# Flow Language Reference Manual

PLT 4115 Fall 2015

Mitchell Gouzenko (mag2272), Zachary Gleicher (zjg2012)
Adam Chelminski (apc2142), Hyonjee Joo (hj2339)

# 1 Introduction

Flow is a language that aims to process streams of input using the Kahn Process Network (KPN) model through a variety of user-defined transformations. As the name of the language suggests, the goal of Flow is to enable cascading of data over seamless sequences of operations and functions. Data passes through processes, each of which have a well-defined input and output protocol. Using channels, Flow makes it intuitive to connect processes with each other. Flow is compiled into multithreaded C code.

# 2 Lexical Conventions

## 2.1 Identifiers

An identifier is a name, consisting of ASCII letters, digits, and '_' characters. The first character of the identifier must be either a letter or '_'. Identifiers are case-sensitive. Below are the parsing rules for an identifier:

```
IDENTIFIER :=
    ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

## 2.2 Key Words

Keywords are reserved; they have syntactic and semantic purposes and thus cannot be used as identifiers. The keywords in flow are:

| | |
|---|---|
| `int` | `if` |
| `double` | `else` |
| `char` | `continue` |
| `bool` | `return` |
| `proc` | `in` |
| `for` | `out` |
| `while` | `break` |
| `channel` | `list` |
| `string` | `struct` |
| `void` | `true` |
| `false` | |

## 2.3 Punctuation

The punctuators of our language are listed below. Their specific uses will be described in more detail throughout the manual.

| Punctuator | Use | Example |
|---|---|---|
| `,` | list separator | `stuff_list = [0, 1, 2];` |
| `[]` | list delimiter, indexing | `stuff_list = [1]; list[0];` |
| `()` | conditional delimiter, function call, expression grouping | `while (bool)` |
| `{}` | statement block | `if (cond) { statements }` |
| `;` | statement end | `x = 0;` |
| `'` | character literal delimiter | `x = 'h';` |
| `"` | string literal delimiter | `x = "hello";` |

## 2.4 Comments

The character `'#'` introduces a new comment. The rest of the line after `'#'` will be part of a comment.

## 2.5 Operators

| Operator | Use | Associativity |
|---|---|---|
| `.` | Method Application | Left |
| `@` | Retrieve item from channel | Left |
| `*` | Multiplication | Left |
| `/` | Division | Left |
| `%` | Modulo | Left |
| `+` | Addition | Left |
| `-` | Subtraction | Left |
| `==` | Equal to | Non-associative |
| `!=` | Not equal to | Non-associative |
| `<` | Less than | Non-associative |

| | | |
|---|---|---|
| > | Greater Than | Non-associative |
| <= | Less than or equal to | Non-associative |
| >= | Greater than or equal to | Non-associative |
| && | short circuit logical AND | Left |
| \|\| | short circuit logical OR | Left |
| & | bitwise AND | Right |
| \| | bitwise inclusive OR | Right |
| ^ | bitwise XOR | Right |
| << | left shift | Right |
| >> | right shift | Right |
| ~ | bitwise NOT | Right |
| ! | negation | Non-associative |
| = | Assignment | Right |
| -> | Send item to channel | Left |

### 2.5.1 Operator Precedence

The operators are listed from greatest to least precedence:

1. .
2. @ ->
3. ~ !
4. * / %
5. + -
6. << >>
7. < > <= >=
8. == !=
9. & ^ |
10. && ||
11. =

## 2.6 Whitespace

White spaces include blanks, tabs, and newline characters. Flow is not whitespace sensitive. Blocks of code are delimited by curly braces, not indentation.

## 2.7 Literals

There are literals for integers, doubles, booleans, characters, and strings. Descriptions of what these literals can contain are in the next section.

# 3 Types

The examples in this section assume that the relevant identifiers were previously declared. Read about declaration in 4 Declarations.

## 3.1 Primitive Types

### 3.1.1 Integer Type

An integer is a signed 4 byte sequence of digits. An integer literal is a sequence of digits preceded by an optional negative sign. A single zero cannot be preceded by a negative sign.

```
x = 0;
y = -1;
z = 100;
```

### 3.1.2 Double Type

A double type is a signed 8 byte double-precision floating point data type consisting. A double literal contains an optionally signed integer part, a decimal point and a fractional part. Either the integer part or the fractional part can be missing, but not both.

