SYBASE®
An SAP® Company

SPLASH チュートリアル

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01737-01-0500-01
改訂： 2011 年 12 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章 概要

Sybase® Event Stream Processor により使用される言語である CCL には、2 つのコンポーネントがあります。

最初のコンポーネントは、SQL の拡張です。これは、ストリームとその他の高いレベルのコンポーネントの宣言に使用され、データ・フローを記述し、関係演算 (ジョイン、集約、フィルタなど) として表現できるロジックを定義します。 2 つ目のコンポーネントは、SPLASH です。これは、関係演算子を使用して簡単に表現することができないロジックの指定に使用される手続き型言語です。SPLASH は、他の手続き型言語のように、変数、ループ構造体、データ構造、関数定義をサポートします。

第 2 章 基本事項

関数と高度なデータ構造を取り扱う前に、言語の基本事項を理解します。

SPLASH は C/C++ や Java に似ていますが、独自の注意点がいくつか存在します。

定数と単純式

文字列定数と数値式は両方とも SPLASH の基本的な構成要素です。

以下の例のように、従来の "hello world" プログラムの SPLASH バージョンを使用します。

```
print('hello world¥n');
```

この文は、行 `hello world` を Sybase ESP スタジオのコンソール上または Sybase ESP サーバが開始されたターミナル上に表示します。 このサンプル・コードには、一重引用符で囲まれた文字列定数が含まれています。 これが、文字列定数に二重引用符を使用している C/C++ や Java と異なる点です。

数値式は、この言語のもう 1 つの基本的な構成要素です。数値定数は、整数形式 (たとえば、1826)、浮動小数点形式 (たとえば、72.1726)、または固定小数点形式となります。 標準的な算術演算子 (+、-、*、/、^) とカッコから式を形成できます。これは、通常定義されている優先順序で計算されます。 次の式の計算結果は 57 になります。

```
1 + 7 * 8
```

しかし、次の式の計算結果は 64 になります。

```
(1 + 7) * 8
```

SPLASH には、計算関数のホストも含まれます。 次の関数は、値の正弦を返します。

```
sine(1.7855)
```

C/C++ や Java のように、true または false を返すブール式が数値式によって表されます。 整数 0 は false を表し、0 以外の整数は、true を表します。 = (等号)、!= (不等号)、< (より小さい)、> (より大きい) などの比較演算子は、false の場合は 0 を、true の場合は 1 を返します。 演算子 `and`、`or`、`not` を使用してブール式を結合できます。 たとえば、`not(0 >1)` は 1 を返します。

NULL 値

各データ型は、リレーショナル・データベースの null 値と一致するものに対し `null` が記述された、識別済みの空の値を含みます。 null 値は欠損値をコード化します。 それ自体の値も含め、いかなる値とも比較できません。 したがって、式 `(null = null)` と `(null != null)` は、両方とも 0 (false) です。

null が指定されると、多くの組み込み関数は null を返します。たとえば、`sqrt(null)` は null を返します。

値と null は比較できないため、null 値を処理するために特別な SPLASH 関数が存在します。 関数 `isnull` は、引数が null の場合は 1 (true) を返し、それ以外の場合は 0 (false) を返します。 通常、値のシーケンス内には null 以外の値が指定されるため、戻り値が値のシーケンスの最初の null 以外の値である関数 `firstnonnull` が存在します。 次の例は、3 を返します。

```
firstnonnull(null,3,4,5)
```

変数と割り当て

他のプログラミング言語と同様に、SPLASH 内の変数を使用するには、変数の宣言とその変数への値の割り当てが必要です。

たとえば、以下のコードは、整数型の値 (32 ビット整数) を保持する変数 `eventCount` を宣言します。

```
integer eventCount;
```

変数を宣言したら、`:=` 演算子を使用して値を割り当てます。

```
eventCount := 4;
```

その名前を記述して変数の値を使用します。

```
eventCount + 1
```

より簡潔に、宣言と初期値への割り当てを同時に行うことができます。

```
float pi := 3.14159265358979;
money dollarsPerEuro := 1.58d;
```

1 つの宣言で同じ型の変数を複数宣言することもできます。これには、初期値を指定する変数と初期値を指定しない変数を混在させることもできます。 次の宣言では、`pi` と `e` が初期値に設定された、すべてが float データ型の 3 つの変数を記述します。

```
float pi := 3.14159265358979, lambda, e := 2.714;
```

SPLASH のアルファベット以外の文字またはキーワードで始まる変数を二重引用符で囲まれた変数名に変換できます。

```
long "500miles" := 500 * 1760;
string "for" := 'for ever';
```

この機能は、SQL に直接由来します。

明示的に変数を宣言すると、SPLASH は、C や Java のような静的型付き言語となります。 これは、変数の宣言を強制しない Perl や AWK のようなスクリプト言語とは異なります。 より多くの文字を使用すると、コードが管理しやすくなり、コードがより速く実行されるように最適化を行うことができます。

データ型

SPLASH は標準の CCL データ型として、integer、string、float、long、money、money(n)、date、timestamp、bigdatetime、interval、binary、boolean を使用します。

データ型	説明
`integer`	符号付き 32 ビット整数値。指定可能な値の範囲は、-2147483648 ～ +2147483647 (-2^{31} ～ 2^{31-1}) です。この範囲外にある定数値は、長いデータ型として自動的に処理されます。 -2147483648 の値の変数、パラメータ、カラムを初期化する場合は、(-2147483647) -1 を指定してコンパイル・エラーを回避してください。
`long`	符号付き 64 ビット整数値。指定可能な値の範囲は、-9223372036854775808 ～ +9223372036854775807 (-2^{63} ～ 2^{63-1}) です。 -9223372036854775807 の値の変数、パラメータ、カラムを初期化する場合は、(-9223372036854775807) -1 を指定してコンパイル・エラーを回避してください。
`float`	倍精度の 64 ビット浮動小数点数。指定可能な値の範囲は、およそ -10^{308} ～ +10^{308} です。
`string`	UTF-8 でコード化されたバイト値を持つ可変長文字列。最大の文字列の長さはプラットフォームによって異なりますが、65535 バイトを超えることはありません。
`money`	精度をグローバルに適用する符号付き 64 ビット整数値。通貨記号とカンマは入力データ・ストリームではサポートされません。

