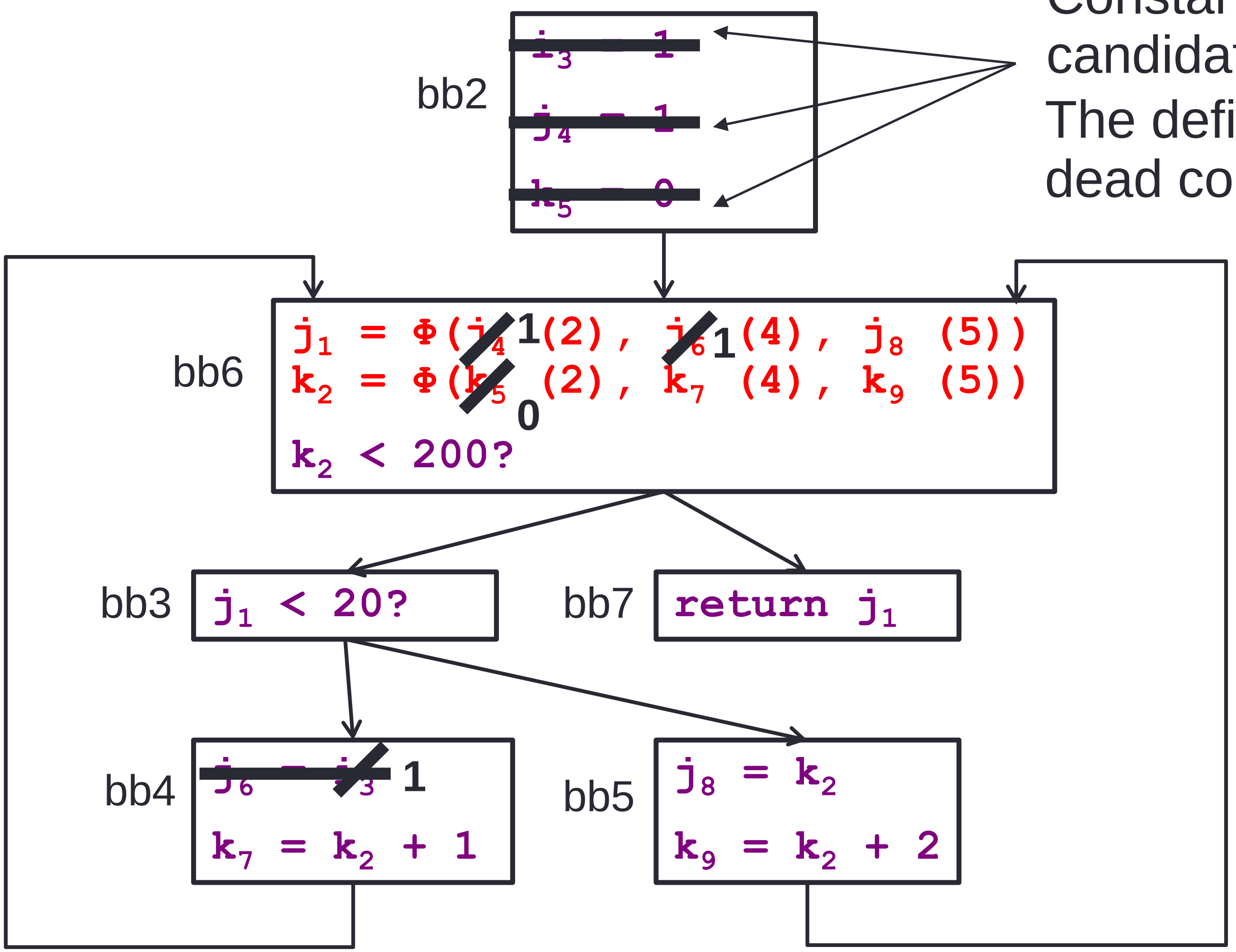
Into SSA *(func.c.018t.ssa)*

```c
int helloworld ()
{
  int i=1, j=1, k=0;

  while (k < 200)
  if (j < 20)
  {
    j = i;
    k++;
  }
  else
  {
    j = k;
    k += 2;
  }

  return j;
}
```




```
helloworld ()
{
  int k;
  int j;
  int i;
  int _10;

  <bb 2>:
  i_3 = 1;
  j_4 = 1;
  k_5 = 0;
  goto <bb 6>;

  <bb 3>:
  if (j_1 <= 19)
    goto <bb 4>;
  else
    goto <bb 5>;

  <bb 4>:
  j_6 = i_3;
  k_7 = k_2 + 1;
  goto <bb 6>;

  <bb 5>:
  j_8 = k_2;
  k_9 = k_2 + 2;

  <bb 6>:
  # j_1 = PHI <j_4(2), j_6(4), j_8(5)>
  # k_2 = PHI <k_5(2), k_7(4), k_9(5)>
  if (k_2 <= 199)
    goto <bb 3>;
  else
    goto <bb 7>;

  <bb 7>:
  _10 = j_1;
  return _10;

}
```

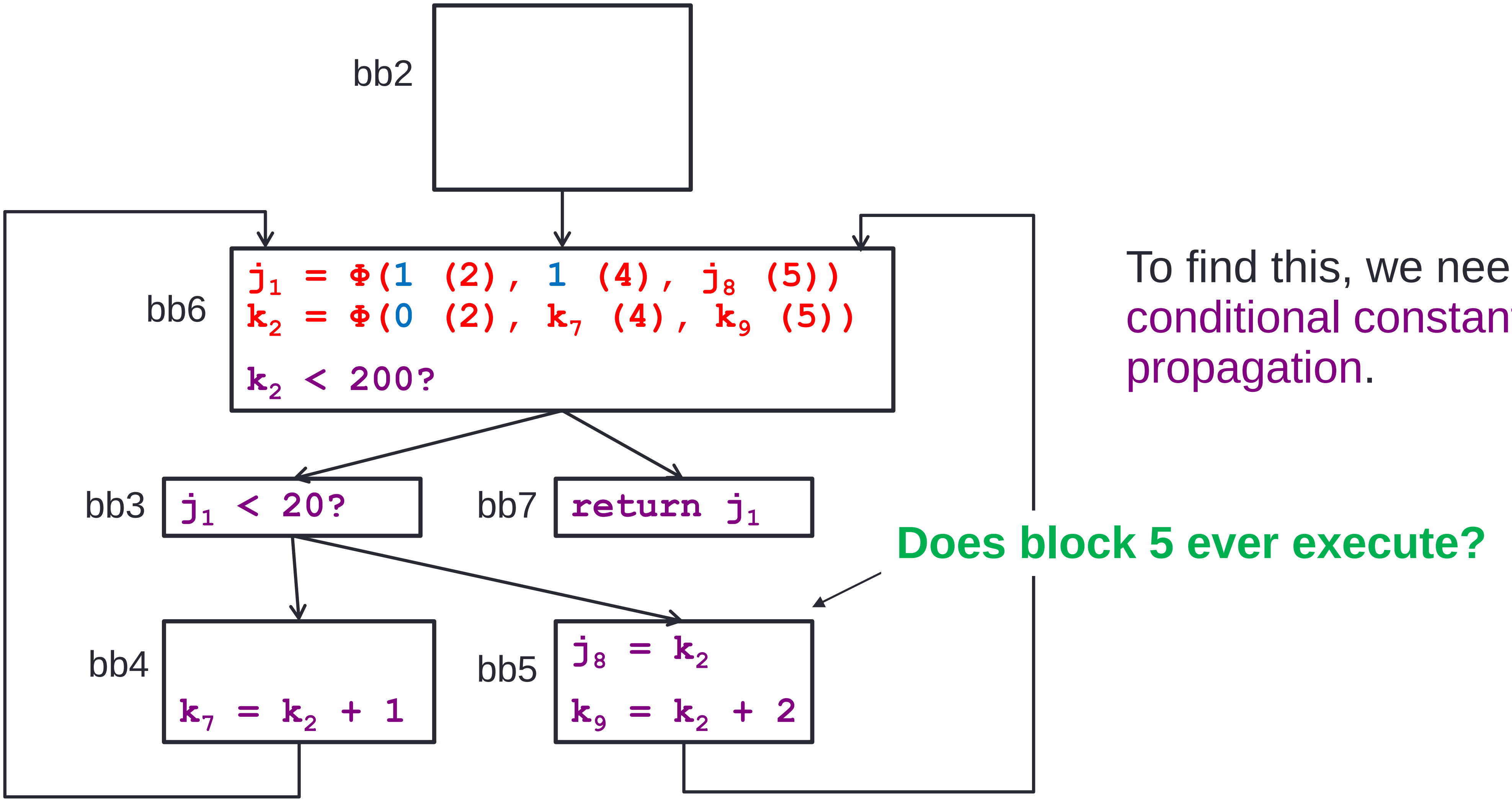
Constant Propagation Example



Constant propagation candidates.

