Declare Your Language

Chapter 2: Syntax Definition

Eelco Visser

IN4303 Compiler Construction
TU Delft
September 2017

TU Delft

Reading Material

TU Delft

The perspective of this lecture on declarative syntax definition is explained more elaborately in this Onward! 2010 essay. It uses an on older version of SDF than used in these slides. Production rules have the form

$X_1 \ldots X_n$ -> N {cons("c")}

instead of

N.c = $X_1 \ldots X_n$

Pure and Declarative Syntax Definition: Paradise Lost and Regained

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
g.h.wachsmuth@tudelft.nl

Abstract

Syntax definitions are pervasive in modern software systems, and serve as the basis for language processing tools like parsers and compilers. Mainstream parser generators pose restrictions on syntax definitions that follow from their implementation algorithm. They hamper evolution, maintainability, and compositionality of syntax definitions. The pureness and declarativity of syntax definitions is lost. We analyze how these problems arise for different aspects of syntax definitions, discuss their consequences for language engineers, and show how the pure and declarative nature of syntax definitions can be regained.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Syntax; D.3.4 [*Programming Languages*]: Processors — Parsing; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design, Languages

Prologue

In the beginning were the *words*, and the words were *trees*, and the trees were words. All words were made through *grammars*, and without grammars was not any word made that was made. Those were the days of the garden of Eden. And there where language engineers strolling through the garden. They made languages which were sets of words by making grammars full of beauty. And with these grammars, they turned words into trees and trees into words. And the trees were natural, and pure, and beautiful, as were the grammars.

Among them were software engineers who made software as the language engineers made languages. And they dwelt with them and they were one people. The language engineers were software engineers and the software engineers were language engineers. And the language engineers made *language software*. They made *recognizers* to know words, and *generators* to make words, and *parsers* to turn words into trees, and *formatters* to turn trees into words.

But the software they made was not as natural, and pure, and beautiful as the grammars they made. So they made software to make language software and began to make language software by making *syntax definitions*. And the syntax definitions were grammars and grammars were syntax definitions. With their software, they turned syntax definitions into language software. And the syntax definitions were language software and language software were syntax definitions. And the syntax definitions were natural, and pure, and beautiful, as were the grammars.

The Fall Now the serpent was more crafty than any other beast of the field. He said to the language engineers,

> *Did you actually decide not to build any parsers?*

And the language engineers said to the serpent,

> *We build parsers, but we decided not to build others than general parsers, nor shall we try it, lest we loose our syntax definitions to be natural, and pure, and beautiful.*

But the serpent said to the language engineers,

> *You will not surely loose your syntax definitions to be natural, and pure, and beautiful. For you know that when you build particular parsers your benchmarks will be improved, and your parsers will be the best, running fast and efficient.*

So when the language engineers saw that restricted parsers were good for efficiency, and that they were a delight to the benchmarks, they made software to make efficient parsers and began to make efficient parsers by making *parser definitions*. Those days, the language engineers went out from the garden of Eden. In pain they made parser definitions all the days of their life. But the parser definitions were not grammars and grammars were not parser definitions. And by the sweat of their faces they turned parser definitions into effi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

Copyright © 2010 ACM 978-1-4503-0236-4/10/10. . . $10.00

https://doi.org/10.1145/1932682.1869535

http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2010-019.pdf

The SPoofax Testing (SPT) language used in the section on testing syntax definitions was introduced in this OOPSLA 2011 paper.

Integrated Language Definition Testing

Enabling Test-Driven Language Development

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Rob Vermaas
LogicBlox
rob.vermaas@logicblox.com

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

The reliability of compilers, interpreters, and development environments for programming languages is essential for effective software development and maintenance. They are often tested only as an afterthought. Languages with a smaller scope, such as domain-specific languages, often remain untested. General-purpose testing techniques and test case generation methods fall short in providing a low-threshold solution for test-driven language development. In this paper we introduce the notion of a *language-parametric testing language (LPTL)* that provides a reusable, generic basis for declaratively specifying language definition tests. We integrate the syntax, semantics, and editor services of a language under test into the LPTL for writing test inputs. This paper describes the design of an LPTL and the tool support provided for it, shows use cases using examples, and describes our implementation in the form of the Spoofax testing language.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Interactive Environments

General Terms Languages, Reliability

Keywords Testing, Test-Driven Development, Language Engineering, Grammarware, Language Workbench, Domain-Specific Language, Language Embedding, Compilers, Parsers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

1. Introduction

Software languages provide linguistic abstractions for a domain of computation. Tool support provided by compilers, interpreters, and integrated development environments (IDEs), allows developers to reason at a certain level of abstraction, reducing the accidental complexity involved in software development (e.g., machine-specific calling conventions and explicit memory management). *Domain-specific* languages (DSLs) further increase expressivity by restricting the scope to a particular application domain. They increase developer productivity by providing domain-specific notation, analysis, verification, and optimization.

With their key role in software development, the correct implementation of languages is fundamental to the reliability of software developed with a language. Errors in compilers, interpreters, and IDEs for a language can lead to incorrect execution of correct programs, error messages about correct programs, or a lack of error messages for incorrect programs. Erroneous or incomplete language implementations can also hinder understanding and maintenance of software.

Testing is one of the most important tools for software quality control and inspires confidence in software [1]. Tests can be used as a basis for an agile, iterative development process by applying test-driven development (TDD) [1], they unambiguously communicate requirements, and they avoid regressions that may occur when new features are introduced or as an application is refactored [2, 31].

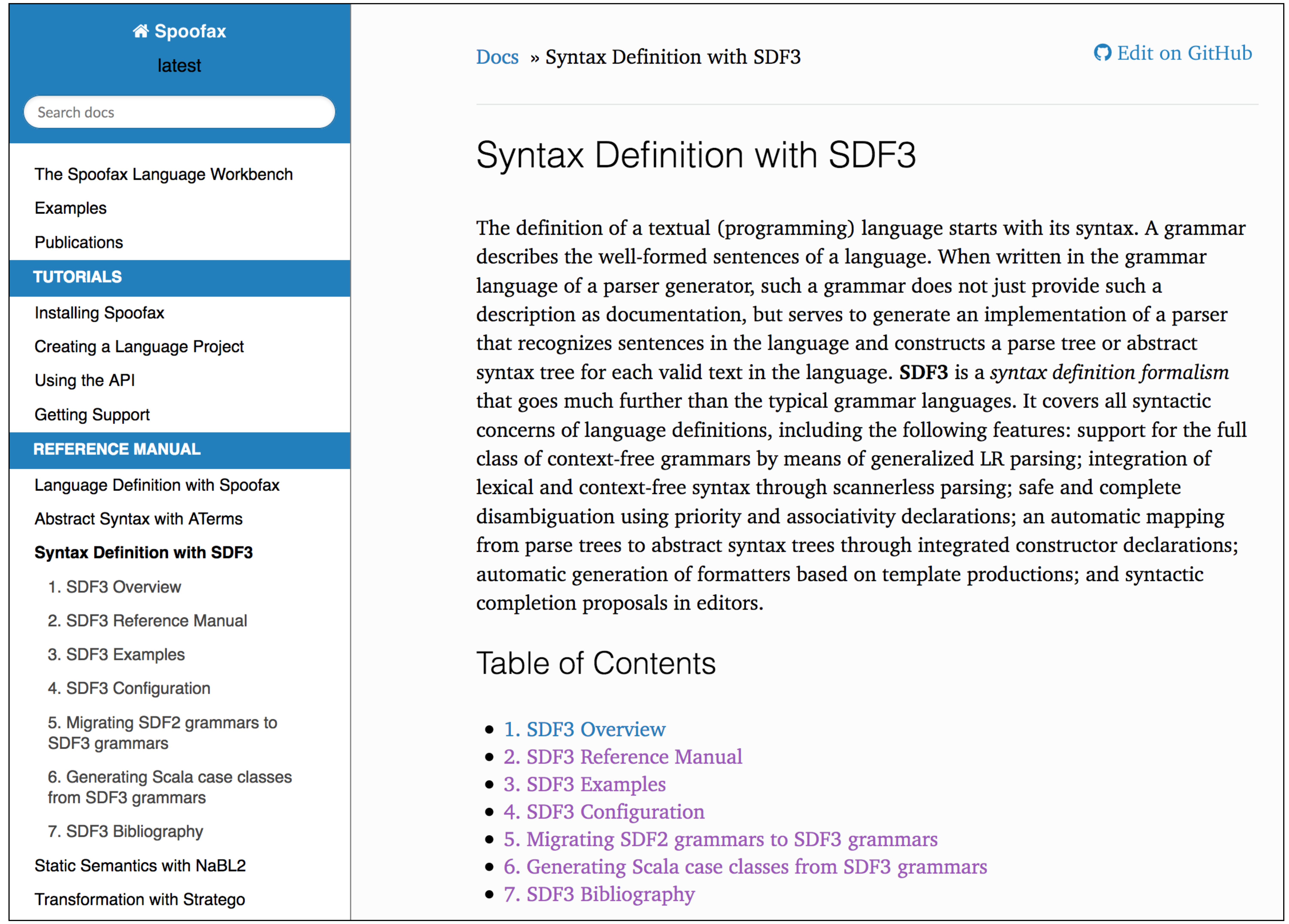
Scripts for automated testing and general-purpose testing tools such as the xUnit family of frameworks [19] have been successfully applied to implementations of general-purpose languages [16, 38] and DSLs [18, 33]. With the successes and challenges of creating such test suites by hand, there has been considerable research into *automatic generation* of compiler test suites [3, 27]. These techniques provide an effective solution for thorough black-box testing of complete compilers, by using annotated grammars to generate input programs.

Despite extensive practical and research experience in testing and test generation for languages, rather less attention has been paid to supporting language engineers in writing tests, and to applying TDD with tools specific to the do-

https://doi.org/10.1145/2076021.2048080

http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2011-011.pdf

The SDF3 syntax definition formalism is documented at the metaborg.org website.



Spoofax
latest
Search docs

The Spoofax Language Workbench
Examples
Publications

TUTORIALS

Installing Spoofax
Creating a Language Project
Using the API
Getting Support

REFERENCE MANUAL

Language Definition with Spoofax
Abstract Syntax with ATerms
Syntax Definition with SDF3
1. SDF3 Overview
2. SDF3 Reference Manual
3. SDF3 Examples
4. SDF3 Configuration
5. Migrating SDF2 grammars to SDF3 grammars
6. Generating Scala case classes from SDF3 grammars
7. SDF3 Bibliography
Static Semantics with NaBL2
Transformation with Stratego

Docs » Syntax Definition with SDF3 Edit on GitHub

Syntax Definition with SDF3

The definition of a textual (programming) language starts with its syntax. A grammar describes the well-formed sentences of a language. When written in the grammar language of a parser generator, such a grammar does not just provide such a description as documentation, but serves to generate an implementation of a parser that recognizes sentences in the language and constructs a parse tree or abstract syntax tree for each valid text in the language. **SDF3** is a *syntax definition formalism* that goes much further than the typical grammar languages. It covers all syntactic concerns of language definitions, including the following features: support for the full class of context-free grammars by means of generalized LR parsing; integration of lexical and context-free syntax through scannerless parsing; safe and complete disambiguation using priority and associativity declarations; an automatic mapping from parse trees to abstract syntax trees through integrated constructor declarations; automatic generation of formatters based on template productions; and syntactic completion proposals in editors.

Table of Contents

http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/index.html

Lab Organization

Lab Organization

- Course website:
  https://tudelft-in4303-2017.github.io
- Lab assignments distributed through GitHub
- Detailed guide for doing lab assignments:
  https://tudelft-in4303-2017.github.io/documentation/git.html

Lab assignment logistics

- Submit your netid and GitHub username on WebLab (https://weblab.tudelft.nl/in4303/2017-2018/assignment/15274/view)
- Grades are published on WebLab
- You will get **read** access to our repository for you: https://github.com/TUDelft-IN4303-2017/student-<netid>
- **Fork** the repository — you can work from this template
- Submit a **pull-request** (PR) to our repository to submit your solution
- On the deadline the PR gets merged automatically, then graded

Lab assignment logistics

- The branch of the PR should be called assignment1 for the first assignment, assignment2 for the second, etc.
- We push new branches for new assignments to our repository
- Sometimes you are instructed to reuse a previous solution
- When a PR is opened or changes are pushed, you can get early feedback
- This feedback is based on an automated grading system
- Early feedback is given only a few times
- So be strategic about when you open the PR and when you push changes

Syntax

What is wrong with this program?

```
package org.metaborg.lang.tiger.interpreter.natives;

import org.metaborg.lang.tiger.interpreter.natives.addI_2NodeGen;
import org.metaborg.meta.lang.dynsem.interpreter.nodes.building.TermBuild;

import com.oracle.truffle.api.source.SourceSection;

public abstract class addI_2 extends TermBuild {

  public addI_2(SourceSection source) {
    super(source)
  }

  public int doInt(int left, int right) {
    left + right
  }

  public static TermBuild create(SourceSection source, TermBuild left, TermBuild right) {
    addI_2NodeGen.create(source, left, right)
  }

}
```

What is wrong with this program?

```
let def int power(x: int, n: int) =
      if n <= 0 then 1
      else x * power(x, n - 1)
 in power(3, 10)
end
```

Declarative Syntax Definition

What is Syntax?

In linguistics, **syntax** (/ˈsɪntæks/[1][2]) is the set of rules, principles, and processes that govern the structure of sentences in a given language, specifically word order and punctuation.

The term *syntax* is also used to refer to the study of such principles and processes.[3]

The goal of many syntacticians is to discover the syntactic rules common to all languages.

In mathematics, *syntax* refers to the rules governing the behavior of mathematical systems, such as formal languages used in logic. (See logical syntax.)

The word *syntax* comes from Ancient Greek: σύνταξις "coordination", which consists of σύν *syn*, "together," and τάξις *táxis*, "an ordering".

https://en.wikipedia.org/wiki/Syntax

Syntax (Programming Languages)

In computer science, the **syntax** of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.

This applies both to programming languages, where the document represents source code, and markup languages, where the document represents data.

The syntax of a language defines its surface form.[1]

Text-based computer languages are based on sequences of characters, while visual programming languages are based on the spatial layout and connections between symbols (which may be textual or graphical).

Documents that are syntactically invalid are said to have a syntax error.

https://en.wikipedia.org/wiki/Syntax_(programming_languages)

Syntax Definition

Syntax

- The set of rules, principles, and processes that govern the structure of sentences in a given language
- The set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language

How to describe such a set of rules?

- An algorithm that recognizes (the structure of) well-formed sentences

What is the ***Purpose*** of a Syntax Definition?

Parsing programs during compilation

Checking syntax in an editor

Coloring tokens in an editor

Formatting a program (after refactoring it)

Proposing completions in an editor

Randomly generating well-formed test programs

Explaining the syntax rules to a programmer

How do you do that?

Parsing programs during compilation

- an algorithm that recognizes sentence and constructs structure

Checking syntax in an editor

- an algorithm that finds syntax errors in program text (and recovers from previous errors)

Coloring tokens in an editor

- assign colors to syntactic categories of tokens

Formatting a program (after refactoring it)

- map a syntax tree to text

Proposing completions in an editor

- determine what syntactic constructs are valid here

Randomly generating well-formed test programs

- some probabilistic algorithm?

Explaining the syntax rules to a programmer

- let them read the parsing algorithm for the language?

Separation of Concerns

Language Design

- define the properties of a language
- done by a language designer

Language Implementation

- implement tools that satisfy properties of the language
- done by a language implementer

Can we automate the language implementer?