```
a = 0.1;
b = -1.1;
i = 1.;
j = .2;
```

### 3.1.3 Boolean Type

A boolean literal is either the `true` keyword or the `false` keyword, and occupies 1 byte. A boolean is its own type and cannot be compared to a non-boolean variable. Therefore, evaluating false == 0, would result in an error.

```
x = true;
y = false;
```

### 3.1.4 Character Types

A character is a 1 byte data type, and encodes ASCII characters as numbers. A character literal is a single ASCII character enclosed by single quotes.

```
c = 'x';
d = 'd' + 2; # d is equal to 'f'
```

### 3.1.5 Void Type

The `void` type can be used to declare a function that does not return anything. It has no other use in the Flow language.

## 3.2 Non-Primitive Types

In Flow there are 5 non-primitive types: strings, lists, structs, channels, and processes.

### 3.2.1 String Type

A string is a sequence of characters. A string literal is placed between double quotes. String literals are sequences of ASCII characters, enclosed by double quotes. Strings are immutable. Declared strings are automatically initialized to the empty string `""`;

```
name = "Steven";
```

Strings support the following built-in functions:

- `length(string a)`
    - Returns the length of the string as an integer.
- `string[index n]`
    - Returns the character at index n. Returns an error if index is out of bounds.
- `strCpy(string a)`
    - Returns a new copy of the string.

```
string name;

name = "Steven";

int steven_length = length(name); # sets steven_length to 6
int sarah_length = length("sarah"); # sets sarah_length to 5
```

```
string name = "Steven";

char c = name[0];

c == 'S' // Evaluates to true
```

### 3.2.2 List Type

A list is a mutable, sequential collection of elements of the same type.

```
y = ['a', 'b', 'c'];
```

Lists support the following built-in functions:

- `append(list a, element x)`
  - Appends an element to the end of the list.
- `pop(list a)`
  - Removes and returns the last element of the list.
- `length(list a)`
  - Returns the length of a list as an integer.
- Lists can be indexed using the `[]` operator. This returns the element at this index.

```
char list y = ['a', 'b', 'c];
int x = length(y);
char foo = pop(y); # Pops 'c'. Y is now ['a', 'b']
append(y, foo);      # Put 'c' back. Y is now ['a', 'b', 'c']
```

### 3.2.3 Struct Type

A struct is a data type that allows for a programmer to define a grouping of various primitive and non-primitive values that can easily be stored into a single variable—the instance of the struct.

Data members in a struct instance be accessed using the `'.'` operator. Data within a struct does not need to be initialized.

```
struct Person = {
     string name;
     int age;
}
```

```
Person firstPerson;

firstPerson.name = "Steven";
firstPerson.age =  30;
```

### 3.2.4 Process Type

In Flow, a process is an independent unit that performs work on zero or more incoming streams of tokens. The process type allows the programmer to define the work done at a node in the Kahn Process network.

A process may act as a sender for zero or more channels, as well as a receiver for zero or more channels. The workflow for deploying a process consists of first defining the process and then binding it with the necessary arguments. In a compiled Flow program, each process runs on a separate thread.

### 3.2.5 Channel Type

Channels are unbounded FIFO structures that connect processes to other processes. At any time, a channel may contain a buffer of zero or more tokens - elements that have been been sent to that channel but not removed from it. The tokens that a channel holds must be of a uniform type that is determined by its declaration.

A channel must be bound to exactly one sending process and one receiving process. Only the bound sending process may send tokens to the channel, and only the bound receiving process may receive tokens from the channel. The receiving process is guaranteed to receive tokens in the order in which they were sent.

Channels may not be queried for size, nor can the next item in a channel be read without removing it from the channel.

# 4 Declarations

In Flow, the act of declaration associates an identifier with a particular semantic meaning. In particular, a declaration can associate an identifier with a primitive type, string, list, function, process, or channel. Function declaration and process declaration will not be covered here, and will instead be covered in Section 7.2 and Section 7.3 respectively.

If an identifier has been declared as a primitive type or non-primitive type, initializing it gives that identifier a concrete initial value. If an identifier has been declared as a function or process, defining the identifier gives it a body.

For channels, a declaration is simultaneously a definition.

## 4.1 Primitive Type Declaration and Initialization

All built-in primitive types have associated with them a keyword, which is used to declare an identifier of that type. The table below enumerates all types and their keywords.

| **Primitive Type** | **Keyword** |
|---|---|
| Integer | `int` |
| Double precision floating point v | `double` |
| Character | `char` |
| Boolean | `bool` |
| Void (only for function return type) | `void` |

Identifiers associated with primitive types may be declared without being initialized. They may also be simultaneously declared and initialized using the assignment operator, '='. This pattern is best summed up as follows:

```
primitive_declaration :=
    primitive_declarator;
  | primitive_declarator = expr;

primitive_declarator :=
    primitive_type IDENTIFIER

primitive_type :=
    int
  | double
  | char
  | bool
  | void
```

In a valid declaration, the type of the expression must match against `primitive_type`. Below are some valid declarations for primitive types.

