Data-Flow Analysis as Model Checking

Anna-Lena Lamprecht, lamprecht@cs.uni-potsdam.de

Chair for Service and Software Engineering
Institute for Informatics and Computational Science
University of Potsdam, Germany

25.9.2014

Data-Flow Analysis via Model Checking

- Model Checking with CTL
- (Intraprocedural) Data-Flow Analysis as Model Checking
    - From Programs to Program Models
    - Exemplary Data-Flow Properties and their Analysis via Model Checking
- Higher-Level Applications
- Outlook: Constraint-Based Workflow Design

Model Checking

- Technique for automatic **formal verification** of finite state systems.

- The model checking process can be divided into three main tasks:

  1. **Modeling**:
     Convert a design (software or hardware) into a formalism accepted by a model-checking tool.

  2. **Specification**:
     State the properties that the design must satisfy (some logical formalism, common is modal logic).

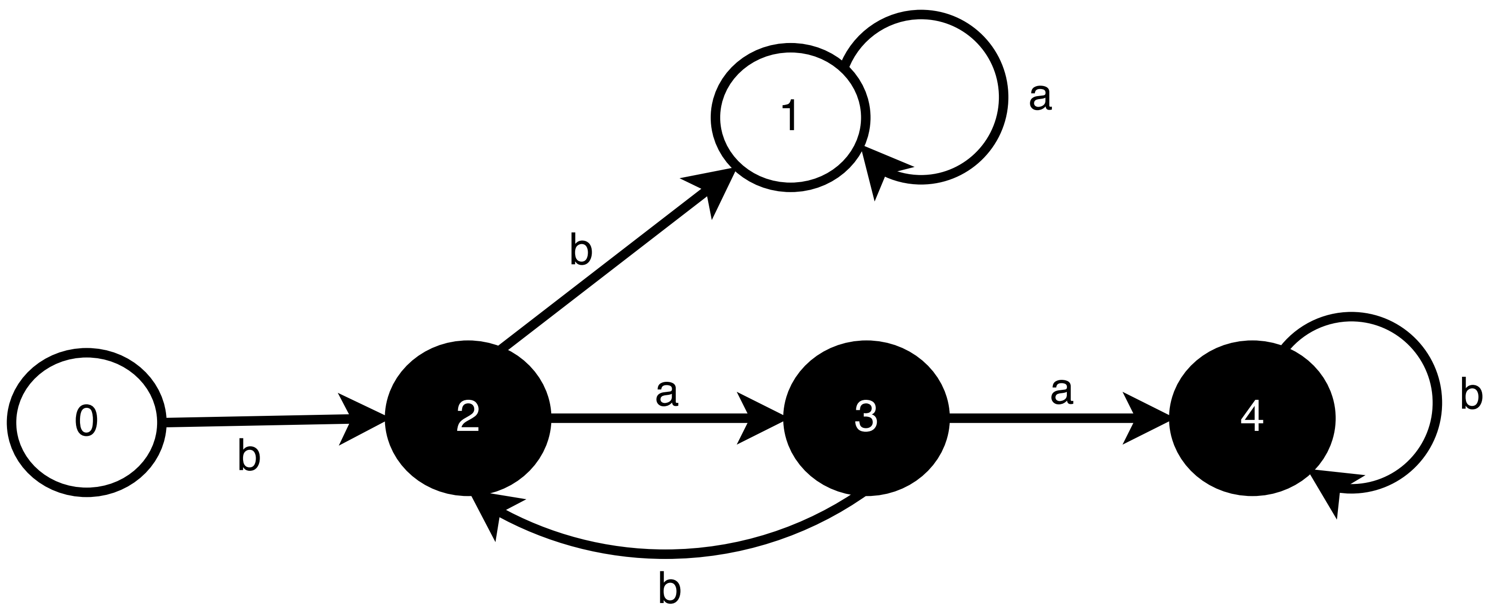
  3. **Verification**:
     Check if the model satisfies the specification (ideally completely automatic).

Modeling: Kripke Transition Systems

- A Kripke transition system (KTS) is a structure $M = (S, Act, \rightarrow, AP, I)$ where
  - S is a finite set of **states**.
  - Act is a finite set of **actions**.
  - $\rightarrow \subseteq S \times Act \times S$ is a total **transition relation**.
  - AP is a set of **atomic propositions**.
  - $I : S \rightarrow 2^{AP}$ is an **interpretation function** that labels states with subsets of AP.

Modeling: Kripke Transition Systems

- Example:



M = (*S*, *Act*, →, *AP*, *I*) with

S = {0, 1, 2, 3, 4}

Act = {a, b}

→ = {(0,b,2), (1,a,1), (2,a,3), (2,b,1), (3,b,2), (3,a,4), (4,b,4)}

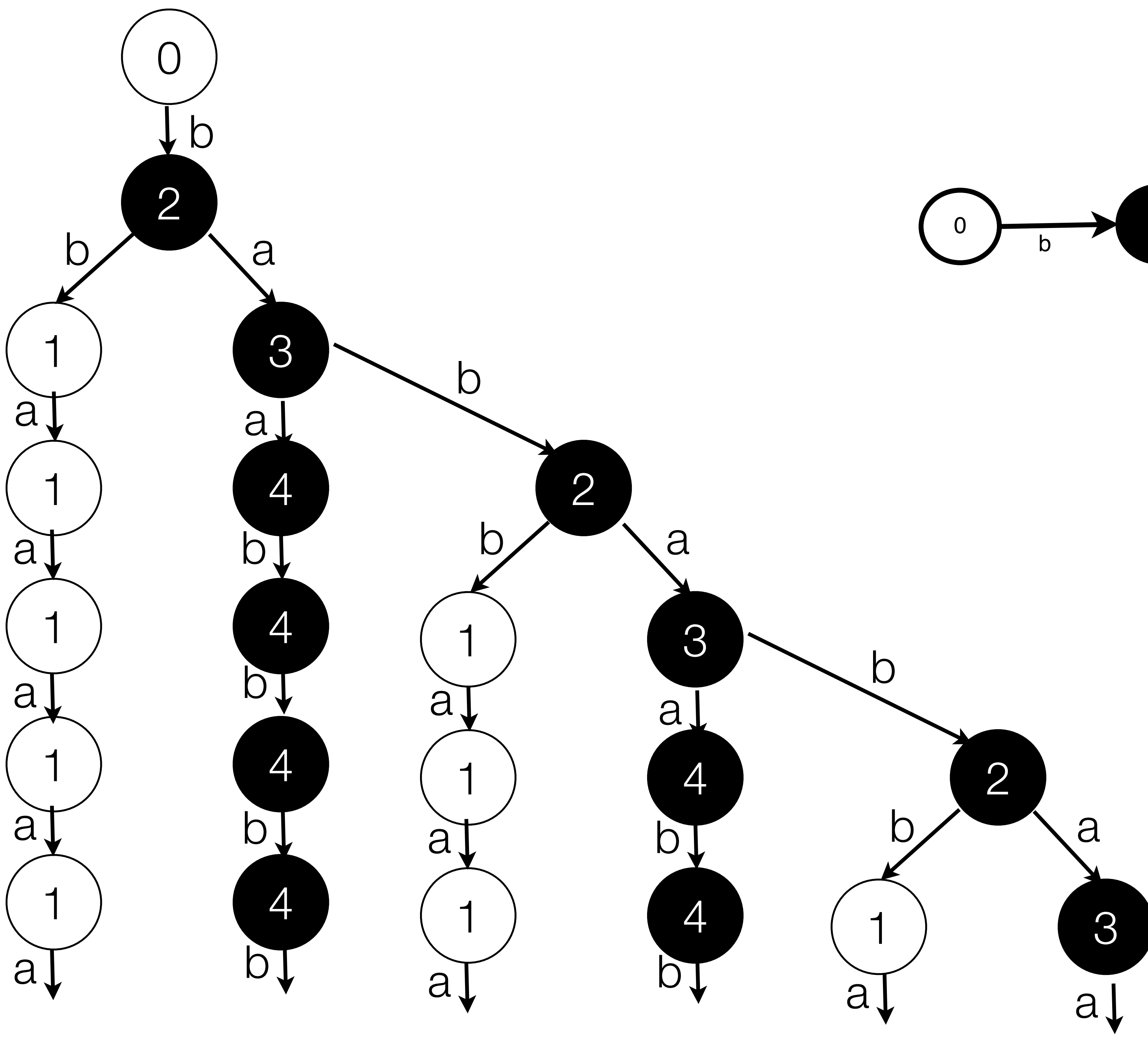
AP = {"black", "white"}

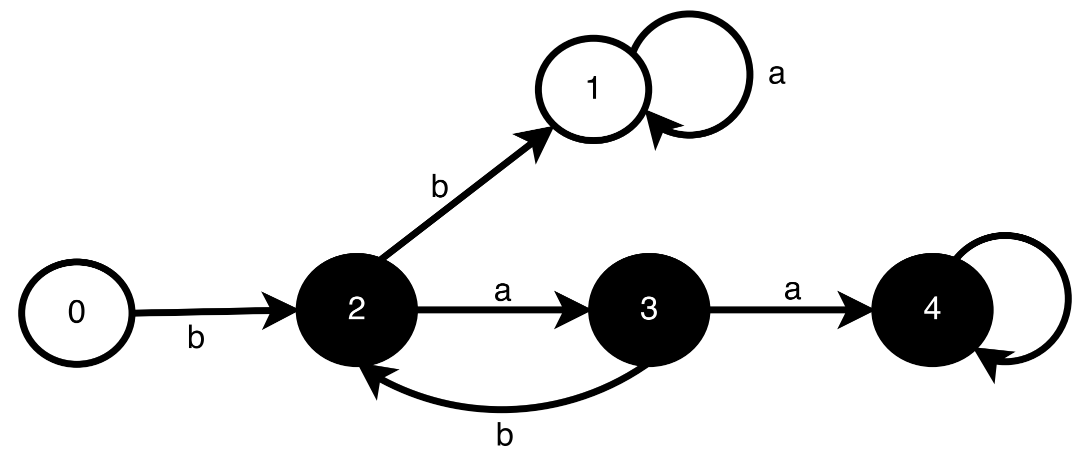
I = {I(0) = I(1) = „white“, I(2) = I(3) = I(4) = „black“}

Specification: CTL

- CTL: **Computation Tree Logic**, a high-level specification language
- A subset of the modal μ-calculus, but with easier-to-understand operators.
- Conceptually, CTL formulas describe **properties of computation trees**:
  - Designate a state in the model as **initial state**,
  - unwind the structure into an **infinite tree** with the initial state at the root,
  - then this tree show **all possible executions** of the model.

Computation Tree Example





Specification: CTL

- **CTL operators** consist of two parts:
  - The **path quantifier**, i.e. **A** („for all“) or **E** („exists“): states on which paths of the computation three the formula must hold.
  - The **state quantifier**, i.e. **X** („next“), **G** („globally“), **F** („finally“), **SU** („strong until“) or **WU** („weak until“): expresses when, on certain paths, the formula must hold.

Specification: CTL

- A CTL formula can be generated according to the following BNF:

  $$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \Phi \Rightarrow \Phi \mid$$

  $$\begin{aligned}
  &EX(\Phi) \mid EF(\Phi) \mid EG(\Phi) \mid ESU(\Phi,\Phi) \mid EWU(\Phi,\Phi) \mid \\
  &AX(\Phi) \mid AF(\Phi) \mid AG(\Phi) \mid ASU(\Phi,\Phi) \mid AWU(\Phi,\Phi) \mid \\
  &EX_{back}(\Phi) \mid EF_{back}(\Phi) \mid EG_{back}(\Phi) \mid ESU_{back}(\Phi,\Phi) \mid EWU_{back}(\Phi,\Phi) \mid \\
  &AX_{back}(\Phi) \mid AF_{back}(\Phi) \mid AG_{back}(\Phi) \mid ASU_{back}(\Phi,\Phi) \mid AWU_{back}(\Phi,\Phi)
  \end{aligned}$$

- *p* (atomic propositions) and boolean connectives as in propositional logic.
- CTL operators as described on the previous slide.
- $_{back}$ denotes backward operators.

CTL Exercise

- What do the following formulas mean?
  - EG(black)
  - EX(AG(black))
  - AWU(black, white)
- Write CTL formulas expressing:
  - "All roads lead to Rome."
  - "It is possible that is does not snow before Christmas."

Verification

- The (global) **model checking problem**:
Given a KTS $M = (S, Act, \rightarrow, AP, I)$ and a temporal formula Φ, find the set of all sates in S that satisfy Φ:

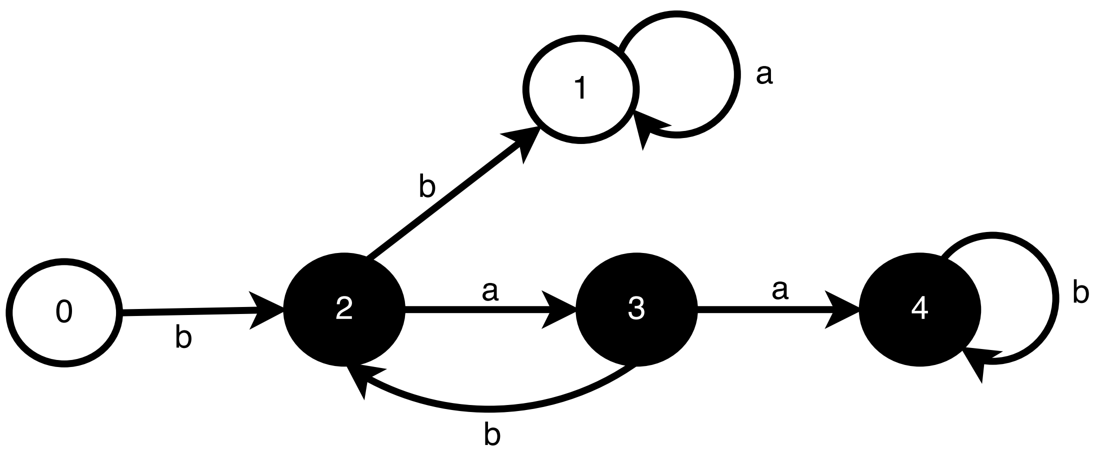
$$\{\mathbf{s} \in \mathbf{S} \mid \mathbf{s} \models \boldsymbol{\Phi}\}$$

- For verification, CTL formulas are translated into μ-calculus formulas.

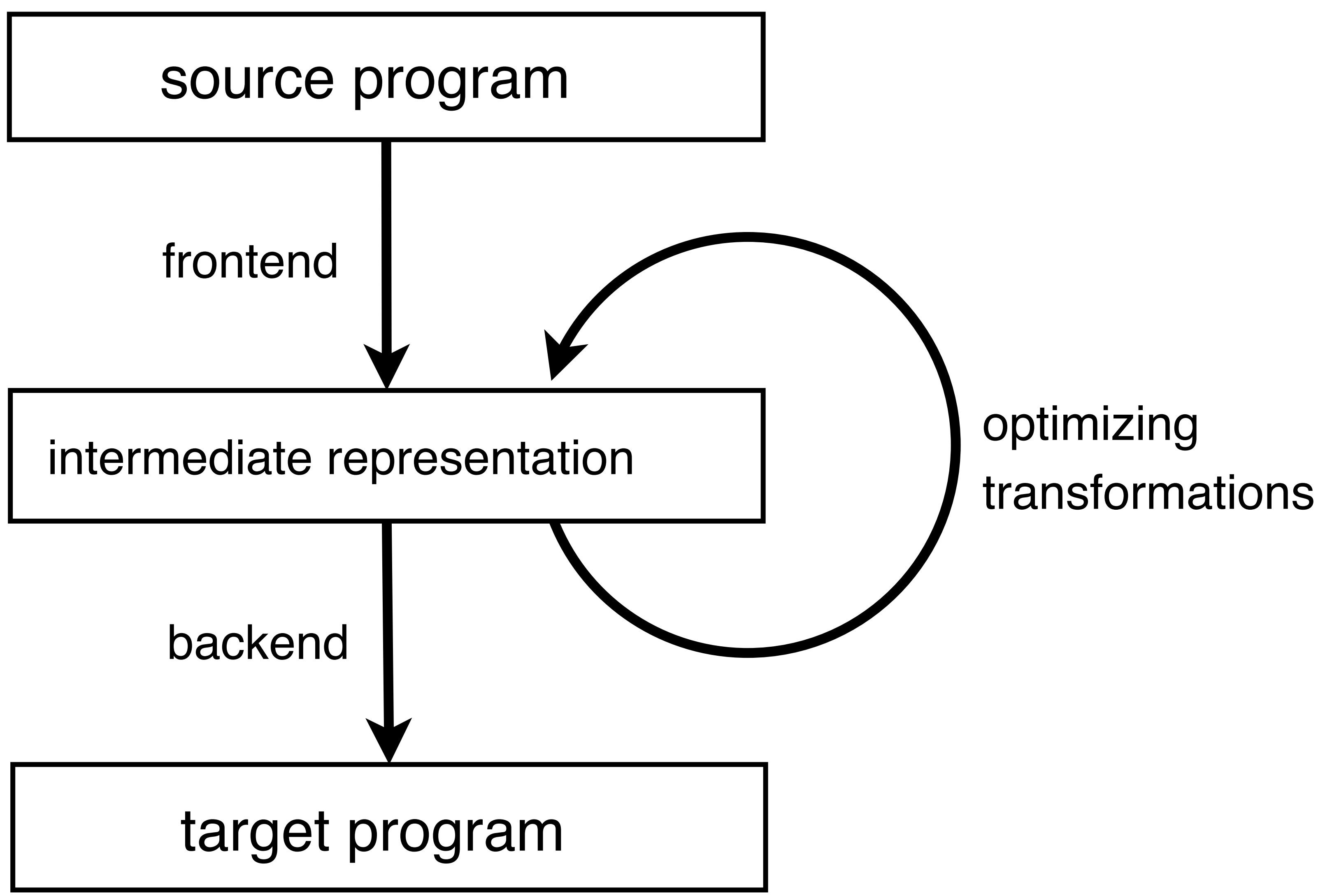
- Verification algorithm:
not discussed today, but be assured that it works. :-)

Global Model Checking Example

- Where do the following formulas hold?
  - EG(black)
  - EX(AG(black))
  - AWU(black, white)



Background: Optimizing Compilers



Data-Flow Analysis

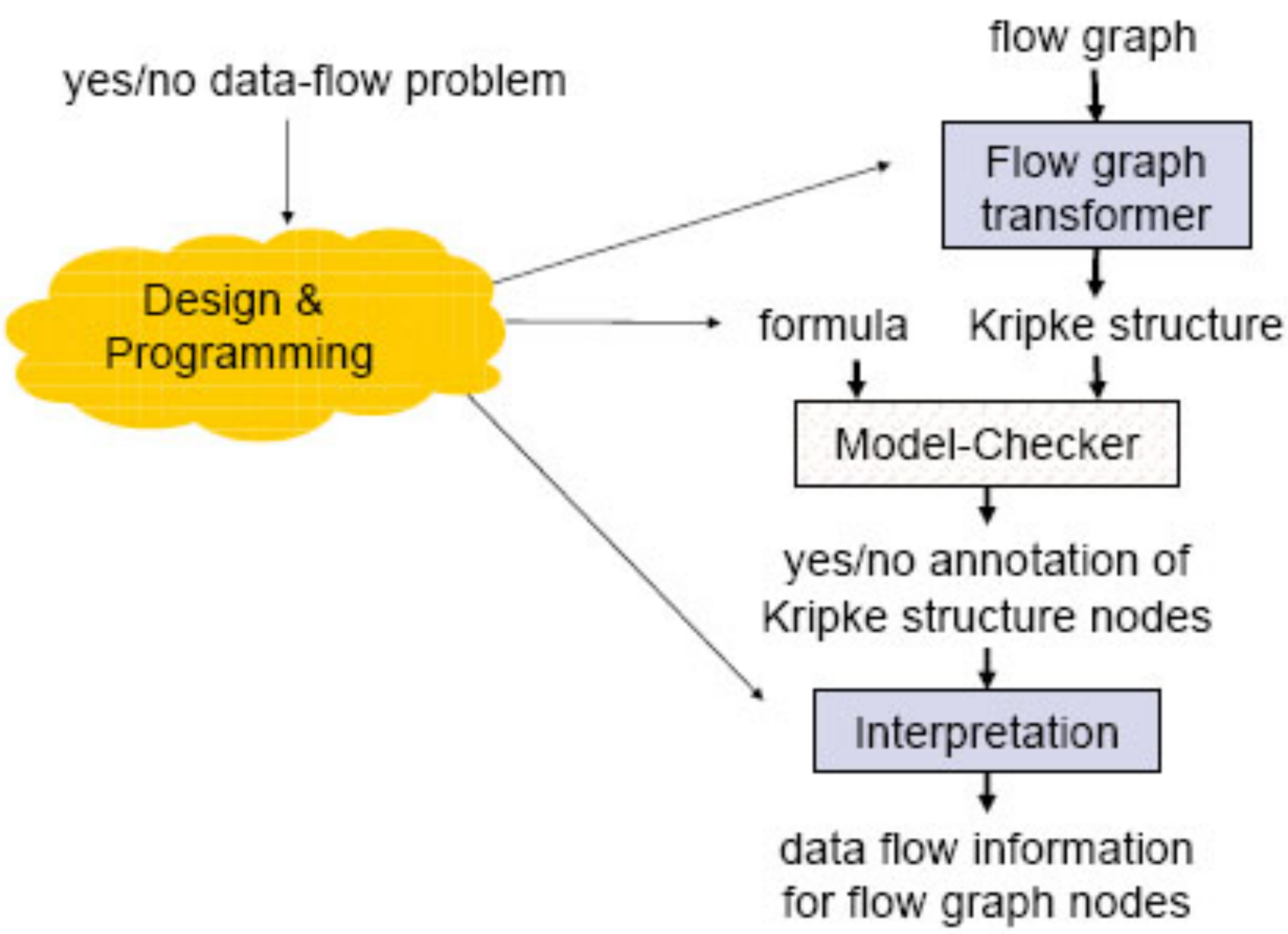
- Collection of information that is useful for or prerequisite to **code improvement** by automatic **identification of program points** enjoying particular **properties**.

