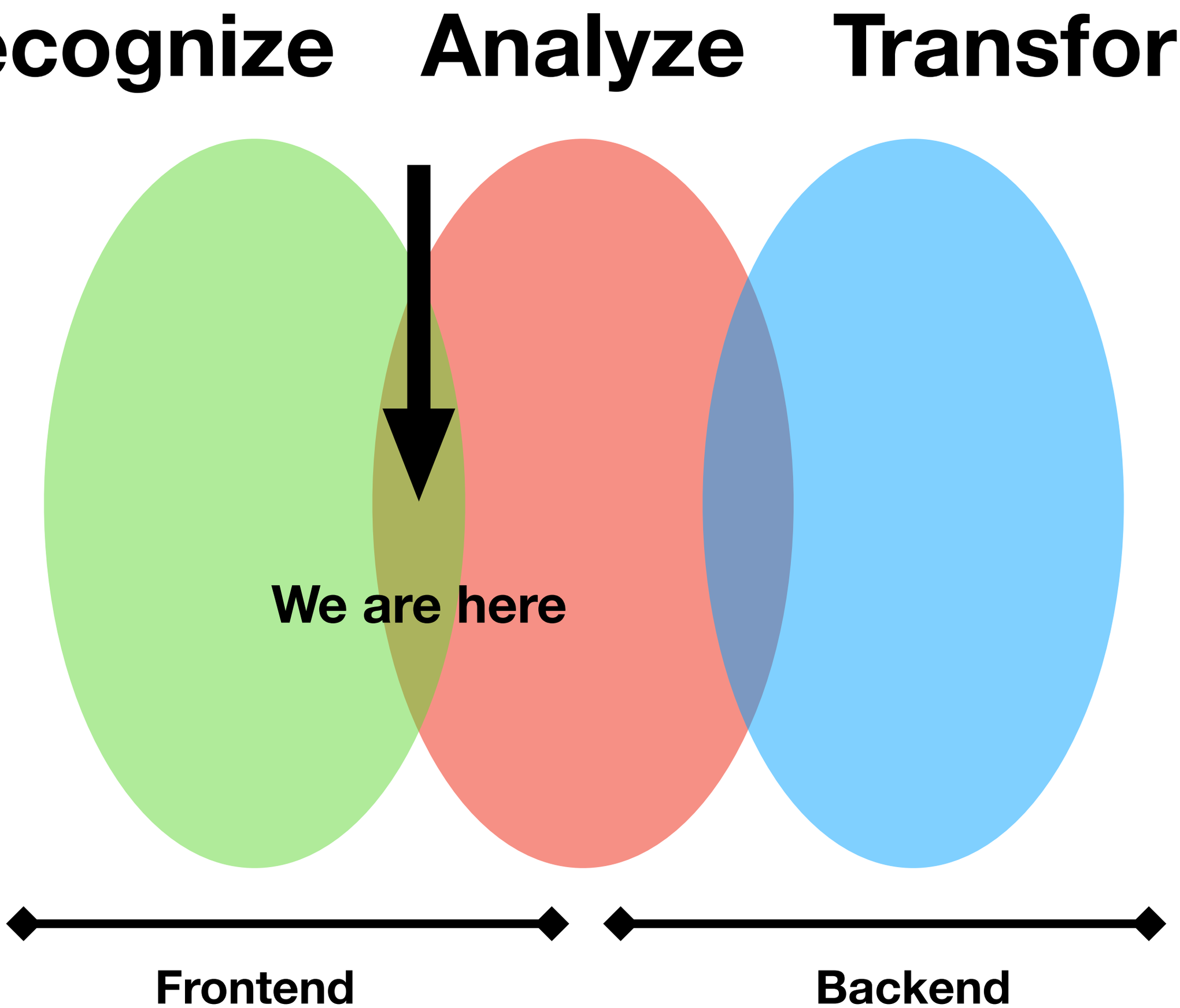
CSC488/2107 Winter 2019 — Compilers & Interpreters

https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

Agenda

- Abstract Syntax Trees


Recognize
Analyze
Transform
We are here
Frontend
Backend

Recognize

- Lexical structure
- Syntactic structure
- Highly language/syntax specific

- Data flow:

  Stream of *Characters*

  ➥ Stream of *Tokens*

  ➥ *Parse Tree* (Concrete Syntax)

Analyze

- Semantic meaning
- Less language specific
- Data flow:

  Parse Tree

  ➥ *Abstract Syntax Tree* (possibly with annotations and/or associated symbol tables)

Transform (Lower)

