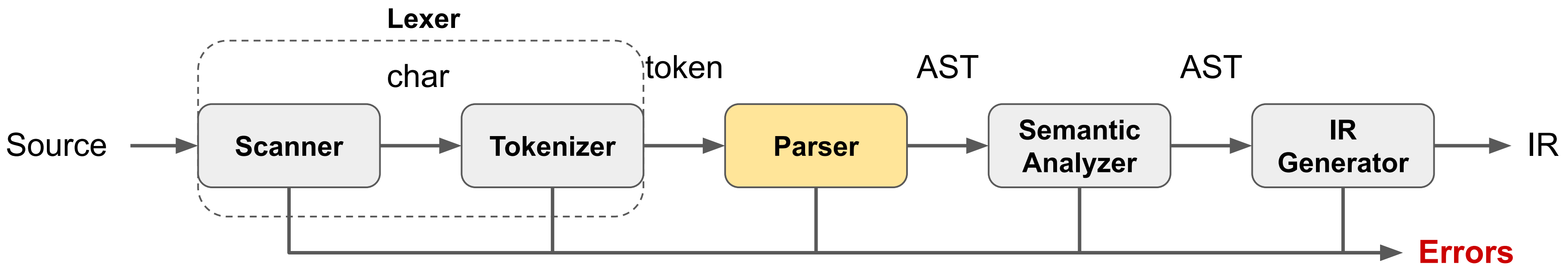
Compiling Techniques

Lecture 7: Abstract Syntax

Where are we?



A parser does more than simply recognize syntax.

In a multi-pass compiler, the parser builds a **syntax tree**, that can either be:

- a concrete syntax tree (aka parser tree) that directly corresponds to the parsers context-free grammar;
- a simplified abstract syntax tree (AST) that abstract some details away.

Example: Concrete Syntax Tree (Parse Tree)

Example: Grammar for arithmetic expressions in EBNF form

```
Expr   ::= Term ( (‘+’ | ‘-’) Term)*
Term   ::= Factor ( (‘*’ | ‘/’) Factor)*
Factor ::= number | ‘(‘ Expr ‘)’
```

Removing EBNF syntax

```
Expr    ::= Term Terms
Terms   ::= (‘+’ | ‘-’) Term Terms | ε
Term    ::= Factor Factors
Factors ::= (‘*’ | ‘/’) Factor Factors | ε
Factor  ::= number | ‘(‘ Expr ‘)’
```

Example: Concrete Syntax Tree (Parse Tree)

Example: Grammar for arithmetic expressions in EBNF form

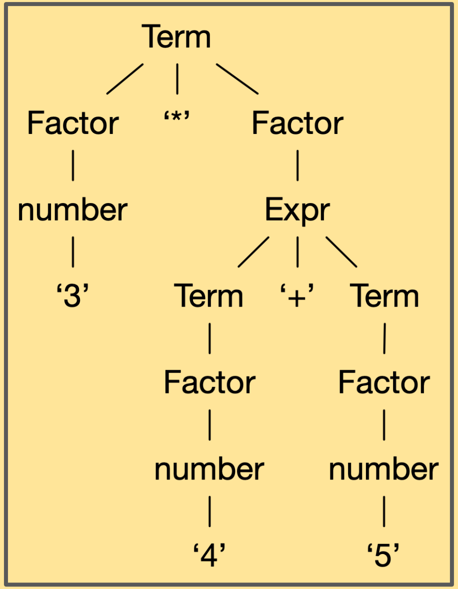
```
Expr   ::= Term ( (‘+’ | ‘-’) Term)*
Term   ::= Factor ( (‘*’ | ‘/’) Factor)*
Factor ::= number | ‘(‘ Expr ‘)’
```

Removing EBNF syntax + simplifications

```
Expr     ::= Term ((‘+’ | ‘-’) Expr | ε)
Term     ::= Factor ((‘*’ | ‘/’) Term | ε)
Factor   ::= number | ‘(‘ Expr ‘)’
```

Example: Concrete Syntax Tree (Parse Tree)

Concrete Syntax Tree for
3 * (4 + 5)



Grammar for arithmetic expression

```
Expr    ::= Term ((‘+’ | ‘-’) Expr | ε)
Term    ::= Factor ((‘*’ | ‘/’) Term | ε)
Factor  ::= number | ‘(‘ Expr ‘)’
```

The concrete syntax tree contains a lot of unnecessary information!

It is possible to simplify the tree by removing redundant information.

