Program Analysis (70020)
While Language

Herbert Wiklicky

Department of Computing
Imperial College London

herbert@doc.ic.ac.uk
h.wiklicky@imperial.ac.uk

Autumn 2021
We use the following syntactic categories:

- $a \in \text{AExp}$ arithmetic expressions
- $b \in \text{BExp}$ boolean expressions
- $S \in \text{Stmt}$ statements
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
  a &::= x | n | a_1 \text{op} a_2 \\
  b &::= \text{true} | \text{false} | \text{not} b | b_1 \text{op} b_2 | a_1 \text{op} r a_2 \\
  S &::= x := a | \text{skip} | S_1 ; S_2 | \text{if} b \text{then} S_1 \text{else} S_2 | \text{while} b \text{do} S_3
\end{align*}
\]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \]
\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op } b_2 \mid a_1 \text{ op } r \ a_2 \]
\[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
a & ::= x \mid n \\
b & \\
S & ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S
\end{align*}
\]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x | n | a_1 \text{ op}_a a_2 \]

\[ b \]

\[ S ::= x := a | \text{skip} | S_1 ; S_2 | \text{if } b \text{ then } S_1 \text{ else } S_2 | \text{while } b \text{ do } S \]
The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \text{ op}_a a_2 \]

\[ b ::= \text{true} \]

\[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \]
Abstract Syntax of \texttt{WHILE}

The syntax of the language \texttt{WHILE} is given by the following abstract syntax:

\begin{align*}
a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
b & ::= \text{true} \mid \text{false} \\
S & \\
\end{align*}
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
a & ::= x | n | a_1 \ op_a a_2 \\
b & ::= \text{true} | \text{false} | \text{not} \ b \\
S &
\end{align*}
\]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \text{ op}_a a_2 \]
\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \]
\[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \]
The syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
    b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
    S & = x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S_3
\end{align*}
\]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \ op_a \ a_2 \]
\[ b ::= \ true \mid \ false \mid \ not \ b \mid b_1 \ op_b \ b_2 \mid a_1 \ op_r \ a_2 \]
\[ S ::= x := a \]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \text{ op}_a a_2 \]
\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \]
\[ S ::= x ::= a \]
\[ \mid \text{skip} \]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x | n | a_1 \text{ op}_a a_2 \]

\[ b ::= \text{true} | \text{false} | \text{not} b | b_1 \text{ op}_b b_2 | a_1 \text{ op}_r a_2 \]

\[ S ::= x ::= a \]

\[ | \text{skip} \]

\[ | S_1; S_2 \]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
b & ::= \text{true} \mid \text{false} \mid \text{not} \ b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
S & ::= x := a \\
    & \mid \text{skip} \\
    & \mid S_1 ; S_2 \\
    & \mid \text{if} \ b \ \text{then} \ S_1 \ \text{else} \ S_2
\end{align*}
\]
Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \text{ op}_a a_2 \]
\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \]
\[ S ::= x := a \]
\[ \mid \text{skip} \]
\[ \mid S_1 ; S_2 \]
\[ \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \]
\[ \mid \text{while } b \text{ do } S \]
Syntactical Categories

We assume some countable/finite set of variables is given;

\[
\begin{align*}
    x, y, z, \ldots & \in \text{Var} \quad \text{variables} \\
n, m, \ldots & \in \text{Num} \quad \text{numerals}
\end{align*}
\]
Syntactical Categories

We assume some countable/finite set of variables is given;

\[ x, y, z, \ldots \in \text{Var} \quad \text{variables} \]
\[ n, m, \ldots \in \text{Num} \quad \text{numerals} \]
\[ \ell, \ldots \in \text{Lab} \quad \text{labels} \]

