Program Analysis (70020)
While Language

Herbert Wiklicky

Department of Computing
Imperial College London

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

Autumn 2022
Syntactic Constructs

We use the following syntactic categories:

\[ a \in \text{AExp} \quad \text{arithmetic expressions} \]
\[ b \in \text{BExp} \quad \text{boolean expressions} \]
\[ S \in \text{Stmt} \quad \text{statements} \]
Abstract Syntax of WHILE

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

\[ a \]
\[ b \]
\[ S \]
Abstract Syntax of WHILE

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

\[ a ::= x \]

\[ b = \text{true} \quad \text{or} \quad \text{false} \quad \text{or} \quad \text{not} \quad b \quad \text{or} \quad b_1 \quad \text{op} \quad b_2 \quad \text{or} \quad a_1 \quad \text{op} \quad r \quad a_2 \]

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

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

\[ a ::= x | n \]
\[ b ::= true | false | \neg b | b_1 \mathbin{op} b_2 | a_1 \mathbin{op} r a_2 \]
\[ S ::= x := a | \text{skip} | S_1 ; S_2 | \text{if} b \text{then} S_1 \text{else} S_2 | \text{while} b \text{do} S \]


Abstract Syntax of WHILE

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

\[ a ::= x \mid n \mid a_1 \ op_a \ a_2 \]

\[ b \]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The syntax of the language \textbf{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 \texttt{WHILE}

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

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

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

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

\[
a \ ::= \ x
\]

\[
b
\]

\[
S
\]
The labelled syntax of the language WHILE is given by the following abstract syntax:

\[
a ::= x | n
\]

\[
b
\]

\[
S ::= \{x := a\} \ell \mid \{\text{skip}\} \ell \mid S_1 ; S_2 \mid \text{if} \{b\} \ell \text{then} S_1 \text{else} S_2 \mid \text{while} \{b\} \ell \text{do} S_5
\]
The labelled syntax of the language \textsc{While} is given by the following \textbf{abstract syntax}:

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \ op_a \ a_2 \\
b \\
S & ::= \llbracket x := a \rrbracket \ell \mid \llbracket \text{skip} \rrbracket \ell \mid S_1 ; S_2 \mid \text{if} \llbracket b \rrbracket \ell \text{then} S_1 \text{else} S_2 \mid \text{while} \llbracket b \rrbracket \ell \text{do} S_5
\end{align*}
\]
The labelled syntax of the language WHILE is given by the following abstract syntax:

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

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

\begin{align*}
a & ::= x \mid n \mid a_1 \ op_a \ a_2 \\
b & ::= \text{true} \mid \text{false} \\
S & \\
\end{align*}
The labelled syntax of the language \texttt{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 \\
S &
\end{align*}
Labelled Syntax of WHILE

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

\[ a ::= x | n | a_1 \text{ op}_a a_2 \]
\[ b ::= \text{true} | \text{false} | \text{not} b | b_1 \text{ op}_b b_2 \]
\[ S ::= [x := a] \ell | [\text{skip}] \ell | S_1 ; S_2 | \text{if } [b] \ell \text{ then } S_1 \text{ else } S_2 | \text{while } [b] \ell \text{ do } S_5 \]

5/21
Labelled Syntax of \textsc{While}

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

\[
\begin{align*}
  a & ::= \ x | \ n | \ a_1 \ op_a \ a_2 \\
  b & ::= \ true | \ false | \ not \ b | \ b_1 \ op_b \ b_2 | \ a_1 \ op_r \ a_2 \\
  S & = \ [ x \ := a ] \ \ell | \ [ \text{skip} ] \ \ell | \ S_1 ; \ S_2 | \text{if} \ [b] \ \ell \ \text{then} \ S_1 \ \text{else} \ S_2 | \text{while} \ [b] \ \ell \ \text{do} \ S
\end{align*}
\]
The labelled syntax of the language **WHILE** is given by the following *abstract syntax*:

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

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \ op_a \ a_2 \\
b & ::= \text{true} \mid \text{false} \mid \text{not} \ b \mid b_1 \ op_b \ b_2 \mid a_1 \ op_r \ a_2 \\
S & ::= [x := a]^l \\
& \quad \mid [\text{skip}]^l
\end{align*}
\]
The labelled syntax of the language \texttt{WHILE} is given by the following \textbf{abstract syntax}:

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

The \textit{labelled} syntax of the language \texttt{WHILE} is given by the following \textbf{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]^l \\
& \quad \mid [\text{skip}]^l \\
& \quad \mid S_1;S_2 \\
& \quad \mid \text{if } [b]^l \ \text{then } S_1 \ \text{else } S_2
\end{align*}
Labelled Syntax of WHILE

