

IBM Research Report

SPL Stream Processing Language Specification

**Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar, Giuliano
Losa, Mark Mendell, Howard Nasgaard, Robert Soulé, Kun-Lung Wu**

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598



SPL Stream Processing Language Specification

Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Giuliano Losa,
Mark Mendell, Howard Nasgaard, Robert Soulé, and Kun-Lung Wu.

November 4, 2009

Contents

1	Introduction	2
1.1	Language Overview	2
1.2	Summary of Changes from SPADE to SPL	3
1.3	Grammar Notation	3
1.4	Lexical Syntax	4
2	Types	4
2.1	Primitive Types	5
2.2	Composite Types	7
2.3	Value Semantics	9
2.4	Type Conversions	10
3	Expression Language	11
3.1	Expression Operators	11
3.2	Mapped Operators	13
3.3	Statements	14
3.4	Functions	15
4	Streams	18
4.1	Operator Invocation Head	18
4.2	Operator Invocation Body	20
4.3	Stream Logic	22
4.4	Punctuation	23
4.5	Windows	23
4.6	Error Semantics	25
5	Operators	26
5.1	Composite Operators	26
5.2	Stream and Operator Instance Names	30
5.3	Operator Parameters	31
5.4	Domain Toolkits	35
5.5	Shared Variables	36
5.6	Primitive Operators	37
6	Program Structure	40
6.1	Compile-Time Entities	40
6.2	Compilation and Deployment	41
6.3	Dynamic Application Composition	43
6.4	Embedded Documentation	45
A	VWAP Example	46
B	Grammar Overview	47
C	Acknowledgments	50

Abstract

This document is the official specification for SPL. SPL is a programming language for high-performance distributed stream processing systems. An SPL program describes a data flow graph, where the vertices are operator instances and the directed edges are streams. SPL is extensible by allowing users to define their own primitive operators in a native language, and by allowing users to define their own parameterizable composite operators in SPL itself. Furthermore, SPL permits users to dynamically compose independently launched streaming programs. Thanks to its generality, SPL makes it easy to express applications from a large variety of domains, including but not limited to financial trading, transportation monitoring, manufacturing control, security, healthcare, etc.

1 Introduction

This document describes the language design of SPL, the Stream Processing Language. SPL is the programming language for InfoSphere Streams, IBM's high-performance distributed stream processing system [1]. This document only focuses on the syntax and semantics of user-visible features; a description of the implementation design or the inner workings of the compiler is out of scope for this document.

This document is sprinkled with paragraphs containing auxiliary information:

- Practical advice: Best practices and conventions for users.
- Implementation note: Note about how the compiler or runtime implements a feature.
- For SPADE users: Comparisons between SPL and its predecessor language SPADE.
- For language experts: Terminology from the programming language community.
- Language design rationale: Justification for decisions where we had to reconcile conflicting design goals.

In addition, there are numerous code examples. IBM's SPL compiler is continuously tested on these examples. While some examples are semantically incomplete (e.g., using undefined identifiers), all examples are syntactically valid.

1.1 Language Overview

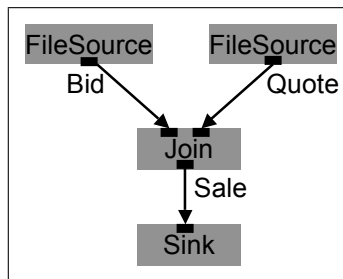


Figure 1: Stream graph for sale-join example.

SPL is a stream programming language. Figure 1 shows an example stream graph. The following code shows how it could be implemented in SPL.

```
composite SaleJoin { // 1
graph // 2
  stream<string8 buyer, string8 item, decimal64 price> Bid = FileSource() { // 3
    param file: "BidSource.dat"; format: csv; // 4
```

```

    } // 5
    stream<string8 seller, string8 item, decimal64 price> Quote = FileSource() { // 6
        param file: "QuoteSource.dat"; format: csv; // 7
    } // 8
    stream<string8 buyer, string8 seller, string8 item> Sale = Join(Bid; Quote) { // 9
        window Bid : sliding, time(30); //10
            Quote : sliding, count(50); //11
        param match : Bid.item == Quote.item && Bid.price >= Quote.price; //12
        output Sale : item = Bid.item; //13
    } //14
    () as Sink = FileSink(Sale) { param file: "Result.dat"; format: csv; } //15
} //16

```

An SPL program consists of one or more composite operators. A composite operator defines a stream graph. The vertices of the stream graph are operator invocations, and the edges are streams. An operator invocation defines output streams by invoking a stream operator on input streams. For example, the operator invocation in Line 9 invokes the `Join` operator on two input streams `Bid` and `Quote` to produce one output stream `Sale`. An operator invocation may also have zero input streams (e.g., `FileSource` in Lines 3 and 6) or zero output streams (e.g., `FileSink` in Line 15). The body of an operator invocation customizes the way the operator works. For example, the `window`, `param`, and `output` clauses in Lines 10-13 customizes how the `Join` operator is invoked.

This language specification is written bottom-up, starting from types such as `string8` or `int32` (Section 2). Values of these types are manipulated by expressions, such as `Bid.item == Quote.item` (Section 3). But the core concept of SPL is the operator invocation, such as `Sale = Join(Bid; Quote)` (Section 4). SPL allows users to define their own primitive or composite operators, such as `SaleJoin` (Section 5). Finally, an SPL program specifies functions and operators in namespaces (Section 6). The appendix contains an extended example (A), a grammar overview (B), acknowledgements (C), and an index.

1.2 Summary of Changes from Spade to SPL

For SPADE users: The main influence for SPL was its predecessor language SPADE. It is described at http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.spade.html. There is also a research paper about SPADE in SIGMOD'08 [4]. The SPL specification is self-contained, so it should be accessible to people unfamiliar with the old SPADE.

SPL adds some features missing from SPADE, removes others, and changes many features. Newly added features include composite operators, shared variables, and a richer data model (`tuple`, `map`, `decimal`, `timestamp`, etc.). Removed features include bundles, bulk functions, and the preprocessor. The most visible change in SPL is the syntax, which has become more readable, especially for programmers familiar with C or Java. For example, in SPADE, the operator invocation for `Sale` looks like this:

```

stream Sale(buyer: String, seller: String, item: String) # 1
:= Join(Bid <time(30)>; Quote <count(50)>) # 2
  [ $1.item = $2.item & $1.price >= $2.price ] # 3
  { item := $1.item } # 4

```

Language design rationale: The design of SPL has the same spirit as SPADE: the primary goal is permitting efficient implementation on distributed hardware. Most changes aim to make the language simpler and more uniform. When given a choice between code readability and writeability, we usually opted for readability.

1.3 Grammar Notation

This language specification adopts a flavor of BNF (Backus Naur Form) to describe the syntax of language features, following the following conventions:

<i>italics</i>	Non-terminal
<i>ALL_CAPS_ITALICS</i>	Token, e.g., <i>ID</i> for identifiers
'fixed-width font'	Verbatim text, quoted to avoid confusion with meta-characters, e.g., '('
(...)	Grouping, to disambiguate meta-syntax precedence
... ...	Alternatives, match either syntax left or right of bar
...?	The preceding syntax is optional
...*	The preceding syntax is repeated zero or more times
...+	The preceding syntax is repeated one or more times
...*,	Comma-separated list of zero or more items
...+,	Comma-separated list of one or more items
...*;	Semicolon-separated list of zero or more items
...+;	Semicolon-separated list of one or more items
<i>nonTerminal ::= ...</i>	Rule definition

1.4 Lexical Syntax

SPL files are written in unicode using UTF-8 encoding. Most syntactic elements, including keywords and identifiers, use the subset of unicode that overlaps with ASCII characters (the subset 32-126 of the Latin-1 alphabet ISO 8859-1). The only two constructs where other unicode characters are valid are unicode string literals and comments. Identifiers start with an ASCII letter or underscore, followed by ASCII letters, digits, or underscores. The syntax for literal values of primitive types (numbers, booleans, strings, etc.) is similar to that of Java or C++, and is described in Section 2.1.

SPL has two forms of comments: single-line comments (from `//` to the end of the line) and delimited comments (between `/*` and `*/`). Delimited comments can span multiple lines, and can be followed by regular code in the same line.

SPL syntax is not sensitive to indentation or line breaks. SPL syntax is case-sensitive, for example, `mud` and `Mud` are different identifiers.

SPL uses lexical scoping for identifiers. In other words, a declaration in an inner scope shadows declarations of the same identifier in statically enclosing scopes. SPL does not permit synonymous entities of different categories in the same scope. For example, it is a compile-time error if a program declares both an operator named `f` and a function named `f` in the same scope. As another example, it is a compile-time error if a program declares both a type named `t` and a variable named `t` in the same scope. In other words, SPL does not segregate scopes by identifier categories, unlike for example Java. That also means that identifiers in inner scopes shadow identifiers in outer scopes even when they are of a different category. For example, a locally declared stream `s` hides any function `s` in an outer scope.

Practical advice: We encourage developers to imitate the indentation and line breaks style demonstrated in the examples in this document. We encourage starting identifiers for operators, aggregations, and user-defined types with an upper-case letter, and starting functions, variables, and built-in types with a lower-case letter. We encourage `CamelCase` instead of `underscore_style` to separate words in identifiers. We encourage starting shared variable identifiers with `s_`.

For SPADE users: In SPADE, single-line comments started with `#` and delimited comments were surrounded by `#*...*#`.

2 Types

SPL has a wide array of primitive types tailored for streaming, and a small set of composite types inspired by scripting languages but statically checked. Figure 2 shows all types arranged in a hierarchy. This section describes the types, how to define them, and how to convert values between different types.

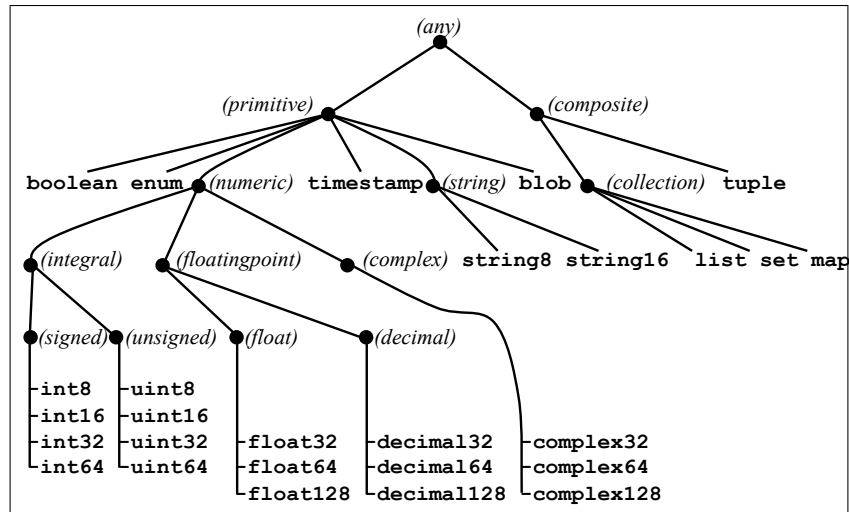


Figure 2: Hierarchy of SPL types.

2.1 Primitive Types

A primitive type, such as `int32` or `string8`, is one that is not composed of other types. SPL supports the following primitive types:

<code>boolean</code>	true or false
<code>enum</code>	user-defined enumeration of identifiers
<code>intb</code>	signed b -bit integer, one of:
<code>int8</code>	(-128 to 127)
<code>int16</code>	(-32,768 to 32,767)
<code>int32</code>	(-2,147,483,648 to 2,147,483,647)
<code>int64</code>	(-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
<code>uintb</code>	unsigned b -bit integer, one of:
<code>uint8</code>	(0 to 255)
<code>uint16</code>	(0 to 65,535)
<code>uint32</code>	(0 to 4,294,967,295)
<code>uint64</code>	(0 to 18,446,744,073,709,551,615)
<code>floatb</code>	IEEE 754 binary b -bit floating point number, one of:
<code>float32</code>	(single-precision, equivalent to <code>float</code> in Java)
<code>float64</code>	(double-precision, equivalent to <code>double</code> in Java)
<code>float128</code>	(same as <code>binary128</code> in the IEEE 754-2008 standard)
<code>decimalb</code>	IEEE 754 decimal b -bit floating point number, one of:
<code>decimal32</code>	(significand 7 decimal digits, exponents 10^{-95} to 10^{94})
<code>decimal64</code>	(significand 16 decimal digits, exponents 10^{-383} to 10^{384})
<code>decimal128</code>	(significand 34 decimal digits, exponents $10^{-6,143}$ to $10^{6,144}$)
<code>complexb</code>	$2b$ -bit complex number, one of:
<code>complex32</code>	(both real and imaginary part are <code>float32</code>)
<code>complex64</code>	(both real and imaginary part are <code>float64</code>)
<code>complex128</code>	(both real and imaginary part are <code>float128</code>)
<code>timestamp</code>	point in time, with nanosecond precision
<code>string8</code>	string of UTF-8 unicode characters, based on ICU library
<code>string16</code>	string of UTF-16 unicode characters, based on ICU library
<code>blob</code>	sequence of raw bytes

`string8[n]` bounded-length string of at most n bytes worth of characters

Besides these primitive types, SPL also defines metatypes for compile-time entities in operator parameters, see Section 5.3.4.

An example for an enumeration is

```
type LogLevel = enum { error, info, debug, trace };
```

Any of the identifiers `error`, ..., `trace` can be used where a value of enumeration `LogLevel` is expected. The scope of the identifiers `error`, ..., `trace` is the same as the scope containing the type definition. Enumerations are ordered (they permit comparison with `<`, `>`, `<=`, and `>=`) but not numeric (they do not permit arithmetic with `+`, `-`, `*`, etc.).

Like in C/Java, literals for `int`, `uint`, `float`, and `decimal` can have optional type suffixes. For example, `123` is signed (`int32`) whereas `123u` is unsigned (`uint32`). One suffix indicates the kind of number:

```
s -> signed integer (default for integer literals)
u -> unsigned integer
f -> binary floating-point (default for floating point literals)
d -> decimal floating-point
```

Another suffix indicates the number of bits:

```
b (byte)            -> 8-bit
h (half-word)      -> 16-bit
w (word)            -> 32-bit (default for integer literals)
l (long)            -> 64-bit (default for floating point literals)
q (quad-word)      -> 128-bit
```

Some more examples for literals with type suffix: `0.0005 (float64)`, `0.5e-3 (float64)`, `3.5d (decimal64)`, `3.5w (float32)`, `123d (decimal64)`, `123dq (decimal128)`,

String literals are written in double-quotes. A type suffix indicates the number of bits: `b` indicates `string8` and `h` indicates `string16`, the default (without suffix) being `string8`. Both `string8` and `string16` can contain any unicode characters, they differ only in their encoding (UTF8 vs. UTF16). String literals can use escape sequences of the form `\uhhhh`, where the four hexadecimal digits `hhhh` specify a character. For example, `"Bu\u011fra"` uses the escape `\u011f` to specify a `ř` with a u-shaped accent on top in a `string8`. Recall from Section 1.4 that SPL files are written in UTF-8, so letters such as `ř` can also appear directly in a string literal, without the escape sequence.

Strings may contain internal null characters, which, unlike in C, are not considered terminating. In other words, the length of a string is independent from whether or not it contains characters whose encoding is zero.

Literals for complex numbers are written by using the type as a function name, as in `complex32(1.0, 2.0)`.

A timestamp stores TAI (International Atomic Time) with nanosecond precision. It uses a 128 bit representation, where 64 bits store the seconds since the epoch as a signed integer, 32 bits store the nanoseconds, and 32 bits store an optional identifier of the machine where the measurement was taken, which can be useful for after-the-fact drift compensation. The SPL library provides various functions for manipulating timestamps:

- Getting TAI on the local machine. Usually, this involves obtaining UTC from the operating system, then adding the correct leap second offset.
- Converting back and forth between a TAI timestamp value and an ISO time string. An example for an ISO time string is `"1960-01-01T23:03:20"`.
- Converting back and forth between a TAI timestamp value and its second, nanosecond, and machine identifier components represented as integers.

- Converting back and forth between TAI, UTC, and UT time values, by consulting the appropriate conversion tables.

Many operators and functions are overloaded to work with different types. For example, the operator `+` can add various types of numbers, but it can also concatenate strings. Likewise, the function `length(x)` is overloaded to accept `x` of type `string8` or `string16`.

SPL offers a bounded-size variant of some types. For example, `string8[5]` can store any strings whose UTF-8 encoding occupies at most 5 bytes. The compiler prohibits implicit conversions from unbounded to bounded types, but the user can override that by explicit casts. Type bounds, whether in variable declarations or in casts, must be compile-time constants. A cast from any string to a bounded string truncates the value if it is too long. SPL limits all strings, bounded or unbounded, UTF-8 or UTF-16, to at most $2^{31}-1$ characters. SPL does not offer bounded UTF-16 strings.

Blobs are sequences of at most $2^{63}-1$ raw bytes.

Implementation note: The language specification purposely does not specify a byte order, because users are oblivious to these details within an SPL application. Compilers are expected to provide flags to choose, for example, network byte order or native byte order. The on-the-wire format for the streaming middleware will be specified in a separate forthcoming document. The internal representation of both bounded and unbounded strings and blobs stores a separate length field. As with all types, the exact layout is implementation dependent and not exposed at the language level. Typically, bounded types would be padded in case the length is lower than the bound, thus allowing subsequent attributes to be stored at a fixed offset in a network packet. The implementation may also reduce the number of bits in the length field according to the bound to save space.

Practical advice: Use the `decimal` floating point types in financial, commercial, and user-centric programs to avoid losing decimal digits to binary rounding. Use TAI instead of UTC or UT wherever possible. For unstructured data of bounded size, use a `list<uint8>[n]` instead of a `blob`, see Section 2.2.

