Pyccel Documentation

Release 1

A. Ratnani

Oct 18, 2018

Contents

Pyccel stands for Python extension language using accelerators.

The aim of **Pyccel** is to provide a simple way to generate automatically, parallel low level code. The main uses would be:

1. Convert a *Python* code (or project) into a Fortran
2. Accelerate *Python* functions by converting them to *Fortran* then calling *f2py*.

Pyccel can be viewed as:

- *Python-to-Fortran* converter
- a compiler for a *Domain Specific Language* with *Python* syntax

Pyccel comes with a selection of **extensions** allowing you to convert calls to some specific python packages to Fortran. The following packages will be covered (partially):

- numpy
- scipy
- mpi4py
- h5py

Todo: add links for additional information

CHAPTER 1

Pyccel documentation contents

1.1 First Steps with Pyccel

This document is meant to give a tutorial-like overview of Pyccel.

The green arrows designate "more info" links leading to advanced sections about the described task.

By reading this tutorial, you'll be able to:

- compile a simple *Pyccel* file
- get familiar with parallel programing paradigms
- create, modify and build a *Pyccel* project.

1.1.1 Install Pyccel

Install Pyccel from a distribution package with

```
$ python setup.py install --prefix=MY_INSTALL_PATH
```

If **prefix** is not given, you will need to be in *sudo* mode. Otherwise, you will have to update your *.bashrc* or *.bash_profile* file with. For example:

```
export PYTHONPATH=MY_INSTALL_PATH/lib/python2.7/site-packages/:$PYTHONPATH
export PATH=MY_INSTALL_PATH/bin:$PATH
```

Todo: add installation using **pip**

For the moment, *Pyccel* generates only *fortran* files. Therefor, you need to have a *fortran* compiler. To install **gfortran**, run:

```
$ sudo apt install gfortran
```

In order to use the commands **`pyccel-quickstart`** and **`pyccel-build`**, you will need to install **cmake**:

```
$ sudo apt install cmake
```

1.1.2 Simple Examples

In this section, we describe some features of *Pyccel* on simple examples.

Hello World

Create a file *helloworld.py* and copy paste the following lines (be careful with the indentation)

See `hello world script`.

Now, run the command:

```
pyccel helloworld.py --execute
```

the result is:

```
> * Hello World!!
```

The generated *Fortran* code is

```
program main

implicit none

!
call helloworld()

contains
! ........................................
subroutine helloworld()
  implicit none


  print * ,'* Hello World!!'

end subroutine
! ........................................


end
```

Matrix multiplication

Create a file *matrix_multiplication.py* and copy paste the following lines

See `matrix multiplication script`.

Now, run the command:

```
pyccel matrix_multiplication.py --execute
```

This will parse the *Python* file, generate the corresponding *Fortran* file, compile it and execute it. The result is:

```
-1.0000000000000000         0.0000000000000000         -2.0000000000000000         1.
↪0000000000000000
```

The generated *Fortran* code is

```
program main

implicit none
real(kind=8), pointer :: a (:, :)
real(kind=8), pointer :: c (:, :)
real(kind=8), pointer :: b (:, :)
integer :: i
integer :: k
integer :: j
integer :: m    = 4
integer :: n    = 2
integer :: p    = 2

!
n = 2
m = 4
p = 2
allocate(a(0:n-1, 0:m-1)); a = 0.0
allocate(b(0:m-1, 0:p-1)); b = 0.0
allocate(c(0:n-1, 0:p-1)); c = 0.0
do i = 0, -1 + n, 1
  do j = 0, -1 + m, 1
    a(i, j) = i - j
  end do

end do
do i = 0, -1 + m, 1
  do j = 0, -1 + p, 1
    b(i, j) = i + j
  end do

end do
do i = 0, -1 + n, 1
  do j = 0, -1 + p, 1
    do k = 0, -1 + p, 1
      c(i, j) = a(i, k)*b(k, j) + c(i, j)
    end do

  end do

end do
print * ,c

end
```

Functions and Subroutines

Create a file *functions.py* and copy paste the following lines

See `functions script`.

Now, run the command:

```
pyccel functions.py --execute
```

This will parse the *Python* file, generate the corresponding *Fortran* file, compile it and execute it. The result is:

```
4.0000000000000000
8.0000000000000000
```

Now, let us take a look at the *Fortran* file

```
program main

implicit none
real(kind=8) :: y1   = 2.00000000000000
real(kind=8) :: x1   = 1.00000000000000
real(kind=8) :: z
real(kind=8) :: t
real(kind=8) :: w

!
x1 = 1.0d0
y1 = 2.0d0
w = 2*f(x1, y1) + 1.0d0
call g (x1, w, z, t)
print * ,z
print * ,t

contains
! ........................................
real(kind=8) function f(u, v)  result(t)
implicit none
real(kind=8), intent(in)  :: u
real(kind=8), intent(in)  :: v

t = u - v

end function
! ........................................

! ........................................
subroutine g(x, v, t, z)
  implicit none
  real(kind=8), intent(out)  :: t
  real(kind=8), intent(out)  :: z
  real(kind=8), intent(in)  :: x
  real(kind=8), intent(in)  :: v
  real(kind=8) :: m

  m = -v + x
  t = 2.0d0*m
  z = 2.0d0*t

end subroutine
! ........................................
```

(continues on next page)

(continued from previous page)

```
end
```

Matrix multiplication using OpenMP

Todo: a new example without pragmas

Note: **Openmp** is activated using the flag **–openmp** in the command line.

The following plot shows the scalability of the generated code on **LRZ** using $(n, m, p) = (5000, 7000, 5000)$.

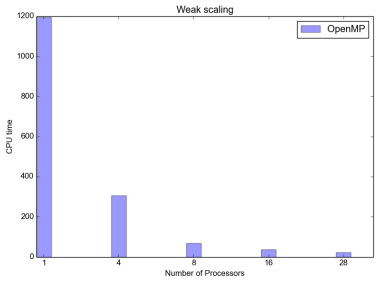

Fig. 1: Weak scalability on LRZ. CPU time is given in seconds.

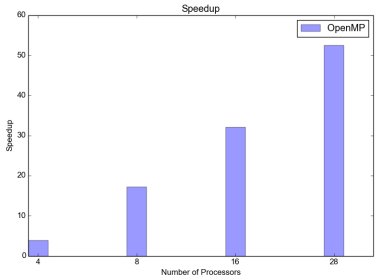

Fig. 2: Speedup on LRZ

Poisson solver using MPI

Todo: add an example

1.1.3 More topics to be covered

- *Pyccel extensions*:
  - *Math support in Pyccel*,

  - *numpy*,
  - *scipy*,
  - *mpi4py*,
  - *h5py*,
  - ...
- *Pyccel compiler*:
  - *Working with projects*,
  - *Rules*,

1.2 The Python Language

In this chapter, we describe the Python syntax and interpretor behavior that we would like to cover. Since more details about the Python language can be found on the web, we will only specify how we would like to use Python having in mind the final Fortran code.

1.2.1 Datatypes

Native types

Python	Fortran
int	int
float	real(4)
float64	real(8)
complex	complex(16)
tuple	static array
list	dynamic array

- Default type for floating numbers is **double precision**. Hence, the following stamtement

```
x = 1.
```

will be converted to

```
real(8) :: x

x = 1.0d0
```

Slicing

- When assigning a slice of **tuple**, we must allocate memory before (tuples are considered as static arrays). Therefor, the following python code

```
a = (1, 4, 9, 16)
c = a[1:]
```

will be converted to

```
integer :: a (0:3)
integer, allocatable :: c(:)

a = (/ 1, 4, 9, 16 /)
c = allocate(c(1,3))
c = a(1 : )
```

Todo: memory allocation within the scope of definition

Dynamic *vs* Static typing

Since our aim is to generate code in a low-level language, which is in most cases of static typed, we will have to devise an alternative way to construct/find the appropriate type of a given variable. This can be done by including the concept of *constructors* or use specific *headers* to assist *Pyccel* in finding/infering the appropriate type.

Let's explain this more precisely; we consider the following code

```
n = 5
x = 2.0 * n
```

In this example, **n** will be interprated as an **integer** while **x** will be a **double** number, so everything is fine.

The problem arises when using a function, like in the following example

```
def f(n):
  x = 2.0 * n
  return x

n = 5
x = f(n)
```

Now the question is what would be the signature of **f** if there was no call to it in the previous script?

To overcome this ambiguity, we rewrite our function as

```
#$ header f(int)
def f(n):
  x = 2.0 * n
  return x
```

Such an implementation still makes sens inside *Python*. As you can see, the type of *x* is infered by analysing our *expressions*.

1.2.2 Python Restrictions

Native *Python* objects are **implicitly** typed. This means that the following instructions are valid since the assigned *rhs* has a *static* type.

```
x = 1                    # OK
y = 1.                   # OK
s = 'hello'              # OK
z = [1, 4, 9]            # OK
t = (1., 4., 9., 16.0)   # OK
```

Concerning *lists* and *tuples*, all their elements must be of the same type.

```
z = [1, 4, 'a']          # KO
t = (1., 4., 9., [])     # KO
```

1.2.3 Operators and Expressions

1.2.4 Control Flow

1.2.5 Functions

Unlike *c/c++*, *Python* functions do not have separate header files or interface/implementation sections like *Pascal/Fortran*.

A function is simply defined using:

```
def f(u,v):
    t = u - v
    return t
```

As we are targeting a *strongly typed* language, unfortunately, the first thing to do is to add a *header* as the following:

```
#$ header f(double, double) results(double)
def f(u,v):
    t = u - v
    return t
```

this tells *Pyccel* that the input/output arguments are of *double precision* type.

You can then call **f** even in a given expression:

```
x1 = 1.0
y1 = 2.0

w    = 2 * f(x1,y1) + 1.0
```

You can also define functions with multiple *lhs* and call them as in the following example:

```
#$ header g(double, double)
def g(x,v):
    m = x - v
    t =  2.0 * m
    z =  2.0 * t
    return t, z

x1 = 1.0
y1 = 2.0

z, t = g(x1,y1)
```

1.2.6 Modules

1.2.7 Oriented Object Programming

Let's take this example; we consider the following code

```
from pyccel.ast.core import Variable, Assign
from pyccel.ast.core import ClassDef, FunctionDef, Module
from pyccel import fcode
x = Variable('double', 'x')
y = Variable('double', 'y')
z = Variable('double', 'z')
t = Variable('double', 't')
a = Variable('double', 'a')
b = Variable('double', 'b')
body = [Assign(y,x+a)]
translate = FunctionDef('translate', [x,y,a,b], [z,t], body)
attributs   = [x,y]
methods     = [translate]
Point = ClassDef('Point', attributs, methods)
incr = FunctionDef('incr', [x], [y], [Assign(y,x+1)])
decr = FunctionDef('decr', [x], [y], [Assign(y,x-1)])
module=Module('my_module', [], [incr, decr], [Point])
code=fcode(module)
print(code)
```

In this example, we created a Class *Point* that represent a point in 2d with two functions *incr* and *decr* The results in Fortran looks like

```
module mod_my_module

implicit none

type, public :: Point
  real(kind=8) :: x
  real(kind=8) :: y
  contains
  procedure :: translate => Point_translate

end type Point
contains

! ........................................
real(kind=8) function incr(x)  result(y)
implicit none
real(kind=8), intent(in)  :: x

y = 1 + x

end function
! ........................................

! ........................................
real(kind=8) function decr(x)  result(y)
implicit none
real(kind=8), intent(in)  :: x

y = -1 + x

end function
! ........................................
```

(continues on next page)

(continued from previous page)

```
! ........................................
subroutine translate(x, y, a, b, z, t)
implicit none
real(kind=8), intent(in)  :: a
real(kind=8), intent(in)  :: b
real(kind=8), intent(out)  :: t
real(kind=8), intent(inout)  :: y
real(kind=8), intent(in)  :: x
real(kind=8), intent(out)  :: z

y = a + x

end subroutine
! ........................................

end module
```

Notice that in Fortran the class must be in Module that's why the class and the functions where put in a module in the Python code.

1.2.8 Code Documention

We recommand the use of Google or Numpy documentation styles.

1.2.9 Working with Legacy code

1.2.10 Input and Output

1.2.11 Functional Programming

1.3 The Python Standard Library

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

1.3.1 Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	*divmod()*	*input()*	*open()*	*staticmethod()*
all()	*enumerate()*	*int()*	*ord()*	*str()*
any()	*eval()*	*isinstance()*	*pow()*	*sum()*
basestring()	*execfile()*	*issubclass()*	*print()*	*super()*
bin()	*file()*	*iter()*	*property()*	*tuple()*
bool()	*filter()*	*len()*	*range()*	*type()*
bytearray()	*float()*	*list()*	*raw_input()*	*unichr()*
callable()	*format()*	*locals()*	*reduce()*	*unicode()*
chr()	*frozenset()*	*long()*	*reload()*	*vars()*
classmethod()	*getattr()*	*map()*	*repr()*	*xrange()*
cmp()	*globals()*	*max()*	*reversed()*	*zip()*
compile()	*hasattr()*	*memoryview()*	*round()*	*__import__()*
complex()	*hash()*	*min()*	*set()*	
delattr()	*help()*	*next()*	*setattr()*	
dict()	*hex()*	*object()*	*slice()*	
dir()	*id()*	*oct()*	*sorted()*	

Some of these functions like *abs* are covered in the *Pyccel beta* version, while others like *all* will be covered in the *Pyccel lambda* version. Finally, there are also some functions that are under the *Pyccel restriction* and will not be covered.

abs (*x*)
Pyccel alpha, Python documentation for abs

all (*x*)
Pyccel lambda, Python documentation for all

any (*x*)
Pyccel lambda, Python documentation for any

basestring (*x*)
Pyccel restriction, Python documentation for basestring

bin (*x*)
Pyccel restriction, Python documentation for bin

bool (*x*)
Pyccel alpha, Python documentation for bool

bytearray (*x*)
Pyccel restriction, Python documentation for bytearray

callable (*object*)
Pyccel lambda, *Pyccel omicron*, Python documentation for callable

chr (*i*)
Pyccel alpha, Python documentation for chr

classmethod (*function*)
Pyccel omicron, Python documentation for classmethod

cmp (*x*, *y*)
Pyccel beta, Python documentation for cmp

compile (*source*, *filename*, *mode*[, *flags*[, *dont_inherit*]])
Pyccel beta, Python documentation for compile

class complex([*real*[, *imag*]])
Pyccel alpha, Python documentation for class

delattr(*object*, *name*)
Pyccel restriction, Python documentation for delattr

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
Pyccel restriction, Python documentation for dict

divmod(*a*, *b*)
Pyccel alpha, Python documentation for divmod

enumerate(*sequence*, *start=0*)
Pyccel lambda, Python documentation for enumerate

eval(*expression*[, *globals*[, *locals*]])
Pyccel beta, *Pyccel lambda*, Python documentation for eval

execfile(*filename*[, *globals*[, *locals*]])
Pyccel beta, Python documentation for execfile

file(*name*[, *mode*[, *buffering*]])
Pyccel restriction, Python documentation for file

filter(*function*, *iterable*)
Pyccel lambda, Python documentation for filter

class float([*x*])
Pyccel alpha, Python documentation for float

format(*value*[, *format_spec*])
Pyccel beta, Python documentation for format

class frozenset([*iterable*])
:noindex:
Pyccel restriction, Python documentation for frozenset

getattr(*object*, *name*[, *default*])
Pyccel restriction, Python documentation for file

globals()
Pyccel restriction, Python documentation for globals

hasattr(*object*, *name*)
Pyccel restriction, Python documentation for hasattr

hash(*object*)
Pyccel restriction, Python documentation for hash

help([*object*])
Pyccel restriction, Python documentation for help

hex(*x*)
Pyccel restriction, Python documentation for hex

id(*object*)
Pyccel beta, Python documentation for id

input([*prompt*])
Pyccel beta, Python documentation for input

class int(*x=0*)

class int(*x*, *base=10*)
Pyccel alpha, Python documentation for int

isinstance(*object*, *classinfo*)
Pyccel omicron, Python documentation for isinstance

issubclass(*class*, *classinfo*)
Pyccel restriction, Python documentation for issubclass

iter(*o*[, *sentinel*])
Pyccel lambda, Python documentation for iter

len(*s*)
Pyccel alpha, Python documentation for len

class list([*iterable*])
:noindex:
Pyccel alpha, *Pyccel lambda*, Python documentation for list

class long(*x=0*)
class long(*x*, *base=10*)
Pyccel beta, Python documentation for long

locals()
Pyccel restriction, Python documentation for locals

map(*function*, *iterable*, ...)
Pyccel lambda, Python documentation for map

max(*iterable*[, *key*])
max(*arg1*, *arg2*, **args*[, *key*])
Pyccel alpha, *Pyccel lambda*, Python documentation for max

memoryview(*obj*)
:noindex:
Pyccel restriction, Python documentation for memoryview

min(*iterable*[, *key*])
min(*arg1*, *arg2*, **args*[, *key*])
Pyccel alpha, *Pyccel lambda*, Python documentation for min

next(*iterator*[, *default*])
Pyccel lambda, Python documentation for next

class object
Pyccel beta, *Pyccel omicron*, Python documentation for object

oct(*x*)
Pyccel restriction, Python documentation for oct

open(*name*[, *mode*[, *buffering*]])
Pyccel beta, Python documentation for open

ord(*c*)
Pyccel restriction, Python documentation for ord

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*)
Pyccel alpha, Python documentation for print

pow(*x*, *y*[, *z*])
Pyccel alpha, Python documentation for pow

class property([*fget*[, *fset*[, *fdel*[, *doc*]]]])
Pyccel omicron, Python documentation for property

range(*stop*)
range(*start*, *stop*[, *step*])
Pyccel alpha, Python documentation for range

raw_input([*prompt*])
Pyccel beta, Python documentation for raw_input

reduce(*function*, *iterable*[, *initializer*])
Pyccel lambda, Python documentation for reduce

reload(*module*)
Pyccel restriction, Python documentation for reload

repr(*object*)
Pyccel beta, Python documentation for repr

reversed(*seq*)
Pyccel lambda, Python documentation for reversed

round(*number*[, *ndigits*])
Pyccel alpha, Python documentation for round

class set([*iterable*])
:noindex:
Pyccel lambda, Python documentation for func-set

setattr(*object*, *name*, *value*)
Pyccel restriction, Python documentation for setattr

class slice(*stop*)
class slice(*start*, *stop*[, *step*])

Pyccel alpha, Python documentation for slice

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])
Pyccel lambda, Python documentation for sorted

staticmethod(*function*)
Pyccel omicron, Python documentation for staticmethod

class str(*object=''*)
Pyccel beta, Python documentation for str

sum(*iterable*[, *start*])
Pyccel lambda, Python documentation for sum

super(*type*[, *object-or-type*])
Pyccel omicron, Python documentation for super

tuple([*iterable*])
Pyccel alpha, *Pyccel lambda*, Python documentation for tuple

class type(*object*)
class type(*name*, *bases*, *dict*)

Pyccel omicron, Python documentation for type

unichr(*i*)
Pyccel restriction, Python documentation for unichr

unicode(*object=''*)
unicode(*object*[, *encoding*[, *errors*]])
Pyccel restriction, Python documentation for unicode

vars([*object*])
Pyccel restriction, Python documentation for vars

xrange(*stop*)
xrange(*start*, *stop*[, *step*])
Pyccel alpha, Python documentation for xrange

zip([*iterable*, ...])
Pyccel lambda, Python documentation for zip

__import__(*name*[, *globals*[, *locals*[, *fromlist*[, *level*]]]])
Pyccel restriction, Python documentation for __import__

1.3.2 Non-essential Built-in Functions

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

apply(*function*, *args*[, *keywords*])
Pyccel restriction, Python documentation for apply

buffer(*object*[, *offset*[, *size*]])
Pyccel restriction, Python documentation for buffer

coerce(*x*, *y*)
Pyccel restriction, Python documentation for coercive

intern(*string*)
Pyccel restriction, Python documentation for intern

1.3.3 Built-in Constants

A small number of constants live in the built-in namespace. They are:

False
Pyccel alpha,

True
Pyccel alpha,

None
The sole value of `types.NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function.