データ型	説明
`money(n)`	小数点以下 1 ～ 15 桁の可変精度をサポートする符号付き 64 ビット整数値。通貨記号とカンマは入力データ・ストリームではサポートされませんが、小数点はサポートされます。 指定された精度によって、サポートされる値の範囲が変化します。 `money(1)`: -922337203685477580.8 ～ 92233720368547758O.7 `money(2)`: -92233720368547758.08 ～ 92233720368547758.07 `money(3)`: -9223372036854775.808 ～ 9223372036854775.807 `money(4)`: -922337203685477.5808 ～ 922337203685477.5807 `money(5)`: -92233720368547.75808 ～ 92233720368547.75807 `money(6)`: -92233720368547.75808 ～ 92233720368547.75807 `money(7)`: -922337203685.4775808 ～ 922337203685.4775807 `money(8)`: -92233720368.54775808 ～ 92233720368.54775807 `money(9)`: -9223372036.854775808 ～ 9223372036.854775807 `money(10)`: -922337203.6854775808 ～ 922337203.6854775807 `money(11)`: -92233720.36854775808 ～ 92233720.36854775807 `money(12)`: -9223372.036854775808 ～ 9223372.036854775807 `money(13)`: -922337.2036854775808 ～ 922337.2036854775807 `money(14)`: -92233.72036854775808 ～ 92233.72036854775807 `money(15)`: -9223.372036854775808 ～ 9223.372036854775807 -92,233.72036854775807 の値の変数、パラメータ、カラムを初期化する場合は、(-9...7) -1 を指定してコンパイル・エラーを回避してください。 money 定数の精度を明示的に指定するには、**Dn** 構文を使用します (**n** は精度を表します)。たとえば、100.1234567D7、100.12345D5 というように指定します。 `money(n)` 型間の暗黙的変換は、サポートされません。これは、範囲または精度を失うリスクがあるためです。**cast** 関数を実行して、異なる精度を持つ money 型を処理します。

データ型	説明
`bigdatetime`	マイクロ秒の精度のタイムスタンプ。デフォルト・フォーマットは YYYY-MM-DDTHH:MM:SS:SSSSSS です。 すべての数値データ型は暗黙的に `bigdatetime` にキャストされます。 変換規則は、以下のようにデータ型ごとに異なります。 • `boolean`、`integer`、`long` の値はすべて、元のフォーマットで `bigdatetime` に変換されます。 • `money(n)` と `float` の整数部分の値のみ `bigdatetime` に変換されます。**cast** 関数を使用して `money(n)` と `float` の値を、精度を指定した `bigdatetime` に変換します。 • `date` の値はすべて 1000000 で乗算されてマイクロ秒単位に変換され、`bigdatetime` フォーマットを満たします。 • `timestamp` の値はすべて 1000 で乗算されてマイクロ秒単位に変換され、`bigdatetime` フォーマットを満たします。
`timestamp`	ミリ秒の精度のタイムスタンプ。デフォルト・フォーマットは YYYY-MM-DDTHH:MM:SSS です。
`date`	ミリ秒の精度のタイムスタンプ。デフォルト・フォーマットは YYYY-MM-DDTHH:MM:SSS です。

データ型	説明
`interval`	2 つのタイムスタンプ間のマイクロ秒の数値を示す、符号付き 64 ビット整数値。スペース区切りのフォーマットで複数の単位を使用して、`interval` を指定します。たとえば、"5 Days 3 hours 15 Minutes" と指定します。interval カラムに送信される外部データは、マイクロ秒単位と見なされます。`string` データに変換された `interval` 値、または `string` データから変換された `interval` 値では、単位指定はサポートされていません。 `interval` を指定する場合は、指定された間隔が適切なマイクロ秒単位の数値に変換される際に 64 ビットの整数値 (`long`) に収まるようにしてください。`interval` の各単位の、マイクロ秒に変換される際の long に収まる最大許容値は以下のとおりです。 • MICROSECONDS (MICROSECOND, MICROS): +/- 9223372036854775807 • MILLISECONDS (MILLISECOND, MILLIS): +/- 9223372036854775 • SECONDS(SECOND, SEC): +/- 9223372036854 • MINUTES(MINUTE, MIN): +/- 153722867280 • HOURS(HOUR,HR):+/- 2562047788 DAYS(DAY): +/- 106751991 カッコ内の値は、`interval` の単位の代替名です。 ある単位の最大値を指定すると、他の単位の指定ができなくなるか、オーバフローが発生します。各単位は、一度だけ指定できます。
`binary`	ロー・バイナリ・バッファを表します。値の最大長はプラットフォームによって異なりますが、65535 バイトを超えることはありません。NULL 文字を使用できます。
`boolean`	値は true または false です。`boolean` の許容範囲外の値のフォーマットは、0/1/false/true/y/n/on/off/yes/no で、大文字と小文字は区別されません。

SPLASH のプログラムは、数値型の値を他の型に自動的に変換します (可能な場合)。

```
float e := 2.718281828459;
integer years := 10;
```

上記の宣言を行うと、次の式がデータ型 float の有効な式となります。

```
1000.0d * (e ^ (0.05 * years))
```

整数型の変数 `years` は、money 型の定数値 1000.0d として float に自動的に変換されます。

必要に応じて、cast 演算を使用して値をダウンキャストします (大きい精度の数値をより小さい精度の数値に変換します)。 たとえば、次の式は、数値の小数部分をトランケートして、float 値を整数に変換します。

```
cast(integer, 1000.0d * (e ^ (0.05 * years)))
```

データ型 date、timestamp、bigdatetime の値を数値のように使用して計算を行うことができます。 たとえば、date 値に 10 を加えると、将来的に date 値は 10 秒になります。 同様に、timestamp 値に 10 を加えると、将来的に timestamp 値は 10 ミリ秒になります。 精度はその型で暗黙的に指定されます。