- that is what language workbenches attempt to do

Declarative Language Definition

Objective

- A workbench supporting design and implementation of programming languages

Approach

- Declarative multi-purpose domain-specific meta-languages

Meta-Languages

- Languages for defining languages

Domain-Specific

- Linguistic abstractions for domain of language definition (syntax, names, types, …)

Multi-Purpose

- Derivation of interpreters, compilers, rich editors, documentation, and verification from single source

Declarative

- Focus on what not how; avoid bias to particular purpose in language definition

A Recipe for Separation of Concerns

Representation

- Standardized representation for <aspect> of programs
- Independent of specific object language

Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

Language-Independent Interpretation

- Formalism interpreted by language-independent algorithm
- Multiple interpretations for different purposes
- Reuse between implementations of different languages

Declarative Syntax Definition

Representation: (Abstract Syntax) Trees

- Standardized representation for structure of programs
- Basis for syntactic and semantic operations

Formalism: Syntax Definition

- Productions + Constructors + Templates + Disambiguation
- Language-specific rules: structure of each language construct

Language-Independent Interpretation

- Well-formedness of abstract syntax trees
  - provides declarative correctness criterion for parsing
- Parsing algorithm
  - No need to understand parsing algorithm
  - Debugging in terms of representation
- Formatting based on layout hints in grammar
- Syntactic completion

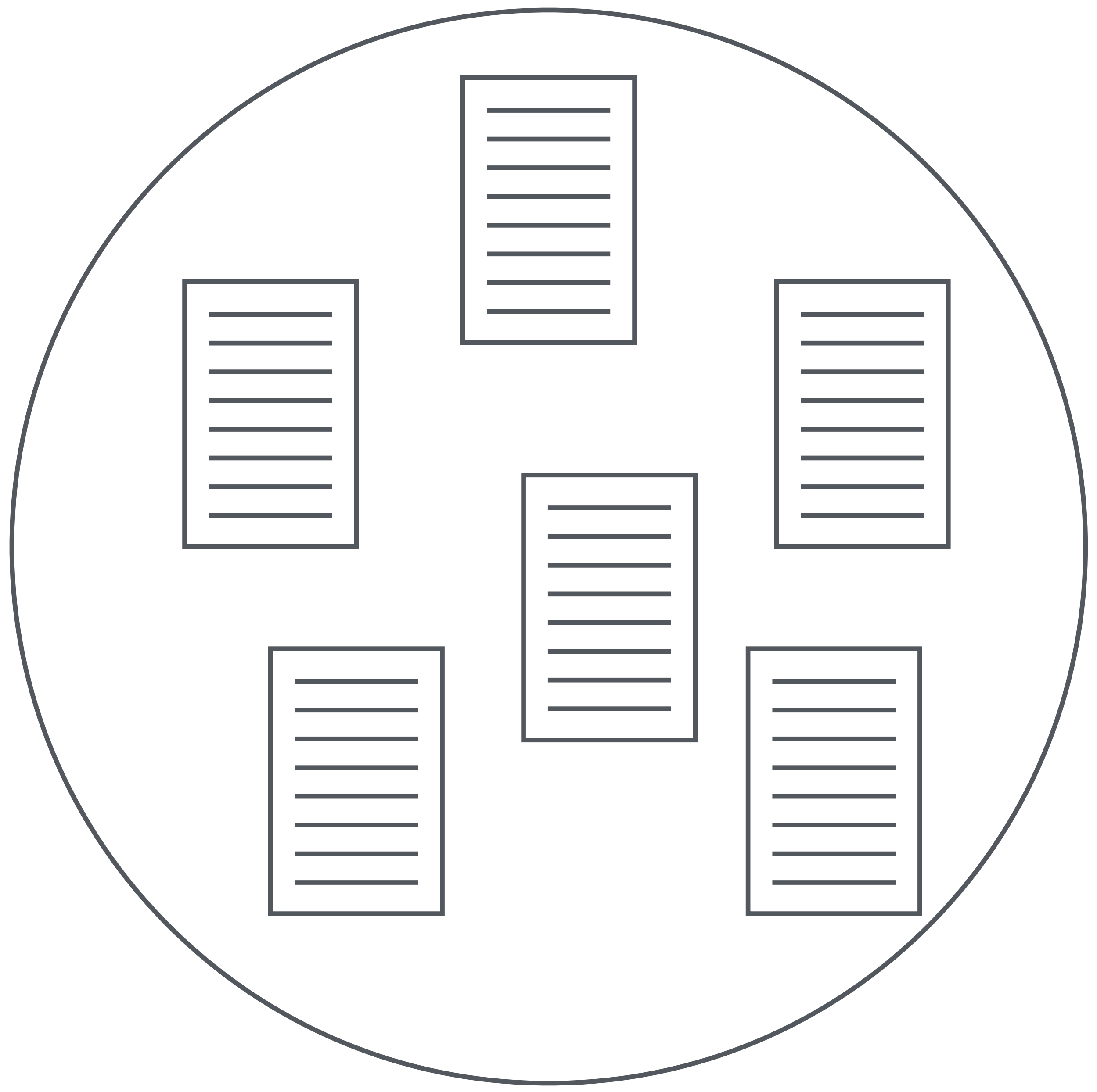
A meta-language for talking about syntax

Formal Languages & Grammars

The set of **rules**, principles, and processes that govern the structure of sentences in a given language

TU Delft

Language = Set of Sentences



```
fun (x : Int) { x + 1 }
```

Text is a convenient interface for writing and reading programs

Formal Language

Vocabulary Σ

- finite, nonempty set of elements (words, letters)
- alphabet

String over Σ

- finite sequence of elements chosen from Σ
- word, sentence, utterance, program

Formal language λ

- set of strings over a vocabulary Σ
- $\lambda \subseteq \Sigma^*$

Formal Grammar

Formal grammar G

Derivation relation $\Rightarrow_G$

Formal language $L(G) \subseteq \Sigma^*$

- $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

Decimal Numbers

G = (N, Σ, P, S)

Num = Digit Num
Num = Digit
Digit = “0”
Digit = “1”
Digit = “2”
Digit = “3”
Digit = “4”
Digit = “5”
Digit = “6”
Digit = “7”
Digit = “8”
Digit = “9”

Decimal Numbers

G = (N, Σ, P, S)

Num = Digit Num
Num = Digit
Digit = "0"
Digit = "1"
Digit = "2"
Digit = "3"
Digit = "4"
Digit = "5"
Digit = "6"
Digit = "7"
Digit = "8"
Digit = "9"

Σ: finite set of terminal symbols

Decimal Numbers

G = (N, Σ, P, S)

Num = Digit Num
Num = Digit
Digit = "0"
Digit = "1"
Digit = "2"
Digit = "3"
Digit = "4"
Digit = "5"
Digit = "6"
Digit = "7"
Digit = "8"
Digit = "9"

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

Decimal Numbers

G = (N, Σ, P, S)

Num = Digit Num
Num = Digit
Digit = “0”
Digit = “1”
Digit = “2”
Digit = “3”
Digit = “4”
Digit = “5”
Digit = “6”
Digit = “7”
Digit = “8”
Digit = “9”

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

S∈N: start symbol

Decimal Numbers

$G = (N, \Sigma, P, S)$

Num = Digit Num
Num = Digit
Digit = "0"
Digit = "1"
Digit = "2"
Digit = "3"
Digit = "4"
Digit = "5"
Digit = "6"
Digit = "7"
Digit = "8"
Digit = "9"

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

$S \in N$: start symbol

$P \subseteq N \times (N \cup \Sigma)^*$: set of production rules

Decimal Numbers: Production

G = (N, Σ, P, S)

Num = Digit Num
Num = Digit
Digit = “0”
Digit = “1”
Digit = “2”
Digit = “3”
Digit = “4”
Digit = “5”
Digit = “6”
Digit = “7”
Digit = “8”
Digit = “9”

Num ⇒	Num = Digit Num
Digit Num ⇒	Digit = “4”
4 Num ⇒	Num = Digit Num
4 Digit Num ⇒	Digit = “3”
4 3 Num ⇒	Num = Digit Num
4 3 Digit Num ⇒	Digit = “0”
4 3 0 Num ⇒	Num = Digit
4 3 0 Digit ⇒	Digit = “3”
4 3 0 3	

leftmost derivation

Decimal Numbers: Production

$G = (N, \Sigma, P, S)$

Num = Digit Num
Num = Digit
Digit = "0"
Digit = "1"
Digit = "2"
Digit = "3"
Digit = "4"
Digit = "5"
Digit = "6"
Digit = "7"
Digit = "8"
Digit = "9"

Num ⇒	Num = Digit Num
Digit Num ⇒	Num = Digit Num
Digit Digit Num ⇒	Num = Digit Num
Digit Digit Digit Num ⇒	Num = Digit
Digit Digit Digit Digit ⇒	Digit = "3"
Digit Digit Digit 3 ⇒	Digit = "0"
Digit Digit 0 3 ⇒	Digit = "3"
Digit 3 0 3 ⇒	Digit = "4"
4 3 0 3	

rightmost derivation

Leftmost vs Rightmost Derivation

Step

- replace non-terminal symbol in string by symbols in rhs of production

Derivation

- repeatedly apply steps starting with start symbol
- until string consists only of terminal symbols

Leftmost Derivation

- replace leftmost non-terminal symbol in string

Rightmost Derivation

- replace rightmost non-terminal in string

Binary Expressions

G = (N, Σ, P, S)

Exp = Num
Exp = Exp "+" Exp
Exp = Exp "−" Exp
Exp = Exp "*" Exp
Exp = Exp "/" Exp
Exp = "(" Exp ")"

Binary Expressions

G = (N, Σ, P, S)

Exp = Num
Exp = Exp "+" Exp
Exp = Exp "−" Exp
Exp = Exp "*" Exp
Exp = Exp "/" Exp
Exp = "(" Exp ")"

Σ: finite set of terminal symbols

Binary Expressions

G = (N, Σ, P, S)

Exp = Num
Exp = Exp "+" Exp
Exp = Exp "−" Exp
Exp = Exp "*" Exp
Exp = Exp "/" Exp
Exp = "(" Exp ")"

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

Binary Expressions

G = (N, Σ, P, S)

Exp = Num
Exp = Exp "+" Exp
Exp = Exp "−" Exp
Exp = Exp "*" Exp
Exp = Exp "/" Exp
Exp = "(" Exp ")"

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

S∈N: start symbol

Binary Expressions

$G = (N, \Sigma, P, S)$

Exp = Num
Exp = Exp "+" Exp
Exp = Exp "−" Exp
Exp = Exp "*" Exp
Exp = Exp "/" Exp
Exp = "(" Exp ")"

Σ: finite set of terminal symbols

N: finite set of non-terminal symbols

$S \in N$: start symbol

$P \subseteq N \times (N \cup \Sigma)^*$: set of production rules

Binary Expressions

Exp ⇒	Exp = Exp "+" Exp
Exp + Exp ⇒	Exp = Exp "*" Exp
Exp + Exp * Exp ⇒	Exp = Num
3 + Exp * Exp ⇒	Exp = Num
3 + 4 * Exp ⇒	Exp = Num
3 + 4 * 5	

Generative Grammars

Formal grammar G = (N, Σ, P, S)

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq \underbrace{\boxed{(N\cup\Sigma)^*}}_{\text{context}}\,\overbrace{\boxed{N}}^{\text{nonterminal symbol}}\,\underbrace{\boxed{(N\cup\Sigma)^*}}_{\text{context}} \times \underbrace{\boxed{(N\cup\Sigma)^*}}_{\text{replacement}}$
- start symbol $S \in N$

Chomsky Hierarchy

a, b, c range over $(N \cup \Sigma)^*$

A, B range over N

Type-0: Unrestricted

- $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $a = b$

Type-1: Context-sensitive

- $P \subseteq (N \cup \Sigma)^* \, N \, (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $a \, A \, c = a \, b \, c$

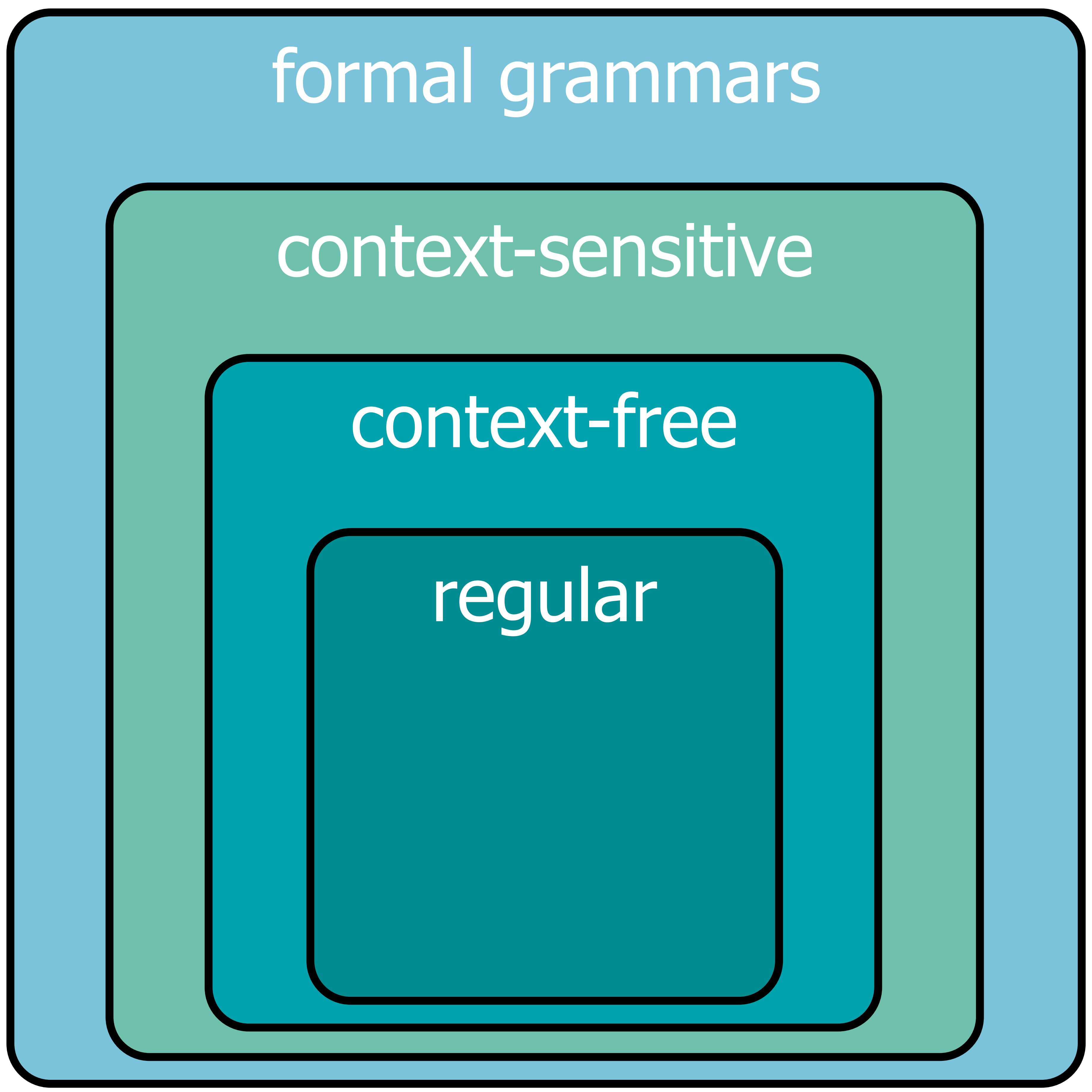
Type-2: Context-free

- $P \subseteq N \times (N \cup \Sigma)^*$
- $A = b$

Type-3: Regular

- $P \subseteq N \times (\Sigma \cup \Sigma N)$
- $A = x$ or $A = xB$

Chomsky Hierarchy



Formal Grammars

Formal grammar $G = (N, \Sigma, P, S)$

Derivation relation $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

$w \Rightarrow_G w' \Leftrightarrow$

$\exists (p, q) \in P: \exists u, v \in (N \cup \Sigma)^*:$

$w = u\ p\ v \wedge w' = u\ q\ v$

Formal language $L(G) \subseteq \Sigma^*$

$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

Recognition: Word Problem

Generation

- Given a grammar, generate a sentence in the language

Language

- The set of all sentences generated by the grammar

Recognition

- Is this sentence (word) in the language generated by the grammar?
- Is there an algorithm for deciding that question?

Recognition: Word Problem

Word problem $\chi L: \Sigma^* \rightarrow \{0,1\}$

- w → 1, if w∈L
- w → 0, else

Decidability

- type-0: semi-decidable
- type-1, type-2, type-3: decidable

Complexity

- type-1: PSPACE-complete
- type-2, type-3: P

Recognizing Binary Expressions

3 + 4 * 5 ⇒	Exp = Num
Exp + 4 * 5 ⇒	Exp = Num
Exp + Exp * 5 ⇒	Exp = Num
Exp + Exp * Exp ⇒	Exp = Exp "*" Exp
Exp + Exp ⇒	Exp = Exp "+" Exp
Exp	

Replace substring that matches right-hand side of production with left-hand-side

The Syntax Definition Formalism SDF3

A human readable, machine processable notation for declarative syntax definition

TU Delft

Productions

```
module Numbers

imports NumberConstants

context-free syntax

  Exp = IntConst

  Exp = Exp "+" Exp
  Exp = Exp "-" Exp
  Exp = Exp "*" Exp
  Exp = Exp "/" Exp

  Exp = Exp "="  Exp
  Exp = Exp "<>" Exp
  Exp = Exp ">"  Exp
  Exp = Exp "<"  Exp
  Exp = Exp ">=" Exp
  Exp = Exp "<=" Exp

  Exp = Exp "&"  Exp
  Exp = Exp "|"  Exp
```

```
1 + 3 <= 4 - 35 & 12 > 16
```

Encoding Sequences (Lists)

```
function printboard() = (
  for i := 0 to N-1 do (
    for j := 0 to N-1 do
      print(if col[i]=j then " O" else " .");
    print("\n")
  );
  print("\n")
)
```

```
module Control-Flow

imports Identifiers
imports Variables

context-free syntax

  Exp = "(" ExpList ")"
  Exp = "if" Exp "then" Exp "else" Exp
  Exp = "if" Exp "then" Exp
  Exp = "while" Exp "do" Exp
  Exp = "for" Var ":=" Exp "to" Exp "do" Exp

  ExpList = // empty list
  ExpList = ExpList ";" Exp
```

Sugar for Sequences and Optionals

```
context-free syntax

  Exp = "(" {Exp ";"}* ")"

context-free syntax

  // automatically generated

  {Exp ";"}* = // empty list
  {Exp ";"}* = {Exp ";"}* ";" Exp

  {Exp ";"}+ = Exp
  {Exp ";"}+ = {Exp ";"}+ ";" Exp

  Exp* = // empty list
  Exp* = Exp* Exp

  Exp+ = Exp
  Exp+ = Exp+ Exp

  Exp? =     // no expression
  Exp? = Exp // one expression
```

```
function printboard() = (
  for i := 0 to N-1 do (
    for j := 0 to N-1 do
      print(if col[i]=j then " O" else " .");
    print("\n")
  );
  print("\n")
)
```