Numerals (integer constants) will not be further defined and neither will the operators:

\[ op_a \in \text{Op}_a \quad \text{arithmetic operators, e.g. } +, -, \times, \ldots \]
\[ op_b \in \text{Op}_b \quad \text{boolean operators, e.g. } \land, \lor, \ldots \]
\[ op_r \in \text{Op}_r \quad \text{relational operators, e.g. } =, <, \leq, \ldots \]
The labelled syntax of the language WHILE is given by the following abstract syntax:

\[ a \]
\[ b \]
\[ S \]
The labelled syntax of the language \texttt{WHILE} is given by the following abstract syntax:

\begin{align*}
a & ::= \ x \\
b & \\
S &
\end{align*}
The labelled syntax of the language **WHILE** is given by the following **abstract syntax**:

\[
\begin{align*}
  a &::= x \mid n \\
  b &
\end{align*}
\]

\[
\begin{align*}
  S &::= \begin{cases} x := a \mid \ell & \\
  \text{skip} \mid \ell & \\
  S_1 ; S_2 & \\
  \text{if } b \text{ then } S_1 \text{ else } S_2 & \\
  \text{while } b \text{ do } S &
\end{cases}
\end{align*}
\]
The labelled syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \text{op}_a a_2 \\
b \\
S & ::= \llbracket x := a \rrbracket \ell \mid \llbracket \text{skip} \rrbracket \ell \mid S_1 ; S_2 \mid \text{if} \llbracket b \rrbracket \ell \text{then } S_1 \text{else } S_2 \mid \text{while} \llbracket b \rrbracket \ell \text{do } S \end{align*}
\]
Labelled Syntax of \textsc{While}

The \textbf{labelled} syntax of the language \textsc{While} is given by the following \textbf{abstract syntax}:

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
    b & ::= \text{true} \\
    S &
\end{align*}
\]
The **labelled** syntax of the language **WHILE** is given by the following **abstract syntax**:

\[
\begin{align*}
a & ::= \ x \mid n \mid a_1 \ op_a \ a_2 \\
b & ::= \ true \mid false \\
S & 
\end{align*}
\]
Labelled Syntax of WHILE

The *labelled* syntax of the language *WHILE* is given by the following *abstract syntax*:

\[
\begin{align*}
    a &::= \ x \mid n \mid a_1 \op a \ a_2 \\
    b &::= \ true \mid false \mid not \ b \\
    S &\\
\end{align*}
\]
Labelled Syntax of WHILE

The labelled syntax of the language WHILE is given by the following abstract syntax:

\[
\begin{align*}
ap & ::= \ x \mid n \mid a_1 \ op_a \ a_2 \\
bp & ::= \ true \mid false \mid not \ b \mid b_1 \ op_b \ b_2 \\
S &
\end{align*}
\]
The **labelled** syntax of the language **WHILE** is given by the following **abstract syntax**:

\[
\begin{align*}
a & ::= x | n | a_1 \text{ op}_a a_2 \\
b & ::= \text{true} | \text{false} | \text{not } b | b_1 \text{ op}_b b_2 | a_1 \text{ op}_r a_2 \\
S & \\
\end{align*}
\]
Labelled Syntax of \textsc{While}

The \textit{labelled} syntax of the language \textsc{While} is given by the following \textbf{abstract syntax}:\

\begin{align*}
a &::= x \mid n \mid a_1 \mathit{op}_a a_2 \\
b &::= \mathit{true} \mid \mathit{false} \mid \mathit{not} \ b \mid b_1 \mathit{op}_b b_2 \mid a_1 \mathit{op}_r a_2 \\
S &::= [x := a]^\ell
\end{align*}
The labelled syntax of the language \textsc{While} is given by the following \textbf{abstract syntax}:

\begin{align*}
a & ::= x \mid n \mid a_1 \, \text{op}_a \, a_2 \\
b & ::= \text{true} \mid \text{false} \mid \text{not} \, b \mid b_1 \, \text{op}_b \, b_2 \mid a_1 \, \text{op}_r \, a_2 \\
S & ::= [ x := a ]^l \\
     & \mid [ \text{skip} ]^l
\end{align*}
The **labelled** syntax of the language **WHILE** is given by the following **abstract syntax**:

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
S & ::= [x := a]^\ell \\
    & \mid \text{skip}^\ell \\
    & \mid S_1;S_2
\end{align*}
\]
The labelled syntax of the language WHILE is given by the following abstract syntax:

\[ a ::= x \mid n \mid a_1 \text{ op}_a a_2 \]

\[ b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \]

\[ S ::= [x := a]^\ell \]

\[ \mid [\text{skip}]^\ell \]

\[ \mid S_1 ; S_2 \]

\[ \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \]
The **labelled** syntax of the language **WHILE** is given by the following **abstract syntax**:

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 \; op_a \; a_2 \\
    b & ::= true \mid false \mid not \; b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2 \\
    S & ::= [x := a]^\ell \\
        & \mid [skip]^\ell \\
        & \mid S_1;S_2 \\
        & \mid if \; [b]^\ell \; then \; S_1 \; else \; S_2 \\
        & \mid while \; [b]^\ell \; do \; S
\end{align*}
\]
An Example in WHILE

An example of a program written in this WHILE language is the following one which computes the factorial of the number stored in \( x \) and leaves the result in \( z \):

\[
\begin{align*}
y &:= x \quad 1 \\
z &:= 1 \quad 2 \\
y &:= y - 1 \quad 5 \\
y &:= 0 \quad 6 \\
z &:= z \times y \quad 4 \\
\end{align*}
\]
An Example in WHILE

An example of a program written in this WHILE language is the following one which computes the factorial of the number stored in $x$ and leaves the result in $z$:

\[
\begin{align*}
[y &:= x]^{1}; \\
[z &:= 1]^{2}; \\
\textbf{while} [y > 1]^{3} \textbf{ do } ( \\
[z &:= z \ast y]^{4}; \\
[y &:= y - 1]^{5}); \\
[y &:= 0]^{6}
\end{align*}
\]
An Example in WHILE

An example of a program written in this WHILE language is the following one which computes the factorial of the number stored in $x$ and leaves the result in $z$:

\[
\begin{align*}
[y := x] &^1; \\
[z := 1] &^2; \\
\text{while } [y > 1] &^3 \text{ do (} \\
[ z := z \times y ] &^4; \\
[ y := y - 1 ] &^5); \\
[y := 0] &^6
\end{align*}
\]

Note the use of meta-symbols, brackets, to group statements.
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language WHILE as follows:

\[
a
\]

\[
b
\]

\[
S
\]
Concrete Syntax of \texttt{WHILE}

To avoid using brackets (as meta-symbols) we could also use the \texttt{concrete syntax} of the language \texttt{WHILE} as follows:

\[
\begin{align*}
    a & ::= x \\
    b & \\
    S & \\
\end{align*}
\]
Concrete Syntax of \textsc{While}

To avoid using brackets (as meta-symbols) we could also use the \textbf{concrete syntax} of the language \textsc{While} as follows:

\[
\begin{align*}
  a & ::= x \mid n \\
  b & \\
  S & 
\end{align*}
\]
Concrete Syntax of \texttt{WHILE}

To avoid using brackets (as meta-symbols) we could also use the \textit{concrete syntax} of the language \texttt{WHILE} as follows:

\begin{align*}
    a & ::= \ x \mid n \mid a_1 \ \text{op}_a \ a_2 \\
    b & \\
    S & 
\end{align*}
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language WHILE as follows:

\[
\begin{align*}
  a &::= x | n | a_1 \, op_a \, a_2 \\
  b &::= true \\
  S &::= x := a | \text{skip} | S_1 ; S_2 | \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi } | \text{while } b \text{ do } S \text{ od}
\end{align*}
\]
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the concrete syntax of the language WHILE as follows:

\[
\begin{align*}
   a & ::= x \mid n \mid a_1 \textit{ op}_a a_2 \\
   b & ::= \textit{true} \mid \textit{false} \\
   S & 
\end{align*}
\]
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language WHILE as follows:

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
    b & ::= \text{true} \mid \text{false} \mid \text{not } b \\
    S & \\
\end{align*}
\]
Concrete Syntax of \texttt{WHILE}

To avoid using brackets (as meta-symbols) we could also use the \textbf{concrete syntax} of the language \texttt{WHILE} as follows:

\[
\begin{align*}
  a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
  b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \\
  S &
\end{align*}
\]
Concrete Syntax of \textsc{WHILE}