```
int x;                # uninitialized integer
int y = 1;            # initialized integer
bool b;               # uninitialized boolean
```

```
char c = 'a'          # initialized char
```

## 4.2 Non-Primitive Type Declaration and Initialization

### 4.2.1 String Declaration and Initialization

Strings are created with the `string` keyword.

```
string_declaration :=
    string_declarator;
  | string_declarator = STRING_LITERAL;

string_declarator :=
    string IDENTIFIER
```

`STRING_LITERAL` is a valid ascii string enclosed in double quotes. Uninitialized strings are automatically initialized to the empty string.

```
string foo;         # foo == ""
string bar = "baz"  # initialized string
```

### 4.2.2 List Declaration and Initialization

Lists of any type are declared with the `list` keyword.

```
list_declarator :=
    list_type list IDENTIFIER

list_declaration :=
    list_declarator;
  | list_declarator = list_initializer;

list_type :=
    primitive_type
  | string
  | IDENTIFIER
```

`list_type` can be one of the keywords for a primitive type, the `string` keyword, or an identifier for a struct type (discussed in the following two sections). The `list_initializer`

above allows for lists to have initial values. A comma-separated list of initial values may be assigned to the list inside square brackets.

```
list_initializer :=
    [ expr_list ]
```

Appropriate syntax for list declaration and initialization is:

```
string list foo;             # An empty list of strings
int list bar = [1, 2, 1+2]; # An initialized list of ints
```

### 4.2.3 Structs

Structs describe programmer-defined data types, and are declared with the `struct` keyword:

```
struct_declaration :=
    struct IDENTIFIER { struct_member_list }

struct_member_list :=
    struct_member_declarator
  | struct_member_list, struct_member_declarator
```

The member list is a comma separated list of declarators, which enumerate the members of the struct. `struct_member_declarator` is defined below. Notice that structs can have other structs as members.

```
struct_member_declarator :=
    primitive_declarator
  | string_declarator
  | list_declarator
  | struct_instance_declarator
```

### 4.2.4 Struct Instances

After a struct is declared, a number of struct instances can be declared and initialized.

```
struct_instance_declarator :=
    IDENTIFIER IDENTIFIER
```

The first `IDENTIFIER` is an identifier for a struct that was previously declared, whereas the second `IDENTIFIER` is the name of the new instance of that struct.

```
struct_instance_declaration :=
    struct_instance_declarator;
  | struct_instance_declarator = { dot_initializer_list }
```

A `dot_initializer_list` can be used to give concrete values to the struct instance members.

```
dot_initializer_list :=
    . IDENTIFIER = expr
  | . IDENTIFIER = expr, dot_initializer_list
```

```
struct dog {
    string breed;
    int size;
}

dog yorkie;                 # uninitialized dog
dog mastiff = {             # initialized dog
    .breed = "mastiff",
    .size = 100;
}
```

### 4.2.5 Channel Declaration

Channel declaration uses the `channel` keyword:

```
channel_declaration :=
    channel_type channel IDENTIFIER;

channel_type :=
    primitive_type
  | string
  | IDENTIFIER
```

`channel_type` is the type of token the channels will hold, and it can be a primitive type, a string, or a struct type. An example of a valid declaration is:

```
char channel x;
int channel y;
```

Channels do not have values, and consequently cannot be initialized. When a channel is declared, Flow creates the corresponding FIFO data structure.

# 5 Expressions

An expression can be a combination of literals, primitive types, non-primitive types, operators, and functions that compute and return a value.

```
expr_list :=
    expr
  | expr, expr_list

expr :=
    INT_LITERAL
  | DOUBLE_LITERAL
  | STRING_LITERAL
  | CHAR_LITERAL
  | BOOL_LITERAL
  | IDENTIFIER
  | @IDENTIFIER
  | expr -> IDENTIFIER
  | function_call
  | expr + expr
  | expr - expr
  | expr * expr
  | expr / expr
  | expr % expr
  | expr == expr
  | expr != expr
  | expr < expr
  | expr > expr
  | expr <= expr
  | expr >= expr
  | expr << expr
  | expr >> expr
  | expr ^ expr
  | expr & expr
  | expr | expr
  | expr && expr
  | expr || expr
  | IDENTIFIER = expr
  | IDENTIFIER[INT_LITERAL] = expr
  | IDENTIFIER[INT_LITERAL]
  | (expr)
```

```
  | ~expr
  | !expr
```

### 5.0.1 Function call

Function calls can be made with the the () punctuator and an optional expression list.