Language design rationale: SPL has an unusually large zoo of primitive types, because in network applications, users need a lot of control over the data representation to achieve performance. Tight representation is important both to keep the data on the wire small, and to reduce serialization and deserialization overheads. The `timestamp` type is designed to allow a high degree of precision as well as avoid overflow conditions (it can represent values ranging over billions of years) by following widely accepted standards. The bounded types permit efficient marshaling and unmarshaling of network packets: if all parts have a fixed size, then parts can be found at a fixed offset without decoding the whole. We considered using slicing instead of casts to convert unbounded to bounded values, but decided against it because unlike type bounds, slicing parameters are not necessarily compile-time constants, and slicing does not alter the type.

For SPADE users: Several primitive types are new in SPL, e.g., `uint`, `string16`, `decimal`, or `timestamp`.

2.2 Composite Types

A composite type is the result of applying a type constructor to one or more type arguments. For example, the composite type `list<int32>` is the result of applying the type constructor `list<T>` to the type argument `T=int32`. For SPADE users: The only composite types in SPADE were `matrix` and `list`; SPL drops `matrix`, but introduces additional composite types `tuple`, `set`, and `map`. Also, in SPADE, a list could only contain primitive values; in SPL, composite types nest arbitrarily.

2.2.1 Collections

SPL has three built-in collections (`list`, `set`, and `map`) with the following type constructors:

<code>list<T></code>	list (random-access zero-indexed sequence)
<code>set<T></code>	set (unordered collection without duplicates)
<code>map<K, V></code>	map (unordered mapping from key type <i>K</i> to value type <i>V</i>)
<code>list<T>[n]</code>	like <code>list<T></code> , but bounded-length
<code>set<T>[n]</code>	like <code>set<T></code> , but bounded-length
<code>map<K, V>[n]</code>	like <code>map<K, V></code> , but bounded-length

Like primitive strings, collection types have bounded-length variants, which follow the same rules. For example, a `list<int32>[2]` can store at most two integers. It is legal to declare bounded-size types with unbounded elements, such as `list<string8>[3]`, though that obviously does not offer the same marshaling optimization opportunities. SPL limits all collections, bounded or unbounded, to at most $2^{31}-1$ elements.

The literal syntax for list, set, and map values is as follows:

<code>listLiteral</code>	<code>::=</code>	<code>[' expr* , ']</code>	<code># e.g. [0, x, x+1, x+2]</code>
<code>setLiteral</code>	<code>::=</code>	<code>{' expr* , '}</code>	<code># e.g. { "IBM", "GOOG" }</code>
<code>mapLiteral</code>	<code>::=</code>	<code>{' (expr ':' expr)* , '}</code>	<code># e.g. { "Mon": -1, "Fri": 1 }</code>

List, set, and map types can be arbitrarily nested. In other words, their elements, and in the case of maps even their keys, can be of any type, including other composite types. Empty list, set, and map literals can only occur where their type is clear from the context: casts, variable initializers, assignments, and typed operator parameters.

As we will see in Section 3, composite types have their own operators. You can subscript a list or map with `l[i]`, check membership in a collection with “`x in s`”, iterate over a collection with “`for(T x in s) ...`”, and so on. There are also various functions to work on collections; for example, `setUnion(set1, set2)` or `setIntersection(set1, set2)`. For lists and maps, the left operand of the `in` operator is the key. In other words, “`0z in m`” checks whether “`0z`” is a key of the map `m`. Value (as opposed to key) membership tests use functions, not the `in` operator.

Implementation note: Lists are implemented via arrays, but unlike C, the language uses static or dynamic checks to protect against out-of-bounds accesses. Sets and maps are implemented via hash tables. They are unordered and support constant-time lookup.

2.2.2 Tuples

A tuple is a set of attributes, and an attribute is a named value. For example, the tuple `{sym="Fe", no=26}` consists of two attributes `sym="Fe"` and `no=26`. The type for this tuple is `tuple<string8 sym, int32 no>`. Tuples are similar to database rows in that they have an unordered set of named and typed attributes. Tuples differ from objects in Java or C++ in that they do not have methods.

Tuple types can extend other tuple types, for example:

```
type Loc2d = tuple<int32 x, int32 y>; // 1
Loc3d = Loc2d, tuple<int32 z>; // 2
```

The resulting type `Loc3d` is equivalent to `tuple<int32 x, int32 y, int32 z>`. In tuple extension, the identifier may also be the name of a stream as a shorthand for the type of the tuples in that stream. For example:

```
stream<int32 x, int32 y> LocStream = FileSource() {/*...*/} // 1
type LocWithId = LocStream, tuple<string8 id>; // 2
```

The resulting type `Loc3d` is equivalent to `tuple<int32 x, int32 y, string8 id>`. The syntax for tuple types is:

```

tupleType ::= 'tuple' '<' tupleBody '>'
tupleBody ::= attributeDecl+, # attributes
              | ( ID | tupleType )+, # tuple type extension
attributeDecl ::= type ID

```

In other words, the body of the tuple type either consists of a comma-separated list of attribute declarations, or a comma-separated list of tuple types, either by name or written in-place. Tuple types can be arbitrarily nested. In other words, their attributes can be of any type, including other composite types, even other tuple types. Here is an example:

```

type Loc2d = tuple<int32 x, int32 y>; // 1
  Sensor = tuple<Loc2d loc, string8 color>; // 2

```

The resulting type `Sensor` is equivalent to `tuple<tuple<int32 x, int32 y> loc, string8 color>`. For example, given a variable `Sensor s`, the expression `s.loc.x` accesses the x-coordinate.

As we will see in Section 5.1.2, the explicit `tuple<...>` type constructor can be omitted at the top level of a type definition. That means that the previous example can also be written like this:

```

type Loc2d = int32 x, int32 y; // 1
  Sensor = Loc2d loc, string8 color; // 2

```

The literal syntax for tuple values is as follows:

```

tupleLiteral ::= '{' ( ID '=' expr )*, '}' # e.g. { x=1, y=2 }

```

You can access an attribute of a tuple with dot notation, e.g., `myLocation.x`.

Language design rationale: Attributes within a tuple are unordered for semantic consistency with relational databases. Users should treat tuples as a high-level concept, and not order fields in a misguided effort to optimize the representation. Such an effort would be futile, because the runtime implementation uses a canonical ordering of attributes anyway, and the representation is invisible to the user.

2.3 Value Semantics

All primitive and composite SPL types have value semantics, not reference semantics. That means that:

- An assignment (“=”) copies the contents, instead of aliasing the location.
- An equality comparison (“==”) compares the contents, instead of comparing the locations.

One consequence of value semantics is that modifying the original after an assignment does not change any copies. Consider this example:

```

mutable map<string8, tuple<int32 x, int32 y>> places = { }; // 1
mutable tuple<int32 x, int32 y> here = { x=1, y=2 }; // 2
places["Hawthorne"] = here; // 3
here.y = 3; // 4

```

Line 3 initializes variable `here` with the value `{x=1,y=2}`. Line 4 assigns a copy of that value into the map at key `"Hawthorne"`. Line 5 modifies the version of the value in variable `here`, so variable `here` now contains `{x=1,y=3}`. However, this does not affect the copy at `places["Hawthorne"]`, which is still `{x=1,y=2}`. Another consequence of the value semantics is that since there is no notion of a “reference”, there is no notion of “null”, nor can there be any cyclic data structures in SPL.

Language design rationale: We picked value semantics for SPL because it is more natural, and more efficient, to treat streaming data as pure copies rather than having an identity in an address space. Disallowing references also prevents null pointer errors, simplifies memory management, and prevents unintended side effects of mutating a value stored in a collection or used as a map key.

Practical advice: If you need a more powerful type system than that provided by SPL, use SPL's extension mechanisms and implement your functionality in a low-level language such as C++ or Java.

2.4 Type Conversions

SPL is statically typed and strongly typed. The static typing is enforced by explicit type declarations and compile-time type checking. The strong typing is enforced by providing almost no implicit conversions between types. Instead of implicit conversions, SPL offers explicit conversions (casts) where it makes sense. Explicit conversions look like function calls, e.g., `int32(2.5)` returns `2`.

SPL allows the following explicit type conversions:

- From any type to the same type (identity conversion).
- From any type to any string type and back (string conversion).
- From any enumeration type to any integral type and back. In conversions, enumerations are numbered from zero. For example, given `type t = enum{a,b,c};`, then `int32(a) == 0`, and `t(1) == b`.
- From any non-complex number type to any other non-complex number type.
- From `timestamp` to `float64` and back, where the `float64` represents seconds.
- From any complex number type to any other complex number type.
- From any list type to any other list type, bounded or unbounded, if the element types are either identical, or the element types are convertible and primitive.
- From any set type to any other set type, bounded or unbounded, if the element types are either identical or convertible and primitive.
- From any map type to any other map type, bounded or unbounded, if the key and value types are either identical or convertible and primitive.
- From a tuple type `WideT` to a tuple type `NarrowT` if `WideT` has a superset of the attributes of `NarrowT`. The conversion discards excess attributes and cannot be reverted.

Note that SPL does not permit conversions from and to boolean. For example, `boolean(myInt)` is illegal, because it is shorter and less ambiguous to use integer comparison such as `myInt!=0`.

Types in SPL are equivalent if they are the same primitive types, or if they are composed from equivalent types using the same type constructor. For example, after the following:

```
type LocT1 = int32 x, int32 y; // 1
      LocT2 = int32 y, int32 x; // different order, but same attributes // 2
```

variables of these two types can be assigned to each other:

```
LocT1 loc1 = { x=1, y=2 }; // 1
LocT2 loc2 = loc1; // this is legal // 2
```

As far as SPL is concerned, `LocT1` and `LocT2` are the same type; the fact that they have different names and attribute orders is irrelevant, because a tuple is an unordered set of attributes. Therefore, the assignment between variables `loc1` and `loc2` is legal, and does not constitute a type conversion. (For language experts: SPL uses structural equivalence.)

SPL permits implicit conversions in two places: variable initializers and subscripts. For example, the variable initializer `int8 x = 3;` implicitly converts from `int32` to `int8`, as if it had been written `int8 x = int8(3);`. (For language experts: Implicit conversion in initializers prevents what is known as “type stuttering”, the unnecessary repetition of the type type in the initializer.)

The other place where SPL permits implicit conversions is in subscripts. For example, let `v` be a list. Lists indices are always `uint32`, so the subscript `v[9]` implicitly converts from `int32` to `uint32`, as if it had been written `v[uint32(9)]`. An out-of-bounds subscript causes a runtime error, independently of whether or not there was an implicit conversion involved. Not all subscripts are convertible. For example, subscripting a list `v` with a string `v["nine"]` is a compile-time error.

Unlike some other languages, SPL has no implicit conversion from `int` to `string` (no `"num" + 1`), from `int` to `float` (no `1+2.0`), or from `int` to `boolean` (no `while(1)...`). To perform these conversions, you must make them explicit: `string8(1)` or `float64(1)`.

Language design rationale: SPL adopts a strong static type system, because it saves time for runtime checks, and errors that are not prevented statically are difficult to track down in a distributed system. SPL uses structural equivalence, because that facilitates converting data to and from external applications, files, or databases that do not share the same type system. We acknowledge that structural equivalence can lead to types being considered equivalent even when they were intended to differ, but we decided for structural equivalence anyway to avoid adapter bloat, which can slow down programs and clutter up code.

For SPADE users: Both SPADE and SPL are statically typed and strongly typed, with few implicit conversions. The explicit conversion syntax of SPADE is more verbose than that of SPL.

3 Expression Language

Though SPL is a streaming language, there are many places where it uses expressions found in traditional imperative or functional languages. Some expressions are evaluated at runtime: in operator output assignments or operator parameters. Some expressions are evaluated purely at compile time, during code generation: these include operator configurations and window clauses. And finally, there are places where SPL accepts not only expressions, but even statements like those in traditional languages: this is the case in operator logic, and in function definitions.

3.1 Expression Operators

Expression operators are used to compute values, e.g., with `+` or `-`. They are not to be confused with stream operators, used to connect streams into a data flow graph. The SPL operator table looks more or less like that of C or Java:

<code>e(...)</code>	N	Function call or type cast
<code>e[...]</code>	N	Map lookup, or subscript or slice into string / blob / list
<code>e.id</code>	1	Tuple attribute access
<code>++ -- ! - ~</code>	1	Increment, decrement, prefix logical / arithmetic / bitwise negation
<code>* / %</code>	2	Multiplication, division, remainder
<code>+ -</code>	2	Addition, subtraction
<code><< >></code>	2	Bitwise shift left, right
<code>< <= > >= != ==</code>	2	Comparison (value semantics)
<code>&</code>	2	Bitwise and
<code>^</code>	2	Bitwise xor
<code> </code>	2	Bitwise or
<code>&&</code>	2	Logical and (short-circuit)
<code> </code>	2	Logical or (short-circuit)
<code>?:</code>	3	Ternary conditional
<code>in</code>	2	Membership in list / map / set
<code>= += -= *= ...</code>	2	Assignment (value semantics)

In this table, precedence goes from high (at the top) to low (at the bottom). Literals, such as strings, maps, tuples, lists, etc., have higher precedence than the highest-precedence operator. The middle column of the table indicates arity (the number of operands). All the binary operators (arity 2) are left-associative.

Besides the simple assignment operator (`=`), the following operators first perform a regular binary operation, and then an assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, and `|=`.

As mentioned in Section 2.3, SPL uniformly uses value semantics. Hence, even in the case of composite values, assignments always copy the contents, and comparisons always compare the contents, never the location, of a value.

Side-effecting expressions are only allowed as top-level statements, not nested in other expressions, e.g.:

```

y = x++      + x++      ; //illegal, since "++" has side effect           // 1
x++;
//legal as top-level statement                                           // 2
y = (x=x+1) + (x=x+1); //illegal, since "=" has side effect             // 3
x = x + 1;
//legal as top-level statement                                           // 4
y = inc(x)   + inc(x) ; //may be legal, if "inc" does not have side effects // 5
inc(x);
//legal as top-level statement                                           // 6

```

Restricting side-effects to top-level statements makes code more readable, prevents common programming mistakes, and may lead to more optimization opportunities. The compiler checks whether a function such as `inc` is side-effect free by checking that it is not stateful and none of its parameters are mutable (see Section 3.4).

The right-shift operator (`>>`) behaves differently for signed and unsigned integers. For signed integers, it fills in with the sign bit, whereas for unsigned integers, it fills in with zero. Use casts if you want to override this behavior.

The ordered comparison operators `<`, `>`, `<=`, and `>=` work on all ordered types. That includes strings, timestamps, enumerations, blobs, and all numeric types except complex numbers. Using `<`, `>`, `<=`, or `>=` on complex numbers, containers, tuples, or booleans is a compile-time error.

String, blob, and list subscripts can either retrieve a single element, or can perform slicing. Map subscripts can only refer to one element, not a slice, since maps are unordered. All string, blob, and list indexing is zero-based, and slices include their lower bound but exclude their upper bound. For example, if `a` is a list, then `a[2:5]` is the same as the list `[a[2], a[3], a[4]]`. If the lower bound is omitted, the slice starts at element zero; if the upper bound is omitted, the slice continues until the last element. For example, if the last element of `a` has index 9, then `a[7:]` is the same as the list `[a[7], a[8], a[9]]`. Here is the syntax:

```

subscriptExpr ::= expr '[' subscript ']'
subscript     ::= expr | ( expr? ':' expr? )

```

Invalid subscripts cause runtime errors. Section 4.6 explains what happens upon errors. A subscript is invalid if it is out-of-bounds for its collection. This is true both in expressions and on the left-hand-side of assignments; use builtin functions like `insert` or `insertM` to create new elements in a collection. For example, if list `v` has 3 elements, then only indices $0 \leq i \leq 2$ are valid, and `v[i]` is a runtime error for all $i \geq 3$. On the other hand, a slice `x[lower:upper]` is valid even if `lower` or `upper` is out of bounds. Here are some examples:

```

list<string8> x = ["a", "b", "c"];           // 1
string8 s = x[4]; //runtime error: index out of bounds // 2
mutable list<string8> y;                   // 3
y = x[1 : 5]; // ["b", "c"]                // 4
y = x[5 : 5]; // [ ]                       // 5
y = x[5 : 1]; // [ ]                       // 6
y = x[5 : ]; // [ ]                       // 7
y = x[0 : 2]; // ["a", "b"]               // 8
y = x[ : 2]; // ["a", "b"]               // 9
y = x[2 : 0]; // [ ]                     //10

```

Practical advice: If you are not certain whether a subscript is valid, guard it with a defensive membership test, for example:

```

if ("0z" in places)           // 1
    munchkinland = places["0z"]; // 2

```

Language design rationale: The rules for subscripting with index or slice match the rules in the Python language. They also match the rules in the old SPADE language, except that the upper bound of a slice was inclusive in SPADE but is exclusive in SPL.