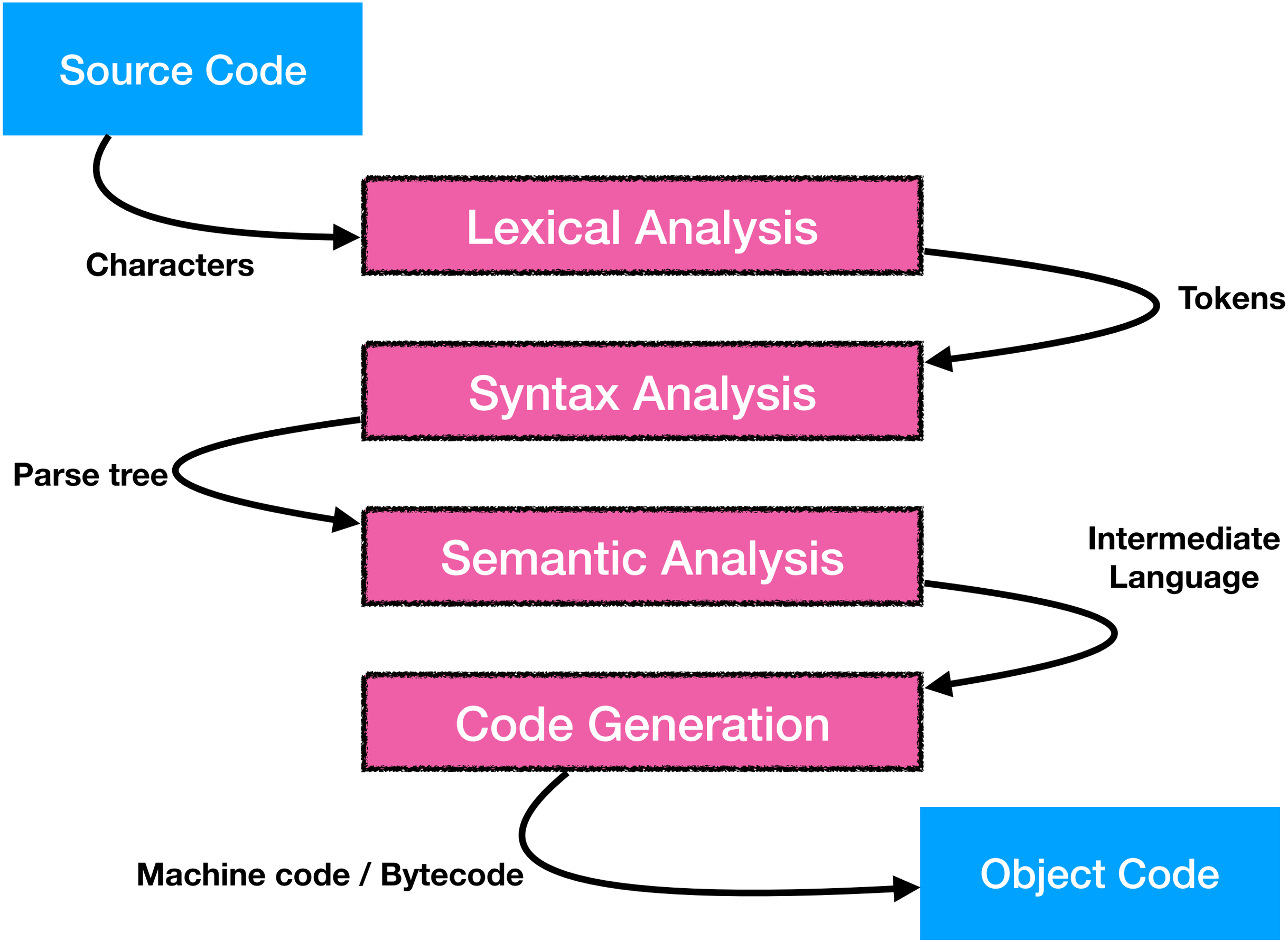
- Memory layout
- Optimization (optional)
- Code generation
- Very target specific

