Intermediate representations

Advanced Compiler Construction

Michel Schinz – 2022-03-03

Intermediate representations

The term **intermediate representation** (**IR**) or **intermediate language** designates the data-structure(s) used by the compiler to represent the program being compiled.

Choosing a good IR is crucial, as many analyses and transformations (e.g. optimizations) are substantially easier to perform on some IRs than on others.

Most non-trivial compilers actually use several IRs during the compilation process, and they tend to become more low-level as the code approaches its final form.

Code example

To illustrate the differences between the various intermediate representations, we will use a program fragment to compute and print the greatest common divisor (GCD) of 2016 and 714.

The L_3 version of that fragment could be:

```
(rec loop ((x 2016)
           (y 714))
     (if (= 0 y)
         (int-print x)
         (loop y (% x y))))
```

IR #1:
CPS/L_3

CPS/L_3: a functional IR

A **functional IR** is an intermediate representation that is close to a (very) simple functional programming language.

Typical functional IRs have the following characteristics:

- all primitive operations (e.g. arithmetic operations) are performed on atomic values (variables or constants), and the result of these operations is always named,
- variables cannot be re-assigned.

As we will see later, some of these characteristics are shared with more mainstream IRs, like SSA.

CPS/L_3 is the functional IR used by the L_3 compiler.

Local continuations

A crucial notion in CPS/L_3 is that of **local continuation**.

A local continuation is similar to a (local) function but with the following restrictions:

- continuations are not "first class citizens": they cannot be stored in variables or passed as arguments – the only exception being the return continuation (described later),
- continuations never return, and must therefore be invoked in tail position.

These restrictions enable continuations to be compiled much more efficiently than normal functions. This is the only reason why continuations exist as a separate construct.

Uses of continuations

Continuations are used for two purposes in CPS/L_3:

1. To represent code blocks which can be “jumped to” from several locations, by invoking the continuation.
2. To represent the code to execute after a function call. For that purpose, every function gets a continuation as argument, which it must invoke with its return value.

CPS/L_3 grammar

T ::= (**let$_p$** ((N (P A ...))) T)
| (**let$_c$** ((N (**cnt** (N ...) T))) T)
| (**let$_f$** ((N (**fun** (N N ...) T)) ...) T)
| (**app$_c$** A A ...)
| (**app$_f$** A N A ...)
| (**if** (C A A) N N)
| (**halt** A)

N ::= name

atom

L ::= literal (e.g. integer, character, boolean or unit)

A ::= N | L

P ::= + | − | * | / | % | ...

C ::= < | <= | = | ...

CPS/L_3 local binding

(**let**$_p$ ((n (p a_1 ...))) e)

Binds the name n to the result of the application of primitive p to the value of a_1, ... in expression e.

The primitive p cannot be a logical (i.e. boolean) primitive, as such primitives are only meant to be used in conditional expressions – see later.

CPS/L_3 functions

(**let**$_f$ ((f$_1$ (**fun** (r$_1$ n$_{1,1}$...) b$_1$)) ...) e)

Binds the names f$_1$, … to functions with arguments n$_{1,1}$, … and return continuation r$_1$, … in expression e.

The functions can be mutually recursive.

The return continuation takes a single argument: the return value. Applying it is interpreted as returning from the function.

(**app**$_f$ f r a$_1$ …)

Applies the function bound to f to return continuation r and arguments a$_1$, …

The name r must either be bound by an enclosing `let`$_c$ or be the name of the return continuation of the current function.

CPS/L_3 local continuations

(**let$_c$** ((c$_1$ (**cnt** (n$_{1,1}$ …) b$_1$)) …) e)

Binds the names c$_1$,… to local continuations with arguments n$_{1,1}$, … and body b$_1$,… in expression e.

Interpretation: similar to a local function that does not return.

(**app$_c$** c a$_1$ …)

Applies the continuation bound to c to the value of a$_1$, … The name c must either be bound by an enclosing **let$_c$** or be the name of the return continuation of the current function.

Interpretation:

- if c designates a local continuation, jump to it (with arguments),
- if c designates the current return continuation, return from the current function, with the given return value.

CPS/L_3 control constructs

(**`if`** (p a_1 a_2) c_t c_f)

Tests whether the condition p is true for the value of a_1 and a_2, then applies continuation c_t if it is, or c_f if it isn't. Both c_t and c_f must be parameterless continuations.

The primitive p must be a logical primitive.

Note: `if` is a branching form of continuation application for parameterless continuations. It is therefore a conditional version of $\mathbf{app}_c$.

(**`halt`** a)

Halts program execution, exiting with the value of a, which must be an integer.

Continuation scopes

The scoping rules of CPS/L_3 are mostly the "obvious ones". The only exception is the rule for continuation variables, which are not visible in nested functions!

For example, in the following code:

```
(letc ((c0 (cnt (r) (appf print r))))
  (letf ((f (fun (c1 x)
                 (letp ((t (+ x x)))
                   (c1 t))))))
```

c_0 is *not* visible in the body of `f`!

This guarantees that continuations are truly local to the function that defines them, and can therefore be compiled efficiently.

CPS/L_3 syntactic sugar

To make CPS/L_3 programs easier to read and write, we allow the collapsing of nested occurrences of $\texttt{let}_p$ and $\texttt{let}_c$ expressions to a single `let*` expression. We also allow the elision of $\texttt{app}_f$ and $\texttt{app}_c$ in applications. Example:

```
(letₚ ((c₁ (+ 1 2)))
  (letₚ ((c₂ (+ 2 3)))
    (letₚ ((c₃ (+ c₁ c₂)))
      … (appf f c₃))))
```

⇕

```
(let* ((c₁ (+ 1 2))
       (c₂ (+ 2 3))
       (c₃ (+ c₁ c₂)))
  … (f c₃))
```

GCD in CPS/L_3

The CPS/L_3 version of the GCD program fragment looks as follows:

```
(letc ((loop
         (cnt (x y)
           (let* ((ct (cnt ()
                        (appf print x)))
                  (cf (cnt ()
                        (letp ((t (% x y)))
                          (appc loop y t)))))
             (if (= y 0) ct cf)))))
      (appc loop 2016 714))
```

(To simplify, no return continuation is passed to `print`).

Translation of CL_3 to CPS/L_3

Translating CL_3 to CPS/L_3

The translation from CL_3 to CPS/L_3 is specified as a function denoted by $[\![\cdot]\!]$ and taking two arguments:

1. T, the CL_3 term to be translated,
2. C, the translation context, a CPS/L_3 term containing a hole into which an atom containing the value of the translated term has to be plugged.