3.2 Mapped Operators

SPL takes inspiration from Matlab and supports auto-vectorization (“mapping”) of expression operators to work over lists and maps. There are two kinds of mapped operators: non-dotted and dotted. Non-dotted binary operators such as `*`, `+=`, or `-` are mapped if one operand is a scalar and the other a list or map. Here are some examples:

```

mutable list<int32> ls = [1, 2, 3];           // 1
ls = 2 * ls; // 2 * [1, 2, 3] == [2, 4, 6] // 2
ls += 2; // [2, 4, 6] + 2 == [4, 6, 8] // 3
ls = ls - 1; // [4, 6, 8] - 1 == [3, 5, 7] // 4
mutable map<string8, int32> mp = {"x":1, "y":2}; // 5
mp = 2 * mp; // 2 * {"x":1, "y":2} == {"x":2, "y":4} // 6
mp += 2; // {"x":2, "y":4} + 2 == {"x":4, "y":6} // 7
mp = mp - 1; // {"x":4, "y":6} - 1 == {"x":3, "y":5} // 8

```

SPL also has dotted mapped operators such as `.+` or `.*`. If both operands are equal-length lists or maps with the same key set, the dotted operators work on corresponding pairs of elements at a time. For example:

```

[3,2] .* [5,4] == [3*5, 2*4] == [15,8] // multiply two lists
{"x":4, "y":6} .- {"x":3, "y":1} == {"x":1, "y":5} // subtract two maps

```

Dotted operators have the same precedence as their non-dotted counterparts:

<code>.*</code>	<code>./</code>	<code>.%</code>	Mapped multiplication, division, remainder			
<code>.+</code>	<code>.-</code>		Mapped addition, subtraction			
<code>.<<</code>	<code>.>></code>		Mapped bitwise shift left, right			
<code>.<</code>	<code>.<=</code>	<code>.></code>	<code>.>=</code>	<code>!=</code>	<code>==</code>	Mapped comparison
<code>.&</code>						Mapped and
<code>.^</code>						Mapped xor
<code>. </code>						Mapped or

For SPADE users: SPADE implicitly mapped both expression operators and functions over lists. SPL does not provide implicitly mapped functions, because the user can achieve the same effect with an explicit function and loops, and because doing it implicitly would cause confusion with side effects and with SPL's richer type system.

3.3 Statements

Statements are the traditional building blocks of imperative languages. As a streaming language, SPL does not need statements most of the time, but they are permitted inside operator logic and function definitions. SPL's assortment of statements is deliberately simple; if you want to write sophisticated imperative code, you should use SPL's extension features and put it in a low-level language such as C++ or Java.

A local variable declaration consists of an optional `mutable` modifier and a type, followed by a comma-separated list of declared identifiers with optional initializer expressions, followed by a semicolon:

$$localDecl ::= \text{'mutable'}? \text{ type } (\text{ ID } (\text{'=' } \text{ expr })?)^+, \text{'};'$$

An example local variable declaration is `mutable int32 i=2, j=3;`. The syntax is similar to C or Java, except that the type does not get tangled up with the variable. For example, SPL does not have declarations like `int32 x,y[],z;`. Immutable local variables (without `mutable` modifier) must be initialized.

A block consists of zero or more statements or variable declarations, surrounded by curly braces:

$$blockStmt ::= \text{'\{'} \text{ stmt}^* \text{'\}'}$$

An example block is `{ int32 i=0; foo(i, 2); }`. Local variables declared in a block remain in scope until the end of the block. Blocks are often used as bodies for control statements like `if/while/for`, or as function bodies.

An expression statement consists of an expression followed by a semicolon:

$$exprStmt ::= \text{ expr } \text{'};'$$

An example expression statement is `foo(i, 2);`. Obviously, the purpose of an expression statement is its side-effect. The only operator that have non-error side-effects are assignments, certain function calls, and increment/decrement (`++/--`) operators.

An `if` statement can have an optional `else` clause:

$$ifStmt ::= \text{'if'} (\text{'(' } \text{ expr } \text{'\}') } \text{ stmt } (\text{'else'} \text{ stmt })?$$

Dangling `else` is resolved to the innermost `if`; you can override this with blocks. SPL does not have a C-style `switch` statement.

A `for` statement loops over the elements of a list, set, or map:

$$\text{forStmt} ::= \text{'for' ' (' type ID 'in' expr ')'} \text{ stmt}$$

SPL's `for` loops are similar to “`for (type ID : expr)`” loops in Java 5, but use the Python-style `in` instead of the colon (`:`). SPL does not have a C-style 3-part `for` loop. If the expression is a list or set, the loop variable `ID` iterates over the elements; if the expression is a map, the loop variable `ID` iterates over the keys.

A `while` statement looks just like in C or Java:

$$\text{whileStmt} ::= \text{'while' ' (' expr ')'} \text{ stmt}$$

A `break` statement abruptly exits a `while` or `for` loop:

$$\text{breakStmt} ::= \text{'break' ';'}$$

A `continue` statement abruptly jumps to the next iteration of a `while` or `for` loop:

$$\text{continueStmt} ::= \text{'continue' ';'}$$

A `return` statement abruptly exits a function, optionally returning a value:

$$\text{returnStmt} ::= \text{'return' expr? ';'}$$

To summarize, SPL supports the following assortment of statements:

$$\begin{aligned} \text{stmt} ::= & \text{localDecl} \mid \text{blockStmt} \mid \text{exprStmt} \\ & \mid \text{ifStmt} \mid \text{forStmt} \mid \text{whileStmt} \\ & \mid \text{breakStmt} \mid \text{continueStmt} \mid \text{returnStmt} \end{aligned}$$

For SPADE users: SPADE supported a similar assortment of statements in custom logic in `Functor` operators.

Language design rationale: Strictly speaking, SPL does not really need statements, because the user can put such logic in native (C++ or Java) code. We support statements anyway, because it is often easier to write SPL statements than to go to a native language. Also, code written in pure SPL is more portable across back-end languages, and offers more opportunities for front-end optimizations.

3.4 Functions

SPL functions are similar to C functions: they can take parameters, return a value or return `void`, and are defined at the top level, nested in a namespace and in nothing else. Functions are called from expressions, which can occur in many places in an SPL program. There are two kinds of functions: non-native functions are written in SPL, whereas native functions are written in C/C++ or Java. Both kinds of functions can be invoked from SPL with the same syntax, the caller is oblivious to the implementation language.

3.4.1 Non-Native Functions

Non-native functions are written directly in SPL, not in a native language like C/C++ or Java. Here is an example non-native function definition:

```
int32 twice(int32 x) { return x + x; }
```

An SPL function definition consists of a head and a block:


```

functionDef      ::= functionHead blockStmt
functionHead    ::= functionModifier* type ID '(' functionFormal* ',' ')'
functionModifier ::= 'public' | 'stateful'
functionFormal  ::= 'mutable'? type ID

```

The function head contains optional modifiers, the return type, identifier, and list of parameter definitions. Functions can be overloaded. In other words, there can be multiple functions with the same name in the same scope, as long as they have different parameter signatures. For example, there could be two function definitions for `print(int32)` and `print(string8)`, such that `print(42)` would call the former and `print("the answer")` would call the latter. While SPL permits overloading on parameter types, it forbids overloading on return value types. For example, declaring both `int32 f()` and `string8 f()` in the same scope is a compile-time error.

3.4.2 Native Functions

Native function prototypes are declared with SPL syntax in a special comment or an XML model file, but native function implementations are defined in a native (C/C++ or Java) file. Here is an example native function prototype:

```
<ordered T> T max (list<T>)
```

This declares a generic `max` function that works on lists of any ordered type `T`. An ordered type is a type for which the ordered comparison operators (`<`, `>`, `<=`, and `>=`) are defined, including strings, timestamps, enumerations, blobs, and all numeric types except complex numbers. Here is the syntax:

```

functionPrototype ::= genericFormals functionModifier* type ID '(' protoFormal* ',' ')'
genericFormals    ::= ( '<' typeFormal+ '>' )? ( '[' boundsFormal+ ']' )?
typeFormal        ::= typeFormalConstraint ID
typeFormalConstraint ::= 'any' | 'collection' | 'complex' | 'composite' | 'decimal'
                    | 'enum' | 'float' | 'floatingpoint' | 'integral' | 'list' | 'map'
                    | 'numeric' | 'ordered' | 'primitive' | 'set' | 'string' | 'tuple'
boundsFormal      ::= ID
protoFormal       ::= formalModifier* type ID?

```

A generic type formal such as `<ordered T>` in the function prototype can match any actual parameter type subject to two restrictions: first, the actual parameter type must obey the `typeFormalConstraint`; and second, the match must be the same even if it occurs in multiple parameters. Consider the following native function prototype:

```
<list T> T concat (T vals1, T vals2)
```

In this case, the first restriction requires that the parameters must be of some list types, and the second restriction requires that both parameters are of the same list type. For instance, the call `concat([1,2], [3,4,5])` is correct, because both parameters are of type `T==list<int32>`.

Besides generic type formals, a native function prototype also has optional generic bounds formals. For example, here is a prototype that overloads the `max` function for bounded lists:

```
<ordered T>[N] T max (list<T>[N])
```

Again, if the bounds-formal appears multiple times in the prototype, then all the matching bounds from actual parameter types must be identical. If there are ambiguous overloaded versions of the same generic native function in the same scope, the compiler flags an error.

For language experts: Native function call resolution, type checking, and overload conflict detection uses a standard unification algorithm, augmented with checks for the constraints of type formals.

Practical advice: Use the following guidelines to decide whether to write your functions in SPL or as native (C/C++ or Java) code. Write an SPL function if you want to avoid the burden of going to a different file or language, or if you want the portability of code that can be generated in either C/C++ or Java, or you want the compiler to check statelessness for you. Write a native function if you want to reuse existing native code, or if you need generics, or if the native language permits a more natural implementation, or if the native function is significantly faster than one written in SPL.

For SPADE users: All functions in SPADE were native.

3.4.3 Function Parameters

By default, all parameters are deeply immutable. In other words, the function body can not modify the parameter itself or any part of it. For example:

```
void f(list<string8> s) { // 1
    s = ["hi", "there"]; //error: parameter s is not mutable // 2
    s[0] = "hello"; //error: parameter s is not mutable // 3
} // 4
```

The previous example would cause compile-time errors. Parameters can only be modified if they are declared with the `mutable` modifier. The following example is legal:

```
void f(mutable list<string8> s) { // 1
    s = ["hi", "there"]; //legal: s is mutable // 2
    s[0] = "hello"; //legal: s is mutable // 3
} // 4
```

A stateful function is a function that is non-deterministic or has side-effects. A function is non-deterministic if it does not consistently yield the same result each time it is called with the same inputs. A function has side-effects if it modifies state observable outside the function. For the purposes of this definition, “state observable outside the function” includes global variables in native code, files, the network, etc., but excludes mutable parameters. By default, functions are stateless unless annotated with the `stateful` modifier. Native functions are also stateless by default, and the SPL compiler is aware of which native functions are explicitly marked as stateful. (For language experts: functions that are stateless and have no mutable parameters are pure.)

All parameters (mutable or immutable, primitive or composite) are passed by-reference. In other words, the formal parameter in the callee is an alias of the actual parameter in the caller. Note that this is a marked difference from assignments, which have deep-copy semantics (Section 2.3). Function parameters have by-reference semantics (like “`T &v`” in C++), because copying large data into a function would be inefficient when operating only on a small part of that data. The following example illustrates SPL’s parameter passing semantics:

```
void callee(mutable list<int32> x, mutable list<int32> y) { // 1
    x[0] = 1; // 2
    y = [3,4]; // 3
} // 4
void caller() { // 5
    mutable list<int32> a = [0,0], b = [0,0,0]; // 6
    callee(a, b); // 7
} // 8
```

The assignment `x[0]=1` in the callee yields `a[0]==1` in the caller. And the assignment `y=[3,4]` in the callee yields `b==[3,4]` in the caller. If the caller passes a computed value that does not have a storage location, then the callee stores it in a temporary, but modifications have no side effect on the caller. If you prefer by-value semantics for parameters, you can easily emulate them by copying the by-reference parameters into local variables.

Functions themselves can be passed as parameters to operators, because an operator invocation is fully resolved at compile time. But SPL does not permit passing functions to other functions, or storing them in variables, because it makes code harder to understand for the user, and it is harder to optimize indirect invocations in a static compiler. (For language experts: SPL provides neither first-class functions nor higher-order functions.)

Practical advice: Try to write and call stateless functions whenever possible. If you use mostly stateless functions, you are less likely to write buggy code, and the compiler is more likely to be able to optimize your code, for example, by parallelizing it, or by partially evaluating it at compile time.

Language design rationale: When designing SPL, we considered to categorically outlaw stateful functions. However, we found that this was hard to enforce for native functions, and that some stateful functions are useful, for example, functions that interact with external resources such as databases. Furthermore, stateful functions can yield better performance through memoization or when they make a small modification to a large data structure. Therefore, we decided to permit them in SPL, but the language design encourages mostly writing stateless functions. Unfortunately, it is impractical to statically check statelessness for native functions, so library vendors must be careful to declare native functions stateful when they read or write outside state.

4 Streams

The purpose of SPL is to allow users to create streams of data, which are connected and manipulated by operator invocations. SPL programs are designed to be deployed on distributed hardware for scaling [1]. The main goals of SPL are scalability (exploiting distributed hardware), extensibility (encapsulating low-level code in high-level operators), and usability (easing the writing of scalable and extensible code).

A stream is an infinite sequence of tuples. As we saw in Section 2, a tuple is simply an instance of a tuple type, consisting of named attributes. Each stream is the result of an operator invocation. An operator invocation consumes zero or more input streams and defines zero or more output streams. Each time a tuple arrives on one of the input streams, the operator fires, and can produce tuples on output streams. An operator invocation in SPL returns streams, analogously to how a function invocation in a traditional language returns values. However, given that a stream is an infinite sequence of tuples, each operator invocation is active for the duration of the program execution. This section shows how to define streams of tuples by invoking operators.

4.1 Operator Invocation Head

An operator invocation invokes an operator on input streams and defines output streams. Each operator invocation has a head and a body. The head lists the connected streams and the operator used to process these streams. The body elaborates on how the operator is to be invoked. The BNF syntax is:

$$opInvoke ::= opInvokeHead\ opInvokeBody$$

The head of an operator invocation lists the output and input streams and the name of the operator that processes the data. Figure 3 shows a contrived operator invocation that illustrates multiple output and input streams as well as aliases. The SPL code for this example is as follows:

```
( stream<int32 a1, int32 a2> SomeLongNameForAnOutputStream as A;           // 1
  stream<float64 b> AnotherLongNameForAnOutputStream as B                 // 2
```

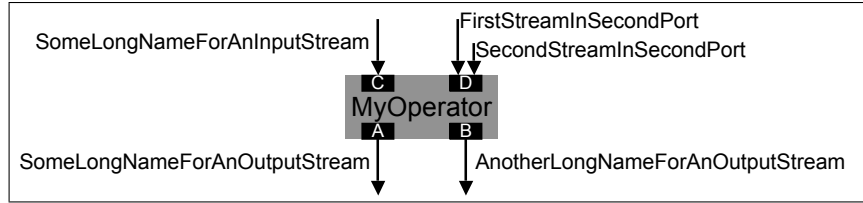


Figure 3: Graphical representation of the information in an operator invocation head.

```

) = MyOperator(                                     // 3
  SomeLongNameForAnInputStream as C;                // 4
  stream<float64 i> FirstStreamInSecondPort, SecondStreamInSecondPort as D // 5
) {                                                 // 6
  output A: a1=1, a2=max(n);                        // 7
  B: b = C.i + D.i;                                // 8
}                                                  // 9

```

Lines 1 and 2 show the two output streams defined by this operator invocation, separated by a semicolon. Each has a type, a name, and an optional alias (`as A` and `as B`). The type lists the attributes of tuples on the stream; the name can be used as input to other operator invocations; and the alias can be used internally in this stream's body. Line 3 shows the name of the operator to invoke, in this case, `MyOperator`. Finally, Lines 4 and 5 show two input ports, separated by a semicolon. Each has an optional type, one or more stream names separated by commas, and an optional alias (`as C` and `as D`). Port D combines two input streams, `FirstStreamInSecondPort` and `SecondStreamInSecondPort`, separated by commas. The general syntax is:

$$\begin{aligned}
 opInvokeHead & ::= opOutputs ('as' ID)^? '=' ID opInputs \\
 opOutputs & ::= opOutput \mid '(opOutput^*;)' \\
 opOutput & ::= streamType ID ('as' ID)^? \\
 opInputs & ::= '(portInputs^*;)' \\
 portInputs & ::= streamType^? ID^+, ('as' ID)^?
 \end{aligned}$$

A port is a point at which streams connect to an operator invocation. Figure 3 shows ports as little black rectangles. Each output port produces exactly one output stream, but an input port can combine more than one input stream. When there is exactly one output port, the parentheses around the outputs can be omitted. When there are zero output ports, the empty parentheses and the operator instance name are mandatory, e.g., `() as A = B() {}`. When there are multiple streams on the same input port, tuples coming from different streams are interleaved. The order of the interleaving is non-deterministic in the sense that no particular order is imposed by the runtime. The `as`-alias notation assigns short names to ports; it is optional, and is useful to keep code readable in the presence of long stream names.

Output ports carry a mandatory stream type, and input ports carry an optional stream type. A stream type just consists of a tuple body:

$$streamType ::= 'stream' '<' tupleBody '>'$$

If there are multiple input streams on the same port, they must all have the same type. If the type of an input port is explicitly specified, it must be equivalent to the type of the stream(s) connecting to it. The `tupleBody` component of the stream type may be either a list of attributes, or a list of tuple types (extension), where identifiers can refer to types of other streams by their stream names. For details, see the description of tuple types in Section 2.2.2.

Practical advice: Instead of using the `as` alias notation, we encourage using short stream names in the

first place. Stream names that are too long indicate that it might be time to decompose the program into composite operators, see Section 5.1.

Language design rationale: By putting just the connected streams and the operator name in the head, we make it possible for users to quickly scan an program's topology. A folding editor can even hide the stream bodies while the user looks at the stream heads. Types on input streams are redundant, because they also occur when those streams are defined, as outputs of other operator invocations. Nevertheless, we allow users to optionally specify these types when they feel that they make code more readable. This can also lead to more understandable compiler error messages when there is a type mismatch.