```
function_call:=
    IDENTIFIER()
  | IDENTIFIER(expr_list)
```

### 5.0.2 Casting

Integers can be converted into doubles and doubles can be converted into integers using the following built-in functions:

- `to_double(expr)`
    - casts the given expression to a double
    - valid expressions evaluate to integer literals
- `to_int(expr)`
    - casts the given expression to an int
    - valid expressions evaluate to double or character literals
- `to_char(expr)`
    - casts the given expression to an character
    - valid expressions evaluate to int literals

### 5.0.3 Multiplicative Expression

Multiplicative operators include multiplication (*), division (/), and modulo (%).

Both expressions in a multiplicative operation must evaluate to the same type, which is either an integer or double. A multiplicative operator cannot be evaluated between a double and integer. The arithmetic operator is evaluated on the two expressions and the resulting integer or double is returned.  Note that dividing by 0 will return an error.

### 5.0.4 Additive Expression

Additive operators include addition (+) and subtraction (-).

Both expressions in an additive operation must evaluate to the same type, which is either an integer or double. An additive operator cannot be evaluated between a double and integer. The additive operator is evaluated on the two expressions and the resulting integer or double is returned.

### 5.0.5 Shift Expression

Shift operators include shift left (<<) and shift right (>>).

The left expression should be an integer representing the value to be shifted. The right expression is an integer representing the shift width.

### 5.0.6 Relational Expression

Relational operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=).

Both expressions must evaluate to the same type, which is either an integer or double. A relational operator cannot be evaluated between a double and integer. The relational operator is evaluated on the two expressions and the resulting boolean is returned.

### 5.0.7 Equality Expression

Equality operators: equal to (==) and not equal to (!=).

Both expressions must evaluate to the same type. The equality operator is evaluated on the two expressions and the resulting boolean is returned.

### 5.0.8 Logical-Bitwise Expression

Logical bitwise operators include the bit-wise OR (|) and the bitwise AND (&). The operands of logical bitwise operators must be integers.

### 5.0.9 Logical Expression

Logical operators include AND (&&) and OR (||).

The operands to a logical operator must be booleans, and the result of the expression is also a boolean.

# 6 Statements

A statement expresses an action to be carried out by the computer. Statements end with the semicolon punctuator.

```
stmt:=
    expr_stmt
  | compound_stmt
  | selection_stmt
  | iteration_stmt
  | declaration_stmt
  | jump_stmt
```

### 6.0.1 Declaration Statements

```
declaration_stmt:=
    primitive_declaration
  | string_declaration
  | list_declaration
  | struct_declaration
  | struct_instance_declaration
  | channel_declaration
```

### 6.0.2 Expression Statements

An expression can be a combination of literals, primitive types, non-primitive types operators, and functions that compute and returns a value.

```
expr_stmt :=
    expr;
```

### 6.0.3 Compound Statements

Compound statements can be considered as block.

```
compound_stmt :=
    { stmt_list }

stmt_list :=
    stmt
  | stmt stmt_list
```

### 6.0.4 Selection Statements

A selection statement executes a list of statements based off the value of expressions. An expression within an `if` must evaluate to a boolean. An `if` statement does not need to be accompanied by an `else` statement.

```
selection_stmt:=
    if ( expr ) stmt
  | if ( expr ) stmt else stmt
```

### 6.0.5 Iteration Statements

```
expr_opt :=
    ε
  | expr

iteration_stmt:=
    while ( expr ) stmt
  | for ( expr_opt; expr_opt; expr_opt ) stmt
```

#### 6.0.5.1 While Statement

The `while` statement evaluates a boolean expression and executes the list of statements if the expression evaluates to true. With each iteration the expression in the body of the loop updates. If the expression evaluates to false, the while statement terminates and the list of statements is not executed.

#### 6.0.5.2 For Statement

The `for` statement performs iterations of the block of code. The first `expr_opt` is executed prior to entering and executing the statement. The second `expr_opt` is the condition that needs to be met for the statement block to execute. The third and final `expr_opt` is executed at the end of every iteration. All three expressions are optional.

### 6.0.6 Jump Statements

Jump statements shift the control of a program to a different part of the code.

