formura 文法仕様書

構造格子サブワーキンググループ

2016 年 1 月 22 日

概要

この文書では、ステンシル計算用のドメイン特化言語 formura[Lattice Sub Working Group, 2015] の文法と意味論をまとめます。

目次

1　この仕様書で用いる BNF について

この仕様書では文法を定義するのに、例文とあわせて BNF (Backus-Naur form, バッカス・ナウア記法) を用います。そこで、`formura` 文法の説明をはじめる前に BNF について導入的な説明を行います。

BNF とは、言語の文法を厳密に定義するのに用いられる記法の一つです。BNF では、終端記号から出発して、さまざまな非終端記号を定義してゆくことで文法を構築します。

‘`a`’ ‘`b`’ のように、一重引用符でくくられ等幅フォントで印字される項は終端記号 (terminal) であり、その文字列そのものを指します。これに対し、⟨*a*⟩ や ⟨*b*⟩ のように、山括弧でくくられたイタリック印字の項は非終端記号 (non-terminal) と呼ばれ、⟨*a*⟩ や ⟨*b*⟩ という名前が何を指すのかは他所で定義されていることを示します。定義には、BNF を用いることもありますが、自然言語で説明することもあります。

BNF において、左辺を右辺で定義するのには記号 `::=` を用います。左辺には常にひとつの非終端記号が来ます。右辺において、項をスペース区切りで連ねると、それらの項を並べた文法を意味し、また、縦棒 | は複数の形式からいずれか 1 つ選ぶことを意味します。

例えば、次の例において、⟨*digit-nonzero*⟩ は 0 以外の数字 1 文字を表し、⟨*digit*⟩ は 0 を含む 10 種類の数字を表します。

⟨*digit-nonzero*⟩ ::= ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

⟨*digit*⟩ ::= ‘`0`’ | ⟨*digit-nonzero*⟩

また、次の BNF では、⟨*function-definition*⟩ を、‘`function`’、⟨*argument-list*⟩、⟨*function-body*⟩、‘`end`’ を並べたものとして定義しています。

⟨*function-definition*⟩ ::= ‘`function`’ ⟨*argument-list*⟩ ⟨*function-body*⟩ ‘`end`’

次に、広く用いられている拡張 BNF として、{ } と [ ] という 2 種類の括弧があります。{⟨*a*⟩} は、⟨*a*⟩ の 0 回以上の繰り返しを表します。いっぽう [⟨*a*⟩] は、⟨*a*⟩ の 0 回または 1 回の繰り返し（⟨*a*⟩ を省略してもよいこと）を表します。

例えば、以下の例では ⟨*natural-number*⟩（正の整数）を、0 でない数字で始まり、その後いくつでも数字が連なるもの、として定義しています。

⟨*natural-number*⟩ ::= ⟨*digit-nonzero*⟩ { ⟨*digit*⟩ }

さらに、以下の例では、省略可能を示す括弧 [ ] も使って整数全体を表す文法を定義しています。具体的には、整数とは ‘`0`’、または、⟨*natural-number*⟩、または ‘`-`’ ⟨*natural-number*⟩ である、と定義されています。それぞれの形式は 0、正、負の整数に対応しています。

⟨*integer*⟩ ::= ‘`0`’ | [‘`-`’] ⟨*natural-number*⟩

最後に、本仕様書独自の記法として、三点リーダー (...) を前二項に作用する繰り返し記法とします。⟨*a*⟩ ⟨*b*⟩ ... という並びで ⟨*a-separated-by-b*⟩、つまり、⟨*b*⟩ で区切られた一つ以上の ⟨*a*⟩ の繰り返しを表すことにします。

⟨*a-separated-by-b*⟩ ::= ⟨*a*⟩
| ⟨*a*⟩ ⟨*b*⟩ ⟨*a-separated-by-b*⟩

たとえば、次の文法は、丸括弧で全体をくくられ、‘,’ で区切られた、1 個以上の ⟨*term*⟩ のリストを表しています。

⟨*comma-separated-list*⟩ ::= ‘(’ ⟨*term*⟩ ‘,’ ... ‘)’

上記の文法は、通常の拡張 BNF を用いれば以下のようにも表せます。

⟨*comma-separated-list*⟩ ::= ‘(’ ⟨*term*⟩ { ‘,’ ⟨*term*⟩ } ‘)’

応用例として、丸括弧で全体をくくられ、‘,’ で区切られた、0 個以上の ⟨*term*⟩ のリストは次の記法で表せます。

⟨*comma-separated-list*⟩ ::= ‘(’ [⟨*term*⟩ ‘,’ ...] ‘)’

2 formura のインストールと使用方法

2.1 バイナリインストール

formura の公式サイト https://github.com/nushio3/formura に行き、案内に従って最新の release をダウンロードしてください。

2.2 ソースインストール

formura はプログラミング言語 Haskell で書かれています。そこで、Haskell プログラムの開発環境を整えるためのプログラム、stack を使います。stack は Windows, Mac OS X, 様々な Linux ディストリビューションに対応しています。

stack の公式サイト http://docs.haskellstack.org/en/stable/README.html に行き、指示に従って stack をインストールしてください。