データ型の省略形

データ型の省略形を使用して、データ型に代替名を指定します。 長いデータ型の名前を使用する場合、省略形が特に役立ちます。

データ型に代替名を指定するには、`typedef` 宣言を使用します。 以下の例では、money データ型の別の名前として `euros` を宣言します。

```
typedef money euros;
```

次の例では、このデータ型の変数 `price` を宣言します。

```
euros price := 10.70d;
```

また、`typeof` 演算子を使用して、データ型の定義を簡略化できます。 この演算子は式のデータ型を返します。

```
typeof(price) newPrice := 10.70d;
```

上記の例は、以下のように記述する場合と同じです。

```
money newPrice := 10.70d;
```

ブロック

コードを複数のブロックに分割して変数を宣言し、その値をブロックに対してローカルに設定します。

文のブロックは、中カッコ内に記述されます。

```
{
  float pi := 3.1415926;
  float circumference := pi * radius;
}
```

この例では、ブロックに対してローカルな変数 `pi` を宣言し、その変数を使用して変数 `circumference` を設定します。 変数の宣言 (ただし、型の省略形は含まな

い) は複数の文に分散して挿入されることがあります。この宣言をブロックの先頭に置く必要はありません。

ブロック内にブロックをネストさせる場合、通常のスコープ・ルールが適用されます。次の SPLASH コードは、“here” の代わりに “there” を出力します。

```
{
  string t := 'here';
  {
    string t := 'there';
    print(t);
  }
}
```

制御構造体

SPLASH の制御構造体は、C や Java の制御構造体と似ています。

条件付き実行には、`if` 文と `switch` 文を使用します。 たとえば、次のブロックでは温度の符号に応じて異なる値に文字列変数を設定します。

```
if (temperature < 0) {
  display := 'below zero';
} else if (temperature = 0) {
  display := 'zero';
} else {
  display := 'above zero';
}
```

`switch` 文は、多数の選択肢の中から選択を行います。

```
switch (countryCode) {
  case 1:
    continent := 'North America';
    break;
  case 33:
  case 44:
  case 49:
    continent := 'Europe';
    break;
  default:
    continent := 'Unknown';
}
```

`switch` の後の式は、integer、long、money、money(n)、float、date、timestamp、または bigdatetime とすることができます。

`while` 文は、ループをコード化します。 たとえば、次の式は、0 ～ 9 の 2 乗和を計算します。

```
integer i := 0, squares := 0;
while (i < 10) {
  squares := squares + (i * i);
```

```
  i++;
}
```

この例では、演算子 ++ を使用して変数 i に 1 を加えます。 break 文はループを終了します。また、continue は先頭でループを再度開始します。

最後に、exit 文を使用して、SPLASH コードのブロックの実行を停止できます。この文は Event Stream Processor を停止しません。ブロックのみ停止します。

第 3 章 レコード・イベント

Event Stream Processor では、ストリームにより“「レコード・イベント」”が処理されます。 イベントとは、フィールド名と値を関連付ける複合値であるレコードであり、挿入や更新などの操作でもあります。 フィールドの一部は、キー・フィールドとして指定されている場合があります。

レコード・タイプ

レコード・タイプは、レコード内の各フィールドの名前とデータ型を含むレコードの構造を定義します。

次に、レコード・タイプの例を示します。

```
[ string Symbol; | integer Shares; float Price; ]
```

このタイプは、3 つのフィールドを持つレコードを記述します。string フィールド Symbol は、| 記号の左側にあるため、唯一のキー・フィールドです。integer フィールドは、共有の数を示します。float フィールドは、価格を示します。

各フィールドで、基本的なデータ型 (integer、long、float、money、money(n)、string、date、bigdatetime、timestamp、interval、binary、または boolean) を 1 つ持つようにします。 レコードはネストできません。

レコード・タイプは長いため、`typedef` を使用して短い名前を指定すると便利なことがよくあります。

```
typedef [ string Symbol; | integer Shares; float Price; ] rec_t;
```

この例の `typedef` では、(最初の例と同様に) 3 つのフィールドを持つレコードを作成し、`rec_t` の名前を指定します。

レコード値

名前が指定されたレコードがある場合、静的または可変の値を割り当てることができます。

タイプ `rec_t` の 2 つのレコード値は、以下のようになります。

```
[ Symbol='T'; | Shares=10; Price=20.15; ]
[ Symbol='GM'; | Shares=5; Price=16.81; ]
```

レコード変数を割り当てることができます。 次の例では、レコード変数を宣言し、それにレコード値を割り当てます。

```
rec_t rec := [ Symbol='T'; | Shares=10; Price=22.88; ];
```

レコードのフィールドの値を取得するには、“.” 演算子を使用します。 たとえば、式 `rec.Symbol` は文字列 “T” を返します。

レコード値には null を指定できます。 null のレコードのフィールドにアクセスしようとすると、null が返されます。

新しいレコードを再度作成する必要なく、レコードのフィールドの値を変更できます。次の例では、Shares フィールドの値を 80 に変更します。

```
rec.Shares := 80;
```

キー・フィールド

ストリームは、ユニーク・キーごとに最大 1 つのレコードを格納します。つまり、キー・フィールドの値は、ストリーム内で一意である必要があります。

以下のレコードはそれぞれユニーク・キーを持ちます。

```
[ Market = 'NYSE'; Symbol='T'; | Shares=10; Price=22.88; ]
[ Market = 'NYSE'; Symbol='GM'; | Shares=5; Price=16.81; ]
```

次の 3 つ目のレコードは最初のレコードと一致します。

```
[ Market = 'NYSE'; Symbol='T'; | Shares=10; Price=20.15; ]
```

これらの両方をストリーム内に格納することはできません。ただし、更新を使用して最初のレコードをこのレコードで上書きすることはできます。

レコード・キャスティング

レコードは、コンテキストに応じて暗黙的に強制されます。 Extra フィールドは削除され、欠落したフィールドは null となります。

たとえば、次の割り当てによって、割り当てが行われる前に Extra フィールドが削除されます。

```
rec_t rec := [ Symbol='T'; | Shares=10; Price=22.88; Extra=1; ];
```