```
jump_stmt :=
    return expr;
  | return;
  | continue;
  | break;
```

#### 6.0.6.1 Return Statements

The keyword `return` can be used in a function to return control of the program to the calling function or process. If the function has a return type, an expression of that type must come after the return keyword.

#### 6.0.6.2 Continue Statements

The keyword `continue` can be added in a `while` or `for` statement to prematurely finish an iteration of the loop so that the loop can start again.

#### 6.0.6.3 Break Statement

The keyword `break` can be added in a `while` or `for` statement to prematurely terminate and exit from the loop.

# 7 Function and Process Declaration and Definition

## 7.1 Declaring Arguments to Channels and Processes

Functions and processes are both entities that can be invoked with arguments. In this section, we will thoroughly define `arg_declaration_list`, which is a comma separated list of argument declarations:

```
arg_declaration_list :=
    arg_declaration
  | arg_declaration_list, arg_declaration


arg_declaration :=
    primitive_declarator
  | string_declarator
  | list_declarator
  | IDENTIFIER IDENTIFIER
  | in channel_type IDENTIFIER
  | out channel_type IDENTIFIER
```

### 7.1.1 Primitive Types, Strings, and Lists as Arguments

A primitive type, string, or list can be declared as an argument simply with a declarator of that type

### 7.1.2 Structs as Arguments

A struct can be declared as an argument with the following pattern:

```
    IDENTIFIER IDENTIFIER
```

The first `IDENTIFIER` is the identifier associated with the struct type, and the second `IDENTIFIER` is the name of the argument.

### 7.1.3 Channels as Arguments

Channels may be passed as arguments to both processes and functions. The syntax for declaring channels as arguments differs from that used to simply declare channels. Each channel argument declaration follows this pattern:

```
    DIRECTION TYPE IDENTIFIER
```

where `DIRECTION` is either `in` or `out`, `TYPE` is the type of token the channel holds, and `IDENTIFIER` is the channel's identifier. If `DIRECTION` is `in`, the channel may only be read by a process. Conversely, if `DIRECTION` is `out`, the channel may only be written to by a process. To reiterate: this syntax is valid only in the argument declaration list for functions and processes.

## 7.2 Function Declaration and Definition

Functions must be declared with a return type and a list of arguments. Functions may only be declared and defined at the top level. The return type of a function must be a primitive type.

```
function_declaration :=
    function_declarator;
  | function_declarator {}
  | function_declarator {stmt_list}

function_declarator :=
    primitive_type IDENTIFIER()
  | primitive_type IDENTIFIER( arg_declaration_list )
```

A function definition can simultaneously act as its declaration; in other words, a function need not be declared before it is defined, but it must be declared before it is used.

```
// Function declaration and definition
int sum(int x, int y) {
     return x + y;
}

int i = sum(1, 2); # i == 3
```

## 7.3 Process Declaration and Definition

Processes must be declared with a list of arguments. Processes don't return anything, so a return type is not allowed. Processes may only be declared and defined at the top level. A process is declared with the `proc` keyword as follows:

```
process_declaration :=
    process_declarator;
  | process_declarator { }
  | process_declarator { proc_body }

process_declarator :=
    proc IDENTIFIER( arg_declaration_list )
```

As with functions, processes must be defined, and a process definition can act as its declaration.

`proc_body` is a list of statements intended to embody the work done by this particular process. The only difference between `proc_body` and a `stmt_list` is that `proc_body` may not include the declaration of channels, whereas `stmt_list` may. This check is made during semantic analysis rather than during parsing.

As an example, here is the definition of a process that interleaves two input streams and produces one output stream. This example uses the `@` and `->` operators, which are discussed in 11 Program Structure.

```
proc interleaver(in int inchan1, in int inchan2, out int ochan){
      int current_token;
      bool i = false;
      for(;;){
            if(i) @inchan1 -> ochan;
            else  @inchan2 -> ochan;
            i = !i;
      }
}
```

The `interleaver` process has three arguments, all of which are channels. The first two, `inchan1` and `inchan2`, are input channels. That means that `interleaver` can only fetch items from these channels, and never send items down them. The last stream, `ochan,` is an output channel, meaning that `interleaver` can only send items to this channel, and never fetch items from it.

# 8 Scope

Scope in Flow follows the same semantics as C. There exists global scope and block scope. Globally scoped variables can be accessed anywhere in a program. Block scoped variables exists

in blocks (compound statements) such that variables declared within a block are accessible within the block and any inner blocks. If a variable from an inner block declares the same name as a variable in an outer block, the visibility of the outer variable within that block ends at the point of declaration of the inner variable.

# 9 Program Structure

At the top level, a Flow program consists of global variable declarations, function declarations, and process declarations.