The definitions becomes dead code. Why?

Constant Propagation Example

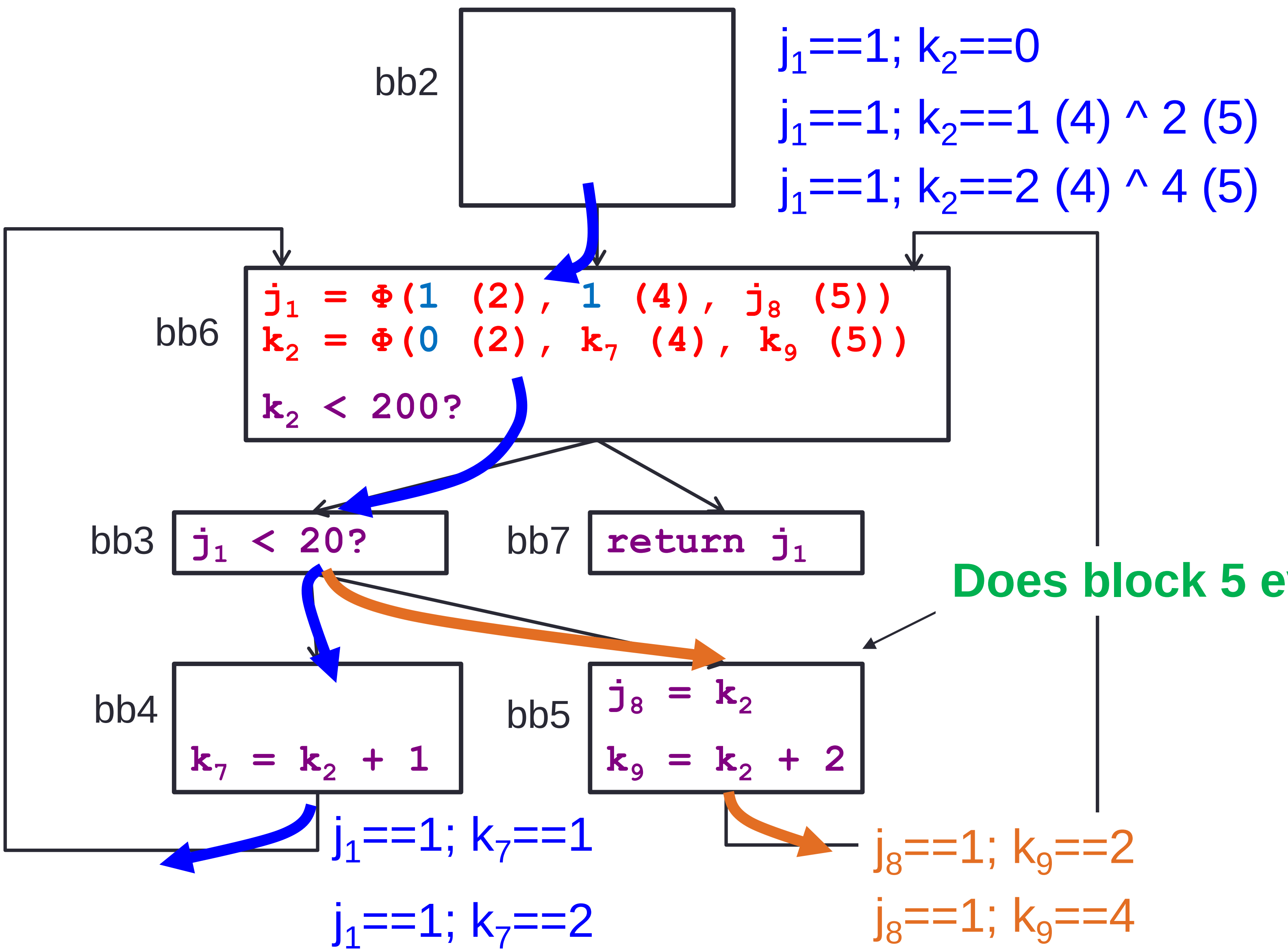


To find this, we need conditional constant propagation.

Conditional Constant Propagation

We found an invariant:

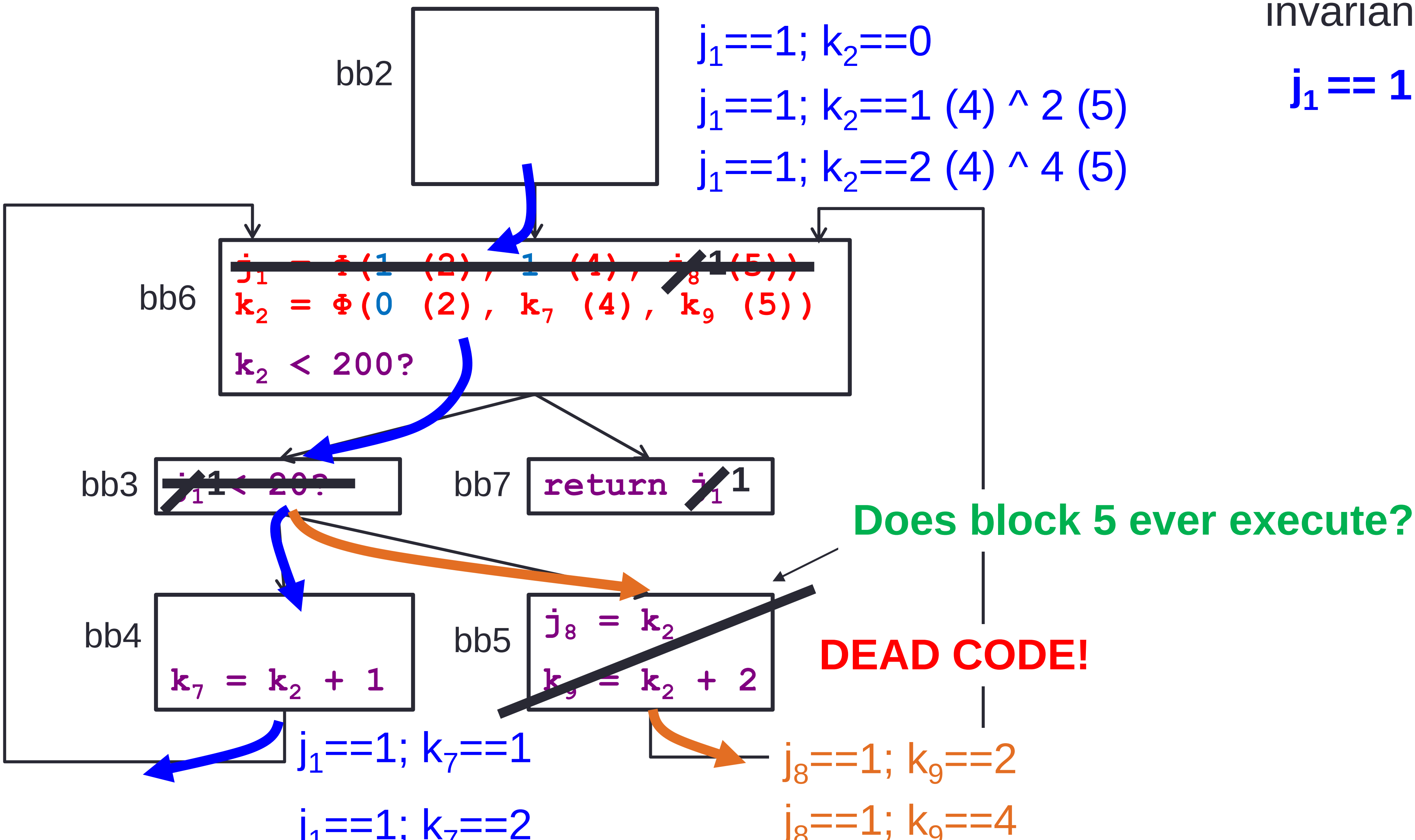
$j_1 == 1$



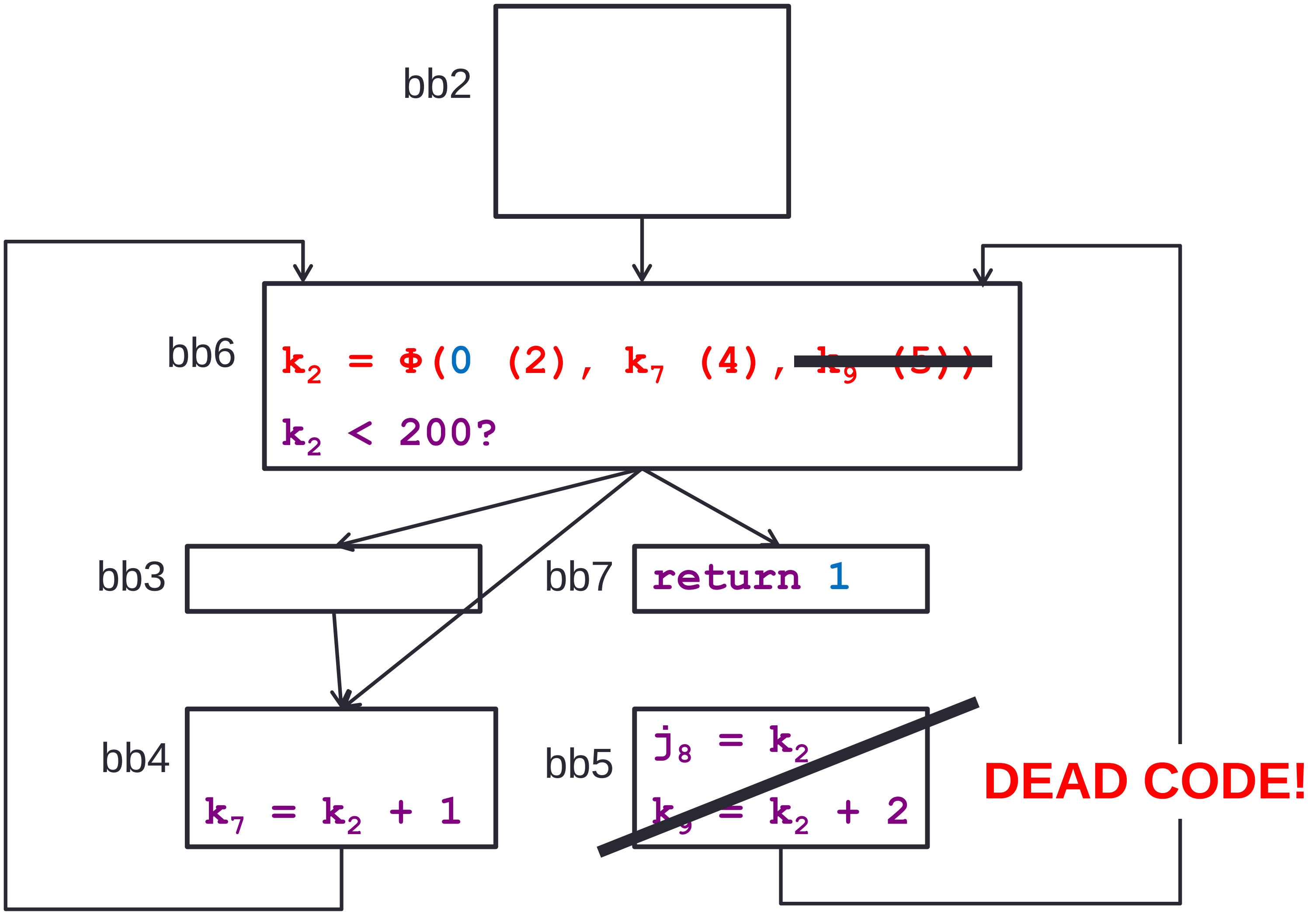
Conditional Constant Propagation

We found an invariant:

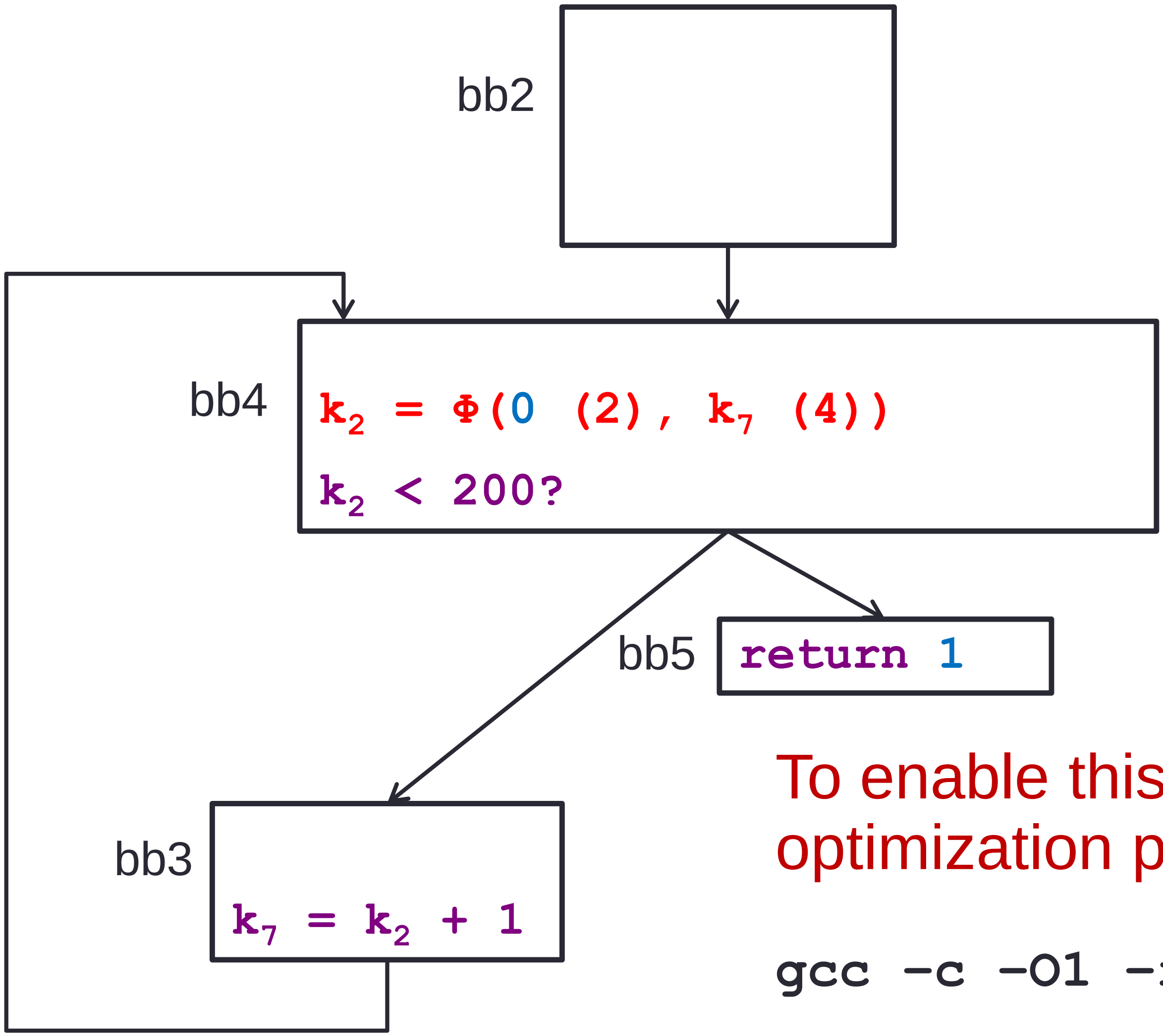
$j_1 == 1$



Conditional Constant Propagation



Conditional Constant Propagation



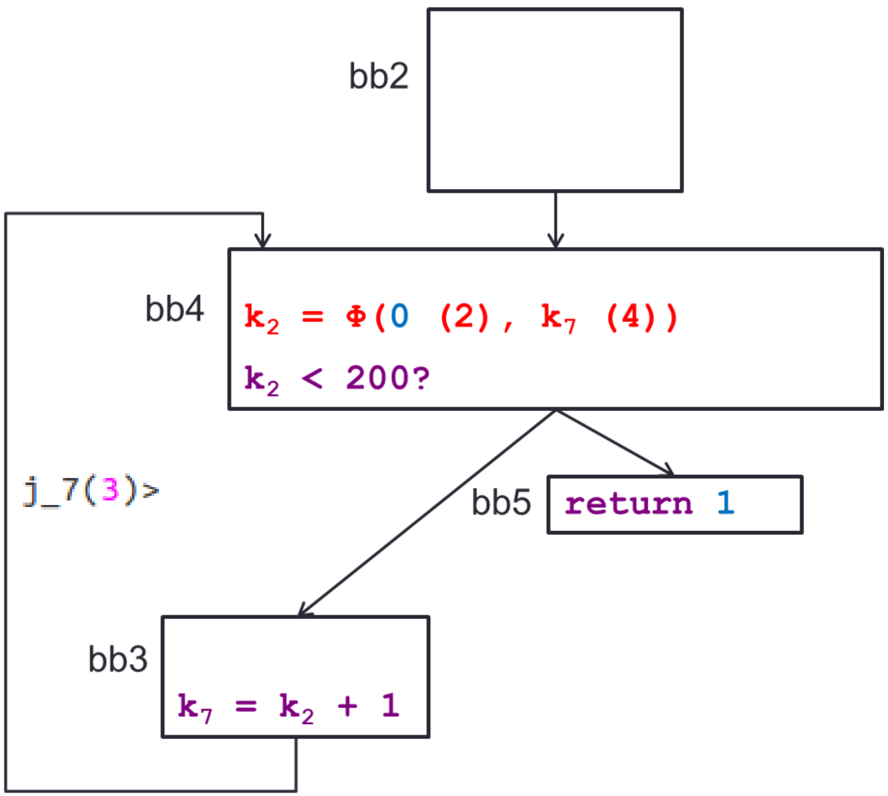
To enable this in GCC we need optimization pass **do_ssa_ccp** (*)

```
gcc -c -O1 -fdump-tree-ccp func.c
```