Using Sugar for Sequences

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec    = FunDec+
  FunDec = "function" Id "(" {FArg ","}* ")" "=" Exp
  FunDec = "function" Id "(" {FArg ","}* ")" ":" Type "=" Exp
  FArg   = Id ":" Type
  Exp    = Id "(" {Exp ","}* ")"
```

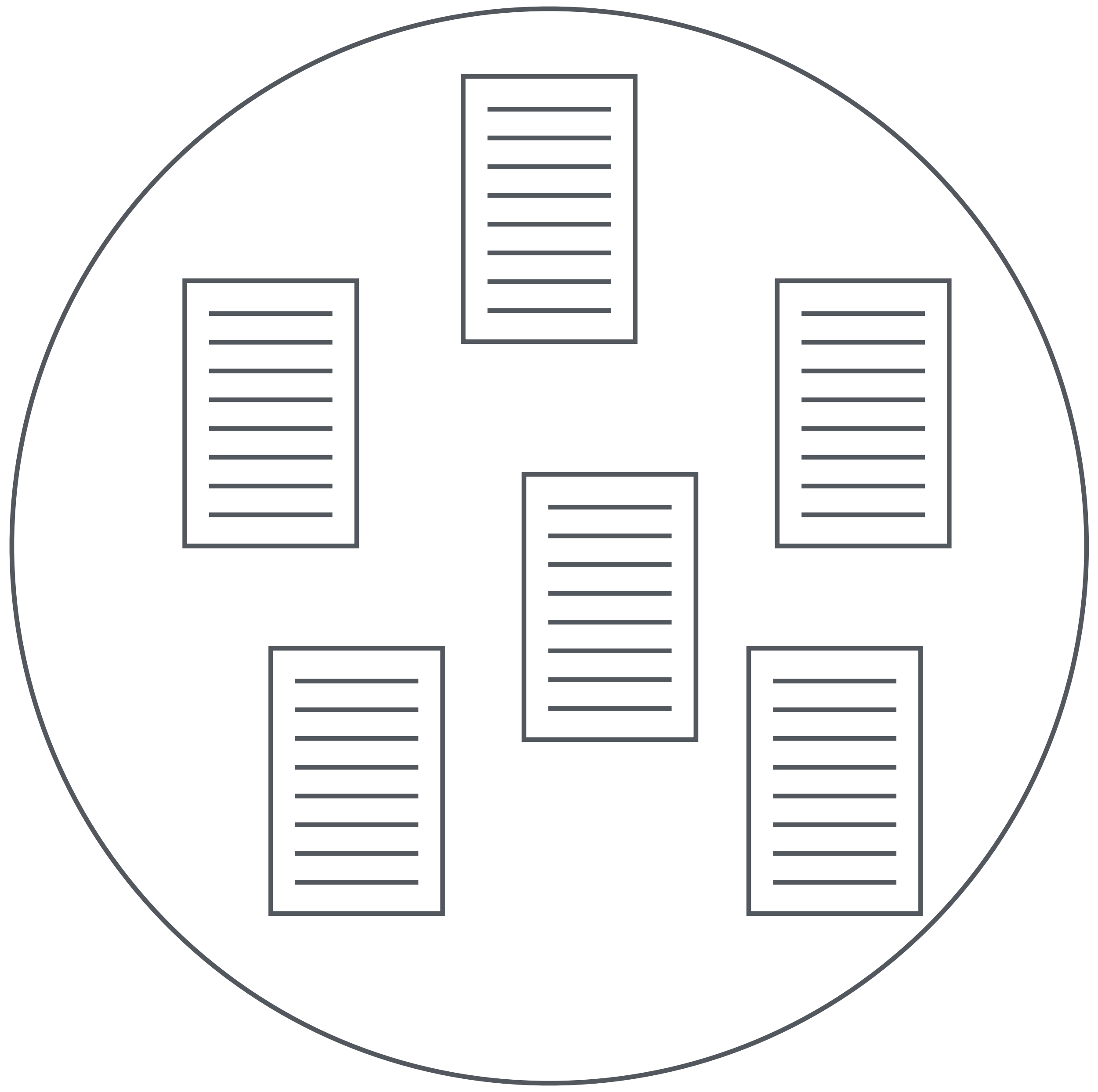
```
let function power(x: int, n: int): int =
      if n <= 0 then 1
      else x * power(x, n - 1)
 in power(3, 10)
end
```

```
module Bindings

imports Control-Flow Identifiers Types Functions Variables

sorts Declarations
context-free syntax

  Exp          = "let" Dec* "in" {Exp ";"}* "end"
  Declarations = "declarations" Dec*
```

Structure

The set of rules, principles, and processes that govern the **structure** of sentences in a given language

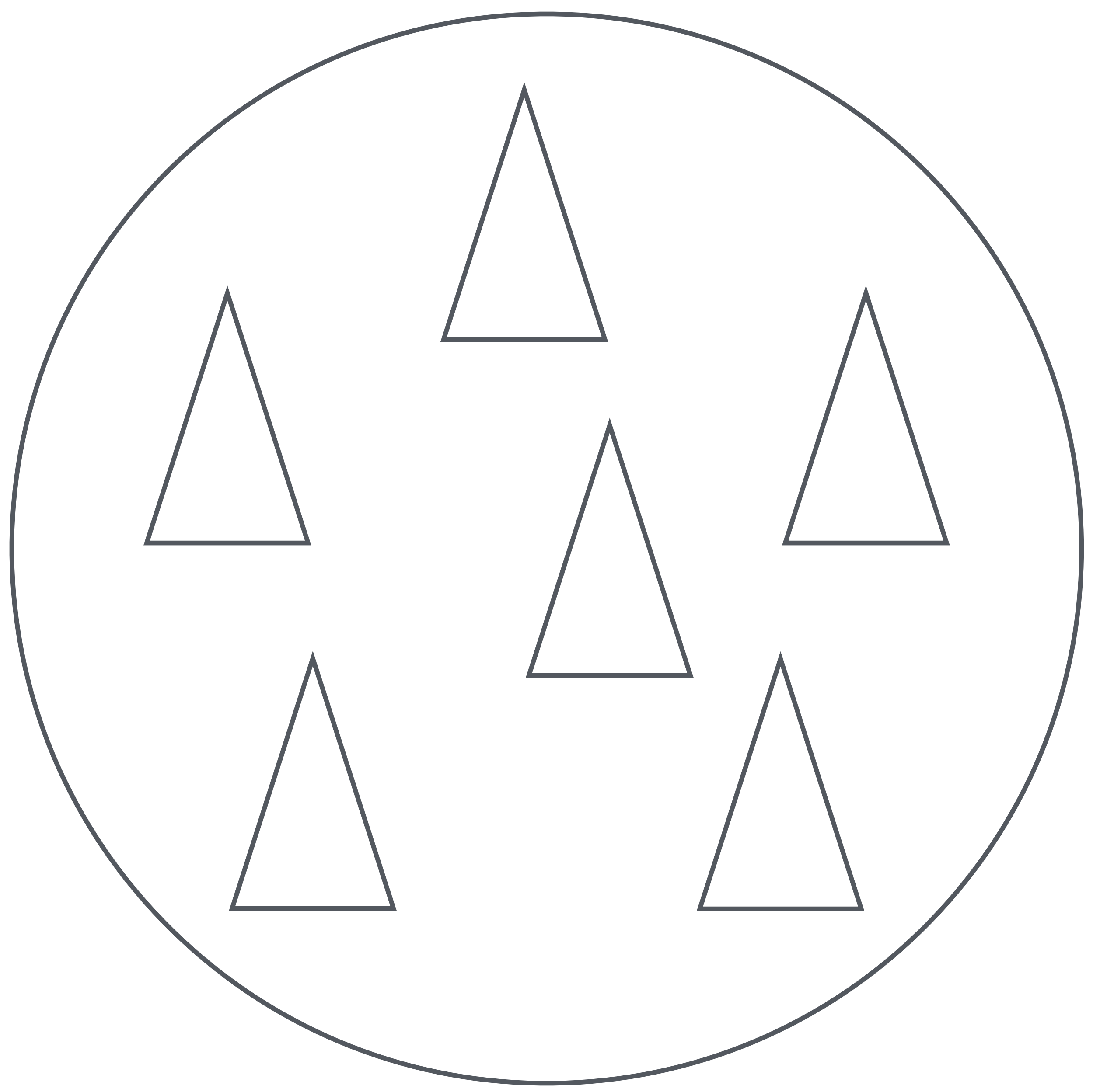
Language = Set of Sentences

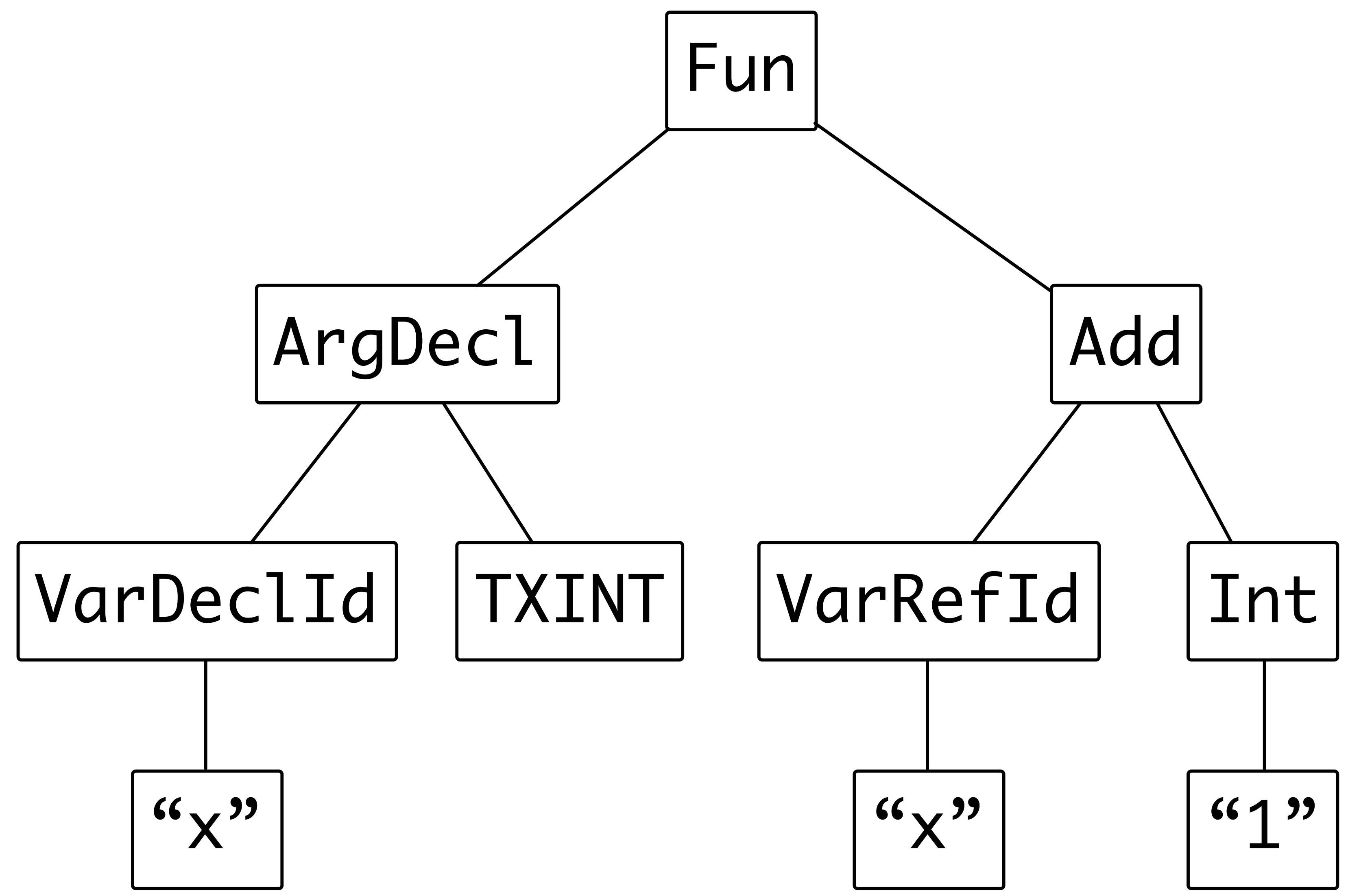


```
fun (x : Int) { x + 1 }
```

Text is a convenient interface for writing and reading programs

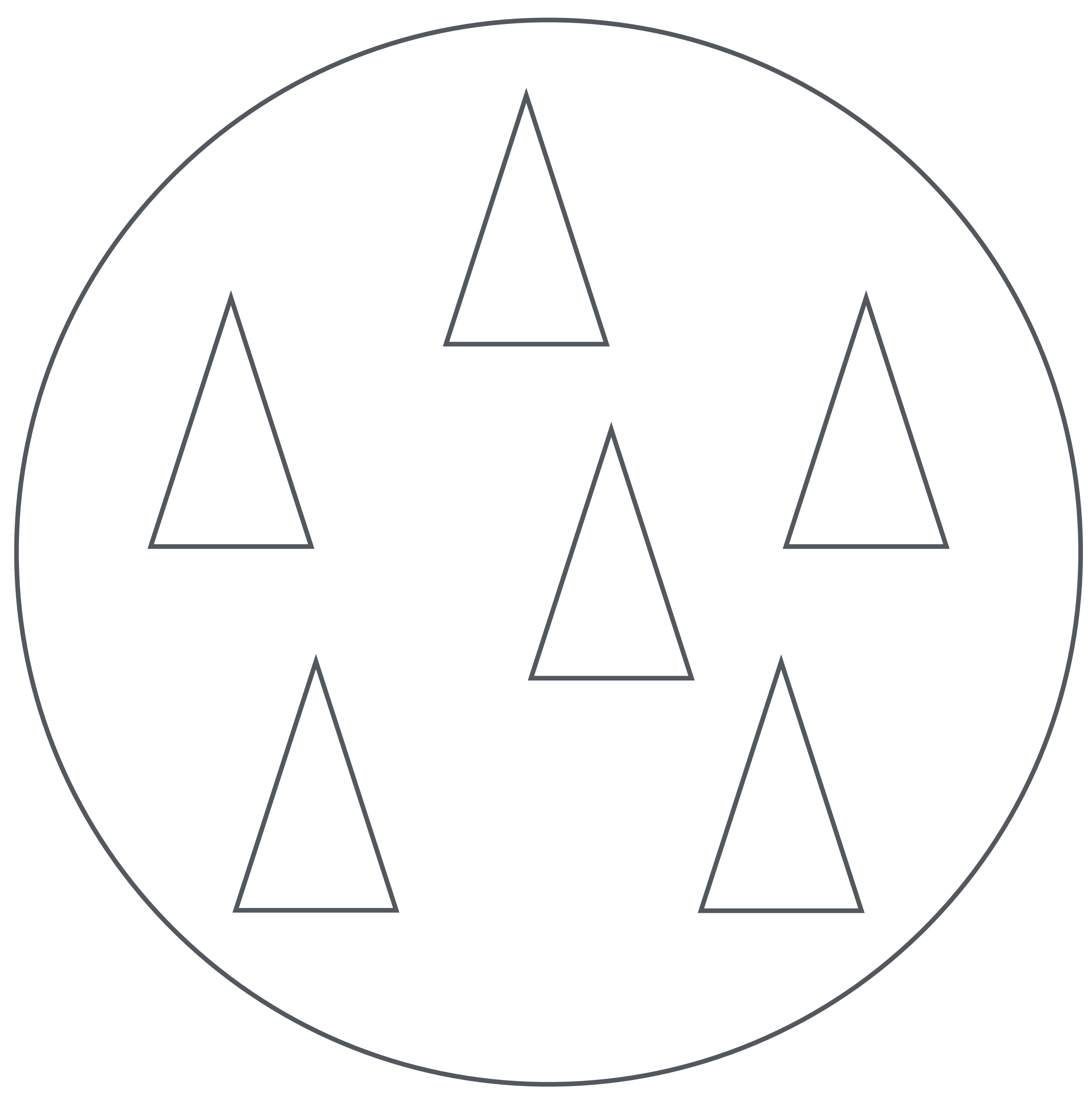
Language = Set of Trees





Tree is a convenient interface for transforming programs

Tree Transformation

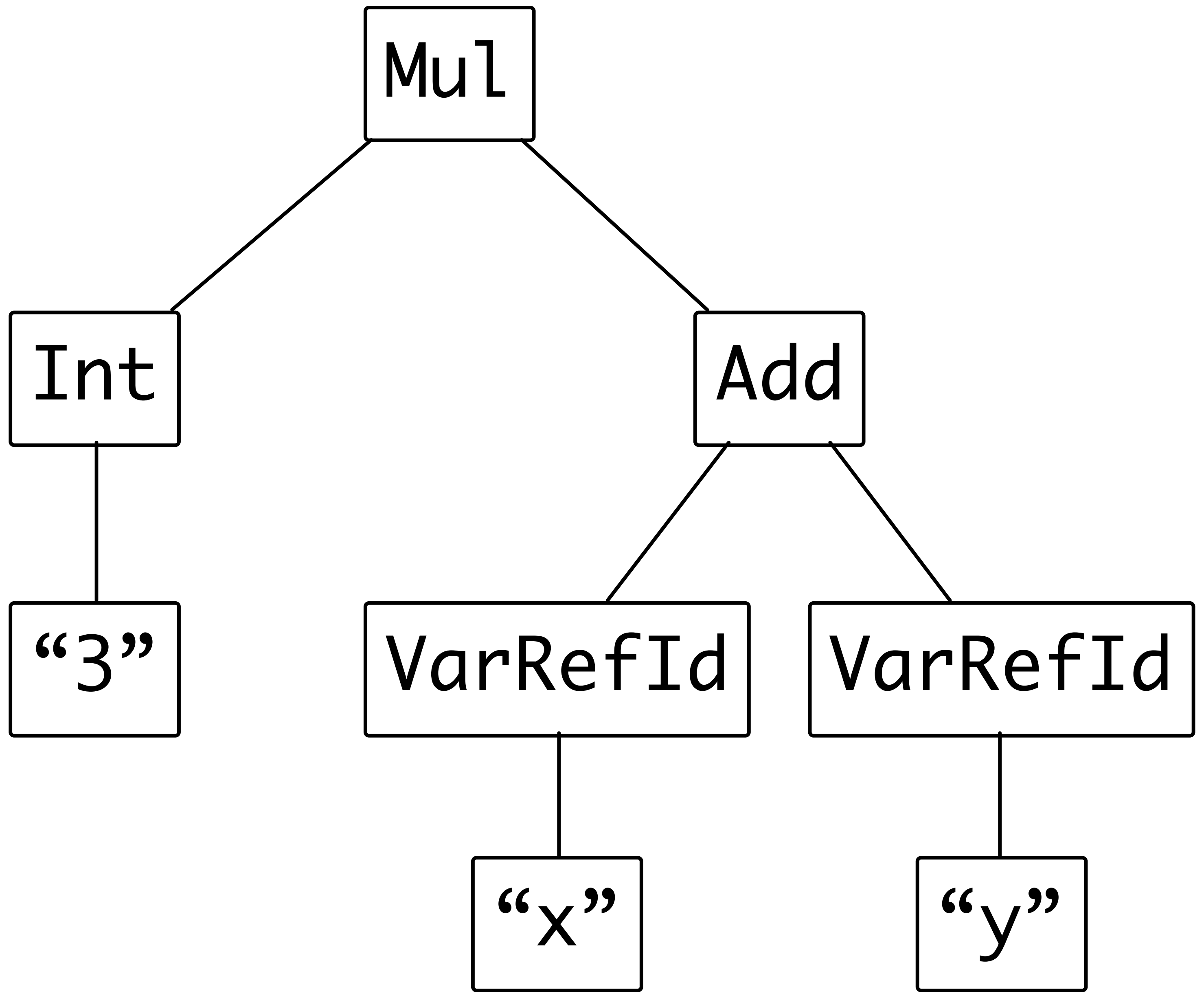


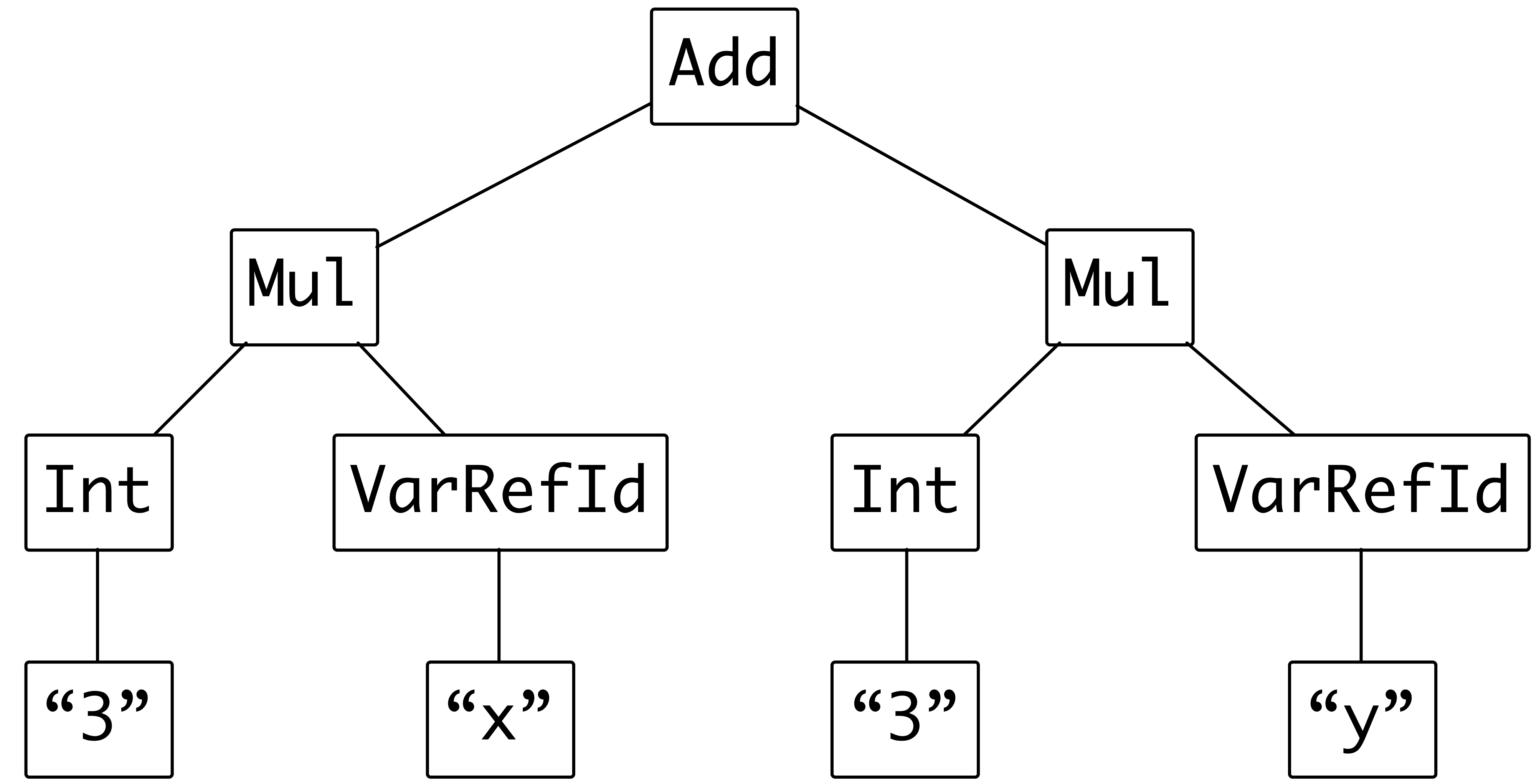
Syntactic
coloring
outline view
completion

Semantic
transform
translate
eval
analyze
refactor
type check

Tree is a convenient interface for transforming programs

Tree Transformation