次に、formura のレポジトリからソースコードをダウンロードし、レポジトリのルートディレクトリ内で stack install コマンドを実行してください。

```
$ git clone git@github.com:nushio3/formura.git
$ cd formura
$ stack install
```

formura の実行ファイル群が stack のインストール先（デフォルトではホームディレクトリの.local/bin 以下）にインストールされます。

2.3 使用方法

コマンドラインプログラム formura に formura ソースコードを渡すと、C++ 言語のコードとヘッダが生成されます。生成するコードのファイル名は-o で指定できます。

```
$ formura example/diffusion.f31 -o output/diffusion.cpp
```

上記のコマンドにより、output/diffusion.h および output/diffusion.cpp が生成されます。生成されるコードの内容については §5 で詳しく解説します。

アルファベットで始まる識別子名に使える文字
アルファベットで始まる識別子名の 2 文字目以降に使える文字
記号文字で始まる識別子名に使える文字

	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

表 1　formura における 7 バイト ASCII 文字の分類。

3　formura 文法

3.1　文字

formura のソースコードは UTF-8 でエンコードされている必要があります。formura では以下の文字種別を定義します：空白文字 ⟨*space-character*⟩、アルファベット ⟨*alphabet-character*⟩、数字 ⟨*digit-character*⟩、記号文字 ⟨*symbol-character*⟩、文区切り文字 ⟨*punctuation-character*⟩。

空白文字は、ユニコード空白文字のうち、改行文字 \n を除いたものです。また、‘#’ から改行の直前までの文字列はコメントであり、字句解析上は空白文字とみなされます。

アルファベットはユニコードのアルファベットとアンダースコア _ からなります。アルファベットは、Unicode の General_Category 値が Letter である文字であり、英語の大小文字のほか、ひらがな、カタカナ、漢字、ハングルなどもアルファベットに含まれます。General_Category 値の詳細については以下の URL を参照してください。

http://www.unicode.org/reports/tr44/tr44-14.html%23GC_Values_Table

記号文字は、ユニコードの印刷可能文字のうちアルファベットと空白文字でないもの全てから、formura で特殊な意味を割り当てられている、以下の記号を除いたものです。’"\(),;[]#

例として、7Bit ASCII の範囲内の文字の分類については表 1 を参照してください。

3.2　字句解析

formura の字句は、空白、数値および文字列リテラル、識別子名、各種括弧からなります。

空白は、空白文字を並べたものです。空白は、字句を区切る以外の用途をもたず、単語区切りにはいくつあっても構いません。特に、formura はインデントに依存した文法を持ちません。

数値リテラルは一般的な浮動小数点記法に準じます (TODO: 詳しく)。formura の数値リテラルはデフォルトで、任意精度有理数型と解釈されます。

文字列リテラルは二重引用符 ‘"’ で囲まれた任意の文字列です (TODO: 詳しく)。

formura の識別子名 ⟨*identifier-name*⟩ は、ユーザーが変数名、演算子名などとして自由に使える文字列です。識別子名を作る方法は 2 通りあります：

- アルファベットから始まり、0 個以上のアルファベットと数字からなる列が続くものは識別子名です。
- 1 個以上の記号文字からなる文字列は識別子名です。

⟨*identifier-name*⟩ ::= ⟨*alphabet-character*⟩ {⟨*alphabet-character*⟩ | ⟨*digit-character*⟩}
| ⟨*symbol-character*⟩ {⟨*symbol-character*⟩}

formura の標準ライブラリでは、主に変数名や関数名にはアルファベット文字列、中置演算子名には記号文字列を用いていますが、formura の文法において変数名と演算子名との扱いが異なるわけではありません。したがって変数名に記号を用いたり、アルファベットからなる中置演算子を定義することができます。ただし、アルファベットと記号文字を混在させた識別子は許されないことにしています。これは、a+b のような式が、間にスペースがない場合に、ひとつの識別子 'a+b' としてではなく、加算式 $a + b$ として解釈されるようにするためです。

次の各行の文字列は formura の識別子名です。

```
alpha_beta_gamma
sanma398
HoNU
@
-.-
@*@-
!@$%^&*+=-/<>~.
```

次の各行の文字列は識別子名ではありません。

```
call/cc
solve-problem-life-matter-universe
398san
@kemi
#
\(^o^)/
```

3.3 文

formura のプログラムは複数の文 ⟨*statement*⟩ を文区切り ⟨*statement-delimiter*⟩ で区切ったものです。formura の文は、セミコロンもしくは改行で区切られます。

⟨*formura-program*⟩ ::= ⟨*statement*⟩ ⟨*statement-delimiter*⟩ ...