NotImplemented
Pyccel restriction,

Ellipsis
Pyccel restriction,

__debug__
Pyccel alpha,

1.3.4 Magic methods

More details can be found here

		Magic methods		
__abs__()	*__ge__()*	*__itruediv__()*	*__reversed__()*	*__trunc__()*
__add__()	*__get__()*	*__ixor__()*	*__rfloordiv__()*	*__xor__()*
__and__()	*__getattr__()*	*__instancecheck__()*	*__rlshift__()*	
__bool__()	*__getattribute__()*	*__len__()*	*__rmod__()*	
__bytes__()	*__getitem__()*	*__lshift__()*	*__rmul__()*	
__call__()	*__getstate__()*	*__lt__()*	*__ror__()*	
__ceil__()	*__gt__()*	*__le__()*	*__round__()*	
__complex__()	*__hash__()*	*__mod__()*	*__rpow__()*	
__contains__()	*__iadd__()*	*__missing__()*	*__rrshift__()*	
__copy__()	*__iand__()*	*__mul__()*	*__rshift__()*	
__deepcopy__()	*__idivmod__()*	*__ne__()*	*__rsub__()*	
__del__()	*__ifloordiv__()*	*__neg__()*	*__rtruediv__()*	
__delattr__()	*__ilshift__()*	*__next__()*	*__rxor__()*	
__delitem__()	*__imod__()*	*__new__()*	*__set__()*	
__dir__()	*__imul__()*	*__or__()*	*__setattr__()*	
__divmod__()	*__index__()*	*__pos__()*	*__setitem__()*	
__eq__()	*__int__()*	*__pow__()*	*__setstate__()*	
__enter__()	*__invert__()*	*__radd__()*	*__slots__()*	
__exit__()	*__ior__()*	*__rand__()*	*__str__()*	
__format__()	*__ipow__()*	*__rdivmod__()*	*__sub__()*	
__floordiv__()	*__irshift__()*	*__reduce__()*	*__subclasscheck__()*	
__float__()	*__isub__()*	*__reduce_ex__()*	*__subclasshook__()*	
__floor__()	*__iter__()*	*__repr__()*	*__truediv__()*	

Basics

__init__(*x*)
Pyccel omicron,

__repr__()
Pyccel omicron, *Pyccel beta*,

__str__()
Pyccel omicron, *Pyccel beta*,

__bytes__()
Pyccel restriction,

__format__()
Pyccel omicron, *Pyccel beta*,

Classes That Act Like Iterators

__iter__()
Pyccel omicron, *Pyccel lambda*,

__next__()
Pyccel omicron, *Pyccel lambda*,

__reversed__()
Pyccel omicron, *Pyccel lambda*,

Computed Attributes

__getattribute__()
Pyccel restriction,

__getattr__()
Pyccel restriction,

__setattr__()
Pyccel restriction,

__delattr__()
Pyccel restriction,

__dir__()
Pyccel restriction,

Classes That Act Like Functions

__call__()
Pyccel omicron,

Classes That Act Like Sets

__len__()
Pyccel beta, *Pyccel lambda*,

__contains__()
Pyccel restriction,

Classes That Act Like Dictionaries

__getitem__()
Pyccel restriction,

__setitem__()
Pyccel restriction,

__delitem__()
Pyccel restriction,

__missing__()
Pyccel restriction,

Classes That Act Like Numbers

__add__()
Pyccel omicron,

__sub__()
Pyccel omicron,

__mul__()
Pyccel omicron,

__truediv__()
Pyccel omicron,

__floordiv__()
Pyccel omicron,

__mod__()
Pyccel omicron,

__divmod__()
Pyccel omicron,

__pow__()
Pyccel omicron,

__lshift__()
Pyccel omicron,

__rshift__()
Pyccel omicron,

__and__()
Pyccel omicron,

__xor__()
Pyccel omicron,

__or__()
Pyccel omicron,

__radd__()
Pyccel omicron,

__rsub__()
Pyccel omicron,

__rmul__()
Pyccel omicron,

__rtruediv__()
Pyccel omicron,

__rfloordiv__()
Pyccel omicron,

__rmod__()
Pyccel omicron,

__rdivmod__()
Pyccel omicron,

__rpow__()
Pyccel omicron,

__rlshift__()
Pyccel omicron,

__rrshift__()
Pyccel omicron,

__rand__()
Pyccel omicron,

__rxor__()
Pyccel omicron,

__ror__()

__iadd__()
Pyccel omicron,

__isub__()
Pyccel omicron,

__imul__()
Pyccel omicron,

__itruediv__()
Pyccel omicron,

__ifloordiv__()
Pyccel omicron,

__imod__()
Pyccel omicron,

__idivmod__()
Pyccel omicron,

__ipow__()
Pyccel omicron,

__ilshift__()
Pyccel omicron,

__irshift__()
Pyccel omicron,

__iand__()
Pyccel omicron,

__ixor__()
Pyccel omicron,

__ior__()
Pyccel omicron,

__neg__()
Pyccel omicron,

__pos__()
Pyccel omicron,

__abs__()
Pyccel omicron,

__invert__()
Pyccel omicron,

__complex__()
Pyccel omicron,

__int__()
Pyccel omicron,

__float__()
Pyccel omicron,

__round__()
Pyccel omicron,

__ceil__()
Pyccel omicron,

__floor__()
Pyccel omicron,

__trunc__()
Pyccel omicron,

__index__()
Pyccel omicron,

Classes That Can Be Compared

__eq__()
Pyccel omicron, *Pyccel beta*,

__ne__()
Pyccel omicron, *Pyccel beta*,

__lt__()
Pyccel omicron, *Pyccel beta*,

__le__()
Pyccel omicron, *Pyccel beta*,

__gt__()
Pyccel omicron, *Pyccel beta*,

__ge__()
Pyccel omicron, *Pyccel beta*,

__bool__()
Pyccel omicron, *Pyccel beta*,

Classes That Can Be Serialized

__copy__()
Pyccel beta,

__deepcopy__()
Pyccel beta,

__getstate__()
Pyccel restriction,

__reduce__()
Pyccel omicron, *Pyccel lambda*,

__reduce_ex__()
Pyccel omicron, *Pyccel lambda*,

__setstate__()
Pyccel restriction,

Classes That Can Be Used in a with Block

__enter__()
Pyccel omicron, *Pyccel lambda*,

__exit__()
Pyccel omicron, *Pyccel lambda*,

Others

__new__()
Pyccel restriction,

__del__()
Pyccel omicron,

__slots__()
Pyccel restriction,

__hash__()
Pyccel restriction,

__get__()
Pyccel beta,

__set__()
Pyccel beta,

__subclasscheck__()
Pyccel restriction,

__subclasshook__()
Pyccel restriction,

__instancecheck__()
Pyccel restriction,

1.4 The Pyccel Extensions

These extensions are built in and can be activated by respective entries in the `pyccelext` configuration value.

MPI

The High level support for *MPI* will follow the scipy mpi interface using **mpi4py**.

Lapack

The High level support for *Lapack* will follow the scipy lapack interface.

HDF5

The High level support for *HDF5* will follow the h5py package.

FFT

The High level support for *FFT* will follow the **'scipy fft'_** interface.

Itertools

Following module itertools from the Python3 standard library: Functions creating iterators for efficient looping.

1.4.1 High level interfaces

Math support in Pyccel

h5py

TODO

mpi4py

TODO

numpy

Array manipulation

- copyto

Changing array shape

- reshape
- ravel
- ndarray.flatten

Transpose-like operations

- swapaxes
- ndarray.T
- transpose

Joining arrays

- concatenate
- stack
- column_stack
- block

Splitting arrays

- split
- array_split

Tiling arrays

- tile
- repeat

Adding and removing elements

- delete
- insert
- append
- resize
- trim_zeros
- unique

Rearranging elements

- flip
- fliplr
- flipud
- reshape
- roll
- rot90

The N-dimensional array

TODO

Array creation routines

Ones and zeros

- empty
- empty_like
- eye
- identity

- ones
- ones_like
- zeros
- zeros_like

From existing data

- array
- asarray
- asanyarray
- ascontiguousarray
- asmatrix
- copy
- fromfile
- fromfunction
- loadtxt

Numerical ranges

- arange
- linspace
- logspace
- meshgrid

Building matrices

- diag
- diagflat
- tri
- tril
- triu

Linear algebra (numpy.linalg)

Matrix and vector products

- dot
- vdot
- inner

- outer
- matmul
- tensordot
- linalg.matrix_power
- kron

Decompositions

- linalg.cholesky
- linalg.qr
- linalg.svd

Matrix eigenvalues

- linalg.eig
- linalg.eigh
- linalg.eigvals
- linalg.eigvalsh

Norms and other numbers

- linalg.norm
- linalg.cond
- linalg.det
- linalg.matrix_rank
- trace

Solving equations and inverting matrices

- linalg.solve
- linalg.tensorsolve
- linalg.lstsq
- linalg.inv
- linalg.pinv
- linalg.tensorinv

Mathematical functions

Trigonometric functions

- sin
- cos
- tan
- arcsin
- arccos
- arctan
- hypot
- arctan2
- degrees
- radians
- unwrap
- deg2rad
- rad2deg

Hyperbolic functions

- sinh
- cosh
- tanh
- arcsinh
- arccosh
- arctanh

Rounding

- around
- round
- floor
- ceil

Sums, products, differences

- prod
- sum
- cumprod

- cumsum
- diff
- ediff1d
- gradient
- cross

Exponents and logarithms

- exp
- log

Other special functions

- i0
- sinc

Arithmetic operations

- add
- multiply
- divide
- power
- subtract
- true_divide
- floor_divide
- float_power
- fmod
- mod
- remainder
- divmod

Handling complex numbers

- angle
- real
- imag
- conj

Miscellaneous

- convolve
- sqrt
- square
- absolute
- fabs
- sign
- maximum
- minimum
- fmax
- fmin
- interp

scipy

TODO

itertools

- product(A,B)

OpenACC

Following the same idea for **OpenMP**, there are two levels to work with **OpenACC**, called *level-0* and *level-1*.

level-1

This is a high level that enables the use of *OpenACC* through simple instructions.

class pyccel.stdlib.parallel.openacc.**Range**(*start*, *stop*, *step*, *collapse=None*, *gang=None*, *worker=None*, *vector=None*, *seq=None*, *auto=None*, *tile=None*, *device_type=None*, *independent=None*, *private=None*, *reduction=None*)

__init__(*start*, *stop*, *step*, *collapse=None*, *gang=None*, *worker=None*, *vector=None*, *seq=None*, *auto=None*, *tile=None*, *device_type=None*, *independent=None*, *private=None*, *reduction=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__

list of weak references to the object (if defined)

class pyccel.stdlib.parallel.openacc.**Parallel**(*Async=None*, *wait=None*, *num_gangs=None*, *num_workers=None*, *vector_length=None*, *device_type=None*, *If=None*, *reduction=None*, *copy=None*, *copyin=None*, *copyout=None*, *create=None*, *present=None*, *deviceptr=None*, *private=None*, *firstprivate=None*, *default=None*)

__init__(*Async=None*, *wait=None*, *num_gangs=None*, *num_workers=None*, *vector_length=None*, *device_type=None*, *If=None*, *reduction=None*, *copy=None*, *copyin=None*, *copyout=None*, *create=None*, *present=None*, *deviceptr=None*, *private=None*, *firstprivate=None*, *default=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__

list of weak references to the object (if defined)

Example: Hello world

See `script`.

Example: reduction

See `script`.

OpenMP

There are two levels to work with **OpenMP**, called *level-0* and *level-1*.

level-1

This is a high level that enables the use of *OpenMP* through simple instructions.

class pyccel.stdlib.parallel.openmp.**Range**(*start*, *stop*, *step*, *nowait=None*, *collapse=None*, *private=None*, *firstprivate=None*, *lastprivate=None*, *reduction=None*, *schedule=None*, *ordered=None*, *linear=None*)

__init__(*start*, *stop*, *step*, *nowait=None*, *collapse=None*, *private=None*, *firstprivate=None*, *lastprivate=None*, *reduction=None*, *schedule=None*, *ordered=None*, *linear=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__

list of weak references to the object (if defined)

class pyccel.stdlib.parallel.openmp.**Parallel**(*num_threads=None*, *if_test=None*, *private=None*, *firstprivate=None*, *shared=None*, *reduction=None*, *default=None*, *copyin=None*, *proc_bind=None*)

__init__ (*num_threads=None*, *if_test=None*, *private=None*, *firstprivate=None*, *shared=None*, *reduction=None*, *default=None*, *copyin=None*, *proc_bind=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

Example: Hello world

See `script`.

Example: matrix multiplication

See `script`.

Task Based Parallelism

Todo: implement its grammar and syntax for Task Based Parallelism

1.4.2 Low level interfaces

Blas-Lapack

TODO

fftw

TODO

mpi

Enabling **MPI** is done in two steps:

- you need to have the following import:

```
from pyccel.mpi import *
```

- you need to compile your file with a valid **mpi** compiler:

```
pyccel --language="fortran" --compiler=mpif90 --filename=tests/examples/mpi/ex_0.
↪py
mpirun -n 2 tests/examples/mpi/ex_0
```

Let's start with a simple example (**tests/examples/mpi/ex_0.py**):

```
from pyccel.mpi import *

ierr = mpi_init()

comm = mpi_comm_world
size = comm.size
rank = comm.rank

print ('I process ', rank, ', among ', size, ' processes')

ierr = mpi_finalize()
```

we compile the file using:

```
pyccel --language="fortran" --compiler=mpif90 --filename=tests/examples/mpi/ex_0.py
```

The generated *Fortran* code is

```
program main
use MPI
implicit none
integer, dimension(MPI_STATUS_SIZE) :: i_mpi_status
integer :: ierr
integer :: comm
integer :: rank
integer :: i_mpi_error
integer :: size

!
call mpi_init(ierr)
comm = MPI_comm_world
call mpi_comm_size (comm, size, i_mpi_error)
call mpi_comm_rank (comm, rank, i_mpi_error)
print * ,'I process ',rank,', among ',size,' processes'
call mpi_finalize(ierr)

end
```

now let's run the executable:

```
mpirun -n 4 tests/examples/mpi/ex_0
```

the result is:

```
I process            1 , among            4  processes
I process            2 , among            4  processes
I process            3 , among            4  processes
I process            0 , among            4  processes
```

Note that **comm** is considered as an object in our python file. A communicator has the following attributs:

- **size** : total number of processes within the communicator,
- **rank** : rank of the current process.

A communicator has also (many of) **MPI** procedures, defined as **methods**. The following example shows how to use the **send** and **recv** actions with respect to a given communicator.

(listing of **tests/examples/mpi/ex_1.py**)

```
# coding: utf-8

from pyccel.mpi import *

ierr = mpi_init()

comm = mpi_comm_world
size = comm.size
rank = comm.rank

n = 4
x = zeros(n, double)
y = zeros((3,2), double)

if rank == 0:
    x = 1.0
    y = 1.0

source = 0
dest   = 1
tagx = 1234
if rank == source:
    ierr = comm.send(x, dest, tagx)
    print("processor ", rank, " sent ", x)

if rank == dest:
    ierr = comm.recv(x, source, tagx)
    print("processor ", rank, " got  ", x)

tag1 = 5678
if rank == source:
    x[1] = 2.0
    ierr = comm.send(x[1], dest, tag1)
    print("processor ", rank, " sent x(1) = ", x[1])

if rank == dest:
    ierr = comm.recv(x[1], source, tag1)
    print("processor ", rank, " got  x(1) = ", x[1])


tagx = 4321
if rank == source:
    ierr = comm.send(y, dest, tagx)
    print("processor ", rank, " sent ", y)

if rank == dest:
    ierr = comm.recv(y, source, tagx)
    print("processor ", rank, " got  ", y)

tag1 = 8765
if rank == source:
    y[1,1] = 2.0
    ierr = comm.send(y[1,1], dest, tag1)
    print("processor ", rank, " sent y(1,1) = ", y[1,1])

if rank == dest:
    ierr = comm.recv(y[1,1], source, tag1)
```

(continues on next page)

(continued from previous page)

```
    print("processor ", rank, " got  y(1,1) = ", y[1,1])

tag1 = 6587
if rank == source:
    y[1,:] = 2.0
    ierr = comm.send(y[1,:], dest, tag1)
    print("processor ", rank, " sent y(1,:) = ", y[1,:])

if rank == dest:
    ierr = comm.recv(y[1,:], source, tag1)
    print("processor ", rank, " got  y(1,:) = ", y[1,:])

ierr = mpi_finalize()
```

compile the file and execute it using:

```
pyccel --language="fortran" --compiler=mpif90 --filename=tests/examples/mpi/ex_1.py
mpirun -n 2 tests/examples/mpi/ex_1
```

the result is:

```
processor            0  sent    1.0000000000000000        1.0000000000000000        1.
↪0000000000000000        1.0000000000000000
processor            0  sent x(1) =    2.0000000000000000
processor            0  sent    1.0000000000000000        1.0000000000000000        1.
↪0000000000000000        1.0000000000000000        1.0000000000000000        1.
↪0000000000000000
processor            0  sent y(1,1) =    2.0000000000000000
processor            1  got     1.0000000000000000        1.0000000000000000        1.
↪0000000000000000        1.0000000000000000
processor            1  got  x(1) =    2.0000000000000000
processor            1  got     1.0000000000000000        1.0000000000000000        1.
↪0000000000000000        1.0000000000000000        1.0000000000000000        1.
↪0000000000000000
processor            1  got  y(1,1) =    2.0000000000000000
processor            0  sent y(1,:) =    2.0000000000000000        2.0000000000000000
processor            1  got  y(1,:) =    2.0000000000000000        2.0000000000000000
```

other examples can be found in **tests/examples/mpi**.

OpenACC

This allows to write valid *OpenACC* instructions and are handled in two steps:

- in the grammar, in order to parse the *acc* pragams
- as a Pyccel header. Therefor, you can import and call *OpenACC* functions as you would do it in *Fortran* or *C*.

Examples

See `script`.

Now, run the command:

```
pyccel tests/scripts/openacc/core/ex1.py --compiler=pgfortran --openacc
```

Executing the associated binary gives:

```
number of available OpenACC devices :            1
type of available OpenACC devices   :            2
```

See more OpenACC examples.

OpenMP

This allows to write valid *OpenMP* instructions and are handled in two steps:

- in the grammar, in order to parse the *omp* pragams
- as a Pyccel header. Therefor, you can import and call *OpenMP* functions as you would do it in *Fortran* or *C*.

Examples

See `script`.

Now, run the command:

```
pyccel tests/scripts/openmp/core/ex1.py --openmp
export OMP_NUM_THREADS=4
```

Executing the associated binary gives:

```
> threads number :            1
> maximum available threads :            4
> thread  id :            0
> thread  id :            3
> thread  id :            1
> thread  id :            2
```

See more OpenMP examples.

Matrix multiplication

Let's take a look at the file *tests/examples/openmp/matrix_product.py*, listed below

```
from numpy import zeros

n = 500
m = 700
p = 500

a = zeros((n,m), double)
b = zeros((m,p), double)
c = zeros((n,p), double)

#$ omp parallel
#$ omp do schedule(runtime)
for i in range(0, n):
    for j in range(0, m):
        a[i,j] = i-j
#$ omp end do nowait
```

(continues on next page)

(continued from previous page)

```
#$ omp do schedule(runtime)
for i in range(0, m):
    for j in range(0, p):
        b[i,j] = i+j
#$ omp end do nowait

#$ omp do schedule(runtime)
for i in range(0, n):
    for j in range(0, p):
        for k in range(0, p):
            c[i,j] = c[i,j] + a[i,k]*b[k,j]
#$ omp end do
#$ omp end parallel
```

Now, run the command:

```
pyccel tests/examples/openmp/matrix_product.py --compiler="gfortran" --openmp
```

This will parse the *Python* file, generate the corresponding *Fortran* file and compile it.

Note: **Openmp** is activated using the flag **–openmp** in the command line.

The generated *Fortran* code is

```
program main
use omp_lib
implicit none
real(kind=8), allocatable :: a (:, :)
real(kind=8), allocatable :: c (:, :)
real(kind=8), allocatable :: b (:, :)
integer :: i
integer :: k
integer :: j
integer :: m
integer :: n
integer :: p

!
n = 500
m = 700
p = 500
allocate(a(0:n-1, 0:m-1)) ; a = 0
allocate(b(0:m-1, 0:p-1)) ; b = 0
allocate(c(0:n-1, 0:p-1)) ; c = 0
!$omp parallel
!$omp do schedule(runtime)
do i = 0, n - 1, 1
  do j = 0, m - 1, 1
    a(i, j) = i - j
  end do
end do
!$omp end do  nowait
!$omp do schedule(runtime)
do i = 0, m - 1, 1
```

(continues on next page)

(continued from previous page)

```
  do j = 0, p - 1, 1
    b(i, j) = i + j
  end do
end do
!$omp end do  nowait
!$omp do schedule(runtime)
do i = 0, n - 1, 1
  do j = 0, p - 1, 1
    do k = 0, p - 1, 1
      c(i, j) = a(i, k)*b(k, j) + c(i, j)
    end do
  end do
end do
!$omp end do
!$omp end parallel

end
```

1.4.3 Specifications

OpenACC

Todo: add OpenACC specifications and implement its grammar and syntax

Grammar

In this section, we give the **BNF** used grammar for parsing *openacc*.

```
// The following grammar is compatible with OpenACC 2.5

// TODO: - int-expr when needed (see specs)
//       - use boolean condition for If

Openacc:
  statements*=OpenaccStmt
;

OpenaccStmt: '#$' 'acc' stmt=AccConstructOrDirective;

////////////////////////////////////////////////////
//         Constructs and Directives
////////////////////////////////////////////////////
AccConstructOrDirective:
    AccParallelConstruct
  | AccKernelsConstruct
  | AccDataConstruct
  | AccEnterDataDirective
  | AccExitDataDirective
  | AccHostDataDirective
  | AccLoopConstruct
  | AccAtomicConstruct
```

(continues on next page)

(continued from previous page)

```
  | AccDeclareDirective
  | AccInitDirective
  | AccShutDownDirective
  | AccSetDirective
  | AccUpdateDirective
  | AccRoutineDirective
  | AccWaitDirective
  | AccEndClause
;
//////////////////////////////////////////////////

//////////////////////////////////////////////////
//     Constructs and Directives definitions
//////////////////////////////////////////////////
AccAtomicConstruct:    'atomic'       clauses*=AccAtomicClause;
AccDataConstruct:      'data'         clauses*=AccDataClause;
AccDeclareDirective:   'declare'      clauses*=AccDeclareClause;
AccEnterDataDirective: 'enter' 'data' clauses*=AccEnterDataClause;
AccExitDataDirective:  'exit'  'data' clauses*=AccExitDataClause;
AccHostDataDirective:  'host_data'    clauses*=AccHostDataClause;
AccInitDirective:      'init'         clauses*=AccInitClause;
AccKernelsConstruct:   'kernels'      clauses*=AccKernelsClause;
AccLoopConstruct:      'loop'         clauses*=AccLoopClause;
AccParallelConstruct:  'parallel'     clauses*=AccParallelClause;
AccRoutineDirective:   'routine'      clauses*=AccRoutineClause;
AccShutDownDirective:  'shutdown'     clauses*=AccShutDownClause;
AccSetDirective:       'set'          clauses*=AccSetClause;
AccUpdateDirective:    'update'       clauses*=AccUpdateClause;
AccWaitDirective:      'wait'         clauses*=AccWaitClause;
//////////////////////////////////////////////////

//////////////////////////////////////////////////
//      Clauses for Constructs and Directives
//////////////////////////////////////////////////
AccParallelClause:
    AccAsync
  | AccWait
  | AccNumGangs
  | AccNumWorkers
  | AccVectorLength
  | AccDeviceType
  | AccIf
  | AccReduction
  | AccCopy
  | AccCopyin
  | AccCopyout
  | AccCreate
  | AccPresent
  | AccDevicePtr
  | AccPrivate
  | AccFirstPrivate
  | AccDefault
;

AccKernelsClause:
    AccAsync
  | AccWait
```

(continues on next page)