- Classic approach (e.g. in optimizing compilers):
  DFA algorithm for a property : program ⇒ program points with the property

- Compare with model checking:
  model checker : modal **formulas** × model ⇒ states satisfying the formula

- Idea of DFA-MC:
  Model checkers can be seen as DFA algorithms that have the property of interest as a parameter.

Data-Flow Analysis via Model Checking



- Two transformations necessary:

  1. **Programs** have to be turned into appropriate program **models**

  2. **DFA equations** have to be turned into **modal specifications**

From Programs to Program Models

- Slight **variants of Kripke transition systems** work well for modeling **sequential imperative programs** for DFA purposes.

- A **program model** is a quadruple $P = (S, \rightarrow, AP, I)$, where

  - S is a finite set of nodes or program **states** (representing a single statement of the program), containing one start node (head) and one or more end nodes (tail).

  - $\rightarrow \subseteq S \times \{true, false, default\} \times S$ is a set of labeled transitions that defines the **control flow** of P.

  - AP is a set of **atomic propositions**.

  - $I : S \rightarrow 2^{AP}$ is an **interpretation function** that labels states with subsets of AP.

Program Model Example

- The Fibonacci numbers:

  Fib(0) = 0;

  Fib(1) = 1;

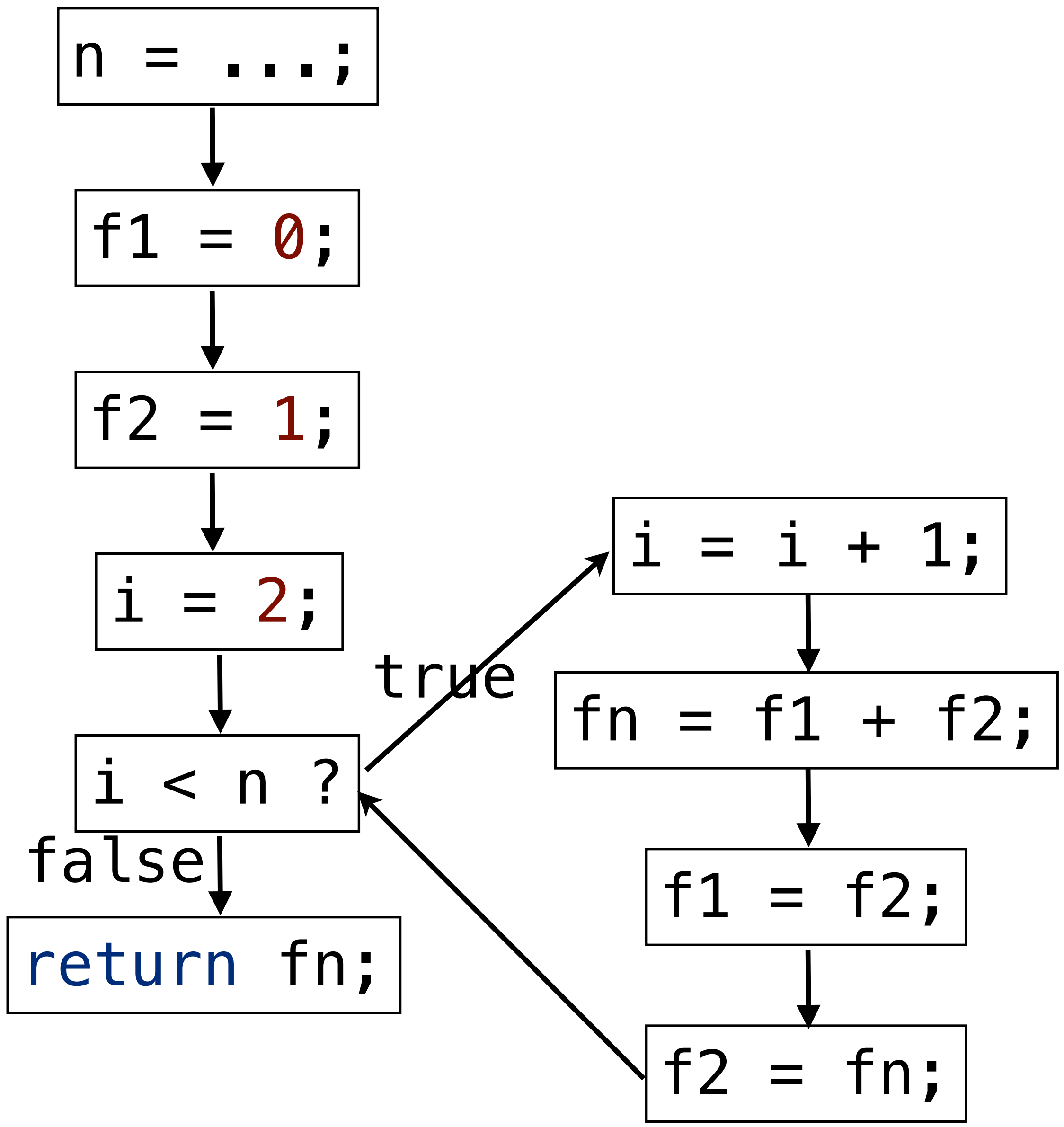
  Fib(n) = Fib(n-1) + Fib(n-2) for n>1

- A function for (iteratively) computing the n^th Fibonacci number:

```
int Fibonacci(int n)
{
    int f1 = 0;
    int f2 = 1;
    int fn;
    for (int i = 2; i < n; i++)
    {
        fn = f1 + f2;
        f1 = f2;
        f2 = fn;
    }
    return fn;
}
```

Program Model Example

- As flow graph:



Basic Properties: isDef, isUsed, isMod

- Note: We use **three-address code** here, i.e. statements consist of one operator, at most one result, and at most two arguments.

- Three basic properties can be defined on the structure of such statements:

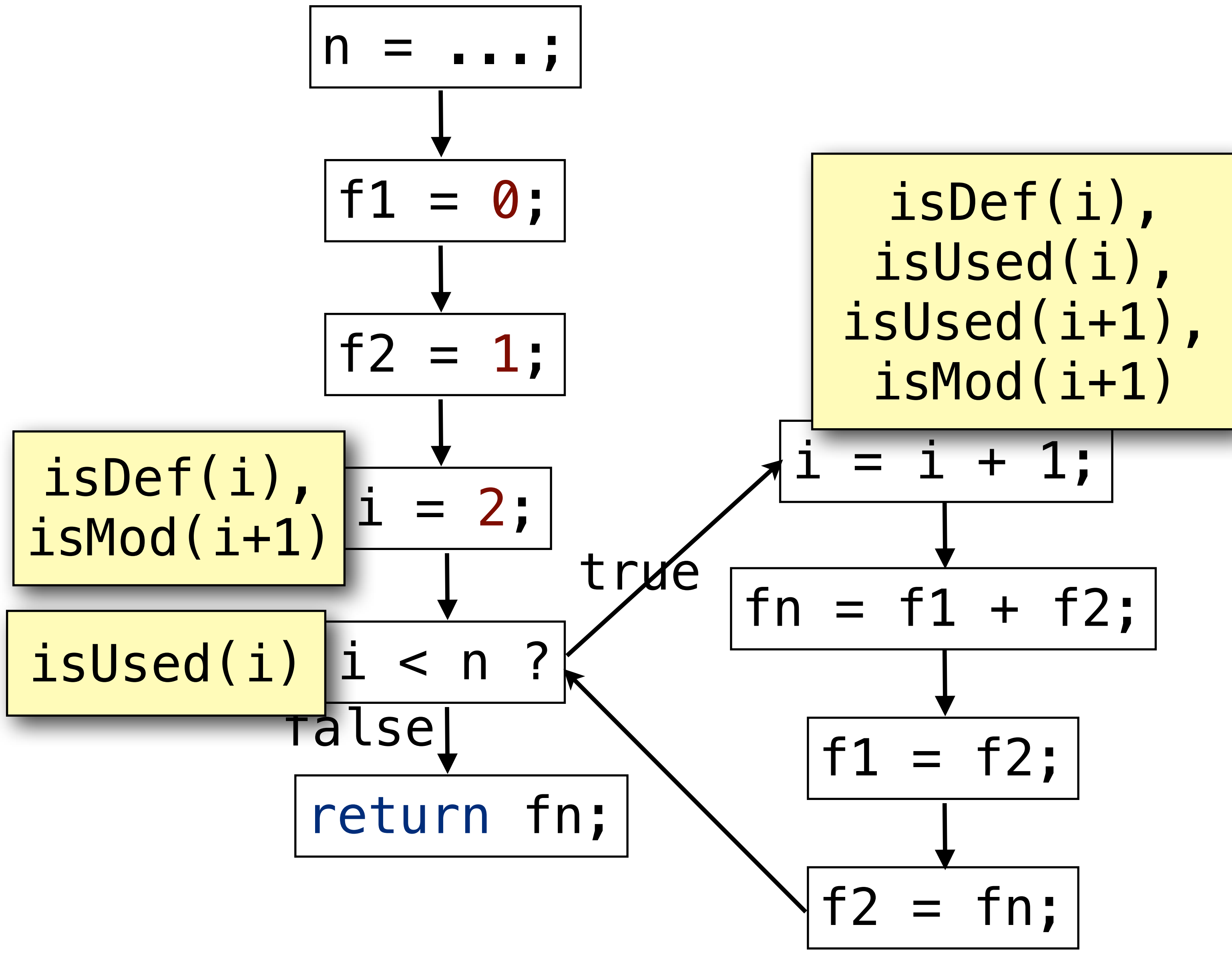
  1. **isDef**: A variable *A* is **defined** if the statement can (potentially) change the value of *A*, for instance by an assignment.

  2. **isUsed**: A variable *A* or an expression *XopY* is **used** if there is any occurrence of *A* or *XopY* as an operand.

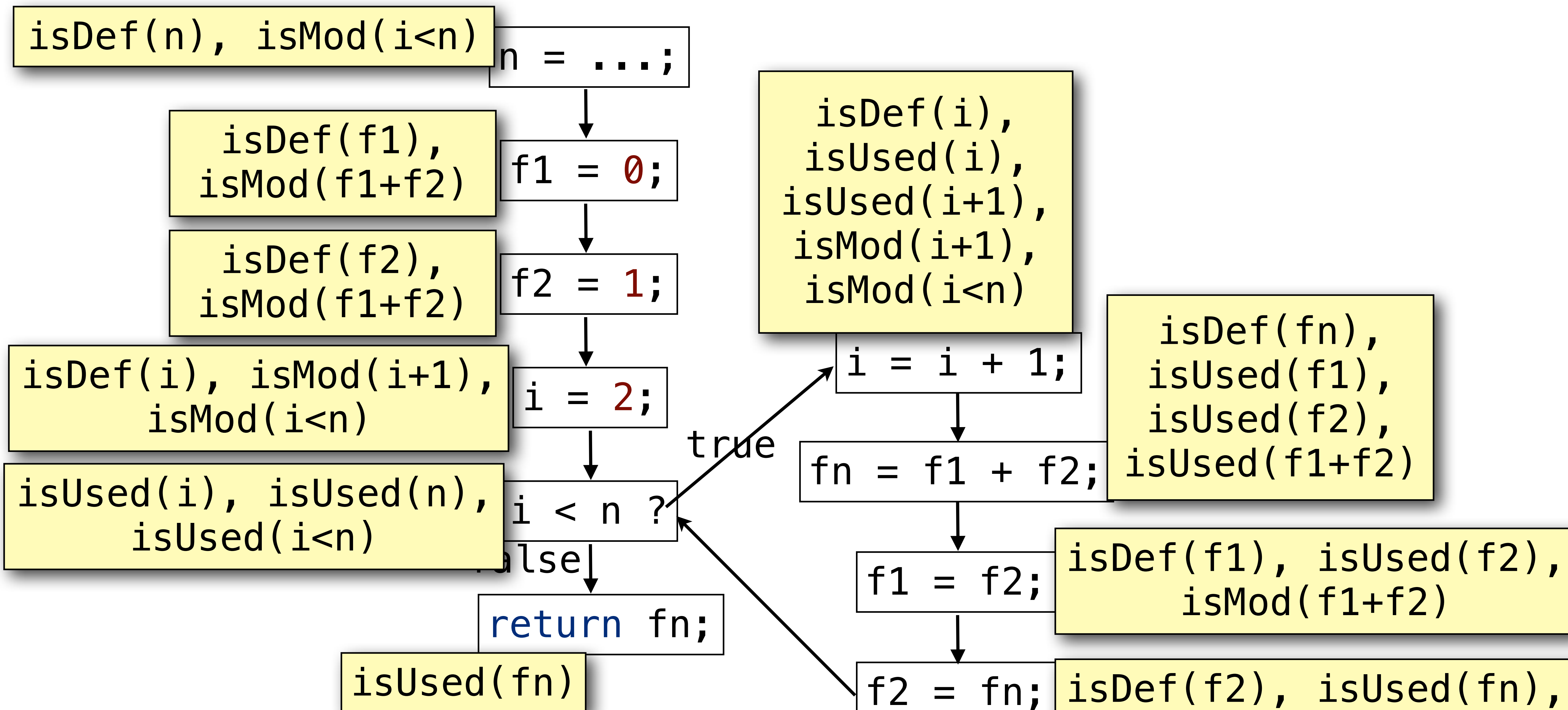
  3. **isMod**: An expression *XopY* is **modified** if the statement defines *X* or *Y*.

Program Model Example: Basic Properties

- **Variables** in the program:
  i, n, f1, f2, fn
- **Expressions** in the program:
  i+1, i<n, f1+f2
- The annotations for variable i and expression i+1 are shown on the right.
- **Exercise**:
  Complete the annotations for the remaining variables and expressions!



Program Model Example: Basic Properties



Data-Flow Analysis

- Four basic DFA problems:
  - Live Variables
  - Very Busy Expressions
  - Available Expressions
  - Reaching Definitions

Live Variables

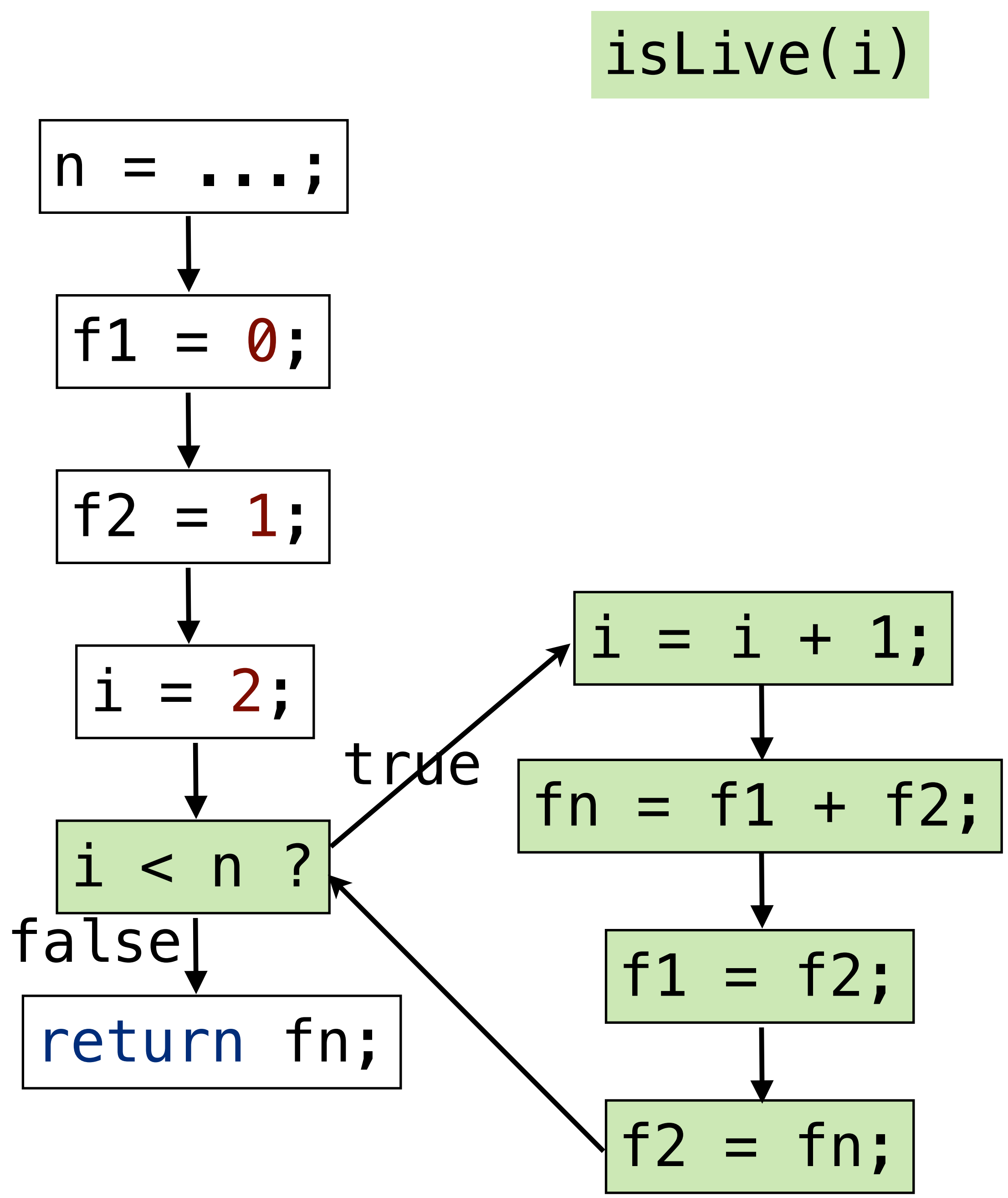
- A variable x is **live** at point p if the value of x at p is used along some path in the flow graph starting at p.