⟨*statement-delimiter*⟩ ::= ';'
| ⟨*newline*⟩

以下に、formura の文と区切り文字の例を示します。

```
a = b; c = d
a[i] = b[i]
a[i] = b[i];;
```

formura の文には、代入文 ⟨*substitution-statement*⟩、型宣言文 ⟨*type-declaration*⟩ 、特殊宣言文 ⟨*special-declaration*⟩、関数定義 ⟨*function-definition*⟩ があります。また、何もない空文 ⟨*empty*⟩ も許されます。

代入文は、⟨ 左辺パターン ⟩ = ⟨ 式 ⟩ の形です。型宣言文と特殊宣言文は類似した文法をもっていて、それぞれ ⟨ 型名 ⟩ :: ⟨ 変数名 ⟩ および ⟨ 特殊宣言名 ⟩ :: ⟨ 変数名 ⟩ の形をとります。関数定義の形式については後述します。

宣言文の右辺には、カンマ区切りで複数の変数名を書くことができます。

⟨*statement*⟩ ::= ⟨*substitution-statement*⟩ | ⟨*type-declaration*⟩ | ⟨*compound-statement*⟩ | ⟨*special-declaration*⟩ | ⟨*function-definition*⟩ | ⟨*empty*⟩

⟨*substitution-statement*⟩ ::= ⟨*left-hand-side-pattern*⟩ '=' ⟨*expression*⟩

⟨*type-declaration*⟩ ::= ⟨*type-name*⟩ '::' ⟨*variable-name-list*⟩

⟨*special-declaration*⟩ ::= ⟨*special-declaration-name*⟩ '::' ⟨*variable-name-list*⟩
| ⟨*dimension-declaration*⟩

⟨*variable-name-list*⟩ ::= ⟨*variable-name*⟩ ',' ...

3.3.1 特殊宣言文

離散化アルゴリズムの指定に関わる特殊宣言文は、空間次元宣言文 'dimension' と座標軸名宣言文 'axes' です。

空間次元宣言文はステンシル計算に用いる空間グリッドの次元を宣言します。座標軸名宣言文は各座標にわりあてる名前を宣言します。これらの軸名は、formura が生成するコードにおいて、各空間次元の大きさの定数やループ添字変数の名前を作るのに使われます。

とくに、計算空間全体の解像度に対応する定数は、軸の名前を大文字にして 'N' を加えたものになります。たとえば、'axes :: x, y, z' と指定しているならば、解像度は 'NX' × 'NY' × 'NZ' です。これらの変数名は formura のコード側からも参照できます。

例えば次の例では 3 次元の格子を初期化する関数を指定しています。

```
dimension :: 3
axes :: x, y, z

begin function dens = init()
  Real [,,] :: dens
```

```
    dens[i,j,k]  = sin(2 * pi * i / NX) * j + k
  end function
```

他の特殊宣言文、‘intra_node_shape’, ‘mpi_grid_shape’, ‘temporal_blocking_interval’, ‘monitor_interval’については、並列化コードの生成に関わるものですので、§5で詳しく解説します。

3.3.2 関数定義文

関数定義の文法を次に示します。

⟨*function-definition*⟩ ::= ‘begin’ ‘function’ ⟨*return-value*⟩ ‘=’ ⟨*function-name*⟩ ⟨*arguments-pattern*⟩
⟨*statement*⟩ ⟨*statement-delimiter*⟩ ...
‘end’ ‘function’

⟨*return-value*⟩ ::= ⟨*expression*⟩

⟨*arguments-pattern*⟩ ::= ⟨*variable-name*⟩
| ‘(’⟨*arguments-pattern*⟩ ‘,’ ... ‘)’

関数定義は‘begin’ ‘function’で始まり、返値、関数名、引数の指定が続き、その後に関数の本文があって、‘end’ ‘function’で終わります。

関数は0個以上の引数と0個以上の帰値をもつことができ、それぞれ⟨*return-values-pattern*⟩および⟨*arguments-pattern*⟩に従って、タプルで表します。これは、関数が任意個数の入力と任意個数の出力を取れるようにするための仕様です。

関数の入力⟨*arguments-pattern*⟩、返値、および関数内部で登場する変数名については、すべて関数内で型宣言されている必要があります。(TODO:将来的には型推論を入れよう。)

3.3.3 複文

formuraでは、型宣言文および複数の代入文を複合させて簡潔に記述することができます。この形式を複文と呼びます。

⟨*compound-statement*⟩ ::= ⟨*type-name*⟩ ‘::’ (⟨*variable-name*⟩ | ⟨*substitution-statement*⟩) ‘,’ ...