For SPADE users: SPADE solved the problem of long stream names by referring to streams positionally (`$1`, `$2`, etc.); SPL uses aliases instead. SPADE also allowed additional information in stream heads, such as `import/export` modifiers; SPL replaces them by `Import/Export` operators (see Section 6.3).

4.2 Operator Invocation Body

The body of an operator invocation specifies how the operator is to be customized. SPL supports a wide variety of operators with the default toolkits shipped with the compiler, and furthermore, developers can extend SPL with new operators. Each operator can be customized and reused in different places in a data flow graph. To support this customization, SPL supports a versatile customization syntax. All the configuration happens in the operator invocation body, to avoid tangling it with the data flow specification in the operator invocation head. The following example illustrates all possible operator invocation body clauses:

```
stream<string8 buyer, string8 seller, string8 item, uint64 id> Sale      // 1
= Join(Bid; Quote)                                                    // 2
{                                                                       // 3
  logic state                  : mutable uint64 n = 0;                 // 4
    onTuple Bid                : n++;                                  // 5
  window Bid                   : sliding, time(30);                   // 6
    Quote                       : sliding, count(50);                 // 7
  param match                   : Bid.item == Quote.item && Bid.price >= Quote.price; // 8
  output Sale                   : item = Bid.item, id = n;            // 9
  config wrapper                : gdb;                                 //10
}                                                                        //11
```

An operator invocation body can have five clauses:

- The `logic` clause consists of local state that persists over the whole program execution, and statements that execute when a tuple arrives on an input port (see Section 4.3).
- The `window` clause specifies how many previously received tuples of each port to remember for later processing by stateful operators such as `Join`, `Sort`, or `Aggregate` (see Section 4.5).
- The `param` clause contains code snippets, such as expressions, supplied to the operator at invocation time; at runtime, the operator executes them whenever needed for its behavior (see Section 4.2.2).
- The `output` clause assigns values to attributes in output tuples each time the operator submits data to an output stream (see Section 4.2.1).
- The `config` clause gives directives and hints that influence how the compiler builds this operator invocation, or how the runtime system executes it (see Section 4.2.3).

Each clause is optional, and in fact, most operator invocations only use few clauses. The syntax is:

```

opInvokeBody ::= '{'
                ( 'logic' opInvokeLogic+ )?
                ( 'window' opInvokeWindow+ )?
                ( 'param' opInvokeActual+ )?
                ( 'output' opInvokeOutput+ )?
                ( 'config' configuration+ )?
                '}'
opInvokeLogic ::= ( 'onTuple' ID | 'onPunct' ID | 'state' ) ':' stmt
opInvokeWindow ::= ID ':' expr+, ';'
opInvokeActual ::= ID ':' opActual ';'
opInvokeOutput ::= ID ':' ( ID '=' expr )+, ';'

```

Note that although the clauses do not have exactly the same syntax, most of them are special cases of comma-separated expression lists. Also note that the compiler will, in addition to these syntactic constraints, enforce semantic constraints on each clause. For example, only the windows described in Section 4.5 are legal in the `window` clause.

Language design rationale: SPL separates the body of operator invocations into clauses to make them easy to read. Clause order follows execution order: when a tuple arrives, logic executes first, followed by storing history in windows, executing parameter expressions, and finally assigning output attributes when the operator submits a tuple. Since the user rarely needs to see configuration options to understand what the code does, they have their own clause at the bottom of the operator invocation body. An IDE can facilitate writing structured operator invocation bodies, and can facilitate reading them by selectively folding clauses out of sight.

For SPADE users: In SPADE, logic was written in angle brackets (<...>) after a Functor stream head. Windows were specified between angle brackets as part of the input stream list. Parameters were written between square brackets ([...]), and, depending on whether the operator was built-in or user-defined, could be named or unnamed. Output assignments were written between curly braces ({...}) after the parameter section. Configuration options were written after an arrow (->...) at the end of an operator invocation. Here is the operator invocation defining stream `Sale` in SPADE:

```

stream Sale(buyer: String, seller: String, item: String) := Join      # 1
  #* omitted: Spade only allowed logic in Functors *# # logic      # 2
  ( Bid <time(30)>; Quote <count(50)> )                               # windows  # 3
  [ $1.item = $2.item & $1.price >= $2.price ]                     # parameters # 4
  { item := $1.item }                                              # outputs   # 5
  -> wrapper = gdb                                               # configs   # 6

```

4.2.1 Output Clause

The `output` clause assigns values to attributes in output tuples each time the operator submits data to an output stream. For example, “`output Sale: item = Bid.item, id = n;`” assigns values to attributes `item` and `id` in output stream `Sale`. The label, such as `Sale`, is an output stream name or alias, and opens the scope of that stream similarly to a Pascal “with” statement. The right-hand side is a comma-separated list of assignments to attributes.

For most operators, any attribute in the output stream that is not assigned by the output clause is implicitly copied from an attribute with the same name defined on an input stream. If there is not exactly one input attribute to copy, the compiler flags an error. Certain operators may have different forwarding behavior, specified by their operator model (operator models are described in Section 5.6.2).

Practical advice: Avoid calling `stateful` functions in the `output` clause.

4.2.2 Stream Parameters

The `param` clause contains code snippets supplied to the operator at invocation time. For example, the clause “`param match: Bid.item == Quote.item && Bid.price >= Quote.price;`” supplies a boolean expression as the `match` parameter to an operator, such as `Join`. At runtime, the operator executes the expression whenever needed for its behavior. For each time the operator fires, the expression might execute zero, one, or multiple times depending on the operator; but if it executes, it executes after the logic clause and before the output clause. Expressions are just one of a total of five kinds of parameter: (i) a comma-separated list of stream attribute names, (ii) a comma-separated list of expressions, (iii) a function name, (iv) an operator name, or (v) a type. The documentation of an operator specifies which parameters it expects, and the compiler checks that the right kind of entity is passed as a parameter. Primitive operators can only accept parameters of the forms (i) and (ii), composite operators can also accept parameters of the forms (iii), (iv), and (v).

4.2.3 Stream Configuration

The `config` clause specifies hints and directives that influence how the compiler builds this operator invocation, or how the runtime system executes it. Configuration options include deployment directives, debugging support, or high-availability and fault-tolerance configurations. For example, the `config wrapper: gdb` directs the runtime to invoke the operator through a wrapper program, in this case, `gdb` (the GNU debugger). The wrapper is a program that launches another program; in this case, `gdb` launches the execution container containing the operator instance. The IDE and tools for SPL will also provide external ways to specify configuration options without editing the source code. In that case, `config` clauses embedded in the source code specify a default, and external configuration options override it.

Practical advice: Embedded `config` clauses are useful during testing and development, but if you want your code to be reused on different infrastructure, avoid hard-wiring anything that is specific to your local infrastructure.

Language design rationale: We considered disallowing embedded configuration options altogether, but decided against that, because they were popular among SPADE users.

4.3 Stream Logic

The `logic` clause consists of local operator state that persists over the whole program execution, and statements that execute when an input port receives a tuple or a punctuation. For example:

```
stream<float64 runningAvg> FStr = Functor(IStr) { // 1
  logic state      : { mutable float64 avg = 0.0, alpha = 0.1; // 2
                      mutable int64 no = 0; } // 3
  onTuple IStr    : { if (no == 0) { // 4
                      avg = sample; //that's IStr.sample // 5
                    } else { // 6
                      float64 beta = 1.0 - alpha; // 7
                      avg = alpha * sample + beta * avg; // 8
                    } // 9
                      no++; //10
                } //11
  onPunct IStr    : no = 0; //12
  param filter    : true; //13
  output FStr     : runningAvg = avg; //14
} //15
```

The `state:` label introduces variable declarations. In this case, there is only one input port `IStr`, and the `onTuple IStr:` label introduces code to be executed each time a tuple arrives on that stream. It opens

up the scope of the input stream, so attributes can be accessed without qualifiers. SPL supports custom logic on each port and the `state` variables are shared among them. The code itself is typically a block, but can be any SPL statement, as described in Section 3.3. Access to operator invocation state is implicitly synchronized. The SPL compiler inserts the necessary locks to ensure that no two threads experience race conditions from concurrent access to state variables in the stream logic clause.

For SPADE users: SPADE supported custom logic only in invocations of the `Functor` operator; SPL supports it on invocations of all operators. Here is the SPADE version of the example:

```

stream FStr(runningAvg: Double) := Functor(IStr)           # 1
  < Double $avg := 0d; Double $alpha := 0.1d; Long $no := 0l; > # 2
  < #code executed when tuple arrives on IStr             # 3
    if($no = 0l) {                                       # 4
      $avg := sample; # that's IStr.sample               # 5
    } else {                                             # 6
      Double $beta := 1.0d - $alpha;                    # 7
      $avg := $alpha * sample + $beta * $avg;           # 8
    }                                                    # 9
    $no := $no + 1;                                     #10
  >                                                       #11
  [ true ] { runningAvg := $avg }                        #12

```

The first angle-bracket delimited section (`<...>`) in SPADE corresponds to the `state:` section in SPL, and the second angle-bracket delimited section in SPADE corresponds to the `portName:` section in SPL. SPL supports multiple logic sections in the same operator, one per port, which can all refer to the same state, e.g., to count tuples from either port.

4.4 Punctuation

A punctuation is a control signal that appears interleaved with the tuples in a stream. SPL supports two kinds of streams: a punctuation-free stream is an infinite sequence of tuples, whereas a punctuated stream is an infinite sequence of tuples interleaved with punctuations. A punctuation separates groups of consecutive tuples on a stream to create window boundaries. For example, consider an invocation of an `Aggregate` operator with a punctuation-based window. Each time this operator invocation receives a punctuation, it aggregates the accumulated tuples since the last punctuation.

An output port of an operator invocation can either generate, remove, or preserve punctuation markers. Examples for punctuation-generating ports include the output from the `Join`, `Aggregate`, and `Punctor` operators. Punctuation-free ports guarantee that their output stream is not punctuated, and punctuation-preserving ports will forward punctuations, but only if there is a unique punctuated input stream.

An input port of an operator invocation can either be punctuation-oblivious or punctuation-expecting. Ports are oblivious to punctuation if there is no window or any window other than punctuation-based windows. Ports with punctuation-based windows expect that the input stream be punctuated. Punctuation-expecting ports must be connected to exactly one input stream. The compiler enforces these rules of punctuation by reporting errors when they are violated.

4.5 Windows

A window is a logical container for tuples recently received by an input port of an operator. Some relational operators rely on windows for their operation; for example, in `Join(A;B)`, when a tuple arrives on port A, SPL updates the window on port A and matches the new tuple against each tuple in the window on port B. The following example illustrates different kinds of windows on different input ports; see later subsections for an exhaustive description.

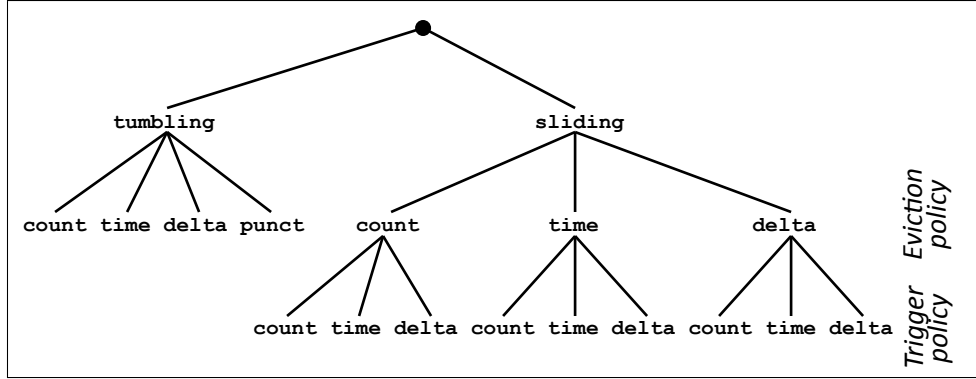


Figure 4: Window types.

```

stream<float32 c> A = MyOperator(C; X, Y as D) { // 1
  window C : sliding, time(10), count(5);      // 2
  D : tumbling, delta(i, 10);                 // 3
}                                              // 4

```

Port C uses a *time*-based *sliding* window over the last 10 seconds, which triggers the operator’s logic after every 5 tuples. Port D, which interleaves tuples from streams X and Y, uses an *attribute-delta* based *tumbling* window, where attribute *i* in the oldest and newest tuple differs by no more than 10. Attribute names are scoped by the port label in the window clause: for example, *i* corresponds to *D.i*. All expressions in windows must be compile-time constants. The following subsections describe available windowing policies in detail.

For SPADE users: In SPADE, the example would have looked like this:

```

stream A(c : Float)                                # 1
  := MyOperator(C <time(10), count(5)>; X, Y <attrib(i, 10)>) # 2
  [] {}                                             # 3

```

SPADE kept window specifications in the operator invocation head next to the input streams, SPL keeps them in the body. Another difference is that SPADE window support was not uniform over all operators.

4.5.1 Tumbling Windows

A tumbling window stores incoming tuples until the window is full, then executes the operator logic, and finally flushes all stored tuples, before starting over from scratch. A tumbling window is specified by providing just an eviction policy, in other words, the window size, which determines when the window is “full”. SPL offers four eviction policies:

- **count(*n*)**: Count-based window, storing *n* tuples.
- **time(*n*)**: Time-based window, storing tuples arriving over a period of *n* seconds.
- **delta(*a*, *n*)**: Attribute-delta based window, storing tuples until the difference between attribute *a* of the oldest and the newest tuple exceeds *n*. The attribute is typically monotonically increasing, and is often a **timestamp**. That way, an attribute-delta based window can emulate a time-based window, but instead of using current time on the local machine, it uses a time attribute streamed along with the data.
- **punct()**: Punctuation-based window, storing tuples until the next punctuation arrives. An upstream operator can generate punctuations, and a downstream operator can use them for windowing.

4.5.2 Sliding Windows

A sliding window is specified by providing both an eviction policy and a trigger policy. For example, `sliding, time(10), count(5)` means the eviction policy keeps the window size to 10 seconds, and the trigger policy executes the operator logic after every 5 tuples. A sliding window expels the oldest tuples to maintain the window size when new tuples are coming in, and it executes the operator logic each time it reaches a trigger. Unlike for tumbling windows, the eviction and trigger policies for sliding windows are independent from each other, so an old tuple can still be in the window if the logic of the operator gets triggered again before the tuple is evicted. The available policies for eviction and trigger are count-based, time-based, and attribute-delta based, but not punctuation-based. In other words, the window can use one of three eviction policies and one of three trigger policies, yielding a total of 9 combinations. If the trigger policy is omitted, it defaults to `count(1)`; for example, “`sliding, count(3)`” is equivalent to “`sliding, count(3), count(1)`”.

4.5.3 Grouping

The flag `perGroup` in the window clause specifies that instead of maintaining just one window per port, the operator invocation maintains one window per port per group. A group is a set of tuples that have the same values for grouping attributes. For example, with the parameter `param groupBy: x, y`, tuples belong to different groups if they differ in either attribute `x` or `y`. When a new tuple arrives on a port, it is routed to the window for its own group. Each group window uses an eviction policy and trigger policy as described above for tumbling and sliding windows. Grouping is frequently used in conjunction with aggregation; Section 5.4 gives a concrete example. While the `perGroup` flag is a language feature, the `groupBy` parameter is not. It is operator-specific, but by convention, most operators implement it to follow the guidelines described here.

4.5.4 History

Expressions in parameters or output assignments of operator invocations can refer directly to tuples received in the past. History access conceptually treats a window as a list of tuples, which can be subscripted by using the input port name. For example, in

```
stream<int32 id, int32 diff, string8 name> Out = Functor(In) { // 1
  window In      : sliding, count(2);                       // 2
  param filter   : name != In[1].name;                     // 3
  output Out     : diff = id - In[1].id;                   // 4
}                                                         // 5
```

the expression `In[1].name` refers to a historical tuple on port `In`, 1 tuple in the past, and extracts the `name` attribute from it. The current tuple is at index 0, so the above filter parameter could also have been written as `In[0].name != In[1].name`. History access does not support slicing.

For SPADE users: SPADE used the hat (^) notation for history access. Here is the SPADE version of the example:

```
stream Out(id: Integer, diff: Integer, name: String) # 1
:= Functor(In) # 2
  [ name != ^1.name ] # 3
  { diff := id - ^1.id } # 4
```

4.6 Error Semantics

Even though SPL has been designed to perform most of its error checking at compile time, runtime errors can still happen in some cases. This includes an invalid subscript for a list, blob, string, or map; a size mismatch in the operands of mapped expression operators; division by zero; or exceptions in libraries (such as C++ or Java exceptions).

When any of these occur, the entire partition (execution container) enclosing the operator invocation dies, writes the error to a log file, and stops accepting new tuples. The SPL distribution may come with additional tools and options, such as a debugging tool, or a “resilient” option where the partition just logs the error and ignores one tuple, but then continues executing.

To help the user with error handling, SPL provides APIs for assertions and logging. Assertions look like calls to an `assert` function with the following signature:

```
stateful void assert(boolean condition, string8 message) { /*native*/ } // 1
stateful void assert(boolean condition) { /*native*/ } // 2
```

They are marked `stateful`, since logging or halting is a side effect. However, unlike regular function calls, and like assertions in C or Java, SPL assertions can be disabled. When an SPL program is compiled with assertions disabled, their parameter expressions are not evaluated, saving time and possibly preventing side effects.

Besides assertions, SPL also provides two other features to help testing and error handling. One is a log function, with the following signature:

```
stateful void log(enum{error,info,debug,trace} logLevel, // 1
                 string8 message, string8 aspect) { /*native*/ } // 2
```

The value `error` means that the log is unmaskable, whereas the values `info`, `debug`, and `trace` mean that logging should be performed when the user requested least verbose, more verbose, or most verbose logging, respectively.

