Shorthand Proposal

This is the simplest proposal, which includes support for ParamSpec and Concatenate

Examples:

`() -> bool`	`Callable[[], bool]`
`(int) -> bool`	`Callable[[int], bool]`
`(int, str) -> bool`	`Callable[[int, str], bool]`
`(int, str,) -> bool`	`Callable[[int, str], bool]`
`async (int) -> bool`	`Callable[[int], Awaitable[bool]]`
`(...) -> bool`	`Callable[..., bool]`
`(**P) -> bool`	`Callable[P, bool]`
`(int, str, **P,) -> bool`	`Callable[Concatenate[int, str, P], bool]`
`(int) -> (str) -> bool`	`Callable[[int], Callable[[str], bool]]`
`(int) -> bool | str`	`Callable[[int], bool | str]`

A disallowed example:

```
(**P, int) -> bool
```

One possible grammar:

```
expression:
    | callable_type_expression
    | other_expression_forms (from the python grammar)

callable_type_expression:
    | shorthand_callable_type_expression

shorthand_callable_type_expression:
    | [ ASYNC ] callable_parameters '->' expression

callable_parameters:
    | '(' ')'
    | '(' param_spec ')'
    | '(' expression [callable_parameters_tail]  ')'

callable_parameters_tail:
    | ','
    | ',' param_spec
    | ',' expression [callable_parameters_tail]

param_spec:
    | '**' NAME [',']
```

sanity check of grammar (sadly in a different syntax, and one that doesn't support whitespace).

See the end of the doc for a more idiomatic PEG grammar.

Because `expression` is a parent of `bitwise_or` in the existing grammar, the bitwise or syntax now used for union types will bind more tightly than the arrow in callable type expressions. And because `...` is a valid expression, it is legal to write `(...) -> bool`.

Xor syntax: allowing either shorthand or a def-like syntax

The XOR proposal would allow both the shorthand syntax above as well as a full def-style syntax that can handle named, default, and variadic arguments. The proposed semantics for the def-style types would be identical to existing Callback protocol semantics.

Note that this means any type specifying a non-positional-only argument can only be implemented by functions with identical parameter names. This was the main reason that the *typing-sig* community agreed def-style alone would probably cause too many problems.

Examples of def-style:

```
(x: int, /) -> bool
(x: int, y: str = ..., /) -> bool
(x: int, y: str = ... *, **kwargs: Any) -> bool
```

Unfortunately we discovered that in an XOR proposal, def-style declarations have to make annotations required (i.e. to disallow implicit `Any`). Otherwise the type

```
(int) -> bool
```

would be ambiguous; it could be parsed as shorthand for `Callable[[int], bool]` or alternatively as a function with a named argument `int` of type `Any` returning `bool`.

As a result, the grammar requires duplicating the existing `params` form in entirety, except with annotations required. This adapted PEG grammar should work:

```
def_style_callable_type_expression:
    | [ ASYNC ] '(' annotated_params ')' '->' expression

annotated_params:
    | annotated_parameters

annotated_parameters:
    | slash_no_default param_no_default* annotated_param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

slash_no_default:
    | annotated_param_no_default+ '/' ','
    | annotated_param_no_default+ '/' &')'
slash_with_default:
    | annotated_param_no_default* annotated_param_with_default+ '/' ','
    | annotated_param_no_default* annotated_param_with_default+ '/' &')'

star_etc:
    | '*' annotated_param_no_default annotated_param_maybe_default* [kwds]
    | '*' ',' annotated_param_maybe_default+ [kwds]
    | kwds
kwds: '**' annotated_param_no_default


annotated_param_no_default:
    | annotated_param ','
    | annotated_param  &')'
```

```
annotated_param_with_default:
    | annotated_param default ','
    | annotated_param default  &')'
annotated_param_maybe_default:
    | annotated_param default? ','
    | annotated_param default?  &')'
annotated_param: NAME annotation

annotation:
    | ':' expression
```

With this, we can take

```
callable_type_expression:
    | shorthand_callable_type_expression
    | def_style_callable_type_expression
```

Extended Shorthand Syntax

Both the shorthand and def-style proposals are well-defined, with agreement on typing-sig as to exactly the potential syntax and the semantic interpretation. Extended shorthand is more complicated with many possible variations. Here I'll specify one possible form of extended shorthand syntax.