(continued from previous page)

```
  | AccNumGangs
  | AccNumWorkers
  | AccVectorLength
  | AccDeviceType
  | AccIf
  | AccCopy
  | AccCopyin
  | AccCopyout
  | AccCreate
  | AccPresent
  | AccDevicePtr
  | AccDefault
;

AccDataClause:
    AccIf
  | AccCopy
  | AccCopyin
  | AccCopyout
  | AccCreate
  | AccPresent
  | AccDevicePtr
;

AccEnterDataClause:
    AccIf
  | AccAsync
  | AccWait
  | AccCopyin
  | AccCreate
;

AccExitDataClause:
    AccIf
  | AccAsync
  | AccWait
  | AccCopyout
  | AccDelete
  | AccFinalize
;

AccHostDataClause: AccUseDevice;

AccLoopClause:
    AccCollapse
  | AccGang
  | AccWorker
  | AccVector
  | AccSeq
  | AccAuto
  | AccTile
  | AccDeviceType
  | AccIndependent
  | AccPrivate
  | AccReduction
;
```

(continues on next page)

(continued from previous page)

```
AccAtomicClause: AccAtomicStatus;

AccDeclareClause:
    AccCopy
  | AccCopyin
  | AccCopyout
  | AccCreate
  | AccPresent
  | AccDevicePtr
  | AccDeviceResident
  | AccLink
;

AccInitClause:
    AccDeviceType
  | AccDeviceNum
;

AccShutDownClause:
    AccDeviceType
  | AccDeviceNum
;

AccSetClause:
    AccDefaultAsync
  | AccDeviceNum
  | AccDeviceType
;

AccUpdateClause:
    AccAsync
  | AccWait
  | AccDeviceType
  | AccIf
  | AccIfPresent
  | AccSelf
  | AccHost
  | AccDevice
;

AccRoutineClause:
    AccGang
  | AccWorker
  | AccVector
  | AccSeq
  | AccBind
  | AccDeviceType
  | AccNoHost
;

AccWaitClause: AccAsync;
////////////////////////////////////////////////////

////////////////////////////////////////////////////
//              Clauses definitions
////////////////////////////////////////////////////
AccAsync: 'async' '(' args+=ID[','] ')';
```

(continues on next page)

(continued from previous page)

```
AccAuto: 'auto';
AccBind: 'bind' '(' arg=STRING ')';
AccCache: 'cache' '(' args+=ID[','] ')';
AccCollapse: 'collapse' '(' n=INT ')';
AccCopy: 'copy' '(' args+=ID[','] ')';
AccCopyin: 'copyin' '(' args+=ID[','] ')';
AccCopyout: 'copyout' '(' args+=ID[','] ')';
AccCreate: 'create' '(' args+=ID[','] ')';
AccDefault: 'default' '(' status=AccDefaultStatus ')';
AccDefaultAsync: 'default_async' '(' args+=ID[','] ')';
AccDelete: 'delete' '(' args+=ID[','] ')';
AccDevice: 'device' '(' args+=ID[','] ')';
AccDeviceNum: 'device_num' '(' n=INT ')';
AccDevicePtr: 'deviceptr' '(' args+=ID[','] ')';
AccDeviceResident: 'device_resident' '(' args+=ID[','] ')';
AccDeviceType: 'device_type' '(' args+=ID[','] ')';
AccFirstPrivate: 'firstprivate' '(' args+=ID[','] ')';
AccFinalize: 'finalize';
AccGang: 'gang' '(' args+=AccGangArg[','] ')';
AccHost: 'host' '(' args+=ID[','] ')';
AccIf: 'if' cond=ID;
AccIfPresent: 'if_present';
AccIndependent: 'independent';
AccLink: 'link' '(' args+=ID[','] ')';
AccNoHost: 'nohost';
AccNumGangs: 'num_gangs' '(' n=INT ')';
AccNumWorkers: 'num_workers' '(' n=INT ')';
AccPresent: 'present' '(' args+=ID[','] ')';
AccPrivate: 'private' '(' args+=ID[','] ')';
AccReduction: 'reduction' '('op=AccReductionOperator ':' args+=ID[','] ')';
AccSeq: 'seq';
AccSelf: 'self' '(' args+=ID[','] ')';
AccTile: 'tile' '(' args+=ID[','] ')';
AccUseDevice: 'use_device' '(' args+=ID[','] ')';
AccVector: 'vector' ('(' args+=AccVectorArg ')')?;
AccVectorLength: 'vector_length' '(' n=INT ')';
AccWait: 'wait' '(' args+=ID[','] ')';
AccWorker: 'worker' ('(' args+=AccWorkerArg ')')?;
AccEndClause: 'end' construct=AccConstructs;
////////////////////////////////////////////////

////////////////////////////////////////////////
AccReductionOperator: ('+' | '-' | '*' | '/');
AccDefaultStatus: ('none' | 'present');
AccAtomicStatus: ('read' | 'write' | 'update' | 'capture');
AccWorkerArg: ('num' ':')? arg=INT ;
AccVectorArg: ('length' ':')? arg=INT ;
AccConstructs: ('parallel' | 'loop' | 'kernels');

AccGangArg: AccGangStaticArg | AccGangNumArg;
AccGangNumArg: ('num' ':')? arg=INT ;
AccGangStaticArg: 'static' ':' arg=INT ;

NotaStmt: /.*$/;
////////////////////////////////////////////////
```

See `script`.

OpenMP

We follow OpenMP 4.5 specifications.

Grammar

In this section, we give the **BNF** used grammar for parsing *openmp*.

```
// TODO: - linear:   improve using lists. see specs
//       - parallel: add if parallel

Openmp:
  statements*=OpenmpStmt
;

OpenmpStmt:
  '#$' 'omp' stmt=OmpConstructOrDirective
;

////////////////////////////////////////////////////
//         Constructs and Directives
////////////////////////////////////////////////////
OmpConstructOrDirective:
    OmpParallelConstruct
  | OmpLoopConstruct
  | OmpSingleConstruct
  | OmpEndClause
;
////////////////////////////////////////////////////

////////////////////////////////////////////////////
//     Constructs and Directives definitions
////////////////////////////////////////////////////
OmpParallelConstruct: 'parallel' clauses*=OmpParallelClause;
OmpLoopConstruct:     'do'       clauses*=OmpLoopClause;
OmpSingleConstruct:   'single'   clauses*=OmpSingleClause;
////////////////////////////////////////////////////

////////////////////////////////////////////////////
//      Clauses for Constructs and Directives
////////////////////////////////////////////////////
OmpParallelClause:
    OmpNumThread
  | OmpDefault
  | OmpPrivate
  | OmpShared
  | OmpFirstPrivate
  | OmpCopyin
  | OmpReduction
  | OmpProcBind
;

OmpLoopClause:
    OmpPrivate
  | OmpFirstPrivate
  | OmpLastPrivate
```

(continues on next page)

(continued from previous page)

```
  | OmpLinear
  | OmpReduction
  | OmpSchedule
  | OmpCollapse
  | OmpOrdered
;

OmpSingleClause:
    OmpPrivate
  | OmpFirstPrivate
;
////////////////////////////////////////////////

////////////////////////////////////////////////
//              Clauses definitions
////////////////////////////////////////////////
OmpNumThread: 'num_threads' '(' thread=ThreadIndex ')';
OmpDefault: 'default' '(' status=OmpDefaultStatus ')';
OmpProcBind: 'proc_bind' '(' status=OmpProcBindStatus ')';
OmpPrivate: 'private' '(' args+=ID[','] ')';
OmpShared: 'shared' '(' args+=ID[','] ')';
OmpFirstPrivate: 'firstprivate' '(' args+=ID[','] ')';
OmpLastPrivate: 'lastprivate' '(' args+=ID[','] ')';
OmpCopyin: 'copyin' '(' args+=ID[','] ')';
OmpReduction: 'reduction' '('op=OmpReductionOperator ':' args+=ID[','] ')';
OmpCollapse: 'collapse' '(' n=INT ')';
OmpLinear: 'linear' '(' val=ID ':' step=INT ')';
OmpOrdered: 'ordered' ('(' n=INT ')')?;
OmpSchedule: 'schedule' '(' kind=OmpScheduleKind (',' chunk_size=INT)? ')';
OmpEndClause: 'end' construct=OpenmpConstructs (simd='simd')? (nowait='nowait')?;
////////////////////////////////////////////////

////////////////////////////////////////////////
OmpScheduleKind: ('static' | 'dynamic' | 'guided' | 'auto' | 'runtime' );
OmpProcBindStatus: ('master' | 'close' | 'spread');
OmpReductionOperator: ('+' | '-' | '*' | '/');
OmpDefaultStatus: ('private' | 'firstprivate' | 'shared' | 'none');
OpenmpConstructs: ('single' | 'parallel' | 'do');

ThreadIndex: (ID | INT);
NotaStmt: /.*$/;
////////////////////////////////////////////////
```

See `script`.

Directives

OpenMP directives for Fortran are specified as follows:

```
sentinel directive-name [clause[ [,] clause]...]
```

The following sentinels are recognized in fixed form source files:

```
!$omp | c$omp | *$omp
```

Constructs

parallel

The syntax of the parallel construct is as follows:

```
!$omp parallel [clause[ [,] clause] ... ]
  structured-block
!$omp end parallel
```

where *clause* is one of the following:

```
if([parallel :] scalar-logical-expression)
num_threads(scalar-integer-expression)
default(private | firstprivate | shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(reduction-identifier : list)
proc_bind(master | close | spread)
```

The **end parallel** directive denotes the end of the **parallel** construct.

Todo: add restrictions (page 49)

Loop

The syntax of the loop construct is as follows:

```
!$omp do [clause[ [,] clause] ... ]
  do-loops
[!$omp end do [nowait]]
```

where *clause* is one of the following:

```
private(list)
firstprivate(list)
lastprivate(list)
linear(list[ : linear-step])
reduction(reduction-identifier : list)
schedule([modifier [, modifier]:]kind[, chunk_size])
collapse(n)
ordered[(n)]
```

If an **end do** directive is not specified, an **end do** directive is assumed at the end of the do-loops.

sections

The syntax of the sections construct is as follows:

```
!$omp sections [clause[ [,] clause] ... ]
  [!$omp section]
    structured-block
  [!$omp section
    structured-block]
  ...
!$omp end sections [nowait]
```

where *clause* is one of the following:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(reduction-identifier : list)
```

single

The syntax of the single construct is as follows:

```
!$omp single [clause[ [,] clause] ... ]
  structured-block
!$omp end single [end_clause[ [,] end_clause] ... ]
```

where *clause* is one of the following:

```
private(list)
firstprivate(list)
```

and *end_clause* is one of the following:

```
copyprivate(list)
nowait
```

workshare

The syntax of the workshare construct is as follows:

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

The enclosed structured block must consist of only the following:

```
array assignments
scalar assignments
FORALL statements
FORALL constructs
WHERE statements
WHERE constructs
atomic constructs
critical constructs
parallel constructs
```

simd

The syntax of the simd construct is as follows:

```
!$omp simd [clause[ [,] clause ... ]
  do-loops
[!$omp end simd]
```

where *clause* is one of the following:

```
safelen(length)
simdlen(length)
linear(list[ : linear-step])
aligned(list[ : alignment])
private(list)
lastprivate(list)
reduction(reduction-identifier : list)
collapse(n)
```

If an **end simd** directive is not specified, an **end simd** directive is assumed at the end of the *do-loops*.

declare simd

The syntax of the declare simd construct is as follows:

```
!$omp declare simd [(proc-name)] [clause[ [,] clause] ... ]
```

where *clause* is one of the following:

```
simdlen(length)
linear(linear-list[ : linear-step])
aligned(argument-list[ : alignment])
uniform(argument-list)
inbranch
notinbranch
```

Loop simd

The syntax of the Loop simd construct is as follows:

```
!$omp do simd [clause[ [,] clause] ... ]
  do-loops
[!$omp end do simd [nowait] ]
```

where *clause* can be any of the clauses accepted by the **simd** or **do** directives, with identical meanings and restrictions.

If an **end do simd** directive is not specified, an **end do simd** directive is assumed at the end of the do-loops.

Todo: finish the specs and add more details.

Data-Sharing Attribute Clauses

default

The syntax of the **default** clause is as follows:

```
default(private | firstprivate | shared | none)
```

shared

The syntax of the **shared** clause is as follows:

```
shared(list)
```

private

The syntax of the **private** clause is as follows:

```
private(list)
```

firstprivate

The syntax of the **firstprivate** clause is as follows:

```
firstprivate(list)
```

lastprivate

The syntax of the **lastprivate** clause is as follows:

```
lastprivate(list)
```

reduction

The syntax of the **reduction** clause is as follows:

```
reduction(reduction-identifier : list)
```

linear

The syntax of the **linear** clause is as follows:

```
linear(linear-list[ : linear-step])
```

where *linear-list* is one of the following:

```
list
modifier(list)
```

where *modifier* is one of the following:

```
val
```

Data Copying Clauses

copyin

The syntax of the **copyin** clause is as follows:

```
copyin(list)
```

copyprivate

The syntax of the **copyprivate** clause is as follows:

```
copyprivate(list)
```

Data-mapping Attribute Rules and Clauses

map

The syntax of the *map* clause is as follows:

```
map([ [map-type-modifier[,]] map-type : ] list)
```

where *map-type* is one of the following:

```
to
from
tofrom
alloc
release
delete
```

and *map-type-modifier* is **always**.

defaultmap

The syntax of the *defaultmap* clause is as follows:

```
defaultmap(tofrom:scalar)
```

declare reduction Directive

The syntax of the *declare reduction* directive is as follows:

```
!$omp declare reduction(reduction-identifier : type-list : combiner)
[initializer-clause]
```

where:

1. *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the following operators: **+**, **-**, ***** , **.and.**, **.or.**, **.eqv.**, **.neqv.**, or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.
2. *type-list* is a list of type specifiers
3. *combiner* is either an assignment statement or a subroutine name followed by an argument list
4. *initializer-clause* is **initializer** (*initializer-expr*), where *initializer-expr* is **omp_priv** = *expression* or *subroutine-name* (*argument-list*)

1.5 The Pyccel Compiler

Typical processing using **Pyccel** can be splitted into 3 main stages:

1. First, we parse the *Python* file or text, and we create an *intermediate representation* (**IR**) that consists of objects described in **pyccel.parser.syntax**
2. Most of all these objects, when it makes sens, implement the property **expr**. This allows to convert the object to one or more objects from **pyccel.ast.core**. All these objects are extension of the **sympy.core.Basic** class. At the end of this stage, the **IR** will be converted into the *Abstract Syntax Tree* (AST).
3. Using the **Codegen** classes or the user-friendly functions like **fcode**, the **AST** is converted into the target language (for example, *Fortran*)

Note: Always remember that **Pyccel** core is based on **sympy**. This can open a very wide range of applications and opportunities, like automatically evaluate the *computational complexity* of your code.

Note: There is an intermediate step between 2 and 3, where one can walk through the AST and modify the code by applying some recursive functions (ex: mpify, openmpfy, ...)

Todo: add diagram

The idea behind **Pyccel** is to use the available tools for **Python**, without having to implement everything as it is usually done for every new language. The aim of using such high level language is to ensure a user-friendly framework for developping massively parallel codes, without having to work in a hostile environment such as *Fortran* or an obscur language like *c++*. Most of all, compared to other *DSLs* for HPC, all elements and parallel paradigms of the language are exposed.

Among the very nice tools for *Python* developpers, Pylint is used for **static** checking. This allows us to avoid writing a **linter** tool or having to implement an advanced tool to handle **errors**. Following the **K.I.S.S** paradigm, we want to keep it *stupid* and *simple*, hence if you are getting errors with *Pylint*, do not expect *Pyccel* to run!! We assume that you are capable of writing a **valid** *Python* code. If not, then try first to learn *Python* before trying to do fancy things!

1.5.1 Compiling a single file

TODO

Syntax analysis

Semantic analysis

Code generation

Backend compilation

1.5.2 Setting up a project

The root directory of a Pyccel collection of pyccel sources is called the *source directory*. This directory also contains the Pyccel configuration file `conf.py`, where you can configure all aspects of how Pyccel converts your sources and builds your project.

Pyccel comes with a script called **`pyccel-quickstart`** that sets up a source directory and creates a default `conf.py` with the most useful configuration values. Just run

```
$ pyccel-quickstart -h
```

for help.

For example, runing:

```
$ pyccel-quickstart poisson
```

will create a directory **poisson** where you will find, inside it:

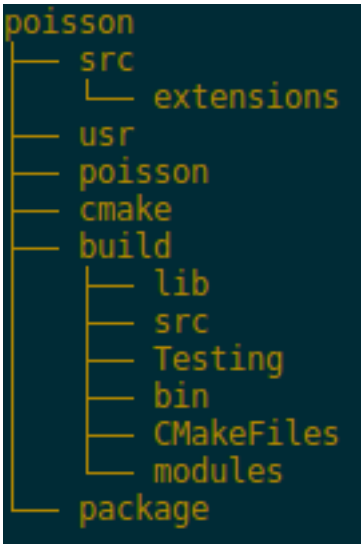

Fig. 3: Structure of the **poisson** project after running **`pyccel-quickstart`**.

1.5.3 Defining document structure

Let's assume you've run **pyccel-quickstart** for a project **poisson**. It created a source directory with `conf.py` and a directory **poisson** that contains a master file, `main.py` (if you used the defaults settings). The main function of the *master document* is to serve as an example of a **main program**.

1.5.4 Adding content

In Pyccel source files, you can use most features of standard *Python* instructions. There are also several features added by Pyccel. For example, you can use multi-threading or distributed memory programing paradigms, as part of the Pyccel language itself.

1.5.5 Running the build

Now that you have added some files and content, let's make a first build of the project. A build is started with the **pyccel-build** program, called like this:

```
$ pyccel-build application
```

where *application* is the *application directory* you want to build.

Refer to the **pyccel-build man page <pyccel-build>** for all options that **pyccel-build** supports.

Notice that **pyccel-quickstart** script creates a build directory *build directory* in which you can use **cmake** or `Makefile`. In order to compile *manualy* your project, you just need to go to this build directory and run

```
$ make
```

1.5.6 Basic configuration

Todo: add basic configurations.

1.5.7 Contents

Syntax analysis

We use RedBaron to parse the *Python* code. **BNF** grammars are used to parse *headers*, *OpenMP* and *OpenAcc*. This is based on the textX project.

Static verification

For more details, we highly recommand to take a look at Pylint documentation.

Pylint messages

To understand what *Pylint* is trying to tell you, please take a look at Pylint codes.

Retrieving messages from Pylint

Pyccel uses the parse library to retrieve error messages from *Pylint*.

Semantic analysis

TODO

Type Inference

Code generation

TODO

Working with projects

TODO

Rules

In this section, we provide a set of rules that were used to ensure the **one-to-one** correspondance between *Python* and *Fortran*. These rules are applied while annotating the *AST*.

Note: the first letter describes the message nature (W: warning, E: error, . . .)

Note: the second letter describes the related section (F: function, E: expression, . . .)

Todo: bounds check

- **EF001**: a function with with at least one argument or returned value, must have an associated header
- **ES001**: **Except** statement is not covered by pyccel
- **ES002**: **Finally** statement is not covered by pyccel
- **ES003**: **Raise** statement is not covered by pyccel
- **ES004**: **Try** statement is not covered by pyccel
- **ES005**: **Yield** statement is not covered by pyccel
- **WF001**: all returned arguments must be atoms, no expression is allowed
- **WF002**: a returned variable should appear in the function header. Otherwise, *Type Inference* is used.
- **WC001**: class defined without **__del__** method. It will be added automatically

Syntax

- class attributes are defined as *DottedName* objects. This means that an expression

```
self.x = x
```

  can be *printed*, while

```
self.x = self.y
```

  will lead to an error, since *sympy* can not manipulate *DottedName*, which is suppose to be a *string* and not a *Symbol*. This is fixed in the *semantic* stage by converting *Symbol* objects to *Variable*.

Semantic

Type inference

Functions

- a function that has no argument and no result does not need a header.
- Functions with no parameterized arguments must have a header
- Functions with only parameterized arguments does not need a header
- all returned arguments must be in general atoms, no expression is allowed
- results are computed at the decoration stage

1.6 Overview

Todo: add diagram

1.6.1 Syntax

We use RedBaron to parse the *Python* code. For *headers*, *OpenMP* and *OpenAcc* we use textX

In order to achieve *syntax analysis*, we first use *RedBaron* to get the **FST** (Full Syntax Tree), then we convert its nodes to our *sympy* **AST**. During this stage

- variables are described as *sympy* **Symbol** objects

Note: a **Symbol** can be viewed as a variable with **undefined type**

In the *semantic analysis* process, we *decorate* our *AST* and

- use **type inference** to get the type of every *symbol*
- change *Symbol* objects to **Variable** when it is possible

Note: since our target language is *Fortran*, we only convert variables that have a *type*.

Full Syntax Tree (FST)

Abstract Syntax Tree (AST)

1.7 Man Pages

These are the applications provided as part of Pyccel.

1.7.1 Core Applications

pyccel-quickstart

The **pyccel-quickstart** script generates a Pyccel documentation set. It is called like this:

```
$ pyccel-quickstart [options] [projectdir]
```

where *projectdir* is the Pyccel documentation set directory in which you want to place. If you omit *projectdir*, files are generated into current directory by default.

The **pyccel-quickstart** script has several options:

-q, --quiet
Quiet mode that will skips interactive wizard to specify options. This option requires *-a* and *-v* options.

-h, --help, --version
Display usage summary or Pyccel version.

Structure options

--sep
If specified, separate source and build directories.

--dot=DOT
You can define a prefix for the temporary directories: build, etc You can enter another prefix (such as ".") to replace the underscore.

Project basic options:

-a AUTHOR, **--author**=AUTHOR
Author names. (see `copyright`).

-v VERSION
Version of project. (see `version`).

-r RELEASE, **--release**=RELEASE
Release of project. (see `release`).

-l LANGUAGE, **--language**=LANGUAGE
Low-level language. (see `language`).

--suffix-library=SUFFIX_LIBRARY
Suffix of 3 letters for the project. (see source_suffix).

--master=MASTER
Master file name. (see master_doc).

--compiler=COMPILER
A valid compiler. (see compiler_doc).

--include INCLUDE
path to include directory. (see compiler_doc).

--libdir LIBDIR
path to lib directory. (see compiler_doc).

--libs LIBS
list of libraries to link with. (see compiler_doc).

--convert-only
Converts pyccel files only without build. (see convertion_doc).

Extension options

--ext-blas
Enable *pyccelext.blas* extension.

--ext-math
Enable *pyccelext.math* extension.

pyccel-build

The **pyccel-build** script builds a Pyccel documentation set. It is called like this:

```
$ pyccel-build [options] sourcedir [filenames]
```

where *sourcedir* is the *source directory*. Most of the time, you don't need to specify any *filenames*.

The **pyccel-build** script has several options:

-h, --help, --version
Display usage summary or Pyccel version.

General options

--output-dir OUTPUT_DIR
Output directory.

--convert-only
Converts pyccel files only without build. (see convertion_doc).

-b BUILDER
builder to use (default: fortran)

-a
write all files (default: only write new and changed files)

-E
don't use a saved environment, always read all files

-j N
build in parallel with N processes where possible

Build configuration options

-c PATH
path where configuration file (conf.py) is located (default: same as SOURCEDIR)

-D setting=value
override a setting in configuration file

Console output options

-v
increase verbosity (can be repeated)

-q
no output on stdout, just warnings on stderr

-Q
no output at all, not even warnings

-W
turn warnings into errors

1.8 Code Analysis

1.8.1 Complexity

We consider the following matrix-matrix product example

```
n = int()
n = 2

x = zeros(shape=(n,n), dtype=float)
y = zeros(shape=(n,n), dtype=float)
z = zeros(shape=(n,n), dtype=float)

for i in range(0, n):
    for j in range(0, n):
        for k in range(0, n):
            z[i,j] = z[i,j] + x[i,k]*y[k,j]
```

now let's run the following command:

```
$ pyccel --filename=tests/complexity/inputs/ex3.py --analysis
arithmetic cost         ~ n**3*(ADD + MUL)
memory cost             ~ WRITE + n**3*(3*READ + WRITE)
computational intensity ~ (ADD + MUL)/(3*READ + WRITE)
```

Let's now consider the following block version of the matrix-matrix product

```
n = int()
b = int()
m = int()
n = 10
b = 2
p = n / b

x = zeros(shape=(n,n), dtype=float)
y = zeros(shape=(n,n), dtype=float)
z = zeros(shape=(n,n), dtype=float)

r = zeros(shape=(b,b), dtype=float)
u = zeros(shape=(b,b), dtype=float)
v = zeros(shape=(b,b), dtype=float)

for i in range(0, p):
    for j in range(0, p):
        for k1 in range(0, b):
            for k2 in range(0, b):
                r[k1,k2] = z[i+k1,j+k2]
        for k in range(0, p):
            for k1 in range(0, b):
                for k2 in range(0, b):
                    u[k1,k2] = x[i+k1,k+k2]
                    v[k1,k2] = y[k+k1,j+k2]
            for ii in range(0, b):
                for jj in range(0, b):
                    for kk in range(0, b):
                        r[ii,jj] = r[ii,jj] + u[ii,kk]*v[kk,jj]
        for k1 in range(0, b):
            for k2 in range(0, b):
                z[i+k1,j+k2] = r[k1,k2]
```

the analysis is done again using:

```
$ pyccel --filename=tests/complexity/inputs/ex4.py --analysis
arithmetic cost         ~ DIV + b**3*p**3*(ADD + MUL)
memory cost             ~ 2*READ + 3*WRITE + b**2*p**2*(2*READ + 2*WRITE + p*(2*READ␣
↪+ 2*WRITE + b*(3*READ + WRITE)))
computational intensity ~ (ADD + MUL)/(3*READ + WRITE)
```

Now, let us assume we have two level of memories, the **fast** memory represents the **L2** cache. By giving the variables that live in the cache, using **local_vars**, the analysis gives:

```
$ pyccel --filename=tests/complexity/inputs/ex4.py --analysis --local_vars="u,v,r"
arithmetic cost         ~ DIV + b**3*p**3*(ADD + MUL)
memory cost             ~ 2*READ + 3*WRITE + b**2*p**2*(2*READ*p + READ + WRITE)
computational intensity ~ b*(ADD + MUL)/(2*READ)
```

As we can see, the computational intensity is now a linear function of the block size b. Therefor, this algorithm will take more advantage of the spatial locality of data.

Todo: remove local_vars from **pyccel** command line and use Annotated Comments instead.

Todo: for the moment, we only cover the **for** statement. Further work must be done for **if** and **while** statements.

Todo: add probability law for the **if** statement.

Todo: how to handle the **while** statement?

Arithmetic

Todo: add Fusion Mul-Add (FMA) instruction

Todo: add table of costs for all instructions

Memory

We describe here our *Memory model*. It follows the work of J. Demmel and his collaborators on the matrix multiplication. More details can be found in J. Demmel's talk

Here are our assumptions:

1. Two levels of memory: *fast* and *slow*
2. All data are initially in *slow* memory

1.9 Road Map

Todo: add diagram

CHAPTER 2

API

2.1 pyccel

2.1.1 pyccel package

Subpackages

pyccel.ast package

Subpackages

pyccel.ast.parallel package

Submodules

pyccel.ast.parallel.basic module

class pyccel.ast.parallel.basic.**Basic**
Bases: sympy.core.basic.Basic

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

dtype(*attr*)
Returns the datatype of a given attribut/member.

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

pyccel.ast.parallel.communicator module

class pyccel.ast.parallel.communicator.**Communicator**
Bases: *pyccel.ast.parallel.basic.Basic*

Represents a communicator in the code.

size: int the number of processes in the group associated with the communicator

rank: int the rank of the calling process within the group associated with the communicator

Examples

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

duplicate()
This routine creates a duplicate of the communicator other has the same fixed attributes as the communicator.

Examples

group
the group associated with the communicator.

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

rank

size

split(*group*)
Split the communicator over colors by returning the new comm associated to the processes with defined color.

Examples

class pyccel.ast.parallel.communicator.**UniversalCommunicator**
Bases: *pyccel.ast.parallel.basic.Basic*

Represents the communicator to all processes.

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

group

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

size

pyccel.ast.parallel.communicator.**split**(*comm*, *group*, *rank=None*)
Splits the communicator over a given color.

Examples

pyccel.ast.parallel.group module

class pyccel.ast.parallel.group.**Difference**
Bases: sympy.sets.sets.Complement

Represents the difference between two groups.

Examples

```
>>> from pyccel.ast.parallel.group import Group, Difference
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
>>> Difference(g1, g2)
{1, 3, 6, 7}
```

default_assumptions = {}

class pyccel.ast.parallel.group.**Group**
Bases: sympy.sets.sets.FiniteSet

Represents a group of processes.

processes: Set set of the processes within the group

rank: int the rank of the calling process within the group

Examples

```
>>> from pyccel.ast.parallel.group import Group
>>> g = Group(1, 2, 3, 4)
>>> g
{1, 2, 3, 4}
>>> g.size
4
```

communicator

compare(*other*)
This routine returns the relationship between self and other If self and other contain the same processes, ranked the same way, this routine returns MPI_IDENT If self and other contain the same processes, but ranked differently, this routine returns MPI_SIMILAR Otherwise this routine returns MPI_UNEQUAL.

default_assumptions = {}

difference(*other*)
Returns in newgroup all processes in self that are not in other, ordered as in self.

Examples

As a shortcut it is possible to use the '-' operator:

```
>>> from pyccel.ast.parallel.group import Group
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
>>> g1.difference(g2)
{1, 3, 6, 7}
>>> g1 - g2
{1, 3, 6, 7}
```

intersection(*other*)
Returns in newgroup all processes that are in both groups, ordered as in self.

Examples

```
>>> from pyccel.ast.parallel.group import Group
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
>>> g1.intersection(g2)
{8, 9}
```

processes
Returns the set of the group processes.

rank

set_communicator(*comm*)

size
the number of processes in the group.

translate(*rank1*, *other*, *rank2*)
This routine takes an array of n ranks (ranks1) which are ranks of processes in self. It returns in ranks2 the corresponding ranks of the processes as they are in other. MPI_UNDEFINED is returned for processes not in other

union(*other*)
Returns in newgroup a group consisting of all processes in self followed by all processes in other, with no duplication.

Examples

As a shortcut it is possible to use the '+' operator:

```
>>> from pyccel.ast.parallel.group import Group
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
>>> g1.union(g2)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> g1 + g2
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

class pyccel.ast.parallel.group.**Intersection**
Bases: sympy.sets.sets.Intersection

Represents the intersection of groups.

Examples

```
>>> from pyccel.ast.parallel.group import Group, Intersection
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
```

(continues on next page)

(continued from previous page)

```
>>> Intersection(g1, g2)
{8, 9}
```

default_assumptions = {}

class pyccel.ast.parallel.group.**Range**

Bases: sympy.sets.fancysets.Range

Representes a range of processes.

Examples

```
>>> from pyccel.ast.parallel.group import Range
>>> from sympy import Symbol
>>> n = Symbol('n')
>>> Range(0, n)
Range(0, n)
```

default_assumptions = {}

class pyccel.ast.parallel.group.**Split**

Bases: *pyccel.ast.parallel.group.Group*

Splits the group over a given color.

Examples

```
>>> from pyccel.ast.parallel.group import UniversalGroup
>>> from pyccel.ast.parallel.communicator import UniversalCommunicator, split
>>> colors = [1, 1, 0, 1, 0, 0]
>>> g = Split(UniversalGroup(), colors, 0)
>>> g
{2, 4, 5}
>>> comm = split(UniversalCommunicator(), g)
Communicator({2, 4, 5}, None)
```

default_assumptions = {}

class pyccel.ast.parallel.group.**Union**

Bases: sympy.sets.sets.Union

Represents the union of groups.

Examples

```
>>> from pyccel.ast.parallel.group import Group, Union
>>> g1 = Group(1, 3, 6, 7, 8, 9)
>>> g2 = Group(0, 2, 4, 5, 8, 9)
>>> Union(g1, g2)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

default_assumptions = {}

class pyccel.ast.parallel.group.**UniversalGroup**

Bases: sympy.sets.fancysets.Naturals

Represents the group of all processes. Since the number of processes is only known at the run-time and the universal group is assumed to contain all processes, it is convinient to consider it as the set of all possible processes which is nothing else than the set of Natural numbers.

np: Symbol a sympy symbol describing the total number of processes.

communicator

default_assumptions = {}

np = np

processes

size
the total number of processes.

pyccel.ast.parallel.mpi module

class pyccel.ast.parallel.mpi.**MPI**
Bases: *pyccel.ast.parallel.basic.Basic*

Base class for MPI.

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

pyccel.ast.parallel.mpi.**mpify**(*stmt*, ***options*)
Converts some statements to MPI statments.

stmt: stmt, list statement or a list of statements

pyccel.ast.parallel.openacc module

class pyccel.ast.parallel.openacc.**ACC**
Bases: *pyccel.ast.parallel.basic.Basic*

Base class for OpenACC.

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

class pyccel.ast.parallel.openacc.**ACC_Async**
Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Async
>>> ACC_Async('x', 'y')
async(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'async'

variables

class pyccel.ast.parallel.openacc.**ACC_Auto**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Auto
>>> ACC_Auto()
auto
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'auto'

class pyccel.ast.parallel.openacc.**ACC_Bind**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Bind
>>> ACC_Bind('n')
bind(n)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'bind'

variable

class pyccel.ast.parallel.openacc.**ACC_Collapse**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Collapse
>>> ACC_Collapse(2)
collapse(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_loops

name = 'collapse'

class pyccel.ast.parallel.openacc.**ACC_Copy**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Copy
>>> ACC_Copy('x', 'y')
copy(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'copy'

variables

class pyccel.ast.parallel.openacc.**ACC_Copyin**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Copyin
>>> ACC_Copyin('x', 'y')
copyin(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'copyin'

variables

class pyccel.ast.parallel.openacc.**ACC_Copyout**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Copyout
>>> ACC_Copyout('x', 'y')
copyout(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'copyout'

variables

class pyccel.ast.parallel.openacc.**ACC_Create**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Create
>>> ACC_Create('x', 'y')
create(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'create'

variables

class pyccel.ast.parallel.openacc.**ACC_Default**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Default
>>> ACC_Default('present')
default(present)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = None

status

class pyccel.ast.parallel.openacc.**ACC_DefaultAsync**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_DefaultAsync
>>> ACC_DefaultAsync('x', 'y')
default_async(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'default_async'

variables

class pyccel.ast.parallel.openacc.**ACC_Delete**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Delete
>>> ACC_Delete('x', 'y')
delete(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'delete'

variables

class pyccel.ast.parallel.openacc.**ACC_Device**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Device
>>> ACC_Device('x', 'y')
device(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'device'

variables

class pyccel.ast.parallel.openacc.**ACC_DeviceNum**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_DeviceNum
>>> ACC_DeviceNum(2)
device_num(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_device

name = 'device_num'

class pyccel.ast.parallel.openacc.**ACC_DevicePtr**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_DevicePtr
>>> ACC_DevicePtr('x', 'y')
deviceptr(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'deviceptr'

variables

class pyccel.ast.parallel.openacc.**ACC_DeviceResident**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_DeviceResident
>>> ACC_DeviceResident('x', 'y')
device_resident(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'device_resident'

variables

class pyccel.ast.parallel.openacc.**ACC_DeviceType**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_DeviceType
>>> ACC_DeviceType('x', 'y')
device_type(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'device_type'

variables

class pyccel.ast.parallel.openacc.**ACC_Finalize**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Finalize
>>> ACC_Finalize()
finalize
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'finalize'

class pyccel.ast.parallel.openacc.**ACC_FirstPrivate**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_FirstPrivate
>>> ACC_FirstPrivate('x', 'y')
firstprivate(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'firstprivate'

variables

class pyccel.ast.parallel.openacc.**ACC_For**

Bases: *pyccel.ast.core.ForIterator*, *pyccel.ast.parallel.openacc.ACC*

ACC Loop construct statement.

Examples

body

clauses

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

iterable

loop

name = 'do'

target

class pyccel.ast.parallel.openacc.**ACC_Gang**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Gang
>>> ACC_Gang('x', 'y')
gang(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'gang'

variables

class pyccel.ast.parallel.openacc.**ACC_Host**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Host
>>> ACC_Host('x', 'y')
host(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'host'

variables

class pyccel.ast.parallel.openacc.**ACC_If**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_If
>>> ACC_If(True)
if (True)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'if'

test

class pyccel.ast.parallel.openacc.**ACC_IfPresent**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_IfPresent
>>> ACC_IfPresent()
if_present
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'if_present'

class pyccel.ast.parallel.openacc.**ACC_Independent**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Independent
>>> ACC_Independent()
independent
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'independent'

class pyccel.ast.parallel.openacc.**ACC_Link**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Link
>>> ACC_Link('x', 'y')
link(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'link'

variables

class pyccel.ast.parallel.openacc.**ACC_NoHost**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_NoHost
>>> ACC_NoHost()
nohost
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'nohost'

class pyccel.ast.parallel.openacc.**ACC_NumGangs**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_NumGangs
>>> ACC_NumGangs(2)
num_gangs(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_gang

name = 'num_gangs'

class pyccel.ast.parallel.openacc.**ACC_NumWorkers**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_NumWorkers
>>> ACC_NumWorkers(2)
num_workers(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_worker

name = 'num_workers'

class pyccel.ast.parallel.openacc.**ACC_Parallel**

Bases: *pyccel.ast.core.ParallelBlock*, *pyccel.ast.parallel.openacc.ACC*

ACC Parallel construct statement.

Examples

```
>>> from pyccel.parallel.openacc import ACC_Parallel
>>> from pyccel.parallel.openacc import ACC_NumThread
>>> from pyccel.parallel.openacc import ACC_Default
>>> from pyccel.ast.core import Variable, Assign, Block
>>> n = Variable('int', 'n')
>>> x = Variable('int', 'x')
>>> body = [Assign(x,2.*n + 1.), Assign(n, n + 1)]
>>> variables = [x,n]
>>> clauses = [ACC_NumThread(4), ACC_Default('shared')]
>>> ACC_Parallel(clauses, variables, body)
#pragma parallel num_threads(4) default(shared)
x := 1.0 + 2.0*n
n := 1 + n
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'parallel'

class pyccel.ast.parallel.openacc.**ACC_Present**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Present
>>> ACC_Present('x', 'y')
present(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'present'

variables

class pyccel.ast.parallel.openacc.**ACC_Private**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Private
>>> ACC_Private('x', 'y')
private(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'private'

variables

class pyccel.ast.parallel.openacc.**ACC_Reduction**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Reduction
>>> ACC_Reduction('+', 'x', 'y')
reduction('+': (x, y))
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'reduction'

operation

variables

class pyccel.ast.parallel.openacc.**ACC_Self**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Self
>>> ACC_Self('x', 'y')
self(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'self'

variables

class pyccel.ast.parallel.openacc.**ACC_Seq**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Seq
>>> ACC_Seq()
seq
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'seq'

class pyccel.ast.parallel.openacc.**ACC_Tile**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Tile
>>> ACC_Tile('x', 'y')
tile(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'tile'

variables

class pyccel.ast.parallel.openacc.**ACC_UseDevice**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_UseDevice
>>> ACC_UseDevice('x', 'y')
use_device(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'use_device'

variables

class pyccel.ast.parallel.openacc.**ACC_Vector**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Vector
>>> ACC_Vector('x', 'y')
vector(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'vector'

variables

class pyccel.ast.parallel.openacc.**ACC_VectorLength**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_VectorLength
>>> ACC_VectorLength(2)
vector_length(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n

name = 'vector_length'

class pyccel.ast.parallel.openacc.**ACC_Wait**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Wait
>>> ACC_Wait('x', 'y')
wait(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'wait'

variables

class pyccel.ast.parallel.openacc.**ACC_Worker**

Bases: *pyccel.ast.parallel.openacc.ACC*

Examples

```
>>> from pyccel.parallel.openacc import ACC_Worker
>>> ACC_Worker('x', 'y')
worker(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'worker'

variables

pyccel.ast.parallel.openacc.**accfy**(*stmt*, ***options*)
Converts some statements to OpenACC statments.

stmt: stmt, list statement or a list of statements

pyccel.ast.parallel.openacc.**get_for_clauses**(*expr*)

pyccel.ast.parallel.openacc.**get_with_clauses**(*expr*)

pyccel.ast.parallel.openmp module

class pyccel.ast.parallel.openmp.**OMP**
Bases: *pyccel.ast.parallel.basic.Basic*

Base class for OpenMP.

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

class pyccel.ast.parallel.openmp.**OMP_Collapse**
Bases: *pyccel.ast.parallel.openmp.OMP*

OMP CollapseClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Collapse
>>> OMP_Collapse(2)
collapse(2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_loops

name = 'collapse'

class pyccel.ast.parallel.openmp.**OMP_Copyin**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP CopyinClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Copyin
>>> OMP_Copyin('x', 'y')
copyin(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'copyin'

variables

class pyccel.ast.parallel.openmp.**OMP_Default**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ParallelDefaultClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Default
>>> OMP_Default('shared')
default(shared)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = None

status

class pyccel.ast.parallel.openmp.**OMP_FirstPrivate**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP FirstPrivateClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_FirstPrivate
>>> OMP_FirstPrivate('x', 'y')
firstprivate(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'firstprivate'

variables

class pyccel.ast.parallel.openmp.**OMP_For**

Bases: *pyccel.ast.core.ForIterator*, *pyccel.ast.parallel.openmp.OMP*

OMP Parallel For construct statement.

Examples

body

clauses

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

iterable

loop

name = 'do'

nowait

target

class pyccel.ast.parallel.openmp.**OMP_If**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ParallelIfClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_If
>>> OMP_If(True)
if (True)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'if'

test

class pyccel.ast.parallel.openmp.**OMP_LastPrivate**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP LastPrivateClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_LastPrivate
>>> OMP_LastPrivate('x', 'y')
lastprivate(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'lastprivate'

variables

class pyccel.ast.parallel.openmp.**OMP_Linear**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP LinearClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Linear
>>> OMP_Linear('x', 'y', 2)
linear((x, y): 2)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'linear'

step

variables

class pyccel.ast.parallel.openmp.**OMP_NumThread**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ParallelNumThreadClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_NumThread
>>> OMP_NumThread(4)
num_threads(4)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'num_threads'

num_threads

class pyccel.ast.parallel.openmp.**OMP_Ordered**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP OrderedClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Ordered
>>> OMP_Ordered(2)
ordered(2)
>>> OMP_Ordered()
ordered
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

n_loops

name = 'ordered'

class pyccel.ast.parallel.openmp.**OMP_Parallel**

Bases: *pyccel.ast.core.ParallelBlock*, *pyccel.ast.parallel.openmp.OMP*

OMP Parallel construct statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Parallel
>>> from pyccel.parallel.openmp import OMP_NumThread
>>> from pyccel.parallel.openmp import OMP_Default
>>> from pyccel.ast.core import Variable, Assign, Block
>>> n = Variable('int', 'n')
>>> x = Variable('int', 'x')
>>> body = [Assign(x,2.*n + 1.), Assign(n, n + 1)]
>>> variables = [x,n]
>>> clauses = [OMP_NumThread(4), OMP_Default('shared')]
>>> OMP_Parallel(clauses, variables, body)
#pragma parallel num_threads(4) default(shared)
x := 1.0 + 2.0*n
n := 1 + n
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'parallel'

class pyccel.ast.parallel.openmp.**OMP_Private**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP PrivateClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Private
>>> OMP_Private('x', 'y')
private(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'private'

variables

class pyccel.ast.parallel.openmp.**OMP_ProcBind**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ParallelProcBindClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_ProcBind
>>> OMP_ProcBind('master')
proc_bind(master)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'proc_bind'

status

class pyccel.ast.parallel.openmp.**OMP_Reduction**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ReductionClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Reduction
>>> OMP_Reduction('+', 'x', 'y')
reduction('+': (x, y))
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'reduction'

operation

variables

class pyccel.ast.parallel.openmp.**OMP_Schedule**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP ScheduleClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Schedule
>>> OMP_Schedule('static', 2)
schedule(static, 2)
```

chunk_size

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

kind

name = 'schedule'

class pyccel.ast.parallel.openmp.**OMP_Shared**

Bases: *pyccel.ast.parallel.openmp.OMP*

OMP SharedClause statement.