```
Mul(Int("3"),
    Add(VarRefId("x"),
        VarRefId("y")))
```

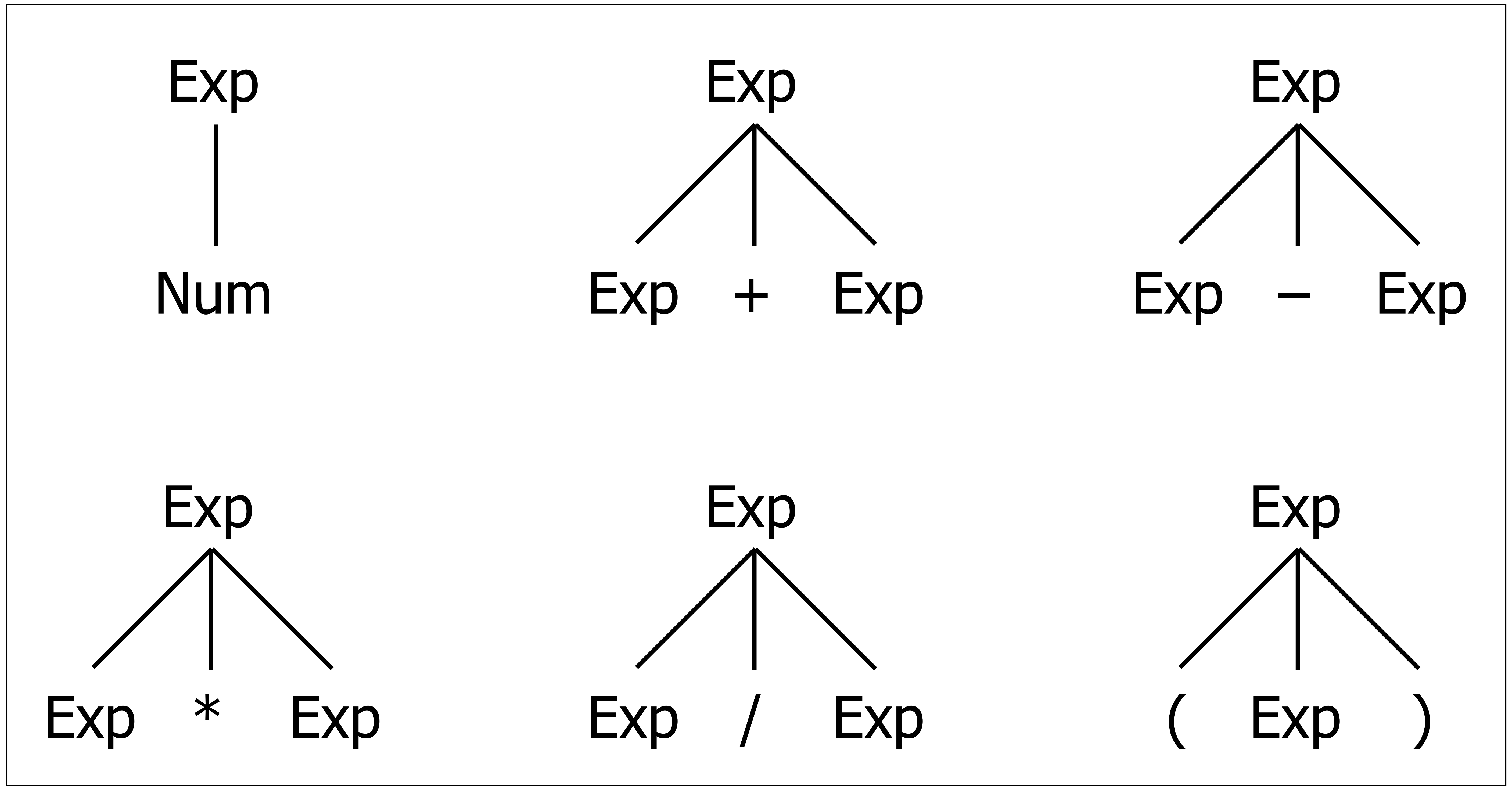
```
Add(Mul(Int("3"),
        VarRefId("x")),
    Mul(Int("3"),
        VarRefId("y")))
```