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

\[
\begin{align*}
a & ::= x | n | a_1 \text{ op}_a a_2 \\
b & ::= \text{true} | \text{false} | \text{not} b | b_1 \text{ op}_b b_2 | a_1 \text{ op}_r a_2 \\
S & ::= [x := a]^l \\
    & | [\text{skip}]^l \\
    & | S_1 ; S_2 \\
    & | \text{if } [b]^l \text{ then } S_1 \text{ else } S_2 \\
    & | \text{while } [b]^l \text{ do } S
\end{align*}
\]
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 \texttt{x} and leaves the result in \texttt{z}:

\begin{verbatim}
y := \texttt{x};
z := 1;
while \texttt{y > 1} do
  \(z := z \times y;\)
  \(y := y - 1;\)
y := 0;
\end{verbatim}
An Example in \texttt{WHILE}

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

\[
\begin{array}{l}
[ \ y := x \ ];^1 \\
[ \ z := 1 \ ];^2 \\
\textbf{while} \ [ y > 1 ]^3 \ \textbf{do} ( \\
\quad [ \ z := z * y ];^4 \\
\quad [ \ y := y - 1 ];^5 \\
\quad [ \ y := 0 ];^6
\end{array}
\]
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 \( z \):

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

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:

\[
\begin{align*}
  a \\
  b \\
  S
\end{align*}
\]
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 WHILE

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

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

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

\[
\begin{align*}
a & \ ::= \ x \mid n \mid a_1 \, op_a \, a_2 \\
b \\
S \\
\end{align*}
\]
Concrete Syntax of \textsc{While}

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

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

$$b ::= \text{true}$$

$$S$$
Concrete Syntax of WHILE

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

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \ op_a \ a_2 \\
b & ::= true \mid false \\
S & 
\end{align*}
\]
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 \mid a_1 \, \text{op}_a \, a_2 \\
b &::= \text{true} \mid \text{false} \mid \text{not} \ 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:

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

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

\[
\begin{align*}
a & ::= x \mid n \mid a_1 \ op_a \ a_2 \\
b & ::= true \mid false \mid not \ b \mid b_1 \ op_b \ b_2 \mid a_1 \ op_r \ a_2 \\
S &
\end{align*}
\]
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 \texttt{WHILE}

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

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

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

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

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

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

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

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

The \textbf{while}-loop terminates immediately after the test fails.
Elementary Blocks

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:

- $[\text{x} := a]^\ell$, or
- $[\text{skip}]^\ell$, as well as
Elementary Blocks

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

- $[x := a]^{\ell}$, or
- $[\text{skip}]^{\ell}$, as well as
- tests of the form $[b]^{\ell}$. 
To access the statements or test associated with a label in a program we use the function

\[
\text{blocks} : \text{Stmt} \rightarrow \mathcal{P}(\text{Block})
\]

\[
\text{blocks}([ x := a ]^\ell) = \{[ x := a ]^\ell\}
\]

\[
\text{blocks}([ \text{skip } ]^\ell) = \{[ \text{skip } ]^\ell\}
\]

\[
\text{blocks}(S_1; S_2) = \text{blocks}(S_1) \cup \text{blocks}(S_2)
\]

\[
\text{blocks}(\text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2) = \{[b]^\ell\} \cup \text{blocks}(S_1) \cup \text{blocks}(S_2)
\]

\[
\text{blocks}(\text{while } [b]^\ell \text{ do } S) = \{[b]^\ell\} \cup \text{blocks}(S)
\]
Then the set of labels occurring in a program is given by

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

where

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

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

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

which maps statements to sets of flows:

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

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

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

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

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

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

\[
\begin{align*}
[z := 1] & \quad (1) \\
[x > 0] & \quad (2) \\
[z := z \cdot y] & \quad (3) \\
[x := x - 1] & \quad (4)
\end{align*}
\]
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.
Reverse Flow

In order to formulate *backward analyses* we require a function that computes reverse flows:

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

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

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

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

In case final(S) contains just one element that will be the unique entry node for the flow graph with nodes labels(S) and edges 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)
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 $S_\star$,
**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 $S_\star$, and
- **AExp_\star** to represent the set of non-trivial arithmetic subexpressions in $S_\star$ as well as $AExp(a)$ and $AExp(b)$ 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 $S_\star$,
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) 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 $S_\star$. 

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

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

$$\forall \ell_1 \in \text{final}(S_\star) \ \forall \ell_2 \in \text{Lab} : (\ell_1, \ell_2) \notin \text{flow}(S_\star)$$
Label Consistency

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

$$[B_1]^\ell, [B_2]^\ell \in blocks(S) \text{ implies } B_1 = B_2$$
A statement, $S$, is label consistent if and only if:

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

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 $B$. 