(*) Source code is in <HERO_SDK>/hero-gcc-toolchain/src/riscv-gcc/gcc/tree-ssa-ccp.c

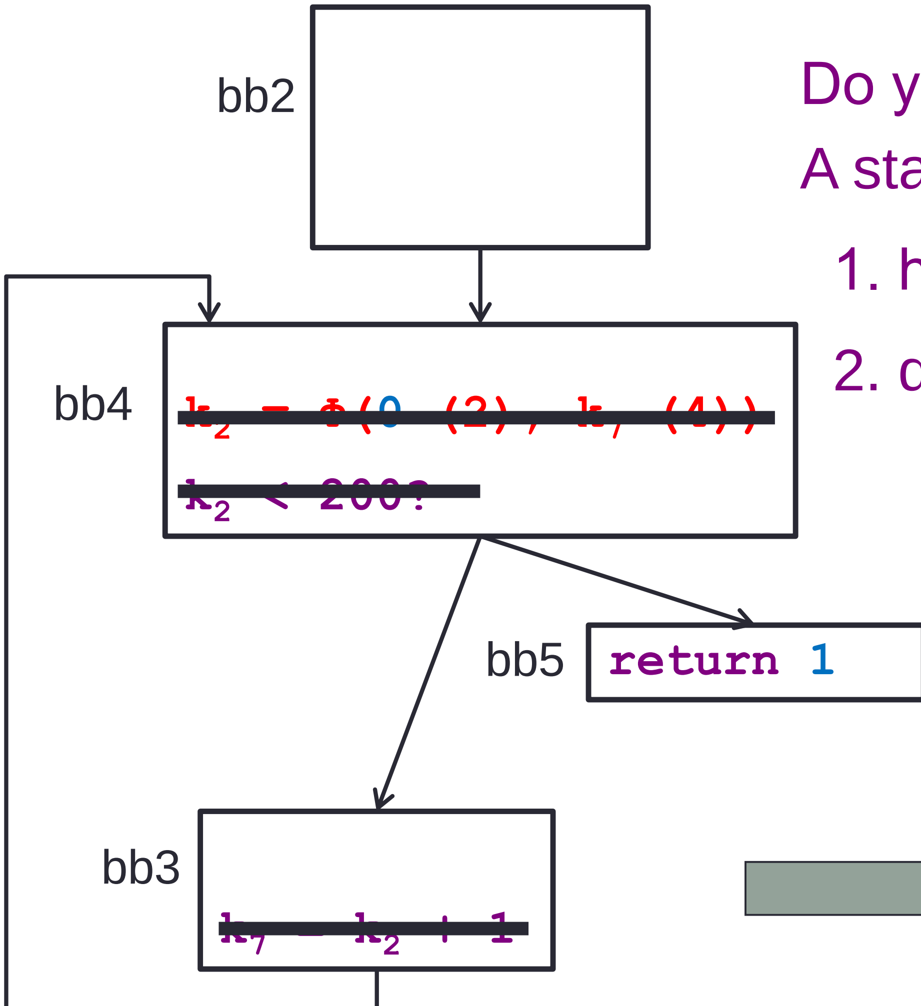
Conditional Constant Propagation *(func.c.031t.ccp1)*

```
;; Function helloworld (helloworld, funcdef_no=38, decl_uid=3152,
symbol_order=43)

Folding predicate j_1 <= 19 to 1
Removing basic block 5
Removing basic block 3
helloworld ()
{
  int k;
  int j;
  int i;

  <bb 2>:
  goto <bb 4>;

  <bb 3>:
  j_7 = j_2 + 1;

  <bb 4>:
  # j_2 = PHI <0(2), j_7(3)>
  if (j_2 <= 199)
    goto <bb 3>;
  else
    goto <bb 5>;

  <bb 5>:
  return 1;

}
```




```
helloworld ()
{
  int k;
  int j;
  int i;
  int _10;

  <bb 2>:
  i_3 = 1;
  j_4 = 1;
  k_5 = 0;
  goto <bb 6>;

  <bb 3>:
  if (j_1 <= 19)
    goto <bb 4>;
  else
    goto <bb 5>;

  <bb 4>:
  j_6 = i_3;
  k_7 = k_2 + 1;
  goto <bb 6>;

  <bb 5>:
  j_8 = k_2;
  k_9 = k_2 + 2;

  <bb 6>:
  # j_1 = PHI <j_4(2), j_6(4), j_8(5)>
  # k_2 = PHI <k_5(2), k_7(4), k_9(5)>
  if (k_2 <= 199)
    goto <bb 3>;
  else
    goto <bb 7>;

  <bb 7>:
  _10 = j_1;
  return _10;

}
```

Aggressive Dead Code Elimination



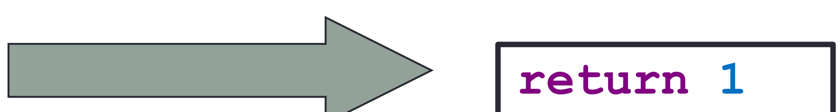
Do you notice the "useless" code on the left?

A statement is live, iff:

1. has side effect (I/O, call, store, …), and
2. defines a var which is used in live stm.

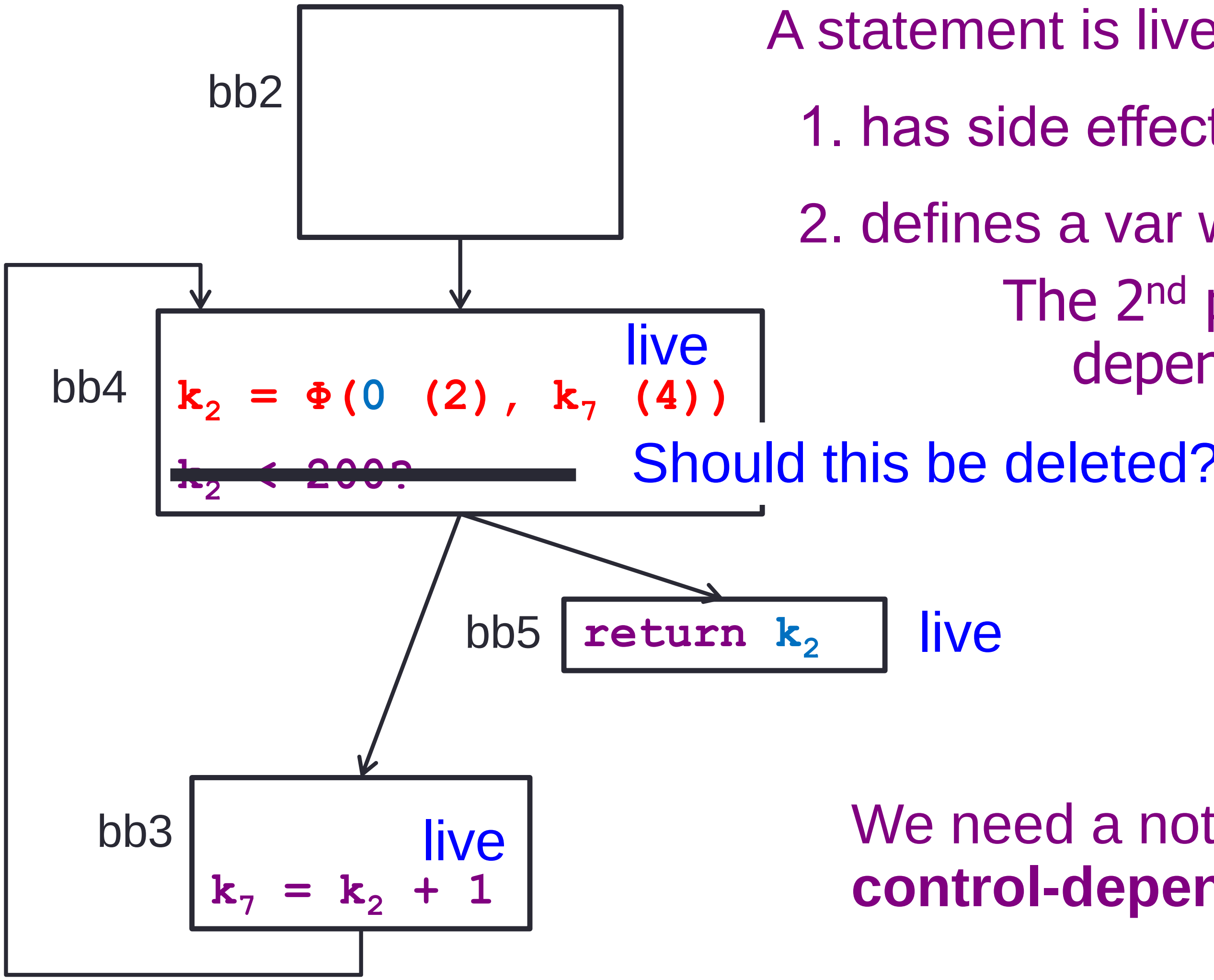
Lattice algorithm again:

1. init live stm
2. mark all other dead until proven to be live.



This function can be further inlined and eliminated!

Aggressive Dead Code Elimination: pitfalls



A statement is live, iff:

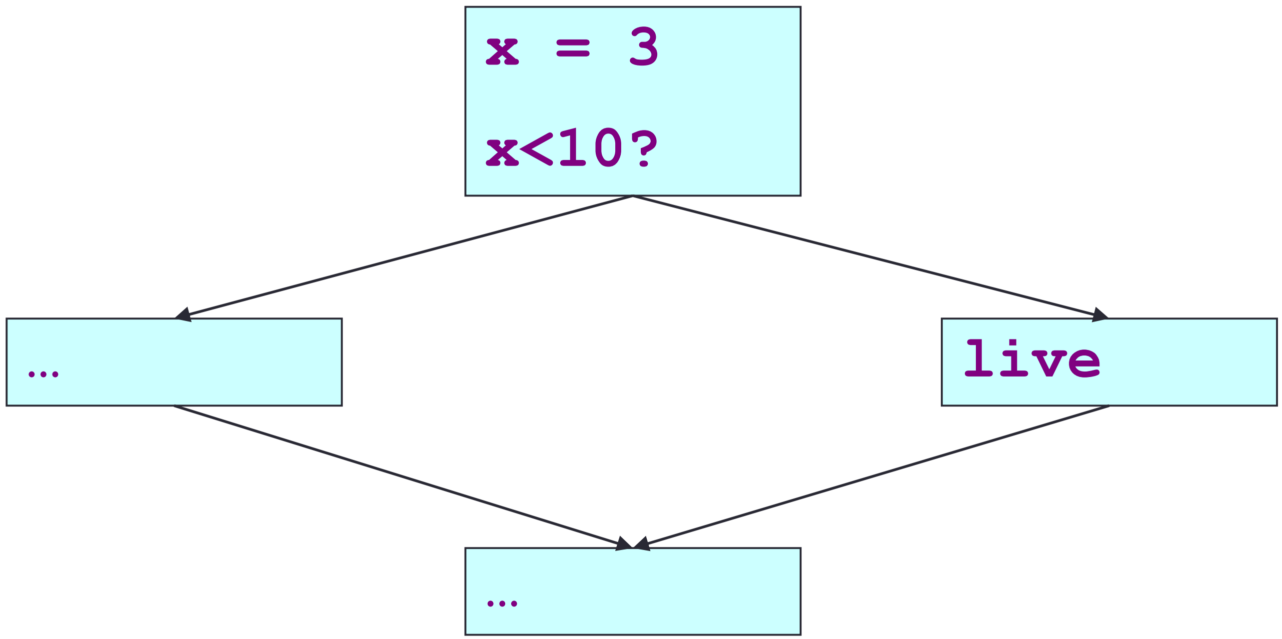
1. has side effect (I/O, call, store, …), and
2. defines a var which is used in live stm.

The 2nd property is call data-dependency!

We need a notion of **control-dependency!**

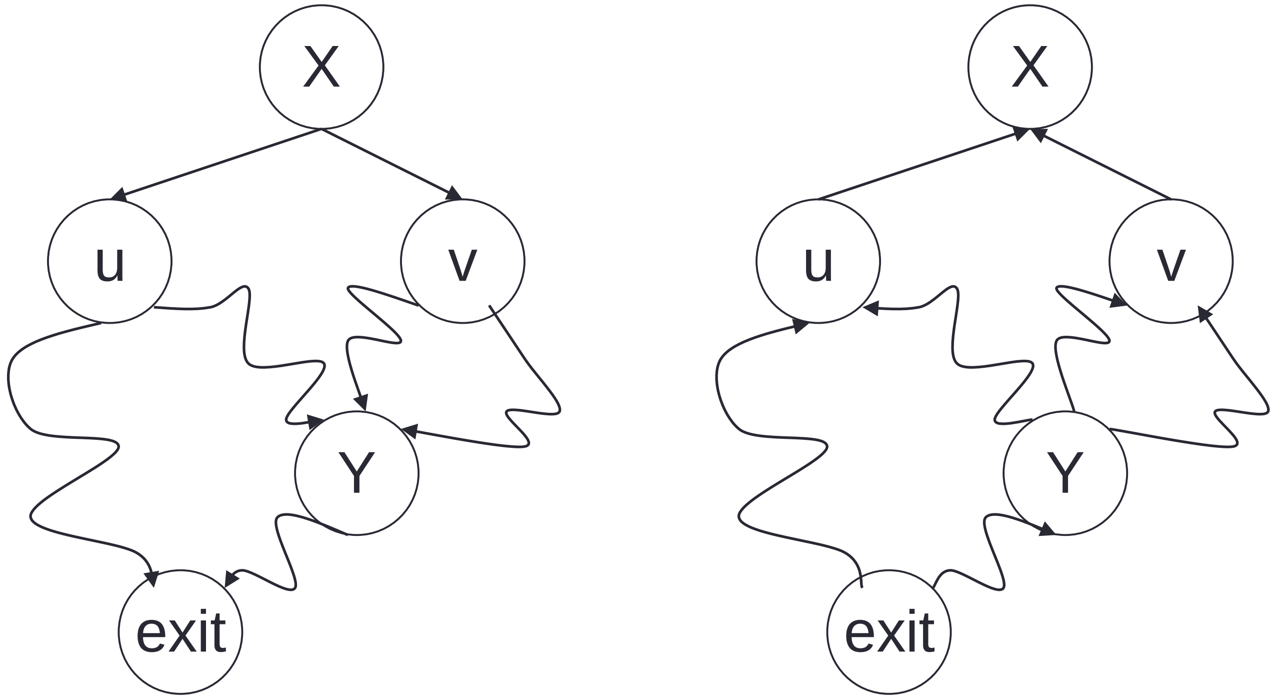
Fixing this problem

- If a statement S is live, then
  - if T S is control-dependent on T, T should also be live