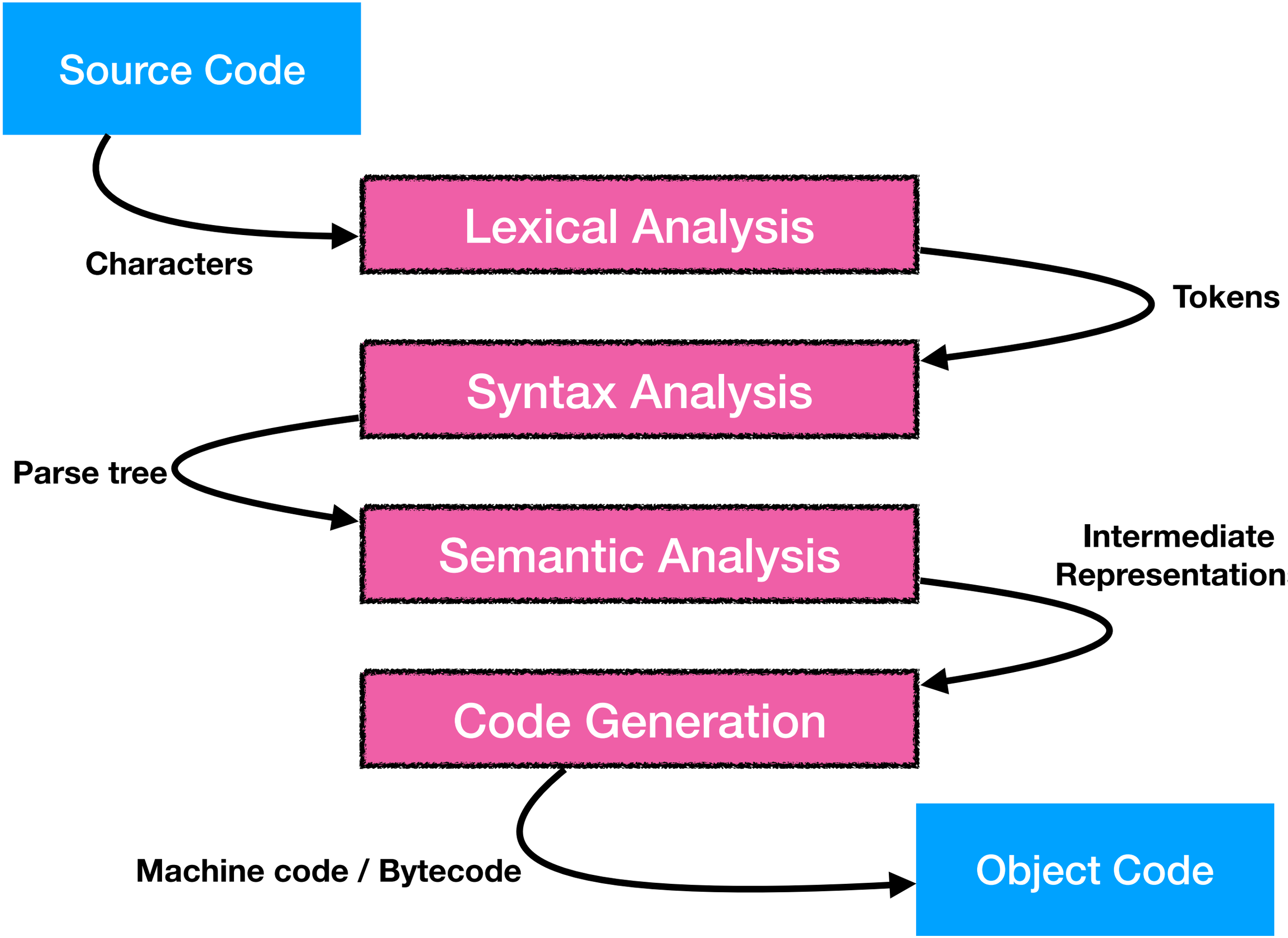
- Data flow:

  Abstract Syntax Tree

  ➥ *Intermediate Languages/Representations* (optional)

  ➥ *Target Machine Code*


Source Code
Characters
Lexical Analysis
Tokens
Syntax Analysis
Parse tree
Semantic Analysis
Intermediate Language
Code Generation
Machine code / Bytecode
Object Code


Source Code
Characters
Lexical Analysis
Tokens
Syntax Analysis
Parse tree
Semantic Analysis
Intermediate Representations
Code Generation
Machine code / Bytecode
Object Code

Concrete *vs* Abstract Syntax Trees

Arithmetic Expressions

S ⟶ E

E ⟶ T

E ⟶ E + T

E ⟶ E - T

T ⟶ T * F

T ⟶ T / F

T ⟶ F

F ⟶ id

F ⟶ num

F ⟶ (E)