To avoid using brackets (as meta-symbols) we could also use the \textbf{concrete syntax} of the language \textsc{WHILE} as follows:

\begin{align*}
a & ::= \ x | \ n | \ a_1 \ op_a \ a_2 \\
b & ::= \ true | \ false | \ not \ b | \ b_1 \ op_b \ b_2 | \ a_1 \ op_r \ a_2 \\
S &
\end{align*}
Concrete Syntax of **WHILE**

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language **WHILE** as follows:

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
    b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
    S & ::= x := a
\end{align*}
\]
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language WHILE as follows:

\[
\begin{align*}
    a &::= x \mid n \mid a_1 \text{ op}_a a_2 \\
    b &::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
    S &::= x := a \\
        &\quad | \text{skip}
\end{align*}
\]
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the concrete syntax of the language WHILE as follows:

\[\begin{align*}
a & ::= \ x \ | \ n \ | \ a_1 \ op_a \ a_2 \\
b & ::= \ true \ | \ false \ | \ not \ b \ | \ b_1 \ op_b \ b_2 \ | \ a_1 \ op_r \ a_2 \\
S & ::= \ x := a \\
& \ | \ skip \\
& \ | \ S_1;S_2
\end{align*}\]
Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the concrete syntax of the language WHILE as follows:

\[
\begin{align*}
  a &::= x \mid n \mid a_1 \text{ op}_a a_2 \\
  b &::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
  S &::= x := a \\
  &\quad | \text{skip} \\
  &\quad | S_1;S_2 \\
  &\quad | \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}
\end{align*}
\]
Concrete Syntax of \texttt{WHILE}

To avoid using brackets (as meta-symbols) we could also use the \textit{concrete syntax} of the language \texttt{WHILE} as follows:

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\
b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\
S & ::= x ::= a \\
& \mid \text{skip} \\
& \mid S_1;S_2 \\
& \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
& \mid \text{while } b \text{ do } S \text{ od}
\end{align*}
\]
When presenting examples of Data Flow Analyses we will use a number of operations on programs and labels. The first of these is

\[ \text{init} : \text{Stmt} \rightarrow \text{Lab} \]

which returns the initial label of a statement:

\[
\begin{align*}
\text{init}(\left[ x := a \right]^\ell) &= \ell \\
\text{init}(\left[ \text{skip} \right]^\ell) &= \ell \\
\text{init}(S_1;S_2) &= \text{init}(S_1) \\
\text{init}(\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2) &= \ell \\
\text{init}(\text{while } [b]^\ell \text{ do } S) &= \ell
\end{align*}
\]
Final Labels

We will also need a function which returns the set of final labels in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (e.g. in the conditional):

\[
\text{final : Stmt} \rightarrow \mathcal{P}(\text{Lab})
\]

- \(\text{final}([\ x := a]\) = \{\ell\}
- \(\text{final}([\ \text{skip}\ ]\) = \{\ell\}
- \(\text{final}(S_1;S_2) = \text{final}(S_2)\)
- \(\text{final}(\text{if }[b]\ \text{then } S_1 \text{ else } S_2) = \text{final}(S_1) \cup \text{final}(S_2)\)
- \(\text{final}(\text{while }[b]\ \text{do } S) = \{\ell\}\)

The \textbf{while}-loop terminates immediately after the test fails.
The building blocks of our analysis is given by **Block** is the set of statements, or elementary blocks, of the form:
The building blocks of our analysis is given by **Block** is the set of statements, or elementary blocks, of the form:

- $[ x := a ]^\ell$, or
Elementary Blocks

The building blocks of our analysis is given by Block is the set of statements, or elementary blocks, of the form:

- $[x := a]$, or
- $[\text{skip}]$, as well as
The building blocks of our analysis is given by Block is the set of statements, or elementary blocks, of the form:

- \([ x := a ]^\ell\), or
- \([ \text{skip} ]^\ell\), as well as
- tests of the form \([b]^\ell\).
Blocks