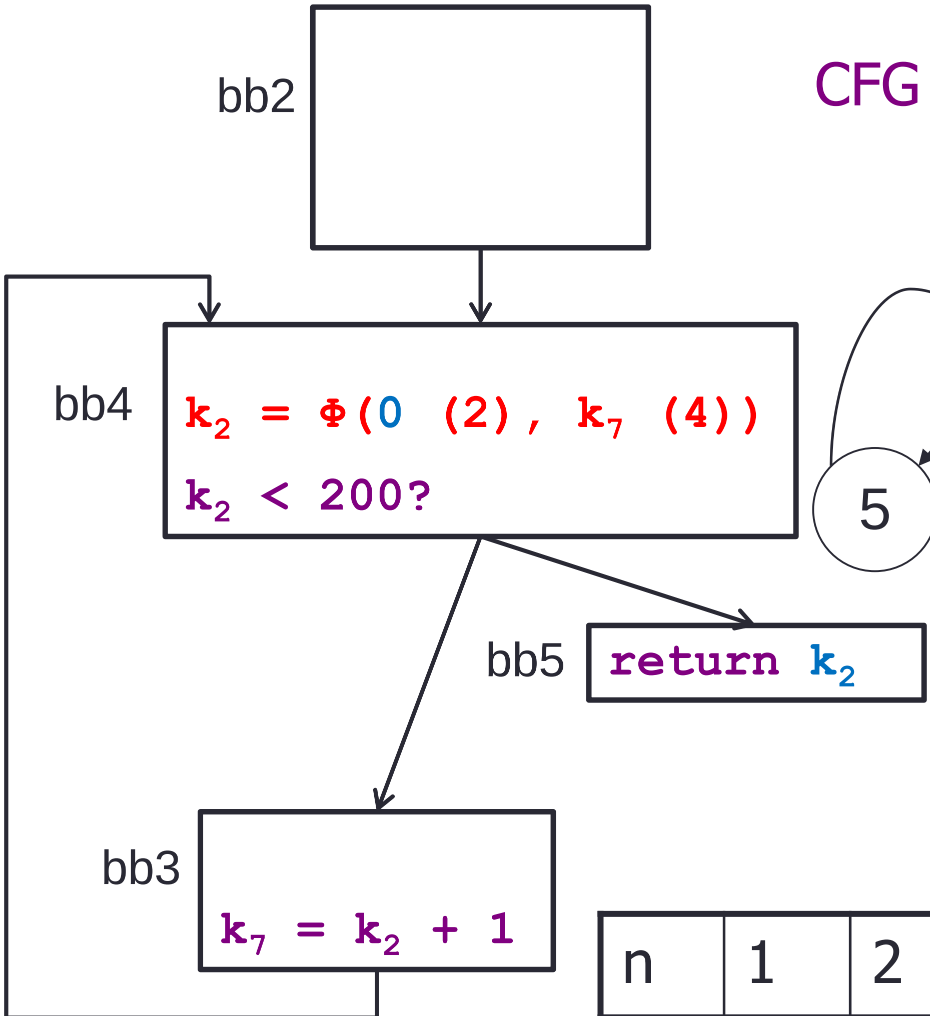


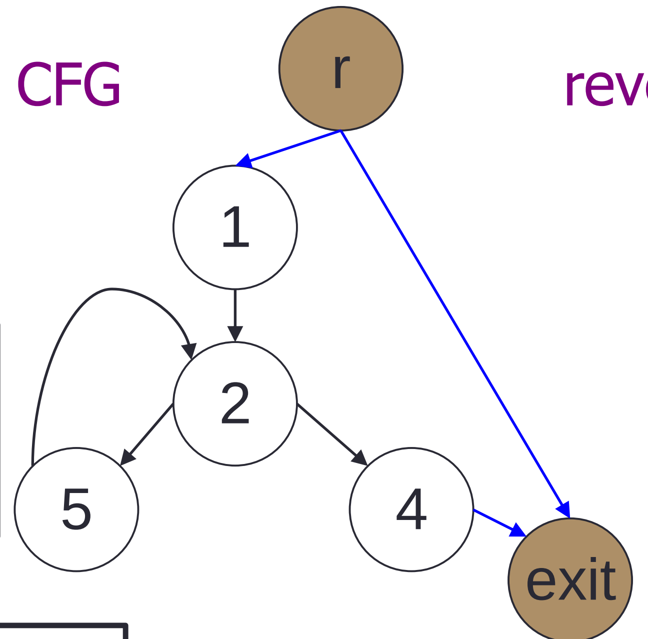
Control Dependency

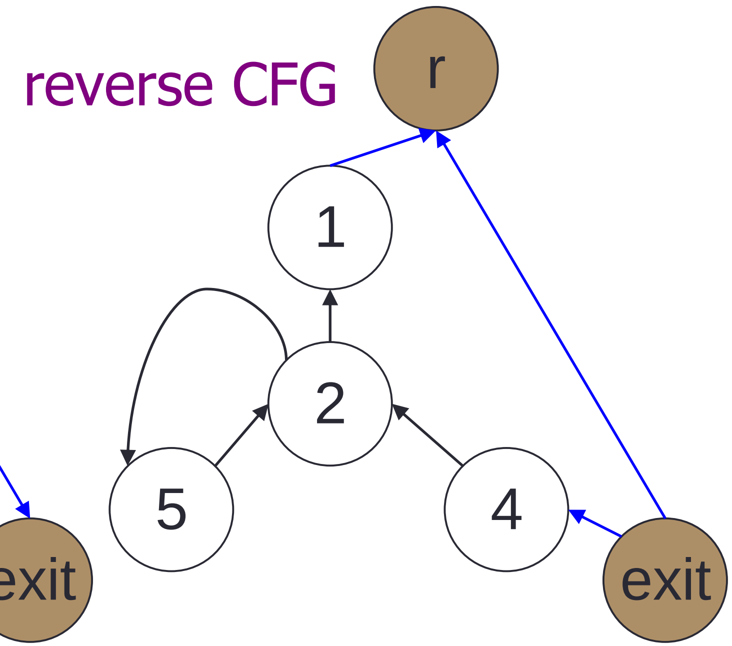
- A block Y is control-dependent on X iff
  - there exists an edge X->v, which v->exit goes through Y
  - there exists a path X->exit which does not go through y