Parse Tree / Concrete Syntax Tree

$S \longrightarrow E$

$E \longrightarrow T$

$E \longrightarrow E + T$

$E \longrightarrow E - T$

$T \longrightarrow T * F$

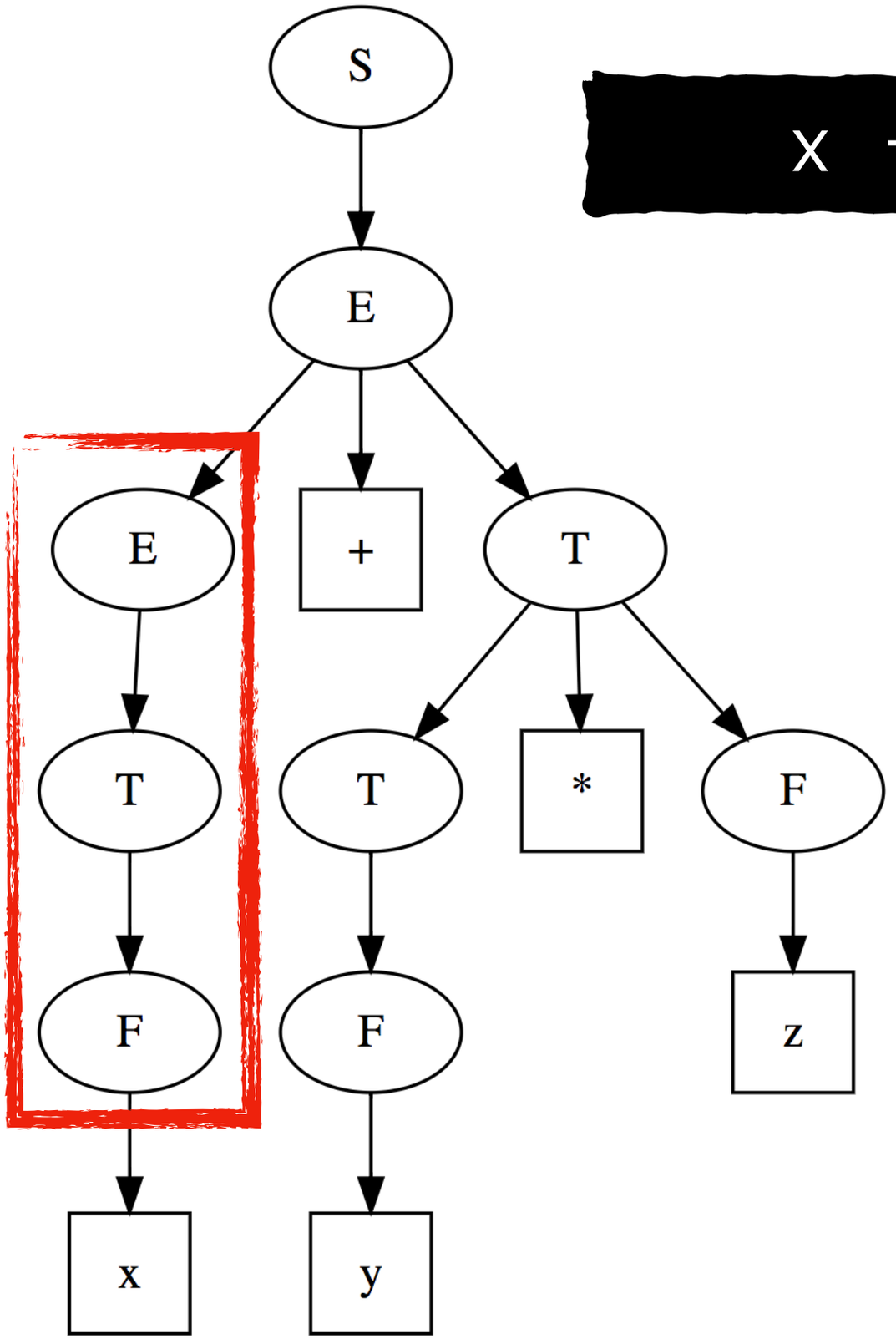
$T \longrightarrow T / F$

$T \longrightarrow F$

$F \longrightarrow id$

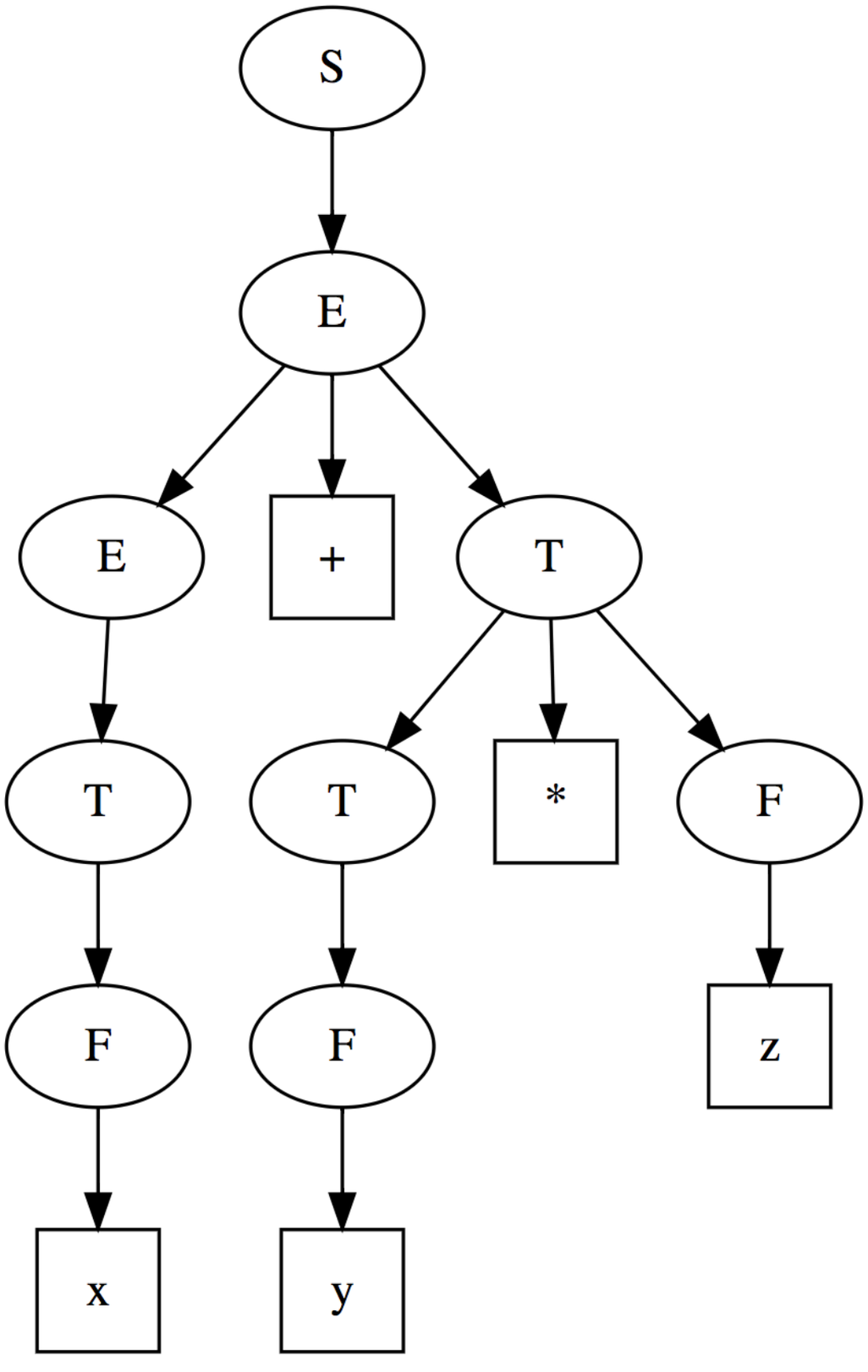
$F \longrightarrow num$

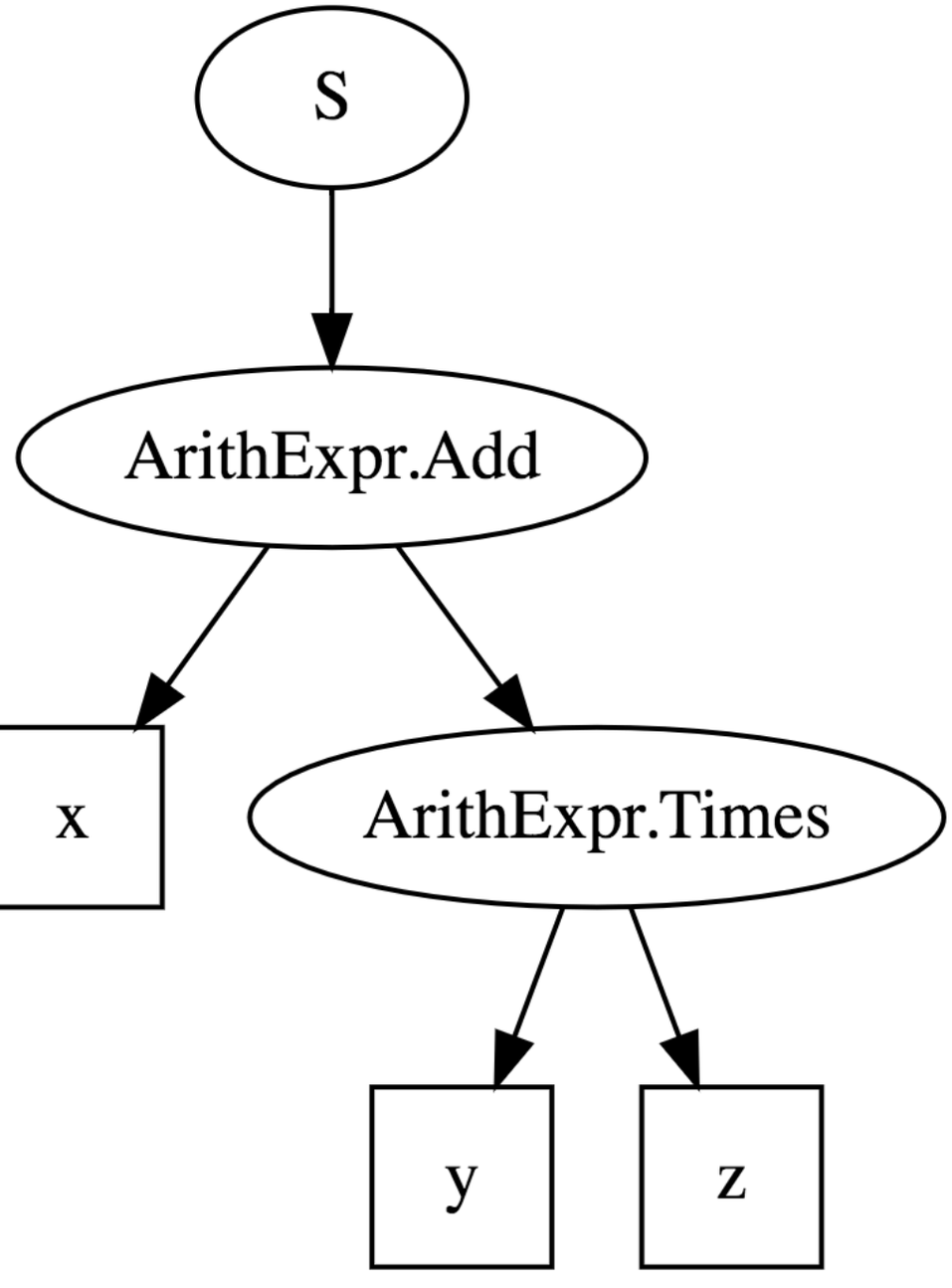
$F \longrightarrow (E)$



x + y * z

Concrete *vs* Abstract Syntax Tree





Abstract Syntax Trees (AST)

- An *abstract syntax tree* (AST) is a form of *intermediate representation*, an internal data structured used within and between stages the compiler pipeline
- Parser generates AST nodes via *actions*
  - A successful parse should generate a equivalent representation of the input source
  - Certain details like extraneous parentheses will be dropped
- Tree is initially a *directed-acyclic graph* (DAG)
  - Thanks to the bottom-up operation of an *LR* parser, tree is built from leaves (terminals) up to the root
- Later transformations can add cycles and additional links to facilitate tree traversal

Implementing AST

- Highly source language specific
- Node datatype definitions for each important aspect of language:
  - Simplified Lisp-style S-expressions
