```
-- ASDL's 5 builtin types are:
-- identifier, int, string, object, constant

module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
                             string? type_comment)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string? type_comment)
          | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)
```

```
      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue

      -- XXX Jython will be different
      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int? conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
```

```
        | ExtSlice(slice* dims)
        | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)

    type_ignore = TypeIgnore(int lineno, string tag)
}
```