From Syntax to Semantics

- The parser analyses the Syntax, ensuring that the raw text that forms the input program is syntactically well-formed.

- In the Semantic Analysis we check if a syntactically well-formed program is also semantically well-formed.

We check if the program has a well-defined meaning.
Syntax vs. Semantic Error

**ChocoPy programs with Syntax Errors**

```
def foo():
    4 + 3
  ^
```

```
def foo():
    4 + 3
  ---------^   4 + plus 3
```

```
def foo():
    4 + 3
  -------^   4 + +
```

If a program has a syntax error, we cannot build a valid AST for it!

**ChocoPy programs with Semantic Errors**

```
def foo():
x + 3
```

```
def foo():
"4" + 3
```

```
def foo():
4 = 3
```

```
def foo():
foo(3)
```

```
def foo():
x: int = 4
```

```
def foo():
x = "3"
```

- x not declared
- Can't add str and int
- Can't assign to a literal
- foo expects no argument
- Can't assign str to int variable
Programs with Semantic Errors have no meaning!

- We all have an intuition of what this program should \textit{mean}:
  
  \begin{verbatim}
  def add(x: int, y: int) -> int:
      return x + y
  \end{verbatim}

- Our intuition can be mathematically formalized with an \textit{operational semantics}

- Eventually we want to generate instructions corresponding to the operational semantics, here to perform an \texttt{add} instruction on two integer values.

- If our program has semantic errors, it has no operational semantics, and we do not know what instructions to generate.

  These programs have no meaning!
Q: How to detect Semantic Errors? A: Semantic Analysis

We are going to look at three different Semantic Analysis each checking for another kind of Semantic Error:

1. **Assign Target Analysis**
   - Check that the left-hand side of an assignment is a valid target.

2. **Name Analysis**
   - Check that all names (of variables and functions) are declared before they are used.

3. **Type Analysis**
   - Check that the program is well-typed given a set of typing rules.
Semantic Analysis as AST Tree Traversals

Each semantic analysis is implemented as a pass traversing the AST and checking for semantic errors.

To help implement semantic analysis passes, we first implement a generic AST traversal.
AST Traversal in xDSL

- Reminder: in xDSL all AST nodes are represented as Operations
- The nested tree structure is achieved by Regions

```cpp
choco.ast.binary_expr() ["op" = "+"] {
  choco.ast.literal() ["value" = 3 : !i32]
}
{
  choco.ast.binary_expr() ["op" = "+"] {
    choco.ast.literal() ["value" = 4 : !i32]
  }
  choco.ast.literal() ["value" = 5 : !i32]
}
```
First simple AST Visitor

One class with two functions:

- **traverse** to iterate over the tree nodes
- **visit** is called once per AST node, should be overloaded by subclass

xDSL makes writing an AST visitor super easy!

class Visitor:

    def traverse(self, operation: Operation):
        for r in operation.regions:
            for op in r.ops:
                self.traverse(op)

                self.visit(operation)

    def visit(self, operation: Operation):
        pass
Simple Printer with AST Visitor

- Print name of each operation in the AST.
- SimplePrinter is a subclass of Visitor and overloads the visit method

```python
class Visitor:
    def traverse(self, operation: Operation): ...  
    def visit(self, operation: Operation): pass

class SimplePrinter(Visitor):
    def visit(self, operation: Operation):
        print(operation.name)  # print operation name

SimplePrinter().traverse(BinaryExpr.get( ... ))
```
A better AST Visitor

What if we only want to visit AST nodes of a certain type?