  - Haskell-style GADTs
  - Object-oriented class hierarchy
- Consider desirable properties like *structural equivalency*:
  - Python: `(1, 2, "CSC", [ 488 ]) == (1, 2, "CSC", [ 488 ])`
    extended to custom datatypes/classes
  - Example: `AssignStmt(lhs=L, rhs=R) == AssignStmt(lhs=L, rhs=R)`

PLY *parser actions*

```
def p_E(self, p):
    ''' E    : T
             | E '+' T
             | E '-' T       '''
    pass

def p_T(self, p):
    ''' T    : F
             | T '*' F
             | T '/' F       '''
    pass

def p_F(self, p):
    ''' F    : NUMBER
             | IDENT
             | '(' E ‘)'     '''
    pass
```

```
def p_E_T(self, p):
    ''' E : T '''
    …

def p_E_plus(self, p):
    ''' E : E '+' T '''
    p[0] = ArithExpr.Add(…)

def p_E_minus(self, p):
    ''' E : E '-' T '''
    p[0] = ArithExpr.Minus(…)
```

AST for CSC488 Source Language

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident=‘k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident=‘k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k  = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident=‘k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer  k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
k = x + y * z
```

```
AssignStmt(
 lhs=VarRef(name=Ident(ident='k')),
 rhs=ArithExpr.Plus(
   left=VarRefExpr(
     ref=VarRef(name=Ident(ident='x'))),
   right=ArithExpr.Times(
     left=VarRefExpr(
       ref=VarRef(name=Ident(ident='y'))),
     right=VarRefExpr(
       ref=VarRef(name=Ident(ident='z'))))))
```

```
k = x + y * z
```

```
AssignStmt(
  lhs=VarRef(name=Ident(ident='k')),
  rhs=ArithExpr.Plus(
    left=VarRefExpr(
      ref=VarRef(name=Ident(ident='x')))),
    right=ArithExpr.Times(
      left=VarRefExpr(
        ref=VarRef(name=Ident(ident='y')))),
      right=VarRefExpr(
        ref=VarRef(name=Ident(ident='z'))))))
```

```
k = x + y * z
```

```
AssignStmt(
 lhs=VarRef(name=Ident(ident='k')),
 rhs=ArithExpr.Plus(
   left=VarRefExpr(
     ref=VarRef(name=Ident(ident='x'))),
   right=ArithExpr.Times(
     left=VarRefExpr(
       ref=VarRef(name=Ident(ident='y'))),
     right=VarRefExpr(
       ref=VarRef(name=Ident(ident='z'))))))
```

```
k = x + y * z
```

```
AssignStmt(
 lhs=VarRef(name=Ident(ident='k')),
 rhs=ArithExpr.Plus(
   left=VarRefExpr(
     ref=VarRef(name=Ident(ident='x'))),
   right=ArithExpr.Times(
     left=VarRefExpr(
       ref=VarRef(name=Ident(ident='y'))),
     right=VarRefExpr(
       ref=VarRef(name=Ident(ident='z'))))))
```