```
Mul(e1, Add(e2, e3)) -> Add(Mul(e1, e2), Mul(e1, e3))
```

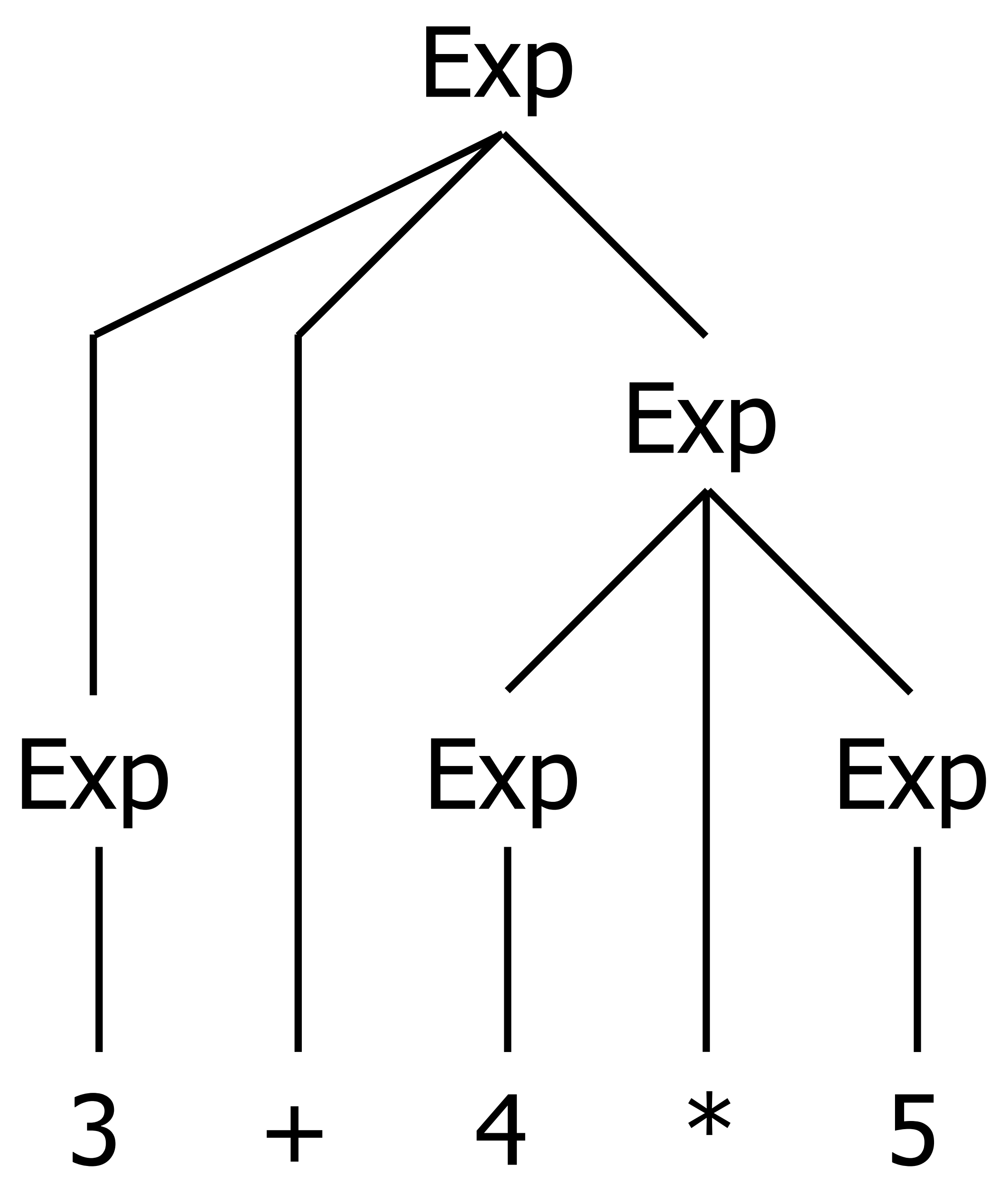
Context-free Grammar = Tree Construction Rules

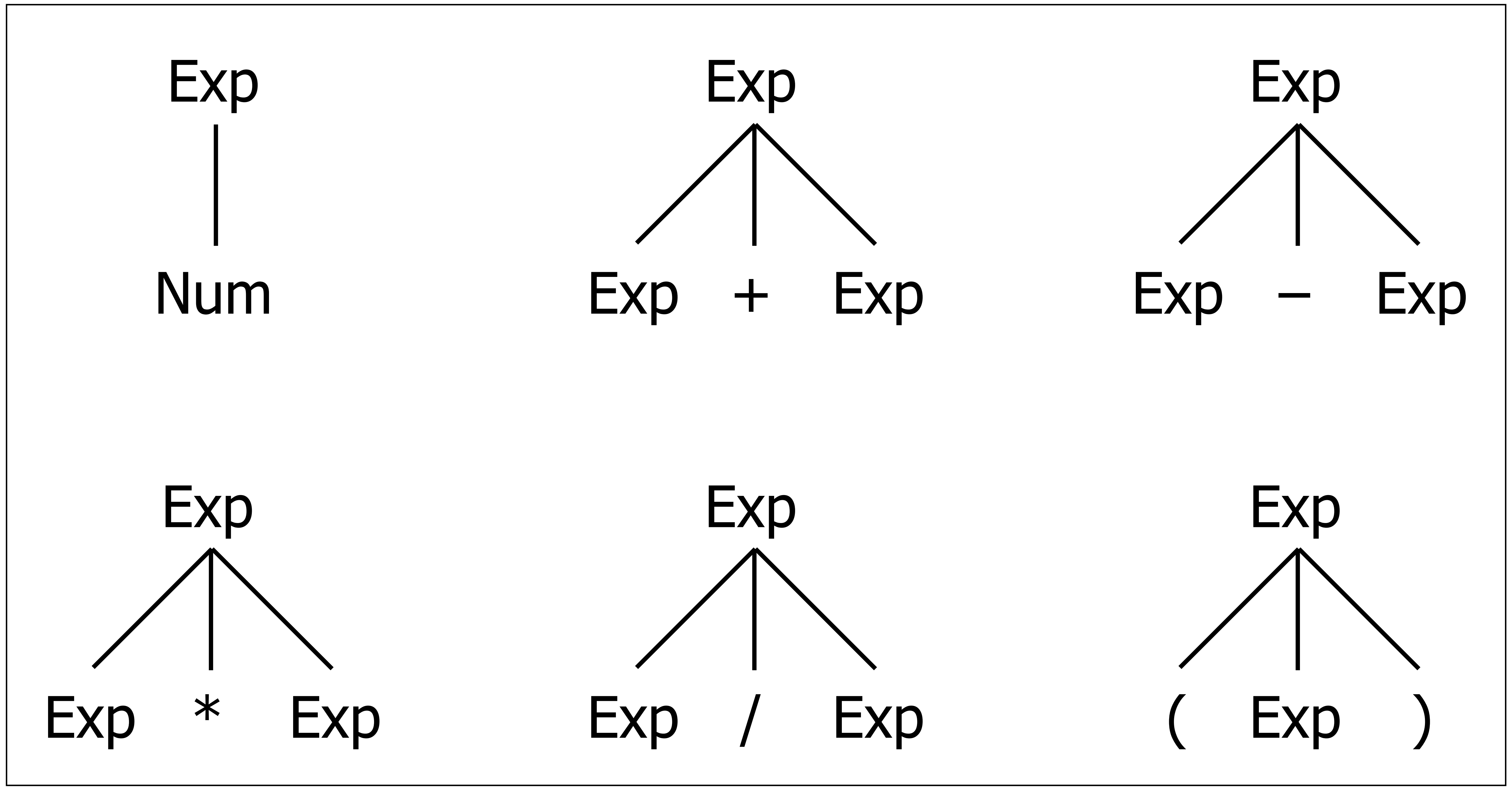
```
Exp = Num
Exp = Exp “+” Exp
Exp = Exp “–” Exp
Exp = Exp “*” Exp
Exp = Exp “/” Exp
Exp = “(” Exp “)”
```

=



Tree Construction





Parse Tree vs Abstract Syntax Tree

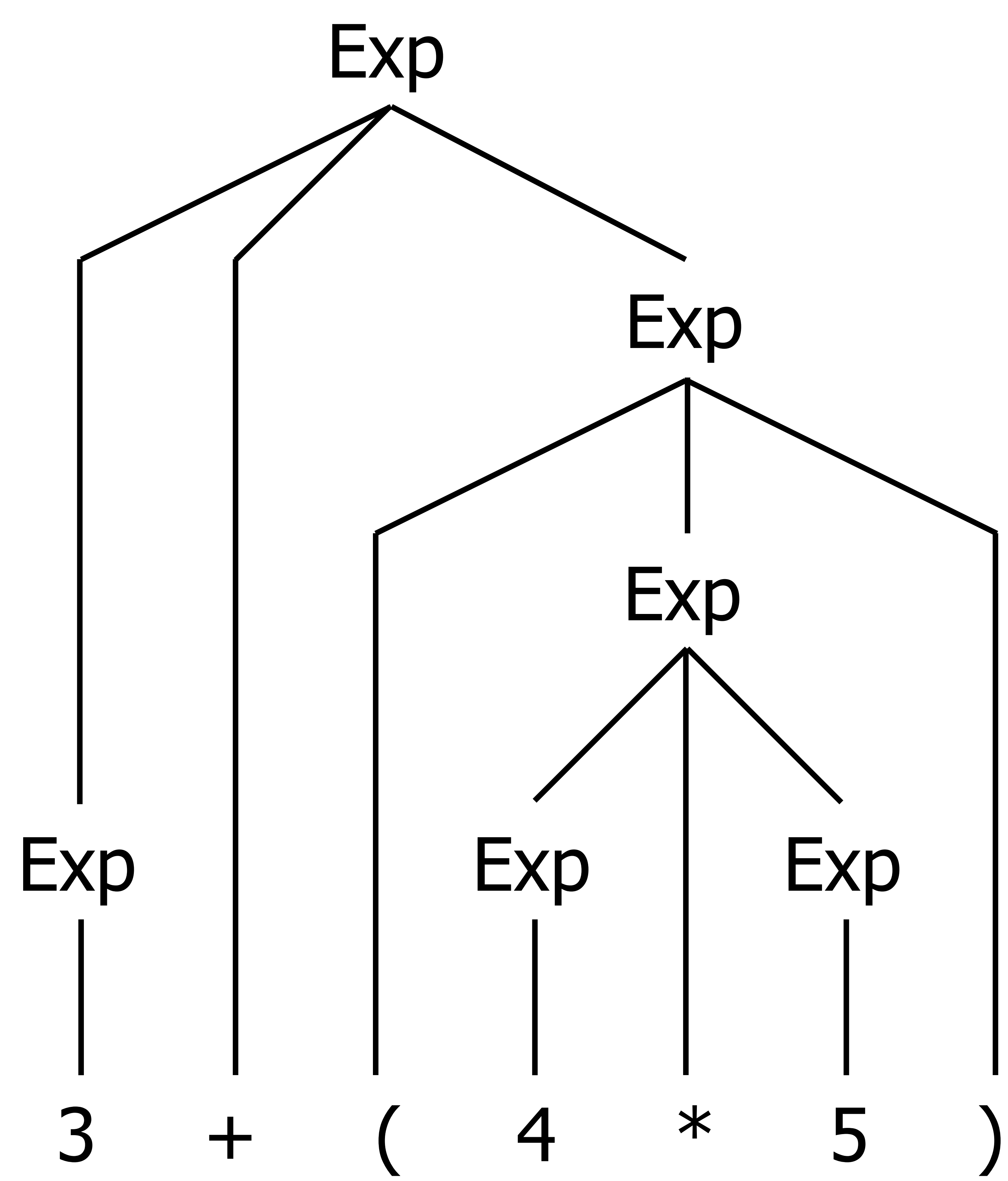
Parse trees

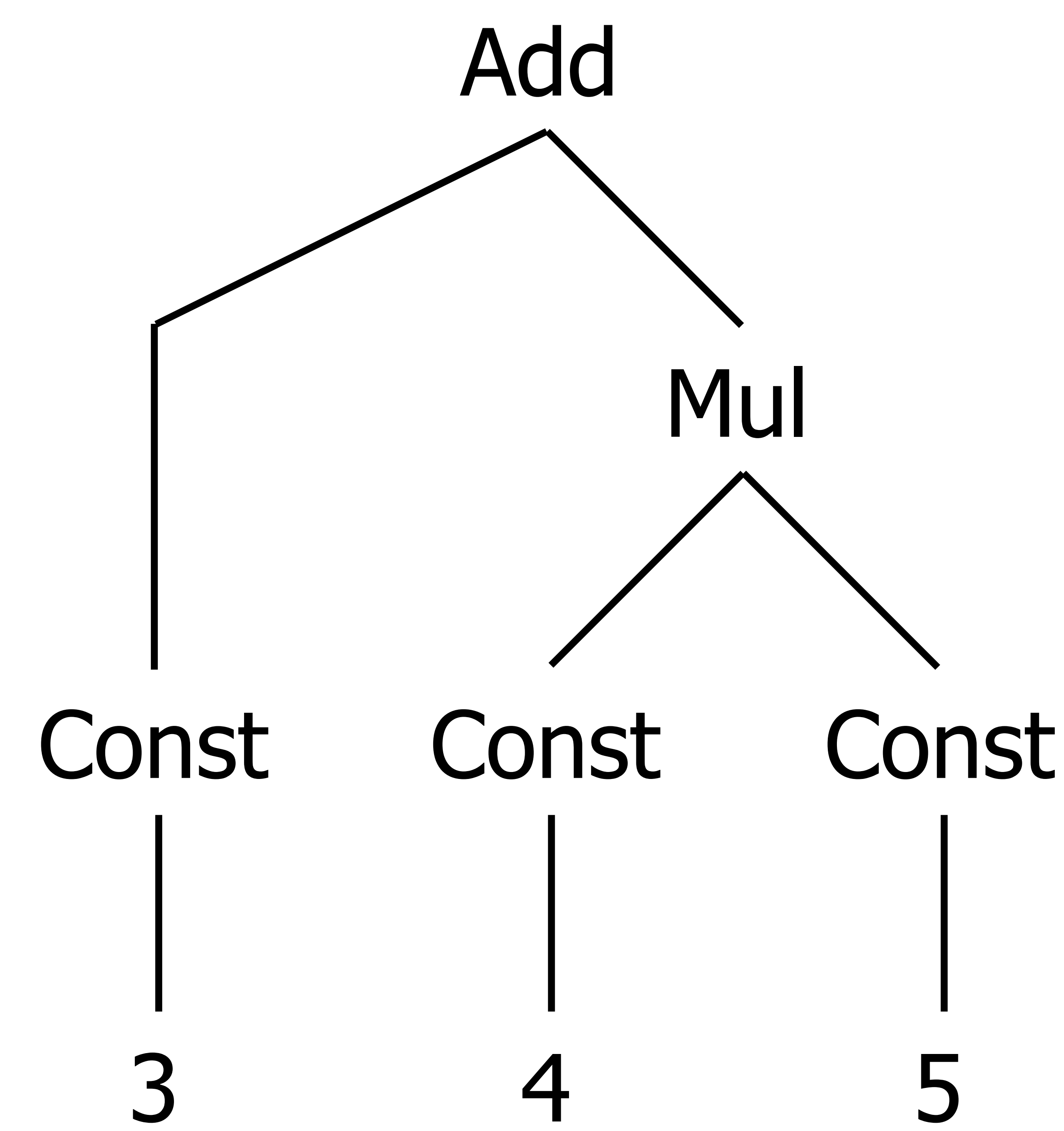
- interior nodes: nonterminal symbol
- leaf nodes: terminal symbols

Abstract syntax trees (ASTs)

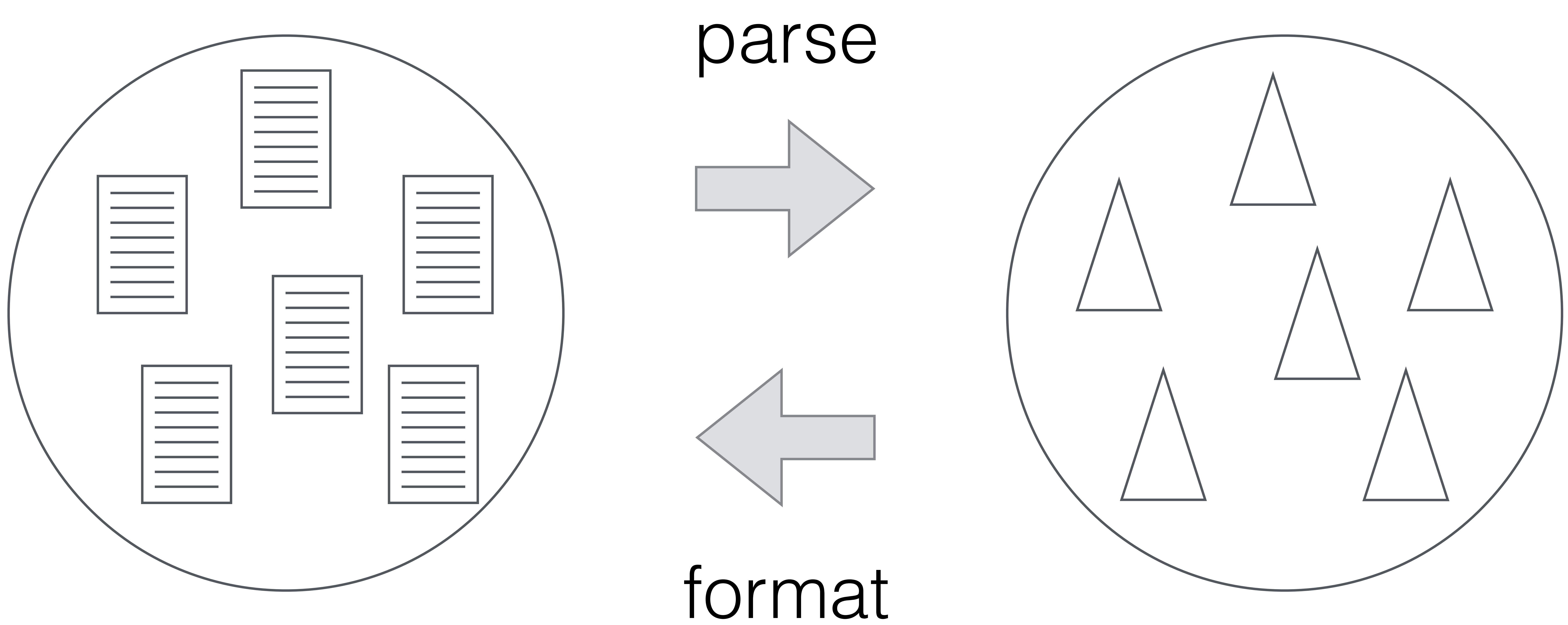
- abstract over terminal symbols
- convey information at nodes
- abstract over injective production rules

Parse Tree vs Abstract Syntax Tree



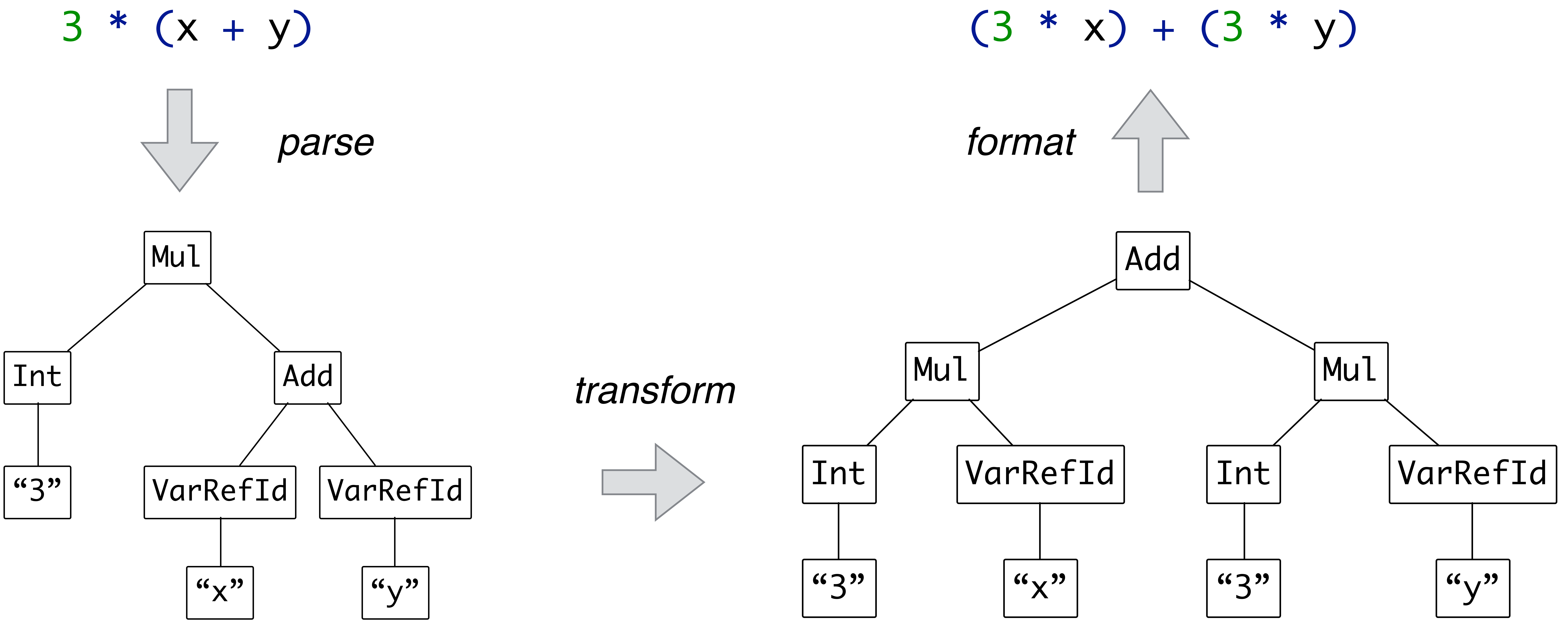


Language = Sentences and Trees



Different representations convenient for different purposes

From Text to Tree and Back



SDF3 Productions Define Trees *and* Sentences

```
Exp.Int = INT
Exp.Add = Exp "+" Exp
Exp.Mul = Exp "*" Exp
```

format (tree to text)	+	trees (structure)	=>	parse (text to tree)

```
parse(s) = t  where format(t) == s (modulo layout)
```

Basic Terms

Term is defined inductively as

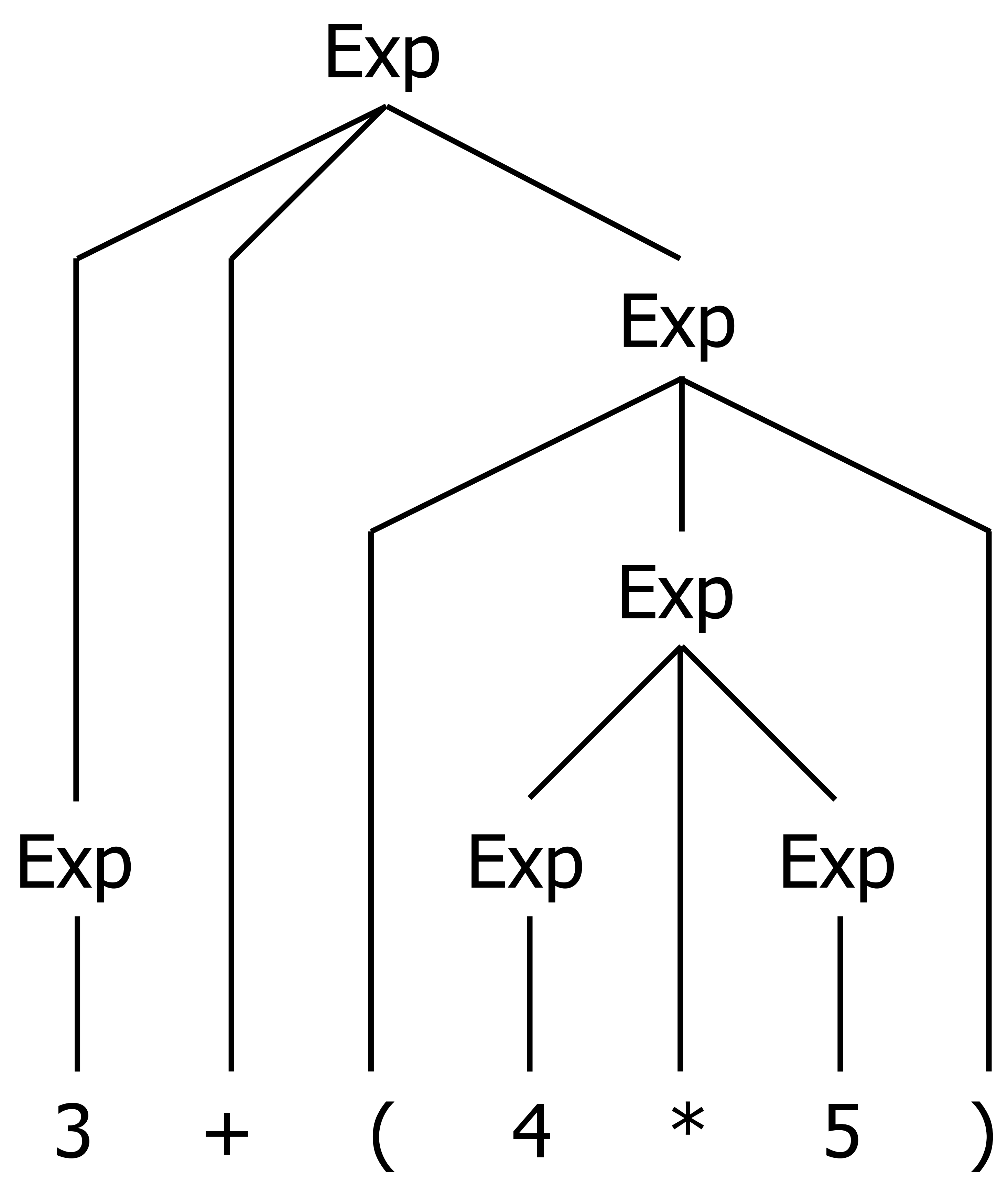
- If $t_1, \ldots, t_n$ is a term, then $c(t_1, \ldots, t_n)$ is a term, where c is a constructor symbol

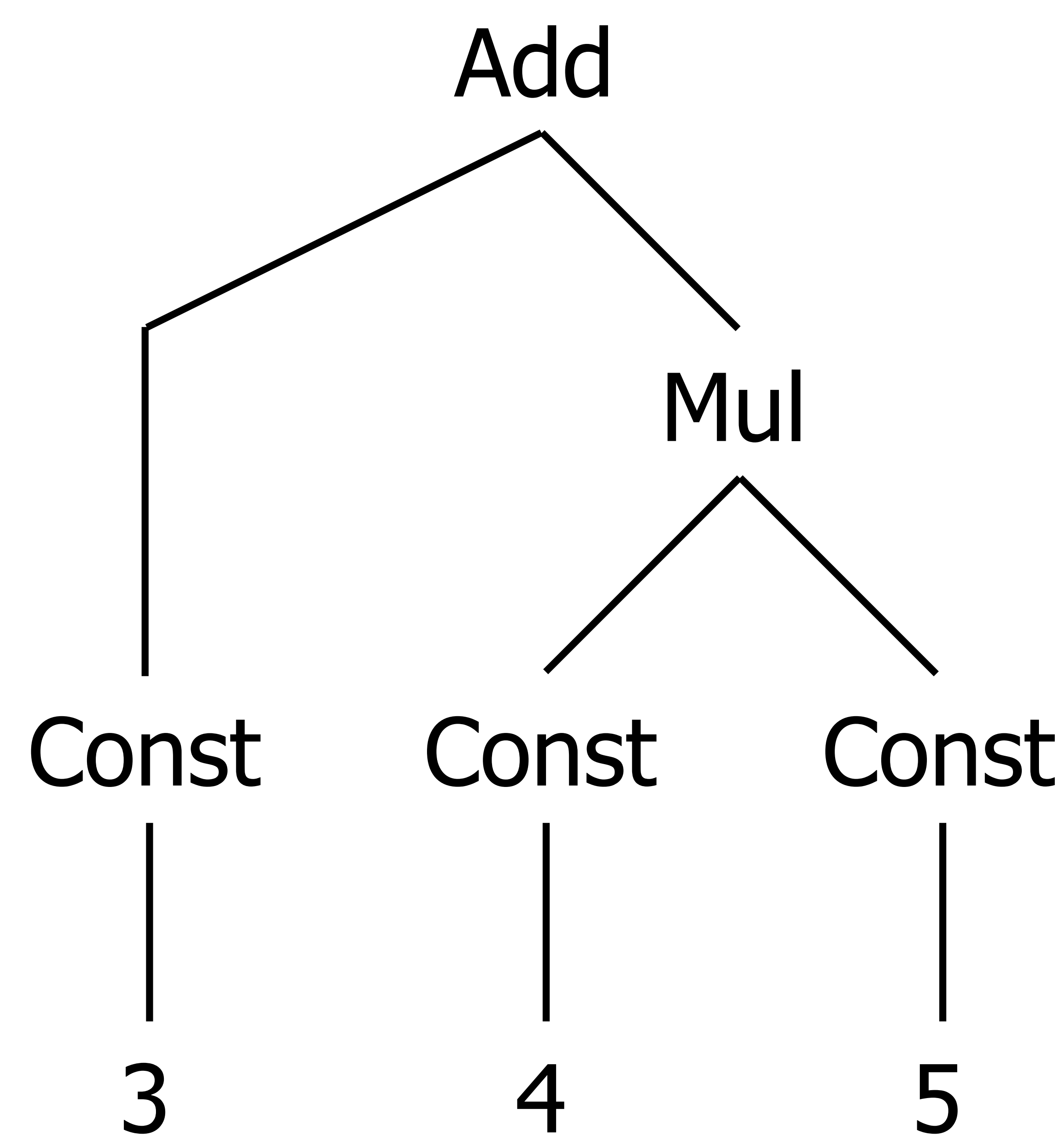
Terms

Term is defined inductively as

- A string literal is a term
- An integer constant is a term
- If t_1, …, t_n are terms, then
  - $c(t_1, \ldots, t_n)$ is a term, where c is a constructor symbol
  - $(t_1, \ldots, t_n)$ is a term (tuple)
  - $[t_1, \ldots, t_n]$ is a term (list)

Parse Tree vs Abstract Syntax Tree vs Term






