Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program.

The concept of *expression availability* is useful in dealing with this situation.

Expressions

Any given program contains a finite number of expressions (i.e. computations which potentially produce values), so we may talk about the *set of all expressions* of a program.

```
int z = x * y;
print s + t;
int w = u / v;
 ⋮
```

program contains expressions { `x*y`, `s+t`, `u/v`, ... }

Availability

Availability is a data-flow property of expressions:
"Has the value of this expression already been computed?"

```
    ⋮
    int z = x * y;
}
```

Availability

At each instruction, each expression in the program is either available or unavailable.

We therefore usually consider availability from an instruction's perspective: each instruction (or node of the flowgraph) has an associated *set* of available expressions.

```
   int z = x * y;
   print s + t;
n: int w = u / v;
   ⋮
```

avail(*n*) = { `x*y`, `s+t` }

Availability

So far, this is all familiar from live variable analysis.

Note that, while expression availability and variable liveness share many similarities (both are simple data-flow properties), they do differ in important ways.

By working through the low-level details of the availability property and its associated analysis we can see where the differences lie and get a feel for the capabilities of the general data-flow analysis framework.

Available expression analysis

Available expressions is a *forwards* data-flow analysis: information from past instructions must be propagated forwards through the program to discover which expressions are available.

```
                t = x * y;
print x * y;                 if (x*y > 0)

      ⋮
      int z = x * y;
    }
```

Available expression analysis

Unlike variable liveness, expression availability flows *forwards* through the program.

As in liveness, though, each instruction has an effect on the availability information as it flows past.

Available expression analysis

An instruction makes an expression available when it *generates* (computes) its current value.

Available expression analysis

{ }

```
print a*b;
```
GENERATE a*b

{ a*b }

```
c = d + 1;
```
GENERATE d+1

{ a*b, d+1 }

```
e = f / g;
```
GENERATE f/g

{ a*b, d+1, f/g }

Available expression analysis

An instruction makes an expression unavailable when it *kills* (invalidates) its current value.

Available expression analysis

{ a*b, c+1, d/e, d-1 }

```
a = 7;
```

KILL a*b

{ c+1, d/e, d-1 }

```
c = 11;
```

KILL c+1

{ d/e, d-1 }

```
d = 13;
```

KILL d/e, d-1

{ }

Available expression analysis

As in LVA, we can devise functions *gen*(n) and *kill*(n) which give the sets of expressions generated and killed by the instruction at node n.

The situation is slightly more complicated this time: an assignment to a variable x kills *all expressions in the program* which contain occurrences of x.

Available expression analysis

So, in the following, E_x is the set of expressions in the program which contain occurrences of x.

gen(`x = 3`) = { }

kill(`x = 3`) = E_x

gen(`print x+1`) = { `x+1` }

kill(`print x+1`) = { }

gen(`x = x + y`) = { `x+y` }

kill(`x = x + y`) = E_x

Available expression analysis

As availability flows forwards past an instruction, we want to modify the availability information by *adding* any expressions which it generates (they become available) and *removing* any which it kills (they become unavailable).

{ `y+1` }

gen(`print x+1`) = { `x+1` }

{ `x+1`, `y+1` }

{ `x+1`, `y+1` }

kill(`x = 3`) = E_x

{ `y+1` }

Available expression analysis

If an instruction both generates and kills expressions, we must remove the killed expressions *after* adding the generated ones (cf. removing *def*(*n*) *before* adding *ref*(*n*)).

`{ x+1, y+1 }`

```
x = x + y
```

`{ y+1 }`

gen(`x = x + y`) = { `x+y` }

kill(`x = x + y`) = E_x

Available expression analysis

So, if we consider *in-avail*(*n*) and *out-avail*(*n*), the sets of expressions which are available immediately *before* and immediately *after* a node, the following equation must hold:

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \cup \mathit{gen}(n)\Big) \setminus \mathit{kill}(n)$$

Available expression analysis

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \cup gen(n)\Big) \setminus kill(n)$$

in-avail(*n*) = { `x+1`, `y+1` }

n: $\boxed{\texttt{x = x + y}}$

out-avail(*n*) = (*in-avail*(*n*) ∪ *gen*(*n*)) \ *kill*(*n*)