This function is written in a "mixfix" notation, as follows:

$[\![T]\!]\ C$

and must return a CPS/L_3 term that includes both:

1. the translation of the CL_3 term T, and
2. the context C with its hole plugged by an atom containing the value of (the translation of) T.

Translation context

The **translation context** is a *partial* CPS/L_3 term representing the translation of the CL_3 expression surrounding the one being translated.

It is partial in the sense that it contains a single **hole**, representing an as-yet-unknown atom.

This hole is meant to be plugged eventually by the translation function, once it knows the atom in question.

Translation context

In what follows, and in the implementation, the translation context will be represented as a meta-function with a single argument – the hole.

For example, the following meta-function represents a partial CPS/L_3 `halt` node where the argument of `halt` is not yet known:

```
λx(halt x)
```

A context being a meta-function, its hole can be plugged simply by function application at the meta-level. For exemple, assuming the above context is called C, its hole can be plugged with the atom `1` as follows:

```
C[1]
```

producing the complete CPS/L_3 term `(halt 1)`.

The translation in Scala

In Scala – the meta-language in the L_3 project – the translation function $[\![\cdot]\!]$ is defined as a function with the following profile:

```
def CL3ToCPS(t: CL3Tree,
             c: Atom ⇒ CPSTree): CPSTree
```

In the body of that function, plugging the context `c` with an atom bound to a Scala value named `a` is done using Scala function application:

```
c(a)
```

CL_3 to CPS/L_3 translation (1)

Note: in the following expressions, all underlined names are fresh.

⟦a⟧ C *where* a *is an atom (i.e. name or literal)* =

C[a]

⟦(let ((n₁ e₁) (n₂ e₂) …) e)⟧ C =

⟦e₁⟧(λv₁ (letₚ ((n₁ (id v₁)))

⟦(let ((n₂ e₂) …) e)⟧ C))

⟦(let () e)⟧ C =

⟦e⟧ C

`id` = identity primitive (returns its argument)

CL_3 to CPS/L_3 translation (2)

$[\![(\texttt{letrec}\ ((f_1\ (\texttt{fun}\ (n_{1,1}\ n_{1,2}\ldots)\ e_1))\ldots)\ e)]\!]\,C =$

$(\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (\underline{c}\ n_{1,1}\ n_{1,2}\ldots)$

$[\![e_1]\!](\lambda v\ (\texttt{app}_c\ c\ v)))\ldots)$

$[\![e]\!]\,C)$

$[\![(e\ e_1\ e_2\ldots)]\!]\,C =$

$[\![e]\!](\lambda v\,[\![e_1]\!](\lambda v_1\,[\![e_2]\!](\lambda v_2\ldots$

$(\texttt{let}_c\ ((\underline{c}\ (\texttt{cnt}\ (\underline{r})\ C[r])))$

$(\texttt{app}_f\ v\ c\ v_1\ v_2\ldots))))$

CL_3 to CPS/L_3 translation (3)

$[\![$(if (@p e_1 ...) e_2 e_3)$]\!]$ C *where* p *is a logical primitive* =

(let$_c$ ((c (cnt (r) C[r])))
(let$_c$ ((ct (cnt () $[\![e_2]\!](\lambda v_2$ (app$_c$ c v_2))))
(let$_c$ ((cf (cnt () $[\![e_3]\!](\lambda v_3$ (app$_c$ c v_3))))
$[\![e_1]\!](\lambda v_1$ ... (if (@p v_1 ...) ct cf)))))

$[\![$(if e_1 e_2 e_3)$]\!]$ C =

$[\![$(if (@= e_1 #f) e_3 e_2)$]\!]$ C

CL_3 to CPS/L_3 translation (4)

⟦`(@` p e_1 e_2 …`)`⟧ C *where* p *is a logical primitive* =

⟦`(if (@`p e_1 e_2 …`) #t #f)`⟧ C

⟦`(@` p e_1 e_2 …`)`⟧ C *where* p *is not a logical primitive* =

left as an exercise

⟦`(halt` e`)`⟧ C =

left as an exercise

Initial context

In which context should a complete program be translated?

The simplest answer is a context that halts execution with an exit code of 0 (signalling no error), that is:

```
λv (halt 0)
```

An alternative would be to do something with the value v produced by the whole program, e.g. use it as the exit code instead of 0, print it, etc.

Exercise

Translate the following L_3 expression:

```
(f 1 2)
```

in the initial context, λv `(halt 0)`.

Better translation of CL_3 to CPS/L_3

Improving the translation

The translation presented before has two shortcomings:

1. it produces terms containing useless continuations, and
2. it produces suboptimal CPS/L_3 code for some conditionals.

One solution to improve the translation is to define several different translations depending on the *source* (i.e. L_3) context in which the expression to translate appears.

Useless continuations

The first problem can be illustrated with the CL_3 term:

```
(letrec ((f (fun (g) (g)))) f)
```

which – in the empty context – gets translated to:

```
(letf ((f (fun (c g)
             (letc ((j (cnt (r) (appc c r))))
               (appf g j)))))
   f)
```

instead of the equivalent and more compact:

```
(letf ((f (fun (c g) (appf g c))))
  f)
```

Suboptimal conditionals (1)

The second problem can be illustrated with the CL_3 term:

```
(if (if a b #f) x y)
```

which, in the empty context, gets translated to:

```
(let* ((ci1 (cnt (v1) v1))
       (ct1 (cnt () (appc ci1 x)))
       (cf1 (cnt () (appc ci1 y)))
       (ci2 (cnt (v2) (if (= v2 #f) cf1 ct1)))
       (ct2 (cnt () (appc ci2 b)))
       (cf2 (cnt () (appc ci2 #f))))
  (if (= a #f) cf2 ct2))
```

Suboptimal conditionals (2)

A much better translation for:

```
(if (if a b #f) x y)
```

would be:

```
(let* ((ci1 (cnt (v1) v1))
       (ct1 (cnt () (appc ci1 x)))
       (cf1 (cnt () (appc ci1 y)))
       (ca1 (cnt () (if (= b #f) cf1 ct1))))
  (if (= a #f) cf1 ca1))
```

which immediately applies continuation `cf1` if a is false.

Source contexts

These two problems have in common the fact that the translation could be better if it depended on the source context in which the expression to translate appears.

- In the first example, the function call could be translated more efficiently because it appears as the last expression of the function (i.e. it is in **tail position**).
- For the second example, the nested if expression could be translated more efficiently because it appears in the condition of another if expression and one of its branches is a simple boolean literal (here `#f`).

Therefore, instead of having one translation function, we should have several: one per source context worth considering!