複文で宣言されるすべての変数の型は同じです。複文の例を以下に示します。

```
double :: c = 3, d, e = 6
```

これは次の文と同じです。

```
double :: c = 3
double :: d
double :: e = 6
```

3.3.4 プログラム例

以下に代入文、型宣言文、特殊宣言文、関数定義文の例を示します。

型名	意味
‘integer’	整数
‘rational’	無限精度有理数
‘float’	単精度実数
‘double’	倍精度実数
‘Real’	ユーザー定義型の実数
‘complex_float’	単精度複素数
‘complex_double’	倍精度複素数
‘Complex’	ユーザー定義型の複素数
‘string’	文字列型

表 2　要素型 ⟨*element-type-name*⟩ に含まれる名前とその意味

```
dimension :: 2
axes :: X,Z

a = b

complex_double :: c, d, e, f

begin function y = sin(x)
  double :: y, x
  y = x - x**3 / 6 + x**5 / 120 - x**7 / 5040
end function

begin function (next_x, next_y) = update(x,y)
  double[,] :: y, x, next_x, next_y
  next_x =  x * y
  next_y =  x + y
end function
```

3.4　型

formura の型は、要素型、および型のグリッド (格子)、型のタプル (組)、型のベクトル (ランダムアクセス可能な配列) からなります。

要素型の一覧については表 2 を参照してください。グリッドは、ステンシル計算の格子を表現するデータ型です。これに対しベクトルは、通常の意味でのランダムアクセス可能な配列です。また、タプルは複数の (異なってもよい) 型の組です。

⟨*type-name*⟩ ::= ⟨*element-type-name*⟩ (要素型)
| ⟨*type-name*⟩ '[' ⟨*offset-expression*⟩ ',' ... ']' (グリッド)
| '(' ⟨*type-name*⟩ ',' .. ')' (タプル)
| ⟨*type-name*⟩ '(' ⟨*vector-size*⟩ ',' ... ')' (ベクトル)

⟨*element-type-name*⟩ ::= 'integer' | 'rational' | 'float' | 'double' | 'complex_float' | 'complex_double'
| 'Real' | 'Complex' | 'string'

以下に要素型、グリッド、タプル、ベクトルを定義する例を示します。⟨*offset-expression*⟩ の値が 0 のときは省略できる仕様です。また、グリッドのタプルのベクトルの…といった複雑な型を作る場合、型コンストラクタは左から順に結合することに注意してください。

グリッド、ベクトルはいずれも多次元にすることができます。ただし、グリッドの次元は 'dimension' 宣言で指定したのと同じでなくてはなりません。また、2 重以上のグリッド (グリッドのグリッド等) はサポートされません。

```
Real :: a
Real[1/2,0] :: b
Real[1/2,] :: b2
(Real, Real, Real)[,] :: velocity
Real(3)[,] :: velocity_as_array_of_structure
Real[,](3) :: velocity_as_structure_of_array
```

formura において、n 次元のベクトルは n 重のベクトルと等価であり、さらに m 要素の一次元ベクトルは同じ型を m 個要素にもつ m-タプルと等価です。例えば以下の文において、4 つの変数 't1' ... 't4' はすべて互換性があります。

```
Real(3,2) :: t1
Real(3)(2) :: t2
(Real, Real, Real)(2) :: t3
((Real, Real, Real),(Real, Real, Real)) :: t4

t4(i,j) = t1(i,j)
t3(i) = t2(i)
```

3.5 式

formura の式は、即値 ⟨*literal*⟩、変数名 ⟨*variable-name*⟩、二項演算子式 ⟨*binary-operator-expression*⟩、単項演算子式 ⟨*unary-operator-expression*⟩、if 文 ⟨*if-then-else-expression*⟩、括弧式 ⟨*parenthesis-expression*⟩、格子アクセス式 ⟨*grid-access-expression*⟩、要素アクセス式 ⟨*projection-expression*⟩、関数呼び出し ⟨*function-call-expression*⟩ から構成されます。⟨*literal*⟩ には数字リテラルと文字列リテラルとがあり、⟨*variable-name*⟩ の文法は ⟨*identifier-name*⟩ と同一です。それ以外の式の文法は以下のとおりです。

⟨*binary-operator*⟩	(結合性)	⟨*unary-operator*⟩	意味
‘’ (併置)	左結合		関数呼び出し、配列アクセス等
‘`.`’	右結合		関数合成
‘ `**`’	右結合		冪乗
‘ `*`’, ‘ `/`’	左結合		乗除算
‘ `+`’, ‘ `-`’	左結合	‘ `+`’, ‘ `-`’	加減算、符号
‘`<`’, ‘`<=`’, ‘`==`’, ‘`!=`’,‘`>=`’, ‘`>`’	多結合		比較演算子
		‘`!`’, ‘`not`’	論理否定 (not)
‘`&&`’, ‘`and`’	左結合		論理積 (and)
‘`||`’, ‘`or`’	左結合		論理和 (or)
‘`|`’	右結合		格子定義域の拡張
‘`,`’	多結合		カンマ区切りリスト

表 3　演算子と優先順位

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*variable-name*⟩ | ⟨*binary-operator-expression*⟩ | ⟨*unary-operator-expression*⟩
| ⟨*if-then-else-expression*⟩ | ⟨*parenthesis-expression*⟩ | ⟨*grid-access-expression*⟩
| ⟨*projection-expression*⟩ | ⟨*function-call-expression*⟩

⟨*binary-operator-expression*⟩ ::= ⟨*expression*⟩ ⟨*binary-operator*⟩ ⟨*expression*⟩