Idea: have separate visit methods for each AST node type!

class Visitor:
    def traverse(self, operation: Operation):
        for r in operation.regions:
            ...

        if isinstance(operation, BinaryExpr):
            self.visit_binary_expr(operation)
        elif isinstance(operation, Literal):
            self.visit_literal(operation)

    def visit_binary_expr(self, e: BinaryExpr): pass
    def visit_literal(self, l: Literal): pass
A generic better AST Visitor

Use Python dynamic reflection features to avoid boilerplate code:

class Visitor:
    def traverse(self, op: Operation):
        # get class name of operation in snake_case
        op_class_name = camel_to_snake(type(op).__name__)
        for r in op.regions:
            ...
        # check if subclass has implemented a method with name visit_op_class_name
        # return method if it exists; otherwise None is returned
        visit = get_method(self, f"visit_{op_class_name}")
        if visit:
            visit(op)  # if the visit_op_class_name method exists call it
A flexible AST Visitor

What if we want to influence the traversal of certain AST nodes?

**Idea:** allow subclasses to implement `traverse_class_name` methods and prefer them over the generic traversal!

```python
class Visitor:
    def traverse(self, op: Operation):
        class_name = camel_to_snake(type(op).__name__)
        traverse = get_method(self, f"traverse_{class_name}"")
        if traverse:  # if a traverse_class_name method exists call it
            traverse(operation)
        else:  # otherwise do the generic traversal
            for r in op.regions:
                ...
        visit = get_method(self, f"visit_{class_name}"")
        if visit:
            visit(op)
```
Assign Target Analysis

- The grammar from CW1 allows for arbitrary expressions on the left-hand side of an assignment, but this allows for example:
  \[ 4 = x + 1 \]

- Assign Target Analysis
  - Check that left-hand side of all assignments is either:
    - a variable name; or
      \[ x = 4 + 5 \]
    - an index into a list
      \[ x[\theta] = 4 + 5 \]
class AssignVisitor(Visitor):
    # visit every assign AST node
    def visit_assign(self, assign: Assign):
        # select the target operation
        target_op = assign.target.ops[0]
        # check if it is a variable name or an index expression
        if isinstance(target_op, ExprName):
            return
        if isinstance(target_op, IndexExpr):
            return
        # if not: raise a Semantic Error
        raise SemanticError(
            f'Found {type(target_op).__name__} as the left-hand side of an assignment. '
            f'Expected to find variable name or index expression only.')

AssignVisitor().traverse(module)
return module
Name Analysis

- Check names of variables and functions are declared before they are used
- We need to remember what names have been declared
  - For this we construct a context (aka, environment) that reflects the scopes in the program

```python
1 x: int = 4
2 def foo(x: int):
3     print(x)
4 def bar():
5     y: int = 0
6     y = x * x
7     print(y)
8
9 foo(5)
10 bar()
```

```
CtxType = Dict[str, Optional['CtxType']]
ctx: CtxType = {
    "x": None, # variable from line 1
    "foo": {
        "x": None  # parameter from line 2
    },
    "bar": {
        "y": None # variable from line 5
    },
}
```
Scopes

**Definition**
The *scope of an identifier* is the part of the program where that identifier is valid.

- It is *only legal* to refer to an identifier within their scope.
- It is *illegal* to declare two identifiers with the same name and the same scope.
- It is *legal* to declare a variable in a nested scope, this then *shadows* the identifier in the outer scope which can no longer be accessed.
- Variables that are not declared inside a function have *global scope*. 
Name Analysis

- Can we construct the scoping context while we traverse the AST?
Name Analysis

- Can we construct the scoping context while we traverse the AST?
- **No!** Consider for example:

```python
def foo():
    bar()

def bar():
    foo()
```

- To check `foo`, we need to know that `bar` is a valid name
- To check `bar`, we need to know that `foo` is a valid name