Besides assertions and logging, SPL also provides an `abort()` function, which unconditionally terminates the application instance, even when compiled with assertions disabled.

Language design rationale: We considered adding full-fledged exception handling to SPL, with `try/catch/finally` statements. But we decided against that because if the user anticipates the exception, it is easy to check with if-statements, whereas if the exception is unanticipated, the logging functionality that SPL provides is probably more helpful than exception handlers a user could write.

5 Operators

A stream operator can be invoked to transform input streams to output streams. SPL supports two kinds of operators: primitive operators and composite operators. A composite operator contains a reusable stream subgraph. An SPL application can be viewed as a hierarchy of operator invocations, where the leaves are primitive operators, each level groups graphs of operators into composite operators, and the root is a main operator to be deployed as an application instance on the streaming middleware. All operator invocations follow the syntax described in Section 4, irrespective of whether they invoke primitive or composite operators.

This section is about how to define new operators that can then be invoked to define streams. Most streaming languages are tailored towards a single application domain. For example, StreamSQL targets the stream-relational domain [2]. SPL, on the other hand, is designed to address a diverse set of domains. Key to this diversity is that SPL users can define their own operators, either in SPL itself or by reusing legacy code written in C++ or Java.

For SPADE users: SPADE only supported primitive operators, no composite operators.

5.1 Composite Operators

A composite operator contains a reusable stream subgraph, which can itself be invoked to define streams. Each composite operator definition has a head and a body. The head lists input and output ports, and the body specifies the graph and its parameters. The syntax is:

$$\textit{compositeDef} ::= \textit{compositeHead} \textit{compositeBody}$$

The head of a composite operator definition names the operator and lists its ports. For example:

```

composite M (output K, L; input G, H) { // 1
  /*...*/                               // 2
}                                          // 3

```

The composite operator M has two output ports, K and L, and two input ports, G and H. The syntax is:

```

compositeHead ::= 'composite' ID ( '(' compositeInOut+; ')' )?
compositeInOut ::= ( 'input' | 'output' ) ( streamType? ID )+,
streamType ::= 'stream' '<' tupleBody '>'

```

The body of a composite operator definition specifies what kind of parameters it accepts, and contains a stream graph. Besides these two clauses (**param** and **graph**), the body can also contain a **type** clause with type definitions, a **var** clause with shared variable definitions, and a **config** clause with compiler hints and directives. All five clauses are optional. Here is the syntax:

```

compositeBody ::= '{'
  ( 'param' compositeFormal+ )? # Section 5.3
  ( 'type' typeDef+ )? # Section 5.1.2
  ( 'var' sharedVarDef+ )? # Section 5.5
  ( 'graph' opInvoke+ )? # Section 5.1.1
  ( 'config' configuration+ )? # Section 5.1.3
  '}'

```

Practical advice: We encourage you to use composite operators whenever you write a large program, because it becomes easier to understand when you break it down into smaller subgraphs. We also encourage you to make your composite operators reusable, and put them into libraries, so they can be invoked in different programs. Finally, you can use parameterizable composite operators to implement distributed computing patterns such as map-reduce.

Language design rationale: Besides the obvious structuring and reuse advantages, composite operators help visualizing both static and dynamic views of programs. And composite operators serve as a foundation for higher-level stream programming, for example, with ontology-based planning. The syntax for composite operators separates the head from the body, to make it easy to see port connections, and splits the body into multiple clauses, to make it easier to read and enable folding editors.

5.1.1 Graph Clause

The graph clause of a composite operator describes a stream data flow subgraph, which can then be expanded in different contexts when the operator is invoked. Here is an example:

```

composite M (output K, L; input G, H) { // 1
  graph stream<int32 x> I = O(G) { } // 2
    stream<int32 x> J = P(H) { } // 3
    stream<int32 x> K = Q(I; J) { } // 4
    stream<int32 x> L = R(J) { } // 5
} // 6
(stream<int32 x> C; stream<int32 x> D) = M(A; B) { } // 7
(stream<int32 x> E; stream<int32 x> F) = M(A; B) { } // 8

```

Figure 5 illustrates the graph topology in this example. The original graph contains two invocations using composite operator M: once to transform A and B into C and D, and the other time to transform A and B into E and F. The head of the composite operator definition (output K, L; input G, H) defines input

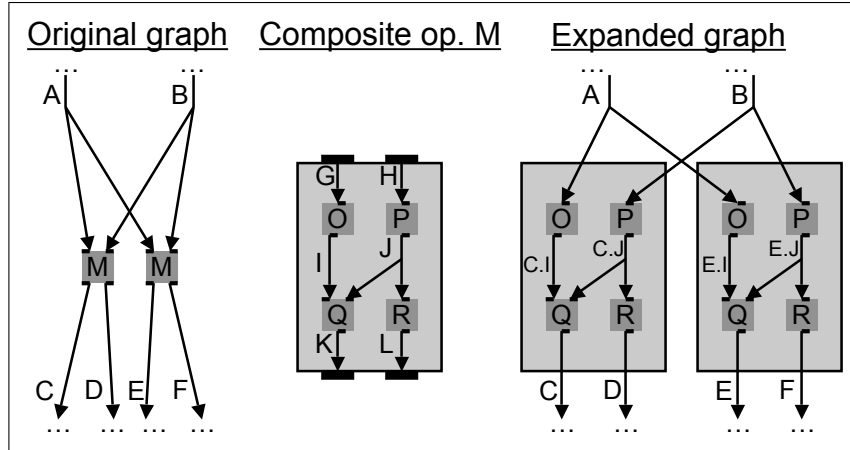


Figure 5: Composite operator expansion example.

and output ports. When the composite operator gets used, actual input and output streams are bound to these ports; for example, for the first invocation, the bindings are $G=A$, $H=B$, $K=C$, and $L=D$. In the expanded graph, intermediate streams such as I and J in the example are duplicated and renamed $C.I$, $C.J$, $E.I$, and $E.J$. Section 5.2 elaborates on SPL’s stream naming scheme. (For language experts: The compile-time expansion of composite operators behaves similarly to macros, but is “hygienic”, since it generates new names to prevent unintended name capture.)

5.1.2 Type Clause

A type definition gives a name to a type. For example:

```

composite Comp ( /*...*/ ) { // 1
  type IntegerList = list<int32>; // 2
  MySchema = string8 x, IntegerList y; // 3
  graph /* ... */ // 4
    stream<MySchema, tuple<int32 z>> SenSource = FileSource() { // 5
      param file: "SenSource.dat"; format: csv; // 6
      config placement: host(SourcePool[0]); // 7
    } // 8
  } // 9

```

The type `MySchema` is defined as a tuple, leaving out the optional constructor `tuple<...>`. The type can be used in operator invocations. In this case, it is used in “`MySchema, tuple<int32 z>`”, which extends `MySchema` by adding another attribute “`int32 z`”. Tuples on stream `SenSource` have the type `tuple<string8 x, list<int32> y, int32 z>`, in other words, all attributes of `MySchema`, plus the additional `z` attribute. SPL also allows using the name of a stream to refer to its type. For example, if there is a stream `A`, then `stream<A> B` declares a stream `B` with the same type as stream `A`, and `stream<A, tuple<int32 i>> C` declares a stream `C` with tuples that extends `A` by an extra attribute `i`.

The syntax of a type definition is:

$$\text{typeDef} ::= \text{'public'}? \text{ID} \text{'=' (type | tupleBody) \text{';'}}$$

Public type definitions (with the `public` modifier) may be used from anywhere in the program, whereas private type definitions (without `public` modifier) are only visible in the same composite operator body that

defines them. Public type definitions must not depend on any operator parameters. To access a type from a different composite operator, qualify it with its operator, e.g., `my.namespace::MyOperator.MyType`.

For SPADE users: Here is what the same example looked like in SPADE:

```
[Typedefs]                                # 1
typedef Integers IntegerList                # 2
[Program]                                   # 3
vstream MySchema(                           # 4
  sx : String, dx : Double, fx : Float, lx : Long,      # 5
  ix : Integer, iy : Integer, bx : Boolean )           # 6
stream SenSource(schemaFor(MySchema), id: Integer)    # 7
:= Source() ["file:///SenSource.dat", nodelays, csvformat]{} # 8
-> node(SourcePool, 0)                             # 9
```

SPADE supported two different kinds of type definitions. Tuple types were defined in the `[Program]` clause with the `vstream` keyword, whereas all other types were defined in the `[Typedefs]` clause with the `typedef` keyword. SPL unifies the two. Also, in SPADE, you had to use the `schemaFor` function to get the type of a stream, whereas in SPL, you can simply use the name of a stream to stand for its type.

5.1.3 Composite Operator Configuration

In SPL, a configuration option gives a directive or a hint to the compiler, for example, to specify deployment, debugging support, or transport mechanisms. As we saw in Section 4.2.3, configurations can be embedded in the source code, or externally at the IDE level. This section describes the `config` clause of a composite operator, which configures all operator invocations in its graph at once. For example:

```
type T = int32 i;                            // 1
stream<T> S1 = Op1(S0) { }                    // 2
composite Comp(output Out; input In) { // 3
  graph stream<T> S2 = Op2(In) { } // 4
    stream<T> S3 = Op3(S2) { } // 5
    stream<T> Out = Op4(S3) { } // 6
  config wrapper : gdb;                      // 7
}                                              // 8
stream<T> S4 = Comp(S1) { }                  // 9
stream<T> S5 = Op5(S4) { }                  //10
```

The `config wrapper: gdb` directs the runtime to invoke all operators used in the graph `Comp` using `gdb` (the GNU debugger) as a wrapper program.

(For SPADE users: SPADE specified configuration options after `->` in operator invocations. Since SPADE had no composite operators, you had to specify the configuration repeatedly on each individual operator invocation:

```
vstream T(i : Integer)                        # 1
stream S1(schemaFor(T)) := Op1(S0) [ ] { }   # 2
stream S2(schemaFor(T)) := Op2(S1) [ ] { } -> wrapper=gdb # 3
stream S3(schemaFor(T)) := Op3(S2) [ ] { } -> wrapper=gdb # 4
stream S4(schemaFor(T)) := Op4(S3) [ ] { } -> wrapper=gdb # 5
stream S5(schemaFor(T)) := Op5(S4) [ ] { }   # 6
```

Specifying configuration options separately on each operator invocation is still allowed in SPL, but doing it at the composite operator level can prevent repetitive code.)

When there are configuration options both on the definition and the invocation of a composite operator, the invocation `config` overrides the definition `config`. Consider this example:

```

namespace com.ibm.spl.MyNs; // 1
composite MyOp(output V; input U) { // 2
  graph // 3
    stream<int32 i> V = Functor(U) { // com.ibm.spade.relational.Functor // 4
      config // 5
        wrapper: sdb; // instance config, rank 1 // 6
      } // 7
    config // 8
      wrapper: gdb; // definition config, rank 3 // 9
  } //10
} //11
composite Main { //11
  graph //12
    stream<int32 i> X = FileSource() { /*...*/ } //13
    stream<int32 i> Y = MyOp(X) { //14
      config //15
        wrapper: console; // instance config, rank 2 //16
      } //17
    config //18
      wrapper: valgrind; // definition config, rank 4 //19
  } //20
}

```

The operator instance `Y.V` has the config `wrapper: sdb`. In general, a config for a primitive operator instance is resolved by first going up in the operator instance hierarchy and checking for configs attached to operator instances, then going up in the operator instance hierarchy and checking for configs attached to composite operator definitions.

Practical advice: While this section illustrates the `config` clause with some examples, these examples are not themselves part of the language specification. Configuration options, along with other runtime behaviors, will be defined in separate documents (for example, the InfoSphere Streams “SPL Config Reference”).

5.2 Stream and Operator Instance Names

SPL has rules for giving names to operators, operator instances, streams, and ports. For example:

```

(stream<T> Stream1 as S1; stream<T> Stream2 as S2) as OpInstance // 1
  = MyOp(Stream3 as S3; Stream4 as S4) { /*body of opInvoke*/ } // 2

```

The name `MyOp` identifies the invoked operator, whereas the name `OpInstance` identifies the operator instance. Names `Stream1`, `Stream2`, `Stream3`, and `Stream4` identify streams, whereas names `S1`, `S2`, `S3`, and `S4` identify ports and can only be used in the body of the operator invocation.

If the `as`-clause with the explicit operator instance name (as `OpInstance` in the example) is missing, then the output stream names serve as operator instance name. For example, in `stream<T> A = Foo1(I){ }`, the name `A` identifies both a stream and the operator instance. If there are multiple output streams, then all of them are synonyms for the operator instance. For example, in `(stream<T> A; stream<T> B) = Foo2(I){ }`, both names `A` and `B` identify different streams but the same operator instance. If there are zero output streams, then the operator instance name is mandatory. For example, `() = Foo0(I){ }` is a syntax error, whereas `() as FooInstance = Foo0(I){ }` is valid.

The above rules describe simple names for streams and operator instances in the main composite operator; streams and operator instances nested in other composite operators are identified by qualified names. (The difference between a simple name and a qualified name is that a qualified name has qualifier tokens “.” or “:.”). SPL uses an outside-in naming scheme: given a stream with the simple name `Z` in a composite operator instance with the name `X.Y`, the qualified stream name is `X.Y.Z`. The streams `C.I`, `C.J`, `E.I`, and `E.J` in Figure 5 are good examples for this outside-in naming scheme. Qualified stream or operator instance names are used in dynamic application composition and in external tooling, such as the IDE or the debugger.

5.3 Operator Parameters

Different invocations of the same operator can yield different expansions, by providing actual parameter values for formal parameter placeholders declared in the operator definition.

5.3.1 Operators as Parameters to Composite Operators

A composite operator encapsulates a subgraph and some of the operators in that subgraph can be given to the composite operator as parameters at the time of invocation. Thus, composite operators can act as generic graph stencils, which are useful for implementing high-level distributed well-structured computation paradigms such as map-reduce [3]. The following code is a parametric version of the composite operator M defined earlier:

```

composite M (output K, L; input G, H) { // 1
  param operator $Q; //operator (primitive or composite) // 2
  graph stream<int32 x> I = O(G) { } // 3
    stream<int32 x> J = P(H) { } // 4
    stream<int32 x> K = $Q(I; J) { } // 5
    stream<int32 x> L = R(J) { } // 6
} // 7
(stream<int32 x> C; stream<int32 x> D) = M(A; B) { param Q: S; } // 8
(stream<int32 x> E; stream<int32 x> F) = M(A; B) { param Q: T; } // 9

```

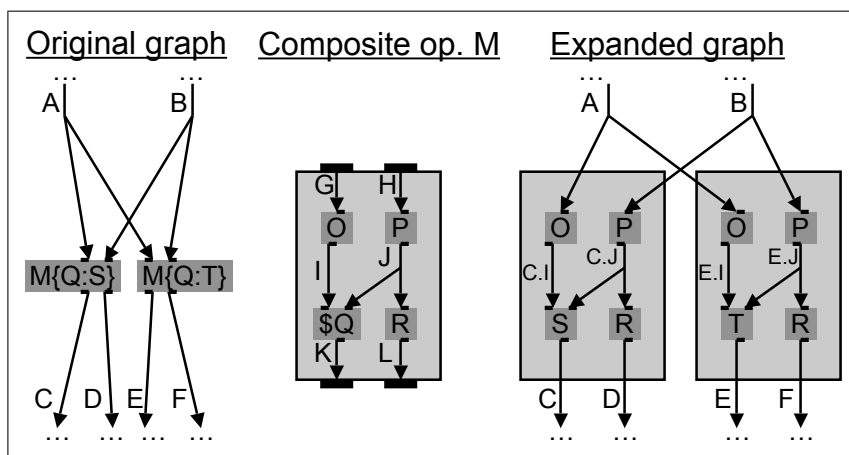


Figure 6: Composite operator parameter example.

Figure 6 illustrates how the composite operator and its parameter get expanded. The first expansion, on the left, replaces the placeholder parameter `$Q` by the actual operator `S`. The second expansion, on the right, replaces the placeholder parameter `$Q` by the actual operator `T`.

5.3.2 Other Operator Parameters

Both composite and primitive operators can be customized by parameters such as the name of an attribute in a connected stream, or an expression to be used to filter data. The following example illustrates this for a composite operator `Mogrifier`:

```

composite Mogrifier(output Out; input In) { // 1
  param attribute $attr; // 2
  expression $expr; // 3

```



```

graph stream<In> Out = Functor(In) { // 4
    param filter: $attr < $expr; // 5
    output Out : $attr = $attr + 1; // 6
} // 7
} // 8
stream<int32 i, int32 j> A = FileSource() { /*...*/ } // 9
stream<int32 i, int32 j> B = Mogrifier(A) { //10
    param attr: i; //11
    expr: 2 * j; //12
} //13

```

Binding the actual attribute `i` to placeholder `$attr` and the actual expression `2*j` to placeholder `$expr` yields the following expansion:

```

stream<int32 i, int32 j> A = FileSource() { /*...*/ } // 1
stream<int32 i, int32 j> B = Functor(A) { // 2
    param filter: i < 2 * j; // 3
    output B : i = i + 1; // 4
} // 5

```

5.3.3 Operator Parameter Passing Semantics

Operator parameters in SPL behave similarly to macro expansion, but the semantics are designed to prevent macro mistakes common in languages like C. In particular, SPL defines a name resolution rule to prevent accidental name capture and a syntactic confinement rule to prevent unintended re-association.