⟨*unary-operator-expression*⟩ ::= ⟨*unary-operator*⟩ ⟨*expression*⟩

⟨*if-then-else-expression*⟩ ::= ‘`if`’ ⟨*expression*⟩ ‘`then`’ ⟨*expression*⟩ ‘`else`’ ⟨*expression*⟩

⟨*parenthesis-expression*⟩ ::= ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*offset-expression*⟩ ::= ⟨*variable-name*⟩ ‘`[`’ ⟨*offset-expression*⟩ ‘`,`’ ... ‘`]`’

⟨*grid-access-expression*⟩ ::= ⟨*variable-name*⟩ ‘`[`’ ⟨*cursor-expression*⟩ ‘`,`’ ... ‘`]`’

⟨*projection-expression*⟩ ::= ⟨*variable-name*⟩ ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*function-call-expression*⟩ ::= ⟨*function-name*⟩ ‘`(`’ [⟨*expression*⟩ ‘`,`’ ...] ‘`)`’

3.6　演算子と優先順位

演算子のあいだには優先順位と結合性が定義されています。表 3 を参照してください。

優先順位が高い演算子ほど強く結合します。例えば a+b*c は a+(b*c) の意味に、-b*c は-(b*c) の意味に、-b<c は (-b) < c の意味になります。

結合性は、優先順位が同じ演算子どうしが連続した場合の解釈のしかたを決めます。例えば '**' は右結合なので、3**3**3 は 3**(3**3) の意味になりますが、'-' は左結合なので、3-4-5 は (3-4)-5 の意味になります。

多結合演算子は、連続して登場した場合には一塊として解釈されます。例えば、formura において 3<a<5 は (3<a)&&(a<5) を意味します。

3.7 式の左辺

formura の代入文の左辺に来る ⟨*left-hand-side-pattern*⟩ については、変数名、タプルのパターンマッチ、格子変数へのアクセスが書けます。

⟨*left-hand-side-pattern*⟩ ::= ⟨*variable-name*⟩
| ⟨*left-hand-side-pattern*⟩ '[' ⟨*cursor-expression*⟩ ',' ... ']'
| '(' ⟨*left-hand-side-pattern*⟩ ',' .. ')'

以下に、いくつかの代入文を示します。以下の例に示すように、代入文の左辺にスカラー変数が来る場合には変数名パターンが利用でき、グリッド変数が来る場合には変数パターンおよび、その次元数に対応したカーソルパターンを用いた左辺パターンが利用できます。タプルやベクトル変数は、'()' によって添字アクセスします。

```
double :: a, b
double[0,0] :: c, d, e ,f
(double, int) :: g
int :: n

a = b
c[i,j] = d[i,j+1]
e = f
(a,n) = g
n = g(1)
```

3.8 formura 文法の全定義

⟨*formura-program*⟩ ::= ⟨*statement*⟩ ⟨*statement-delimiter*⟩ ...

⟨*statement-delimiter*⟩ ::= ';' | ⟨*newline*⟩

⟨*statement*⟩ ::= ⟨*substitution-statement*⟩
| ⟨*type-declaration*⟩
| ⟨*compound-statement*⟩
| ⟨*special-declaration*⟩
| ⟨*function-definition*⟩
| ⟨*empty*⟩

⟨*substitution-statement*⟩ ::= ⟨*left-hand-side-pattern*⟩ '=' ⟨*expression*⟩

⟨*type-declaration*⟩ ::= ⟨*type-name*⟩ '::' ⟨*variable-name-list*⟩

⟨*special-declaration*⟩ ::= ⟨*special-declaration-name*⟩ '::' ⟨*variable-name-list*⟩
| ⟨*dimension-declaration*⟩

⟨*variable-name-list*⟩ ::= ⟨*variable-name*⟩ ',' ...

⟨*type-name*⟩ ::= ⟨*element-type-name*⟩ (要素型)
| ⟨*type-name*⟩ '[' ⟨*offset-expression*⟩ ',' ... ']' (グリッド)
| '(' ⟨*type-name*⟩ ',' .. ')' (タプル)
| ⟨*type-name*⟩ '(' ⟨*vector-size*⟩ ',' ... ')' (ベクトル)

⟨*element-type-name*⟩ ::= 'integer' | 'rational' | 'float' | 'double' | 'complex_float' | 'complex_double'
| 'Real' | 'Complex' | 'string'

⟨*special-declaration*⟩ ::= 'dimension' | 'axes' | 'intra_node_shape' | 'mpi_grid_shape' |
'temporal_blocking_interval' | 'monitor_interval'

⟨*function-definition*⟩ ::= 'begin' 'function' ⟨*return-value*⟩ '=' ⟨*function-name*⟩ ⟨*arguments-pattern*⟩
⟨*statement*⟩ ⟨*statement-delimiter*⟩ ...
'end' 'function'