Examples

```
>>> from pyccel.parallel.openmp import OMP_Shared
>>> OMP_Shared('x', 'y')
shared(x, y)
```

default_assumptions = {'composite': False, 'even': False, 'integer': False, 'odd':

is_composite = False

is_even = False

is_integer = False

is_odd = False

is_prime = False

is_zero = False

name = 'shared'

variables

pyccel.ast.parallel.openmp.**get_for_clauses**(*expr*)

pyccel.ast.parallel.openmp.**get_with_clauses**(*expr*)

pyccel.ast.parallel.openmp.**ompfy**(*stmt*, ***options*)

Converts some statements to OpenMP statments.

stmt: stmt, list statement or a list of statements

Module contents

Submodules

pyccel.ast.basic module

class pyccel.ast.basic.**Basic**
Bases: sympy.core.basic.Basic

Basic class for Pyccel AST.

default_assumptions = {}

fst

set_fst(*fst*)
Sets the redbaron fst.

pyccel.ast.core module

class pyccel.ast.core.**AddOp**
Bases: *pyccel.ast.core.NativeOp*

default_assumptions = {}

class pyccel.ast.core.**AliasAssign**
Bases: *pyccel.ast.basic.Basic*

Represents aliasing for code generation. An alias is any statement of the form *lhs := rhs* where

lhs [Symbol] at this point we don't know yet all information about lhs, this is why a Symbol is the appropriate type.

rhs [Variable, IndexedVariable, IndexedElement] an assignable variable can be of any rank and any datatype, however its shape must be known (not None)

Examples

```
>>> from sympy import Symbol
>>> from pyccel.ast.core import AliasAssign
>>> from pyccel.ast.core import Variable
>>> n = Variable('int', 'n')
>>> x = Variable('int', 'x', rank=1, shape=[n])
>>> y = Symbol('y')
>>> AliasAssign(y, x)
```

default_assumptions = {}

lhs

rhs

class pyccel.ast.core.**AnnotatedComment**
Bases: *pyccel.ast.basic.Basic*

Represents a Annotated Comment in the code.

accel [str] accelerator id. One among {'omp', 'acc'}

txt: str statement to print

Examples

```
>>> from pyccel.ast.core import AnnotatedComment
>>> AnnotatedComment('omp', 'parallel')
AnnotatedComment(omp, parallel)
```

accel

default_assumptions = {}

txt

class pyccel.ast.core.**Argument**
Bases: sympy.core.symbol.Symbol

An abstract Argument data structure.

Examples

```
>>> from pyccel.ast.core import Argument
>>> n = Argument('n')
>>> n
n
```

default_assumptions = {}

class pyccel.ast.core.**AsName**
Bases: *pyccel.ast.basic.Basic*

Represents a renaming of a variable, used with Import.

Examples

```
>>> from pyccel.ast.core import AsName
>>> AsName('new', 'old')
new as old
```

default_assumptions = {}

name

target

class pyccel.ast.core.**Assert**
Bases: *pyccel.ast.basic.Basic*

Represents a assert statement in the code.

test: Expr boolean expression to check

Examples

default_assumptions = {}

test

class pyccel.ast.core.**Assign**
Bases: *pyccel.ast.basic.Basic*

Represents variable assignment for code generation.

lhs [Expr] Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

rhs [Expr] Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

strict: bool if True, we do some verifications. In general, this can be more complicated and is treated in pyccel.syntax.

status: None, str if lhs is not allocatable, then status is None. otherwise, status is {'allocated', 'unallocated'}

like: None, Variable contains the name of the variable from which the lhs will be cloned.

Examples

```
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from pyccel.ast.core import Assign
>>> x, y, z = symbols('x, y, z')
>>> Assign(x, y)
x := y
>>> Assign(x, 0)
x := 0
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([x, y, z]).T
>>> Assign(A, mat)
A := Matrix([[x, y, z]])
>>> Assign(A[0, 1], x)
A[0, 1] := x
```

default_assumptions = {}

expr

is_alias
Returns True if the assignment is an alias.

is_symbolic_alias
Returns True if the assignment is a symbolic alias.

lhs

like

rhs

status

strict

class pyccel.ast.core.**AugAssign**
Bases: *pyccel.ast.basic.Basic*

Represents augmented variable assignment for code generation.

lhs [Expr] Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

op [NativeOp] Operator (+, -, /, *, %).

rhs [Expr] Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

strict: bool if True, we do some verifications. In general, this can be more complicated and is treated in pyccel.syntax.

status: None, str if lhs is not allocatable, then status is None. otherwise, status is {'allocated', 'unallocated'}

like: None, Variable contains the name of the variable from which the lhs will be cloned.

Examples

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import AugAssign
>>> s = Variable('int', 's')
>>> t = Variable('int', 't')
>>> AugAssign(s, '+', 2 * t + 1)
s += 1 + 2*t
```

default_assumptions = {}

lhs

like

op

rhs

status

strict

class pyccel.ast.core.**Block**

Bases: *pyccel.ast.basic.Basic*

Represents a block in the code. A block consists of the following inputs

variables: list list of the variables that appear in the block.

declarations: list list of declarations of the variables that appear in the block.

body: list a list of statements

Examples

```
>>> from pyccel.ast.core import Variable, Assign, Block
>>> n = Variable('int', 'n')
>>> x = Variable('int', 'x')
>>> Block([n, x], [Assign(x,2.*n + 1.), Assign(n, n + 1)])
Block([n, x], [x := 1.0 + 2.0*n, n := 1 + n])
```

body

declarations

default_assumptions = {}

name

variables

class pyccel.ast.core.**Break**

Bases: *pyccel.ast.basic.Basic*

Represents a break in the code.

default_assumptions = {}

class pyccel.ast.core.**ClassDef**

Bases: *pyccel.ast.basic.Basic*

Represents a class definition.

name [str] The name of the class.

attributs: iterable The attributs to the class.

methods: iterable Class methods

options: list, tuple list of options ('public', 'private', 'abstract')

imports: list, tuple list of needed imports

parent [str] parent's class name

Examples

```
>>> from pyccel.ast.core import Variable, Assign
>>> from pyccel.ast.core import ClassDef, FunctionDef
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> z = Variable('real', 'z')
>>> t = Variable('real', 't')
>>> a = Variable('real', 'a')
>>> b = Variable('real', 'b')
>>> body = [Assign(y,x+a)]
>>> translate = FunctionDef('translate', [x,y,a,b], [z,t], body)
>>> attributs   = [x,y]
>>> methods     = [translate]
>>> ClassDef('Point', attributs, methods)
ClassDef(Point, (x, y), (FunctionDef(translate, (x, y, a, b), (z, t), [y := a +␣
↪x], [], [], None, False, function),), [public])
```

attributes

attributes_as_dict
Returns a dictionary that contains all attributs, where the key is the attribut's name.

default_assumptions = {}

get_attribute(*O*, *attr*)
Returns the attribute attr of the class O of instance self.

hide

imports

interfaces

is_iterable
Returns True if the class has an iterator.

is_with_construct
Returns True if the class is a with construct.

methods

methods_as_dict
Returns a dictionary that contains all methods, where the key is the method's name.

name

options

parent

class pyccel.ast.core.**CodeBlock**
Bases: *pyccel.ast.basic.Basic*

Represents a list of stmt for code generation. we use it when a single statement in python produce multiple statement in the targeted language

body

default_assumptions = {}

lhs

class pyccel.ast.core.**Comment**
Bases: *pyccel.ast.basic.Basic*

Represents a Comment in the code.

text [str] the comment line

Examples

```
>>> from pyccel.ast.core import Comment
>>> Comment('this is a comment')
# this is a comment
```

default_assumptions = {}

text

class pyccel.ast.core.**CommentBlock**
Bases: *pyccel.ast.basic.Basic*

Represents a Block of Comments txt : str

comments

default_assumptions = {}

class pyccel.ast.core.**Concatinate**
Bases: *pyccel.ast.basic.Basic*

Represents the String concatination operation.

left : Symbol or string or List

right : Symbol or string or List

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Concatinate
>>> x = symbols('x')
>>> Concatinate('some_string',x)
some_string+x
>>> Concatinate('some_string','another_string')
'some_string' + 'another_string'
```

args

default_assumptions = {}

is_list

class pyccel.ast.core.**Constant**
Bases: *pyccel.ast.core.ValuedVariable*

Examples

default_assumptions = {}

class pyccel.ast.core.**ConstructorCall**
Bases: sympy.core.expr.AtomicExpr

It serves as a constructor for undefined function classes.

func: FunctionDef, str an instance of FunctionDef or function name

arguments: list, tuple, None a list of arguments.

kind: str 'function' or 'procedure'. default value: 'function'

arguments

cls_variable

default_assumptions = {'commutative': True}

func

is_commutative = True

kind

name

class pyccel.ast.core.**Continue**
Bases: *pyccel.ast.basic.Basic*

Represents a continue in the code.

default_assumptions = {}

class pyccel.ast.core.**Declare**
Bases: *pyccel.ast.basic.Basic*

Represents a variable declaration in the code.

dtype [DataType] The type for the declaration.

variable(s) A single variable or an iterable of Variables. If iterable, all Variables must be of the same type.

intent: None, str one among {'in', 'out', 'inout'}

value: Expr variable value

static: bool True for a static declaration of an array.

Examples

```
>>> from pyccel.ast.core import Declare, Variable
>>> Declare('int', Variable('int', 'n'))
Declare(NativeInteger(), (n,), None)
>>> Declare('real', Variable('real', 'x'), intent='out')
Declare(NativeReal(), (x,), out)
```

default_assumptions = {}

dtype

intent

static

value

variable

class pyccel.ast.core.**Del**
Bases: *pyccel.ast.basic.Basic*

Represents a memory deallocation in the code.

variables [list, tuple] a list of pyccel variables

Examples

```
>>> from pyccel.ast.core import Del, Variable
>>> x = Variable('real', 'x', rank=2, shape=(10,2), allocatable=True)
>>> Del([x])
Del([x])
```

default_assumptions = {}

variables

class pyccel.ast.core.**DivOp**
Bases: *pyccel.ast.core.NativeOp*

default_assumptions = {}

class pyccel.ast.core.**Dlist**
Bases: *pyccel.ast.basic.Basic*

this is equivalent to the zeros function of numpy arrays for the python list.

value [Expr] a sympy expression which represents the initilized value of the list

shape : the shape of the array

default_assumptions = {}

length

val

class pyccel.ast.core.**DottedName**
Bases: *pyccel.ast.basic.Basic*

Represents a dotted variable.

Examples

```
>>> from pyccel.ast.core import DottedName
>>> DottedName('matrix', 'n_rows')
matrix.n_rows
>>> DottedName('pyccel', 'stdlib', 'parallel')
pyccel.stdlib.parallel
```

default_assumptions = {}

name

class pyccel.ast.core.**DottedVariable**
Bases: sympy.core.expr.AtomicExpr, sympy.logic.boolalg.Boolean

Represents a dotted variable.

cls_base

default_assumptions = {}

dtype

lhs

name

names
Return list of names as strings.

rank

rhs

class pyccel.ast.core.**EmptyLine**
Bases: *pyccel.ast.basic.Basic*

Represents a EmptyLine in the code.

text [str] the comment line

Examples

```
>>> from pyccel.ast.core import EmptyLine
>>> EmptyLine()
```

default_assumptions = {}

class pyccel.ast.core.**Enumerate**
Bases: *pyccel.ast.basic.Basic*

Reresents the enumerate stmt

default_assumptions = {}

element

class pyccel.ast.core.**ErrorExit**
Bases: *pyccel.ast.core.Exit*

Exist with error.

default_assumptions = {}

class pyccel.ast.core.**Eval**
Bases: *pyccel.ast.basic.Basic*

Basic class for eval instruction.

default_assumptions = {}

class pyccel.ast.core.**Exit**
Bases: *pyccel.ast.basic.Basic*

Basic class for exists.

default_assumptions = {}

class pyccel.ast.core.**For**
Bases: *pyccel.ast.basic.Basic*

Represents a 'for-loop' in the code.

Expressions are of the form:

"for target in iter: body. . . "

target [symbol] symbol representing the iterator

iter [iterable] iterable object. for the moment only Range is used

body [sympy expr] list of statements representing the body of the For statement.

Examples

```
>>> from sympy import symbols, MatrixSymbol
>>> from pyccel.ast.core import Assign, For
>>> i,b,e,s,x = symbols('i,b,e,s,x')
>>> A = MatrixSymbol('A', 1, 3)
>>> For(i, (b,e,s), [Assign(x,x-1), Assign(A[0, 1], x)])
For(i, Range(b, e, s), (x := x - 1, A[0, 1] := x))
```

body

default_assumptions = {}

insert2body(*stmt*)

iterable

target

class pyccel.ast.core.**ForIterator**
Bases: *pyccel.ast.core.For*

Class that describes iterable classes defined by the user.

default_assumptions = {}

depth

ranges

class pyccel.ast.core.**FunctionCall**
Bases: *pyccel.ast.basic.Basic*

Represents a function call in the code.

arguments

default_assumptions = {}

func

class pyccel.ast.core.**FunctionDef**
Bases: *pyccel.ast.basic.Basic*

Represents a function definition.

name [str] The name of the function.

arguments [iterable] The arguments to the function.

results [iterable] The direct outputs of the function.

body [iterable] The body of the function.

local_vars [list of Symbols] These are used internally by the routine.

global_vars [list of Symbols] Variables which will not be passed into the function.

cls_name: str Class name if the function is a method of cls_name

hide: bool if True, the function definition will not be generated.

kind: str 'function' or 'procedure'. default value: 'function'

is_static: bool True for static functions. Needed for f2py

imports: list, tuple a list of needed imports

decorators: list, tuple a list of proporties

Examples

```
>>> from pyccel.ast.core import Assign, Variable, FunctionDef
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> args        = [x]
>>> results     = [y]
>>> body        = [Assign(y,x+1)]
>>> FunctionDef('incr', args, results, body)
FunctionDef(incr, (x,), (y,), [y := 1 + x], [], [], None, False, function)
```

One can also use parametrized argument, using ValuedArgument

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import Assign
>>> from pyccel.ast.core import FunctionDef
>>> from pyccel.ast.core import ValuedArgument
>>> from pyccel.ast.core import GetDefaultFunctionArg
>>> n = ValuedArgument('n', 4)
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> args        = [x, n]
>>> results     = [y]
>>> body        = [Assign(y,x+n)]
>>> FunctionDef('incr', args, results, body)
FunctionDef(incr, (x, n=4), (y,), [y := 1 + x], [], [], None, False, function, [])
```

arguments

body

cls_name

decorators

default_assumptions = {}

global_vars

header

hide

imports

is_compatible_header(*header*)

Returns True if the header is compatible with the given FunctionDef.

header: Header a pyccel header suppose to describe the FunctionDef

is_procedure

Returns True if a procedure.

is_recursive

is_static

kind

local_vars

name

print_body()

rename(*newname*)
Rename the FunctionDef name by creating a new FunctionDef with newname.

newname: str new name for the FunctionDef

results

set_recursive()

vectorize(*body*, *header*)
return vectorized FunctionDef

class pyccel.ast.core.**FunctionalFor**
Bases: *pyccel.ast.basic.Basic*

.

default_assumptions = {}

index

indexes

loops

target

class pyccel.ast.core.**FunctionalMap**
Bases: *pyccel.ast.core.FunctionalFor*, *pyccel.ast.core.GeneratorComprehension*

default_assumptions = {}

class pyccel.ast.core.**FunctionalMax**
Bases: *pyccel.ast.core.FunctionalFor*, *pyccel.ast.core.GeneratorComprehension*

default_assumptions = {}

name = 'max'

class pyccel.ast.core.**FunctionalMin**
Bases: *pyccel.ast.core.FunctionalFor*, *pyccel.ast.core.GeneratorComprehension*

default_assumptions = {}

name = 'min'

class pyccel.ast.core.**FunctionalSum**
Bases: *pyccel.ast.core.FunctionalFor*, *pyccel.ast.core.GeneratorComprehension*

default_assumptions = {}

name = 'sum'

class pyccel.ast.core.**GeneratorComprehension**
Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

class pyccel.ast.core.**GetDefaultFunctionArg**
Bases: *pyccel.ast.basic.Basic*

Creates a FunctionDef for handling optional arguments in the code.

arg: ValuedArgument, ValuedVariable argument for which we want to create the function returning the default value

func: FunctionDef the function/subroutine in which the optional arg is used

Examples

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import Assign
>>> from pyccel.ast.core import FunctionDef
>>> from pyccel.ast.core import ValuedArgument
>>> from pyccel.ast.core import GetDefaultFunctionArg
>>> n = ValuedArgument('n', 4)
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> args        = [x, n]
>>> results     = [y]
>>> body        = [Assign(y,x+n)]
>>> incr = FunctionDef('incr', args, results, body)
>>> get_n = GetDefaultFunctionArg(n, incr)
>>> get_n.name
get_default_incr_n
>>> get_n
get_default_incr_n(n=4)
```

You can also use **ValuedVariable** as in the following example

```
>>> from pyccel.ast.core import ValuedVariable
>>> n = ValuedVariable('int', 'n', value=4)
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> args        = [x, n]
>>> results     = [y]
>>> body        = [Assign(y,x+n)]
>>> incr = FunctionDef('incr', args, results, body)
>>> get_n = GetDefaultFunctionArg(n, incr)
>>> get_n
get_default_incr_n(n=4)
```

argument

default_assumptions = {}

func

name

class pyccel.ast.core.**If**

Bases: *pyccel.ast.basic.Basic*

Represents a if statement in the code.

args : every argument is a tuple and is defined as (cond, expr) where expr is a valid ast element and cond is a boolean test.

Examples

```
>>> from sympy import Symbol
>>> from pyccel.ast.core import Assign, If
>>> n = Symbol('n')
>>> If(((n>1), [Assign(n,n-1)]), (True, [Assign(n,n+1)]))
If(((n>1), [Assign(n,n-1)]), (True, [Assign(n,n+1)]))
```

bodies

default_assumptions = {}

class pyccel.ast.core.**IfTernaryOperator**
Bases: *pyccel.ast.core.If*

class for the Ternery operator

default_assumptions = {}

class pyccel.ast.core.**Import**
Bases: *pyccel.ast.basic.Basic*

Represents inclusion of dependencies in the code.

target [str, list, tuple, Tuple] targets to import

Examples

```
>>> from pyccel.ast.core import Import
>>> from pyccel.ast.core import DottedName
>>> Import('foo')
import foo
```

```
>>> abc = DottedName('foo', 'bar', 'baz')
>>> Import(abc)
import foo.bar.baz
```

```
>>> Import(['foo', abc])
import foo, foo.bar.baz
```

default_assumptions = {}

source

target

class pyccel.ast.core.**IndexedElement**
Bases: sympy.tensor.indexed.Indexed

Represents a mathematical object with indices.

Examples

```
>>> from sympy import symbols, Idx
>>> from pyccel.ast.core import IndexedVariable
>>> i, j = symbols('i j', cls=Idx)
>>> IndexedElement('A', i, j)
A[i, j]
```

It is recommended that IndexedElement objects be created via IndexedVariable:

```
>>> from pyccel.ast.core import IndexedElement
>>> A = IndexedVariable('A')
>>> IndexedElement('A', i, j) == A[i, j]
False
```

todo: fix bug. the last result must be : True

default_assumptions = {'commutative': True}

dtype

is_commutative = True

order

precision

rank

Returns the rank of the `IndexedElement` object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

class pyccel.ast.core.**IndexedVariable**

Bases: `sympy.tensor.indexed.IndexedBase`

Represents an indexed variable, like x in x[i], in the code.