- Otherwise x is **dead** at p.

- Useful for, e.g.: Dead Assignment Elimination, register allocation.

- The following CTL formula specifies the states at which x is live:

  isLive(x) = ESU(¬isDef(x), isUsed(x))

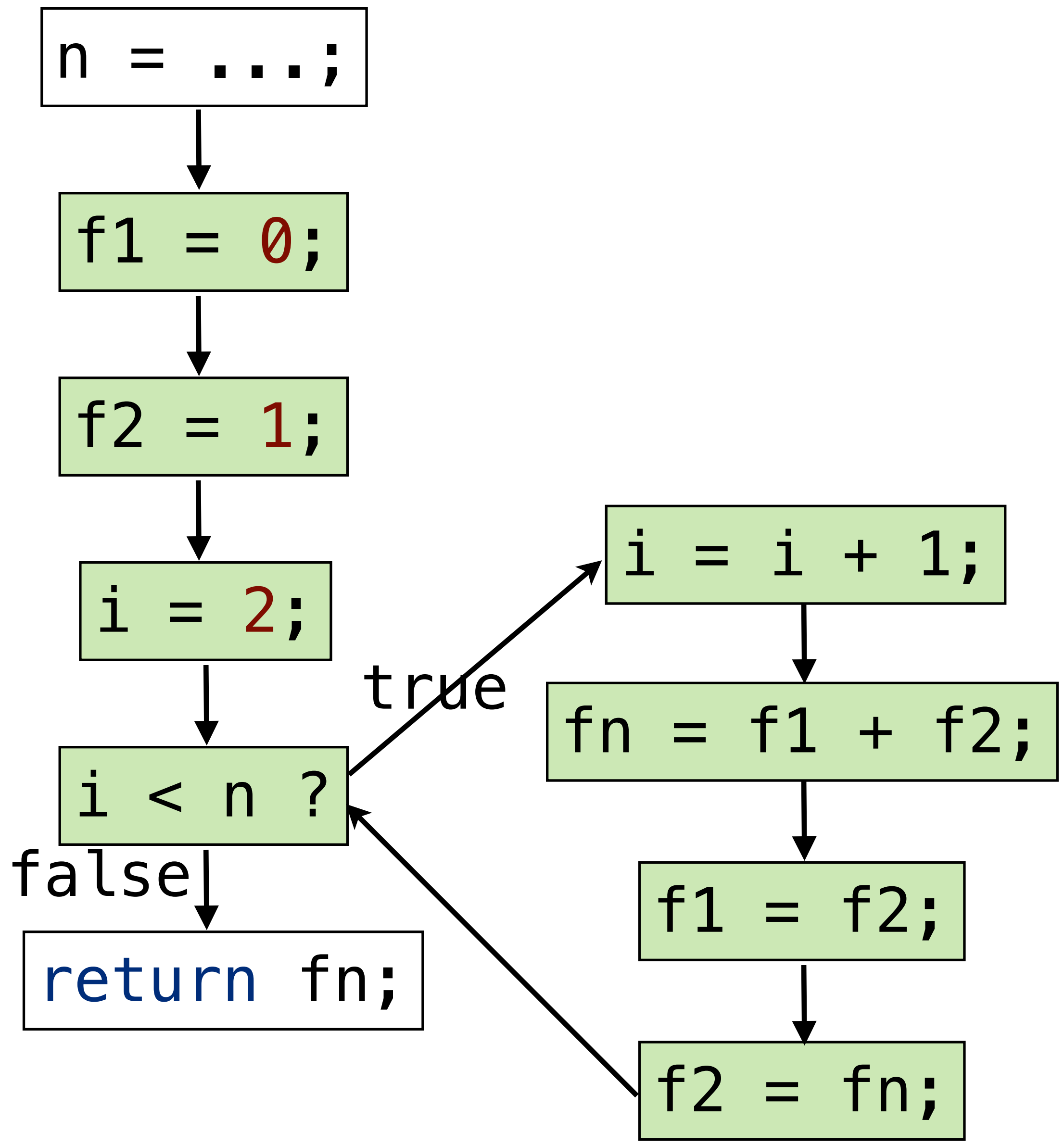


Live Variables: Exercise

- Choose one of the remaining variables (i.e. $x \in \{n, f1, f2, fn\}$) and determine the states where isLive(x) holds.

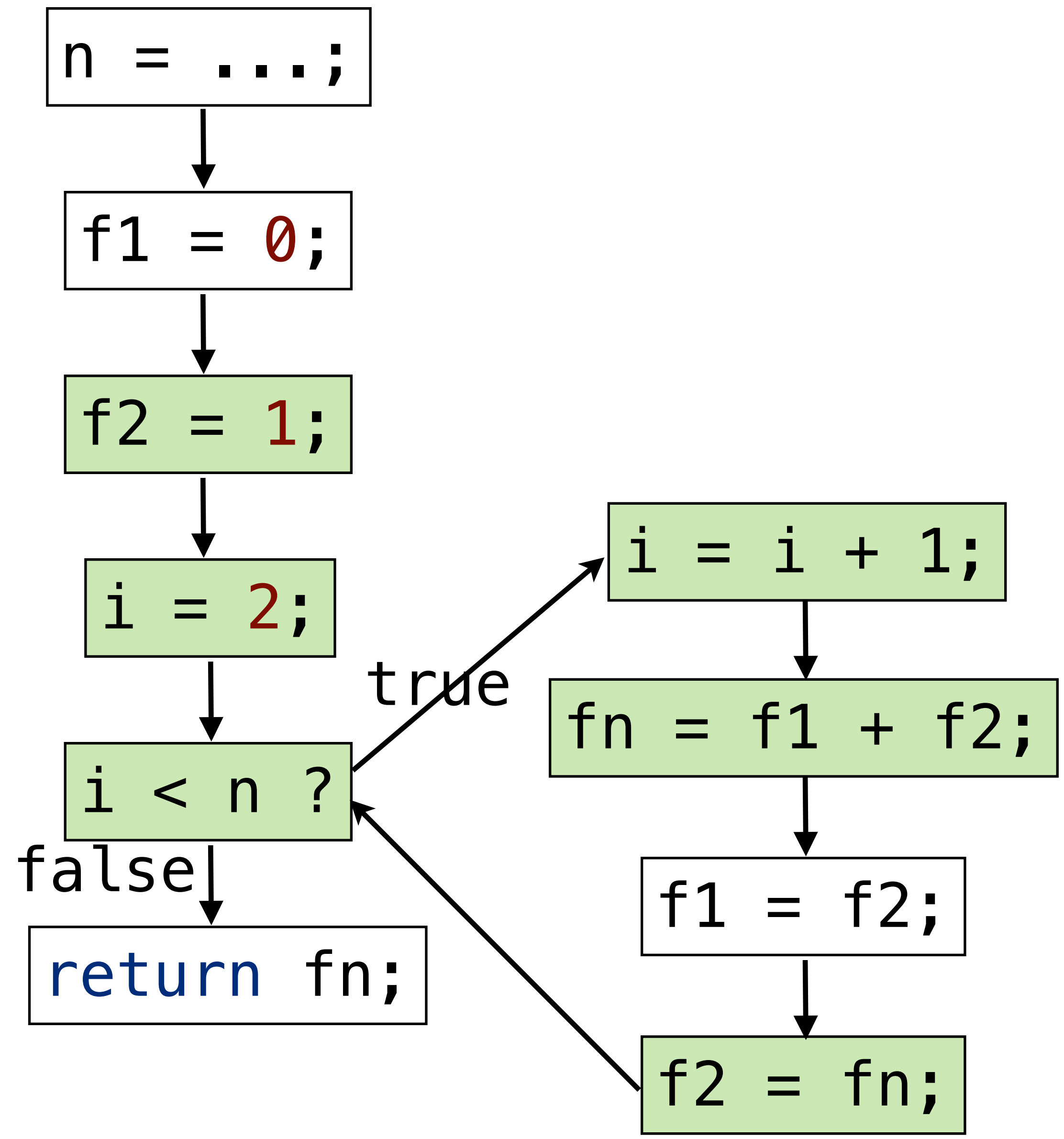
Live Variables: isLive(n)

`isLive(n)`



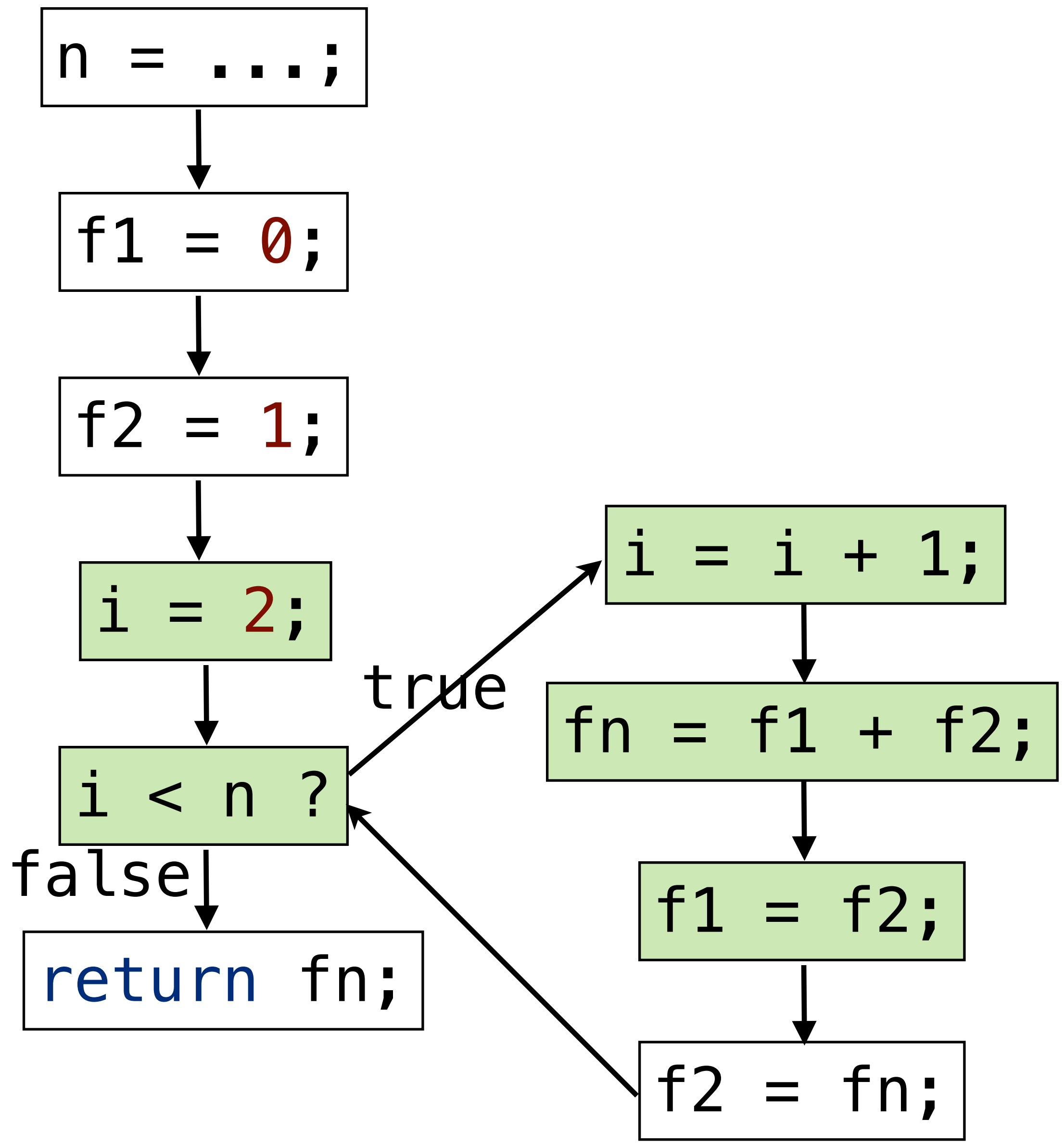
Live Variables: isLive(f1)

`isLive(f1)`



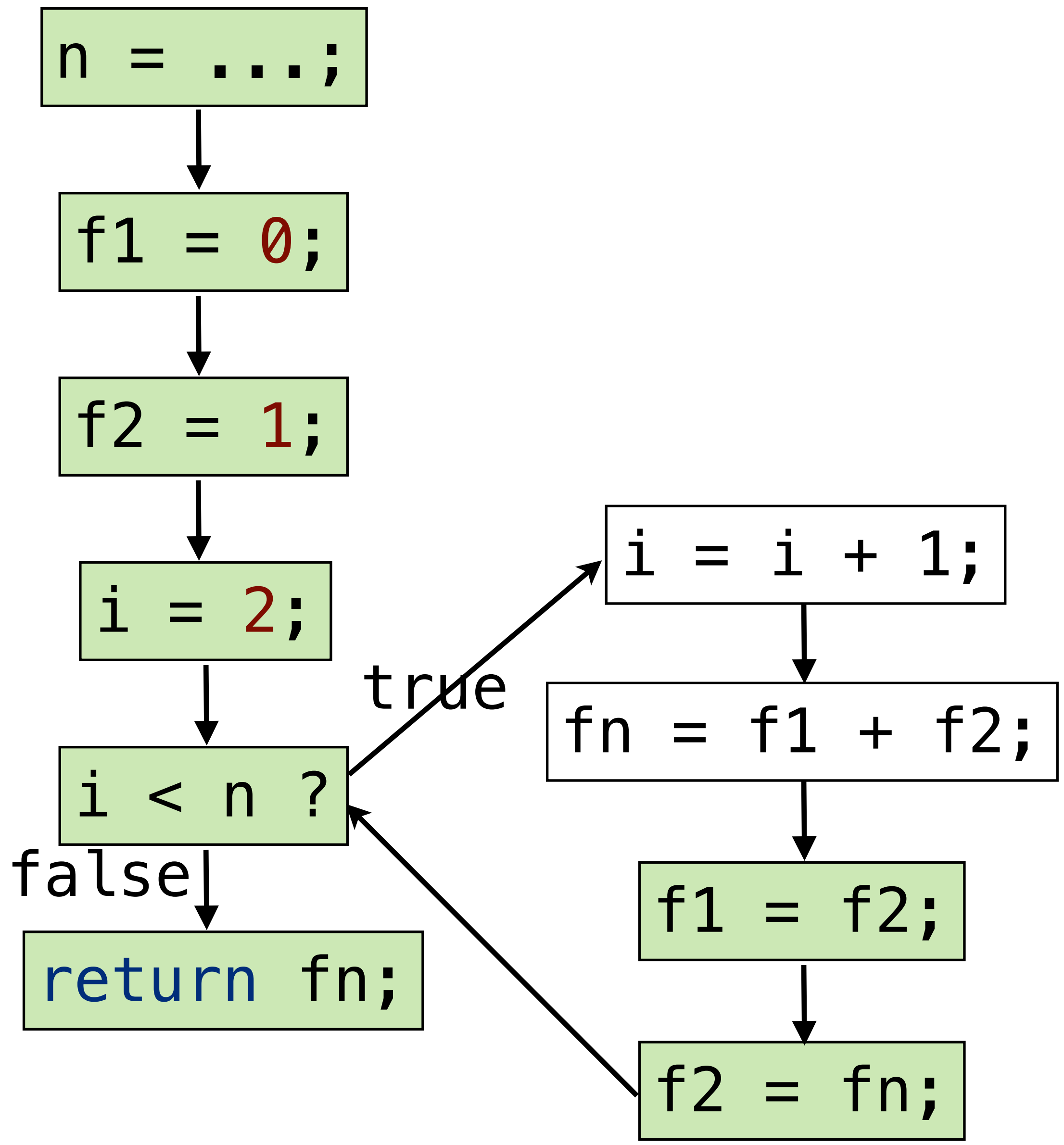
Live Variables: isLive(f2)

`isLive(f2)`



Live Variables: isLive(fn)

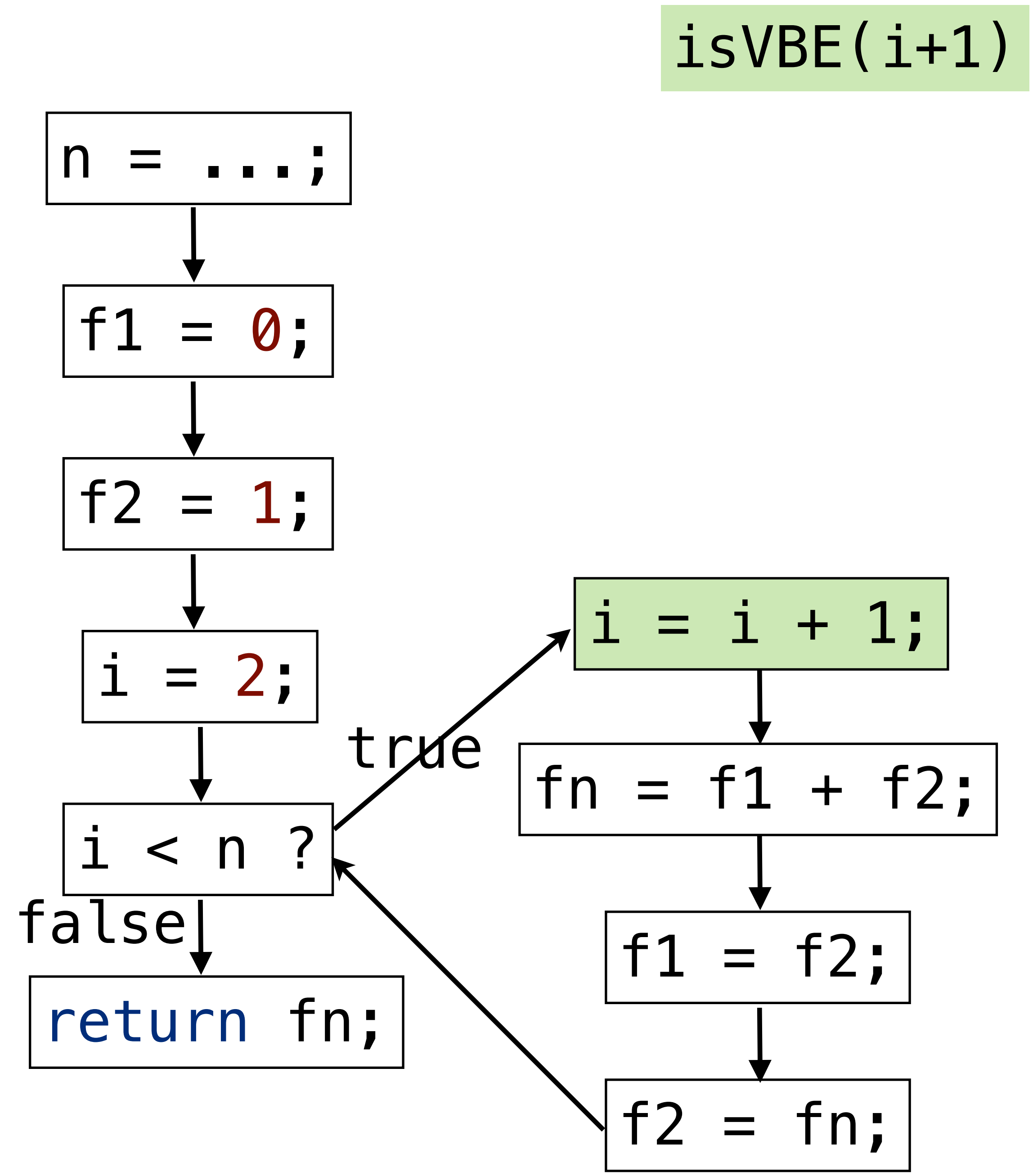
`isLive(fn)`



Very Busy Expressions

- An expression *e* = *XopY* is **very busy** at point p if along every path from p control comes to a computation of *XopY* before any definition of *X* or *Y*.
- Useful for, e.g.: expression hoisting.
- The following CTL formula specifies the states at which e is very busy:

  isVBE(e) =
  ASU(¬isMod(e), isUsed(e))