```
Add(
  Const(3),
  Mul(
    Const(4),
    Const(5)))
```

Productions with Constructors

```
context-free syntax

  Exp.Int     = IntConst
  Exp.Uminus  = "-" Exp
  Exp.Times   = Exp "*" Exp
  Exp.Divide  = Exp "/" Exp
  Exp.Plus    = Exp "+" Exp
  Exp.Minus   = Exp "-" Exp
  Exp.Eq      = Exp "=" Exp
  Exp.Neq     = Exp "<>" Exp
  Exp.Gt      = Exp ">" Exp
  Exp.Lt      = Exp "<" Exp
  Exp.Geq     = Exp ">=" Exp
  Exp.Leq     = Exp "<=" Exp
  Exp.And     = Exp "&" Exp
  Exp.Or      = Exp "|" Exp
```

```
1 + 3 <= 4 - 35 & 12 > 16
```

```
And(
  Leq(
    Plus(Int("1"), Int("3"))
  , Minus(Int("4"), Int("35"))
  )
, Gt(Int("12"), Int("16"))
)
```

List Terms

```
(
  for j := 0 to N-1 do
    print(if col[i]=j then " O" else " .");
  print("\n")
)
```

```
Seq(
  [ For(
      Var("j")
    , Int("0")
    , Minus(Var("N"), Int("1"))
    , Call(
        "print"
      , [ If(
            Eq(Subscript(Var("col"), Var("i")), Var("j"))
          , String("\" O\"")
          , String("\" .\"")
          )
        ]
      )
    )
  , Call("print", [String("\"\\n\"")])
  ]
)
```

```
module Control-Flow

imports Identifiers
imports Variables

context-free syntax

  Exp.Seq     = "(" {Exp ";"}* ")"
  Exp.If      = "if" Exp "then" Exp "else" Exp
  Exp.IfThen  = "if" Exp "then" Exp
  Exp.While   = "while" Exp "do" Exp
  Exp.For     = "for" Var ":=" Exp "to" Exp "do" Exp
  Exp.Break   = "break"
```

More List Terms

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec     = FunDec+
  FunDec = "function" Id "(" {FArg ","}* ")" "=" Exp
  FunDec = "function" Id "(" {FArg ","}* ")" ":" Type "=" Exp
  FArg    = Id ":" Type
  Exp     = Id "(" {Exp ","}* ")"
```

```
function power(x: int, n: int): int =
  if n <= 0 then 1
  else x * power(x, n - 1)
```

```
FunDec(
  "power"
  , [FArg("x", Tid("int")), FArg("n", Tid("int"))]
  , Tid("int")
  , If(Leq(Var("n"), Int("0"))
      , Int("1")
      , Times(Var("x")
             , Call(
                 "power"
                 , [Var("x"), Minus(Var("n"), Int("1"))]
               )
        )
    )
)
```

Testing Syntax Definitions with SPT

Are you defining the right language?

Test Suites

```
module example-suite

language Tiger
start symbol Start

test name
  [[...]]
  parse succeeds

test another name
  [[...]]
  parse fails
```

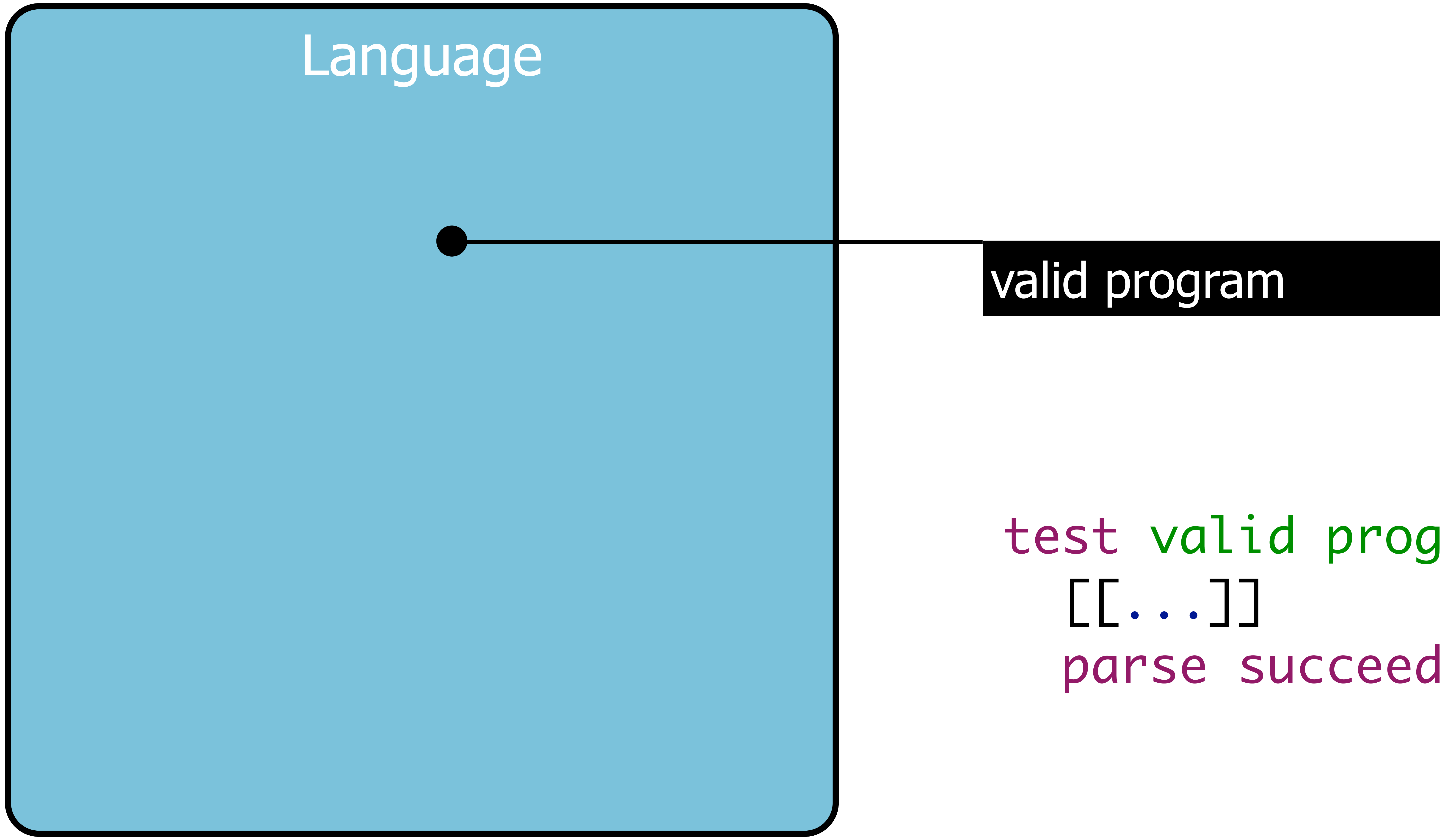
SPT: SPoofax Testing language

Success is Failure, Failure is Success, …

```
module success-failure

language Tiger

test this test succeeds [[
  1 + 3 * 4
]] parse succeeds

test this test fails [[
  1 + 3 * 4
]] parse fails

test this test fails [[
  1 + 3 *
]] parse succeeds

test this test succeeds [[
  1 + 3 *
]] parse fails
```

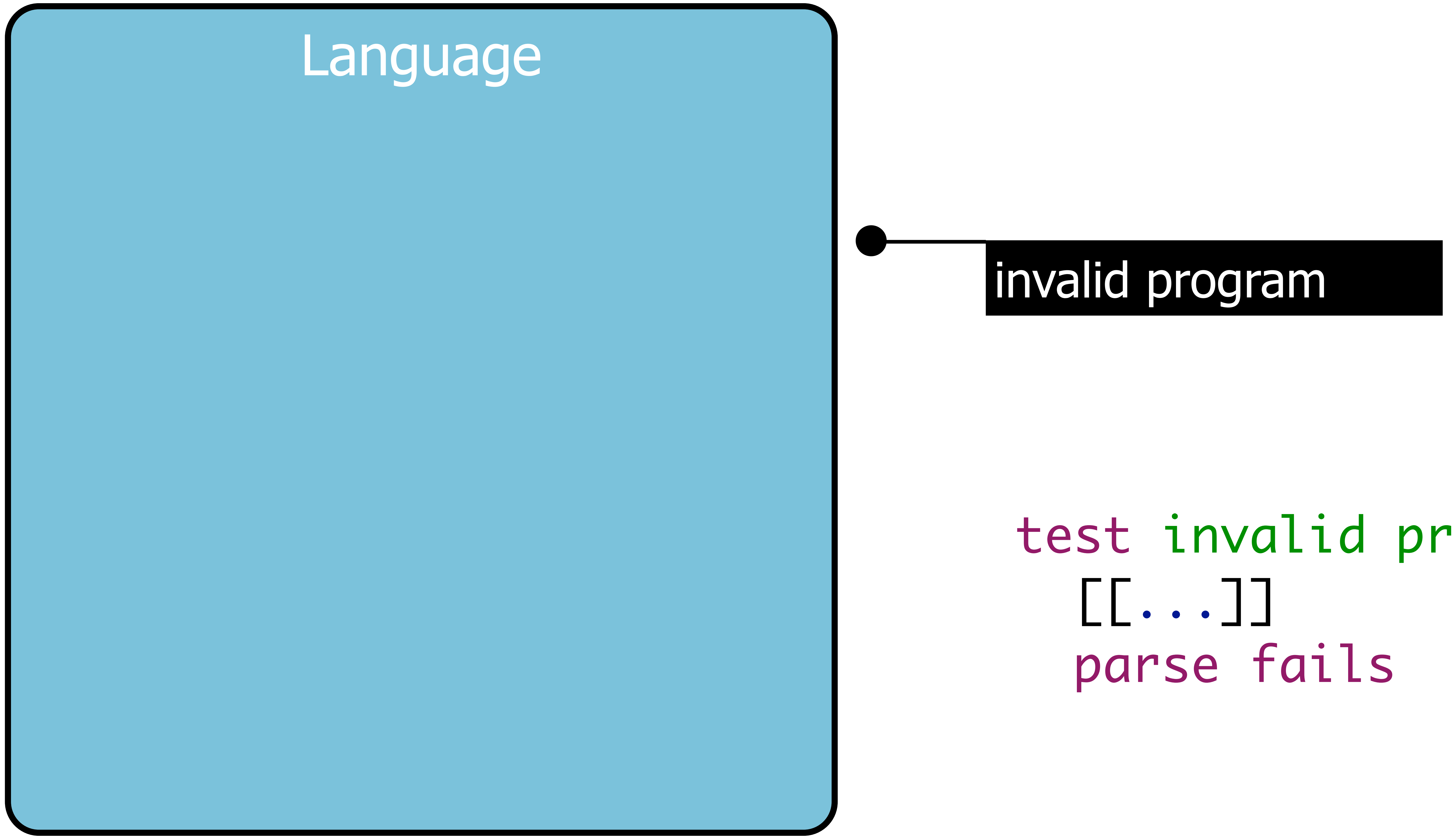

```
module success-failure

language Tiger

test this test succeeds [[
  1 + 3 * 4
]] parse succeeds

test this test fails [[
  1 + 3 * 4
]] parse fails

test this test fails [[
  1 + 3 *
]] parse succeeds

test this test succeeds [[
  1 + 3 *
]] parse fails
```

Test Cases: Valid




```
test valid program
  [[...]]
  parse succeeds
```

Test Cases: Invalid

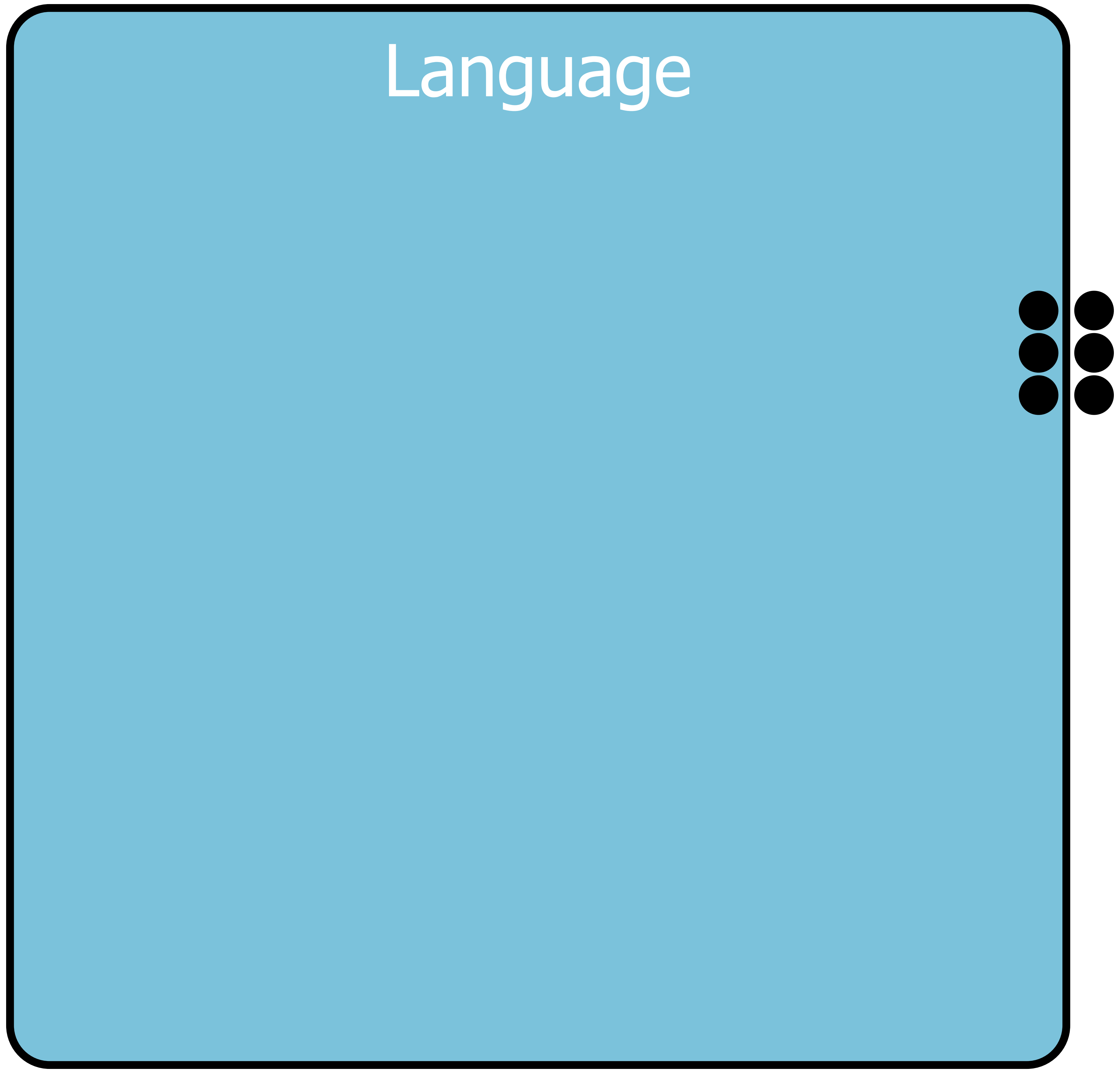



```
test invalid program
  [[...]]
  parse fails
```