To access the statements or test associated with a label in a program we use the function

$$blocks : \textbf{Stmt} \rightarrow \mathcal{P}(\textbf{Block})$$

- $blocks([\ x := a \ ]^\ell) = \{[\ x := a \ ]^\ell\}$
- $blocks([\ \text{skip} \ ]^\ell) = \{[\ \text{skip} \ ]^\ell\}$
- $blocks(S_1;S_2) = blocks(S_1) \cup blocks(S_2)$
- $blocks(\text{if } [b]^\ell \ \text{then } S_1 \ \text{else } S_2) = \{[b]^\ell\} \cup blocks(S_1) \cup blocks(S_2)$
- $blocks(\text{while } [b]^\ell \ \text{do } S) = \{[b]^\ell\} \cup blocks(S)$
Then the set of labels occurring in a program is given by

\[
\text{labels : Stmt} \rightarrow \mathcal{P}(\text{Lab})
\]

where

\[
\text{labels}(S) = \{ \ell \mid [B]^\ell \in \text{blocks}(S) \}
\]

Clearly \( \text{init}(S) \in \text{labels}(S) \) and \( \text{final}(S) \subseteq \text{labels}(S) \).
Flow

\[ \text{flow} : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab} \times \text{Lab}) \]

which maps statements to sets of flows:

\[
\begin{align*}
\text{flow([ } x := a ]^{\ell}) &= \emptyset \\
\text{flow([ skip ]^{\ell})} &= \emptyset \\
\text{flow}(S_1;S_2) &= \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
& \quad \{ (\ell, \text{init}(S_2)) \mid \ell \in \text{final}(S_1) \} \\
\text{flow(if [ } b ]^{\ell} \text{ then } S_1 \text{ else } S_2) &= \text{flow}(S_1) \cup \text{flow}(S_2) \cup \\
& \quad \{ (\ell, \text{init}(S_1)), (\ell, \text{init}(S_2)) \} \\
\text{flow(while [ } b ]^{\ell} \text{ do } S) &= \text{flow}(S) \cup \{ (\ell, \text{init}(S)) \} \cup \\
& \quad \{ (\ell', \ell) \mid \ell' \in \text{final}(S) \}
\end{align*}
\]
Consider the following program, power, computing the \( x \)-th power of the number stored in \( y \):

\[
\begin{align*}
\text{[ } & z := 1 ]^1; \\
\text{while } & [ x > 1 ]^2 \text{ do (} \\
& \quad \text{[ } z := z \ast y ]^3; \\
& \quad \text{[ } x := x - 1 ]^4); \\
\end{align*}
\]
An Example Flow

Consider the following program, \textit{power}, computing the $x$-th power of the number stored in $y$:

\[
\begin{align*}
[ & z := 1 ]^1; \\
\text{while } [ & x > 1]^2 \text{ do (} \\
[ & z := z \ast y ]^3; \\
[ & x := x - 1 ]^4); \\
\end{align*}
\]

We have $\text{labels}(\text{power}) = \{1, 2, 3, 4\}$, $\text{init}(\text{power}) = 1$, and $\text{final}(\text{power}) = \{2\}$. The function \textit{flow} produces the set:

\[
\text{flow}(\text{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}
\]
Flow Graph

```
x := x - 1
```

```
z := z * y
```

```
x > 0
```

```
z := 1
```

Diagram:
- [z := 1]¹
- [x > 0]²
  - yes → [z := z * y]³
  - no
- [x := x - 1]⁴
Forward Analysis

The function $flow$ is used in the formulation of forward analyses. Clearly $init(S)$ is the (unique) entry node for the flow graph with nodes $labels(S)$ and edges $flow(S)$. Also

$$
labels(S) = \{ init(S) \} \cup \{ \ell \mid (\ell, \ell') \in flow(S) \} \cup \{ \ell' \mid (\ell, \ell') \in flow(S) \}
$$

and for composite statements (meaning those not simply of the form $[B]^{\ell}$) the equation remains true when removing the $\{ init(S) \}$ component.
In order to formulate *backward analyses* we require a function that computes reverse flows:

\[
flow^R : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab} \times \text{Lab})
\]

\[
flow^R(S) = \{(\ell, \ell') \mid (\ell', \ell) \in flow(S)\}
\]