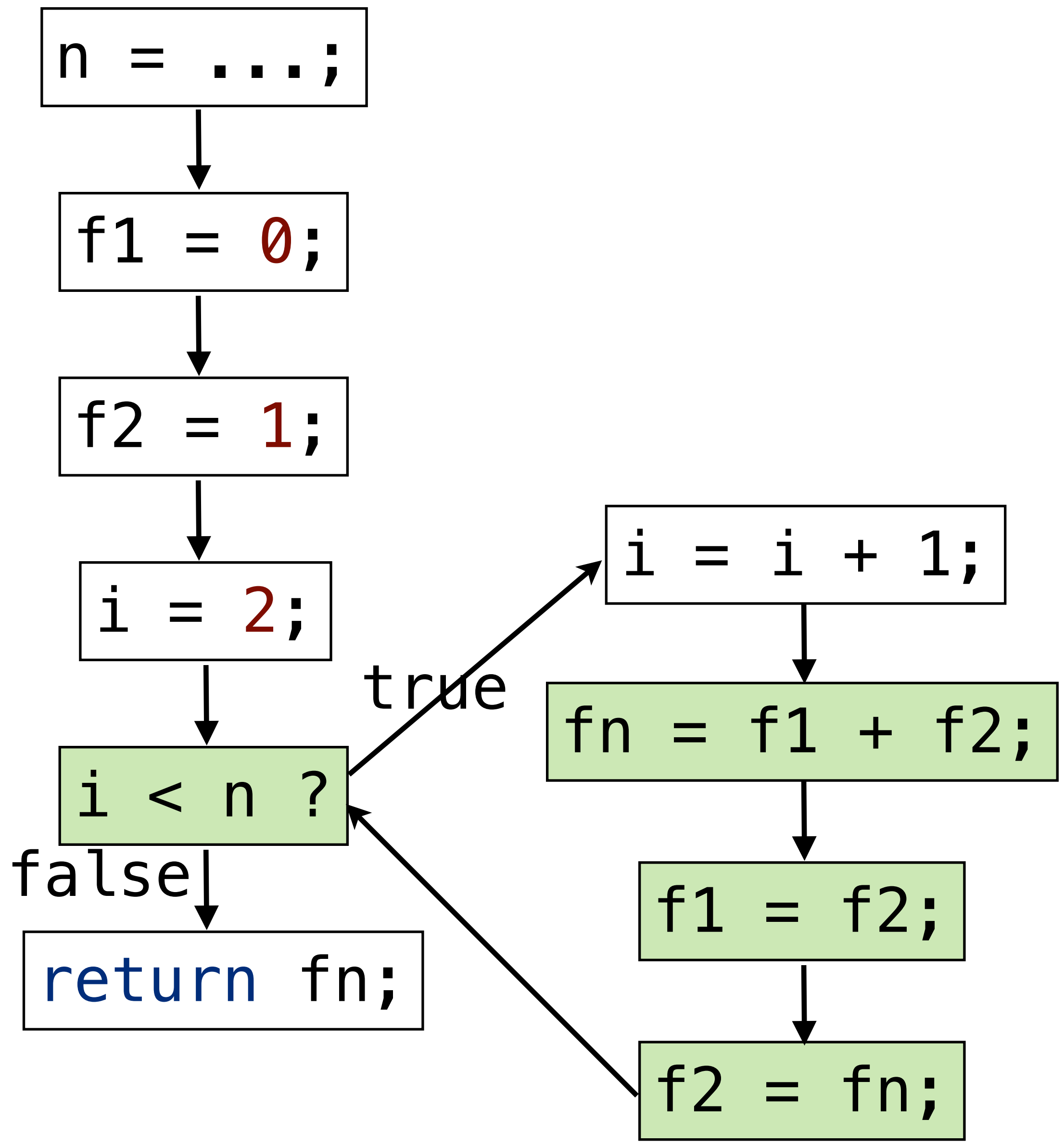


Very Busy Expressions: Exercise

- Choose one of the remaining expressions (i.e. e $\in$ {i<n, f1+f2}) and determine the states where isVBE(e) holds.

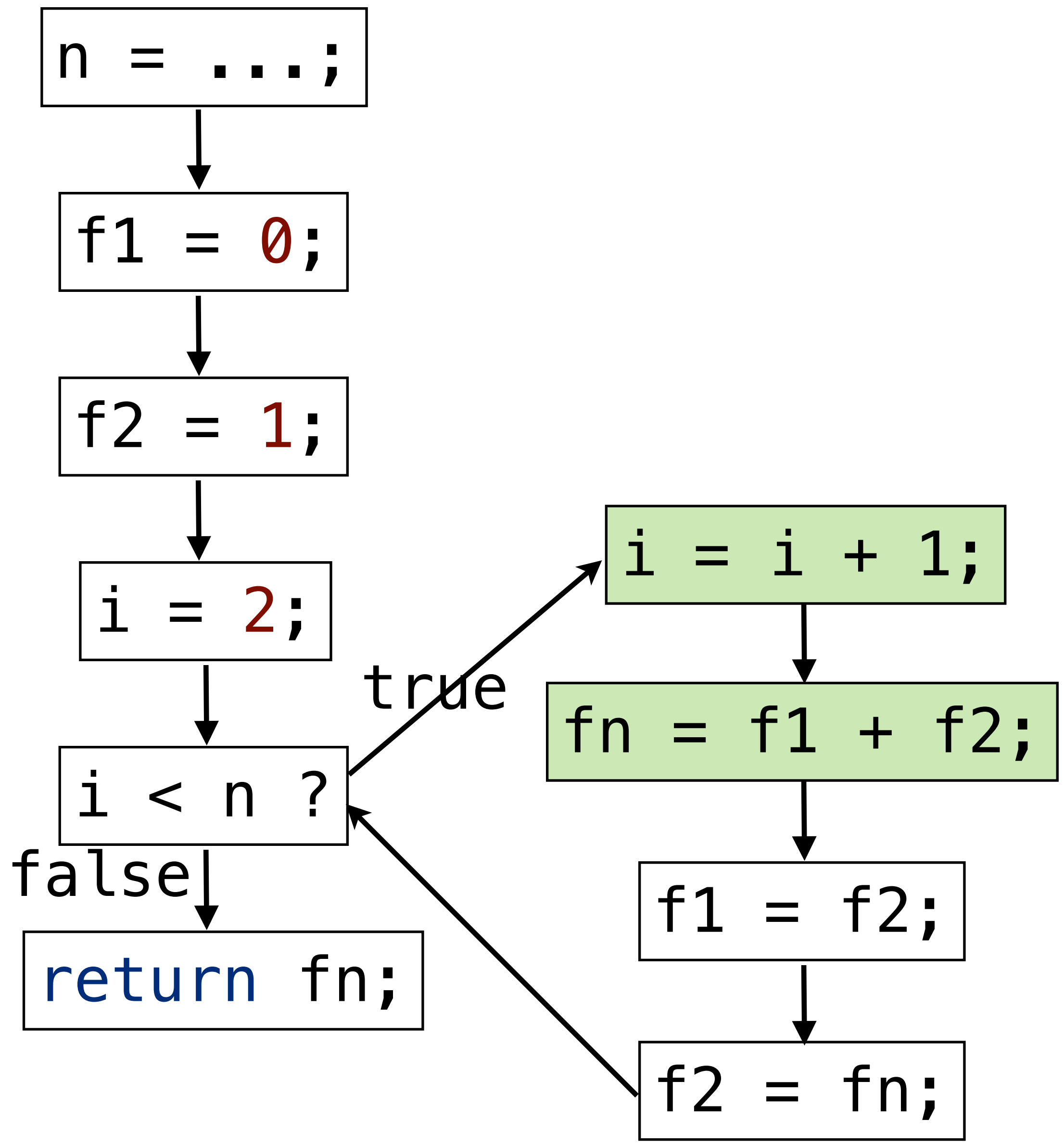
Very Busy Expressions: isVBE(i<n)

`isVBE(i<n)`



Very Busy Expressions: isVBE(i<n)

`isVBE(f1+f2)`



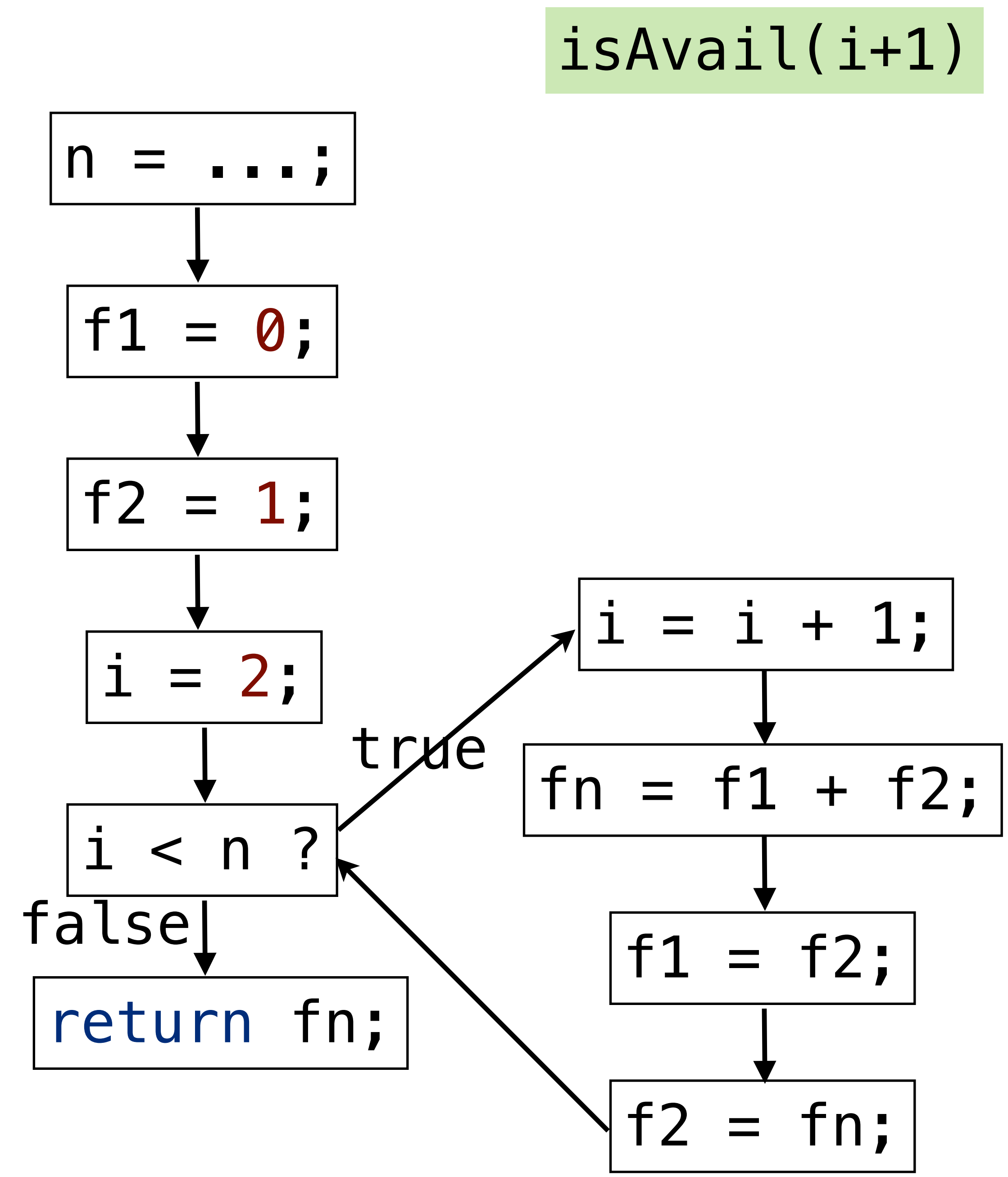
Available Expressions

- An expression $e = XopY$ is **available** at a point p if every path (not necessarily cycle-free) from the initial node to p contains $XopY$, and after the last such occurrence prior to reaching p, there are no subsequent definitions of X or Y.

- Useful for, e.g.: Common Subexpr. Elimination.

- The following CTL formula specifies the states on which e is available:

  isAvail(e) = $AX_{back}(ASU_{back}(\neg$isMod(e), isGen(e)))
  where
  isGen(e) = isUsed(e) $\wedge$ $\neg$isMod(e)

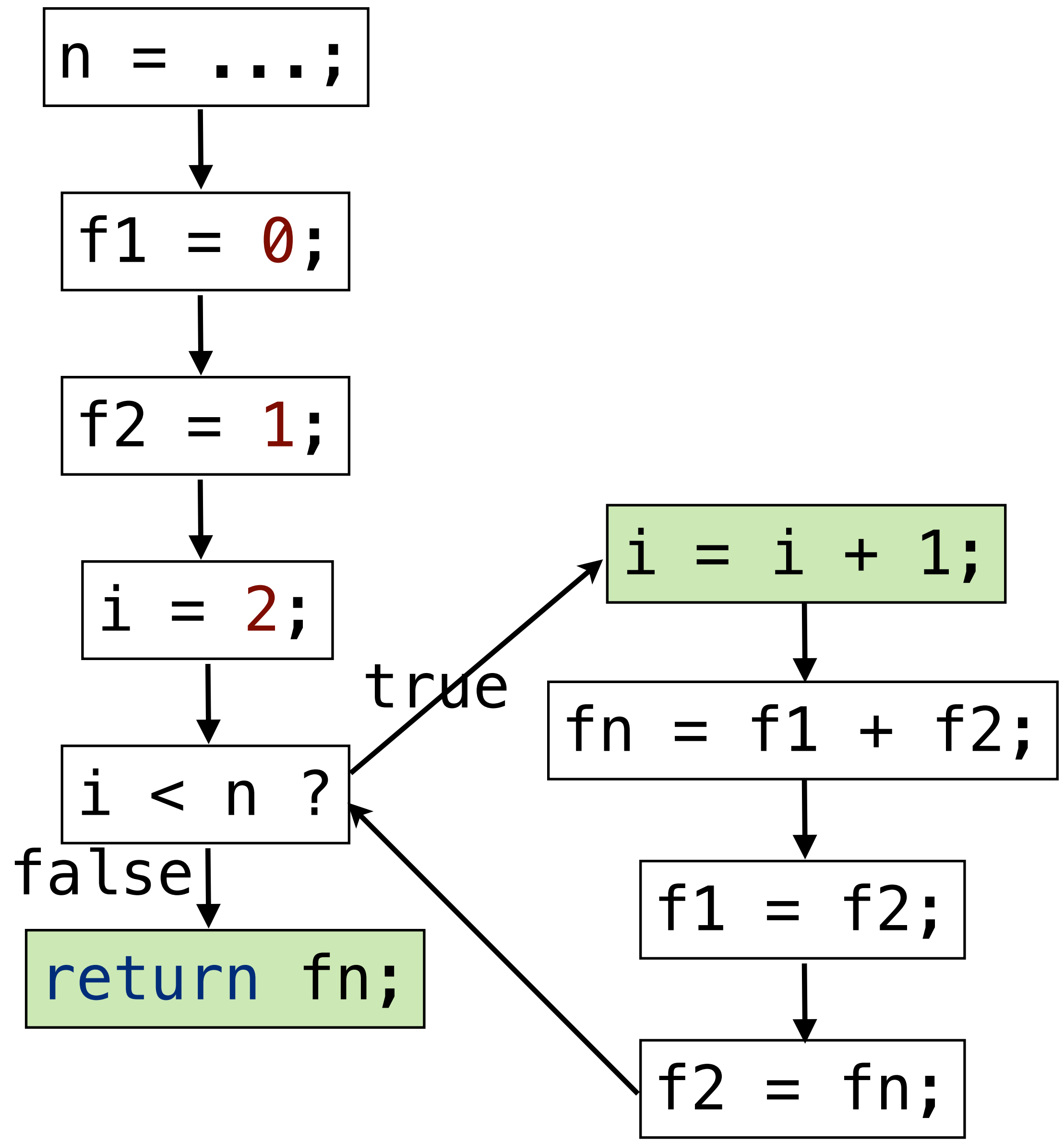


Available Expressions: Exercise

- Choose one of the remaining expressions (i.e. $e \in \{i<n, f1+f2\}$) and determine the states where isAvail(e) holds.

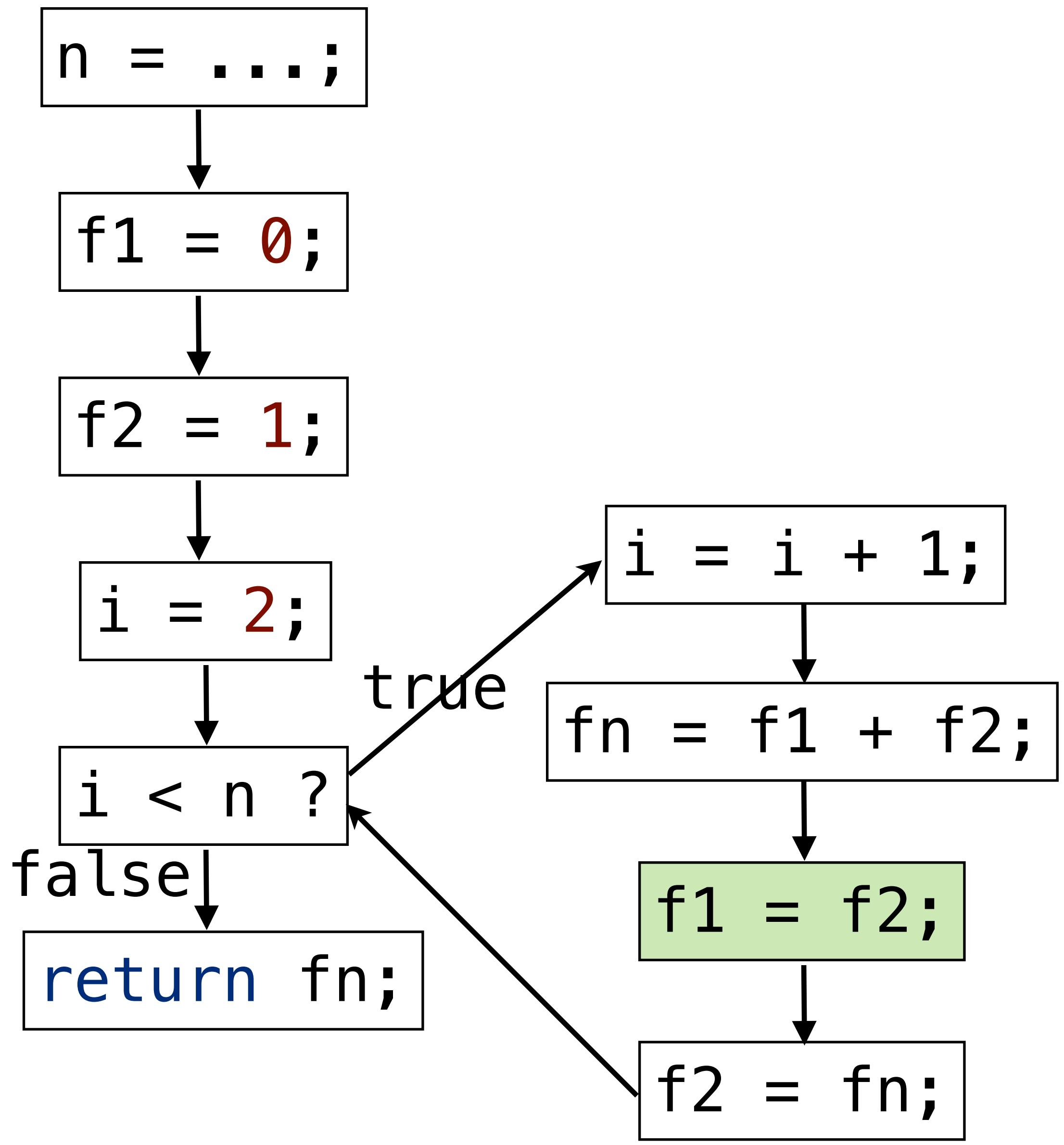
Available Expressions: isAvail(i<n)