これに対し、次の例は、変数 `rec` を Price フィールドが null のレコードに割り当てます。

```
rec_t rec := [ Symbol='T'; | Shares=10; ];
```

また、SPLASH はキー・フィールドに関しては柔軟に対応します。たとえば、以下の例のように Symbol フィールドをキー・フィールドにし忘れた場合でも、Symbol フィールドはキー・フィールドになります。

```
rec_t rec := [ | Symbol='T'; Shares=10; Price=22.88; Extra=1; ];
```

キー・フィールドは null にできません。以下の割り当ては有効ですが、これをダウンストリームのストリームに送信することはできません。

```
rec_t rec := [ | Shares=10; Price=22.88; Extra=1; ];
```

これは、以降のトピックで詳しく説明します。

隠しフィールド

レコードには 3 つの暗黙的フィールド `ROWID`、`ROWTIME`、`BIGROWTIME` があります。これらは、それぞれ long、date、bigdatetime のデータ型です。 ストリームはこれらの値を自動的に入力します。また、通常の “.” 操作でこれらにアクセスできます。

操作

挿入、更新、削除、アップサート、安全削除のいずれかの操作は各イベント内で暗黙的です。

各操作には同等の数値コードがあり、これらの数値に対する `insert`、`update`、`delete`、`upsert`、`safedelete` の特殊定数があります。

- “insert” はレコードの挿入を意味します。 これらのキーを持つレコードがすでに存在している場合は、ランタイム・エラーとなります。
- “update” はレコードの更新を意味します。 これらのキーを持つレコードが存在しない場合は、ランタイム・エラーとなります。
- “delete” はレコードの削除を意味します。 これらのキーを持つレコードが存在しない場合は、ランタイム・エラーとなります。
- “upsert” は、これらのキーを持つレコードが存在しない場合は、レコードの挿入を意味し、それ以外の場合は、レコードの更新を意味します。 これは、“insert” または “update” による潜在的なランタイム・エラーを回避します。
- “safedelete” は、これらのキーを持つレコードがすでに存在している場合は、レコードの削除を意味し、それ以外の場合は、レコードの無視を意味します。これは、“delete” による潜在的なランタイム・エラーを回避します。

レコード・イベントが作成されるときは、操作は insert に設定されます。

関数 `getOpcode` を使用してイベントから操作を取得し、関数 `setOpcode` を使用して操作を設定します。 関数 `setOpcode` は、コピーを作成せずにレコード・イ

ベントを変更します。 たとえば、以下では、insert と safedelete の数値コード (それぞれ 1 と 13) を出力します。

```
[ integer k; | string data;] v;
v := [k=9;|];
print('opcode=', string(getOpcode(v)), '¥n');
setOpcode(v,safedelete);
print('opcode=', string(getOpcode(v)), '¥n');
```

レコード・イベント内の操作は、ストリームおよびイベント・キャッシュ内で使用されます。 これは、以降のトピックで詳しく説明します。

第 4 章 SPLASH 関数

CCL には多数の組み込み関数があり、そのすべてを SPLASH 式で使用できます。 SPLASH で独自の関数を記述することもできます。 これは、すべてのストリームで使用するためにグローバル・ブロックで宣言することも、1 つのストリームで使用するためにローカル・ブロックで宣言することもできます。 関数は、内部的に他の関数を呼び出すことも、自身を再帰的に呼び出すこともできます。

注意： C/C++ または Java で独自の関数を記述することもできます。 ライブラリの構築と Event Stream Processor 内部からのライブラリの呼び出しについて詳しくは、参照情報を確認してください。

SPLASH 関数を宣言する構文は C に似ています。通常、関数は次のようになります。

```
type functionName(type1 arg1, ..., typen argn) { ... }
```

ここで各 “type” は SPLASH のタイプで、各 `arg` は引数の名前です。 関数の本文は文のブロックで、変数の宣言から開始できます。 この関数によって返される値は、ブロック内の `return` 文によって返される値と同じです。

次に、再帰関数の例を示します。

```
integer factorial(integer x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
```

次に、2 つの相互再帰関数の例を示します (偶数または奇数を計算する場合は、特に非効率的な方法です)。

```
string odd(integer x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }
string even(integer x) {
   if (x = 0) {
     return 'even';
   } else {
    return odd(x-1);
   }
}
```

C とは異なり、“odd” 関数を宣言するのに “even” 関数のプロトタイプは必要ありません。

次の 2 つの関数は、複数の引数と入力レコードを示します。

```
integer sumFun(integer x, integer y) {
  return x+y;
}
string getField([ integer k; | string data;] rec) {
  return rec.data;
}
```

SPLASH を実際に使用すると一度だけ計算を定義できます。 現在の価格、満期までの日数、将来の値上げの予測に基づいて、債券の値を計算する方法があるとします。 以下のように、関数を記述してプロジェクト内の多くの場所で使用できます。

```
float bondValue(float currentPrice,
                integer daysToMature,
                float inflation)
{
  ...
}
```

第 5 章 高度なデータ構造

SPLASH を使用すると、後で使用できるようにデータ構造内にデータを格納できます。 3 つの主なタイプとして、ベクトル、辞書、イベント・キャッシュがあります。

辞書とベクトルのデータ構造はグローバルに定義できますが、グローバルに使用できるのは読み取りのみに限られています。辞書またはベクトルのデータ構造に書き込みを行うことができるのは 1 つのストリームのみです。また、ストリームが書き込みを行っている間は、他のストリームはそのデータ構造に対して書き込みを行うことも読み取りを行うこともできません。グローバルな辞書またはベクトルのデータ構造の管理に使用する基本となるオブジェクトはスレッドセーフではありません。ストリームには、書き込みを行っている際のグローバルな辞書またはベクトルのデータ構造に対する排他アクセスが必要です。あるストリームが書き込みを行っている際に他のストリームがこれらのデータ構造にアクセスできるようにすると、サーバ障害が発生することがあります。