The **name resolution rule** for names in actual parameters is that the SPL compiler resolves them in the context of the operator invocation, not in the context of the operator definition. As we will see in Section 5.5, you can define variables in the `var` clause of a composite operator. The following example illustrates how SPL resolves names when there are multiple choices:

```

composite Comp1(output O1; input I1) { // 1
    var int32 s_t = 1; // 2
    graph stream<int32 x> O1 = Comp2(stream<int32 x> I1) { param cond: x < s_t; } // 3
} // 4
composite Comp2(output O2; input I2) { // 5
    param expression $cond; // 6
    var int32 s_t = 2; // 7
    graph stream<I2> O2 = Functor(stream<int32 x> I2) { param filter: $cond; } // 8
} // 9

```

The actual value for parameter `cond` in Line 3 is `x < s_t`, which contains two names `x` and `s_t`. The names are resolved in the context of the operator invocation in Line 3, where `x` is `I1.x` and `s_t` is `Comp1.s_t`, which is 1. If the names were resolved in the context of the operator definition, then, since `$cond` is used in Line 8, `x` would be `I2.x` and `s_t` would be `Comp2.s_t`, which is 2. This would make no difference for `x`, because `I1.x` and `I2.x` are the same, but it would make a difference for `s_t`, because `Comp1.s_t` and `Comp2.s_t` differ. By resolving `s_t` to `Comp1.s_t`, the language shields the author of `Comp1` from implementation details of `Comp2`. The rule “resolve names in the invocation, not the definition” supports encapsulation, because it hides implementation details of the operator, rather than allowing them to leak out by capturing names. (For language experts: SPL operator parameters have “call-by-name” semantics. But note that they are used with partial evaluation: everything is expanded at build time. If the expanded expression only uses build-time values, it is fully evaluated at build time. On the other hand, if the expanded expression also involves runtime values, only the build-time subexpressions are partially evaluated, and the rest is evaluated at runtime.)

The **syntactic confinement rule** for SPL operator parameters is that argument expressions are implicitly parenthesized. The implicit parentheses “confine” them from associating in unexpected ways when they get used. A corollary of this rule is that you cannot pass something that does not start out confined; for example, an incomplete expression like “x +” is not valid as an actual parameter to an operator. The following example illustrates the syntactic confinement rule:

```

composite M(output Out; input In) { // 1
  param expression<int32> $q; // 2
  graph stream<float32 percent> Out = Functor(In) { // 3
    output Out: percent = float32(100 * $q); // 4
  } // 5
} // 6
stream<int32 x> A = FileSource() { /*...*/ } // 7
stream<float32 percent> B = M(A) { param q: x - 1; } // 8

```

The actual parameter `x - 1` on Line 8 is substituted for the parameter `$q` on Line 4. Because of the syntactic confinement rule, `100 * $q` associates like `100 * (x - 1)`, not like `(100 * x) - 1`. As with the name resolution rule, the syntactic confinement rule leads to better encapsulation: when invoking a operator, you need not worry about precedence context in the operator definition. This prevents unintended results. (For language experts: Parameter expressions yield a subtree in the AST and will not spill out of this subtree during expansion.)

Language design rationale: We chose this approach for operator customization, because it makes uses of operators concise, provides flexible typing, and naturally generalizes to primitive operators. It also simplifies the implementation, since certain entities (operator, type, and function parameters) are fully resolved at compile time, and need not be treated as first-class values at runtime.

5.3.4 Operator Parameter Types and Default Values

A parameter declaration in a composite operator definition must specify a metatype, and can optionally specify a default value. For example:

```

composite ThresholdFilter(output Filtered; input Unfiltered) { // 1
  param expression<float32> $threshold : 5.0; // 2
  graph stream<float32 x> Filtered = Functor(Unfiltered) { // 3
    param filter : x > $threshold; // 4
  } // 5
} // 6

```

The declared type of parameter `$threshold` is `expression<float32>`. This type serves as documentation and helps produce better error message. For example, if the composite operator is used with the wrong type:

```
stream<float32 x> B = ThresholdFilter(A) { param threshold: "abc"; }
```

then the SPL compiler gives the error message:

```
type mismatch: $threshold expects a float32, but "abc" is a string8
```

Besides the metatype `expression<T>`, where `T` is a concrete runtime type, a parameter type can also be just `expression` without a type parameter, indicating that an expression of any type works as long as the expansion type-checks. In addition to `expression`, parameter type declarations can also use other metatypes:

```

metaType ::= 'attribute'
           | 'expression' typeArgs?
           | 'function' typeArgs?
           | 'operator'
           | 'type'
typeArgs ::= '<' type+ '>'

```

For example, `attribute` is the name of one or more attributes in an input stream, and `operator` is a primitive or composite operator. The metatype `function` also has optional type parameters. For example, `function<void,int32,int32>` accepts any function returning `void` with two formal of type `int32`.

Still using the same example, the declared default value of parameter `$threshold` is 5.0. This default value makes it easier to use the composite operator, because the user can omit the parameter. For example, the following invocation is short yet complete:

```
stream<float32 x> B = ThresholdFilter(A) { }
```

This is equivalent to providing the default. Besides making operators more reusable, default values also serve as documentation, because they give an example for the kind of value expected by a parameter.

The syntax for composite operator parameter declarations is:

```
compositeFormal ::= metaType ID ( ':' opActual )? ';' ;
```

5.3.5 Example for Composite Operator with Logic

Section 4.3 describes the logic clause in an operator invocation, which is an expressive mechanism for writing imperative code in SPL without having to write native (Java or C++) code. Section 5.1 describes composite operators, which are a mechanism for creating reusable operators in SPL. This subsection does not introduce any new language features, it merely points out that by combining logic with composite operators, you can write expressive and reusable operators in SPL without native code. Here is an example:

```

composite KeyedMapper(output Output; input Mappings, Queries) { // 1
  param type      $MapType; // 2
  attribute $key; // 3
  attribute $val; // 4
  graph stream<Output> Output = Custom(Mappings; Queries) { // 5
    logic // 6
      state          : mutable $MapType m = { }; // 7
      onTuple Mappings : m[$key] = $val; // 8
      onTuple Queries  : if ($key in m) // 9
                          submit(Output, {key=$key, val=m[$key]}); //10
  } //11
} //12
stream<string8 key, int32 val> Lights = KeyedMapper(Sensors; Queries) { //13
  param MapType : map<string8, int32>; //14
  key           : Sensors.color; //15
  val           : intensity; //16
} //17

```

The `state` clause (Line 7) declares a map variable used by the per-port logic. There are two input ports: `Mappings` and `Queries`. The `Mappings` port logic (Line 8) records or overwrites a key-value pair in the map and does not produce any output. The `Queries` port logic (Lines 9+10) answers queries by consulting the

most recent value for that key, as recorded by the `Mappings` logic. The `Queries` port logic sends its outputs, if any, to the `Output` stream by calling the `submit` function. The `submit` function is available in the `Custom` operator, which can receive and send any number of streams, and does not do anything by itself. `Custom` offers a blank slate for customization. The functions available in `onPunct` port logic are:

```
enumNonExistent,WindowMarker,EndOfStreamMarker currentPunct();
<tuple T> void submit(T, enumNonExistent,WindowMarker,EndOfStreamMarker);
```

5.4 Domain Toolkits

SPL groups operators in namespaces (see Section 6). A toolkit is a set of namespaces that logically organize a collection of operators belonging to the same application domain. For example, the stream-relational toolkit groups operators for the stream-relational domain, similarly to `StreamSQL` [2]. Operators in the stream-relational toolkit include `Aggregate`, `Join`, `Sort`, `Functor` (which subsumes `filter` and `project`), and others. Another SPL toolkit is responsible for I/O adaptation, i.e., ingesting and externalizing data, with operators like `FileSource`, `FileSink`, `Import`, and `Export`.

An example for an operator from the relational domain is `Aggregate`. It supports aggregating values from a window using aggregators like `Any`, `Min`, `Sum`, etc. Here is an example invocation of `Aggregate`:

```
stream<string8 ticker, string8 exchange,                // 1
      decimal64 minprice, decimal64 maxprice, decimal64 avgprice, // 2
      decimal64 sumvolume, decimal64 vwap>             // 3
  PreVwap = Aggregate(TradeFilter)                     // 4
{                                                       // 5
  window TradeFilter : sliding, count(4), count(1), perGroup; // 6
  param  groupBy     : ticker, exchange;               // 7
  output PreVwap     : ticker   = Any(ticker), exchange = Any(exchange), // 8
                    minprice = Min(price),  maxprice = Max(price), // 9
                    avgprice = Avg(price),  sumvolume = Sum(volume), //10
                    vwap     = Sum(price * volume); //11
}                                                       //12
```

The `Aggregate` operator requires a window, and in the example, the window has a flag `perGroup` that specifies that the window and the aggregation occur separately for each attribute in the `groupBy` parameter. Section 4.5 describes windows in SPL. In this example, `groupBy` is `ticker/exchange`, which means that each incoming tuple is added to the window corresponding to its own `ticker` and `exchange` value, and triggers an output tuple that aggregates that window. Since all tuples in the window have, by definition, the same values for `ticker` and `exchange`, aggregation with `Any` in Line 9 yields unique values for that pair of attributes. All other outputs must also be aggregated. As the output assignment `vwap = Sum(price * volume)` illustrates, the argument to an aggregator can be an expression, which can involve more than one input attribute. In fact, the argument can even involve a function call, such as `y = Avg(pow(x, 3))`. When aggregators and functions can have the same name, such as `y = Max(Max(i, j))`, Spade treats the outer call as an aggregator and the inner call as a function. To make the code more readable, the user can disambiguate them by explicit qualifiers, e.g., `y = Aggregate::Max(MathLib::Max(i, j))`. But we encourage avoiding such name collisions in the first place by starting function names with lower-case letters and aggregator names with upper-case, e.g., `y = Max(max(i, j))`.

Practical advice: The semantics of aggregation and `groupBy` are defined by the library, not by the language. A separate document will describe the library APIs in more detail.

Language design rationale: The `Aggregate` operator's `groupBy` parameter is a declarative description of functionality that other languages might achieve in a more imperative way. We picked the declarative style because it is familiar to people who know SQL.

For SPADE users: SPADE had BIOPs (built-in operators); SPL replaces BIOPs by operators in the standard library. Some operators in the SPL library are composite operators, others are primitive operators,

but since they are invoked in the same way, that distinction should be transparent to the user. Besides the SPADE BIOPs, SPL also introduces a few new operators, e.g., `Import`, `Export`, and `Custom`. Other operators will be improved in SPL. For example, SPL has more general support for expressions inside aggregations than SPADE.

5.5 Shared Variables

Shared variables contain state that is shared between multiple operators, which can even belong to multiple jobs or be exposed on a dashboard for a user to monitor an application instance. Note that sharing data in a distributed system incurs a high cost and can make synchronization difficult. That said, when used responsibly, this feature is very powerful. For example, shared variables can store control state in adaptive applications, or classifier models in learning applications, or distributed data in map-reduce applications [3].

The syntax for a shared variable definition is:

```

sharedVarDef      ::= sharedVarModifier* type ID ( '=' expr )? sharedVarConfigs
sharedVarModifier ::= 'public' | 'static' | 'mutable'
sharedVarConfigs  ::= ';' | '{' 'config' configuration+ '}'

```

Static shared variables (with the `static` modifier) have only one instance per application scope, whereas instance shared variables (without `static` modifier) are instantiated separately each time their operator is invoked. Since static variables do not belong to any particular operator invocation, they can not depend on operator parameters. Instance variables, on the other hand, can depend on parameters of their operator's invocation.

Public shared variables (with the `public` modifier) may be used from anywhere in the system, whereas private shared variables (without `public` modifier) are only visible in the same composite operator body that defines them. Only static variables may be public, all instance variables are private. To access a shared variable from a different composite operator, qualify it with its operator, e.g., `com.ibm.someNameSpace::DefiningOp.s_var`.

Mutable shared variables (with the `mutable` modifier) can be written or modified, whereas immutable shared variables (without `mutable` modifier) never change throughout the program execution. Immutable variables of composite types (such as a list) are deeply immutable: all of the contents of the variable (including, for example, list elements) are constant.

Shared variables can optionally have an initializer. For immutable shared variables, the initializer is mandatory. If there is a circular dependency between variable initializers, the compiler produces an error message.

The following example illustrates two shared variables, `s_thresh` and `s_m`:

```

composite CompositeWithStateKeepers(output Out; input In) { // 1
  var int32 s_thresh = 10; // 2
  public static mutable map<string8, int32> s_m { // 3
    config lifetime      : eternal; // 4
    consistency         : causal; // 5
    sizeHint            : 128 * 1024 * 1024; // 128 MB // 6
    writesPerSecond    : 5; // 7
    readsPerSecond     : 500; // 8
  } // 9
  graph stream<In> Tmp = Classifier(In) { param usingMap : s_m; } //10
  stream<In> Out = Functor(Tmp) { param filter : x > s_thresh; } //11
} //12

```

Shared variables can optionally be annotated with a number of configuration options. The example illustrates directives for consistency (`atomic > causal > loose`) and lifetime (`eternal > transient`), and

hints for size and access load, which the runtime system uses to pick an appropriate implementation and to allocate resources to implement necessary system support for the shared variables. The exact set of configuration options will be decided as the implementation progresses.

SPL statically enforces some limitations on shared variable usage to help the user and the compiler reason about consistency. To access a shared variable from a different composite operator than the one that defines it, it must be public and static. To access a shared variable from a primitive operator, the operator model must mention that the operator is stateful, and the operator invocation must mention the variable as part of a parameter expression. And finally, to access a shared variable from a function, it must be passed as a parameter; in a function body, shared variables are not in scope under any other name. Note that functions can only modify parameters that are declared mutable (see Section 3.4).

The implementation of shared variables in the middleware is called *state keepers*. While the language itself does not provide locks or synchronization primitives, the library will expose an API for distributed mutex and condition variables in the state keeper implementation.

Practical advice: Shared variables pose performance and consistency challenges. Therefore, you should use shared variables only when absolutely necessary. If you do use them, you should only declare them public or mutable when you have to. In this language reference, all shared variable names start with “s_” to make them stand out in the source code. While this naming convention is optional, we encourage you to follow it in your code as well.

Language design rationale: Given their performance and consistency challenges, we debated whether to support shared variables in SPL at all. We decided to put them in because there are several compelling use cases for them. If we had omitted the language feature, people would have emulated it by hand, and each such work-around would have been hard for other users to understand and for tools to reason about.

5.6 Primitive Operators

The result of expanding all composite operators is a stream graph where each vertex is a primitive operator instance. Primitive operators are written in a native language (C++ or Java), taking advantage of the performance and productivity of traditional languages for straight-line code. The runtime system calls the primitive operator to process input tuple, and the primitive operator calls the runtime system to submit output tuples. In operator invocations, primitive operators behave the same as composite operators, attaching streams to ports, passing parameters, defining windows, and so on (Section 4).

Although all primitive operators behave the same on the invocation side, they can follow different styles on the implementation side: with or without runtime reflection, and with or without compile-time specialization. These styles give the authors of primitive operators choices for trading off performance, generality, and usability. In particular, runtime reflection improves the generality of an operator by allowing it to work in different circumstances. Compile-time specialization also improves generality, and also leads to higher performance than runtime reflection, because the code is customized once statically instead of repeatedly checking a condition dynamically. Finally, an operator that uses neither runtime reflection nor compile-time customization is easiest to write, gaining usability at the cost of generality and possibly performance.

For SPADE users: SPADE had different kinds of primitive operators that were invoked and implemented differently. SPL redesigns them so all primitive operators are invoked in the same way, and implemented in a more similar fashion, although SPL still supports different implementation styles for primitive operators. SPADE primitive operator kinds correspond to SPL as follows:

- SPADE built-in operators (BIOPs): SPL has no built-in operators. Instead, SPL provides a default toolkit, with many of the operators that were built into SPADE. They are written without reflection and with compile-time specialization.
- SPADE user-defined operators (UDOPs): SPADE UDOPs correspond to SPL operators without compile-time specialization. They may or may not use runtime reflection.
- SPADE raw UDOPs: these operators in SPADE skipped tuple serialization and deserialization, operating on the raw network representation of tuples instead. SPL’s type system is designed to permit faster

serialization and deserialization, obviating the need for raw UDOPs.

- SPADE user-defined built-in operators (UBOPs): SPADE UDOPs correspond to SPL operators with compile-time specialization. They may or may not use runtime reflection.

5.6.1 Runtime APIs

Primitive operators provide native `process` functions for input tuples:

```
void process(Tuple& tuple, unsigned inputPortNumber) {
    //can inspect actual tuple type at runtime
    //can reflectively forward attributes from input to output
}
```

Process functions can be implemented in either reflective or non-reflective style. To work in non-reflective style, cast the tuple to the specific type:

```
void process(Tuple& tuple, unsigned inputPortNumber) {
    switch(port) {
        case 0: {
            IPort0_type& p0tuple = static_cast<IPort0_type&>(tuple);
            ...
            break;
        }
        ...
    }
}
```

The types `IPort0_type`, `IPort1_type`, etc. are defined to be exactly the types of the tuples that can arrive on the input streams. Primitive operators call native `submit` functions for output tuples:

```
submit(tuple, outputPortNumber);
```

5.6.2 Compile-Time APIs

The author of a primitive operator can optionally specialize the operator at compile time to optimize for statically known information. To do that, they write the operator in mixed-mode (MM), which is a mix of Perl and the native language (Java or C++). The Perl code executes at compile-time, and generates pure native-language code, which executes at runtime. Each primitive operator is described by an “operator model”, which specifies what constitutes a legal operator invocation. For example, the operator model specifies the number of ports and what kinds of streams they support, and specifies the names and metatypes for parameters. As another example, the operator model specifies whether the operator modifies tuple attributes, and if it does not, the compiler checks that parameter expressions do not call functions that can modify tuple attributes.