```
k = x + y * z
```

```
AssignStmt(
 lhs=VarRef(name=Ident(ident='k')),
 rhs=ArithExpr.Plus(
   left=VarRefExpr(
     ref=VarRef(name=Ident(ident='x'))),
   right=ArithExpr.Times(
     left=VarRefExpr(
       ref=VarRef(name=Ident(ident='y'))),
     right=VarRefExpr(
       ref=VarRef(name=Ident(ident='z'))))))
```

```
k = x + y * z
```

```
AssignStmt(
 lhs=VarRef(name=Ident(ident='k')),
 rhs=ArithExpr.Plus(
   left=VarRefExpr(
     ref=VarRef(name=Ident(ident='x'))),
   right=ArithExpr.Times(
     left=VarRefExpr(
       ref=VarRef(name=Ident(ident='y'))),
     right=VarRefExpr(
       ref=VarRef(name=Ident(ident='z'))))))
```

Helpful things to have

- AST nodes that can self-generate an internal representation suitable for regenerating and comparing
  - Python `__repr__`
    - Example: `print(AssignStmt(lhs=…, rhs=…))` `=> "AssignStmt(lhs=…, rhs=…)"`
- A *pretty-printer* that can walk an AST tree and reproduce the equivalent source language program that, when parsed, will produce a structurally equivalent tree (the *inverse* of parsing)

Pretty-printing AST

From this…

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

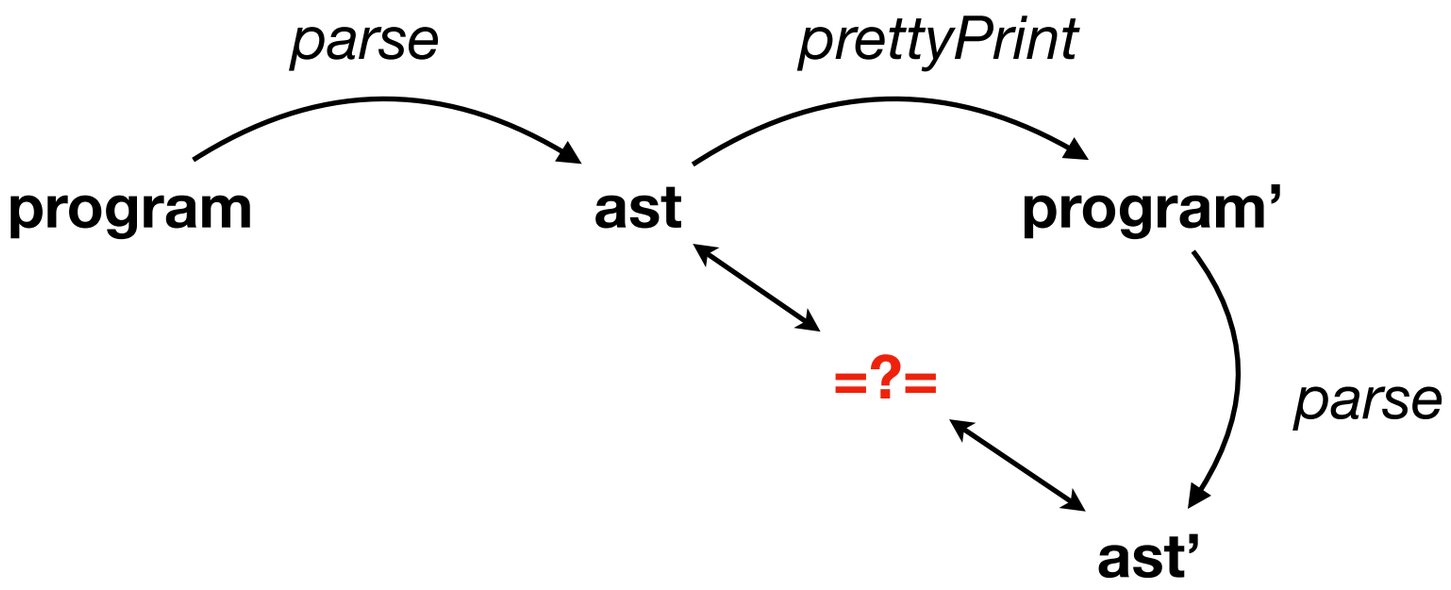
… back to this:

```
var k integer
k = 0
```

Parser-AST roundtrip

```
ast = parse(program)

parse(prettyPrint(ast)) == ast
```



Traversing an AST

- An AST can be traversed in any order: strictly *depth-first* or *breadth-first*, or using a custom strategy

- In an object-oriented context, can use the *visitor pattern* for type-directed handling

- Can be treated as mutable or immutable

  - Mutable: make changes as you go

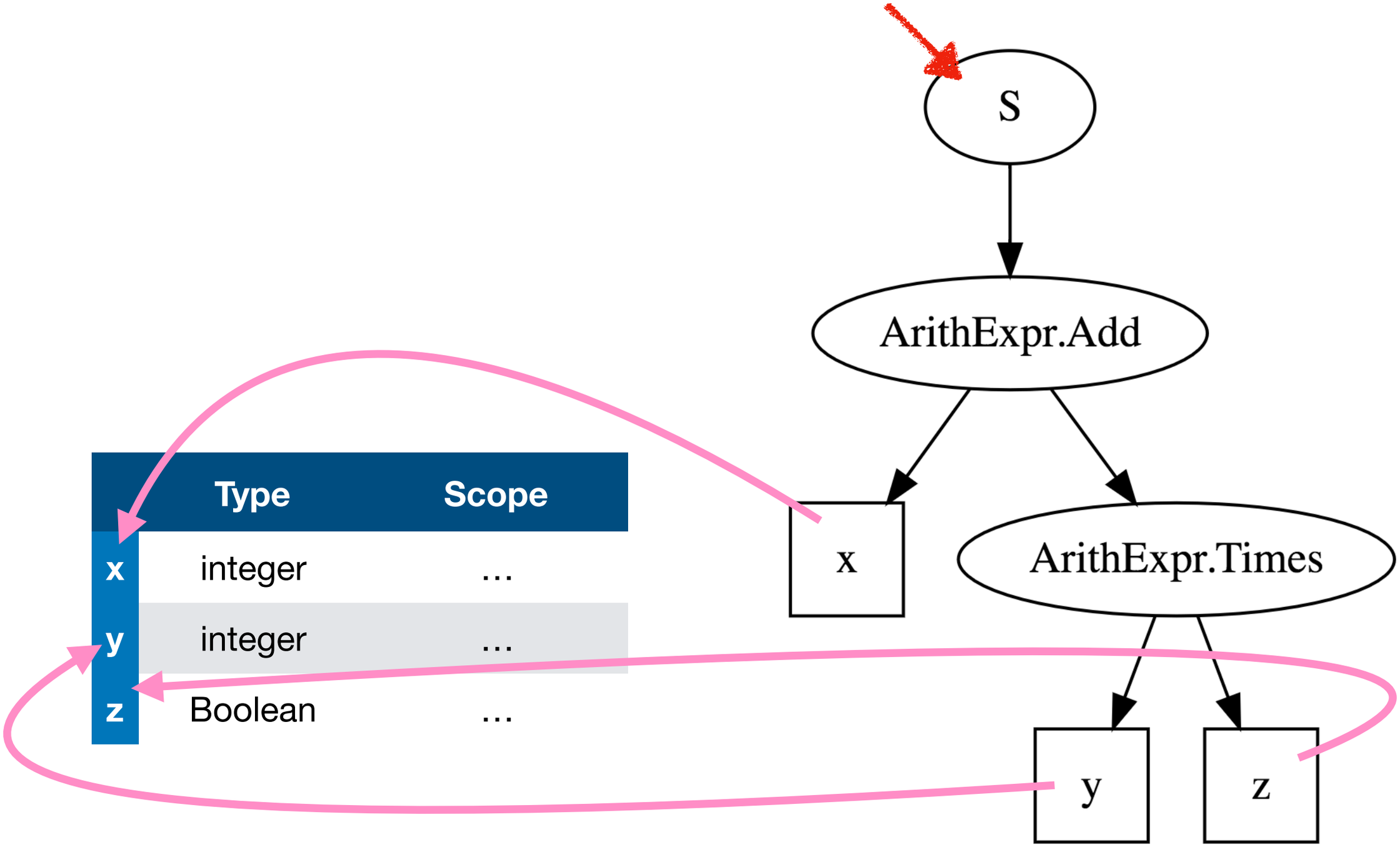
  - Immutable: don’t modify original, generate fresh copy or maintain state separately

Transforming an AST

- A lesson from functional languages like Haskell:

'Tis better to transform than to mutate

- Why?


S
ArithExpr.Add
x
ArithExpr.Times
y
z
Type
Scope
x
integer
…
y
integer
…
z
Boolean
…

```
var k integer   k = 0
```

```
Scope(
 decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ Ident(ident='k') ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=Ident(ident='k')),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

```
var k integer   k = 0
```

```
symbolK = Symbol(ident='k', …)

Scope(
decls=[
  VarDecl(
   typ=ScalarType.INTEGER_TYPE,
   names=[ symbolK ]
  )],
 stmts=[
  AssignStmt(
   lhs=VarRef(name=symbolK),
   rhs=IntegerConstExpr(c=0))],
 program=True)
```

Visitor pattern

```
def visit_AssignStmt(self, stmt):
  # Make preparations…
  lhs = self.visit(stmt.lhs)
  rhs = self.visit(stmt.rhs)

  return AssignStmt(lhs, rhs)

def visit_Ident(self, ident):
  sym = self.lookupIdent(ident)
  if sym is not None:
    return sym

  return self.generateSymbol(ident)
```