これらのデータ構造の使用は、処理中に更新を必要としないが、複数のストリームによって読み取られる比較的静的なデータ (国コードなど) のみに限られています。辞書またはベクトルへのデータの書き込みは、ストリームが読み取りを行う前に完了させてください。

以下の例に示すように、グローバルな辞書またはベクトルを読み取るすべての操作により、isnull チェックが実行されます。

```
typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

ベクトル

ベクトルはすべて同じタイプの値のシーケンスです。 ベクトルのサイズがランタイムに変更できることを除いて、C の配列と同様です。

次のブロックでは、架空のコンポーネントを使用せずに「べき根」を格納する新しいベクトルを作成します。

```
vector(float) roots;
integer i := 0;
float pi := 3.1415926, e := 2.7182818, sum1 := 0;
resize(roots, 8);  // new size is 8, with each element set to null
while (i < 8) {
  roots[i] := e ^ ((pi * i) / 4);
```

```
  i++;
}
```

これにより、空のベクトルが作成され、resize によってサイズ変更が行われ、ベクトルの要素に値が割り当てられます。 ベクトルの最初の要素のインデックスは 0、2 番目の要素のインデックスは 1、以下同様に続きます。 push_back 演算子を使用してベクトルの最後に新しい要素を追加することもできます。たとえば、push_back(roots, e^pi) のように指定します。

次に、roots ベクトルの値の合計の計算方法を示します。

```
i := 0;
while (i < size(roots)) {
  sum1 := sum1 + roots[i];
  i++;
}
```

size 操作はベクトルのサイズを返します。 for ループを使用してベクトルの要素をループ処理することもできます。

```
for (root in roots) {
  sum1 := sum1 + root;
}
```

変数 root は、スコープがループ本文に制限されている新しい変数です。 最初にループに入る際には roots[0]、2 番目にループに入る際には roots[1]、以下同様に続きます。 roots[n] が null であるか、roots にこれ以上要素が存在しない場合は、ループが停止します。

ベクトルの他の 2 つの操作は便利です。new 操作を使用して新しいベクトルを作成できます。

```
roots := new vector(float);
```

古いベクトルは自動的に破棄されます。

辞書

辞書はキーと値を関連付けます。キーと値には、多少アクセスが遅くなる代わりに辞書をベクトルより柔軟にするタイプを指定できます。

次に、通貨変換レートの辞書を作成して初期化する例を示します。

```
dictionary(string, money) convertFromUSD;
convertFromUSD['EUR'] := 1.272d;
convertFromUSD['GBP'] := 1.478d;
convertFromUSD['CAD'] := 0.822d;
```

明確なキーごとに 1 つのみ値を指定できるため、次の文はキー “EUR” に関連付けられた以前の値を上書きします。

```
convertFromUSD['EUR'] := 1.275d;
```

式 convertFromUSD['CAD'] は、辞書から値を抽出します。 一致するキーがない場合、convertFromUSD['JPY'] として、式は null を返します。

関数 remove を使用して、辞書からキーとその値を削除できます。 たとえば、remove(convertFromUSD,'EUR') は、ユーロのキーと対応する値を削除します。 関数 clear は、辞書からすべてのキーを削除します。 empty 操作を使用して辞書にこれ以上キーがないかどうかをテストできます。

辞書のすべての要素をループ処理するために、for ループを使用できます。

```
for (currency in convertFromUSD) {
  if (convertFromUSD[currency] > 1) {
    print('currency ', currency, ' is worth more than one USD.¥n');
  }
}
```

変数 currency は、スコープがループ本文に制限されている変数で、辞書のキーのタイプが指定されます (この場合は文字列が指定されています)。

最後に、new 操作を使用して新しい辞書を作成できます。 たとえば、以下では、空の辞書を作成し、convertFromUSD を割り当てます。

```
convertFromUSD := new dictionary(string, money);
```

ベクトルと辞書の混成および参照セマンティック

ベクトルと辞書を使用する場合、単純な構造を構築することも、複雑な構造を構築することもできます。 たとえば、複数のベクトルからなるベクトルや、複数の辞書から成るベクトルなどの混成構造を構築できます。

たとえば、証券コードごとに、一連の以前の株価を格納します。 次の宣言と関数により、証券コードごとにキーが付けられた、そのような一連の格納された価格が作成されます。

```
dictionary(string, vector(money)) previousPrices;
integer addPrice(string symbol, money price)
{
  vector(money) prices := previousPrices[symbol];
  if (isnull(prices)) {
    prices := new vector(money);
    previousPrices[symbol] := prices;
  }
  push_back(prices, price);
}
```

この例では、コンテナの参照セマンティックを使用します。たとえば、次の割り当てによって、ベクトルのコピーではなくベクトルの参照が返されます。

```
vector(money) prices := previousPrices[symbol];
```

これは参照であるため、push_backによって挿入された値は、次回辞書から読み込まれる際にベクトルに組み込まれます。

参照セマンティックにより、エイリアス、つまり、同じエンティティの代替名を使用できます。 たとえば、以下により、プログラム出力に「エイリアスが定義」されます。

```
dictionary(integer, integer) d0 := new dictionary(integer, integer);
dictionary(integer, integer) d1 := d0;
d1[0] := 1;
if (d0[0] = 1) print('aliased!');
```

イベント・キャッシュ

イベント・キャッシュは、入力ストリームからのイベントをグループ分けして格納するための特別な SPLASH データ構造です。 イベントは、複数のバケットにグループ化されます。 バケットに対し、count、sum、max のような集合演算を実行できます。

入力ストリームの名前を使用して、ストリームのローカル・ブロックでイベント・キャッシュを宣言します。 たとえば、以下では、入力ストリーム Trades のイベント・キャッシュを宣言します。

```
eventCache(Trades) events;
```

希望する数のイベント・キャッシュを入力ストリームに指定できるため、同じストリームで以下を宣言できます。

```
eventCache(Trades) moreEvents;
```

デフォルトでは、入力ストリームのキーによりバケットが決定します。 たとえば、2 つのバケット (記号 “T” を持つイベントのバケットと記号 “CSCO” を持つイベントのバケット) を持つ入力ストリーム Trades がある場合は、以下のように指定します。