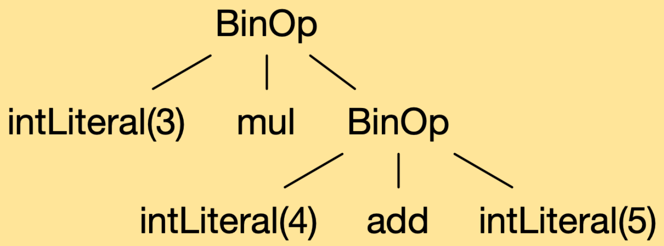
Abstract Grammar

The simplifications lead to a new simpler context-free grammar called Abstract Grammar

Example: Abstract grammar for arithmetic expressions

```
Expr   ::= BinOp | intLiteral
BinOp  ::= Expr Op Expr
Op     ::= add | sub | mul | div
```

***Abstract Syntax Tree* `for 3 * (4 + 5):`**



Choice of Abstract Grammar

For a given concrete grammar, there exists numerous abstract grammars.
We pick the most suitable grammar for the compiler.

Example: Abstract grammar for arithmetic expressions

```
Expr   ::= BinOp | intLiteral
BinOp  ::= Expr Op Expr
Op     ::= add | sub | mul | div
```

Alternative abstract grammar for arithmetic expressions

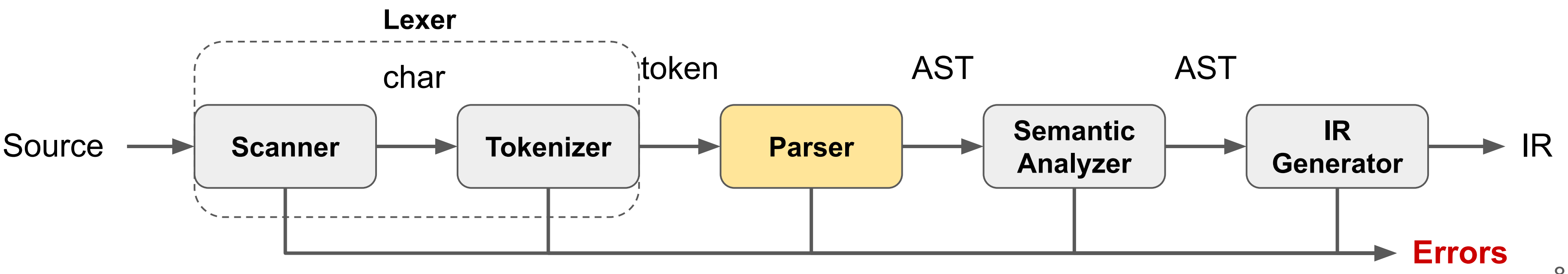
```
Expr   ::= AddOp | SubOp | MulOp | DivOp | intLiteral
AddOp  ::= Expr add Expr
SubOp  ::= Expr sub Expr
MulOp  ::= Expr mul Expr
DivOp  ::= Expr div Expr
```

Abstract Syntax Tree

The Abstract Syntax Tree (AST) forms the main intermediate representation of the compiler's front-end.

We will perform Semantic Analysis on this representation, that is:

- Name analysis (are all names declared before they are used?)
- Type checking



Implementation of the AST

The AST can be implemented like any other tree data structure

```
class Expr(ABC):
    pass

@dataclass
class BinOp(Expr):
    lhs: Expr
    op: str
    rhs: Expr

@dataclass
class IntLiteral(Expr):
    value: int
```

Abstract grammar

```
Expr    ::= BinOp | intLiteral
BinOp   ::= Expr Op Expr
Op      ::= add | sub | mul | div
```

Op should better be implemented as an `Enum`

```
BinOp(IntLiteral(3), "*", BinOp(IntLiteral(4), "+", IntLiteral(5)))
```

xDSL and MLIR

In this course, we use a framework to help us to implement our compiler.

This framework is called `xDSL`. It implements the same concepts that are found in the **MLIR - Multi-Level IR Compiler Framework** that is used in industry.

We will introduce new concepts of the framework as we go along.

Today we discuss how to represent ASTs with xDSL.

https://github.com/xdslproject/xdsl/ https://mlir.llvm.org/

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

`IRDLOperation` is the superclass of all AST nodes

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

`IRDLOperation` is the superclass of all AST nodes

Each `Operation` has a *name*

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

`IRDLOperation` is the superclass of all AST nodes

Each `Operation` has a *name*

Metadata is represented by `Attributes`

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

`IRDLOperation` is the superclass of all AST nodes

Each `Operation` has a *name*

Metadata is represented by `Attributes`

A `region` represents nested structure, such as the children of a node in the AST

Implementation of the AST with xDSL

xDSL helps us to easily define intermediate representations (such as our AST).