`isAvail(i<n)`



Available Expressions: isAvail(f1+f2)

`isAvail(f1+f2)`



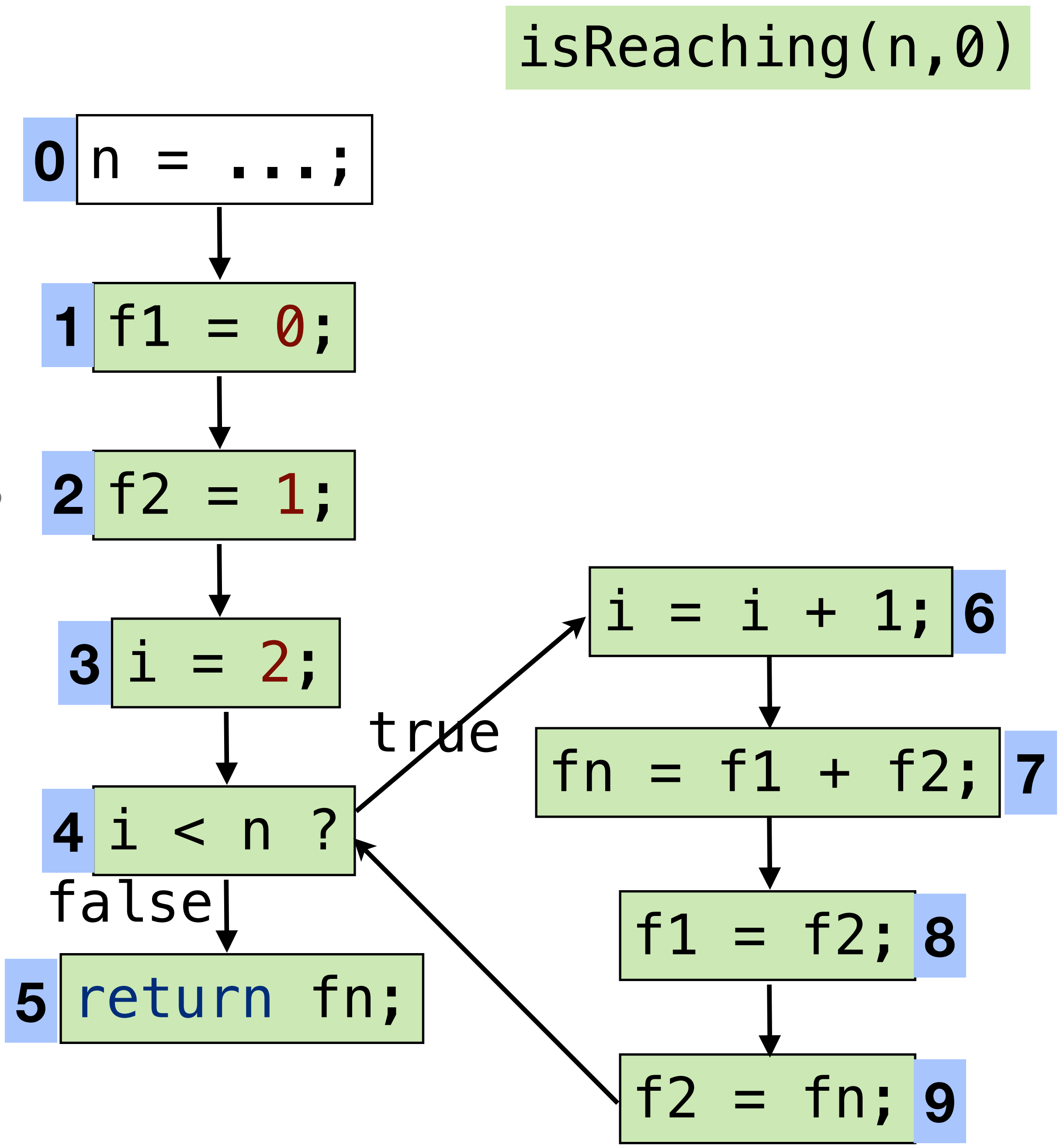
Reaching Definitions

- A definition of a variable *a* **reaches** a point *p* if there is a path in the flow graph from that definition to *p*, such that no other definitions of *a* appear on the path.

- Useful for, e.g.: construction of direct links

- The following formula specifies the states that are reached by the definition of variable a at state s:

  isReaching(a,s) = $EX_{back}(ESU_{back}(\text{isPreserved}(a,s), s))$
  where
  isPreserved(a,s) = $\neg$isDef(a) $\wedge$ EF_{back}(s)

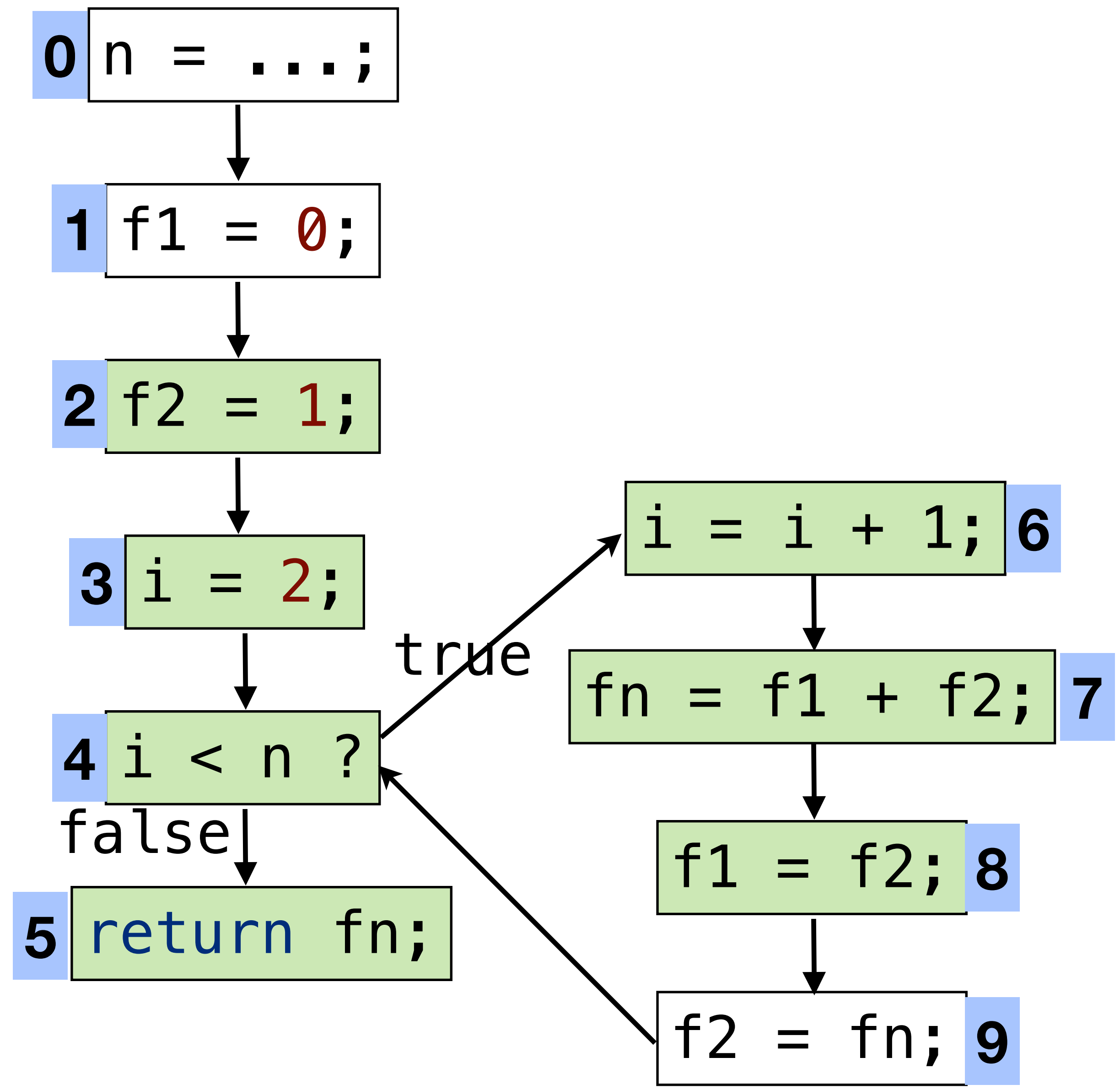


Reaching Definitions: Exercise

- Choose one of the remaining definitions (a,s) and determine the states where isReaching(a,s) holds.

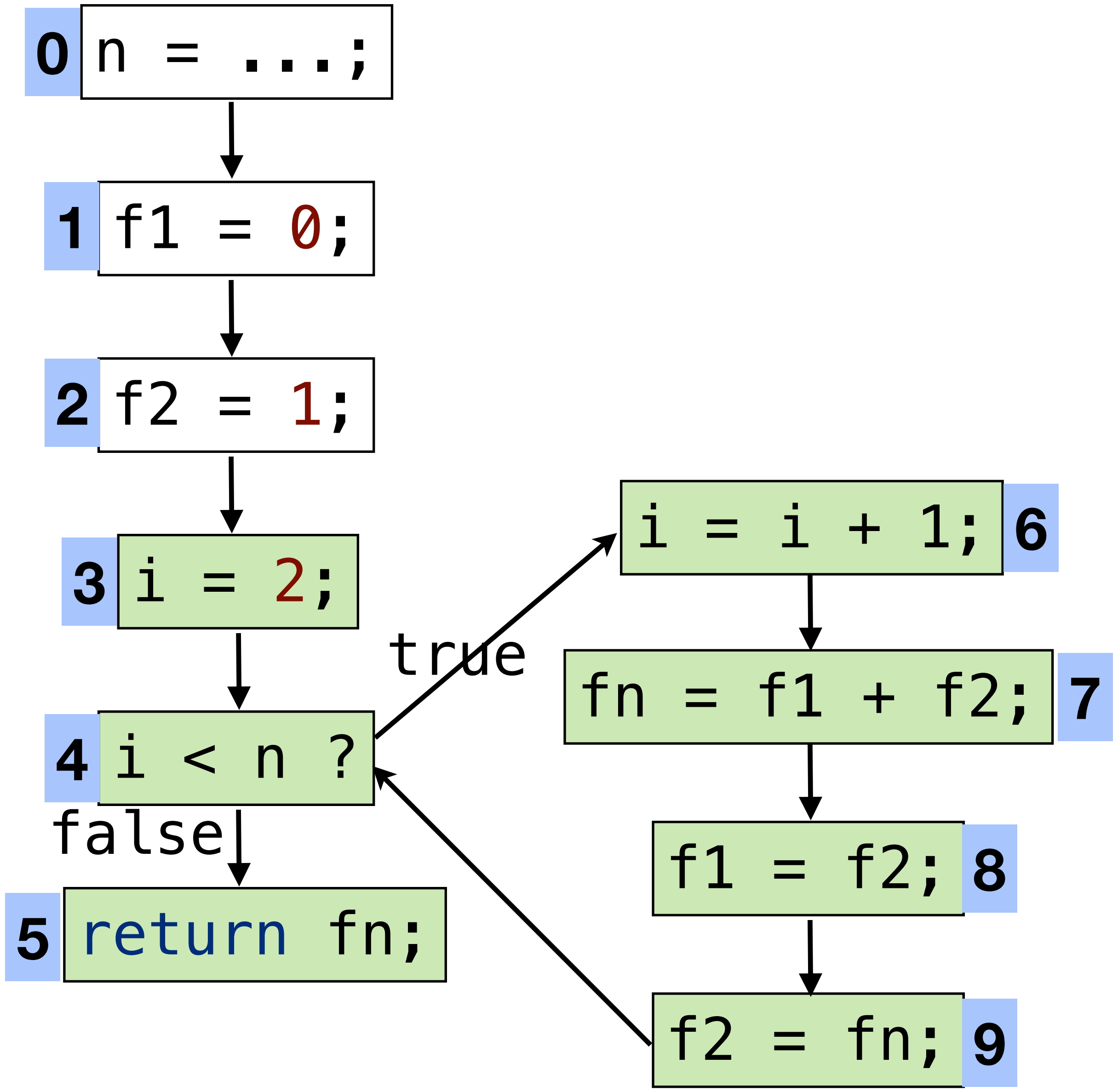
Reaching Definitions: isReaching(f1,1)

`isReaching(f1,1)`



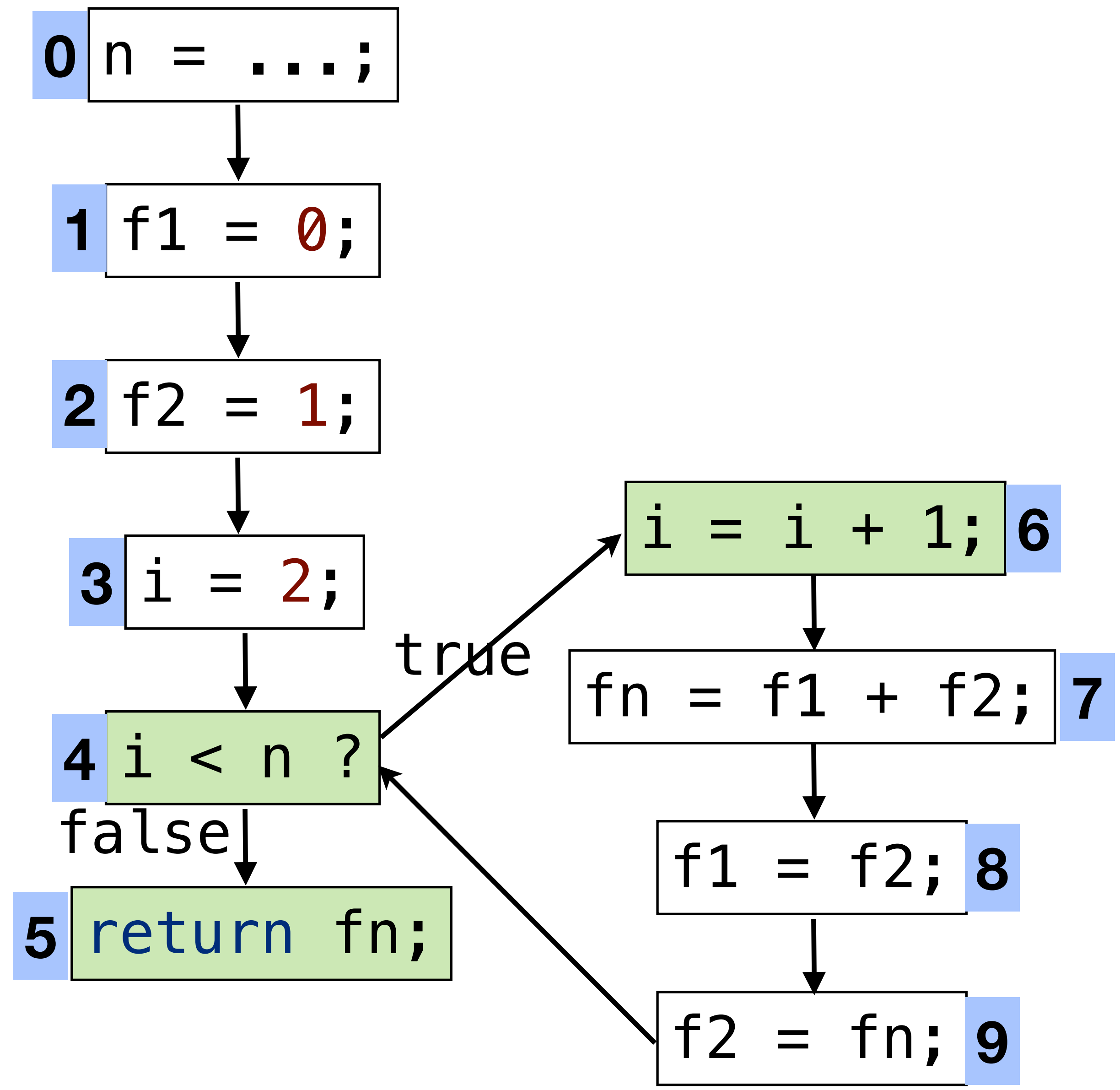
Reaching Definitions: isReaching(f2,2)

`isReaching(f2,2)`



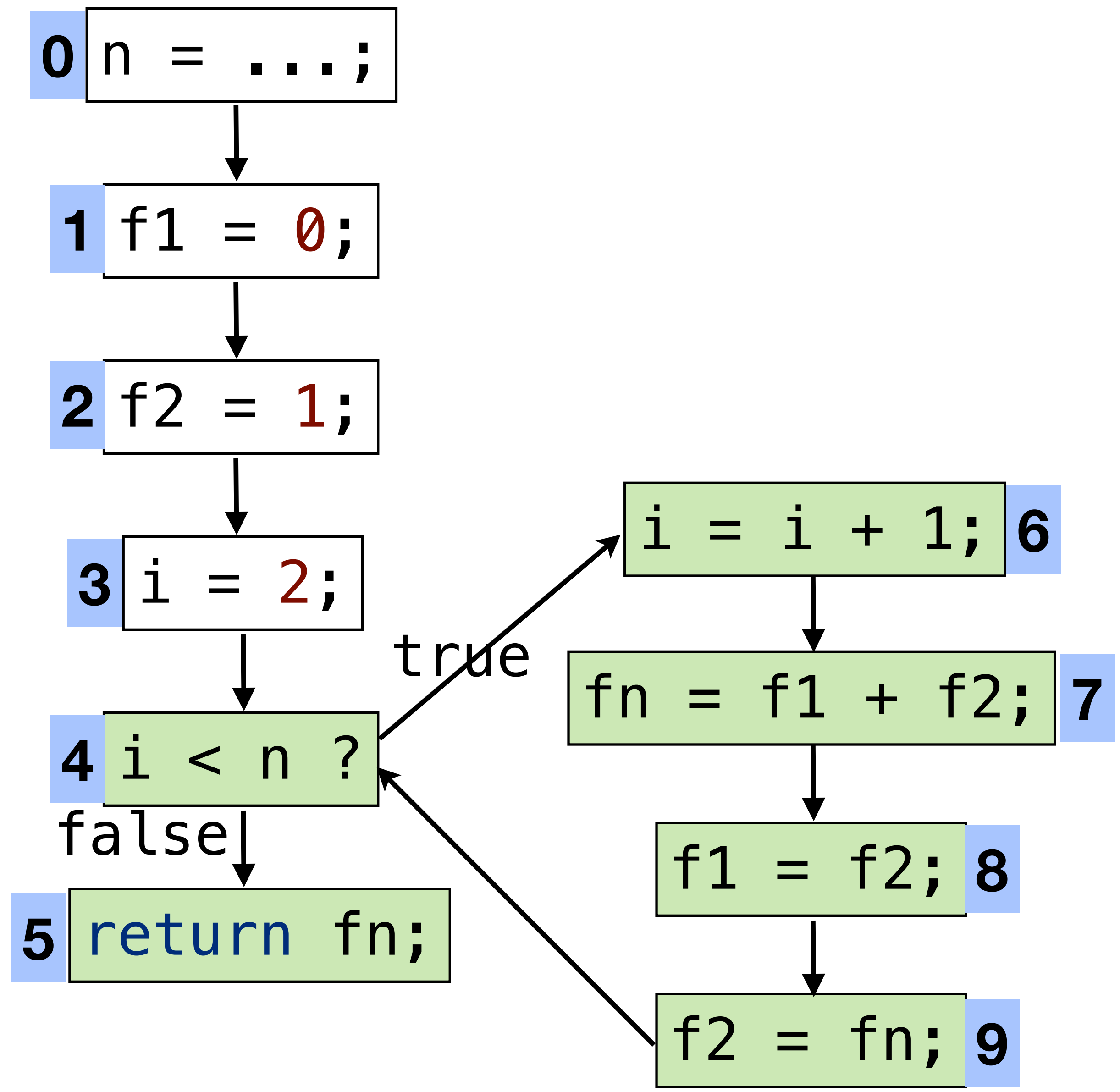
Reaching Definitions: isReaching(i,3)