```
[ Symbol='T'; | Shares=10; Price=22.88; ]
[ Symbol='CSCO'; | Shares=50; Price=15.66; ]
```

挿入、更新、削除の各イベントは、対応するバケットに入ります。 たとえば、次のバケットを保持する削除イベントがストリームに入る場合は、“CSCO” のバケットには 2 つのイベントが存在するようになります。

```
[ Symbol='CSCO'; | Shares=50; Price=15.66; ]
```

イベント・キャッシュを宣言してイベントを結合することにより、その動作を変更できます。

```
eventCache(Trades, coalesce) events;
```

この場合、“CSCO” のバケットにはイベントが存在しなくなります。

集合演算を使用してバケットを計算できます。たとえば、バケットの共有の合計を計算する場合、`sum(events.Shares)` と記述します。 選択するバケットは、デフォルトでは、入力ストリームからの現在のイベントに関連付けられたバケットです。 `keyCache` 操作を使用してバケットを変更できます。

バケットの格納方法を変更するには、2 つの方法があります。たとえば、同じ数の共有を持つ Trades でカラムを指定して、イベントを複数のバケットにグループ分けできます。

```
eventCache(Trades[Shares]) eventsByShares;
```

または、同じ数の共有を持つものと同じ記号を持つものをグループ分けします。

```
eventCache(Trades[Symbol, Shares]) eventsBySymbolShares;
```

あるいは、1 つの大きなバケットにグループ化します。

```
eventCache(Trades[]) eventsAll;
```

フィールドでバケット内のイベントを並び替えることもできます。

```
eventCache(Trades, Price desc) eventsOrderByPrice;
```

これは、価格の降順でイベントを並び替えます。 `nth` 操作を使用して、各要素の順序を自由に変更できます。

入力ストリームに多くの更新がある場合、バケットが非常に大きくなることがあります。 イベントの最大数を指定するか、最大時間を指定するか、またはその両方を指定して、バケットのサイズを制御できます。 たとえば、バケットごとに 10 のイベントを最大数として設定できます。

```
eventCache(Trades[Symbol], 10 events) eventsBySymbol10Events;
```

または、20 秒を最大期間として設定します。

```
eventCache(Trades[Symbol], 20 seconds) eventsBySymbol20Seconds;
```

または、両方を指定します。

```
eventCache(Trades[Symbol], 10 events, 20 seconds) eventsSmall;
```

第 6 章 SPLASH の CCL への統合

CCL は Flex 演算子を使用して SPLASH コードを実行し、イベントを処理します。これらはローカル宣言ブロックを持ちます。これは、SPLASH 関数と変数の宣言のブロックです。 入力ストリームごとに 1 つのメソッド・ブロックを持ち、オプションで、SPLASH で記述されたタイマ・ブロックを持つこともできます。

イベントへのアクセス

イベントが入力ストリームから Flex 演算子に到達すると、その入力ストリームのメソッドが実行されます。

そのメソッドの SPLASH コードには、各入力ストリームに対して 2 つの暗黙的に宣言された変数があります。 1 つは、イベントに対する変数で、もう 1 つは、古いバージョンのイベントに対する変数です。 正確には、入力ストリームの名前が InputStream である場合は、変数は以下のようになります。

- 入力ストリームからのレコード・イベントのタイプを持つ `InputStream`。および
- 入力ストリームからのレコード・イベントのタイプを持つ `InputStream_old`。

入力ストリームのメソッドが実行されている場合、変数 `InputStream` は、そのストリームから到達したイベントにバインドされています。 イベントが更新である場合、変数 `InputStream_old` は、レコードの以前の内容にバインドされており、それ以外の場合は null となります。

注意： 削除イベントには、入力ストリームで以前保持されていたデータが常に入力されています。

Flex 演算子には複数の入力ストリームを含めることができます。たとえば、AnotherInput という別の入力ストリームがある場合は、変数 `AnotherInput` と `AnotherInput_old` は InputStream のメソッド・ブロックで暗黙的に宣言されます。 これらは、メソッド・ブロックが開始されると null に設定されますが、ブロック内で割り当てることができます。

入力ストリームへのアクセス

Flex 演算子のメソッドとタイマ・コード内で、あらゆる入力ストリームのレコードを検査できます。

正確には、以下の暗黙的に宣言された変数があります。

- `InputStream_stream` および
- `InputStream_iterator`。

変数 `InputStream_stream` は、値を検索する際は非常に便利です。`InputStream_iterator` はあまり一般的には使用されません。これは上級ユーザ向けです。

たとえば、以下のレコードを使用して入力ストリーム Trades からのイベントを処理しているとします。

```
[ Symbol='T'; | Shares=10; Price=22.88; ]
```

以下のレコードを含む、最新の収入データを格納している別の入力ストリーム Earnings を使用してもかまいません。

```
[ Symbol='T'; Quarter="2008Q1"; | Value=10000000.00;  ]
```

Earnings からのイベントを処理する場合は、以下を使用して最新の Trades データを検索できます。

```
Trades := Trades_stream[Earnings];
```

同じキー・フィールド Symbol を持つ Trades ストリームのレコード。 Trades ストリーム内に一致するレコードがない場合、結果は null になります。

Trades ストリームからのイベントを処理する場合は、以下を使用して収入データを検索できます。

```
Earnings := Earnings_stream{ [ Symbol = Trades.Symbol; | ] };
```

この構文では、角カッコではなく中カッコを使用しています。これは、意味が異なるためです。 Trades イベントには、Earnings ストリームのキーによって値を検索するための十分なフィールドがありません。特に、フィールド Quarter が欠落しています。 中カッコは、「Symbol フィールドが `Trades.Symbol` である Earnings ストリームのレコードの検索」を意味しています。 一致するレコードがない場合、結果は null になります。

複数のレコードを検索する必要がある場合、`for` ループを使用できます。 たとえば、赤字を検索するために Earnings ストリームをループ処理できます。