Figure 7 shows how primitive operators are compiled. When the SPL compiler encounters an operator invocation, it uses a lookup path to find a definition of either a composite operator or a primitive operator. Suppose it finds a primitive operator. The compiler first consults the operator model, which consists of XML, typically generated by wizards in the IDE. If the operator invocation violates the model, the compiler reports error messages to the user. Otherwise, it combines the operator model with the information from the operator invocation into an operator context, which is a Perl object supplied to the Perl code generator. The primitive operator is written as a mix of Perl snippets (the code generator) and native code (code to be generated), hence the name “mixed”-mode. The Perl code is embedded in the native code using ASP-style tags (`<% ... %>` or `<%= ... %>`), similar to how PHP or JSP code is embedded in HTML code using tags. At code generation time, the Perl code uses a Perl API to access the operator context. In the end,

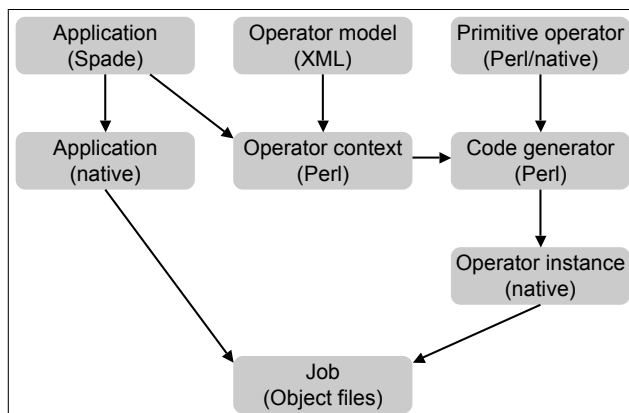


Figure 7: Build process for primitive operators.

both the SPL code and the primitive operators turn into native code, which gets compiled into object files to be deployed as an application instance.

Practical advice: When writing your own primitive operators with mixed-mode, keep in mind that parameters expressions may be expensive or cause side effects. Parameters with side effects are bad practice, but not prohibited. You should program your code generator defensively by maintaining a predictable execution order, and documenting that to the user so they know what to expect when invoking your operator.

Language design rationale: Mixing C++ and Perl code into each other sounds intimidating. However, we argue that this complexity is inherent rather than accidental. Library developers for SPL need a mechanism that supports flexibility, performance, and reuse of legacy code. To achieve flexibility and performance, we need a code generator framework, where compile-time customization provides flexibility without impacting runtime performance. And to achieve performance and reuse of legacy code, the generated code must be native, since a lot of code already exists in native languages, and native languages provide more traditional imperative programming features than SPL aspires to support. Note that users can choose to forgo the flexibility of mixed-mode by authoring primitive operators without Perl, thus avoiding the complexity of code generation.

For SPADE users: SPADE referred to the operator model as “restrictions”. SPL’s primitive operators with mixed-mode use the same mechanism as SPADE *UBOPs*, for “User-defined Built-in OPERators”, named so because they have the performance and ease-of-use of operators that are hardwired and built into other streaming languages. Compared to SPADE, restrictions have been improved. For example, SPL operator models can specify that two input streams must have the same schema; that a parameter must be a number between 0 and 9; or that an operator parameter requires an enumeration, such as the `Source format`, which can take the enumeration constant `csv` for comma-separated values.

SPADE also provided a feature for plugging additional native code from a separate file into an operator, for example, to parse a file in a source adaptor. SPL abandons that feature, because it is cleaner to place that code into an operator of its own. Here is an example for what this extension mechanism looked like in SPADE:

```

stream PerLine(linecount:Integer, wordcount:Integer, charcount:Integer) # 1
:= Source() # 2
  [ "file:///tmp/my_input_file", # 3
    udftxtformat = "WCParser", eolmarker = "\n" ] # 4
  { } # 5

```


6 Program Structure

Section 6.1 describes the SPL source code artifacts, whereas Section 6.2 describes how they map into an application instance that runs on the streaming middleware. Section 6.3 shows how multiple jobs on the same middleware communicate with each other, and Section 6.4 describes how the source code communicates with other static tools.

6.1 Compile-Time Entities

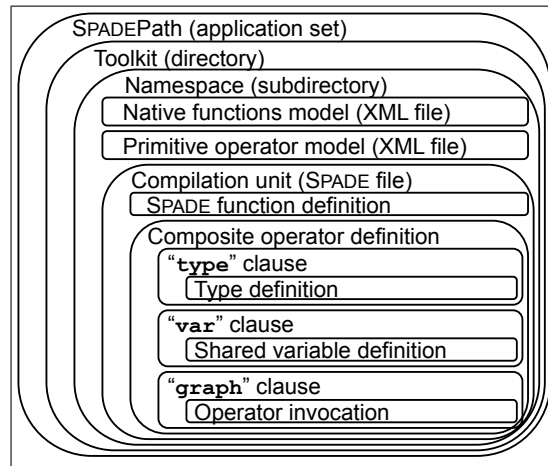


Figure 8: Compile-time entities.

The following example illustrates the various compile-time entities in an SPL file.

```

namespace com.ibm.samples.myNameSpace; // 1
use com.ibm.anotherNameSpace::*; // 2
int32 mySplFunction(int32 x) { return 2 * x; } // 3
stateful void myOtherFunction(mutable list<int32> a) { a = [99]; } // 4
composite SourceSink { // 5
  type MyType = int32 i, int32 j; // 6
  var int32 s_mySharedVar = 10; // 7
  graph stream<MyType> B = FileSource() { param file: "a.dat"; } // 8
    stream<B> C = SomeOperator(B) { param someParam: foo(i) + bar(j); } // 9
  config logLevel : trace; //10
} //11

```

Each SPL file belongs to a namespace (Line 1). The `use` directive makes entities from other namespaces available under a simple name (Line 2). For example, instead of `com.ibm.anotherNameSpace::SomeOperator`, the user can simply write `SomeOperator` (Line 9). An SPL file contains function definitions (Lines 3-4) and composite operator definitions (Lines 5-11). A composite operator can contain clauses with type definitions (Line 6) and shared variable definitions (Line 7). The `graph` clause of a composite operator contains operator invocations that define streams (Lines 8-9). Figure 8 shows how all these compile-time entities fit together. The start symbol for parsing an SPL file is the compilation unit:

```

compilationUnit ::= namespace useDirective* ( compositeDef | functionDef )+
namespace ::= 'namespace' ID+ ';'
useDirective ::= 'use' ID+ '::' ( '*' | ID ) ';'

```

A namespace contains all the operators and functions defined in files belonging to the namespace. Thus, a namespace in SPL is similar to a package in Java. A namespace can also contain primitive operators or native functions. They are not defined in SPL, but can be invoked from SPL. For example, Line 9 invokes an operator `SomeOperator`, which could either be a composite operator from another SPL file, or a primitive operator declared in an XML model. Similarly, Line 9 invokes two functions `foo` and `bar`, which could either be non-native functions from another SPL file, or native functions declared in an XML model file.

Functions and operators are available under their simple name if they are either part of the same namespace, or made available by a `use` directive. Examples for simple names include `FileSource` (Line 8), `SomeOperator` (Line 9), and `foo` (Line 9) above. The `use` directive comes in two forms (similar to Java): it can either refer to a single entity by name, or all entities in a namespace using the wildcard `*`. For example:

```
namespace test;                               // 1
use com.ibm.yourNameSpace::YourOperator; // 2
use com.ibm.yourNameSpace::yourFunction; // 3
use com.ibm.yourNameSpace::*;               // 4
composite F { }                             // 5
```

All files have an implicit wildcard `use` directive for the default namespaces `com.ibm.spl`, `com.ibm.spl.-adapter`, `com.ibm.spl.relational`, and `com.ibm.spl.utility`.

Practical advice: If a composite operator is used across files, it is good style, though not mandatory, to declare it in a file with the same name. For example, an operator `MyOp` should be defined in `MyOp.spl` if it gets used from other files.

Language design rationale: We picked `::` instead of `.` for namespace qualifiers to distinguish them from attribute access, e.g.: `com.ibm.myNameSpace::myOperator.myVariable.someAttribute`.

For SPADE users: Here is the same example program written in the old SPADE.

```
[Application]                                # 1
SourceSink trace                             # 2
[Program]                                     # 3
#include "com.ibm.anotherNameSpace/otherSpadeFile.din" # 4
use      com.ibm.anotherNameSpace.SomeOperator # 5
vstream MyType(i : Integer, j : Integer)      # 6
** no shared variables in Spade, omitted s_mySharedVar ** # 7
stream  B(schemaFor(MyType)) := Source() [ "file:///a.dat" ] {} # 8
stream  C(schemaFor(B)) := SomeOperator(B) [ someParam: foo(i) + bar(j) ] {} # 9
```

SPADE had no composite operators and no function definitions in the SPADE file. Those features are new in SPL.

There are a few program-level SPADE features that have been removed from SPL. SPADE had two preprocessors: one preprocessor with features such as for loops, and another mixed-mode (Perl embedded in SPADE) preprocessor. SPL drops the first preprocessor altogether, and will, at some unspecified time in the future, evolve to incorporate all the other needed scripting capabilities, at which point it drops the second preprocessor too. Note that this is unrelated to mixed-mode operators; there is no plan to drop those from SPL. Besides the preprocessors, SPL drops bundles. Bundles were in the SPADE language to make the preprocessor, and then mixed-mode, easier to use, but can be easily emulated by attaching multiple streams to the same port in SPL.

6.2 Compilation and Deployment

To turn the compile-time entities from Section 6.1 into runtime entities, they must be compiled and deployed as an application instance. To instantiate an application, the user specifies two things: an application scope and a main operator. The application scope is a runtime concept, consisting of a set of application instances, and serves for naming communication partners in dynamic application composition (see below). The main

operator is one of the potential main operators, chosen by the user. A potential main operator is any composite operator whose connection clause is empty and whose formal parameters all have default values. In other words, a potential main has no input or output ports and has no required parameters. The user can pick the main composite explicitly when compiling the application, or if the name is omitted, the compiler looks for an operator with the name `Main` as the default.

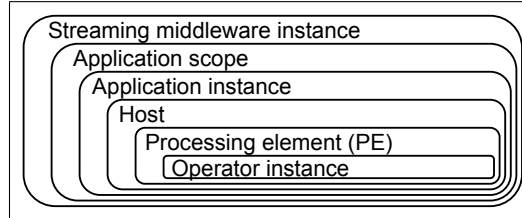


Figure 9: Runtime entities.

Figure 9 illustrates how all the runtime entities fit together. At the outermost level, there is an instance of the streaming middleware. The streaming middleware hosts one or more application scopes, and each application scope consists of one or more application instances. Each application instance has a main composite operator. To create each application instance, the compiler starts from main, and expands all operator invocations in the graph clause: it expands primitive operator invocations by running their code generator, and it expands composite operator invocations by expanding their graph clause. The end result is a directed graph where vertices are primitive operator instances and edges are streams. There can be multiple operator instances corresponding to a single operator invocation in the source code; this happens if the composite operator containing the operator invocation is expanded multiple times, as is the case in Figure 5. All expansion happens at compile time. The compiler may fuse multiple operator instances in the flow graph into units called partitions. The execution container for a partition is a processing element (PE). At runtime, there is a one-to-one correspondence between partitions and PEs, since each partition runs in exactly one PE.

The compilation typically involves source code in multiple files. It starts from an SPL file with the main operator, but must discover other source files when expanding operator or function invocations. This file lookup follows the SPLPATH shown in Figure 8, an ordered list of root directories or toolkit manifests from which to start lookup. A typical implementation of SPL supports SPLPATH specifications with compiler options. For example, if SPLPATH contains `/first/root/dir/`, the compiler looks for files implementing namespace `com.ibm.my.nameSpace` starting at the directory `/first/root/dir/com.ibm.my.nameSpace/`. This directory contains SPL files with functions and composite operators, as well as native source files with native functions and primitive operators. To be more accurate, the implementation of a primitive operator has its own directory, with at least one `generic` subdirectory for code that should work on all platforms, and possibly also platform-specific directories. As of now, SPL assumes homogeneous hardware on the hosts employed by the streaming middleware.

While configuration options can appear in many places in the code (see Sections 4.2.3 and 5.1.3), certain configurations typically only appear in a composite operator that is intended as the “main” composite operator in an application instance. These include the debug level (one of `error`, `info`, `debug`, or `trace`) and host pool declarations (which can be referenced from configuration options in stream or composite operator definitions, where they serve as deployment directives). The following example shows some of those configuration options:

```
namespace test; // 1
composite SourceSink { // main // 2
  graph // 3
    stream<int32 i> A = FileSource() { param file: "a.dat"; } // 4
    () as Sink = FileSink(A) { param file: "b.dat"; } // 5
```

```

config                                                    // 6
  logLevel      : trace;                                // 7
  hostPool      : SourcePool = ["localhost", "10.8.5.6"]; // 8
  defaultPool   : compileTime, size(5);                 // 9
}                                                         //10

```

For SPADE users: Here is the same compilation unit written in the old SPADE.

```

[Application]                                           # 1
SourceSink trace                                       # 2
[Libdefs]                                              # 3
incpath        "/inc1" "/inc2" # UDOP include path    # 4
libpath        "/lib1" "/lib2" # UDOP linking path    # 5
libs           "mylib" "yourlib" # UDOP linked libraries # 6
package        "pk1.h" "pk2.h" # user-defined functions # 7
[Nodepools]                                           # 8
nodepool      SourcePool[] := ("localhost", "10.8.5.6") # 9
nodepool      ComputingPool[15] := ()                 #10
defaultpool   roundrobin                              #11
[Program]                                             #12
stream A(i : Integer) := Source() ["file:///a.dat"] {} #13
Nil := Sink(A) ["file:///b.dat"] {}                  #14

```

SPADE divided the compilation unit into sections labeled with square brackets ([...]). SPADE [Libdefs] become SPL compiler options; [Nodepools] become SPL composite operator configuration options; and the SPADE [Program] becomes the topology of the SPL main operator.

6.3 Dynamic Application Composition

When a composite operator in the source code is intended as the main operator in a compiled application, its graph clause typically invokes operators for I/O. In particular, it uses various source and sink operators for I/O to URLs, files, databases, etc.; and it uses the operators `Import` and `Export` for I/O to other jobs on the same streaming middleware. Another way for multiple jobs to communicate with each other is via shared variables, see Section 5.5.

Streaming applications commonly run for long periods of time. Streams generated by one application often serve as inputs to another, and the second application may be deployed when the first is already running. The `Import` and `Export` operators support dynamic application composition, where one application exports a stream, another application imports the stream, but both applications are instantiated separately and may even be mutually anonymous. For example, the streaming middleware may host one or more long-running backbone applications that carry out the bulk of the data processing. Here is an example of exporting a stream to any applications that may be interested:

```

() as ExportOp = Export(E) {                             // 1
  param howPublished : x="baz", y=[5,6,7]; // 2
}                                                         // 3

```

The `howpublished` parameter specifies properties of the stream as name-value pairs. Users can then launch transient applications that import streams, for example, to visualize results in a dashboard. Here is an example of importing a stream from a different application instance:

```

stream<int32 z> I1 = Import() {                           // 1
  param subscription : x == "baz" && y[0] < 10; // 2
}                                                         // 3

```

The middleware connects the `Import` and `Export` streams if the subscription predicate matches the export properties. If the import predicate matches multiple streams exported by jobs running in the middleware, they are all connected. If there are no matching streams, nothing arrives on the `Import` operator. Note that since the compiler sees only one application at a time, it can not statically check whether the types in publication properties and subscription predicates match. The runtime system is not strict about these types: it will perform implicit conversions where needed before comparisons.

Besides this property-based anonymous connection, SPL also supports connecting streams across application instances by name. This means that the `Export` operator invocation does not need a `howpublished` parameter:

```

composite A(output O; input E) { // 1
  graph () as Invoke3 = Export(E) { } // 2
  stream<E> O = Functor(E) { } // 3
} // 4
composite B(output Invoke2; input F) { // 5
  graph stream<SomeTupleType> Invoke2 = A(F) { } // 6
} // 7
composite Main { // 8
  graph stream<SomeTupleType> G = SomeSourceOp() { } // 9
  stream<SomeTupleType> Invoke1 = B(G) { } //10
} //11

```

The importing application instance again uses the `Import` operator, but this time with a different set of parameters:

```

stream<SomeTupleType> I2 = Import() { // 1
  param applicationScope : "myApplicationScope"; // 2
  //application scope selected when exporting application launched // 3
  applicationName : "some.nameSpace::MainOp"; // 4
  //main operator selected when exporting application launched // 5
  subscription : streamId == "Invoke1.Invoke2.E"; // 6
  //outside-in name in case of nested composite invokes // 7
} // 8

```

Section 5.2 describes the outside-in naming scheme for streams. If the explicit `applicationScope` is omitted, it is implicitly bound to the scope in which the current application was launched. An explicit `applicationScope` can be used equally with predicate-based and name-based subscription. Note that there may be multiple instances of the same application in one application scope. If they export a stream, and an `Import` operator subscribes to that stream by name, the `Import` produces a merge of all the exported streams.

Practical advice: When streams are exported and imported across jobs, there may be a “dual maintenance” problem: the developer must keep both schemas in sync. Therefore, we recommend that you define the schema as a public static tuple type in a separate composite operator. This operator can then be used from both jobs to obtain stream types. There may also be unintended circles of imported and exported streams, so look out to avoid those.

Implementation note: In the current implementation, the SPL compiler represents `howPublished` parameters internally as XML and represents `subscription` parameters internally as XPath. As a result, only a restricted set of SPL expressions are valid for these two parameters, and the compiler will emit error messages if others are used. Also, while the language represents `Import` and `Export` as operators, the implementation erases them (treats them as ghost operators). This has the consequence that it is illegal to directly export an imported stream without sending it through another operator first. Dynamic optimization in the transport fabric ensures that the bandwidth for dynamic application composition connections is only used when a connection is actually in place. As a rule of thumb, it is more efficient to establish a by-name subscription

than a predicate-based anonymous subscription, though once the connections are established, they perform equally well.