A better translation

To solve the two problems, we split the single translation function into three separate ones:

1. $[\![\cdot]\!]_N\ C$, taking as before a term to translate and a context C, whose hole must be plugged with an atom containing the value of the term.
2. $[\![\cdot]\!]_T\ c$, taking a term to translate and the name of a one-parameter continuation c. This continuation is to be applied to the value of the term.
3. $[\![\cdot]\!]_C\ c_t\ c_f$, taking a term to translate and the names of two *parameterless* continuations, c_t and c_f. The continuation c_t is to be applied when the term evaluates to a true value, while the continuation c_f is to be applied when it evaluates to a false value.

The *non-tail* translation

$[\![\cdot]\!]_N$ is called the **non-tail** translation as it is used in non-tail contexts. That is, when the work that has to be done once the term is evaluated is more complex than simply applying a continuation to the term's value.

The *non-tail* translation

For example, the arguments of a primitive are always in a non-tail context, since once they are evaluated, the primitive has to be applied on their value:

$[\![(@\ p\ \ e_1\ \ e_2\ \ldots)]\!]_N\ C$ *where* p *is not a logical primitive* =

$[\![e_1]\!]_N(\lambda v_1\ [\![e_2]\!]_N(\lambda v_2 \ldots$

$(\text{let}_p\ ((n\ (p\ v_1\ v_2\ \ldots))))$

$C[n]))$

The *tail* translation

The **tail** translation $[\![\cdot]\!]_T$ is used whenever the context passed to the simple translation has the form $\lambda v\ (app_c\ \ c\ \ v)$. It gets as argument the name of the continuation c to which the value of expression should be applied.

The *tail* translation

For example, the previous translation of function definition:

$[\![$(letrec ((f$_1$ (fun (n$_{1,1}$ n$_{1,2}$ ...) e$_1$)) ...) e)$]\!]$ C =

  (let$_f$ ((f$_1$ (fun (c n$_{1,1}$ n$_{1,2}$ ...)
    $[\![$e$_1$$]\!]$(λv (app$_c$ c v)))) ...)
    $[\![$e$]\!]$ C)

becomes:

$[\![$(letrec ((f$_1$ (fun (n$_{1,1}$ n$_{1,2}$ ...) e$_1$)) ...) e)$]\!]_N$ C =

  (let$_f$ ((f$_1$ (fun (c n$_{1,1}$ n$_{1,2}$ ...)
    $[\![$e$_1$$]\!]_T$ c)) ...)
    $[\![$e$]\!]_N$ C)

The *cond* translation

The **cond** translation $[\![\cdot]\!]_C$ is used whenever the term to translate is a condition to be tested to decide how execution must proceed. It gets two continuations names as arguments: the first is to be applied when the condition is true, while the second is to be applied when it is false.

The *cond* translation

This translation is used to handle the condition of an `if` expression:

$[\![(\texttt{if}\ e_1\ e_2\ e_3)]\!]_N\ C =$

```
(letc ((c (cnt (r) C[r])))
  (letc ((ct (cnt () [[e2]]T c)))
    (letc ((cf (cnt () [[e3]]T c)))
      [[e1]]C ct cf))))
```

The *cond* translation

Having a separate translation for conditional expressions makes the efficient compilation of conditionals with literals in one of their branch possible:

$[\![$`(if` e_1 e_2 `#f)`$]\!]_C\ c_t\ c_f =$

`(let`$_c$ `((`$\underline{ac}$ `(cnt () `$[\![e_2]\!]_C\ c_t\ c_f$`)))`

$[\![e_1]\!]_C$ `ac` c_f`)`

$[\![$`(if` e_1 `#f #t)`$]\!]_C\ c_t\ c_f =$

$[\![e_1]\!]_C\ c_f\ c_t$

…and so on for all conditionals with at least one constant branch.

The better translation in Scala

In the compiler, the three translations are simply three mutually-recursive functions, with the following profiles:

```
def nonTail(t: CL3Tree)
           (c: Atom ⇒ CPSTree): CPSTree
def tail(t: CL3Tree,
         c: Symbol): CPSTree
def cond(t: CL3Tree,
         ct: Symbol,
         cf: Symbol): CPSTree
```

IR #2:
standard RTL/CFG

Register-transfer language

A **register-transfer language** (**RTL**) is a kind of intermediate representation in which most operations compute a function of several virtual registers (i.e. variables) and store the result in another virtual register.

For example, the instruction adding variables y and z, storing the result in x could be written x ← y + z.

Such instructions are sometimes called **quadruples**, because they typically have four components: the three variables (x, y and z here) and the operation (+ here).

RTLs are very close to assembly languages, the main difference being that the number of virtual registers is usually not bounded.

Control-flow graph

A **control-flow graph** (CFG) is a directed graph whose nodes are the individual instructions of a function, and whose edges represent control-flow. More precisely, there is an edge in the CFG from a node n_1 to a node n_2 if and only if the instruction of n_2 can be executed immediately after the instruction of n_1.