Test Cases

```
module syntax/identifiers

language Tiger start symbol Id

test single lower case [[x]] parse succeeds
test single upper case [[X]] parse succeeds
test single digit      [[1]] parse fails

test single lc digit   [[x1]] parse succeeds
test single digit lc   [[1x]] parse fails
test single uc digit   [[X1]] parse succeeds
test single digit uc   [[1X]] parse fails
test double digit      [[11]] parse fails
```

Test Corner Cases



Testing Structure

The fragment did not parse to the expected ATerm. Parse result was: Mod(Plus(Plus(Int("21"),Int("14")),Int("7"))) Expected result was: Mod(Plus(Int(21), Times(Int(14), Int(7))))

```
module structure

language Tiger

test add times [[
  21 + 14 + 7
]] parse to Mod(Plus(Int(“21"),Times(Int("14"),Int("7"))))

test times add [[
  3 * 7 + 21
]] parse to Mod(Plus(Times(Int(“3"),Int("7")),Int("21")))

test if [[
  if x then 3 else 4
]] parse to Mod(If(Var("x"),Int("3"),Int("4")))
```

Testing Ambiguity

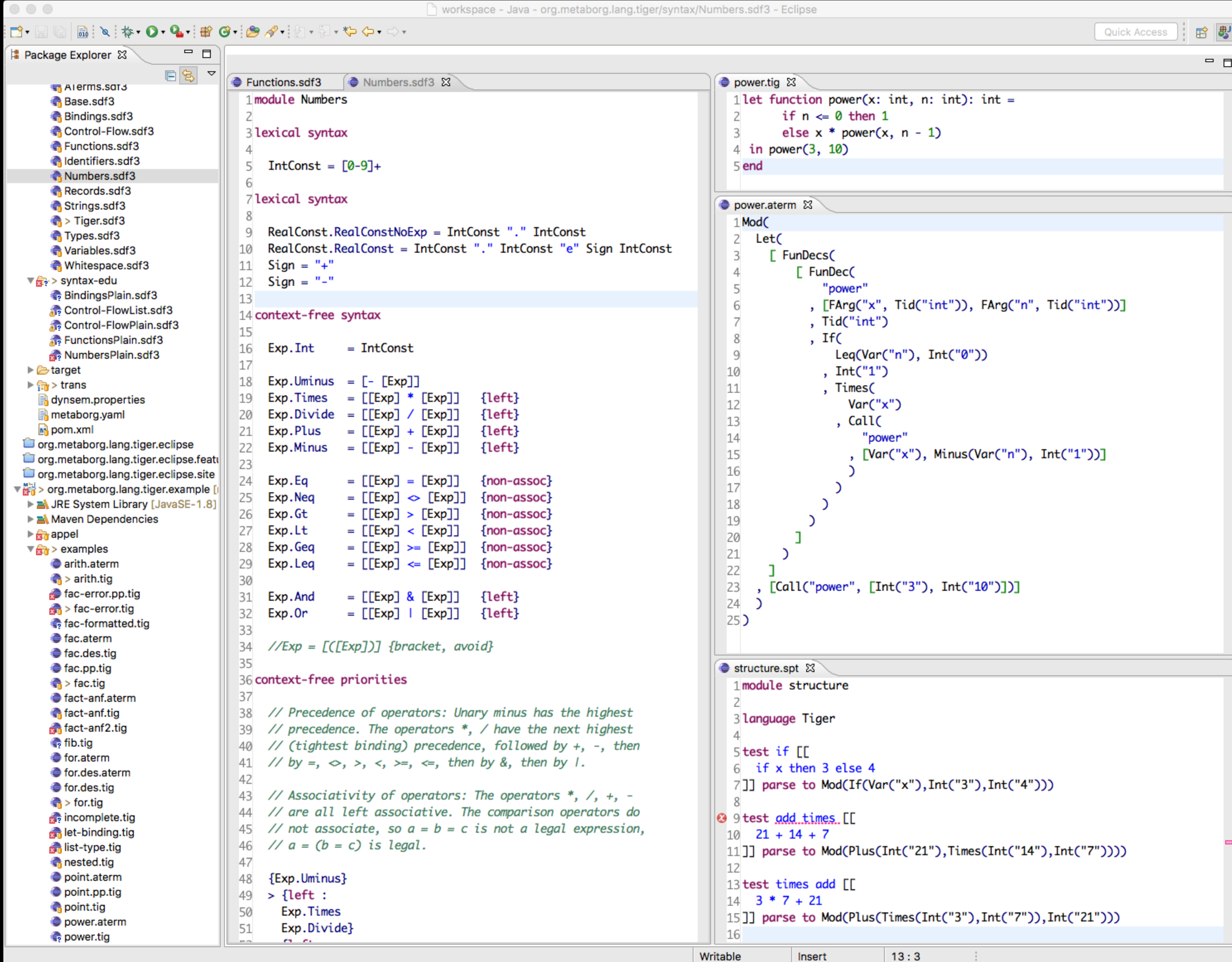
```
module precedence

language Tiger start symbol Exp

test parentheses
  [[(42)]] parse to [[42]]

test left-associative addition
  [[21 + 14 + 7]] parse to [[(21 + 14) + 7]]

test precedence multiplication
  [[3 * 7 + 21]] parse to [[(3 * 7) + 21]]
```

Note : this does not actually work in Tiger, since () is a sequencing construct; but works fine in Mini-Java

Testing Ambiguity

```
module precedence

language Tiger start symbol Exp

test plus/times priority [[
  x + 4 * 5
]] parse to Plus(Var("x"), Times(Int("4"), Int("5")))

test plus/times sequence [[
  (x + 4) * 5
]] parse to Times(
  Seq([Plus(Var("x"), Int("4"))])
, Int("5")
)
```

Syntax Engineering in Spoofax


workspace - Java - org.metaborg.lang.tiger/syntax/Numbers.sdf3 - Eclipse
Quick Access
Package Explorer
Base.sdf3
Bindings.sdf3
Control-Flow.sdf3
Functions.sdf3
Identifiers.sdf3
Numbers.sdf3
Records.sdf3
Strings.sdf3
> Tiger.sdf3
Types.sdf3
Variables.sdf3
Whitespace.sdf3
> syntax-edu
BindingsPlain.sdf3
Control-FlowList.sdf3
Control-FlowPlain.sdf3
FunctionsPlain.sdf3
NumbersPlain.sdf3
target
> trans
dynsem.properties
metaborg.yaml
pom.xml
org.metaborg.lang.tiger.eclipse
org.metaborg.lang.tiger.eclipse.site
> org.metaborg.lang.tiger.example
JRE System Library [JavaSE-1.8]
Maven Dependencies
appel
> examples
arith.aterm
> arith.tig
fac-error.pp.tig
> fac-error.tig
fac-formatted.tig
fac.aterm
fac.des.tig
fac.pp.tig
> fac.tig
fact-anf.aterm
fact-anf.tig
fact-anf2.tig
fib.tig
for.aterm
for.des.aterm
for.des.tig
> for.tig
incomplete.tig
let-binding.tig
list-type.tig
nested.tig
point.aterm
point.pp.tig
point.tig
power.aterm
power.tig
Functions.sdf3
Numbers.sdf3
module Numbers
lexical syntax
IntConst = [0-9]+
lexical syntax
RealConst.RealConstNoExp = IntConst "." IntConst
RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
Sign = "+"
Sign = "-"
context-free syntax
Exp.Int = IntConst
Exp.Uminus = [- [Exp]]
Exp.Times = [[Exp] * [Exp]] {left}
Exp.Divide = [[Exp] / [Exp]] {left}
Exp.Plus = [[Exp] + [Exp]] {left}
Exp.Minus = [[Exp] - [Exp]] {left}
Exp.Eq = [[Exp] = [Exp]] {non-assoc}
Exp.Neq = [[Exp] <> [Exp]] {non-assoc}
Exp.Gt = [[Exp] > [Exp]] {non-assoc}
Exp.Lt = [[Exp] < [Exp]] {non-assoc}
Exp.Geq = [[Exp] >= [Exp]] {non-assoc}
Exp.Leq = [[Exp] <= [Exp]] {non-assoc}
Exp.And = [[Exp] & [Exp]] {left}
Exp.Or = [[Exp] | [Exp]] {left}
//Exp = [([Exp])] {bracket, avoid}
context-free priorities
// Precedence of operators: Unary minus has the highest
// precedence. The operators *, / have the next highest
// (tightest binding) precedence, followed by +, -, then
// by =, <>, >, <, >=, <=, then by &, then by |.
// Associativity of operators: The operators *, /, +, -
// are all left associative. The comparison operators do
// not associate, so a = b = c is not a legal expression,
// a = (b = c) is legal.
{Exp.Uminus}
> {left :
Exp.Times
Exp.Divide}
power.tig
let function power(x: int, n: int): int =
if n <= 0 then 1
else x * power(x, n - 1)
in power(3, 10)
end
power.aterm
Mod(
Let(
[FunDecs(
[FunDec(
"power"
, [FArg("x", Tid("int")), FArg("n", Tid("int"))]
, Tid("int")
, If(
Leq(Var("n"), Int("0"))
, Int("1")
, Times(
Var("x")
, Call(
"power"
, [Var("x"), Minus(Var("n"), Int("1"))]
)
)
)
)
]
)
]
, [Call("power", [Int("3"), Int("10")])]
)
)
structure.spt
module structure
language Tiger
test if [[
if x then 3 else 4
]] parse to Mod(If(Var("x"),Int("3"),Int("4")))
test add times [[
21 + 14 + 7
]] parse to Mod(Plus(Int("21"),Times(Int("14"),Int("7"))))
test times add [[
3 * 7 + 21
]] parse to Mod(Plus(Times(Int("3"),Int("7")),Int("21")))
Writable
Insert
13 : 3

Syntax Engineering in Spoofax

Developing syntax definition

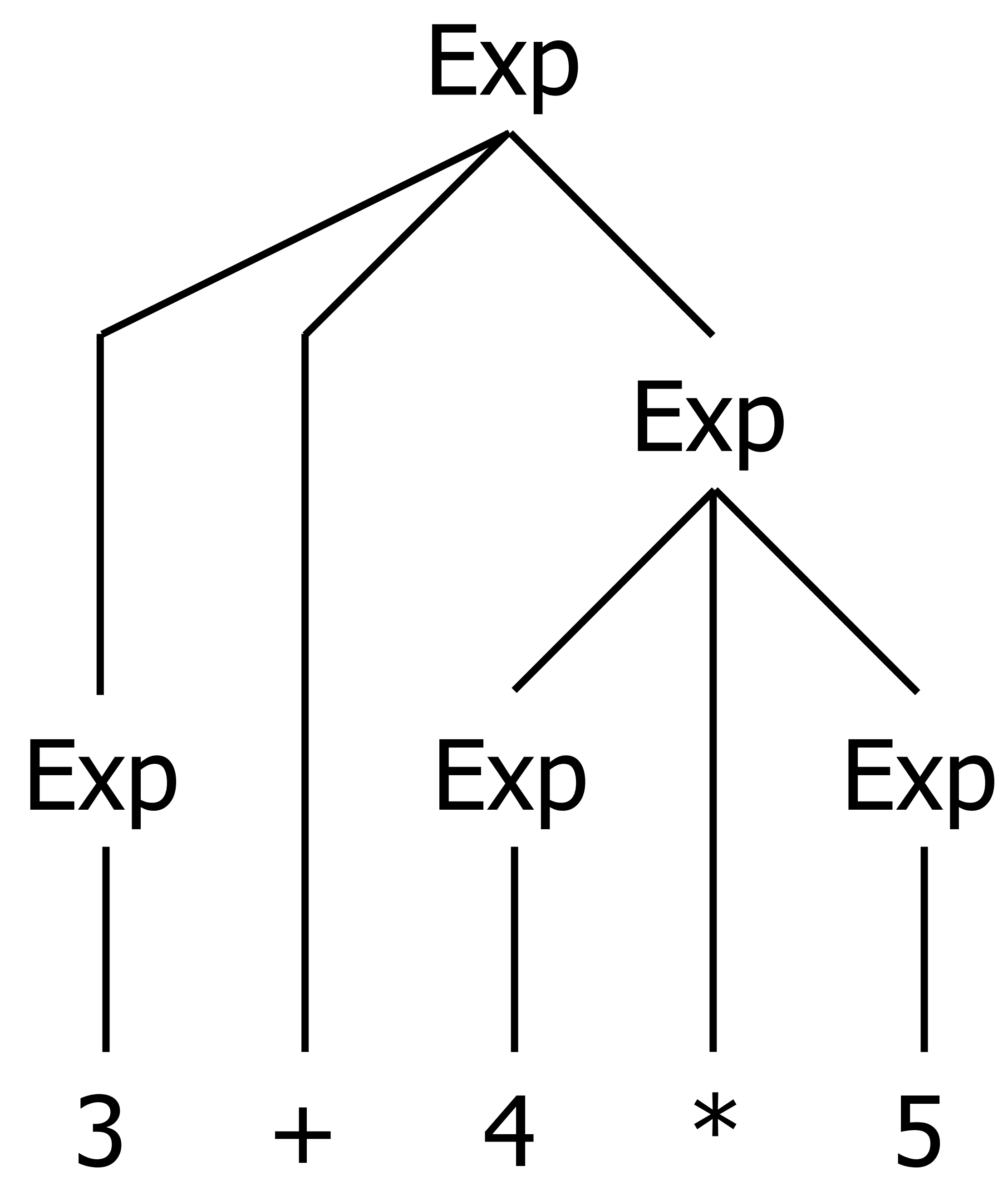
- Define syntax of language in multiple modules
- Syntax checking, colouring
- Checking for undefined non-terminals

Testing syntax definition

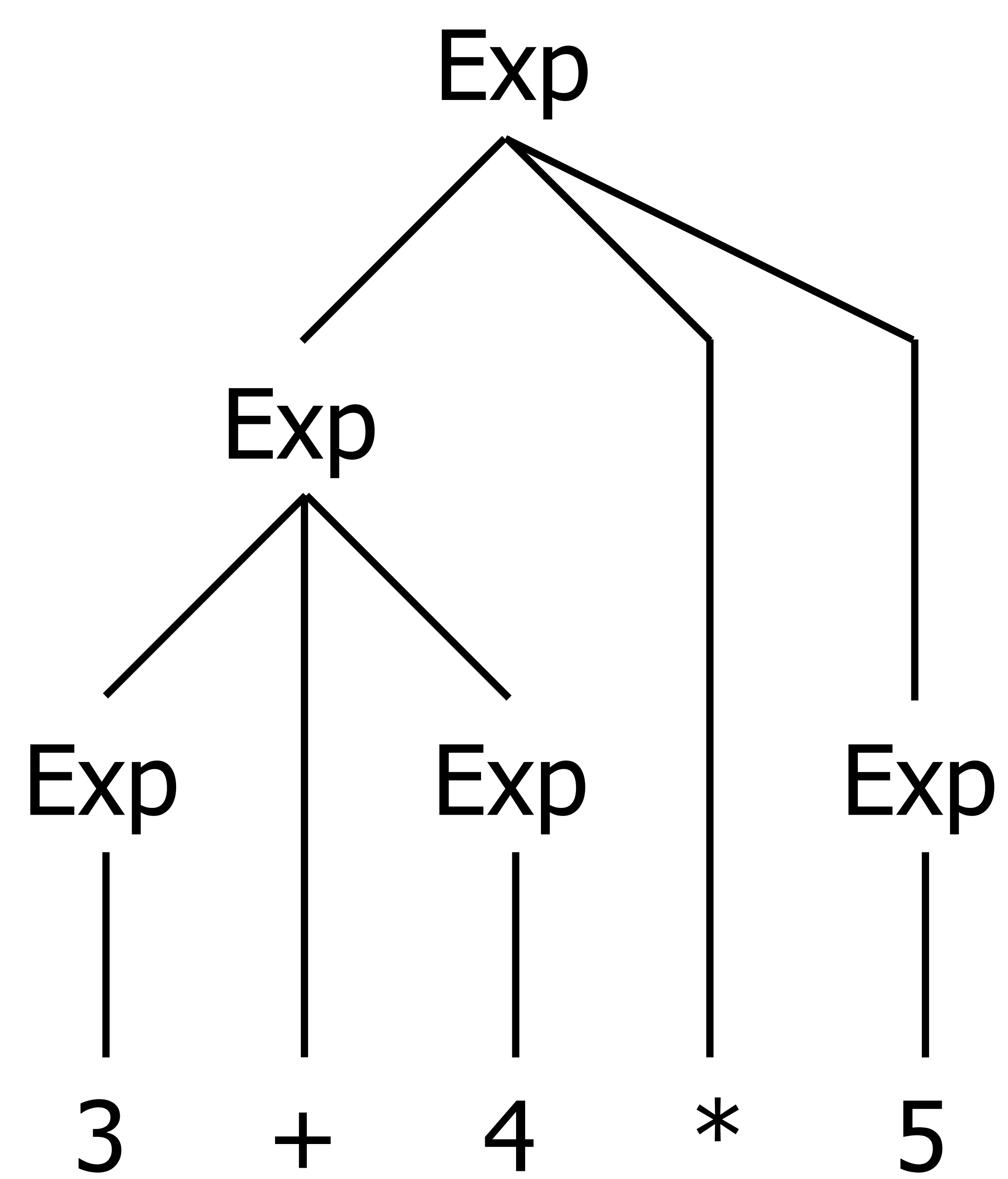
- Write example programs in editor for language under def
- Inspect abstract syntax terms
  - Spoofax > Syntax > Show Parsed AST
- Write SPT test for success and failure cases
  - Updated after build of syntax definition