We implement Name Analysis by traversing the AST twice, first to build the context and then a second time for checking.
classBuildContextVisitor(Visitor):
    name_ctx: NameCtx  # class to manage the name context
    # for every variable definition
    def visit_var_def(self, var_def: VarDef):  # add variable name to the current name context
        self.name_ctx.add_var(var_def.typed_var.ops[0].var_name.data)
Name Analysis: Part I - Construct the Name Context

classBuildContextVisitor(Visitor):
    name_ctx: NameCtx  # class to manage the name context
    # for every variable definition
    def visit_var_def(self, var_def: VarDef):  # add variable name to the current name context
        self.name_ctx.add_var(var_def.typed_var.ops[0].var_name.data)
    # for every function definition
    def traverse_func_def(self, func_def: FuncDef):
        # prepare a visitor for the function body ...
        body_visitor =(BuildContextVisitor(NameCtx(parent_scope=self.name_ctx))
        # ... add the function parameters to the nested name scope ...
        for op in func_def.params.ops:  body_visitor.name_ctx.add_var(op.var_name.data)
        # ... visit the function body to construct the nested name scope.
        for op in func_def.func_body.ops:  body_visitor.traverse(op)
        # finally, add function and nested scope to the current name context
        self.name_ctx.add_func(func_def.func_name.data, body_visitor.name_ctx)
1. Check that variables are declared before they are used

```python
class NameAnalysisVisitor(Visitor):
    name_ctx: NameCtx

    def visit_expr_name(self, expr_name: ExprName):
        if expr_name.id.data in self.name_ctx:
            return
        else:
            raise SemanticError(
                f'[Name Analysis Error]: Identifier `{expr_name.id.data}'
                f' found that was not previously defined.')
```

Name Analysis: Part II - Checking

2. Check that functions are declared before they are called

```python
class NameAnalysisVisitor(Visitor):
    name_ctx: NameCtx
    ...

    def visit_call_expr(self, call_expr: CallExpr):
        if call_expr.func.data in self.name_ctx:
            return
        else:
            raise SemanticError(
                f'
                [Name Analysis Error]: '}
                f"Identifier `{call_expr.func.data}' found that was not previously defined."
            )
    ...
```
Name Analysis: Part II - Checking

3. Make sure that function bodies are checked with the right name context

class NameAnalysisVisitor(Visitor):
    name_ctx: NameCtx
    ...

def traverse_func_def(self, func_def: FuncDef):
    # select the nested name context from the current name context ...
    nested_ctx = self.name_ctx.get_func_ctx(func_def.func_name.data)
    # ... and use the nested name context when traversing the function body
    body_visitor = NameAnalysisVisitor(nested_ctx)
    for op in func_def.func_body.ops:
        body_visitor.traverse(op)
    ...

Name Analysis: Part II - Checking

4. Check that the iteration variable in a for was previously defined

```python
class NameAnalysisVisitor(Visitor):
    name_ctx: NameCtx
    ...
    def visit_for(self, for_op: For):
        if for_op.iter_name.data not in self.name_ctx:
            raise SemanticError(
                f'[Name Analysis Error]: '
                f'Identifier `{for_op.iter_name.data}' found that was not previously defined.'
            )
    ...
```
Name Analysis: Part II - Checking

5. Check that variables in global declarations are declared with global scope

```python
class NameAnalysisVisitor(Visitor):
    name_ctx: NameCtx

    ...;

def visit_global_decl(self, global_decl: GlobalDecl):
    if global_decl.decl_name.data in self.name_ctx.global_scope():
        return
    else:
        raise SemanticError(
            f'[Name Analysis Error]: Identifier `{global_decl.decl_name.data}'
            f' not declared in global scope.
        )
```

Name Analysis putting the parts together

Name Analysis pass first constructs the context and then performs the checking.

def name_analysis(_: MLContext, module: ModuleOp) -> ModuleOp:
    # add print, len, and input functions to the global scope
    name_ctx = NameCtx()
    name_ctx.add_func("print", NameCtx())
    name_ctx.add_func("len", NameCtx())
    name_ctx.add_func("input", NameCtx())
    # first construct name context
   BuildContextVisitor(name_ctx).traverse(module)
    # then perform checking
    NameAnalysisVisitor(name_ctx).traverse(module)
    return module