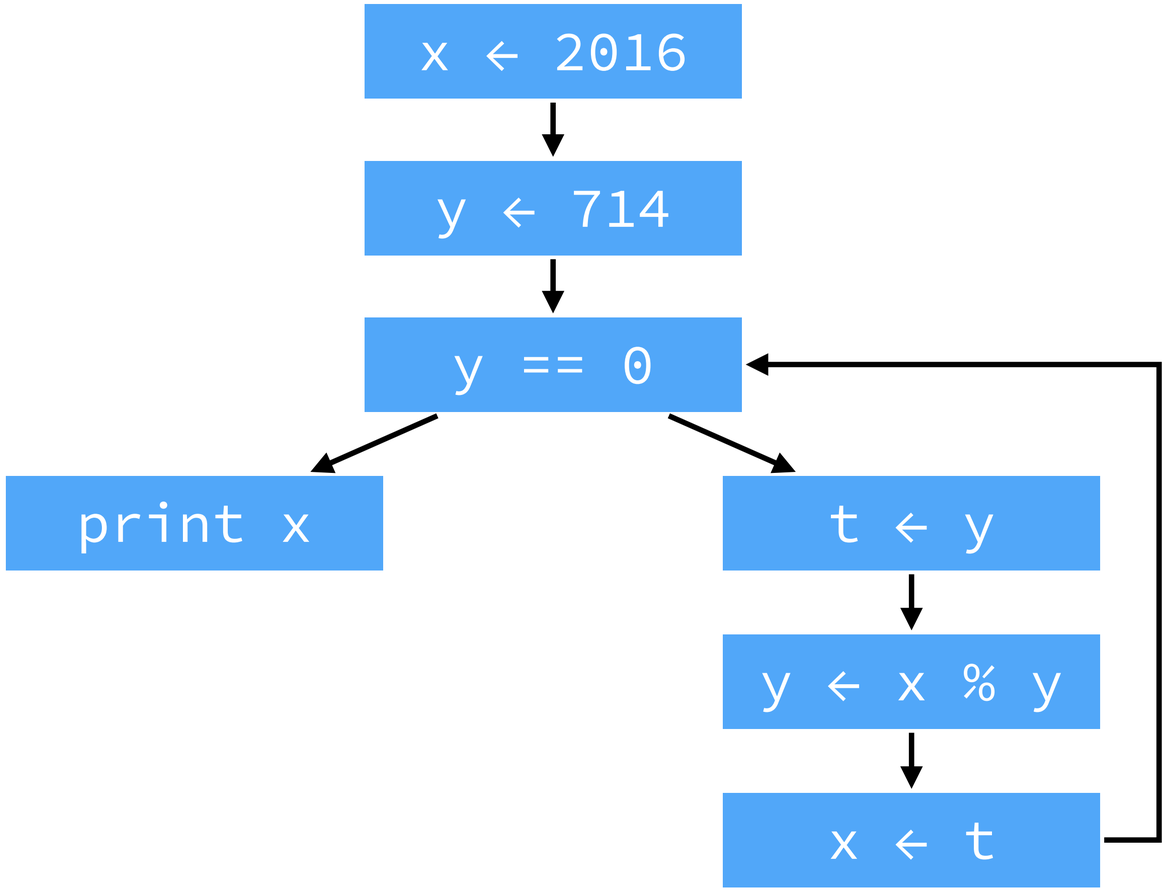
RTL/CFG

RTL/CFG is the name given to intermediate representations where each function of the program is represented as a control-flow graph whose node contain RTL instructions.

This kind of representation is very common in the later stages of compilers, especially those for imperative languages.

RTL/CFG example

Computation of the GCD of 2016 and 714 in a typical RTL/CFG representation.

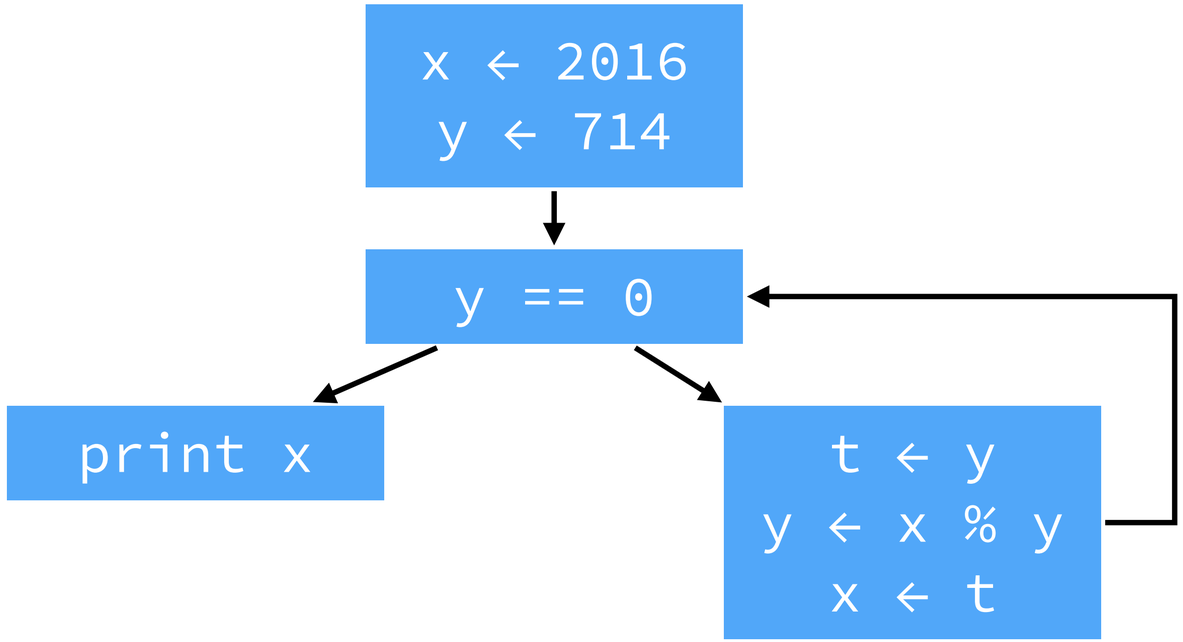


Basic blocks

A **basic block** is a maximal sequence of instruction for which control can only enter through the first instruction of the block and leave through the last. Basic blocks are sometimes used as the nodes of the CFG, instead of individual instructions. This has the advantage of reducing the number of nodes in the CFG, but also complicates data-flow analyses.

RTL/CFG example

The same examples as before, but with basic blocks instead of individual instructions.



RTL/CFG issues

One problem of RTL/CFG is that even very simple optimizations (e.g. constant propagation, common-subexpression elimination) require data-flow analyses. This is because a single variable can be assigned multiple times.

Is it possible to improve RTL/CFG so that these optimizations can be performed without prior analysis?

Yes, by using a single-assignment variant of RTL/CFG!

IR #3:
RTL/CFG in *SSA* form

SSA form

An RTL/CFG program is said to be in **static single-assignment** (**SSA**) form if each variable has only one definition in the program.

That single definition can be executed many times when the program is run – if it is inside a loop – hence the qualifier static.

SSA form is popular because it simplifies several optimizations and analysis, as we will see.

Most (imperative) programs are not naturally in SSA form, and must therefore be transformed so that they are.