`isReaching(i,3)`



Reaching Definitions: isReaching(i,6)

```
isReaching(i,6)
isReaching(fn,7)
isReaching(f1,8)
isReaching(f2,9)
```



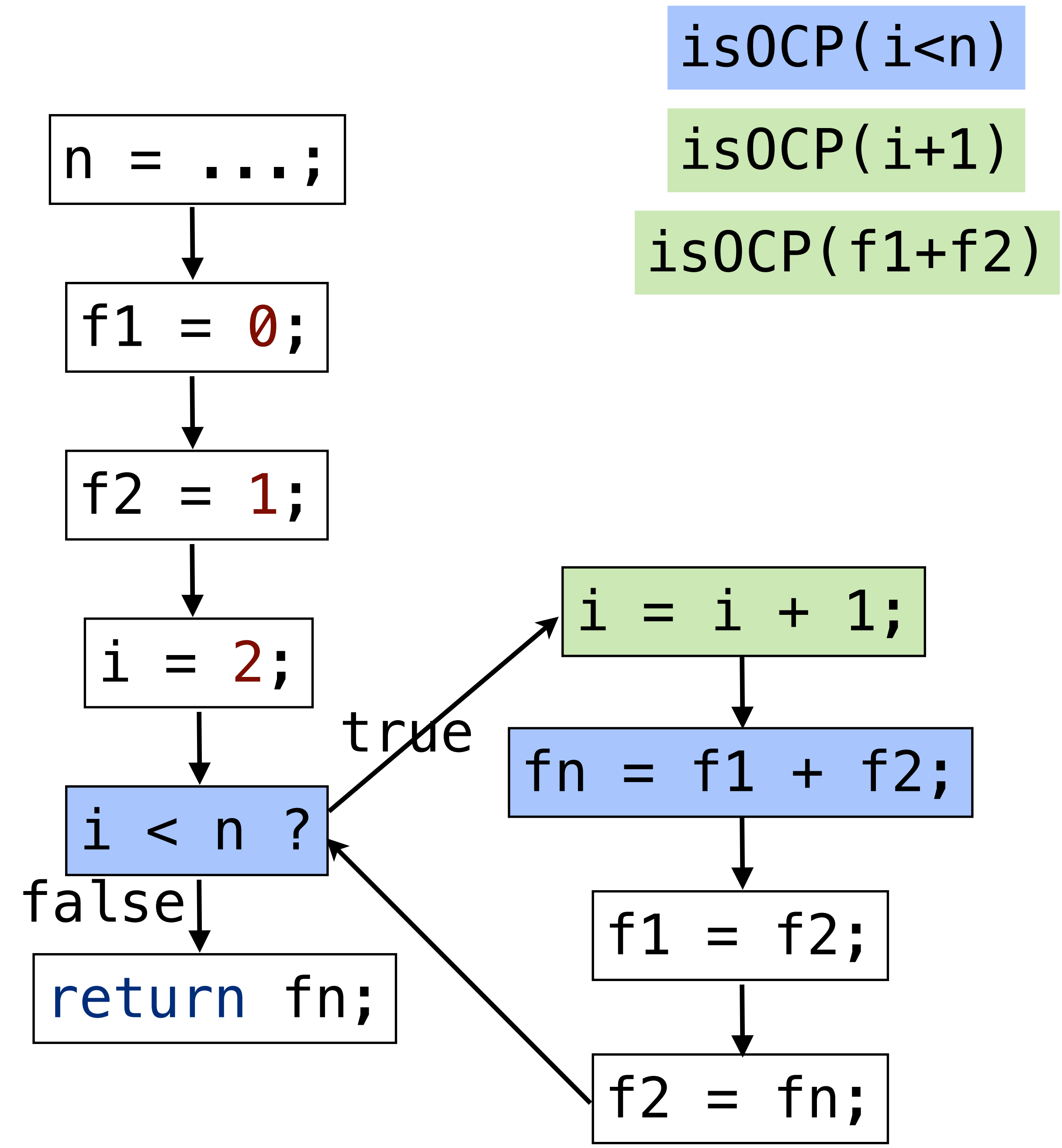
Further Analyses

- Live Definitions
- Common Subexpressions
- Use-Definition Chaining
- Definition-Use Chaining
- Copy Propagation
- Optimal Computation Points
- ...

Optimal Computation Points

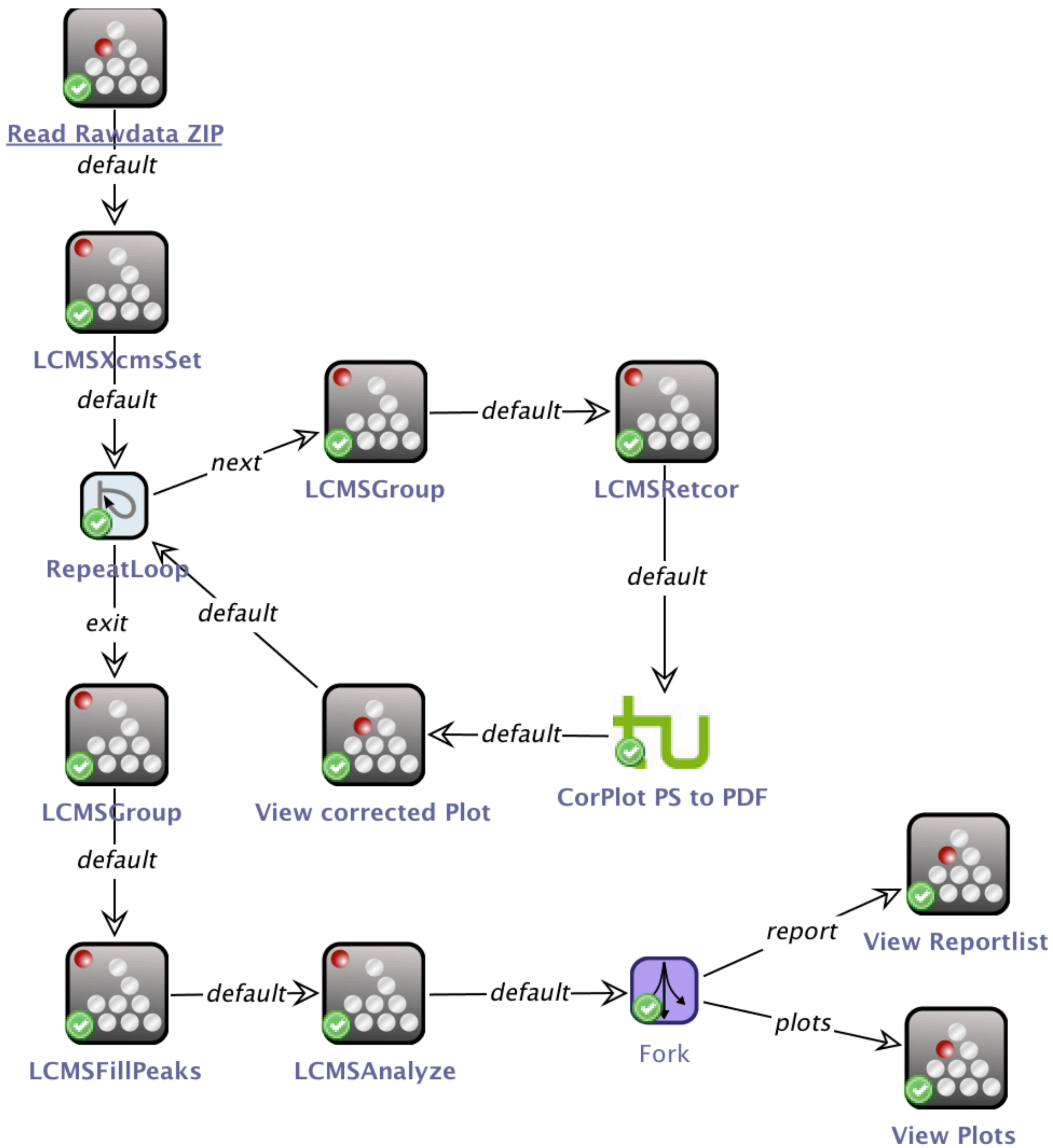
- Consider an expression e=XopY. The following formula specifies the state that is the optimal computation point for e:

  isOCP(e) = isSafe(e) ∧ isEarly(e)
    where
  isSafe(e) = ASU(¬isMod(e), isUsed(e))
    and
  isEarly(e) = AX_{back}(false) ∨ ¬(AX_{back}(AWU_{back}(¬(isMod(e) ∨ AX_{back}(false)), isSafe(e) ∧ ¬isMod(e))))



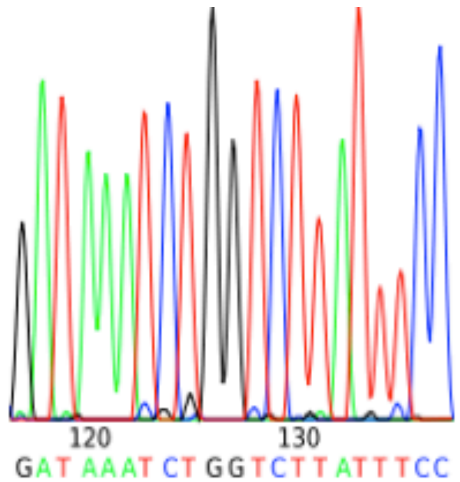
Higher-Level Applications

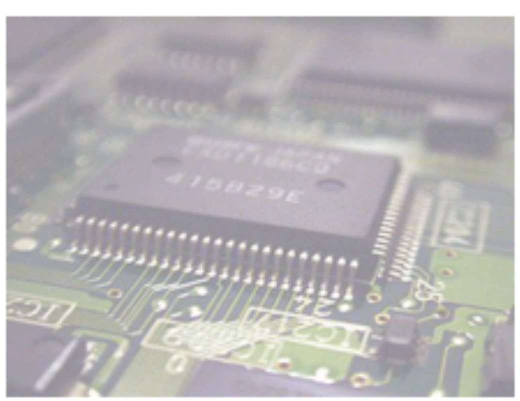
- **Compilers** use highly optimized techniques for data-flow analyses (e.g. bit vector analysis algorithms).
- Model checking is more natural in the context of **model-driven development**, such as model-based workflow design.
- **Services** (the workflow building blocks) can be annotated with **isDef** and **isUsed** information as well.



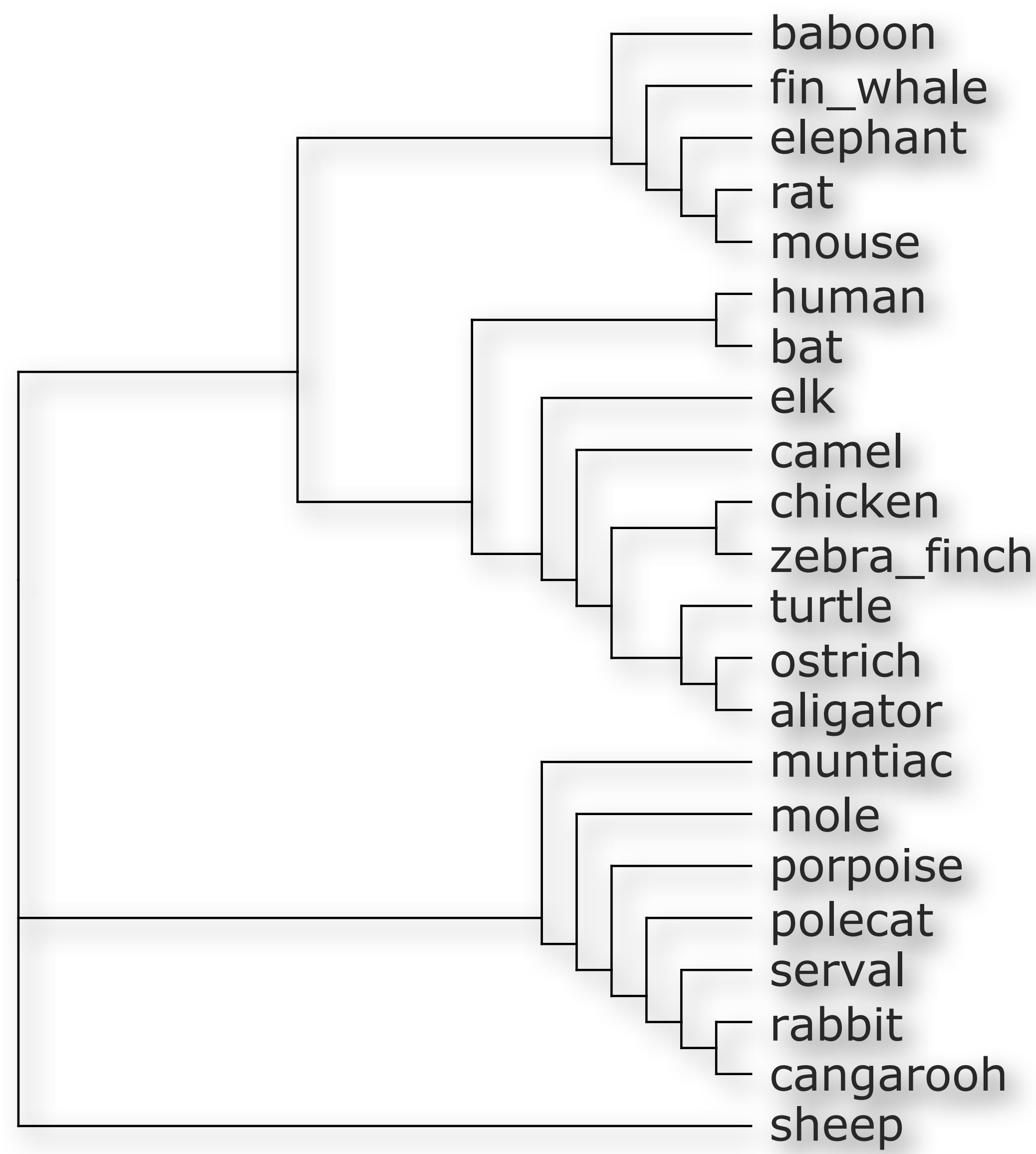
jABC
+
GEAR

Background: Phylogenetic Analyses

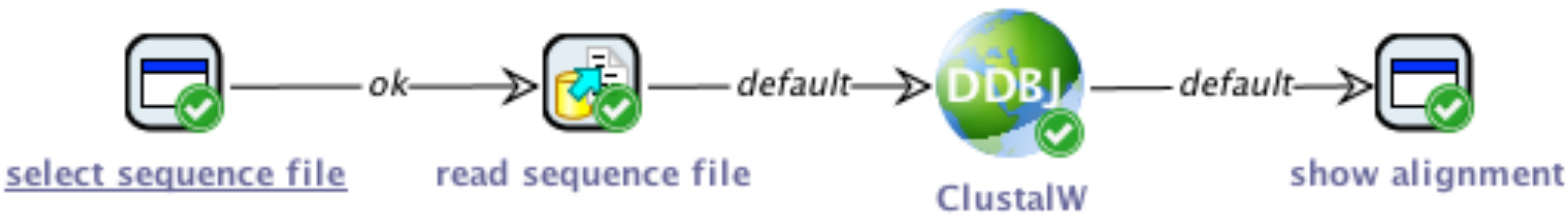




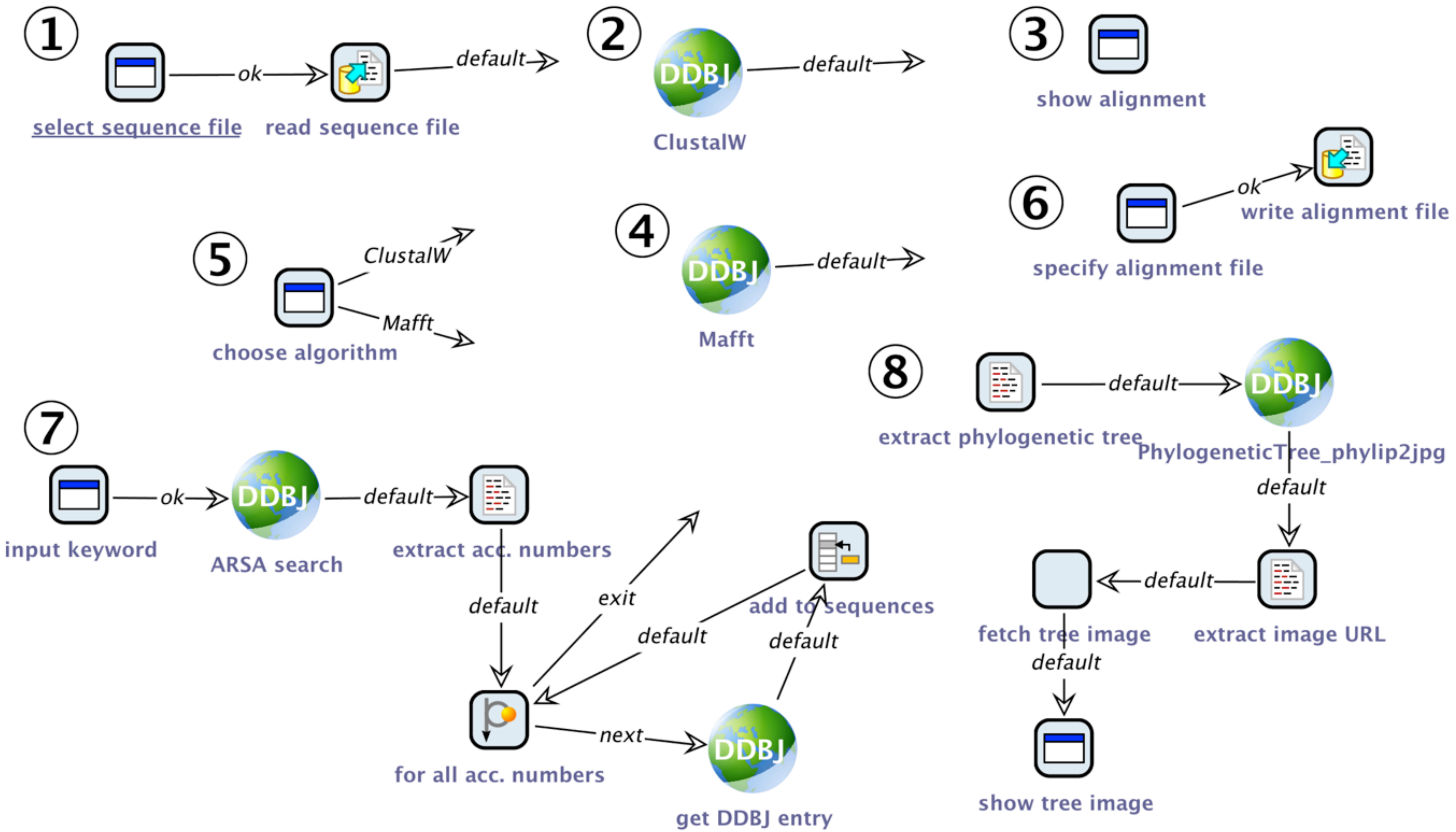
```
CLUSTAL W (1.83) multiple sequence alignment


human           TCTGCCT
bat             TCTGCCT
chicken         -------
zebra_finch     -------
ostrich         TCTGCCT
aligator        TCTGCCT
turtle          TCTGCCT
camel           TCTGCCT
elk             TCTGCCT
rat             TCTGCCT
mouse           TCTGCCT
elephant        TCTGCCT
fin_whale       TCTGCCT
baboon          TCTGCCT
sheep           TCTGCCT
muntiac         TCTGCCT
mole            TCTGCCT
porpoise        TCTGCCT
polecat         -------
serval          -------
rabbit          -------
cangarooh       -------
```