The shorthand examples from before should all work unchanged:

```
  () -> bool                          Callable[[], bool]
  (int) -> bool                       Callable[[int], bool]
  (int, str) -> bool                  Callable[[int, str], bool]
  (int, str,) -> bool                 Callable[[int, str], bool]
  async (int) -> bool                 Callable[[int], Awaitable[bool]]
  (...) -> bool                       Callable[..., bool]
  (**P) -> bool                       Callable[P, bool]
  (int, str, **P,) -> bool            Callable[Concatenate[int, str, P], bool]
  (int) -> (str) -> bool              Callable[[int], Callable[[str], bool]]
```

Furthermore, the following types (not expressible using the existing `Callable`)

```
  # two positional-only arguments, the second having a default
  (int, str = ...) -> bool

  # a positional only argument and an (optionally) named argument
  (int, x: str) -> bool

  # a positional only argument and a keyword-only argument with default
  (int, *, x: str = ...) -> bool

  # a positional only argument, variadic args, and a keyword-only arg
  (int, *args: float, x: str) -> bool

  # a named argument and variadic positional + keyword args
  (x: int, *args: float, **kwargs: Any) -> bool
```

There is no '/' for positional-only arguments, they are expressed using bare types with no name.

It's open to discussion whether we allow values for default or only ellipses, I'll go with the latter for now because it makes the syntax self-contained.

Here's a syntax, again in the BNF subset of python's PEG grammar:

```
callable_parameters:
    | '(' ')'
    | '(' '...' ')'
    | '(' param_spec ')'
    | '(' expression [callable_parameters_positional]  ')'
    | '(' expression '=' '...' [callable_parameters_positional_default]  ')'
    | '(' NAME annotation [callable_parameters_named]  ')'
    | '(' NAME annotation '=' '...' [callable_parameters_named_default]  ')'
    | '(' '*' [callable_parameters_keyword]  ')'
    | '(' '*' NAME annotation [callable_parameters_keyword_default]  ')'

callable_parameters_positional:
    | ',' expression [callable_parameters_positional]
    | callable_parameters_named
    | callable_parameters_positional_default


callable_parameters_positional_default:
    | ',' expression '=' '...' [callable_parameters_positional_default]
    | callable_parameters_named

callable_parameters_named:
    | ',' NAME ':' expression [callable_parameters_named]
    | callable_parameters_named_default
    | args_or_star callable_parameters_keyword

callable_parameters_named_default:
    | ',' NAME annotation '=' '...' [callable_parameters_named]
    | args_or_star callable_parameters_keyword_default

callable_parameters_keyword:
    | ',' NAME annotation [callable_parameters_keyword]
    | callable_parameters_keyword_default

callable_parameters_keyword_default:
    | ','
    | ',' param_spec
    | ',' kwargs
    | ',' NAME annotation  '=' '...' [callable_parameters_keyword_default]

args_or_star
    | '*'
    | '*' NAME annotation

kwargs:
    | '**' NAME annotation [',']

param_spec:
    | '**' NAME [',']
```

```
annotation:
    | ':' expression

expression:
    | callable_type_expression
    | other_expression_forms (from the python grammar)
```

sanity check of syntax (out of date - this checks a simpler syntax with no defaults)

ParamSpec and kwargs both use **. How do we handle it?