⟨*return-value*⟩ ::= ⟨*expression*⟩

⟨*arguments-pattern*⟩ ::= ⟨*variable-name*⟩
| '`(`'⟨*arguments-pattern*⟩ '`,`' ... '`)`'

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*variable-name*⟩
| ⟨*expression*⟩ ⟨*binary-operator*⟩ ⟨*expression*⟩
| ⟨*unary-operator*⟩ ⟨*expression*⟩
| '`if`' ⟨*expression*⟩ '`then`' ⟨*expression*⟩ '`else`' ⟨*expression*⟩
| '`(`' ⟨*expression*⟩ '`)`'
| ⟨*variable-name*⟩ '`[`' ⟨*expression*⟩ '`,`' ... '`]`'
| ⟨*function-name*⟩ '`(`' [⟨*expression*⟩ '`,`' ...] '`)`'

⟨*left-hand-side-pattern*⟩ ::= ⟨*variable-name*⟩
| ⟨*left-hand-side-pattern*⟩ '`[`' ⟨*cursor-expression*⟩ '`,`' ... '`]`'
| '`(`' ⟨*left-hand-side-pattern*⟩ '`,`' .. '`)`'

⟨*variable-name*⟩ ::= ⟨*identifier-name*⟩

⟨*function-name*⟩ ::= ⟨*identifier-name*⟩

⟨*identifier-name*⟩ ::= ⟨*alphabet-character*⟩ {⟨*alphabet-character*⟩ | ⟨*digit-character*⟩}
| ⟨*symbol-character*⟩ {⟨*symbol-character*⟩}

4 formura 文法における演算の意味論

4.1 formura の演算

formura の目的は全ての代入文の左辺の値を計算することです。formura の代入文は上から順番に評価されます。

formura の代入文は静的単一代入です。これは、各行の変数を区別し、ある行の変数は前の行までの変数で定義されていると解釈するという意味です。たとえば、つぎのプログラムは

```
a[i] = sin(i)
a[i] = a[i-1] + a[i+1]
b[i] = a[i]
```

以下のプログラムであるかのように解釈されます。

```
a_0[i] = sin(i)
a_1[i] = a_0[i-1] + a_0[i+1]
b_2[i] = a_1[i]
```

これに伴い、一つの代入文で同一名の変数を複数回設定するような文はエラーとします。

```
(s,o,s) = help()
```

4.2 formura の意味論

スカラー変数には、ただ 1 つの値が対応します。

タプルは要素番号を取って値を返す関数に対応させます。ベクトルに関しては、タプルと同一視することで意味をつけます。

グリッド変数は formura においてステンシル計算を表現する重要な型です。d 次元のグリッド変数は、d 個の有理数 $(r_i, ..^{i\in 0..d-1})$ によりオフセットを指定できます。グリッド変数の値は、d 次元ユークリッド空間において、整数座標から $(r_i, ..)$ だけずれた格子点 $(n_i + r_i, ..^{i\in 0..d-1, n_i\in\mathbb{Z}})$ において定義されます。いわばグリッド変数は $(n_i + r_i, ..)$ を受け取って値を返す関数です。

formura におけるグリッド変数は、必ず (オフセットされた) 任意の格子点にて定義されていると解釈することに注意してください。たとえ境界条件があっても必ずです。このことは、PiTCH 計算モデルを当てはめるうえで必要です。

TODO: フォーマルな表示的意味論を整備する。

4.3 formura の暗黙の型変換

算術二項演算 '+', '*', ... に渡す 2 つの値は、型が一致している必要があります。型が一致しない場合、formura は両者を共通に格納できる「大きな」型への変換を試みます。たとえば、スカラーとタプルの掛け算、スカラーとグリッドの掛け算は、それぞれスカラー値のほうの意味を拡張し、タプルの各要素、グリッドの各要素とそのスカラー値の掛け算を行います。

より詳細に、スカラーとタプル、もしくは、タプルとスカラーの間の演算は、そのスカラーとタプルの各要素間の演算として定義します。また、同数の要素をもつタプルどうしの演算は、同じ位置にある要素どうしの演算として定義します。この定義は、多重のタプルに対しては再帰的に適用されるものとします。

```
z * (a, b, c) = (z*a, z*b, z*c)
(a, b, c) / d = (a/d, b/d, c/d)
(a, b) + (x, y) = (a + x, b + y)
(a, b) - (x, (y, z)) = (a - x, (b - y, b - z))
```

異なる要素数のタプル同士や、要素型が異なるタプル同士、オフセットが異なるグリッド同士は互換性がありません。このような互換性のない型の値どうしの演算はエラーとします。

同様の型拡張ルールは‘`if`’式にも適用されます。‘`if`’式は三項演算子として解釈され、さらに

4.4 添字を明示した演算

グリッド変数に対し、左辺パターンを使って添字を明示的に指定した形の代入文を書いた場合、左辺のグリッド変数の各要素の値を直接、右辺式で定義する、という意味になります。この記法を使った代入文の例を以下に示します。

```
a[i,j] = sin(- i * j * lambda_z) + b[i,j] * c[i,j]
```