Examples

```
>>> from sympy import symbols, Idx
>>> from pyccel.ast.core import IndexedVariable
>>> A = IndexedVariable('A'); A
A
>>> type(A)
<class 'pyccel.ast.core.IndexedVariable'>
```

When an IndexedVariable object receives indices, it returns an array with named axes, represented by an IndexedElement object:

```
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'pyccel.ast.core.IndexedElement'>
```

The IndexedVariable constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedVariable('B', shape=(o, p))
>>> B[i, j].shape
(m, n)
```

todo: fix bug. the last result must be : (o,p)

clone(*name*)

default_assumptions = {'commutative': True}

dtype

is_commutative = True

kw_args

name

order

precision

rank

class pyccel.ast.core.**Interface**
Bases: *pyccel.ast.basic.Basic*

Represent an Interface

cls_name

decorators

default_assumptions = {}

functions

global_vars

hide

imports

is_procedure

kind

name

rename(*newname*)

class pyccel.ast.core.**Is**
Bases: *pyccel.ast.basic.Basic*

Represents a is expression in the code.

Examples

```
>>> from pyccel.ast import Is
>>> from pyccel.ast import Nil
>>> from sympy.abc import x
>>> Is(x, Nil())
Is(x, None)
```

default_assumptions = {}

lhs

rhs

class pyccel.ast.core.**Len**
Bases: sympy.core.function.Function

Represents a 'len' expression in the code.

arg

default_assumptions = {}

dtype

class pyccel.ast.core.**List**
Bases: sympy.core.containers.Tuple

Represent lists in the code with dynamic memory management.

default_assumptions = {}

class pyccel.ast.core.**Load**
Bases: *pyccel.ast.basic.Basic*

Similar to 'importlib' in python. In addition, we can also provide the functions we want to import.

module: str, DottedName name of the module to load.

funcs: str, list, tuple, Tuple a string representing the function to load, or a list of strings.

as_lambda: bool load as a Lambda expression, if True

nargs: int number of arguments of the function to load. (default = 1)

Examples

```
>>> from pyccel.ast.core import Load
```

as_lambda

default_assumptions = {}

execute()

funcs

module

nargs

class pyccel.ast.core.**Map**
Bases: *pyccel.ast.basic.Basic*

Reresents the map stmt

default_assumptions = {}

class pyccel.ast.core.**ModOp**
Bases: *pyccel.ast.core.NativeOp*

default_assumptions = {}

class pyccel.ast.core.**Module**
Bases: *pyccel.ast.basic.Basic*

Represents a module in the code. A block consists of the following inputs

variables: list list of the variables that appear in the block.

declarations: list list of declarations of the variables that appear in the block.

funcs: list a list of FunctionDef instances

classes: list a list of ClassDef instances

imports: list, tuple list of needed imports

Examples

```
>>> from pyccel.ast.core import Variable, Assign
>>> from pyccel.ast.core import ClassDef, FunctionDef, Module
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> z = Variable('real', 'z')
>>> t = Variable('real', 't')
>>> a = Variable('real', 'a')
>>> b = Variable('real', 'b')
>>> body = [Assign(y,x+a)]
>>> translate = FunctionDef('translate', [x,y,a,b], [z,t], body)
>>> attributs   = [x,y]
>>> methods     = [translate]
>>> Point = ClassDef('Point', attributs, methods)
>>> incr = FunctionDef('incr', [x], [y], [Assign(y,x+1)])
>>> decr = FunctionDef('decr', [x], [y], [Assign(y,x-1)])
>>> Module('my_module', [], [incr, decr], [Point])
Module(my_module, [], [FunctionDef(incr, (x,), (y,), [y := 1 + x], [], [], None,␣
↪False, function), FunctionDef(decr, (x,), (y,), [y := -1 + x], [], [], None,␣
↪False, function)], [ClassDef(Point, (x, y), (FunctionDef(translate, (x, y, a,␣
↪b), (z, t), [y := a + x], [], [], None, False, function),), [public])])
```

body

classes

declarations

default_assumptions = {}

funcs

imports

interfaces

name

variables

class pyccel.ast.core.**MulOp**

Bases: *pyccel.ast.core.NativeOp*

default_assumptions = {}

class pyccel.ast.core.**NativeOp**

Bases: *pyccel.ast.basic.Basic*

Base type for native operands.

default_assumptions = {}

class pyccel.ast.core.**NewLine**

Bases: *pyccel.ast.basic.Basic*

Represents a NewLine in the code.

text [str] the comment line

Examples

```
>>> from pyccel.ast.core import NewLine
>>> NewLine()
```

default_assumptions = {}

class pyccel.ast.core.**Nil**
Bases: *pyccel.ast.basic.Basic*

class for None object in the code.

default_assumptions = {}

class pyccel.ast.core.**ParallelBlock**
Bases: *pyccel.ast.core.Block*

Represents a parallel block in the code. In addition to block inputs, there is

clauses: list a list of clauses

Examples

```
>>> from pyccel.ast.core import ParallelBlock
>>> from pyccel.ast.core import Variable, Assign, Block
>>> n = Variable('int', 'n')
>>> x = Variable('int', 'x')
>>> body = [Assign(x,2.*n + 1.), Assign(n, n + 1)]
>>> variables = [x,n]
>>> clauses = []
>>> ParallelBlock(clauses, variables, body)
# parallel
x := 1.0 + 2.0*n
n := 1 + n
```

clauses

default_assumptions = {}

prefix

class pyccel.ast.core.**ParallelRange**
Bases: *pyccel.ast.core.Range*

Representes a parallel range using OpenMP/OpenACC.

Examples

```
>>> from pyccel.ast.core import Variable
```

default_assumptions = {}

class pyccel.ast.core.**Pass**
Bases: *pyccel.ast.basic.Basic*

Basic class for pass instruction.

default_assumptions = {}

class pyccel.ast.core.**Pow**
Bases: sympy.core.power.Pow

default_assumptions = {}

class pyccel.ast.core.**Print**
Bases: *pyccel.ast.basic.Basic*

Represents a print function in the code.

expr [sympy expr] The expression to return.

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Print
>>> n,m = symbols('n,m')
>>> Print(('results', n,m))
Print((results, n, m))
```

default_assumptions = {}

expr

class pyccel.ast.core.**Product**

Bases: *pyccel.ast.basic.Basic*

Represents a Product stmt.

default_assumptions = {}

elements

class pyccel.ast.core.**Program**

Bases: *pyccel.ast.basic.Basic*

Represents a Program in the code. A block consists of the following inputs

variables: list list of the variables that appear in the block.

declarations: list list of declarations of the variables that appear in the block.

funcs: list a list of FunctionDef instances

classes: list a list of ClassDef instances

body: list a list of statements

imports: list, tuple list of needed imports

modules: list, tuple list of needed modules

Examples

```
>>> from pyccel.ast.core import Variable, Assign
>>> from pyccel.ast.core import ClassDef, FunctionDef, Module
>>> x = Variable('real', 'x')
>>> y = Variable('real', 'y')
>>> z = Variable('real', 'z')
>>> t = Variable('real', 't')
>>> a = Variable('real', 'a')
>>> b = Variable('real', 'b')
>>> body = [Assign(y,x+a)]
>>> translate = FunctionDef('translate', [x,y,a,b], [z,t], body)
>>> attributs   = [x,y]
>>> methods     = [translate]
>>> Point = ClassDef('Point', attributs, methods)
>>> incr = FunctionDef('incr', [x], [y], [Assign(y,x+1)])
>>> decr = FunctionDef('decr', [x], [y], [Assign(y,x-1)])
>>> Module('my_module', [], [incr, decr], [Point])
Module(my_module, [], [FunctionDef(incr, (x,), (y,), [y := 1 + x], [], [], None,␣
↪False, function), FunctionDef(decr, (x,), (y,), [y := -1 + x], [], [], None,␣
↪False, function)], [ClassDef(Point, (x, y), (FunctionDef(translate, (x, y, a,␣
↪b), (z, t), [y := a + x], [], [], None, False, function),), [public])])
```

body

classes

declarations

default_assumptions = {}

funcs

imports

interfaces

modules

name

variables

class pyccel.ast.core.**PythonFunction**
Bases: *pyccel.ast.core.FunctionDef*

Represents a Python-Function definition.

default_assumptions = {}

rename(*newname*)
Rename the PythonFunction name by creating a new PythonFunction with newname.

newname: str new name for the PythonFunction

class pyccel.ast.core.**Raise**
Bases: *pyccel.ast.basic.Basic*

Represents a raise in the code.

default_assumptions = {}

class pyccel.ast.core.**Random**
Bases: sympy.core.function.Function

Represents a 'random' number in the code.

default_assumptions = {}

seed

class pyccel.ast.core.**Range**
Bases: *pyccel.ast.basic.Basic*

Represents a range.

Examples

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import Range
>>> from sympy import Symbol
>>> s = Variable('int', 's')
>>> e = Symbol('e')
>>> Range(s, e, 1)
Range(0, n, 1)
```

default_assumptions = {}

size

start

step

stop

class pyccel.ast.core.**Return**
Bases: *pyccel.ast.basic.Basic*

Represents a function return in the code.

expr [sympy expr] The expression to return.

stmts :represent assign stmts in the case of expression return

default_assumptions = {}

expr

stmt

class pyccel.ast.core.**SeparatorComment**
Bases: *pyccel.ast.core.Comment*

Represents a Separator Comment in the code.

mark [str] marker

Examples

```
>>> from pyccel.ast.core import SeparatorComment
>>> SeparatorComment(n=40)
# ........................................
```

default_assumptions = {}

class pyccel.ast.core.**Slice**
Bases: *pyccel.ast.basic.Basic*

Represents a slice in the code.

start [Symbol or int] starting index

end [Symbol or int] ending index

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Slice
>>> m, n = symbols('m, n', integer=True)
>>> Slice(m,n)
m : n
>>> Slice(None,n)
 : n
>>> Slice(m,None)
m :
```

default_assumptions = {}

end

start

class pyccel.ast.core.**String**
Bases: *pyccel.ast.basic.Basic*

Represents the String

arg

default_assumptions = {}

class pyccel.ast.core.**SubOp**
Bases: *pyccel.ast.core.NativeOp*

default_assumptions = {}

class pyccel.ast.core.**Subroutine**(**args*, ***kwargs*)
Bases: sympy.core.function.UndefinedFunction

class pyccel.ast.core.**SumFunction**
Bases: *pyccel.ast.basic.Basic*

Represents a Sympy Sum Function.

body: Expr Sympy Expr in which the sum will be performed.

iterator: a tuple that containts the index of the sum and it's range.

body

default_assumptions = {}

iterator

stmts

class pyccel.ast.core.**SymbolicAssign**
Bases: *pyccel.ast.basic.Basic*

Represents symbolic aliasing for code generation. An alias is any statement of the form *lhs := rhs* where

lhs : Symbol

rhs : Range

Examples

```
>>> from sympy import Symbol
>>> from pyccel.ast.core import SymbolicAssign
>>> from pyccel.ast.core import Range
>>> r = Range(0, 3)
>>> y = Symbol('y')
>>> SymbolicAssign(y, r)
```

default_assumptions = {}

lhs

rhs

class pyccel.ast.core.**SymbolicPrint**
Bases: *pyccel.ast.basic.Basic*

Represents a print function of symbolic expressions in the code.

expr [sympy expr] The expression to return.

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Print
>>> n,m = symbols('n,m')
>>> Print(('results', n,m))
Print((results, n, m))
```

default_assumptions = {}

expr

class pyccel.ast.core.**SympyFunction**
Bases: *pyccel.ast.core.FunctionDef*

Represents a function definition.

default_assumptions = {}

rename(*newname*)
Rename the SympyFunction name by creating a new SympyFunction with newname.

newname: str new name for the SympyFunction

class pyccel.ast.core.**Tensor**
Bases: *pyccel.ast.basic.Basic*

Base class for tensor.

Examples

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import Range, Tensor
>>> from sympy import Symbol
>>> s1 = Variable('int', 's1')
>>> s2 = Variable('int', 's2')
>>> e1 = Variable('int', 'e1')
>>> e2 = Variable('int', 'e2')
>>> r1 = Range(s1, e1, 1)
>>> r2 = Range(s2, e2, 1)
>>> Tensor(r1, r2)
Tensor(Range(s1, e1, 1), Range(s2, e2, 1), name=tensor)
```

default_assumptions = {}

dim

name

ranges

class pyccel.ast.core.**Tile**
Bases: *pyccel.ast.core.Range*

Represente a tile.

Examples

```
>>> from pyccel.ast.core import Variable
>>> from pyccel.ast.core import Tile
>>> from sympy import Symbol
>>> s = Variable('int', 's')
>>> e = Symbol('e')
>>> Tile(s, e, 1)
Tile(0, n, 1)
```

default_assumptions = {}

size

start

stop

class pyccel.ast.core.**TupleImport**
Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

imports

class pyccel.ast.core.**ValuedArgument**
Bases: *pyccel.ast.basic.Basic*

Represents a valued argument in the code.

Examples

```
>>> from pyccel.ast.core import ValuedArgument
>>> n = ValuedArgument('n', 4)
>>> n
n=4
```

argument

default_assumptions = {}

name

value

class pyccel.ast.core.**ValuedVariable**
Bases: *pyccel.ast.core.Variable*

Represents a valued variable in the code.

variable: Variable A single variable

value: Variable, or instance of Native types value associated to the variable

Examples

```
>>> from pyccel.ast.core import ValuedVariable
>>> n  = ValuedVariable('int', 'n', value=4)
>>> n
n := 4
```

default_assumptions = {}

value

class pyccel.ast.core.**Variable**
Bases: sympy.core.symbol.Symbol

Represents a typed variable.

dtype [str, DataType] The type of the variable. Can be either a DataType, or a str (bool, int, real).

name [str, list, DottedName] The sympy object the variable represents. This can be either a string or a dotted name, when using a Class attribut.

rank [int] used for arrays. [Default value: 0]

allocatable: False used for arrays, if we need to allocate memory [Default value: False]

shape: int or list shape of the array. [Default value: None]

cls_base: class class base if variable is an object or an object member

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Variable
>>> Variable('int', 'n')
n
>>> Variable('real', x, rank=2, shape=(n,2), allocatable=True)
x
>>> Variable('int', ('matrix', 'n_rows'))
matrix.n_rows
```

allocatable

clone(*name*)

cls_base

cls_parameters

default_assumptions = {}

dtype

inspect()
inspects the variable.

is_ndarray
user friendly method to check if the variable is an ndarray: 1. have a rank > 0 2. dtype is one among {int, bool, real, complex}

is_optional

is_pointer

is_polymorphic

is_target

name

order

precision

rank

shape

class pyccel.ast.core.**Void**
Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

class pyccel.ast.core.**VoidFunction**
Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

class pyccel.ast.core.**While**
Bases: *pyccel.ast.basic.Basic*

Represents a 'while' statement in the code.

Expressions are of the form:

"while test: body. . . "

test [expression] test condition given as a sympy expression

body [sympy expr] list of statements representing the body of the While statement.

Examples

```
>>> from sympy import Symbol
>>> from pyccel.ast.core import Assign, While
>>> n = Symbol('n')
>>> While((n>1), [Assign(n,n-1)])
While(n > 1, (n := n - 1,))
```

body

default_assumptions = {}

test

class pyccel.ast.core.**With**

Bases: *pyccel.ast.basic.Basic*

Represents a 'with' statement in the code.

Expressions are of the form:

"while test: body..."

test [expression] test condition given as a sympy expression

body [sympy expr] list of statements representing the body of the With statement.

Examples

block

body

default_assumptions = {}

settings

test

class pyccel.ast.core.**ZerosLike**

Bases: sympy.core.function.Function

Represents variable assignment using numpy.zeros_like for code generation.

lhs [Expr] Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

rhs [Variable] the input variable

Examples

```
>>> from sympy import symbols
>>> from pyccel.ast.core import Zeros, ZerosLike
>>> n,m,x = symbols('n,m,x')
>>> y = Zeros(x, (n,m))
>>> z = ZerosLike(y)
```

default_assumptions = {}

init_value

lhs

rhs

class pyccel.ast.core.**Zip**
Bases: *pyccel.ast.basic.Basic*

Represents a zip stmt.

default_assumptions = {}

element

pyccel.ast.core.**allocatable_like**(*expr*, *verbose=False*)
finds attributs of an expression

expr: Expr a pyccel expression

verbose: bool talk more

pyccel.ast.core.**atom**(*e*)
Return atom-like quantities as far as substitution is concerned: Functions , DottedVarviables. contrary to _atom we return atoms that are inside such quantities too

pyccel.ast.core.**float2int**(*expr*)

pyccel.ast.core.**get_assigned_symbols**(*expr*)
Returns all assigned symbols (as sympy Symbol) in the AST.

expr: Expression any AST valid expression

pyccel.ast.core.**get_initial_value**(*expr*, *var*)
Returns the first assigned value to var in the Expression expr.

expr: Expression any AST valid expression

var: str, Variable, DottedName, list, tuple variable name

pyccel.ast.core.**get_iterable_ranges**(*it*, *var_name=None*)
Returns ranges of an iterable object.

pyccel.ast.core.**inline**(*func*, *args*)

pyccel.ast.core.**int2float**(*expr*)

pyccel.ast.core.**is_simple_assign**(*expr*)

pyccel.ast.core.**operator**(*op*)
Returns the operator singleton for the given operator

pyccel.ast.core.**subs**(*expr*, *new_elements*)
Substitutes old for new in an expression after sympifying args.

new_elements : list of tuples like [(x,2)(y,3)]

pyccel.ast.datatypes module

class pyccel.ast.datatypes.**CustomDataType**(*name='__UNDEFINED__'*)
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**DataType**
Bases: *pyccel.ast.basic.Basic*

Base class representing native datatypes

default_assumptions = {}

name

pyccel.ast.datatypes.**DataTypeFactory**(*name, argnames=['_name'], BaseClass=<class 'pyccel.ast.datatypes.CustomDataType'>, prefix=None, alias=None, is_iterable=False, is_with_construct=False, is_polymorphic=True*)

class pyccel.ast.datatypes.**FunctionType**(*domains*)
Bases: *pyccel.ast.datatypes.DataType*

codomain

default_assumptions = {}

domain

class pyccel.ast.datatypes.**NativeBool**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeComplex**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeComplexList**
Bases: *pyccel.ast.datatypes.NativeComplex*, *pyccel.ast.datatypes.NativeList*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeGeneric**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeInteger**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeIntegerList**
Bases: *pyccel.ast.datatypes.NativeInteger*, *pyccel.ast.datatypes.NativeList*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeList**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeNil**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeParallelRange**
Bases: *pyccel.ast.datatypes.NativeRange*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeRange**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeReal**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeRealList**
Bases: *pyccel.ast.datatypes.NativeReal*, *pyccel.ast.datatypes.NativeList*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeString**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeSymbol**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeTensor**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**NativeVoid**
Bases: *pyccel.ast.datatypes.DataType*

default_assumptions = {}

class pyccel.ast.datatypes.**UnionType**
Bases: *pyccel.ast.basic.Basic*

args

default_assumptions = {}

class pyccel.ast.datatypes.**VariableType**(*rhs*, *alias*)
Bases: *pyccel.ast.datatypes.DataType*

alias

default_assumptions = {}

pyccel.ast.datatypes.**datatype**(*arg*)
Returns the datatype singleton for the given dtype.

arg [str or sympy expression] If a str ('bool', 'int', 'real','complex', or 'void'), return the singleton for the corresponding dtype. If a sympy expression, return the datatype that best fits the expression. This is determined from the assumption system. For more control, use the *DataType* class directly.

Returns: DataType

pyccel.ast.datatypes.**get_default_value**(*dtype*)
Returns the default value of a native datatype.

pyccel.ast.datatypes.**is_iterable_datatype**(*dtype*)
Returns True if dtype is an iterable class.

pyccel.ast.datatypes.**is_pyccel_datatype**(*expr*)

pyccel.ast.datatypes.**is_with_construct_datatype**(*dtype*)
Returns True if dtype is an with_construct class.

pyccel.ast.datatypes.**sp_dtype**(*expr*)
return the datatype of a sympy types expression

pyccel.ast.datatypes.**str_dtype**(*dtype*)
return a sympy datatype as string dtype: str, Native Type

pyccel.ast.fortran module

class pyccel.ast.fortran.**Ceil**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.fortran.**Dot**
Bases: sympy.core.function.Function

Represents a 'dot' expression in the code.

expr_l: variable first variable

expr_r: variable second variable

default_assumptions = {}

expr_l

expr_r

class pyccel.ast.fortran.**Max**
Bases: sympy.core.function.Function

Represents a 'max' expression in the code.

default_assumptions = {}

class pyccel.ast.fortran.**Min**
Bases: sympy.core.function.Function

Represents a 'min' expression in the code.

default_assumptions = {}

class pyccel.ast.fortran.**Mod**
Bases: sympy.core.function.Function

Represents a 'mod' expression in the code.

default_assumptions = {}

class pyccel.ast.fortran.**Sign**
Bases: sympy.core.basic.Basic

default_assumptions = {}

rhs

pyccel.ast.headers module

class pyccel.ast.headers.**ClassHeader**
Bases: *pyccel.ast.headers.Header*

Represents class header in the code.

name: str class name

options: str, list, tuple a list of options

Examples

```
>>> from pyccel.ast.core import ClassHeader
>>> ClassHeader('Matrix', ('abstract', 'public'))
ClassHeader(Matrix, (abstract, public))
```

default_assumptions = {}

name

options

class pyccel.ast.headers.**FunctionHeader**

Bases: *pyccel.ast.headers.Header*

Represents function/subroutine header in the code.

func: str function/subroutine name

dtypes: tuple/list a list of datatypes. an element of this list can be str/DataType of a tuple (str/DataType, attr, allocatable)

results: tuple/list a list of datatypes. an element of this list can be str/DataType of a tuple (str/DataType, attr, allocatable)

kind: str 'function' or 'procedure'. default value: 'function'

is_static: bool True if we want to pass arrays in f2py mode. every argument of type array will be preceeded by its shape, the later will appear in the argument declaration. default value: False

Examples

```
>>> from pyccel.ast.core import FunctionHeader
>>> FunctionHeader('f', ['double'])
FunctionHeader(f, [(NativeDouble(), [])])
```

create_definition()

Returns a FunctionDef with empy body.

default_assumptions = {}

dtypes

func

is_static

kind

results

to_static()

returns a static function header. needed for f2py

vectorize(*index*)

add a dimension to one of the arguments specified by it's position

class pyccel.ast.headers.**Header**

Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

class pyccel.ast.headers.**InterfaceHeader**

Bases: *pyccel.ast.basic.Basic*

default_assumptions = {}

funcs

name

class pyccel.ast.headers.**MacroFunction**
Bases: *pyccel.ast.headers.Header*

.

apply(*args*, *results=None*)
returns the appropriate arguments.

arguments

default_assumptions = {}

master

master_arguments

name

results

class pyccel.ast.headers.**MacroVariable**
Bases: *pyccel.ast.headers.Header*

.

default_assumptions = {}

master

name

class pyccel.ast.headers.**MetaVariable**
Bases: *pyccel.ast.headers.Header*

Represents the MetaVariable.

default_assumptions = {}

name

value

class pyccel.ast.headers.**MethodHeader**
Bases: *pyccel.ast.headers.FunctionHeader*

Represents method header in the code.

name: iterable method name as a list/tuple

dtypes: tuple/list a list of datatypes. an element of this list can be str/DataType of a tuple (str/DataType, attr)

results: tuple/list a list of datatypes. an element of this list can be str/DataType of a tuple (str/DataType, attr)

kind: str 'function' or 'procedure'. default value: 'function'

is_static: bool True if we want to pass arrays in f2py mode. every argument of type array will be preceeded by its shape, the later will appear in the argument declaration. default value: False

Examples

```
>>> from pyccel.ast.core import MethodHeader
>>> m = MethodHeader(('point', 'rotate'), ['double'])
>>> m
MethodHeader((point, rotate), [(NativeDouble(), [])], [])
```

(continues on next page)

(continued from previous page)