Straight-line code

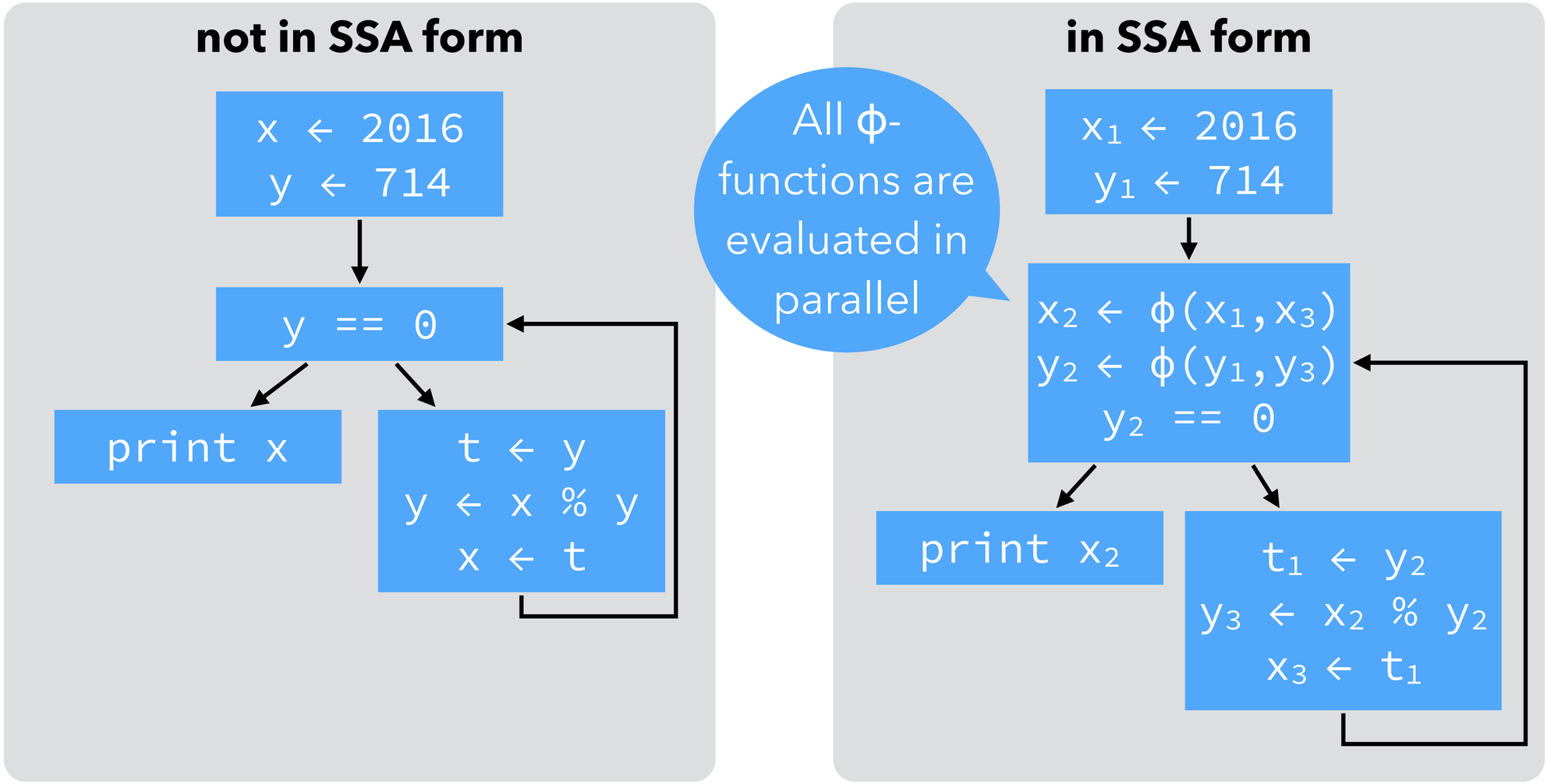
```
x ← 12
y ← 15
x ← x + y
y ← x + 4
z ← x + y
y ← y + 1
```


```
x1 ← 12
y1 ← 15
x2 ← x1 + y1
y2 ← x2 + 4
z1 ← x2 + y2
y3 ← y2 + 1
```

ϕ-functions

Join-points in the CFG — nodes with more than one predecessor — are more problematic, as each predecessor can bring its own version of a given name. To reconcile those different versions, a fictional **ϕ-function** is introduced at the join point. That function takes as argument all the versions of the variable to reconcile, and automatically selects the right one depending on the flow of control.

ϕ-functions example



Evaluation of ϕ-functions

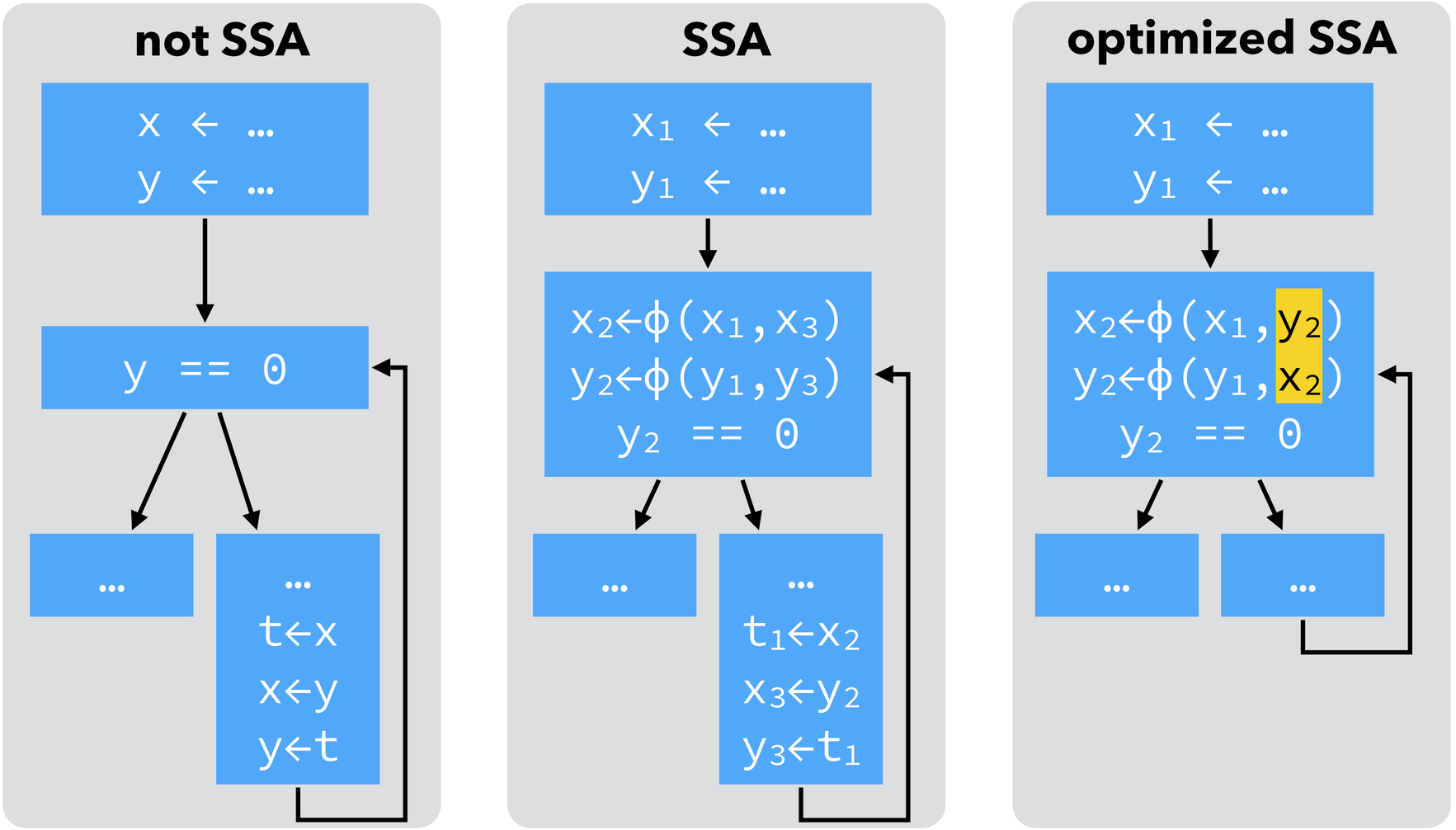
It is crucial to understand that all ϕ-functions of a block are evaluated *in parallel*, and not in sequence as the representation might suggest!
To make this clear, some authors write ϕ-functions in matrix form, with one row per predecessor:

$$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix} \quad \text{instead of} \quad \begin{aligned} x_2 &\leftarrow \phi(x_1, x_3) \\ y_2 &\leftarrow \phi(y_1, y_3) \end{aligned}$$

In the following slides, we will usually stick to the common, linear representation, but keep the parallel nature of ϕ-functions in mind.

Evaluation of ϕ-functions

The following loop illustrates why ϕ-functions must be evaluated in parallel.

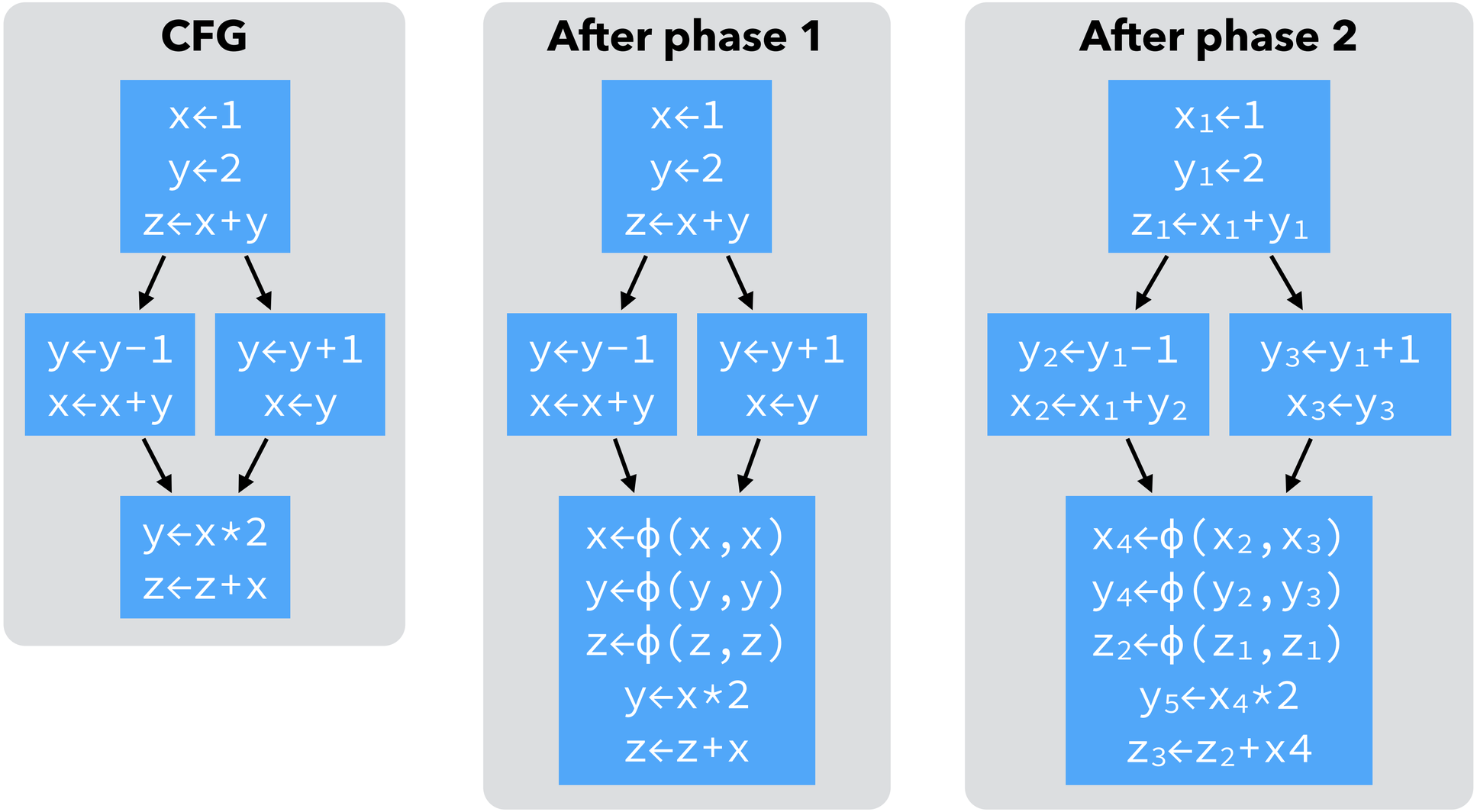


(Naïve) building of SSA form

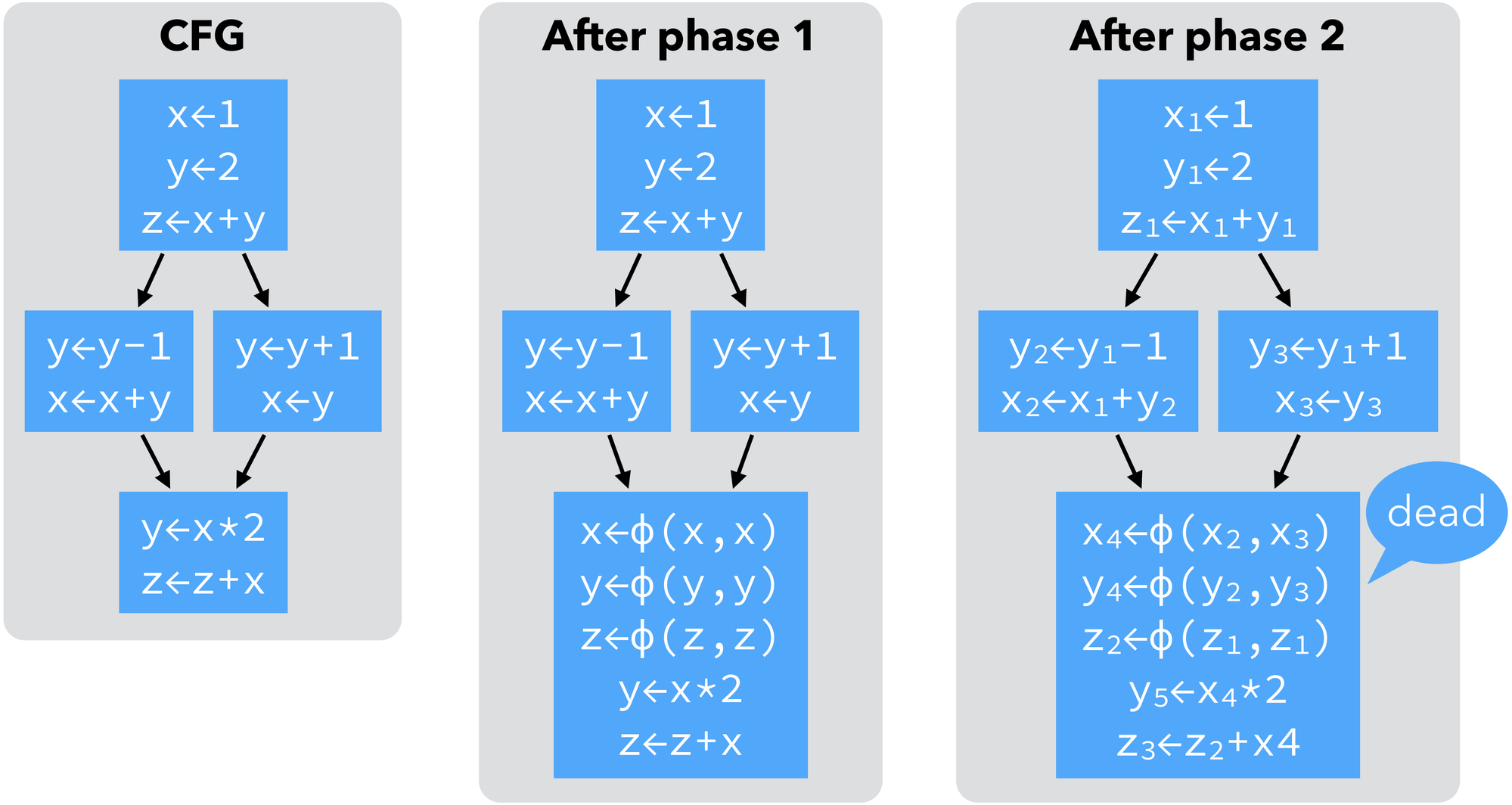
Naïve technique to build SSA form:

- for each variable x of the CFG, at each join point n, insert a ϕ-function of the form $x=\phi(x,\ldots,x)$ with as many parameters as n has predecessors,
- compute reaching definitions, and use that information to rename any use of a variable according to the – now unique – definition reaching it.