In this proposal, I reserve a bare splat like `**P` as indicating use of a `ParamSpec`. We require variadic arguments to have a name even though the name isn't actually relevant to the type, e.g.

```
(x: int, *args: float, **kwargs: Any) -> bool
```

An alternative proposal is to simply splat the type, e.g.

```
(x: int, *float, **Any) -> bool
```

We dislike this option for several reasons:

- It doesn't respect the existing python grammar very well:
    - The existing `type_expression` form explicitly ignores * and ** prefixes
    - So assuming that in a real grammar we use `type_expression` instead of just `expression` in our annotations (I elided this detail in my snippets), we would need significant grammar changes in order to directly apply * and ** to type expressions.
- We know from Pradeep's statistics that in practice, callback types relying on `ParamSpec` will probably be 4-5 times more common than types dealing with `**kwargs`
    - So, I prefer to use the more concise option for the more common use case
    - Since a ParamSpec is always a NAME the above grammar compatibility is not an issue
- I would guess that explicit `*args: type` and `**kwargs: type` is probably more user-friendly anyway

Possible extension of shorthand syntax to PEP 646 TypeVarTuples

Examples:

```
(*Ts) -> bool
(int, *Ts,) -> bool
(int, *Ts, str) -> bool
```

In PEP 646, it is not required that we allow multiple `TypeVarTuples` to be splat into the same Callable. Type checkers may reject such code, but it parses.

If we want to preserve this behavior, something like the following change would work:

```
callable_parameters:
    | '(' ')'
    | '(' param_spec ')'
    | '(' positional_param [callable_parameters_tail]  ')'

callable_parameters_tail:
```

```
    | ','
    | ',' '**' param_spec  [',']
    | ',' positional_param [callable_type_arguments_tail]


positional_param:
    | type
    | '*' NAME
```

Idiomatic PEG grammars for shorthand and extended shorthand

I wrote the shorthand and extended shorthand syntax in extended BNF so I could easily check them. In an actual PEP we would presumably use the lookahead operator instead.

Shorthand syntax might look something like this:

```
annotation:
    | ':' expression

expression:
    | callable_type_expression
    | other_expression_forms (from the python grammar)

callable_type_expression
    | shorthand_callable_type_expression

shorthand_callable_type_expression:
    | [ ASYNC ] callable_parameters '->' expression

callable_parameters:
    | '(' ')'
    | '(' '...' ')'
    | '(' positional_parameter* [param_spec]  ')'

positional_parameter:
    | expression ','
    | expression &')'

param_spec:
    | '**' NAME ','
    | '**' NAME &')'
```

Extended shorthand syntax might look like:

```
annotation:
    | ':' expression

expression:
    | callable_type_expression
    | other_expression_forms (from the python grammar)

callable_parameters:
    | '(' ')'
    | '(' '...' ')'
```

```
    | '(' callable_parameters_positional ')'

callable_parameters_positional:
    | positional_parameter* callable_parameters_positional_default
    | positional_parameter* callable_parameters_named

callable_parameters_positional_default:
    | positional_parameter* callable_parameters_named_default

callable_parameters_named:
    | named_parameter* callable_parameters_named_default
    | named_parameter* callable_parameters_keyword
    | named_parameter* args_or_star callable_parameters_keyword

callable_parameters_named_default:
    | named_parameter* callable_parameters_keyword_default
    | named_parameter* args_or_star callable_parameters_keyword_default

callable_parameters_keyword:
    | named_parameter* callable_parameters_keyword_default

callable_parameters_keyword_default
    | named_default_parameter* [param_spec]
    | named_default_parameter* kwargs

positional_parameter
    | expression ','
    | expression &')'

positional_parameter_default
    | expression '=' '...' ','
    | expression '=' '...' &')'

named_parameter
    | NAME annotation ','
    | NAME annotation &')'

named_parameter_default
    | NAME annotation '=' '...' ','
    | NAME annotation '=' '...' &')'

args_or_star
    | '*'
    | '*' NAME annotation

kwargs:
    | '**' NAME annotation [',']

param_spec:
    | '**' NAME [',']
```