```
for (earningsRec in Earnings_stream) {
  if ( (Trades.Symbol = Earnings.Symbol) and (Earnings.Value < 0) ) {
    negativeEarnings := 1;
```

```
    break;
  }
}
```

SPLASH の他の `for` ループと同様、変数 `earningsRec` は、スコープがループの本文である新しい変数です。 以下のように、若干コンパクトに記述できます。

```
for (earningsRec in Earnings_stream where Symbol=Trades.Symbol) {
  if (Earnings.Value < 0) {
    negativeEarnings := 1;
    break;
  }
}
```

これは、`Trades.Symbol` と等しい Symbol フィールドを持つ Earnings ストリームのレコードに対してのみループ処理します。 `where` セクションのキー・フィールドをリストされる場合は、ループは非常に効率的に実行されます。 それ以外の場合は、`where` フォームは最初のフォームよりも名目上速いというだけです。

Flex 演算子を使用すると、ストリーム自体のレコードにアクセスできます。 たとえば、Flex 演算子の名前が Flex1 である場合、以下のようにして、入力ストリームを使用して好きなだけループを記述できます。

```
for (rec in Flex1) {
  ...
}
```

Output 文

通常、Flex 演算子メソッドでは、あるイベントに対して 1 つ以上のイベントを作成します。 レコードのストアに影響を与え、他のストリームにダウンストリームで送信する、これらのイベントを使用するには、`output` 文を使用します。

以下に、ダウンストリームで送信するために、ある注文書を 10 の新しい注文書に分割するコードを示します。

```
integer i:= 0;
while (i < 10) {
  output setOpcode([Id = i; |
                    Shares = InStream.Shares/10;
                    Price = InStream.Price; ], upsert);
}
```

これらの操作はそれぞれ、特に安全な操作であるアップサートであり、キーを持つレコードが存在しない場合は挿入に変換され、そうでない場合は更新に変換されます。

トランザクションについての注意

Flex 演算子メソッドは一度に 1 つのイベントを処理します。ただし、Event Stream Processor には、トランザクション・ブロック (挿入イベント、更新イベント、削除イベントのグループ) のデータを渡すことができます。

そのような場合、トランザクション・ブロックの各イベント上でメソッドが実行されます。 Event Stream Processor は次のような不変条件を保持します。 ストリームはトランザクション・ブロックを取り込み、トランザクション・ブロックを生成します。 常に、あるブロックは取り込まれ、あるブロックは生成されている状態です。 Flex 演算子はトランザクション・ブロックを分解し、ブロック内の各イベント上でメソッドを実行します。 メソッドは、`output` が 1 つにまとめられるすべてのイベント上で実行されます。 次に、Flex 演算子により、このブロックがそのレコードにアトミックに適用され、ダウンストリームのストリームにブロックが送信されます。

イベントの処理中に正しくないイベントが作成される場合、ブロック全体が拒否されます。 たとえば、null キー・カラムを使用してレコードを出力する場合があります。

```
output [ | Shares = InStream.Shares; Price = InStream.Price; ];
```

このトランザクション・ブロック全体が拒否されます。 同様に、次の暗黙的な挿入を行う場合があります。

```
output [Id = 4; |
        Shares = InStream.Shares;
        Price = InStream.Price; ];
```

4 に設定された ID を持つ Flex 演算子にすでにレコードが存在する場合、ブロックは拒否されます。 Event Stream Processor を起動して `-B` オプションを指定することにより、正しくないトランザクション・ブロックについてのレポートを取得できます。 キー・カラムが null にならないようにして、`setOpcode` を使用してアップサート・イベントまたは安全削除イベントを作成し、トランザクション・ブロックが受け入れられるようにすると便利な場合がよくあります。

トランザクション・ブロックは、他のストリームに送信される前はできるだけ小さくなります。 たとえば、コードで同じキーを使用した 2 つの更新を出力する場合、2 番目の更新のみがダウンストリームに送信されます。 コードで挿入、削除の順に出力する場合、その両方のイベントはトランザクション・ブロックから削除されます。 そのため、複数のイベントを出力できますが、トランザクション・ブロックにはその一部しか格納できません。

第 7 章 サンプル SPLASH コード

SPLASH コードの構造を理解するには、SPLASH コードのサンプルを確認するのが最良の方法です。

次のコード・サンプルでは、SPLASH の使用方法を示します。Event Stream Processor で実行可能な SPLASH を使用するプロジェクトを確認する場合は、「SPLASH を使用したプロジェクト」を参照してください。

内部パルス

株式市場フィードは、ストリームに送信されるいくつかの更新の良い例です。

株式市場フィードが各証券コードの最後のティックを保持するとします。 一部のダウンストリーム計算は、計算コストが高いことがあります。また、すべての変更に対する再計算が必要ない場合があります。 毎秒または 10 秒ごとにのみ再計算を行うことをおすすめします。 コストが高い再計算が、継続的ではなく定期的に行われるようにする更新の収集方法とパルス方法を説明します。

辞書データ構造とタイマ機能により内部パルスをコード化できます。 制御するストリームの名前が InStream であると仮定します。 最初に、Flex 演算子で 2 つのローカル変数を定義します。

```
integer version := 0;
dictionary(typeof(InStream), integer) versionMap;
```

これらの 2 つの変数には各レコードの現在のバージョンとバージョン番号があります。 入力ストリームからのイベントを処理する SPLASH コードは以下のとおりです。

```
{
  versionMap[InStream] := version;
}
```

Flex 演算子内の特別なタイマ・ブロックにより、挿入と更新が送信されます。

```
{
  for (k in versionMap) {
    if (version = versionMap[k])
      output setOpcode(k, upsert);
  }
  version++;
}
```

タイマ・ブロックを実行する間隔を秒単位で設定できます。 現在のバージョンのイベントのみがダウンストリームに送信され、更新の次のセットではバージョン番号が増加します。

InStream に挿入と更新のみがある場合、このコードが機能します。 これは、削除を行うためにコードを拡張する良い例です。

オーダー・ブック

株取引を参考にした次の例では、オーダー・ブックの上位を保持します。

以下のタイプのレコードを持つ、株の買値の Bid というストリームがあるとします (例は、提供側を考慮せずに簡単にしてあります)。