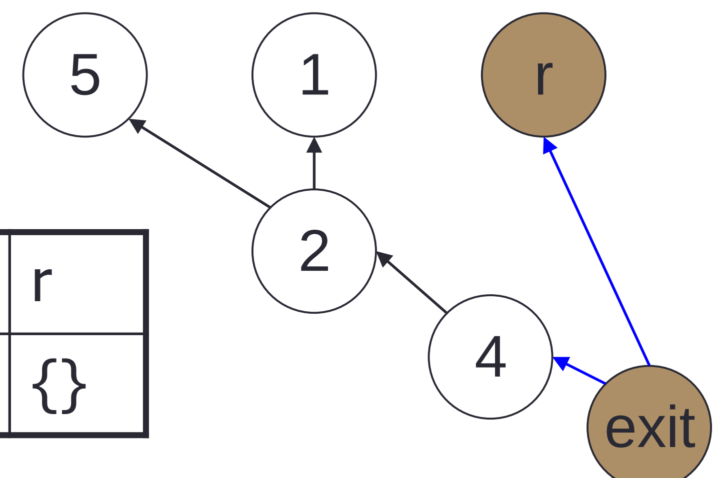


Aggressive Dead Code Elimination Example



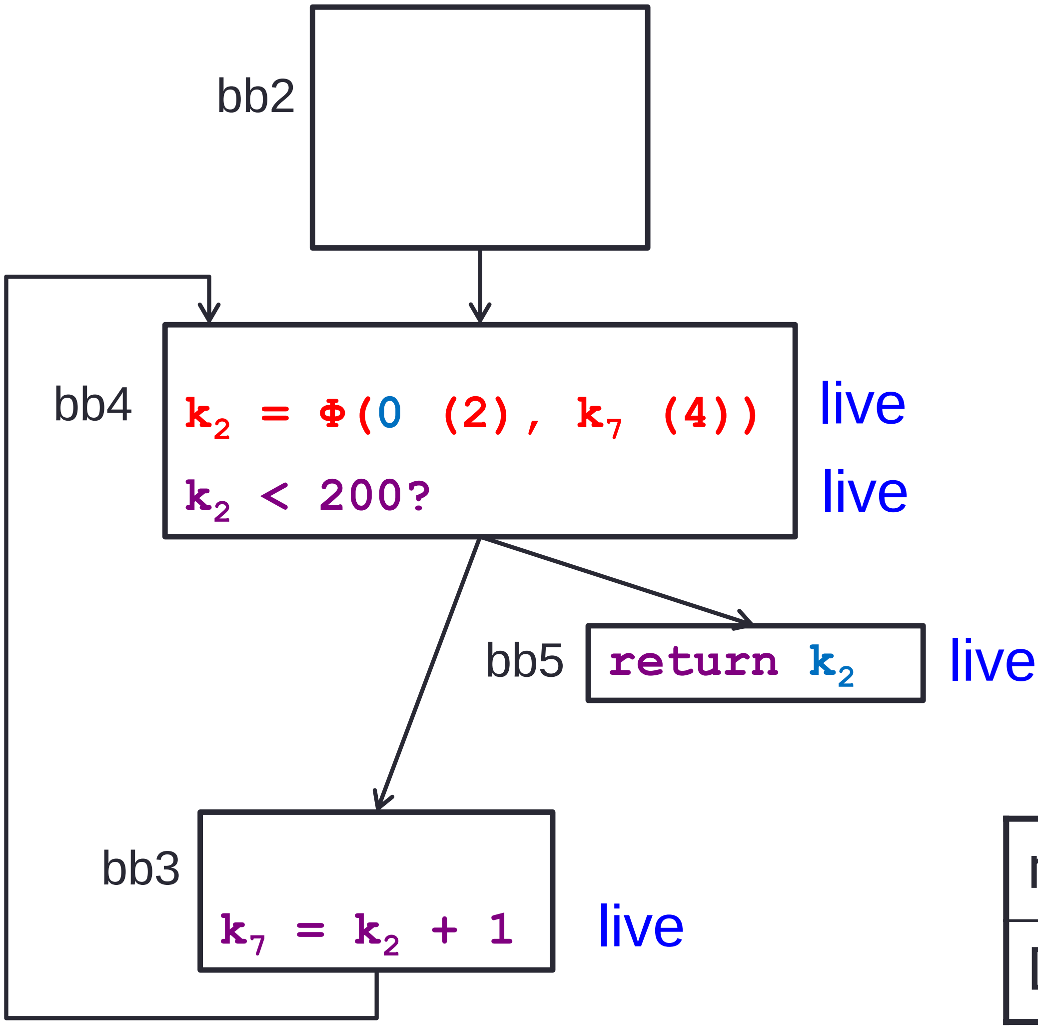


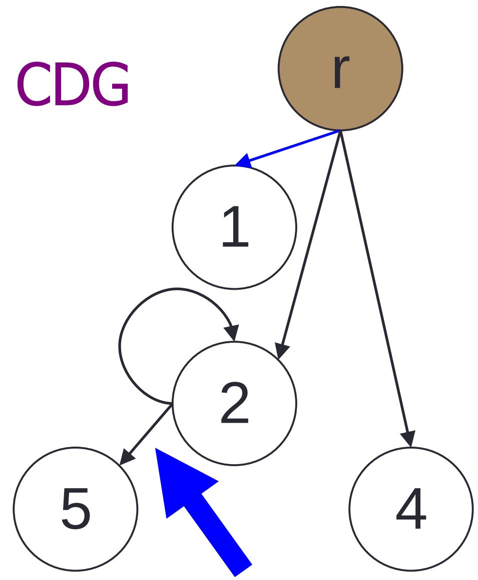




n	1	2	4	5	r
DF	{r}	{2, r}	{r}	{2}	{}

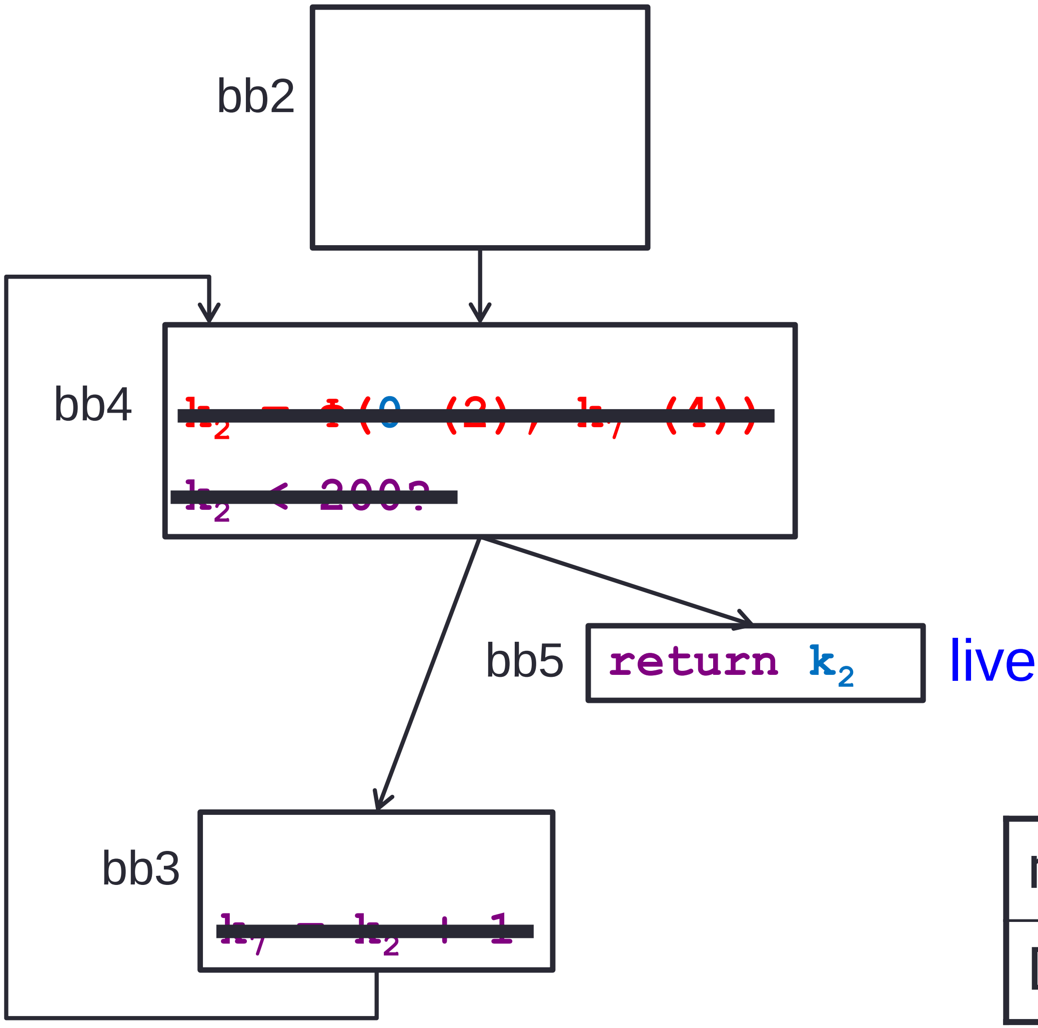
Aggressive Dead Code Elimination Example