= ({ `x+1`, `y+1` } ∪ { `x+y` }) \ { `x+1`, `x+y` }

= { `x+1`, `x+y`, `y+1` } \ { `x+1`, `x+y` } = { `y+1` }

gen(*n*) = { `x+y` }   *kill*(*n*) = { `x+1`, `x+y` }

Available expression analysis

As in LVA, we have devised one equation for calculating *out-avail*(n) from the values of *gen*(n), *kill*(n) and *in-avail*(n), and now need another for calculating *in-avail*(n).

$$\textit{in-avail}(n) = ?$$

n: `x = x + y`

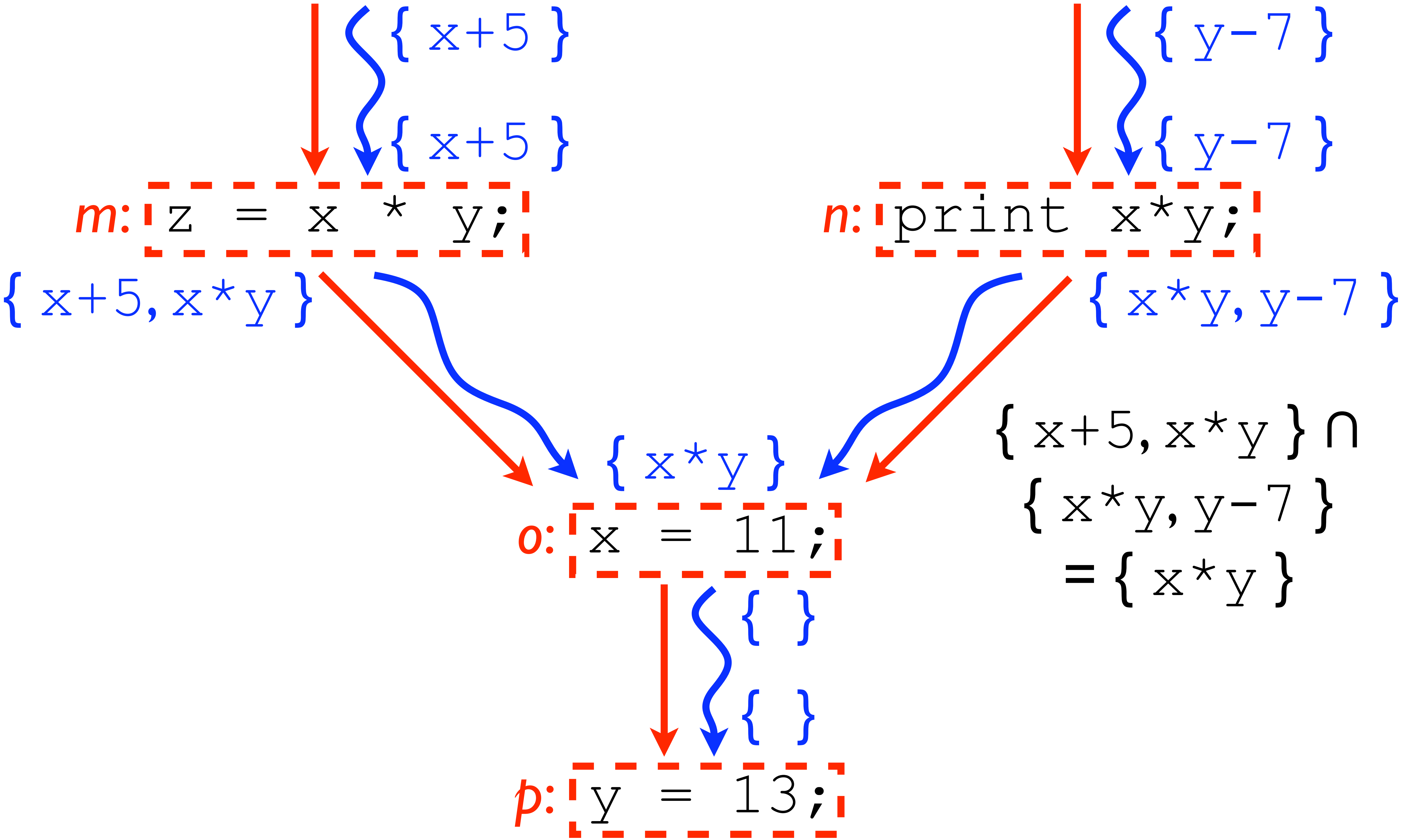
$$\textit{out-avail}(n) = \big(\textit{in-avail}(n) \cup \textit{gen}(n)\big) \setminus \textit{kill}(n)$$

Available expression analysis

When a node *n* has a single predecessor *m*, the information propagates along the control-flow edge as you would expect: *in-avail*(*n*) = *out-avail*(*m*).