```
>>> m.name
'point.rotate'
```

default_assumptions = {}

dtypes

is_static

kind

name

results

class pyccel.ast.headers.**VariableHeader**
Bases: *pyccel.ast.headers.Header*

Represents a variable header in the code.

name: str variable name

dtypes: dict a dictionary for typing

Examples

default_assumptions = {}

dtypes

name

pyccel.ast.macros module

class pyccel.ast.macros.**Macro**
Bases: sympy.core.expr.AtomicExpr

.

argument

default_assumptions = {}

name

class pyccel.ast.macros.**MacroCount**
Bases: *pyccel.ast.macros.Macro*

.

default_assumptions = {}

class pyccel.ast.macros.**MacroShape**
Bases: *pyccel.ast.macros.Macro*

.

default_assumptions = {}

index

class pyccel.ast.macros.**MacroType**
Bases: *pyccel.ast.macros.Macro*

.

default_assumptions = {}

pyccel.ast.macros.**construct_macro**(*name*, *argument*, *parameter=None*)
.

pyccel.ast.numpyext module

class pyccel.ast.numpyext.**Abs**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Acos**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Acot**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Acsc**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Array**
Bases: sympy.core.function.Function

Represents a call to numpy.array for code generation.

arg : list ,tuple ,Tuple,List

arg

default_assumptions = {}

dtype

fprint(*printer*, *lhs*)
Fortran print.

order

precision

rank

shape

class pyccel.ast.numpyext.**Asec**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Asin**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Atan**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Complex**
Bases: sympy.core.function.Function

Represents a call to numpy.complex for code generation.

arg : Variable, Float, Integer

default_assumptions = {}

dtype

fprint (*printer*)
Fortran print.

imag_part

precision

rank

real_part

shape

class pyccel.ast.numpyext.**Complex128**
Bases: *pyccel.ast.numpyext.Complex*

default_assumptions = {}

class pyccel.ast.numpyext.**Complex64**
Bases: *pyccel.ast.numpyext.Complex*

default_assumptions = {}

precision

class pyccel.ast.numpyext.**Cosh**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Empty**
Bases: *pyccel.ast.numpyext.Zeros*

Represents a call to numpy.empty for code generation.

shape : int or list of integers

default_assumptions = {}

fprint (*printer*, *lhs*)
Fortran print.

class pyccel.ast.numpyext.**Float32**
Bases: *pyccel.ast.numpyext.Real*

default_assumptions = {}

precision

class pyccel.ast.numpyext.**Float64**
Bases: *pyccel.ast.numpyext.Real*

default_assumptions = {}

class pyccel.ast.numpyext.**Int**
Bases: sympy.core.function.Function

Represents a call to numpy.int for code generation.

arg : Variable, Real, Integer, Complex

arg

default_assumptions = {}

dtype

fprint(*printer*)
Fortran print.

precision

rank

shape

class pyccel.ast.numpyext.**Int32**
Bases: *pyccel.ast.numpyext.Int*

default_assumptions = {}

class pyccel.ast.numpyext.**Int64**
Bases: *pyccel.ast.numpyext.Int*

default_assumptions = {}

precision

class pyccel.ast.numpyext.**Log**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Max**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Min**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Ones**
Bases: *pyccel.ast.numpyext.Zeros*

Represents a call to numpy.ones for code generation.

shape : int or list of integers

default_assumptions = {}

init_value

class pyccel.ast.numpyext.**Rand**
Bases: *pyccel.ast.numpyext.Real*

Represents a call to numpy.random.random or numpy.random.rand for code generation.

arg : list ,tuple ,Tuple,List

arg

default_assumptions = {}

fprint(*printer*)
Fortran print.

rank

class pyccel.ast.numpyext.**Real**
Bases: sympy.core.function.Function

Represents a call to numpy.Real for code generation.

arg : Variable, Float, Integer, Complex

arg

default_assumptions = {}

dtype

fprint(*printer*)
Fortran print.

precision

rank

shape

class pyccel.ast.numpyext.**Shape**
Bases: *pyccel.ast.numpyext.Array*

Represents a call to numpy.shape for code generation.

arg : list ,tuple ,Tuple,List, Variable

arg

default_assumptions = {}

dtype

fprint(*printer*, *lhs=None*)
Fortran print.

index

rank

shape

class pyccel.ast.numpyext.**Sinh**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Sqrt**
Bases: *pyccel.ast.core.Pow*

default_assumptions = {}

class pyccel.ast.numpyext.**Sum**
Bases: sympy.core.function.Function

Represents a call to numpy.sum for code generation.

arg : list , tuple , Tuple, List, Variable

arg

default_assumptions = {}

dtype

fprint(*printer*, *lhs=None*)
Fortran print.

rank

class pyccel.ast.numpyext.**Tanh**
Bases: sympy.core.function.Function

default_assumptions = {}

class pyccel.ast.numpyext.**Zeros**
Bases: sympy.core.function.Function

Represents a call to numpy.zeros for code generation.

shape [int, list, tuple] int or list of integers

dtype: str, DataType datatype for the constructed array

Examples

default_assumptions = {}

dtype

fprint(*printer*, *lhs*)
Fortran print.

init_value

order

precision

rank

shape

pyccel.ast.utilities module

pyccel.ast.utilities.**builtin_function**(*expr*, *args=None*)
Returns a builtin-function call applied to given arguments.

pyccel.ast.utilities.**builtin_import**(*expr*)
Returns a builtin pyccel-extension function/object from an import.

Module contents

pyccel.calculus package

Submodules

pyccel.calculus.finite_differences module

pyccel.calculus.finite_differences.**compute_stencil_uniform**(*order*, *n*, *x_value*, *h_value*, *x0=0.0*)
computes a stencil of Order order

order: int derivative order

n: int number of points - 1

x_value: float value of the grid point

h_value: float mesh size

x0: float real number around which we compute the Taylor expansion.

Module contents

pyccel.codegen package

Subpackages

pyccel.codegen.printing package

Submodules

pyccel.codegen.printing.ccode module

class pyccel.codegen.printing.ccode.**CCodePrinter**(*settings={}*)
Bases: *pyccel.codegen.printing.codeprinter.CodePrinter*

A printer to convert python expressions to strings of c code

indent_code(*code*)
Accepts a string of code or a list of code lines

language = 'C'

printmethod = '_ccode'

pyccel.codegen.printing.ccode.**ccode**(*expr*, *assign_to=None*, ***settings*)
Converts an expr to a string of c code

expr [Expr] A sympy expression to be converted.

assign_to [optional] When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision [integer, optional] The precision for numbers such as pi [default=15].

user_functions [dict, optional] A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

dereference [iterable, optional] An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

pyccel.codegen.printing.codeprinter module

class pyccel.codegen.printing.codeprinter.**CodePrinter**(*settings=None*)
Bases: `sympy.printing.str.StrPrinter`

The base class for code-printing subclasses.

doprint(*expr*, *assign_to=None*)
Print the expression as code.

expr [Expression] The expression to be printed.

assign_to [Symbol, MatrixSymbol, or string (optional)] If provided, the printed code will set the expression to a variable with name `assign_to`.

pyccel.codegen.printing.fcode module

class pyccel.codegen.printing.fcode.**FCodePrinter**(*settings={}*)
Bases: *pyccel.codegen.printing.codeprinter.CodePrinter*

A printer to convert sympy expressions to strings of Fortran code

indent_code(*code*)
Accepts a string of code or a list of code lines

language = 'Fortran'

printmethod = '_fcode'

pyccel.codegen.printing.fcode.**fcode**(*expr*, *assign_to=None*, ***settings*)
Converts an expr to a string of c code

expr [Expr] A sympy expression to be converted.

assign_to [optional] When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision [integer, optional] The precision for numbers such as pi [default=15].

user_functions [dict, optional] A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

pyccel.codegen.printing.luacode module

A complete code generator, which uses *lua_code* extensively, can be found in *sympy.utilities.codegen*. The *codegen* module can be used to generate complete source code files.

class pyccel.codegen.printing.luacode.**LuaCodePrinter**(*settings={}*)
Bases: *pyccel.codegen.printing.codeprinter.CodePrinter*

A printer to convert python expressions to strings of Lua code

indent_code(*code*)
Accepts a string of code or a list of code lines

language = 'Lua'

printmethod = '_lua_code'

pyccel.codegen.printing.luacode.**lua_code**(*expr*, *assign_to=None*, ***settings*)
Converts an expr to a string of Lua code

expr [Expr] A sympy expression to be converted.

assign_to [optional] When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision [integer, optional] The precision for numbers such as pi [default=15].

user_functions [dict, optional] A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

dereference [iterable, optional] An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

human [bool, optional] If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

locals: dict A dictionary that contains the list of local symbols. these symbols will be preceeded by local for their first assignment.

Examples

```
>>> from sympy import lua_code, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> lua_code((2*tau)**Rational(7, 2))
'8*1.4142135623731*tau.powf(7_f64/2.0)'
>>> lua_code(sin(x), assign_to="s")
's = x.sin();'
```

Simple custom printing can be defined for certain types by passing a dictionary of {“type” : “function”} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "Abs": [(lambda x: not x.is_integer, "fabs", 4),
...           (lambda x: x.is_integer, "ABS", 4)],
...   "func": "f"
... }
>>> func = Function('func')
>>> lua_code(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'(fabs(x) + x.CEIL()).f()'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(lua_code(expr, tau))
tau = if (x > 0) {
    x + 1
```

(continues on next page)

(continued from previous page)

```
} else {
    x
};
```

Support for loops is provided through `Indexed` types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> lua_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(lua_code(mat, A))
A = [x.powi(2), if (x > 0) {
    x + 1
} else {
    x
}, x.sin()];
```

Module contents

pyccel.codegen.templates package

Subpackages

pyccel.codegen.templates.package package

Submodules

pyccel.codegen.templates.package.make_package module

pyccel.codegen.templates.package.make_package.**make_package**(*input_libraries*, *output_package*)

Repackage contents of multiple static libraries into a single archive.

input_libraries: list list of input files

output_package: string path to the contructed library

pyccel.codegen.templates.package.make_package.**parse_input**()

Module contents

Submodules

pyccel.codegen.templates.main module

Module contents

Submodules

pyccel.codegen.cmake module

class pyccel.codegen.cmake.**CMake**(*path*, *prefix=None*, *flags=None*, *flags_fortran=None*, *compiler_fortran=None*)

Bases: *object*

User-friendly class for cmake.

args

build_path

compiler_fortran

configure(*verbose=True*)

flags

flags_fortran

initialize(*src_dir*, *project*, *suffix*, *libname*, *force=True*)

install()

make(*verbose=False*)

path

prefix

pyccel.codegen.codegen module

class pyccel.codegen.codegen.**Codegen**(*expr*, *name*)

Bases: *object*

Abstract class for code generator.

ast
Returns the AST.

body
Returns the body of the source code, if it is a Program or Module.

classes
Returns the classes if Module.

code
Returns the generated code.

doprint(***settings*)
Prints the code in the target language.

export(*filename=None*)

expr
Returns the AST after Module/Program treatment.

imports
Returns the imports of the source code.

interfaces
Returns the interfaces.

is_module
Returns True if a Module.

is_program
Returns True if a Program.

kind
Returns the kind of the source code: Module, Program or None.

language
Returns the used language

modules
Returns the modules if Program.

name
Returns the name associated to the source code

routines
Returns functions/subroutines.

variables
Returns the variables of the source code.

class pyccel.codegen.codegen.**FCodegen**(*expr*, *name*)
Bases: *pyccel.codegen.codegen.Codegen*

pyccel.codegen.compiler module

class pyccel.codegen.compiler.**Compiler**(*codegen*, *compiler*, *flags=None*, *accelerator=None*, *binary=None*, *debug=False*, *inline=False*, *include=[]*, *libdir=[]*, *libs=[]*, *ignored_modules=[]*)
Bases: *object*

Base class for Code compiler for the Pyccel Grammar

accelerator
Returns the used accelerator

binary
Returns the used binary

codegen
Returns the used codegen

compile(*verbose=False*)
Compiles the generated file.

verbose: bool talk more

compiler
Returns the used compiler

construct_flags()
Constructs compiling flags

debug
Returns True if in debug mode

flags
Returns the used flags

ignored_modules
Returns ignored modules

include
Returns include paths

inline
Returns True if in inline mode

libdir
Returns lib paths

libs
Returns libraries to link with

pyccel.codegen.compiler.**clean**(*filename*)
removes the generated files: .pyccel and .f90

filename: str name of the file to parse.

pyccel.codegen.compiler.**execute_file**(*binary*)
Execute a binary file.

binary: str the name of the binary file

pyccel.codegen.compiler.**get_extension**(*language*)
returns the extension of a given language.

language: str low-level target language used in the conversion

pyccel.codegen.compiler.**make_tmp_file**(*filename*, *output_dir=None*)
returns a temporary file of extension .pyccel that will be decorated with indent/dedent so that textX can find the blocks easily.

filename: str name of the file to parse.

output_dir: str directory to store pyccel file

pyccel.codegen.compiler.**preprocess**(*filename*, *filename_out*)
The input python file will be decorated with indent/dedent so that textX can find the blocks easily. This function will write the output code in filename_out.

filename: str name of the file to parse.

filename_out: str name of the temporary file that will be parsed by textX.

pyccel.codegen.compiler.**preprocess_as_str**(*lines*)
The input python file will be decorated with indent/dedent so that textX can find the blocks easily. This function will write the output code in filename_out.

lines: str or list python code as a string

pyccel.codegen.compiler.**separator**(*n=40*)

Creates a separator string. This is used to improve the readability of the generated code.

n: int length of the separator

pyccel.codegen.utilities module

pyccel.codegen.utilities.**compile_fortran**(*filename*, *compiler*, *flags*, *binary=None*, *verbose=False*, *modules=[]*, *is_module=False*, *libs=[]*, *output=''*)

Compiles the generated file.

verbose: bool talk more

pyccel.codegen.utilities.**construct_flags**(*compiler*, *fflags=None*, *debug=False*, *accelerator=None*, *include=[]*, *libdir=[]*)

Constructs compiling flags for a given compiler.

fflags: str Fortran compiler flags. Default is *-O3*

compiler: str used compiler for the target language.

accelerator: str name of the selected accelerator. One among ('openmp', 'openacc')

debug: bool add some useful prints that may help for debugging.

include: list list of include directories paths

libdir: list list of lib directories paths

pyccel.codegen.utilities.**execute_pyccel**(*filename*, *compiler=None*, *fflags=None*, *debug=False*, *verbose=False*, *accelerator=None*, *include=[]*, *libdir=[]*, *modules=[]*, *libs=[]*, *binary=None*, *output=''*)

Executes the full process: - parsing the python code - annotating the python code - converting from python to fortran - compiling the fortran code.

pyccel.codegen.utilities_old module

pyccel.codegen.utilities_old.**build_cmakelists**(*src_dir*, *libname*, *files*, *force=True*, *libs=[]*, *programs=[]*)

pyccel.codegen.utilities_old.**build_cmakelists_dir**(*src_dir*, *force=True*, *testing=False*)

pyccel.codegen.utilities_old.**build_file**(*filename, language, compiler, execute=False, accelerator=None, debug=False, lint=False, verbose=False, show=False, inline=False, name=None, output_dir=None, ignored_modules=['numpy', 'scipy', 'sympy'], pyccel_modules=[], include=[], libdir=[], libs=[], single_file=True*)

User friendly interface for code generation.

filename: str name of the file to load.

language: str low-level target language used in the conversion

compiler: str used compiler for the target language.

execute: bool execute the generated code, after compiling if True.

accelerator: str name of the selected accelerator. One among ('openmp', 'openacc')

debug: bool add some useful prints that may help for debugging.

lint: bool uses PyLint for static verification, if True.

verbose: bool talk more

show: bool prints the generated code if True

inline: bool set to True, if the file is being load inside a python session.

name: str name of the generated module or program. If not given, 'main' will be used in the case of a program, and 'pyccel_m_${filename}' in the case of a module.

ignored_modules: list list of modules to ignore (like 'numpy', 'sympy'). These modules do not have a correspondence in Fortran.

pyccel_modules: list list of modules supplied by the user.

include: list list of include directories paths

libdir: list list of lib directories paths

libs: list list of libraries to link with

Example

```
>>> from pyccel.codegen import build_file
>>> code = '''
... n = int()
... n = 10
...
... x = int()
... x = 0
... for i in range(0,n):
...     for j in range(0,n):
...         x = x + i*j
... '''
>>> filename = "test.py"
>>> f = open(filename, "w")
>>> f.write(code)
>>> f.close()
>>> build_file(filename, "fortran", "gfortran", show=True, name="main")
========Fortran_Code========
program main
implicit none
integer :: i
integer :: x
integer :: j
integer :: n
n = 10
x = 0
do i = 0, n - 1, 1
    do j = 0, n - 1, 1
        x = i*j + x
    end do
end do
end
============================
```

pyccel.codegen.utilities_old.**build_testing_cmakelists**(*src_dir*, *project*, *files*, *force=True*, *libs=[]*)

pyccel.codegen.utilities_old.**construct_tree**(*filename*, *ignored_modules*)
Constructs a tree of dependencies given a file to process.

pyccel.codegen.utilities_old.**execute_file**(*binary*)
Execute a binary file.

binary: str the name of the binary file

pyccel.codegen.utilities_old.**generate_project_conf**(*srcdir*, *project*, ***settings*)
Generates a conf.py file for the project.

pyccel.codegen.utilities_old.**generate_project_init**(*srcdir*, *project*, ***settings*)
Generates a __init__.py file for the project.

pyccel.codegen.utilities_old.**generate_project_main**(*srcdir*, *project*, *extensions*, *force=True*)

pyccel.codegen.utilities_old.**get_extension_external_path**(*ext*)
Finds the path of a pyccel extension external files.

an external file to pyccel, is any low level code associated to its header(s).

pyccel.codegen.utilities_old.**get_extension_path**(*ext*, *module=None*, *is_external=False*)
Finds the path of a pyccel extension (.py or .pyh). A specific module can also be given.

pyccel.codegen.utilities_old.**get_extension_testing_path**(*ext*)
Finds the path of a pyccel extension tests.

pyccel.codegen.utilities_old.**get_parallel_path**(*ext*, *module=None*)
Finds the path of a pyccel parallel package (.py or .pyh). A specific module can also be given.

pyccel.codegen.utilities_old.**get_stdlib_external_path**(*ext*, *module=None*)
Finds the path of a pyccel stdlib external package (.py or .pyh). A specific module can also be given.

pyccel.codegen.utilities_old.**initialize_project**(*base_dir*, *project*, *libname*, *settings*, *verbose=False*)

pyccel.codegen.utilities_old.**load_extension**(*ext*, *output_dir*, *clean=True*, *modules=None*, *silent=True*, *language='fortran'*, *testing=True*)
Load module(s) from a given pyccel extension.

ext: str a pyccel extension is always of the form pyccelext-xxx where 'xxx' is the extension name.

output_dir: str directory where to store the generated files

clean: Bool remove all tempororay files of extension *pyccel*

modules: list, str a list of modules or a module. every module must be a string.

silent: bool talk more

language: str target language

testing: bool enable unit tests

Examples

```
>>> load_extension('math', 'extensions', silent=False)
>>> load_extension('math', 'extensions', modules=['bsplines'])
>>> load_extension('math', 'extensions', modules='quadratures')
```

pyccel.codegen.utilities_old.**load_module**(*filename, language='fortran', compiler='gfortran', accelerator=None, debug=False, verbose=False, show=False, inline=False, name=None, output_dir=None, ignored_modules=['numpy', 'scipy', 'sympy'], pyccel_modules=[], include=[], libdir=[], libs=[], single_file=True*)

Loads a given filename in a Python session. The file will be parsed, compiled and wrapped into python, using f2py.

filename: str name of the file to load.

language: str low-level target language used in the conversion

compiled: str used compiler for the target language.

Example

```
>>> from pyccel.codegen import load_module
>>> code = '''
... def f(n):
...     n = int()
...     x = int()
...     x = 0
...     for i in range(0,n):
...         for j in range(0,n):
...             x = x + i*j
...     print("x = ", x)
... '''
>>> filename = "test.py"
>>> f = open(filename, "w")
>>> f.write(code)
>>> f.close()
>>> module = load_module(filename="test.py")
>>> module.f(5)
x =          100
```

pyccel.codegen.utilities_old.**mkdir_p**(*dir*)

Module contents

pyccel.commands package

Submodules

pyccel.commands.build module

pyccel.commands.build.**build**(*d, silent=False, force=True, dep_libs=[], dep_extensions=['math'], clean=True, debug=True*)

Generates the project from a dictionary.

pyccel.commands.build.**get_parser**()

pyccel.commands.build.**main**(*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Creates a new pyccel project.

pyccel.commands.build.**mkdir_p**(*dir*)

pyccel.commands.console module

class pyccel.commands.console.**MyParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)

Bases: argparse.ArgumentParser

Custom argument parser for printing help message in case of an error. See http://stackoverflow.com/questions/4042452/display-help-message-with-python-argparse-when-script-is-called-without-any-argu

error(*message: string*)

Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

pyccel.commands.console.**pyccel**(*files=None*, *openmp=None*, *openacc=None*, *output_dir=None*, *compiler='gfortran'*)

pyccel console command.

pyccel.commands.ipyccel module

class pyccel.commands.ipyccel.**IPyccel**(*completekey='tab'*, *stdin=None*, *stdout=None*)

Bases: cmd.Cmd

default(*line*)

this method will catch all commands.

do_let(**args*)

.

do_quit(*args*)

Quits the program.

i_line = 0

intro = "\nPyccel 0.9.1 (default, Nov 23 2017, 16:37:01)\n\nIPyccel 0.0.1 -- An enhanc

precmd(*line*)

prompt = '\x1b[1m\x1b[34mIn [0] \x1b[0m'

pyccel.commands.ipyccel.**ipyccel**()

pyccel.commands.ipyccel.**prompt_in**(*x*)

pyccel.commands.ipyccel.**prompt_out**(*x*)

pyccel.commands.ipyccel.**pyccel_parse**(*code*)

pyccel.commands.quickstart module

pyccel.commands.quickstart.**generate**(*d*, *silent=False*)

Generates the project from a dictionary.

pyccel.commands.quickstart.**get_parser**()

pyccel.commands.quickstart.**main**(*argv=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Creates a new pyccel project.

pyccel.commands.quickstart.**mkdir_p**(*dir*)

Module contents

pyccel.complexity package

Submodules

pyccel.complexity.arithmetic module

pyccel.complexity.arithmetic.**count_ops**(*expr*, *visual=None*)

class pyccel.complexity.arithmetic.**OpComplexity**(*filename_or_text*)

Bases: *pyccel.complexity.basic.Complexity*

class for Operation complexity computation.

cost()

Computes the complexity of the given code.

verbose: bool talk more

pyccel.complexity.basic module

class pyccel.complexity.basic.**Complexity**(*filename_or_text*)

Bases: *object*

Abstract class for complexity computation.

ast

Returns the Abstract Syntax Tree.

cost()

Computes the complexity of the given code.

pyccel.complexity.memory module

pyccel.complexity.memory.**count_access**(*expr*, *visual=True*)

returns the number of access to memory in terms of WRITE and READ.

expr: sympy.Expr any sympy expression or pyccel.ast.core object

visual: bool If `visual` is `True` then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

local_vars: list list of variables that are supposed to be in the fast memory. We will ignore their corresponding memory accesses.

class pyccel.complexity.memory.**MemComplexity**(*filename_or_text*)

Bases: *pyccel.complexity.basic.Complexity*

Class for memory complexity computation. This class implements a simple two level memory model

Example

```
>>> code = '''
... n = 10
... for i in range(0,n):
...     for j in range(0,n):
...         x = pow(i,2) + pow(i,3) + 3*i
...         y = x / 3 + 2* x
... '''
```

```
>>> from pyccel.complexity.memory import MemComplexity
>>> M = MemComplexity(code)
>>> d = M.cost()
>>> print "f = ", d['f']
f =  n**2*(2*ADD + DIV + 2*MUL + 2*POW)
>>> print "m = ", d['m']
m =  WRITE + 2*n**2*(READ + WRITE)
>>> q = M.intensity()
>>> print "+++ computational intensity ~", q
+++ computational intensity ~ (2*ADD + DIV + 2*MUL + 2*POW)/(2*READ + 2*WRITE)
```

Now let us consider a case where some variables are supposed to be in the fast memory, (*r* in this test)

```
>>> code = '''
... n = 10
... x = zeros(shape=(n,n), dtype=float)
... r = float()
... r = 0
... for i in range(0, n):
...     r = x[n,i] + 1
... '''
```

```
>>> M = MemComplexity(code)
>>> d = M.cost()
>>> print "f = ", d['f']
f =  ADD*n
>>> print "m = ", d['m']
m =  2*WRITE + n*(READ + WRITE)
>>> q = M.intensity()
>>> print "+++ computational intensity ~", q
+++ computational intensity ~ ADD/(READ + WRITE)
```

Notice, that this is not what we expect! the cost of writing into *r* is 'zero', and therefor, there should be no $n * WRITE$ in our memory cost. In order to achieve this, you must tell pyccel that you have the variable *r* is already in the fast memory. This can be done by adding the argument *local_vars=['r']* when calling the cost method.