```
[integer Id; | string Symbol; float Price; integer Shares; ]
```

ここで、Id はキー・フィールドで、買値を一意に識別します。 買値は変更されることがあり、ストリームにより新しい買値が挿入されるだけでなく以前の買値が更新または削除されることもあります。

目的は、特定の株に対する買値が挿入または変更されたときに、上位 3 つの買値を出力することです。 Position が 1 ～ 3 の範囲の出力のタイプは以下のとおりです。

```
 [integer Position; | string Symbol; float Price; integer Shares; ]
```

たとえば、Bid が以下であるとします。

```
[Id=1; | Symbol='IBM'; Price=43.11; Shares=1000; ]
[Id=2; | Symbol='IBM'; Price=43.17; Shares=900]
[Id=3; | Symbol='IBM'; Price=42.66; Shares=800]
[Id=4; | Symbol='IBM'; Price=45.81; Shares=50]
```

次のイベントを持ちます。

```
[Id=5; | Symbol='IBM'; Price=46.41; Shares=75]
```

ストリームは以下のレコードを出力します。

```
[Position=1; Symbol='IBM'; | Price=46.41; Shares=75]
[Position=2; Symbol='IBM'; | Price=45.81; Shares=50]
[Position=3; Symbol='IBM'; | Price=43.17; Shares=900]
```

注意： 上部に最新の値が表示されます。

この問題を解決する方法の 1 つとして、株ごとにグループ化して、価格ごとにイベントを注文するイベント・キャッシュを使用します。

```
eventCache(Bids[Symbol], coalesce, Price desc) previous;
```

以下のコードは、`depth` 変数によって指定された深さのレベルに、オーダー・ブックの現在のブロックを出力します。

```
{
  integer i := 0;
  string symbol := Bids.Symbol;
  while ((i < count(previous.Id)) and (i < depth) ) {
    output setOpcode([ Position=i; Symbol = symbol; |
                       Price=nth(i,previous.Price);
                       Shares=nth(i,previous.Shares);
                       ], upsert);
    i++;
  }
  while (i < depth) {
    output setOpcode([ Position=i; Symbol=symbol ], safedelete);
    i++;
  }
}
```

第 8 章 SPLASH を使用したプロジェクト

2 つのプロジェクトで SPLASH の使用方法を示します。

以下のプロジェクトは、各証券コードの上位 3 つの株価を示します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

/* ******************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ******************************************************
 * Use Case a:
 *         Keep records corresponding to only the top three
 * distinct values. Delete records that falls of the top
 * three values.
 *
 * Here the trades corresponding to the top three prices
 * per Symbol is maintained. It uses
 * - eventcaches
 * - local UDF
 */
CREATE FLEX Top3TradesFlex
    IN QTrades
    OUT OUTPUT WINDOW Top3Trades SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
        DECLARE
            eventCache(QTrades[Symbol], manual, Price asc)
tradesCache;
            /*
             * Inserts record into cache if in top 3 prices and
returns
             * the record to delete or just the current record if it
was
             * inserted into cache with no corresponding delete.
             */
            typeof(QTrades) insertIntoCache( typeof(QTrades)
```

```
qTrades )
            {
               // keep only the top 3 distinct prices per symbol in
the
                // event cache
                integer counter := 0;
                typeof (QTrades) rec;
                long cacheSz := cacheSize(tradesCache);
                while (counter < cacheSz) {
                    rec := getCache( tradesCache, counter );
                 if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                        // if the price is the same update
                        // the record.
                        deleteCache(tradesCache, counter);
                        insertCache( tradesCache, qTrades );
                        return rec;
                        break;
                    } else if( qTrades.Price < rec.Price) {
                        break;
                    }
                    counter++;
                }

                //Less than 3 distinct prices
                if(cacheSz < 3) {
                    insertCache(tradesCache, qTrades);
                    return qTrades;
                } else {   //Current price is > lowest price
                    //delete lowest price record.
                    rec := getCache(tradesCache, 0);
                    deleteCache(tradesCache, 0);
                    insertCache(tradesCache, qTrades);
                    return rec;
                }

                return null;
            }
        END;

        ON QTrades {
            keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
            typeof(QTrades) rec := insertIntoCache( QTrades );
            if(rec.Id) {
                //When id does not match current id it is a
                //record to delete
                if(rec.Id <> QTrades.Id) {
                    output setOpcode(rec, delete);
                }
                output setOpcode(QTrades, upsert);
            }
        };
    END;
```

以下のプロジェクトでは、30 秒間データを収集して、目的の出力値を計算します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

/* ******************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ******************************************************
 * Use Case b:
 * Perform a computation every N seconds for records
 * arrived in the last N seconds.
 *
 * Here the Nasdaq trades data is collected for 30 seconds
 * before being released for further computation.
 */
CREATE FLEX PeriodicOutputFlex
    IN QTrades
    OUT OUTPUT WINDOW QTradesPeriodicOutput SCHEMA TradesSchema
PRIMARY KEY(Symbol,Price)
    BEGIN
        DECLARE
            dictionary(typeof(QTrades), integer) cache;
                END;
        ON QTrades {
                //Whenever a record arrives just insert into
dictionary.
              //The key of the dictionary is the key to the record.
            cache[QTrades] := 0;
        };
        EVERY 30 SECONDS {
                //Cycle through event cache and output all the rows
                //and delete the rows.
                for (rec in cache) {
                        output setOpcode(rec, upsert);
                }
                clear(cache);
        };
    END;

/**
 * Perform a computation from the periodic output.
 */
CREATE OUTPUT WINDOW QTradesSymbolStats
PRIMARY KEY DEDUCED
AS SELECT
```

```
    q.Symbol,
    MIN(q.Price)        Minprice,
    MAX(q.Price)        MaxPrice,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTradesPeriodicOutput q
GROUP BY
    q.Symbol
;
```

索引

C

F

N

O

S

あ

い

お

く

こ

さ

て

と