```
program :=
    decls EOF

decls :=
    decls declaration_stmt
  | decls function_declaration
  | decls process_declaration
```

The entry point into a flow program is the function `main`. The body of this function may call a series of procedures, perform computations, and, most importantly, define channels and binds processes to those channels. Binding processes to channels establishes concrete links between processes, creating the Kahn Process Network.

When a Flow program is run, the `main` routine is called. After the `main` routine terminates, a Kahn Process Network is constructed and set into motion according to the bindings that were made in the program. Control of execution is turned over to this network. A Flow process terminates when all channels are empty.

## 9.1 The @ and -> Operators

The `@` and `->` operators may only be used from within the body of a process. They are used to retrieve and send tokens to a channel, respectively.

The `@` operator is a unary operator on channel identifiers. Specifically, `@` can only operate on channels with direction `in`. The `@` operator returns the next token in the channel, and blocks if the channel is empty.

The `->` operator expects an expression on the left side, and an `out` channel identifier on the right side. The expression on the left side is evaluated, and the result is sent to the channel. An expression formed with the `->` operator evaluates to the result of the the expression on the left side.

Let us revisit the interleaver process, introduced in 7.3 Process Declaration and Definition:

```
proc interleaver(in int inchan1, in int inchan2, out int ochan){
      int current_token;
      bool i = false;
      for(;;){
            if(i) @inchan1 -> ochan;
            else  @inchan2 -> ochan;
            i = !i;
      }
}
```

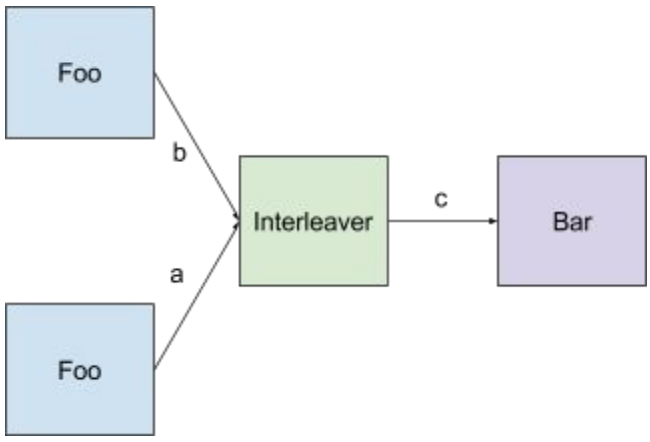
The expression `@inchan1 -> ochan` fetches a token from `inchan1`, and then sends it to `ochan`.

## 9.2 Binding Processes

Binding a process amounts to passing it the appropriate arguments enclosed in `()`. The action of binding a process creates a single node in the resulting Kahn Process Network. A process may be bound an arbitrary amount of times, producing a corresponding number of nodes. Process binding can occur in any block of code. When a bound process finishes and returns, it terminates and will cease to perform work on its channels.

Suppose process `foo` takes a single `out int` channel argument, and process `bar` takes a single `in int` channel argument. Then, these two processes can be bound with `interleaver` to create the KPN pictured on the right:

```
int channel a, b, c;
foo(a);
foo(b);
interleaver(a, b, c);
bar(c);
```



## 9.3 Binding Semantics for Channels and Processes

Since channels must be connected to exactly one receiving process and one sending process, a declared channel **must** have two such processes bound to it before its identifier goes out of scope. If a channel has just one process bound to it, and its identifier goes out of scope, no other portion of the program will be able to bind the second process to the identifier. This notion leads to several programming patterns that must be observed when using the Flow language.

### 9.3.1 Mandatory Channel Binding in Functions

If a function accepts a channel as an argument (see 5.2.1.2 Declaring Channels as Arguments), then it must bind a process to that channel in the body of its routine. The reasoning for this is as follows: suppose a function can accept a channel as an argument, and conditionally bind a process to that channel. Then, a routine that calls this function will not know whether the channel specified as an argument had a process bound to it by the function. The calling routine will consequently not know if it should bind a process to this channel before it goes out of scope. Therefore, it is the responsibility of the function to ensure that it binds a process to every channel that it accepts an argument. Failure to do so is a semantic error and results in undefined behavior.