左辺、右辺ともに、グリッド添字変数には通常の変数と同様、任意の演算を施すことができます。

ただし、グリッド添字変数を含む式をグリッド変数へのアクセスに使う場合は、グリッド添字変数に対して施しうる演算は、定数式の加減算のみです。また、左辺で i 番目の座標軸に使われたパターン変数は、右辺でも同一の座標軸の添字にしか使うことができません。この二つのルールにより、グリッド添字にかんしてステンシル性を担保します。

```
double [] :: a
double [1/2] :: b
double [1/4] :: c
b[i+1/2] = a[i] + a[i+1] + c[i+1/4]
```

上記のように、演算したいグリッド変数どうしのオフセットが異なる場合、添字変数に有理数を加減算することでオフセットを合わせる必要があります。オフセットが異なるグリッド変数どうしの演算は型エラーとなります。

左辺で定義されている添字変数は右辺で省略することができます。また 0 であるオフセット値も省略できます。添字同士の区切りである‘`,`’も省略できます。

グリッド変数の添字の数は‘`dimension`’宣言で定義されているはずです。したがって 0 添字や‘`,`’を省略しても添字の数にかんして曖昧性は生じないことに注意してください。省略されているオフセット値はすべて 0 とみなします。

4.5 添字を省いたグリッド演算

formura では、グリッド変数同士に直接加減乗除などの演算を施すことができます。グリッド変数同士を直接演算する場合、グリッド変数のオフセットが一致している必要があります。

```
double [,] :: a, b, c
a = b + sin(c)
```

さきほどの計算ですが、添字をとことん省略して書くと以下のようになります。

```
double [] :: a
double [1/2] :: b
double [1/4] :: c
b[1/2] = a + a[1] + c[1/4]
```

もっとも、単項 '+' 演算子だけは省略しないほうがわかりやすいかもしれません。

```
b[+1/2] = a + a[+1] + c[+1/4]
```

上記のプログラムのまた違った解釈として、'[+1]' を、配列を一つ左向きにずらす演算子だと思うこともできます。

ただし、次のコードが意味をなさないのと同様、

```
double [0] :: a
a[i] = a[i+1][i+1]
```

次のようなコードも許されません。'[+1]' はあくまでも、グリッド添字変数の省略形であることを忘れないでください。

```
double [0] :: a
a[i] = a[+1][+1]
```

5 formura の生成するコード

5.1 コード実行の流れ

C 言語のプログラムが ‘main’ 関数から開始するように、formura のプログラムでは、‘init’ と ‘step’ という名前の関数が、コード生成の起点として特別な意味を持ちます。

formura のプログラムは、最初に ‘init’ 関数を一つ呼び出してシミュレーションの初期状態を作り、その後 ‘step’ 関数を繰り返し呼び出して状態を更新していきます。C 言語の擬似コードで書けば、次のようになります。

```
simulation_state = init();
for (t = 0; t < T_MAX; ++t) {
  simulation_state = step(simulation_state);
}
```

‘init’ 関数は 0 個の引数をとり、グリッド型の値、またはいくつかのグリッドを含むタプルを返すような関数でなくてはなりません。‘step’ 関数は ‘init’ 関数が返す型を受け取り、同じ型を返す関数である必要があります。

formura は、コード生成にあたって、シミュレーションの独立変数を保持する配列変数を名付けるとき、‘step’ 関数の引数に使われた名前を採用します。なぜなら、formura プログラムの中でシミュレーションの全状態がひとかたまりになっている箇所は三つ（‘init’ 関数の返り値、‘step’ 関数の引数 ‘step’ 関数の返り値）ありますが、このうち「‘step’ 関数の引数」だけが左辺式であって、各変数に独立な名前がついていることが保証されているからです。

5.2 C/C++ 言語からの呼び出し規約

formura の生成するコードの命名規約は、MPI ライブラリと違和感なく使えるように定めることにします。formura は C++ のヘッダとソースコードを組で生成します。ユーザーは通常の方法でコンパイル・自身のプログラムとリンクして実行ファイルを作ってください。

formura が生成したコードを利用するユーザー側のプログラムは次のようになります。

```
#include "mpi.h"
#include "output.h"

int main (int argc, char **argv) {
  Formura_Navigator navi;

  MPI_Init(argc, argv);
  Formura_Init(&navi, MPI_WORLD_COMM);

  while(navi.time_step < T_MAX) {
    Formura_Forward(&navi);
  }

  MPI_Finalize(argc, argv);
}
```

■生成されるコードの内容

formura　の生成するヘッダには、次のものが含まれます。

- シミュレーションの独立変数に対応する配列変数（以下、状態配列 state array と呼びます）
- Formura_Navigator 構造体
- Formura_Init 関数
- Formura_Forward、関数

状態配列は、formura が生成したコードからも、ユーザーからも読み書きされることを想定しています。formura の生成した関数を呼び出していないとき、ユーザーは状態配列を自由に更新することができます。