Here is the definition of our small AST.

```
@irdl_op_definition
class BinOp(IRDLOperation):
    name = "BinOp"
    op = prop_def(StringAttr)
    lhs = region_def()
    Rhs = region_def()

@irdl_op_definition
class IntLiteral(IRDLOperation):
    name = "IntLiteral"
    value = prop_def(IntegerAttr[IntegerType])
```

`IRDLOperation` is the superclass of all AST nodes

Each `Operation` has a *name*

Metadata is represented by `Attributes`

A `region` represents nested structure, such as the children of a node in the AST

A *macro* generates helpful boilerplate code to make printing, testing, etc. easy

Creating Operations with xDSL

xDSL provides a generic and flexible (but verbose) interface to create Operations:

```
node = Op.create(attributes={"key": value}, regions=[...])
```

We can easily hide the boilerplate, for example for `IntLiteral`:

```
class IntLiteral(IRDLOperation):
    @staticmethod
    def get(value: int) -> IntLiteral:
        return IntLiteral.create(attributes={
            "value": IntegerAttr.from_int_and_width(value, 32)})
```

This allows us to write:

```
BinOp.get(IntLiteral.get(3), "*",
          BinOp.get(IntLiteral.get(4), "+", IntLiteral.get(5)))
```

First Benefits of using xDSL

Using a framework like xDSL has many benefits.
For example, can we easily debug and print our created AST:

```
>>> xdsl.printer.Printer().print_op(
     BinOp.get(IntLiteral.get(3), "*",
               BinOp.get(IntLiteral.get(4), "+", IntLiteral.get(5))) )

"BinOp"() <{"op" = "*"}> ({
  "IntLiteral"() <{"value" = 3 : !i32}>
}, {
  "BinOp"() <{"op" = "+"}> ({
    "IntLiteral"() <{"value" = 4 : !i32}>
  }, {
    "IntLiteral"() <{"value" = 5 : !i32}>
  })
})
```

ChocoPy AST in xDSL – `Operations`

The CW1 template provides an implementation of the ChocoPy AST in xDSL which defines the following 22 `Operations`:

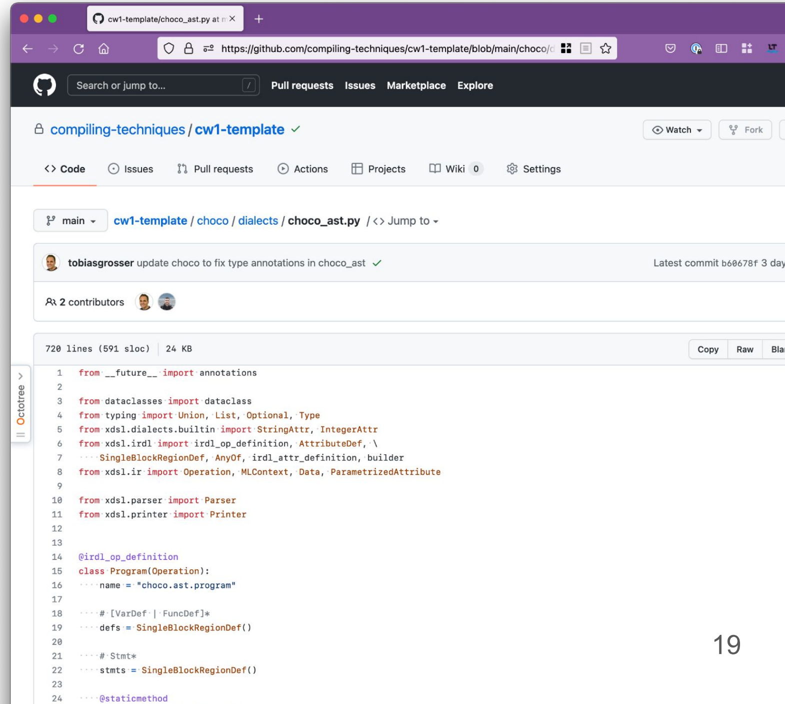
```
Program

TypeName, ListType, TypedVar

FuncDef, GlobalDecl, NonLocalDecl, VarDef

If, While, For, Pass, Return, Assign

Literal, ExprName, UnaryExpr, BinaryExpr,
IfExpr, ListExpr, CallExpr, IndexExpr
```



ChocoPy AST in xDSL – `Attributes`

An **`Attribute`** represents some compile-time metadata of an `Operation`

Examples of `Attributes` in the ChocoPy AST are:

- Names, such as the names of functions, variables, or types
- Literal values, e.g. `4`, “`Hello`”, or `True`
- Operator of binary and unary operations, e.g. `+`, `-`, `/`, `==`, `!=`, …

To represent this different metadata, we use these 4 types of Attributes:

`StringAttr, IntegerAttr, BoolAttr, NoneAttr`

The `NoneAttr` represents the **`None`** value of ChocoPy.

ChocoPy AST in xDSL – `Regions`

We use **`Regions`** to represent nesting.

E.g. `BinaryExpr` has two regions, one for each Operand.

Regions can have more than one `Operation` in them!

Consider for example the `If` Statement:

The second region represents the `then`-block, the third region the `else`-block.

```
"BinaryExpr"() <{"op" = "+"}> ({
    "Literal"() <{"value" = 4 : !i32}>
}, {
    "Literal"() <{"value" = 5 : !i32}>
})
```

```
"If"() ({
  "Literal"() <{"value" = !bool<True>}>
}, {
  "Literal"() <{"value" = 4 : !i32}>
  "Literal"() <{"value" = 8 : !i32}>
}, {
  "Literal"() <{"value" = 15 : !i32}>
  "Literal"() <{"value" = 16 : !i32}>
})
```