When a node has multiple predecessors, the expressions available at the entry of that node are exactly those expressions available at the exit of *all* of its predecessors (cf. "*any* of its successors" in LVA).

Available expression analysis



Available expression analysis

So the following equation must also hold:

$$\mathit{in\text{-}avail}(n) = \bigcap_{p \in \mathit{pred}(n)} \mathit{out\text{-}avail}(p)$$

Data-flow equations

These are the *data-flow equations* for available expression analysis, and together they tell us everything we need to know about how to propagate availability information through a program.

$$\mathit{in\text{-}avail}(n) = \bigcap_{p \in \mathit{pred}(n)} \mathit{out\text{-}avail}(p)$$

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \cup \mathit{gen}(n)\Big) \setminus \mathit{kill}(n)$$

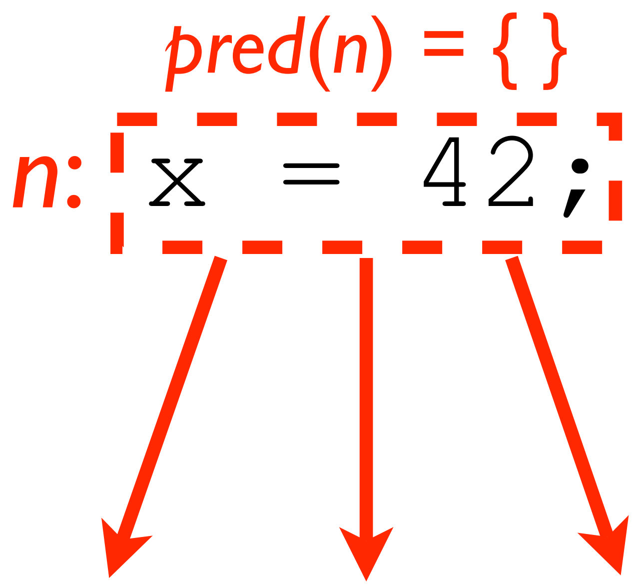
Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall availability equation.

$$avail(n) = \bigcap_{p \in pred(n)} \Big((avail(p) \cup gen(p)) \setminus kill(p) \Big)$$

Data-flow equations

Danger: we have overlooked one important detail.



$$avail(n) = \bigcap_{p \in pred(n)} \left((avail(p) \cup gen(p)) \setminus kill(p)\right)$$

$$= \bigcap \{\,\}$$

$$= U$$ (i.e. *all* expressions in the program)

Clearly there should be *no* expressions available here, so we must stipulate explicitly that $avail(n) = \{\,\}$ if $pred(n) = \{\,\}$.

Data-flow equations

With this correction, our data-flow equation for expression availability is

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \cup gen(p)) \setminus kill(p)) & \text{if } pred(n) \neq \{\,\} \\ \{\,\} & \text{if } pred(n) = \{\,\} \end{cases}$$

Data-flow equations

The functions and equations presented so far are correct, and their definitions are fairly intuitive.

However, we may wish to have our data-flow equations in a form which more closely matches that of the LVA equations, since this emphasises the similarity between the two analyses and hence is how they are most often presented.

A few modifications are necessary to achieve this.

Data-flow equations

$$\mathit{in\text{-}live}(n) = \Big(\mathit{out\text{-}live}(n) \setminus \mathit{def}(n)\Big) \cup \mathit{ref}(n)$$

$$\mathit{out\text{-}live}(n) = \boxed{\bigcup}_{s \in \boxed{\mathit{succ}}(n)} \mathit{in\text{-}live}(s)$$

These differences are inherent in the analyses.

$$\mathit{in\text{-}avail}(n) = \boxed{\bigcap}_{p \in \boxed{\mathit{pred}}(n)} \mathit{out\text{-}avail}(p)$$

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \cup \mathit{gen}(n)\Big) \setminus \mathit{kill}(n)$$