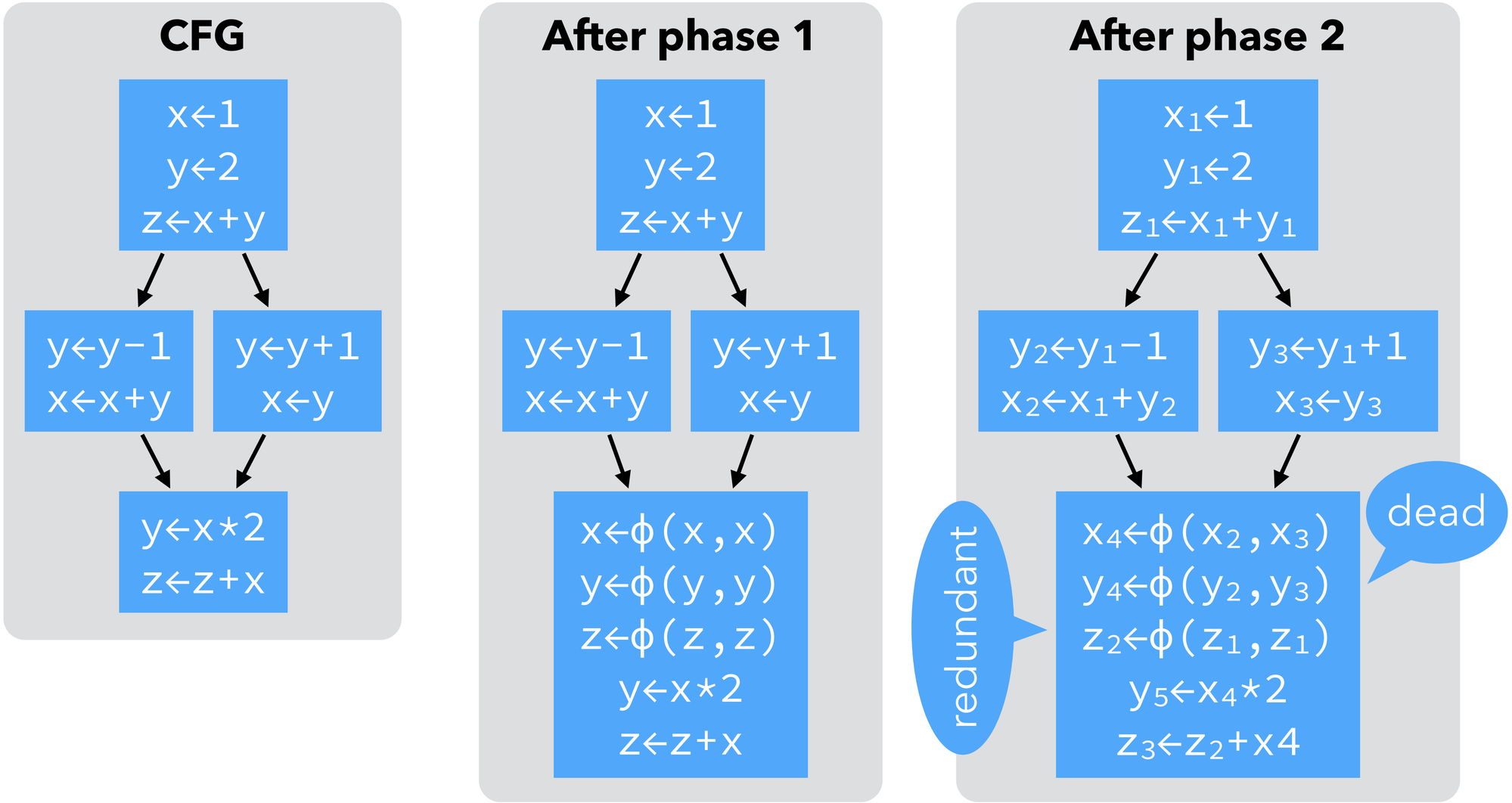
(Naïve) building of SSA form



(Naïve) building of SSA form



(Naïve) building of SSA form



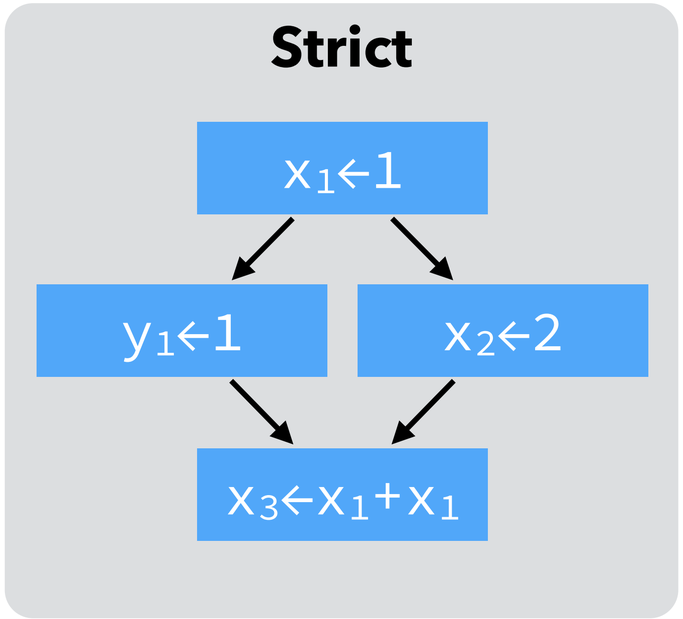
Better building techniques

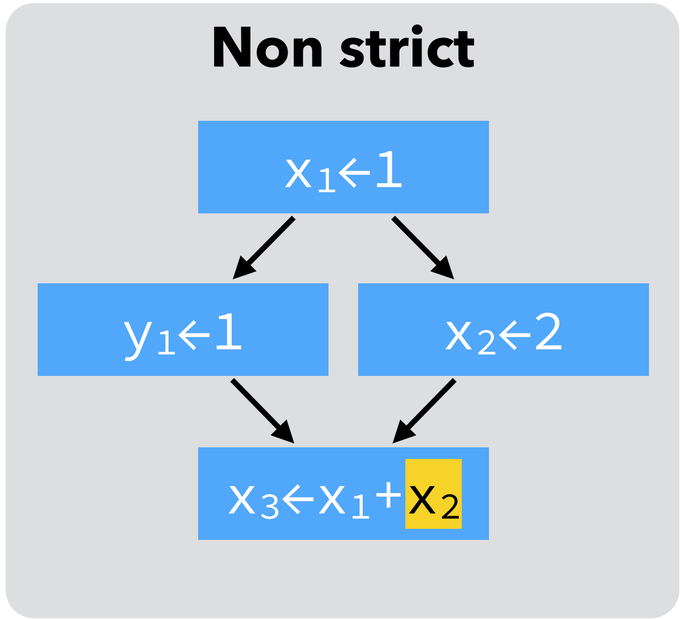
The naïve technique just presented works, in the sense that the resulting program is in SSA form and is equivalent to the original one.
However, it introduces too many ϕ-functions — some dead, some redundant — to be useful in practice. It builds the **maximal** SSA form.
Better techniques exist to translate a program to SSA form.

Strict SSA form

A program is said to be in **strict SSA form** if it is in SSA form and all uses of a variable are dominated by the definition of that variable. (In a CFG, a node n_1 **dominates** a node n_2 if all paths from the entry node to n_2 go through n_1.) Strict SSA form guarantees that no variable is used before being defined.

Strict SSA form





Comparing IRs

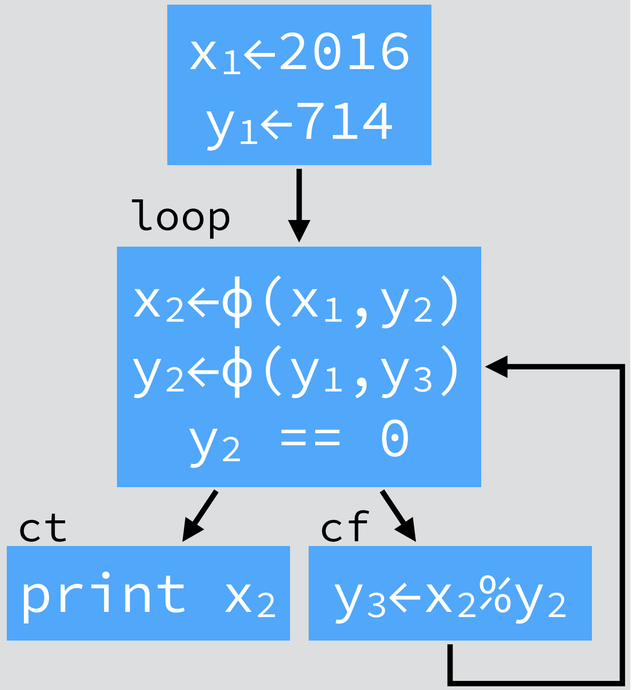
CPS/L_3 *vs* RTL/CFG in SSA

As the correspondences in the table below illustrate, CPS/L_3 is very close to RTL/CFG in SSA form.

RTL/CFG in SSA	≅	CPS/L_3
(named) basic block	≅	continuation
ϕ-function	≅	continuation argument
jump	≅	continuation invocation
strict form	≅	scoping rules

CPS/L_3 *vs* RTL/CFG in SSA

RTL/CFG in SSA form



CPS/L_3

```
(letc (
  (loop
    (cnt (x2 y2)
      (let* ((ct (cnt ()
                   (appf print x2)))
             (cf (cnt ()
                   (letp ((y3 (% x2 y2)))
                     (appc loop y2 y3)))))
        (if (= y2 0) ct cf)))))
  (let* ((x1 2016) (y1 714))
    (appc loop x1 y1)))
```

Summary and references

Claim: continuation-based, functional IRs like CPS/L_3 are SSA done right, and should replace it — or, at the very least, ϕ-functions should be replaced by continuation arguments.
(This is fortunately starting to happen, e.g. the Swift Intermediate Language has basic-blocks with arguments.)

* * *

CPS/L_3 is heavily based on the intermediate representation presented by Andrew Kennedy in *Compiling with Continuations, Continued*, in Proceedings of the International Conference on Functional Programming (ICFP) 2007.