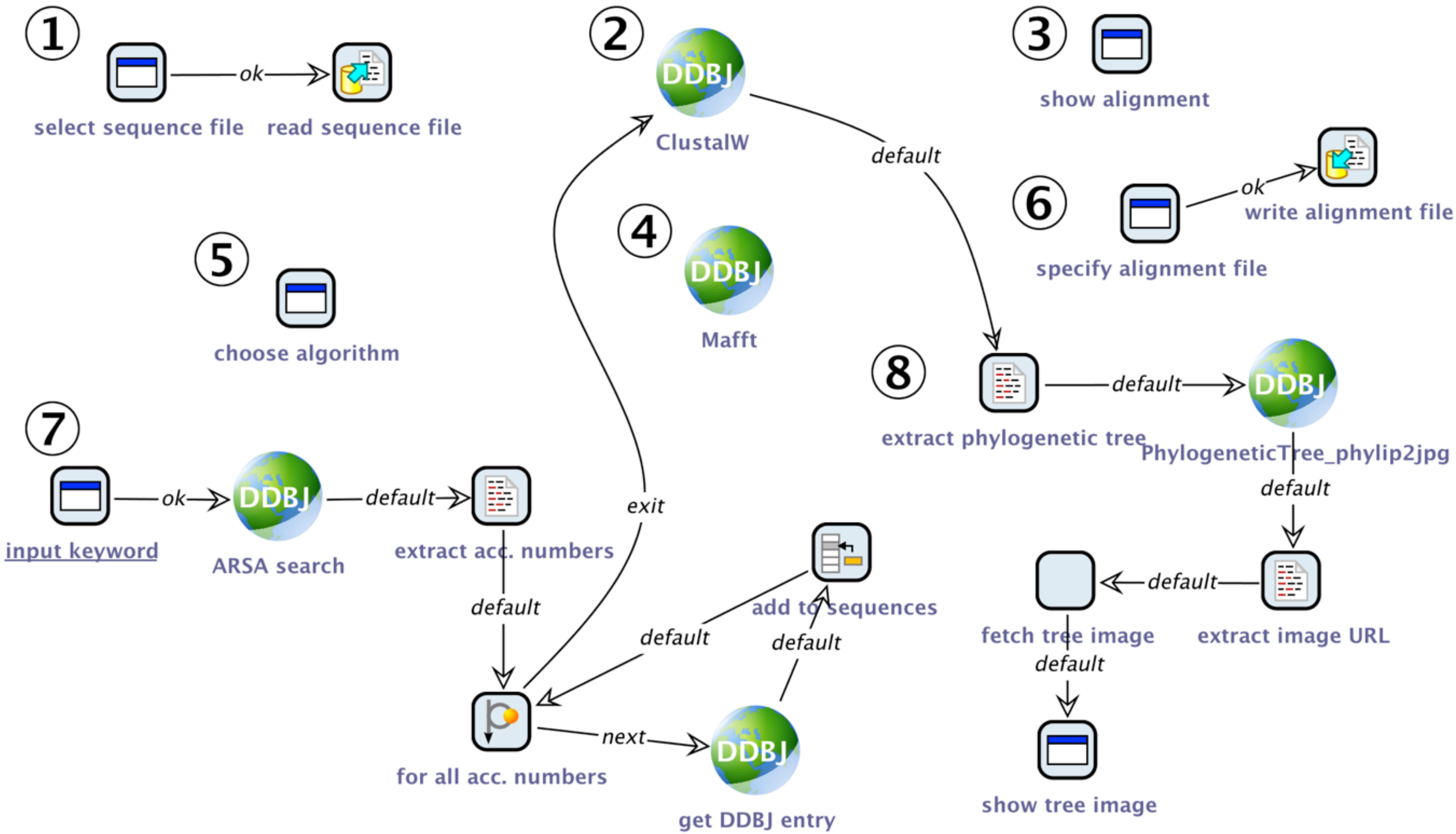
Simple Phylogenetic Analysis Workflow



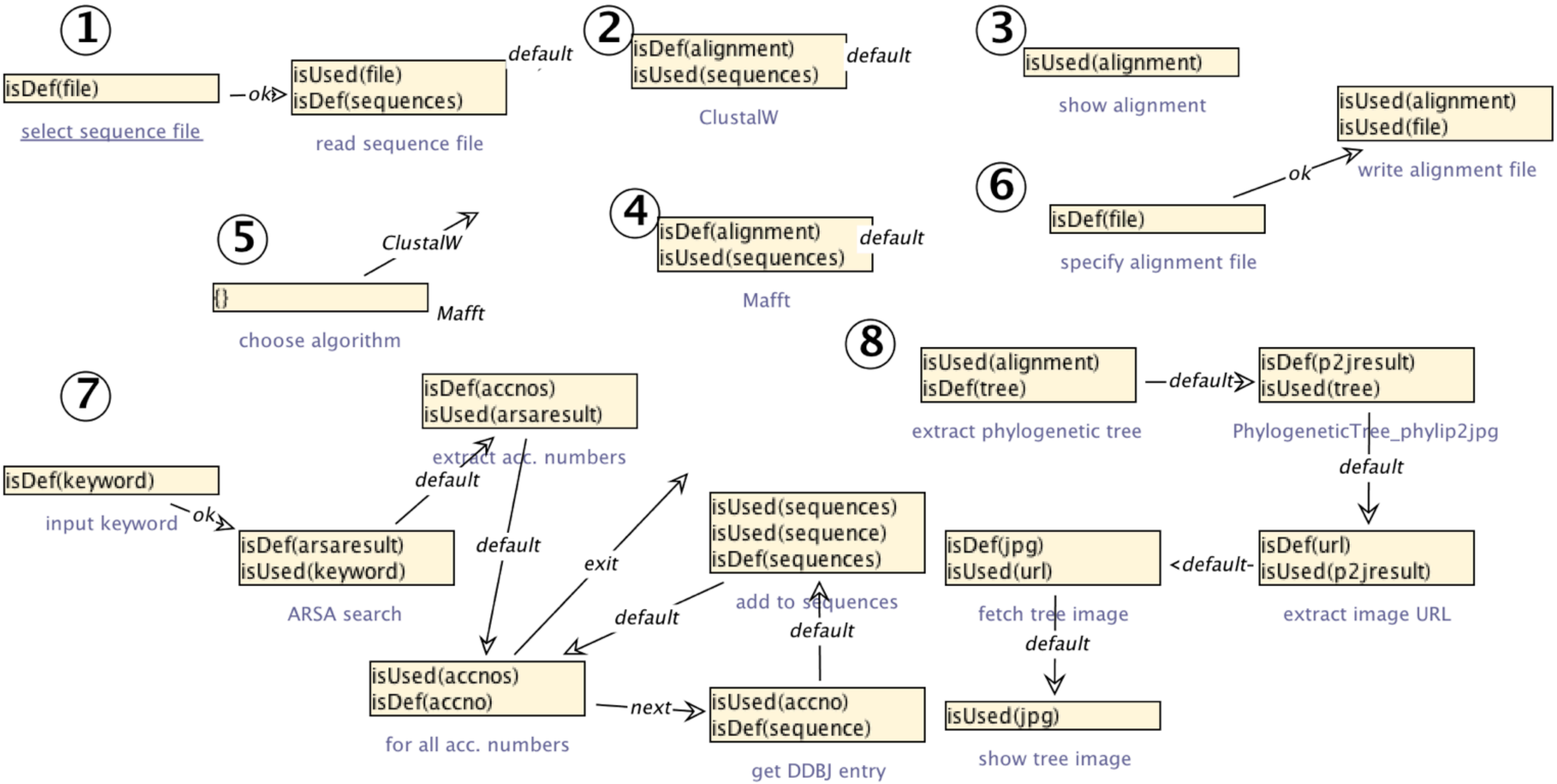
Snippets for Phylogenetic Analysis Workflows



Snippets for Phylogenetic Analysis Workflows



Data-Flow Annotations (isDef, isUsed)



DFA-MC for Higher-Level Applications

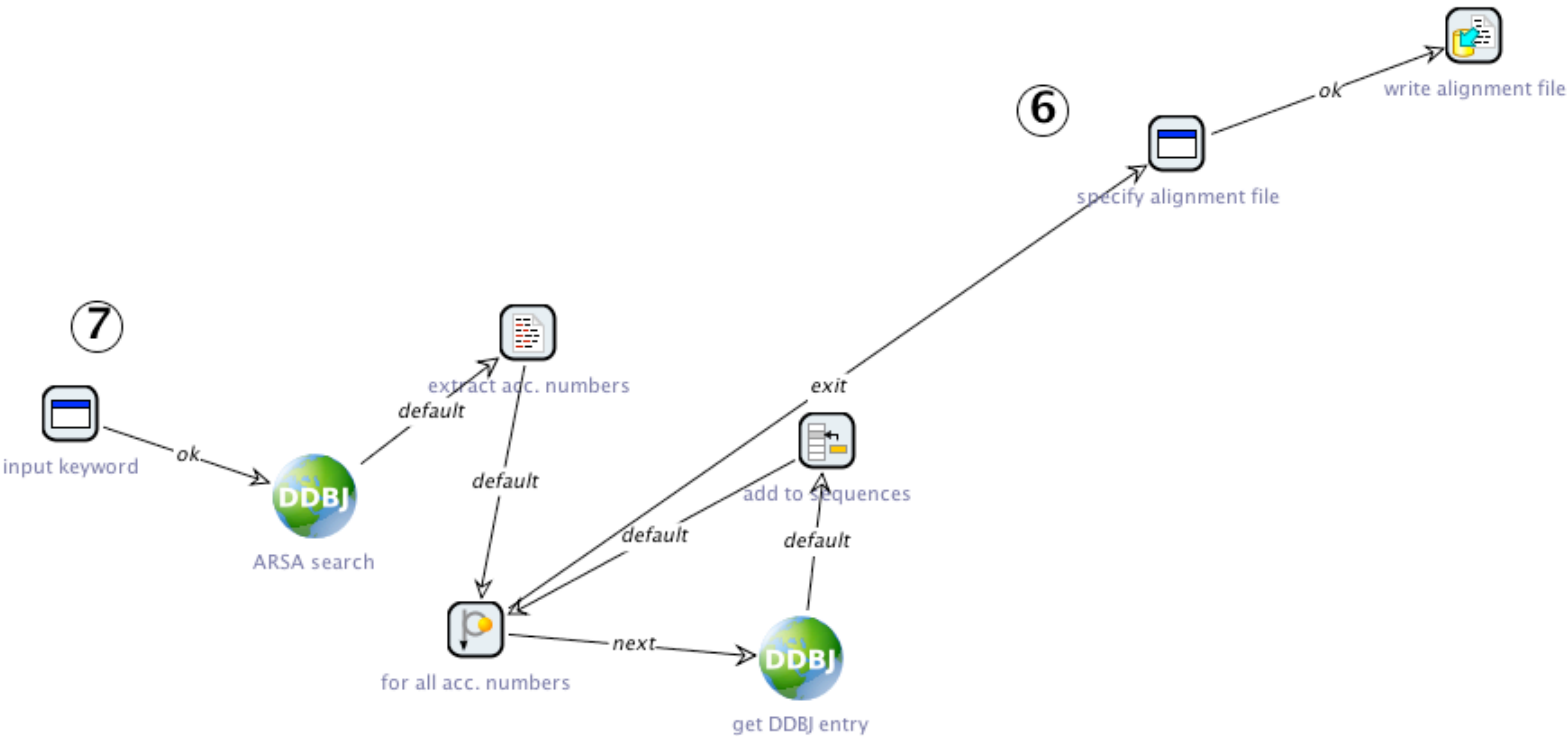
- **Can** compute live/dead variables and reaching definitions information, in workflows with real expressions also very busy and available expressions.
- But: **workflows** do not have the same optimization problems as compilers...
- Other DFA analyses make more sense for workflows, e.g.:
  - Ensuring that if variable x is used, it has been defined before:

    $isUsed(x) \Rightarrow AF_{back}(isDef(x))$

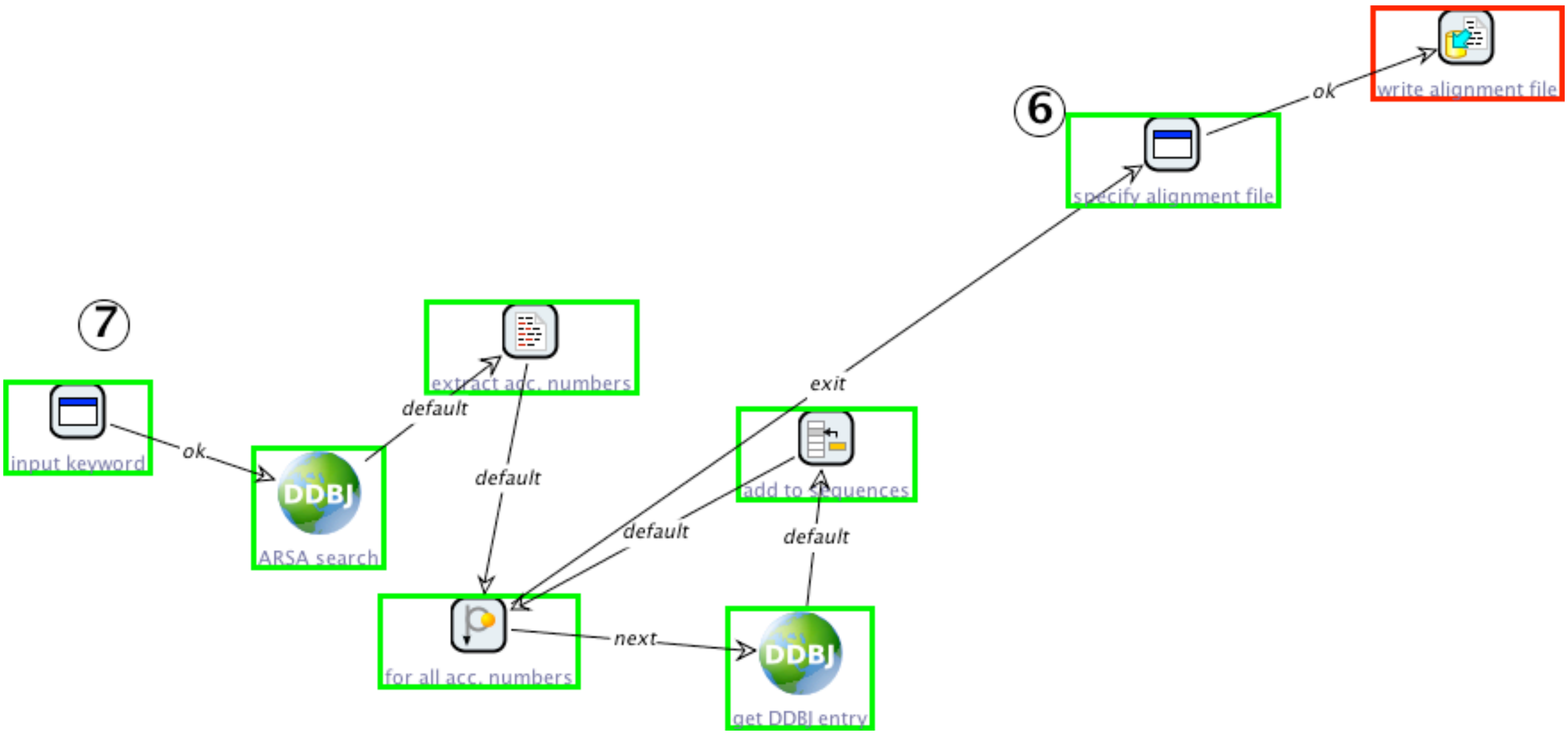
  - Ensuring that if variable x of type y is used, it has been defined with this type before and not been overwritten since:

    $(isUsed(x) \wedge type(x)=y) \Rightarrow ASU_{back}(\neg isDef(x), isDef(x) \wedge type(x)=y)$

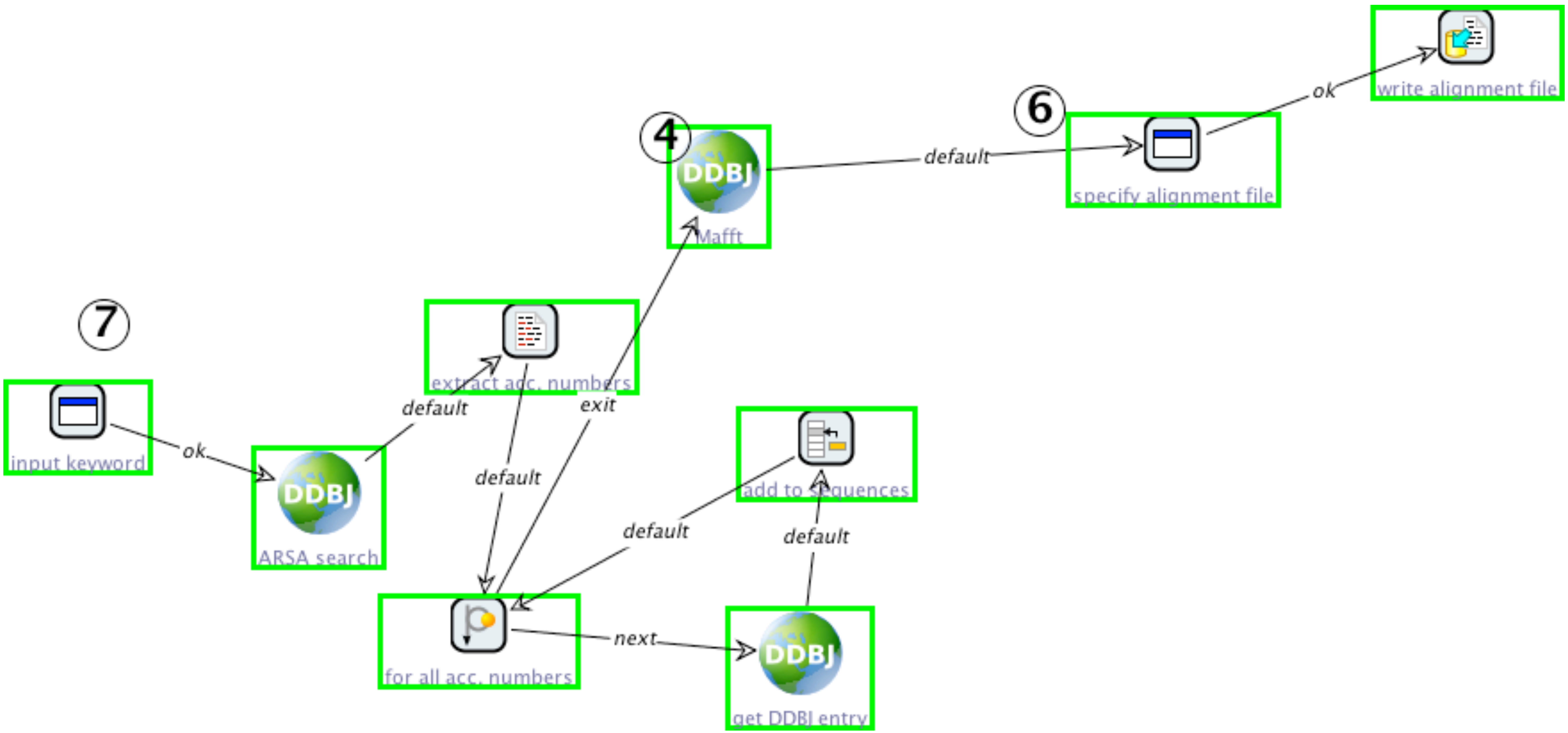
Workflow Variant



isUsed(alignment) $\Rightarrow$ AF$_{back}$(isDef(alignment))



isUsed(alignment) ⇒ AF_{back}(isDef(alignment))



Domain-Specific Constraints

- Even more attractive for (scientific) workflows are formulas that express **domain-specific constraints**, e.g.:
  - Extracting a phylogenic tree only works for ClustalW alignments:

    $(isUsed(x) \wedge type(x)=alignment \wedge extractPhylogeneticTree)$
    $\Rightarrow ASU_{back}(\neg isDef(x), isDef(x) \wedge type(x)=alignment \wedge ClustalW)$

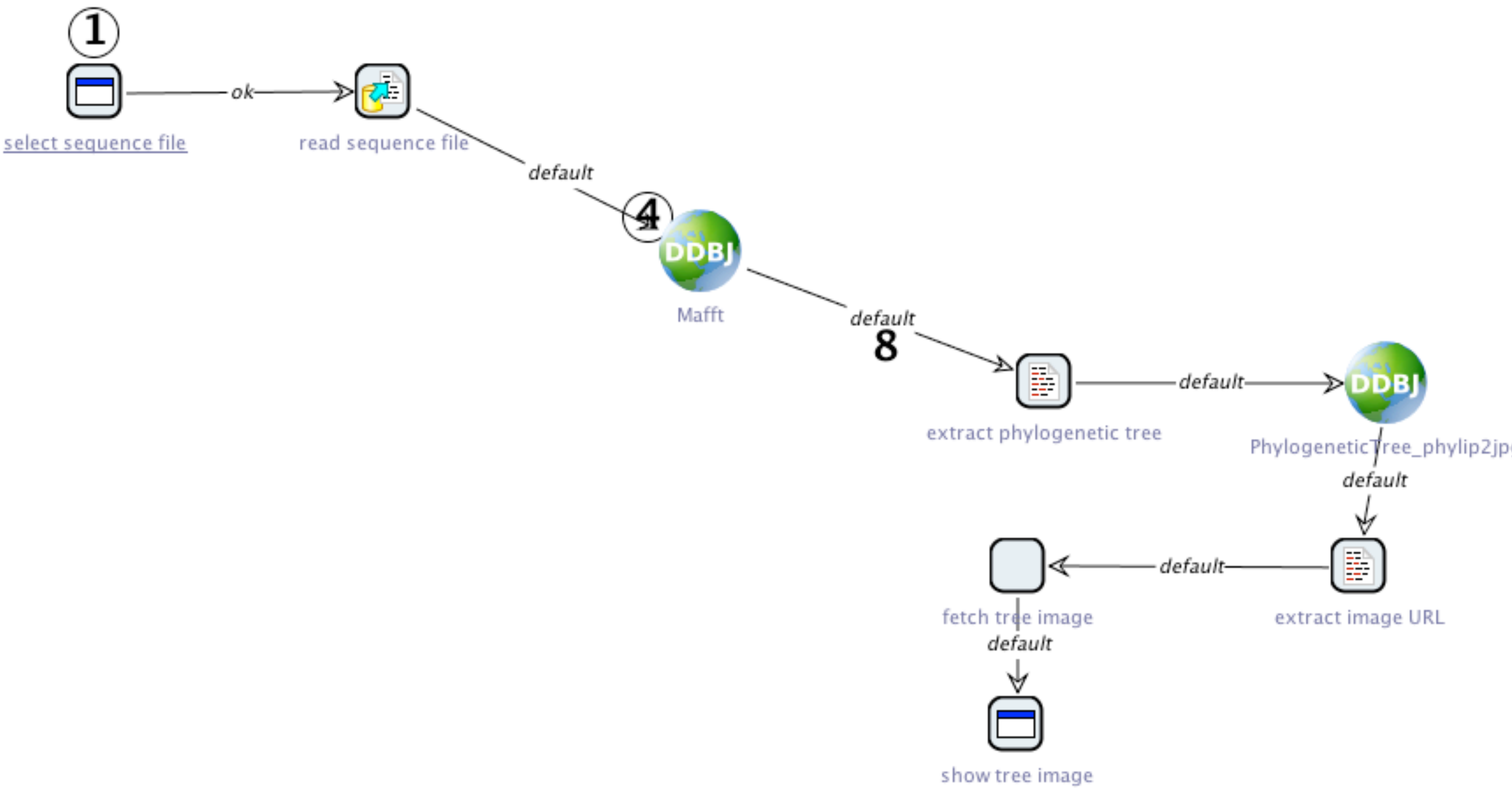
  - Computed alignments should always be saved:

    $isDef(alignment) \Rightarrow AF(isUsed(alignment) \wedge writeAlignmentFile)$

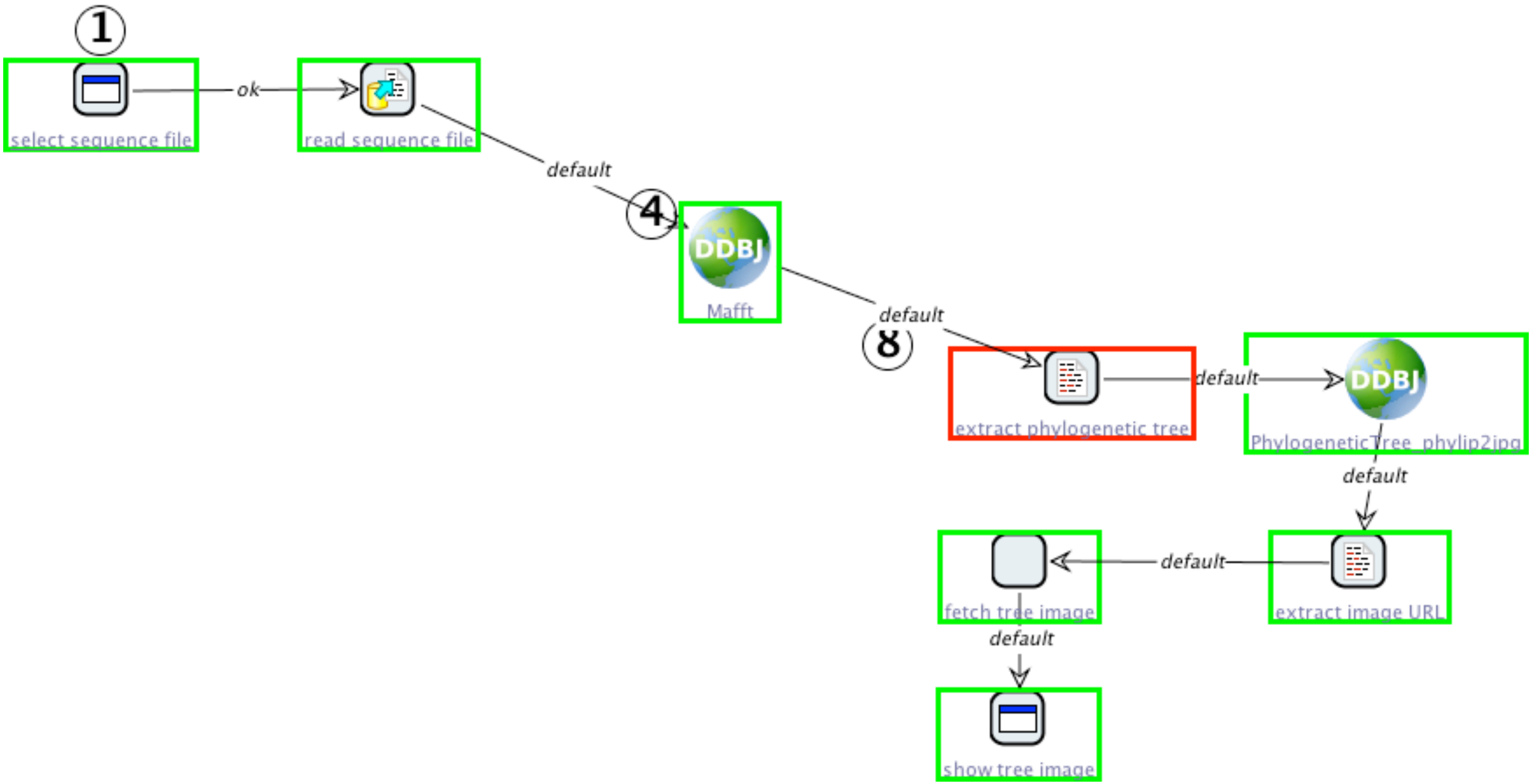
  - No alignment computation before an algorithm has been chosen:

    $AWU(\neg\ alignment, chooseAlgorithm)$

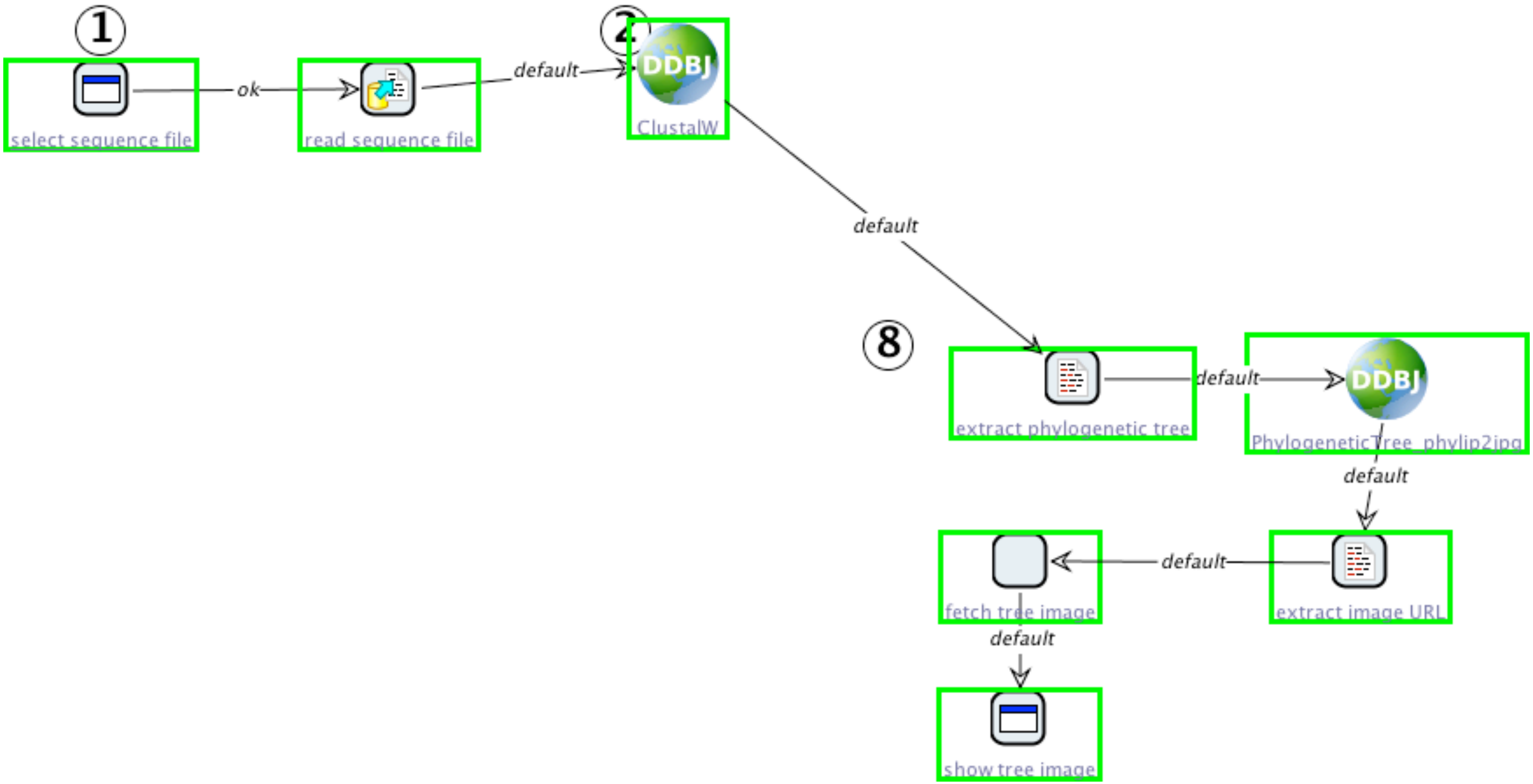
Workflow Variant



"Extracting a phylogenic tree only works for ClustalW alignments"

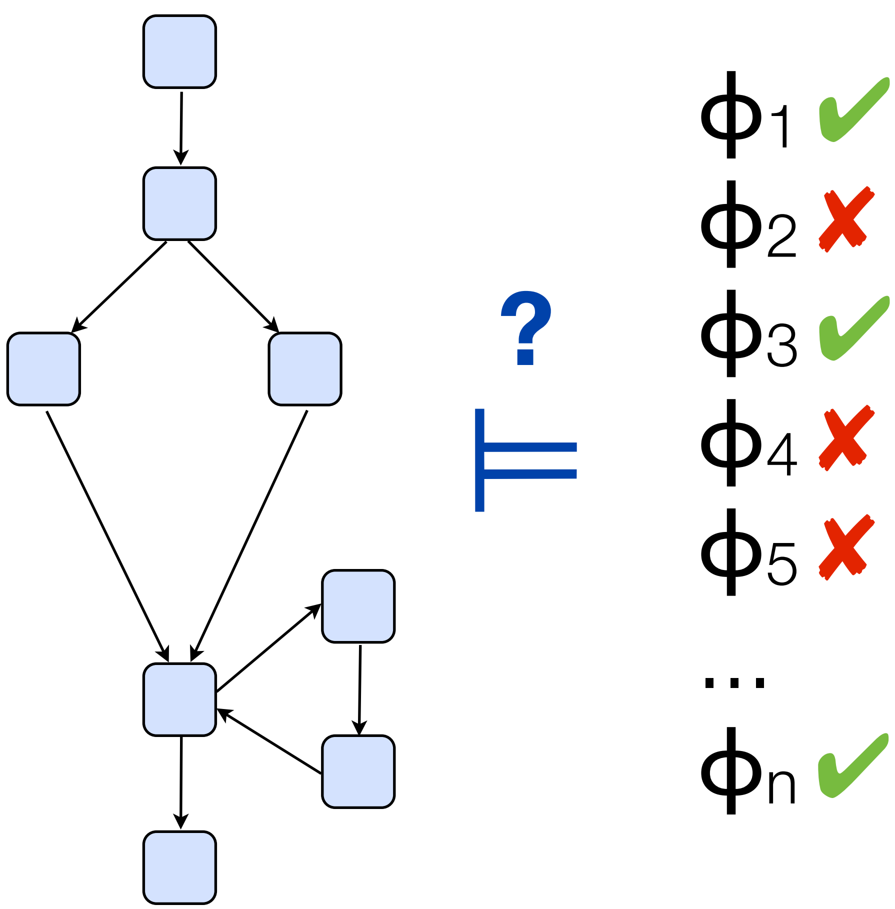


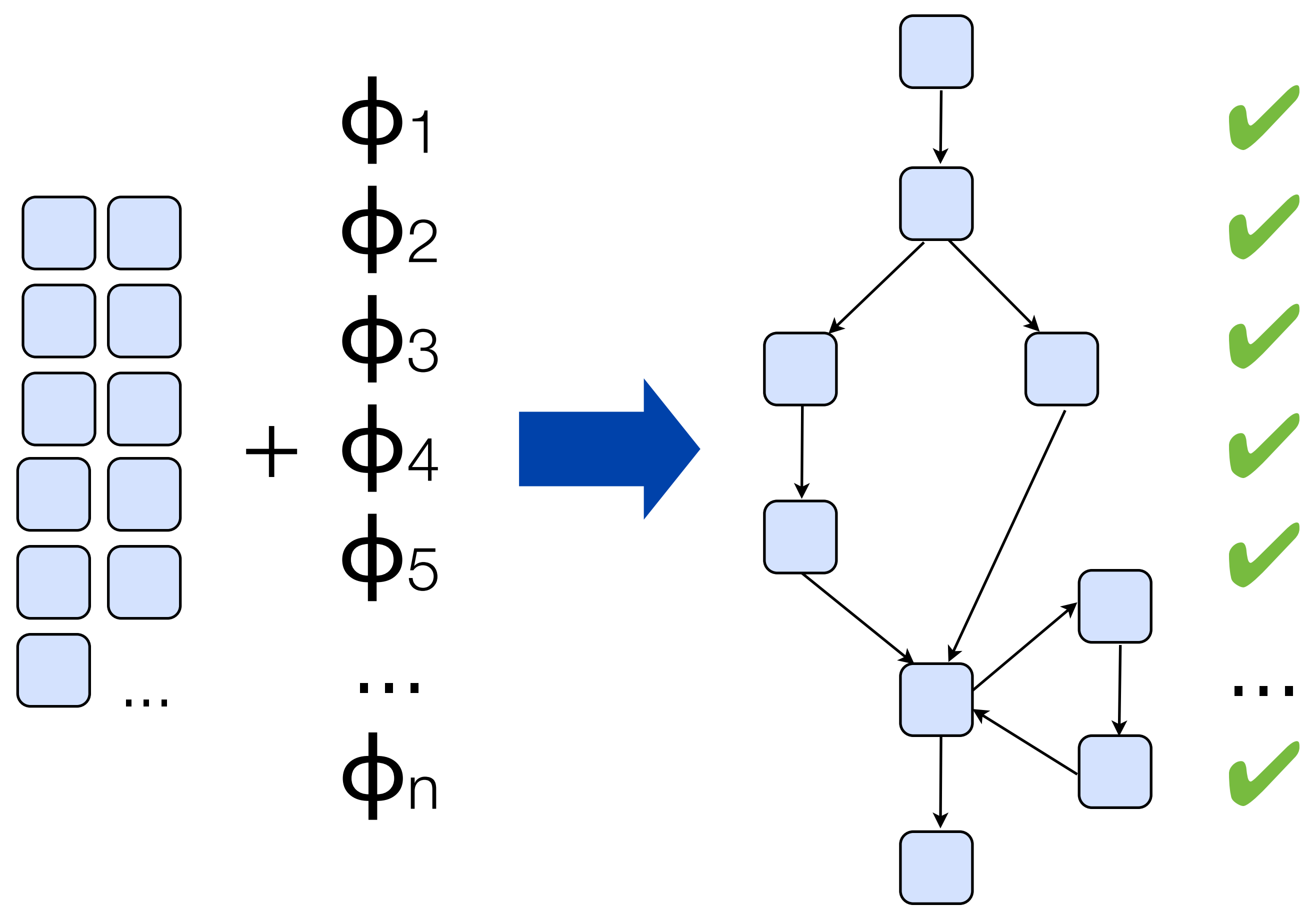
"Extracting a phylogenic tree only works for ClustalW alignments"



Outlook: Constraint-Driven Workflow Design

- **constraint-*guarded***: monitoring of workflow development by continuous **model checking**
- **constraint-*driven***: **synthesis** of workflow models that conform to the constraints by construction





The End

- Thank you!

References

- Clarke, E. M.; Grumberg, O. & Peled, D. A.: *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999
- Lamprecht, A.-L.: *Intraprocedural program analysis via model checking*. Bachelor thesis. Georg-August-Universität Göttingen, 2005
- Lamprecht, A.-L.; Margaria, T. & Steffen, B.: *Data-Flow Analysis as Model Checking Within the jABC.* In Mycroft, A. & Zeller, A. (Eds.) Compiler Construction, 2006, Springer LNCS 3923, pp. 101-104
- Lamprecht, A.-L.: *User-Level Workflow Design - A Bioinformatics Perspective.* 2013, Springer LNCS 8311

References

- Müller-Olm, M.; Schmidt, D. & Steffen, B.: *Model-Checking - A Tutorial Introduction.* Proceedings of the 6th International Symposium on Static Analysis (SAS '99), 1999, Springer LNCS 1694, pp. 330-354
- Schmidt, D. A. & Steffen, B.: *Program Analysis as Model Checking of Abstract Interpretations*. Proceedings of the 5th International Symposium on Static Analysis, 1998, Springer LNCS 1503, pp. 351–380
- Steffen, B.: *Data Flow Analysis as Model Checking*. Proceedings of the International Conference on Theoretical Aspects of Computer Software, Springer, 1991, Springer LNCS 526, pp. 346–365
- Steffen, B.: *Generating Data Flow Analysis Algorithms from Modal Specifications*, Science of Computer Programming, 1993, 21, pp. 115-139