For the power program, \(flow^R\) produces

\[
\{(2, 1), (2, 4), (3, 2), (4, 3)\}
\]
Backward Analysis

In case $\text{final}(S)$ contains just one element that will be the unique entry node for the flow graph with nodes $\text{labels}(S)$ and edges $\text{flow}^R(S)$. Also

$$\text{labels}(S) = \text{final}(S) \cup \{\ell \mid (\ell, \ell') \in \text{flow}^R(S)\} \cup \{\ell' \mid (\ell, \ell') \in \text{flow}^R(S)\}$$
We will use the notation $S_\star$ to represent the program we are analysing (the “top-level" statement)
We will use the notation $S_\star$ to represent the program we are analysing (the “top-level" statement) and furthermore:

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$, 
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$, 
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) occurring in $S_\star$, and 
- $\text{AExp}_\star$ to represent the set of non-trivial arithmetic subexpressions in $S_\star$ as well as $\text{AExp}(a)$ and $\text{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

An expression is trivial if it is a single variable or constant.
Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level" statement) and furthermore:

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$,
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$. 

An expression is trivial if it is a single variable or constant.
Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level" statement) and furthermore:

- **Lab$\star$** to represent the labels ($labels(S_\star)$) appearing in $S_\star$,
- **Var$\star$** to represent the variables ($FV(S_\star)$) appearing in $S_\star$,
- **Block$\star$** to represent the elementary blocks ($blocks(S_\star)$) occurring in $S_\star$, and

An expression is trivial if it is a single variable or constant.
Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level” statement) and furthermore:

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$,
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$,
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) occurring in $S_\star$, and
- $\text{AExp}_\star$ to represent the set of non-trivial arithmetic subexpressions in $S_\star$
Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level” statement) and furthermore:

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$,
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$,
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) occurring in $S_\star$, and
- $\text{AExp}_\star$ to represent the set of non-trivial arithmetic subexpressions in $S_\star$

An expression is trivial if it is a single variable or constant.
Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level” statement) and furthermore:

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$, 
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$, 
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) occurring in $S_\star$, and 
- $\text{AExp}_\star$ to represent the set of non-trivial arithmetic subexpressions in $S_\star$ as well as 
- $\text{AExp}(a)$ and $\text{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

An expression is trivial if it is a single variable or constant.
Isolated Entries & Exits

Program $S_\star$ has *isolated entries* if:

$$\forall \ell \in \text{Lab} : (\ell, \text{init}(S_\star)) \notin \text{flow}(S_\star)$$

This is the case whenever $S_\star$ does not start with a *while*-loop.
Program $S_\star$ has \textit{isolated entries} if:

$$\forall \ell \in \text{Lab} : (\ell, \text{init}(S_\star)) \not\in \text{flow}(S_\star)$$

This is the case whenever $S_\star$ does not start with a \texttt{while}-loop.

Similarly, we shall frequently assume that the program $S_\star$ has \textit{isolated exits}; this means that:

$$\forall \ell_1 \in \text{final}(S_\star) \forall \ell_2 \in \text{Lab} : (\ell_1, \ell_2) \not\in \text{flow}(S_\star)$$
A statement, $S$, is **label consistent** if and only if:

$$[B_1]^\ell, [B_2]^\ell \in blocks(S) \text{ implies } B_1 = B_2$$
Label Consistency

A statement, $S$, is label consistent if and only if:

$$[B_1]^{\ell}, [B_2]^{\ell} \in \text{blocks}(S) \text{ implies } B_1 = B_2$$

Clearly, if all blocks in $S$ are uniquely labelled (meaning that each label occurs only once), then $S$ is label consistent.

When $S$ is label consistent the statement or clause “where $[B]^{\ell} \in \text{blocks}(S)$” is unambiguous in defining a partial function from labels to elementary blocks; we shall then say that $\ell$ labels the block $B$. 