Data-flow equations

$$\textit{in-live}(n) = \Big(\textit{out-live}(n) \setminus \textit{def}(n)\Big) \cup \textit{ref}(n)$$

$$\textit{out-live}(n) = \bigcup_{s \in \textit{succ}(n)} \textit{in-live}(s)$$

These differences are an arbitrary result of our definitions.

$$\textit{in-avail}(n) = \bigcap_{p \in \textit{pred}(n)} \textit{out-avail}(p)$$

$$\textit{out-avail}(n) = \Big(\textit{in-avail}(n) \cup \textit{gen}(n)\Big) \setminus \textit{kill}(n)$$

Data-flow equations

We might instead have decided to define *gen*(*n*) and *kill*(*n*) to coincide with the following (standard) definitions:

- A node *generates* an expression e if it *must* compute the value of e and does not subsequently redefine any of the variables occuring in e.
- A node *kills* an expression e if it *may* redefine some of the variables occurring in e and does not subsequently recompute the value of e.

Data-flow equations

By the old definition:

$gen($ `x = x + y` $) = \{$ `x+y` $\}$

$kill($ `x = x + y` $) = E_x$

By the new definition:

$gen($ `x = x + y` $) = \{\ \}$

$kill($ `x = x + y` $) = E_x$

(The new *kill*(*n*) may visibly differ when *n* is a basic block.)

Data-flow equations

Since these new definitions take account of which expressions are generated *overall* by a node (and exclude those which are generated only to be immediately killed), we may propagate availability information through a node by removing the killed expressions *before* adding the generated ones, *exactly as in LVA*.

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \setminus kill(n)\Big) \cup gen(n)$$

Data-flow equations

From this new equation for *out-avail*(*n*) we may produce our final data-flow equation for expression availability:

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \setminus kill(p)) \cup gen(p)) & \text{if } pred(n) \neq \{\,\} \\ \{\,\} & \text{if } pred(n) = \{\,\} \end{cases}$$

This is the equation you will find in the course notes and standard textbooks on program analysis; remember that it depends on these more subtle definitions of *gen*(*n*) and *kill*(*n*).

Algorithm

- We again use an array, `avail[]`, to store the available expressions for each node.
- We initialise `avail[]` such that each node has *all* expressions available (cf. LVA: *no* variables live).
- We again iterate application of the data-flow equation at each node until `avail[]` no longer changes.

Algorithm

```
for i = 1 to n do avail[i] := U
while (avail[] changes) do
   for i = 1 to n do
```

$$\texttt{avail[i]} := \bigcap_{p \in pred(\texttt{i})} ((\texttt{avail}[p] \setminus kill(p)) \cup gen(p))$$

Algorithm

We can do better if we assume that the flowgraph has a single entry node (the first node in `avail[]`).

Then `avail[1]` may instead be initialised to the empty set, and we need not bother recalculating availability at the first node during each iteration.

Algorithm

```
avail[1] := {}
for i = 2 to n do avail[i] := U
while (avail[] changes) do
   for i = 2 to n do
      avail[i] :=
```

$$\bigcap_{p \in pred(\mathtt{i})} ((\mathtt{avail}[p] \setminus kill(p)) \cup gen(p))$$

Algorithm

As with LVA, this algorithm is guaranteed to terminate since the effect of one iteration is *monotonic* (it only removes expressions from availability sets) and an empty availability set cannot get any smaller.

Any solution to the data-flow equations is safe, but this algorithm is guaranteed to give the *largest* (and therefore most precise) solution.

Algorithm

Implementation notes:

- If we arrange our programs such that each assignment assigns to a distinct temporary variable, we may number these temporaries and hence number the expressions whose values are assigned to them.
- If the program has *n* such expressions, we can implement each element of `avail[]` as an *n*-bit value, with the m^{th} bit representing the availability of expression number *m*.

Algorithm

Implementation notes:

- Again, we can store availability once per basic block and recompute inside a block when necessary. Given each basic block *n* has k_n instructions $n[1], ..., n[k_n]$:

$$avail(n) = \bigcap_{p \in pred(n)} (avail(p) \setminus kill(p[1]) \cup gen(p[1]) \cdots \setminus kill(p[k_p]) \cup gen(p[k_p]))$$