For SPADE users: In SPADE, import/export were not stand-alone operators, but rather, were an optional part of every stream definition. For example, the exporting application instance could have a stream like this:

```
export properties [x: "baz"; y: 5, 6, 7] stream E(z : Integer) # 1
:= Functor(A) [ ] { } # 2
```

SPADE translated properties [x: "baz"; y: 5, 6, 7] into XML like this:

```
<stream>
  <x>baz</x>
  <y>
    <member>5</member>
    <member>6</member>
    <member>7</member>
  </y>
</stream>
```

SPADE expressed import predicates as XPath expressions. Bearing in mind that XPath indexing is 1-based, the SPADE code corresponding to our earlier SPL example looks like this:

```
import stream I2(z : Integer) tapping "stream[x='baz' and y/member[1]<10]"
```

6.4 Embedded Documentation

Implementation note: SPLDOC comments are a special form of delimited comments (see Section 1.4) that contain an extra asterisk (*) in the opening tag. In other words, SPLDOC comments start with /** and end with */. They are a form of embedded documentation, similar to doxygen (C), docstrings (Python), or javadoc (Java). Here is an example SPLDOC comment on a composite operator definition:

```
/** Short sentence describing what the composite operator does. // 1
    Longer description of the operator's functionality and purpose. // 2
  @input G description of input stream G // 3
  @input H description of input stream H // 4
  @param Q description of parameter Q // 5
  @output K description of output stream K // 6
  @output L description of output stream L // 7
  @tags G IBM HPQ // 8
  @tags H ?x // 9
  @tags Q TickerAggregationOperator UserDefinedParameter //10
  @tags K ?x GOOG //11
  @tags L IBM "some multi-word free-flow tag" ?x //12
  @tagfile "http://mariotagsserver.gov/FinanceTags.xml" //13
  @tagvar Company ?x //14
*/ //15
composite M (output K, L; input G, H) { //16
  param operator $Q; //17
  graph stream<int32 x> I = O(G) { } //18
    stream<int32 x> J = P(H) { } //19
    stream<int32 x> K = $Q(I; J) { } //20
    stream<int32 x> L = R(J) { } //21
} //22
```

The comment is semi-structured into optional clauses with labels such as “@input G” or “@tags K”. These clauses can be consumed by other SPL-related tools. For example, the IDE (integrated development environment) uses the @input, @param, and @output clauses to display documentation on a composite operator’s interface. The inquiry services tool uses the @inq clauses to automatically assemble programs from end-user specifications.

Here is an example SPLDOC comment on an operator invocation:

```

/** Split into streams of low and high values.           // 1
    The threshold value is 100: values under 100 are considered // 2
    low, values of 100 or above are considered high.      // 3
    @input I stream with all values, including both low and high // 4
    @output Lo stream with values below 100 only          // 5
    @output Hi stream with values above 100 only          // 6
*/                                                       // 7
(stream<int32 x> Lo; stream<int32 x> Hi) = LoHiSplitter(I) { // 8
    param loFilter : x < 100;                             // 9
        hiFilter : x >= 100;                              //10
    }                                                       //11

```

A VWAP Example

VWAP, or “volume-weighted average price”, is a common calculation in financial trading. Below is an example that uses VWAP to illustrate SPL features.

```

composite VWAP { // 1
param // 2
    expression<set<string8>> $monitoredTickers : { "IBM", "GOOG", "MSFT" }; // 3
// 4
type // 5
    TradeInfoT = decimal64 price, decimal64 volume; // 6
    QuoteInfoT = decimal64 bidprice, decimal64 askprice, decimal64 asksize; // 7
    TradeQuoteT = TradeInfoT, QuoteInfoT, // 8
        tuple<string8 ticker, string8 dayAndTime, string8 ttype>; // 9
    TradeFilterT= TradeInfoT, tuple<timestamp ts, string8 ticker>; //10
    QuoteFilterT= QuoteInfoT, tuple<timestamp ts, string8 ticker>; //11
    VwapT = string8 ticker, decimal64 minprice, decimal64 maxprice, //12
        decimal64 avgprice, decimal64 vwap; //13
//14
graph //15
    stream<TradeQuoteT> TradeQuote = FileSource() { //16
        param file : "TradesAndQuotes.csv.gz"; //17
            format : csv, compressed; //18
            columns : irange(1,4), [5], irange(7,10), [11,15,16]; //19
    } //20
//21
    stream<TradeFilterT> TradeFilter = Functor(TradeQuote) { //22
        param filter : ttype=="Trade" && (ticker in $monitoredTickers); //23
            output TradeFilter : ts = YYYYMMDDhhmmss_ToTimestamp(dayAndTime); //24
    } //25
//26
    stream<QuoteFilterT> QuoteFilter = Functor(TradeQuote) { //27
        param filter : ttype=="Quote" && (ticker in $monitoredTickers); //28

```

```

    output QuoteFilter : ts = YYYYMMDDhhmmss_ToTimestamp(dayAndTime); //29
} //30
//31
stream<VwapT, tuple<decimal64 sumvolume>> PreVwap = Aggregate(TradeFilter) { //32
    window TradeFilter : sliding, count(4), count(1), perGroup; //33
    param groupBy      : ticker; //34
    output PreVwap     : ticker = Any(ticker), vwap = Sum(price*volume), //35
                       minprice = Min(price), maxprice = Max(price), //36
                       avgprice = Avg(price), sumvolume = Sum(volume); //37
} //38
//39
stream<VwapT> Vwap = Functor(PreVwap) { //40
    output Vwap : vwap = vwap / sumvolume; //41
} //42
//43
stream<timestamp ts, decimal64 index> //44
    BargainIndex = Join(Vwap as V; QuoteFilter as Q) //45
{ //46
    window V          : sliding, count(1), perGroup; //47
    Q                 : sliding, count(0); //empty window (one-sided join) //48
    param equalityLHS : V.ticker; // can also be written as nested loop join: //49
    equalityRHS       : Q.ticker; // "condition : V.ticker == Q.ticker" //50
    output BargainIndex : //51
        index = vwap > askprice*100d ? asksizesize*exp(vwap-askprice*100d) : 0d; //52
} //53
//54
() as SinkOp = PerfSink(BargainIndex) { } //55
//56
config //57
    logLevel: trace; //58
} //59

```

B Grammar Overview

This appendix surveys the SPL syntax top-down, with references to the sections that discuss the semantics. Refer to Section 1.4 for the lexical syntax, and Section 1.3 for the grammar notation.

The compilation unit is the start symbol of the SPL grammar.

```

compilationUnit ::= namespace useDirective* # Section 6.1
                  ( compositeDef | functionDef )+
namespace      ::= 'namespace' ID+ ';'
useDirective   ::= 'use' ID+ '::' ( '*' | ID ) ';'

```


Composite operators are defined at the top-level in a namespace.

```

compositeDef ::= compositeHead compositeBody # Section 5.1
compositeHead ::= 'composite' ID ( '(' compositeInOut+; ')' )?
compositeInOut ::= ( 'input' | 'output' ) ( streamType? ID )+
streamType ::= 'stream' '<' tupleBody '>'
compositeBody ::= '{'
  ( 'param' compositeFormal+ )?
  ( 'type' typeDef+ )?
  ( 'var' sharedVarDef+ )?
  ( 'graph' opInvoke+ )?
  ( 'config' configuration+ )?
  '}'
compositeFormal ::= metaType ID ( ':' opActual )? ';' # Section 5.3.4
opInvokeActual ::= ID ':' opActual ';'
opActual ::= type | expr+
configuration ::= ID ':' expr+ ';' # Sections 4.2.3 and 5.1.3

```

Streams are defined in a composite operator's **graph** clause.

```

opInvoke ::= opInvokeHead opInvokeBody # Section 4
opInvokeHead ::= opOutputs ( 'as' ID )? '=' ID opInputs # Section 4.1
opOutputs ::= opOutput | '(' opOutput*; ')'
opOutput ::= streamType ID ( 'as' ID )?
opInputs ::= '(' portInputs*; ')'
portInputs ::= streamType? ID+ ( 'as' ID )?
opInvokeBody ::= '{' # Section 4.2
  ( 'logic' opInvokeLogic+ )?
  ( 'window' opInvokeWindow+ )?
  ( 'param' opInvokeActual+ )?
  ( 'output' opInvokeOutput+ )?
  ( 'config' configuration+ )?
  '}'
opInvokeLogic ::= ( 'onTuple' ID | 'onPunct' ID | 'state' ) ':' stmt # Section 4.3
opInvokeWindow ::= ID ':' expr+ ';' # Section 4.5
opInvokeOutput ::= ID ':' ( ID '=' expr )+ ';' # Section 4.2.1

```

Functions are defined at the top-level in a namespace.

```

functionDef ::= functionHead blockStmt # Section 3.4
functionHead ::= functionModifier* type ID '(' functionFormal*; ')'
functionModifier ::= 'public' | 'stateful'
functionFormal ::= 'mutable'? type ID

```

Shared variables are defined in a composite operator's **var** clause.

```

sharedVarDef ::= sharedVarModifier* type ID # Section 5.5
  ( '=' expr )? sharedVarConfigs
sharedVarModifier ::= 'public' | 'static' | 'mutable'
sharedVarConfigs ::= ';' | '{' 'config' configuration+ '}'

```

Imperative statements can appear in function bodies or the logic clause of an operator invocation.

```

stmt ::= localDecl | blockStmt | exprStmt # Section 3.3
      | ifStmt | forStmt | whileStmt
      | breakStmt | continueStmt | returnStmt
localDecl ::= 'mutable'? type ( ID ( '=' expr )? )+, ';'
blockStmt ::= '{' stmt* '}'
exprStmt ::= expr ';'
ifStmt ::= 'if' '(' expr ')' stmt ( 'else' stmt )?
forStmt ::= 'for' '(' type ID 'in' expr ')' stmt
whileStmt ::= 'while' '(' expr ')' stmt
breakStmt ::= 'break' ';'
continueStmt ::= 'continue' ';'
returnStmt ::= 'return' expr? ';'

```

Expressions can appear in many places in the grammar. For precedence and associativity, see Section 3.1.

```

expr ::= prefixExpr | infixExpr | postfixExpr # Section 3
      | conditionalExpr | '(' expr ')' | ID | literal
prefixExpr ::= prefixOp expr
prefixOp ::= '!' | '-' | '~' | '++' | '--'
infixExpr ::= expr ( infixOp | mappedOp | assignOp ) expr
infixOp ::= '+' | '-' | '*' | '/' | '%' | '<<' | '>>' | '&' | '^' | '|'
      | '&&' | '||' | 'in' | '<' | '<=' | '>' | '>=' | '!=' | '=='
mappedOp ::= '+.' | '-.' | '*.' | './' | '%.' | '<<.' | '>>.' | '&.'
      | '^.' | '|.' | '<.' | '<=.' | '>.' | '>=.' | '!=' | '=='
assignOp ::= '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '<<=' | '>>='
      | '&=' | '^=' | '|='
postfixExpr ::= ID '(' expr* ')'
      | type '(' expr ')'
      | expr '[' subscript ']'
      | expr '.' ID
      | expr postfixOp
subscript ::= expr | ( expr? ':' expr? )
postfixOp ::= '++' | '--'
conditionalExpr ::= expr '?' expr ':' expr

```

Literals are the highest-precedence expressions denoting values.

```

literal ::= primitiveLiteral | listLiteral | setLiteral # Section 3
      | mapLiteral | tupleLiteral
listLiteral ::= '[' expr* ']' # Section 2.2.1
setLiteral ::= '{' expr* '}'
mapLiteral ::= '{' ( expr ':' expr )* '}'
tupleLiteral ::= '{' ( ID '=' expr )* '}' # Section 2.2.2
primitiveLiteral ::= 'true' | 'false' | STRING | FLOAT | INT # Section 2.1;

```

Types are defined in a composite operator's type clause.

```

typeDef ::= 'public'? ID '=' ( type | tupleBody ) ';' # Section 5.1.2
metaType ::= 'attribute' | 'expression' typeArgs? # Section 5.3.4
            | 'function' typeArgs? | 'operator' | 'type'
type ::= ID | 'void' | primitiveType | compositeType # Section 2
typeArgs ::= '<' type+, '>'
typeDims ::= '[' expr '['

```

Primitive types are types without other types as arguments.

```

primitiveType ::= 'boolean' # Section 2.1
                | 'enum' '{' ID*, '}'
                | 'int8' | 'int16' | 'int32' | 'int64' | 'int128'
                | 'uint8' | 'uint16' | 'uint32' | 'uint64' | 'uint128'
                | 'float32' | 'float64' | 'float128'
                | 'decimal32' | 'decimal64' | 'decimal128'
                | 'complex32' | 'complex64' | 'complex128'
                | 'timestamp'
                | 'blob'
                | 'string8' typeDims?
                | 'string16'

```

Composite types are type constructors for composing new types out of other types.

```

compositeType ::= tupleType # Section 2.2
                  | 'list' typeArgs typeDims?
                  | 'map' typeArgs typeDims?
                  | 'set' typeArgs typeDims?
tupleType ::= 'tuple' '<' tupleBody '>' # Section 2.2.2
tupleBody ::= attributeDecl+
              | ( ID | tupleType )+
attributeDecl ::= type ID

```

Native function prototype declarations come from XML files, not regular SPL files.

```

functionPrototype ::= genericFormals functionModifier* # Section 3.4.2
                    type ID (' protoFormal*, ')
genericFormals ::= ( '<' typeFormal+, '>' )? ( '[' boundsFormal+, ']' )?
typeFormal ::= typeFormalConstraint ID
typeFormalConstraint ::= 'any' | 'collection' | 'complex' | 'composite'
                       | 'decimal' | 'enum' | 'float' | 'floatingpoint'
                       | 'integral' | 'list' | 'map' | 'numeric' | 'ordered'
                       | 'primitive' | 'set' | 'string' | 'tuple'
boundsFormal ::= ID
protoFormal ::= formalModifier* type ID?

```

C Acknowledgments

This language specification benefited from substantial, detailed, critical, constructive, and sometimes humorous feedback by many people. We incorporated some suggestions directly, many in modified forms, and for those we did not incorporate, we clarified our rationale. The following people commented on earlier drafts of

this document: Bob Blainey, Norman Cohen, Dan Debrunner, Fred Douglass, Jim Giles, Michel Hack, Paul Jones, Richard King, Tracy Kimbrel, Senthil Nathan, Rodric Rabbah, Anand Ranganathan, Anton Riabov, and Chitra Venkatramani.

References

- [1] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms (DM-SSP)*, 2006.
- [2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 2006.
- [3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, 2004.
- [4] Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and MyungCheol Doo. SPADE: The System S declarative stream processing engine. In *International Conference on Management of Data (SIGMOD)*, 2008.

Index

- alias, 19
- application, 42
- application instance, 41
- application scope, 41, 42
- assignment, 9, 12
- attribute, 8
- attribute-delta based window, 24

- BIOP, 35, 37
- BNF, 3
- bounded type, 6–8
- bundle, 41

- cast, 10, 12
- collection, 7
- comparison, 9
- compilation unit, 40
- complex, 5
- composite type, 7
- config, 22, 29
- consistency, 36
- constraint, 16
- count-based window, 24

- debug level, 42
- decimal, 5
- default namespace, 41
- dynamic application composition, 43

- EBNF, 3
- embedded documentation, 45
- enum, 6
- eviction policy, 24
- export, 43
- expression, 11
- extending tuples, 8

- float, 5
- function, 15

- generic, 16
- grammar, 3
- graph, 27, 42

- history, 25
- host pool, 42

- import, 43
- InfoSphere Streams, 2
- instance, 36

- int, 5

- leap second, 6
- list, 7
- literal, 6
- logic, 22, 34

- main operator, 41
- map, 7
- mixed-mode, 38, 41
- MM, 38
- mutable, 17, 36

- namespace, 40
- native, 15, 37
- non-terminal, 4

- operator
 - composite operator, 26
 - dotted operator, 13
 - expression operator table, 11
 - expression vs. stream operator, 11
 - mapped operator, 13
 - primitive operator, 37
- operator instance, 42
- operator instance name, 30
- operator invocation, 18
- operator model, 21, 38
- ordered type, 12, 16
- output clause, 21
- outside-in naming scheme, 30
- overloading, 16

- parameter
 - parameter to function, 15, 17, 37
 - parameter to operator, 22, 31
- partition, 26, 42
- pass-by-reference, 17
- PE, 42
- perGroup, 25
- Perl, 38
- port, 19, 27
- potential main, 42
- precedence, 12
- preprocessor, 41
- primitive type, 5
- private, 28, 36
- processing element, 42
- properties, 44
- public, 28, 36

punctuation, 23, 24
pure, 17

qualified name, 30

raw UDOP, 37

sale-join example, 2
set, 7
shared variable, 36
sink, 43
slice, 12
sliding window, 25
source, 43
SPADE, 3
SPLDOC, 45
SPLPATH, 40
SPLPATH, 42
start symbol, 40
state keeper, 37
stateful, 17
statement, 14
static, 36
stream, 18
stream name, 30
stream type, 19
string, 5
syntax, 3

TAI, 6
tapping, 45
time-based window, 24
timestamp, 5, 6
toolkit, 35
topology, 27
trigger policy, 25
tumbling window, 24
tuple, 8, 19
type definition, 28
type hierarchy, 5
type suffix, 6

UBOP, 38, 39
UDOP, 37
uint, 5
unicode, 5
use, 40
ustring, 5
UTC, 6

value semantics, 9

window, 23
window size, 24