```
>>> d = M.cost(local_vars=['r'])
>>> print "f = ", d['f']
f =  ADD*n
>>> print "m = ", d['m']
m =  READ*n + WRITE
>>> q = M.intensity(local_vars=['r'])
>>> print "+++ computational intensity ~", q
+++ computational intensity ~ ADD/READ
```

and this is exactly what we were expecting.

cost()
Computes the complexity of the given code.

local_vars: list list of variables that are supposed to be in the fast memory. We will ignore their corresponding memory accesses.

intensity(*d=None*, *args=None*, *local_vars=[]*, *verbose=False*)
Returns the computational intensity for the two level memory model.

d: dict dictionary containing the floating and memory costs. if not given, we will compute them.

args: list list of free parameters, i.e. degrees of freedom.

local_vars: list list of variables that are supposed to be in the fast memory. We will ignore their corresponding memory accesses.

verbose: bool talk more

Module contents

pyccel.parser package

Subpackages

pyccel.parser.syntax package

Submodules

pyccel.parser.syntax.basic module

class pyccel.parser.syntax.basic.**BasicStmt**(**kwargs*)
Bases: *object*

Base class for all objects in Pyccel.

declarations
Returns all declarations related to the current statement by looking into the global dictionary declarations. the filter is given by stmt_vars and local_vars, which must be provided by every extension of the base class.

local_vars
must be defined by the statement.

stmt_vars
must be defined by the statement.

update()

pyccel.parser.syntax.headers module

class pyccel.parser.syntax.headers.**ClassHeaderStmt**(**kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a class header statement in the grammar.

expr

class pyccel.parser.syntax.headers.**FunctionHeaderStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a function header statement in the grammar.

expr

class pyccel.parser.syntax.headers.**FunctionMacroStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing an alias function statement in the grammar.

expr

class pyccel.parser.syntax.headers.**Header**(***kwargs*)
Bases: *object*

Class for Header syntax.

class pyccel.parser.syntax.headers.**HeaderResults**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a HeaderResults in the grammar.

expr

class pyccel.parser.syntax.headers.**InterfaceStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

class represent the header interface statement

expr

class pyccel.parser.syntax.headers.**ListType**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a ListType in the grammar.

expr

class pyccel.parser.syntax.headers.**MacroArg**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.headers.**MacroList**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

reresent a MacroList statement

expr

class pyccel.parser.syntax.headers.**MacroStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.headers.**MetavarHeaderStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a metavar header statement in the grammar.

expr

class pyccel.parser.syntax.headers.**StringStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

expr

class pyccel.parser.syntax.headers.**Type**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a header type in the grammar.

expr

class pyccel.parser.syntax.headers.**TypeHeader**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

class pyccel.parser.syntax.headers.**UnionTypeStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

expr

class pyccel.parser.syntax.headers.**VariableHeaderStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Base class representing a header statement in the grammar.

expr

pyccel.parser.syntax.headers.**parse**(*filename=None*, *stmts=None*, *debug=False*)

pyccel.parser.syntax.himi module

class pyccel.parser.syntax.himi.**DeclareFunctionStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.himi.**DeclareTypeStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.himi.**DeclareVariableStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.himi.**FunctionTypeStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

.

expr

class pyccel.parser.syntax.himi.**HiMi**(***kwargs*)
Bases: *object*

Class for HiMi syntax.

pyccel.parser.syntax.himi.**parse**(*filename=None*, *stmts=None*, *debug=False*)

pyccel.parser.syntax.openacc module

class pyccel.parser.syntax.openacc.**AccAsync**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccAtomicConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccAuto**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccBasic**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

class pyccel.parser.syntax.openacc.**AccBind**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCache**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCollapse**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCopy**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCopyin**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCopyout**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccCreate**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDataConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDeclareDirective**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDefault**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDefaultAsync**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDelete**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDevice**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDeviceNum**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDevicePtr**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDeviceResident**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccDeviceType**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccEndClause**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccEnterDataDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccExitDataDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccFinalize**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccFirstPrivate**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccGang**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccHost**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccHostDataDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccIf**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccIfPresent**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccIndependent**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccInitDirective**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccKernelsConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccLink**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccLoopConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccNoHost**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccNumGangs**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccNumWorkers**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccParallelConstruct**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccPresent**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccPrivate**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccReduction**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccRoutineDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccSelf**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccSeq**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccSetDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccShutDownDirective**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccTile**(**kwargs)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccUpdateDirective**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccUseDevice**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccVector**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccVectorLength**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccWait**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccWaitDirective**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**AccWorker**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

class pyccel.parser.syntax.openacc.**Openacc**(***kwargs*)
Bases: *object*

Class for Openacc syntax.

class pyccel.parser.syntax.openacc.**OpenaccStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.openacc.AccBasic*

Class representing a .

expr

pyccel.parser.syntax.openacc.**parse**(*filename=None*, *stmts=None*, *debug=False*)

pyccel.parser.syntax.openmp module

class pyccel.parser.syntax.openmp.**OmpCollapse**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpCopyin**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpDefault**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpEndClause**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpFirstPrivate**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpLastPrivate**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpLinear**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpLoopConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpNumThread**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpOrdered**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpParallelConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpPrivate**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpProcBind**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpReduction**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpSchedule**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpShared**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**OmpSingleConstruct**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

class pyccel.parser.syntax.openmp.**Openmp**(***kwargs*)
Bases: *object*

Class for Openmp syntax.

class pyccel.parser.syntax.openmp.**OpenmpStmt**(***kwargs*)
Bases: *pyccel.parser.syntax.basic.BasicStmt*

Class representing a .

expr

pyccel.parser.syntax.openmp.**parse**(*filename=None*, *stmts=None*, *debug=False*)

Module contents

Submodules

pyccel.parser.errors module

class pyccel.parser.errors.**ErrorInfo**(*filename*, *line=None*, *column=None*, *severity=None*, *message=''*, *symbol=None*, *blocker=False*)

Representation of a single error message.

pyccel.parser.errors.**Errors**()

pyccel.parser.errors.**ErrorsMode**()

exception pyccel.parser.errors.**PyccelCodegenError**

Bases: exceptions.Exception

exception pyccel.parser.errors.**PyccelError**(*message*, *errors=''*)

Bases: exceptions.Exception

exception pyccel.parser.errors.**PyccelSemanticError**

Bases: exceptions.Exception

exception pyccel.parser.errors.**PyccelSyntaxError**

Bases: exceptions.Exception

pyccel.parser.errors.**make_symbol**(*s*)

pyccel.parser.messages module

pyccel.parser.parser module

class pyccel.parser.parser.**Parser**(*inputs*, *debug=False*, *headers=None*, *static=None*, *show_traceback=True*, *output_folder=''*, *context_import_path={}*)

Bases: *object*

Class for a Parser.

annotate(***settings*)

.

append_parent(*parent*)

.

append_son(*son*)

.

ast

blocking

bounding_box

classes

code

create_variable(*expr*, *store=False*)
.

current_class

d_parsers
Returns the d_parsers parser.

dot(*filename*)
Exports sympy AST using graphviz then convert it to an image.

dump(*filename=None*)
Dump the current ast using Pickle.

filename: str output file name. if not given *name.pyccel* will be used and placed in the Pyccel directory ($HOME/.pyccel)

filename

fst

functions

get_class(*name*)
.

get_class_construct(*name*)
Returns the class datatype for name.

get_function(*name*)
.

get_header(*name*)
.

get_macro(*name*)
.

get_python_function(*name*)
.

get_symbolic_function(*name*)
.

get_variable(*name*)
.

get_variables(*source=None*)

headers

imports

insert_class(*cls*)
.

insert_function(*func*)
.

insert_header(*expr*)
.

insert_import(*expr*)
.

insert_macro(*macro*)
.

insert_python_function(*func*)
.

insert_symbolic_function(*func*)
.

insert_variable(*expr*, *name=None*)
.

is_header_file
Returns True if we are treating a header file.

load(*filename=None*)
Load the current ast using Pickle.

filename: str output file name. if not given *name.pyccel* will be used and placed in the Pyccel directory ($HOME/.pyccel)

macros

metavars

namespace

parents
Returns the parents parser.

parse(*d_parsers=None*, *verbose=False*)
converts redbaron fst to sympy ast.

print_namespace()

python_functions

remove_variable(*name*)
.

semantic_done

set_class_construct(*name*, *value*)
Sets the class datatype for name.

set_current_fun(*name*)
.

show_traceback

sons
Returns the sons parser.

static_functions

symbolic_functions

syntax_done

update_variable(*var*, ***options*)
.

variables

view_namespace(*entry*)

class pyccel.parser.parser.**PyccelParser**(*inputs*, *debug=False*, *headers=None*, *static=None*, *show_traceback=True*, *output_folder=''*, *context_import_path={}*)

Bases: *pyccel.parser.parser.Parser*

pyccel.parser.parser.**get_filename_from_import**(*module*, *output_folder=''*, *context_import_path={}*)

Returns a valid filename with absolute path, that corresponds to the definition of module. The priority order is:

- header files (extension == pyh)
- python files (extension == py)

pyccel.parser.parser.**is_ignored_module**(*name*)

pyccel.parser.utilities module

pyccel.parser.utilities.**acc_statement**(*x*)

pyccel.parser.utilities.**accelerator_statement**(*stmt*, *accel*)

Returns stmt if an accelerator statement. otherwise it returns None. this function can be used as the following >>> if accelerator_statement(stmt, 'omp'):

do stuff ...

In general you can use the functions omp_statement and acc_statement

pyccel.parser.utilities.**fst_move_directives**(*x*)

This function moves OpenMP/OpenAcc directives from loop statements to their appropriate parent. This function will have inplace effect. In order to understand why it is needed, let's take a look at the following exampe

```
>>> code = '''
... #$ omp do schedule(runtime)
... for i in range(0, n):
...     for j in range(0, m):
...         a[i,j] = i-j
... #$ omp end do nowait
... '''
>>> from redbaron import RedBaron
>>> red = RedBaron(code)
>>> red
0   '
```

' 1 #$ omp do schedule(runtime) 2 for i in range(0, n):

for j in range(0, m): a[i,j] = i-j

#$ omp end do nowait

As you can see, the statement *#$ omp end do nowait* is inside the For statement, while we would like to have it outside. Now, let's apply our function

```
>>> fst_move_directives(red)
0   #$ omp do schedule(runtime)
1   for i in range(0, n):
        for j in range(0, m):
            a[i,j] = i-j
2   #$ omp end do nowait
```

pyccel.parser.utilities.**get_comments**(*y*)

pyccel.parser.utilities.**get_default_path**(*name*)
this function takes a an import name and returns the path full bash of the library if the library is in stdlib

pyccel.parser.utilities.**get_defs**(*y*)

pyccel.parser.utilities.**get_ifblocks**(*y*)

pyccel.parser.utilities.**get_ifs**(*y*)

pyccel.parser.utilities.**get_loops**(*y*)

pyccel.parser.utilities.**get_withs**(*y*)

pyccel.parser.utilities.**header_statement**(*stmt*, *accel*)
Returns stmt if a header statement. otherwise it returns None. this function can be used as the following >>> if header_statement(stmt):

do stuff ...

pyccel.parser.utilities.**is_valid_filename_py**(*filename*)
Returns True if filename is an existing python file.

pyccel.parser.utilities.**is_valid_filename_pyh**(*filename*)
Returns True if filename is an existing pyccel header file.

pyccel.parser.utilities.**omp_statement**(*x*)

pyccel.parser.utilities.**read_file**(*filename*)
Returns the source code from a filename.

pyccel.parser.utilities.**reconstruct_pragma_multilines**(*header*)
Must be called once we visit an annotated comment, to get the remaining parts of a statement written on multiple lines.

pyccel.parser.utilities.**view_tree**(*expr*)
Views a sympy expression tree.

Module contents

pyccel.stdlib package

Subpackages

pyccel.stdlib.external package

Submodules

pyccel.stdlib.external.dfftpack module

pyccel.stdlib.external.fitpack module

pyccel.stdlib.external.lapack module

pyccel.stdlib.external.mpi4py module

Module contents

pyccel.stdlib.internal package

Module contents

pyccel.stdlib.parallel package

Submodules

pyccel.stdlib.parallel.mpi module

pyccel.stdlib.parallel.openacc module

class pyccel.stdlib.parallel.openacc.**Parallel**(*Async=None*, *wait=None*, *num_gangs=None*, *num_workers=None*, *vector_length=None*, *device_type=None*, *If=None*, *reduction=None*, *copy=None*, *copyin=None*, *copyout=None*, *create=None*, *present=None*, *deviceptr=None*, *private=None*, *firstprivate=None*, *default=None*)

Bases: *object*

class pyccel.stdlib.parallel.openacc.**Range**(*start*, *stop*, *step*, *collapse=None*, *gang=None*, *worker=None*, *vector=None*, *seq=None*, *auto=None*, *tile=None*, *device_type=None*, *independent=None*, *private=None*, *reduction=None*)

Bases: *object*

class pyccel.stdlib.parallel.openacc.**StopIteration**
Bases: *object*

pyccel.stdlib.parallel.openmp module

class pyccel.stdlib.parallel.openmp.**Parallel**(*num_threads=None*, *if_test=None*, *private=None*, *firstprivate=None*, *shared=None*, *reduction=None*, *default=None*, *copyin=None*, *proc_bind=None*)
Bases: *object*

class pyccel.stdlib.parallel.openmp.**Range**(*start*, *stop*, *step*, *nowait=None*, *collapse=None*, *private=None*, *firstprivate=None*, *lastprivate=None*, *reduction=None*, *schedule=None*, *ordered=None*, *linear=None*)
Bases: *object*

class pyccel.stdlib.parallel.openmp.**StopIteration**
Bases: *object*

Module contents

Submodules

pyccel.stdlib.stdlib module

Module contents

pyccel.symbolic package

Module contents

Submodules

pyccel.decorators module

pyccel.decorators.**lambdify**(*f*)

pyccel.decorators.**python**(*f*)

pyccel.decorators.**sympy**(*f*)

pyccel.decorators.**types**(**args*, ***kw*)

pyccel.epyccel module

class pyccel.epyccel.**ContextPyccel**(*name*, *context_folder=''*, *output_folder=''*)
Bases: *object*

Class for interactive use of Pyccel. It can be used within an IPython session, Jupyter Notebook or ipyccel command line.

compile(***settings*)
Convert to Fortran and compile the context.

constants

folder

functions

imports
Returns available imports from the context as a string.

insert_constant(*d*, *value=None*)
Inserts constants in the namespace.

d: str, dict an identifier string or a dictionary of the form {'a': value_a, 'b': value_b} where *a* and *b* are the constants identifiers and value_a, value_b their associated values.

value: int, float, complex, str value used if d is a string

insert_function(*func*, *types*, *kind='function'*, *results=None*)
Inserts a function in the namespace.

name

os_folder

pyccel.epyccel.**clean_extension_module**(*ext_mod*, *py_mod_name*)
Clean Python extension module by moving functions contained in f2py's "mod_[py_mod_name]" automatic attribute to one level up (module level). "mod_[py_mod_name]" attribute is then completely removed from the module.

ext_mod [types.ModuleType] Python extension module created by f2py from pyccel-generated Fortran.

py_mod_name [str] Name of the original (pure Python) module.

pyccel.epyccel.**compile_fortran**(*source*, *modulename*, *extra_args=''*, *libs=[]*, *compiler=None*, *mpi=False*, *includes=[]*)
use f2py to compile a source code. We ensure here that the f2py used is the right one with respect to the python/numpy version, which is not the case if we run directly the command line f2py ...

pyccel.epyccel.**epyccel**(*func*, *inputs=None*, *verbose=False*, *modules=[]*, *libs=[]*, *name=None*, *context=None*, *compiler=None*, *mpi=False*, *static=None*)
Pyccelize a python function and wrap it using f2py.

func: function, str a Python function or source code defining the function

inputs: str, list, tuple, dict inputs can be the function header as a string, or a list/tuple of strings or the globals() dictionary

verbose: bool talk more

modules: list, tuple list of dependencies

libs: list, tuple list of libraries

name: str name of the function, if it is given as a string

context: ContextPyccel, list/tuple a Pyccel context for user defined functions and other dependencies needed to compile func. it also be a list/tuple of ContextPyccel

static: list/tuple a list of 'static' functions as strings

Examples

The following example shows how to use Pyccel within an IPython session

```
>>> #$ header procedure static f_static(int [:]) results(int)
>>> def f_static(x):
>>>     y = x[0] - 1
>>>     return y
```

```
>>> from test_epyccel import epyccel
>>> f = epyccel(f_static, globals()) # appending IPython history
```

```
>>> header = '#$ header procedure static f_static(int [:]) results(int)'
>>> f = epyccel(f_static, header) # giving the header explicitly
```

Now, **f** is a Fortran function that has been wrapped. It is compatible with numpy and you can call it

```
>>> import numpy as np
>>> x = np.array([3, 4, 5, 6], dtype=int)
>>> y = f(x)
```

You can also call it with a list instead of numpy arrays

```
>>> f([3, 4, 5])
2
```

pyccel.epyccel.**epyccel_mpi**(*mod*, *comm*, *root=0*)

Collective version of epyccel for modules: root process generates Fortran code, compiles it and creates a shared library (extension module), which is then loaded by all processes in the communicator.

mod [types.ModuleType] Python module to be pyccelized.

comm: mpi4py.MPI.Comm MPI communicator where extension module will be made available.

root: int Rank of process responsible for code generation.

fmod [types.ModuleType] Python extension module.

pyccel.epyccel.**get_source_function**(*func*)

Module contents

CHAPTER 3

Technical contents

3.1 Pyccel FAQ

This is a list of Frequently Asked Questions about Pyccel. Feel free to suggest new entries!

3.1.1 How do I...

3.2 Glossary

configuration directory The directory containing `conf.py`. By default, this is the same as the *source directory*, but can be set differently with the **-c** command-line option.

directive A reStructuredText markup element that allows marking a block of content with special meaning. Directives are supplied not only by docutils, but Sphinx and custom extensions can add their own.

document name Since reST source files can have different extensions (some people like `.txt`, some like `.rst` – the extension can be configured with `source_suffix`) and different OSes have different path separators, Sphinx abstracts them: *document names* are always relative to the *source directory*, the extension is stripped, and path separators are converted to slashes. All values, parameters and such referring to "documents" expect such document names.

Examples for document names are `index`, `library/zipfile`, or `reference/datamodel/types`. Note that there is no leading or trailing slash.

environment A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

master document The document that contains the root `toctree` directive.

object The basic building block of Sphinx documentation. Every "object directive" (e.g. `function` or `object`) creates such a block; and most objects can be cross-referenced to.

role A reStructuredText markup element that allows marking a piece of text.

source directory The directory which, including its subdirectories, contains all source files for one Sphinx project.

application directory The directory which contains Pyccel sources, as a package.

build directory The build directory as you may specify it for cmake.

Pyccel alpha Pyccel alpha version

Pyccel beta Pyccel beta release version

Pyccel omicron Pyccel release version for OOP

Pyccel lambda Pyccel release version for Functional Programming

Pyccel restriction Denotes a restriction of Python by Pyccel

TODO

3.3 Changes in Pyccel

TODO

3.4 Pyccel authors

Pyccel is written and maintained by Ratnani Ahmed <ratnaniahmed@gmail.com>.

Other co-maintainers:

-

Other contributors, listed alphabetically, are:

-

Many thanks for all contributions!

CHAPTER 4

Indices and tables

- modindex
- *Glossary*

Python Module Index

p