Formura_Navigator 構造体は、formura の関数からユーザーに対し、いま状態配列に、どの時点・どの範囲の物理量が入っているのか通知するために使います。

たとえば、dimension :: 2 axes :: x, y が指定されている場合、Formura_Navigator 構造体は次のようになります。

```
struct Formura_Navigator {
  int time_step;
  int lower_x;
  int upper_x;
  int lower_y;
  int upper_y;
};
```

これは、t = time_step lower_x $\leq x <$ upper_x lower_y $\leq y <$ upper_y の範囲の値が状態配列に格納されていることを意味しています。

formura は次のことを保証します：monitor_interval タイムステップごとの、計算領域内の物理量は、少なくとも 1 つの MPI ノードから読むことができる。

■Formura_Init 関数

```
int Formura_Init (Formura_Navigator *navi, MPI_Comm comm)
```

Formura_Init 関数は、formura ソースコードの ‘init’ 関数に対応する計算を行い、結果を状態配列に書き込みます。

Formura_Init 関数が返すナビゲータは、必ず time_step = 0 を満たします。

Formura_Init 関数は MPI_Comm 型の引数をひとつ取ります。formura が行う通信は、ここの Formura_Init で指定された MPI コミュニケータを使います。

■Formura_Forward 関数

```
int Formura_Forward (Formura_Navigator *navi)
```

Formura_Forward 関数は、状態配列を読み取り、formura ソースコードの ‘step’ 関数に対応する計算を monitor_interval 回繰り返し行い、結果を状態配列に書き込みます。

Formura_Forward 関数が返すナビゲータの time_step は、直前の値より monitor_interval だけ増加しています。

■formura の生成するコードの並列度や解像度

formura の生成するコードの並列度や解像度は、formura の特殊宣言での指定から決まります。以下に、formura の特殊宣言の一覧を再掲します。

```
dimension :: 3
axes :: x, y, z
intra_node_shape :: 64, 64, 64
mpi_grid_shape :: 40, 20, 5
temporal_blocking_interval :: 2
monitor_interval :: 120
```

各ノードが担当する状態配列の大きさは、およそ ‘intra_node_shape’ 程度の大きさになります。但し、temporal blocking や通信の都合上、この大きさは多少変更されることがあります。これらのノードを、‘mpi_grid_shape’ で指定された数だけ x, y, z 方向に並べた領域が計算領域となります。したがって、上記の例では計算の解像度は $2560 \times 1280 \times 320$ となります。さらに、‘step’ 関数の計算を temporal blocking で 2 ステップぶん一気に計算するコードが生成され、ユーザーコード側からは、120 ステップごとに物理状態を観測することができます。

この仕様に伴って、`Formura_Init` 関数に渡す MPI コミュニケータ内にある MPI ランクの数は、'`mpi_grid_shape`' の積と一致している必要があります。

5.3 Fortran からの呼び出し規約

`formura` は将来的に Fortran からも使えるようにする予定です。ここで障害となるのは、Fortran 処理系ごとに配列変数のメモリ上での表現が異なっている場合がある、ということです。

このため、Fortran 版では、`monitor_interval` ごとに、物理量を Fortran 形式の配列に変換して書き込んでユーザーに見せ、変更があった場合はその配列から再度読み込んでシミュレーションを継続する方式を予定しています。

6 formura 文法の利用

以上の構文を用いて、ステンシル計算を書いてみます。

formura が最低限できる必要があるのは、シンプルなグリッド変数どうしの演算です。具体的には、

1. グリッドの次元とサイズをを宣言する
2. 各グリッド点にある変数の値を計算する
3. 差分スキームを記述する（空間的に隣接するセルの値を取ってくる）
4. 時間積分スキームを記述する（時間的に次のセルに値を渡す）

ことができれば足ります。

以下に、formura を使って記述したプログラムを示します。

```
dimension :: 3
axes :: x, y, z
single_node_shape :: 64, 64, 64
mpi_shape :: 8,8,8



ddx = for(a) (a[i+1/2,j,k] - a[i-1/2,j,k])/2
ddy = for(a) (a[i,j+1/2,k] - a[i,j-1/2,k])/2
ddz = for(a) (a[i,j,k+1/2] - a[i,j,k-1/2])/2

∂ = (ddx,ddy,ddz)

Σ = for (e) e(0) + e(1) + e(2)

begin function init() returns dens_init
      float [] :: dens_init = 0
end function

begin function dens_next = step(dens)
  float :: Dx, Dt
  Dx = 4.2
  Dt = 0.1
  dens_next = dens + Dx**2/Dt * Σ for(i) (∂ i . ∂ i) dens
end function
```

タプル型の設計にあたっては Pierce [2002] を参考にしました。また、タプルの言語への組み込みにあたっては Oliveira et al. [2015] を参考にしました。

参考文献

Lattice Sub Working Group. The formura specification. 2015.

B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular reifiable matching: a list-of-functors approach to two-level types. In Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, pages 82–93. ACM, 2015.

B. C. Pierce. Types and programming languages. MIT press, 2002.