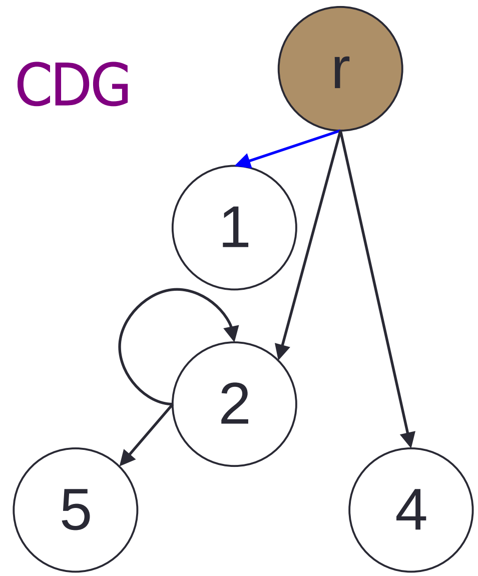




n	1	2	4	5	r
DF	{r}	{2, r}	{r}	{2}	{}

Aggressive Dead Code Elimination Example

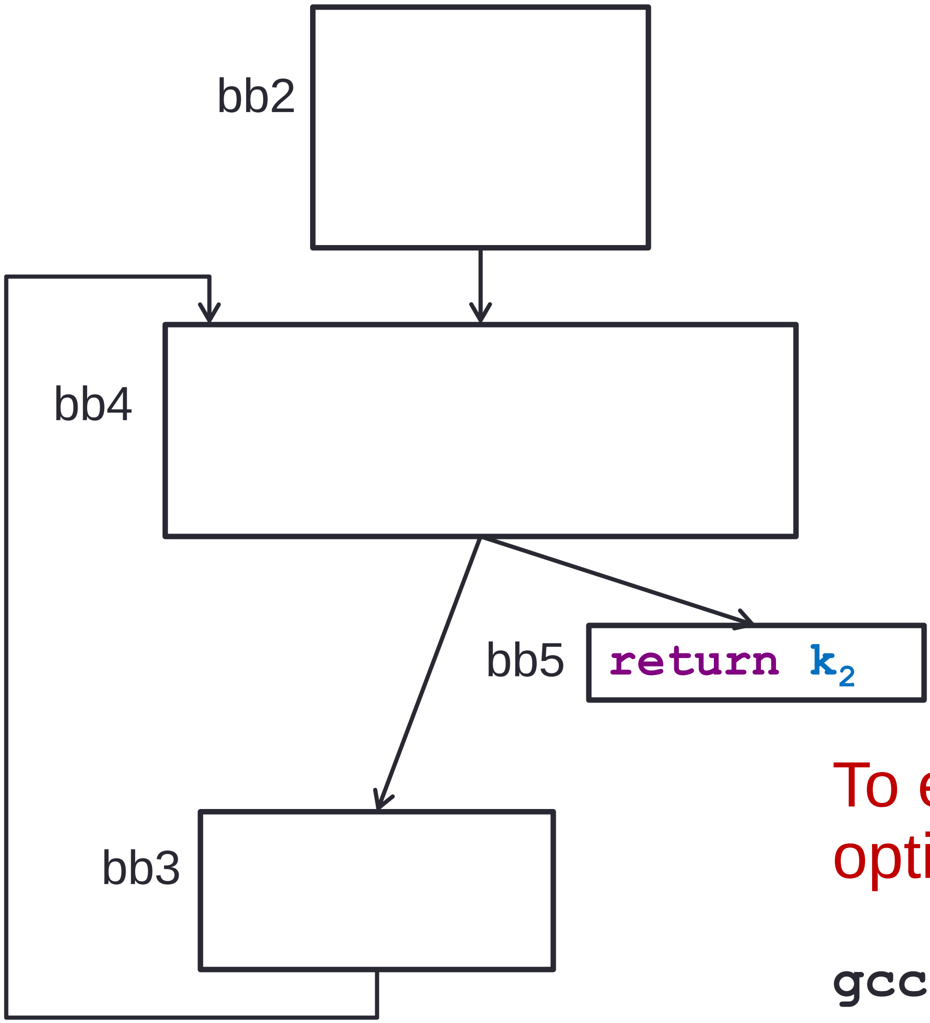





```
return 1
```

n	1	2	4	5	r
DF	{r}	{2, r}	{r}	{2}	{}

Aggressive Dead Code Elimination Example



We need aggressive optimization level

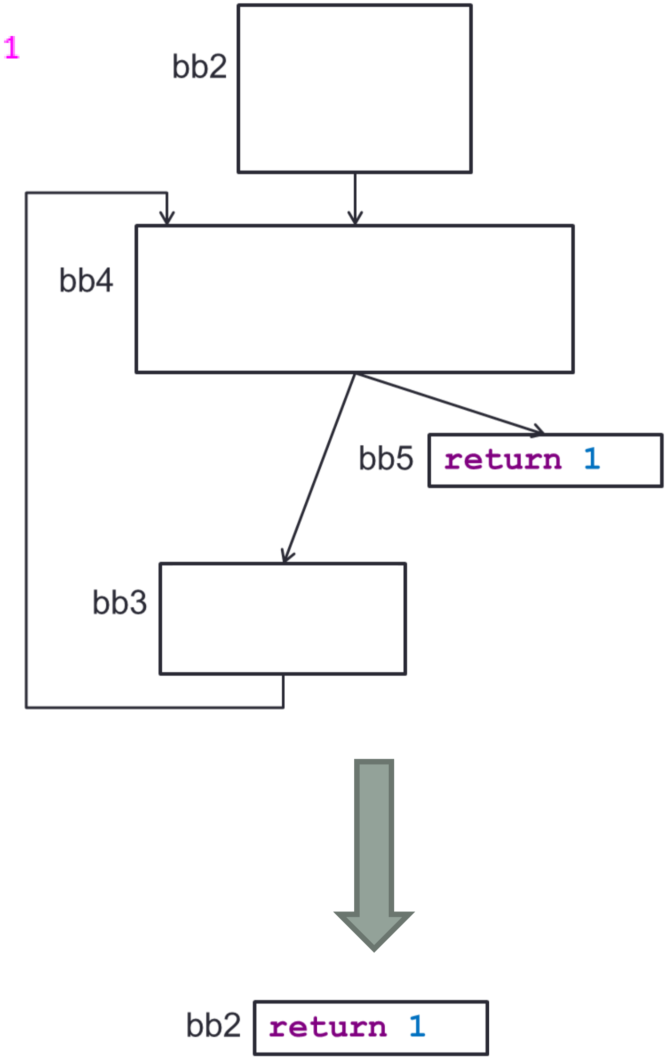
To enable this in GCC we need optimization pass **tree_ssa_cd_dce** (*)

```
gcc -c -O2 -fdump-tree-cddce func.c
```

(*) Source code is in <HERO_SDK>/hero-gcc-toolchain/src/riscv-gcc/gcc/tree-ssa-dce.c

Dead Code Elimination with Control Dependence *(func.c.037t.cddce1)*

```
;; Function helloworld (helloworld, funcdef_no=38, decl_uid=3152,
symbol_order=43)

Folding predicate j_1 <= 19 to 1
Removing basic block 5
Removing basic block 3
helloworld ()
{
  int k;
  int j;
  int i;

  <bb 2>:
  goto <bb 4>;

  <bb 3>:
  j_7 = j_2 + 1;

  <bb 4>:
  # j_2 = PHI <0(2), j_7(3)>
  if (j_2 <= 199)
    goto <bb 3>;
  else
    goto <bb 5>;

  <bb 5>:
  return 1;

}
```




```
;; Function helloworld (helloworld,
symbol_order=43)

;; 2 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2 3 4 5
;;
;; Loop 1
;;  header 4, latch 3
;;  depth 1, outer 0
;;  nodes: 4 3
;; 2 succs { 4 }
;; 3 succs { 4 }
;; 4 succs { 3 5 }
;; 5 succs { 1 }
Removing basic block 3
Merging blocks 2 and 4
Merging blocks 2 and 5
helloworld ()
{
  int k;
  int j;
  int i;

  <bb 2>:
  return 1;

}
```

gcc optimization pipeline

First step is to build the SSA form

```
1 hero-vm hero-vm    259 May 23 18:05 func.c.104t.copyprop3
1 hero-vm hero-vm    259 May 23 18:05 func.c.105t.sincos
1 hero-vm hero-vm    285 May 23 18:05 func.c.107t.crited1
1 hero-vm hero-vm    523 May 23 18:05 func.c.109t.sink
1 hero-vm hero-vm    285 May 23 18:05 func.c.112t.fix_loops
1 hero-vm hero-vm    285 May 23 18:05 func.c.113t.loop
1 hero-vm hero-vm    523 May 23 18:05 func.c.114t.loopinit
1 hero-vm hero-vm    285 May 23 18:05 func.c.115t.lim1
1 hero-vm hero-vm    285 May 23 18:05 func.c.116t.copyprop4
1 hero-vm hero-vm    285 May 23 18:05 func.c.117t.dce3
1 hero-vm hero-vm    285 May 23 18:05 func.c.119t.sccp
1 hero-vm hero-vm    285 May 23 18:05 func.c.122t.copyprop5
1 hero-vm hero-vm    399 May 23 18:05 func.c.128t.ivcanon
1 hero-vm hero-vm    399 May 23 18:05 func.c.135t.cunroll
1 hero-vm hero-vm    353 May 23 18:05 func.c.138t.ivopts
1 hero-vm hero-vm    353 May 23 18:05 func.c.139t.lim3
1 hero-vm hero-vm    350 May 23 18:05 func.c.140t.loopdone
1 hero-vm hero-vm    327 May 23 18:05 func.c.144t.veclower21
1 hero-vm hero-vm    544 May 23 18:05 func.c.146t.reassoc2
1 hero-vm hero-vm    544 May 23 18:05 func.c.147t.slsr
1 hero-vm hero-vm    588 May 23 18:05 func.c.149t.dom2
1 hero-vm hero-vm    327 May 23 18:05 func.c.152t.phicprop2
1 hero-vm hero-vm    327 May 23 18:05 func.c.153t.cddce2
1 hero-vm hero-vm    327 May 23 18:05 func.c.154t.dse2
1 hero-vm hero-vm    336 May 23 18:05 func.c.155t.forwprop4
1 hero-vm hero-vm    336 May 23 18:05 func.c.156t.phiopt3
1 hero-vm hero-vm    336 May 23 18:05 func.c.157t.fab1
1 hero-vm hero-vm    336 May 23 18:05 func.c.160t.copyrename4
1 hero-vm hero-vm    362 May 23 18:05 func.c.161t.crited2
1 hero-vm hero-vm    362 May 23 18:05 func.c.163t.uncprop1
1 hero-vm hero-vm    750 May 23 18:05 func.c.164t.local-pure-const2
1 hero-vm hero-vm    362 May 23 18:05 func.c.190t.nrv
1 hero-vm hero-vm    350 May 23 18:05 func.c.191t.optimized
1 hero-vm hero-vm    415 May 23 18:05 func.c.271t.statistics
1 hero-vm hero-vm   1232 May 23 18:05 func.o
```

Over 100 optimization passes

```
gcc -c -fdump-tree-all func.c
```

(*) Source code is in <HERO_SDK>/hero-gcc-toolchain/src/riscv-gcc/gcc/tree-ssa*.c