Ambiguity & Disambiguation

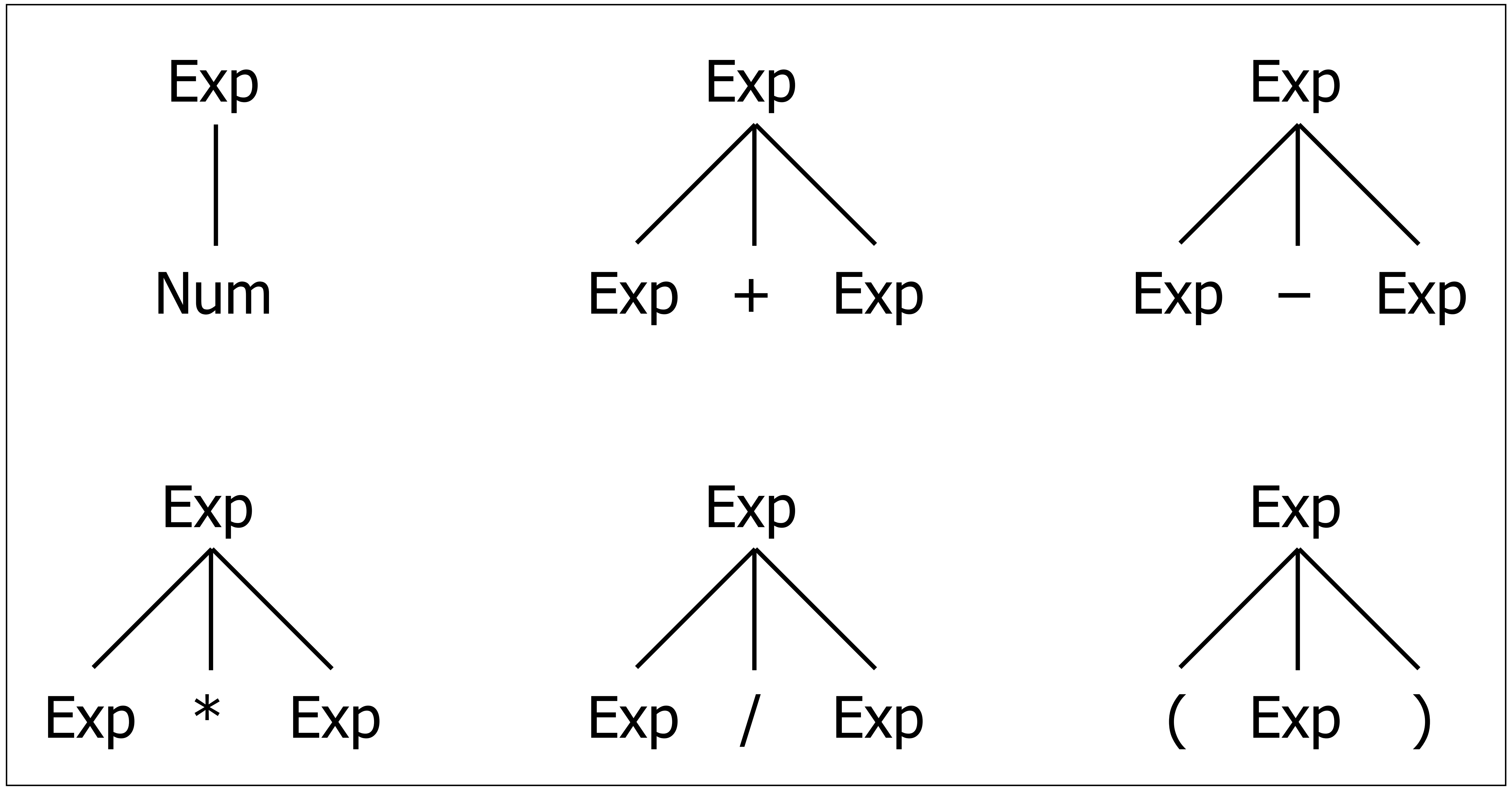
Ambiguity



Exp
Exp
Exp Exp Exp
3 + 4 * 5



Exp
Exp
Exp Exp Exp
3 + 4 * 5



Exp
Num
Exp
Exp + Exp
Exp
Exp – Exp
Exp
Exp * Exp
Exp
Exp / Exp
Exp
(Exp)

Ambiguity

Syntax trees

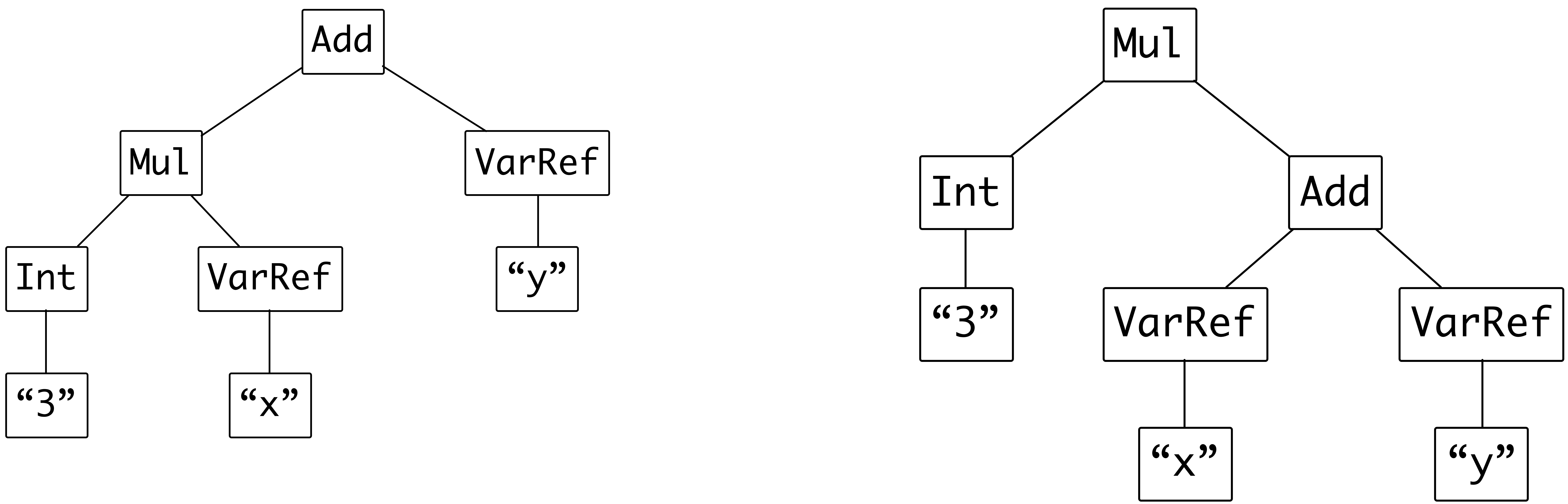
- different trees for same sentence

Derivations

- *different* leftmost derivations for same sentence
- *different* rightmost derivations for same sentence
- NOT just different derivations for same sentence

Ambiguity

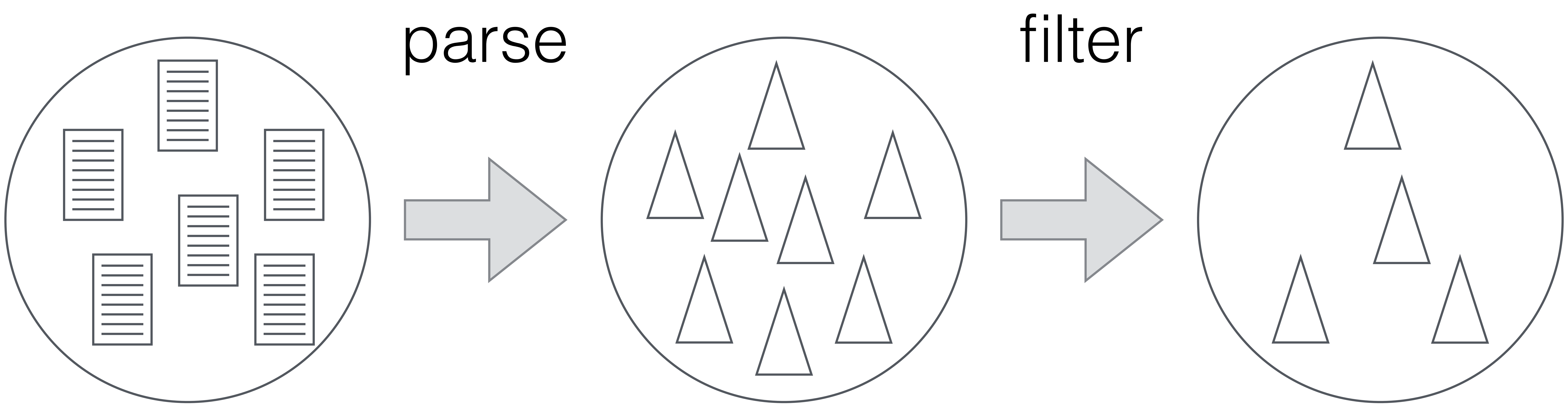
```
3 * x + y
```




```
t1 != t2  /\ format(t1) = format(t2)
```

Debugging Ambiguities

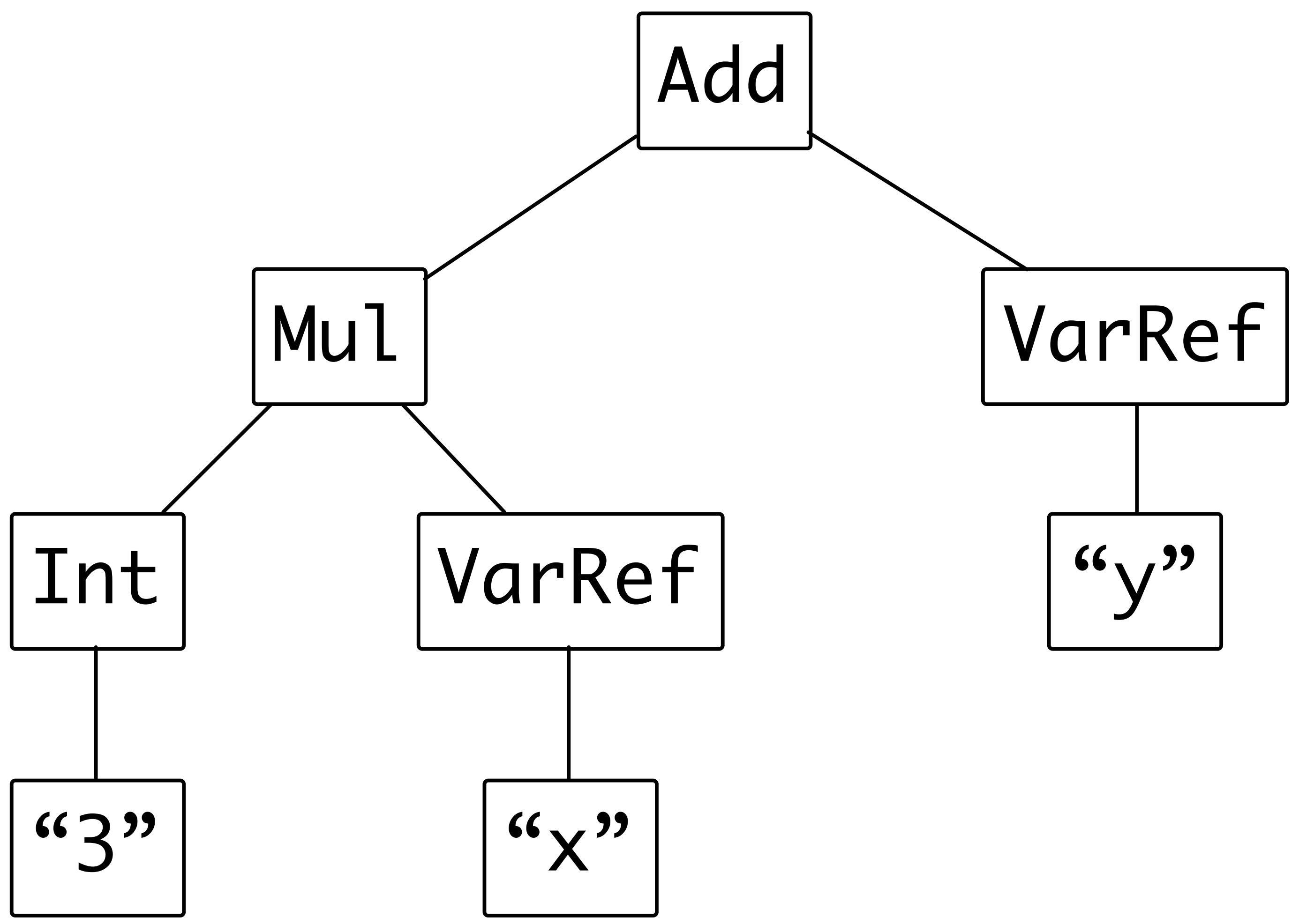
Declarative Disambiguation

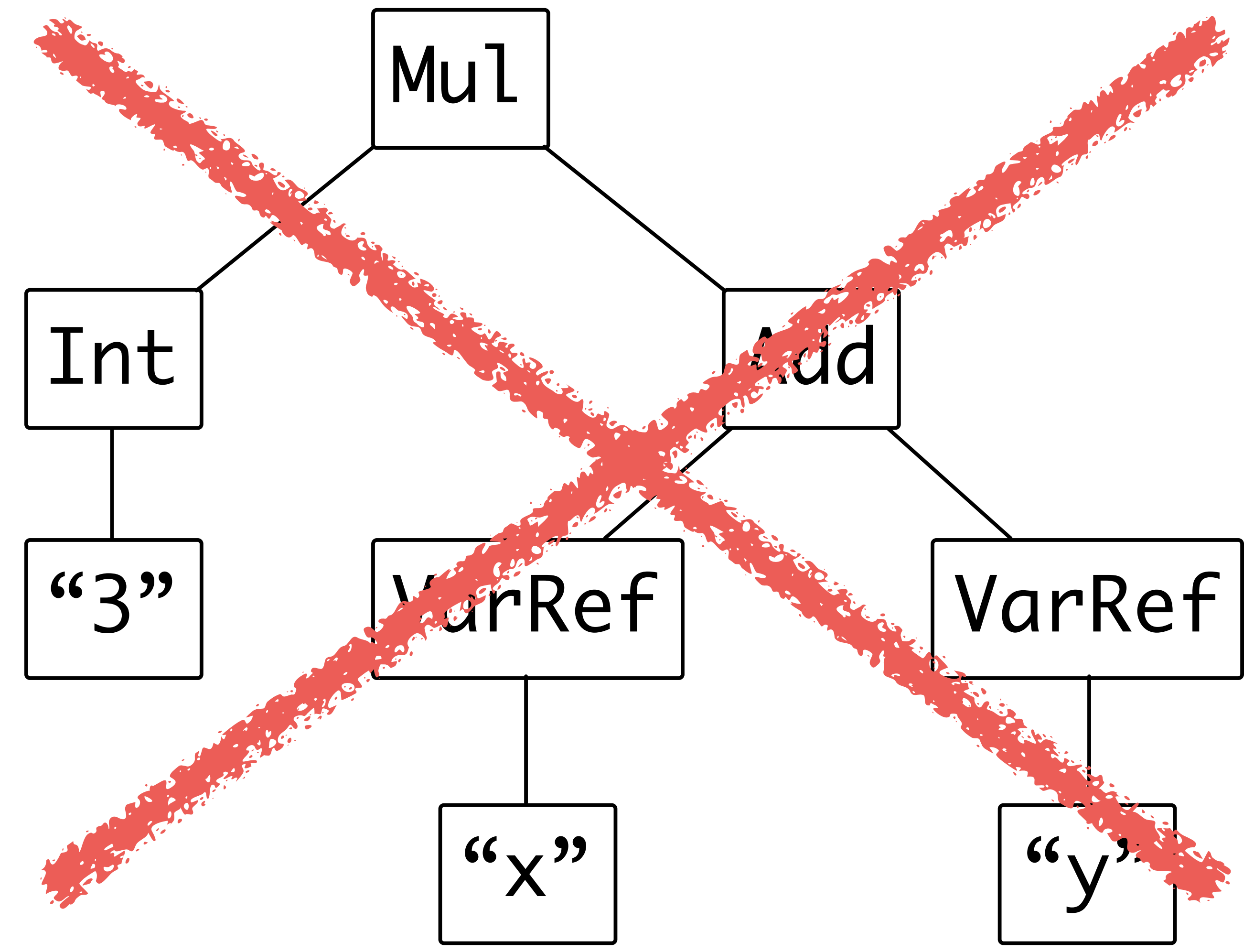


Disambiguation Filters [Klint & Visser; 1994], [Van den Brand, Scheerder, Vinju, Visser; CC 2002]

Priority and Associativity

```
3 * x + y
```






```
context-free syntax
  Expr.Int = INT
  Expr.Add = <<Expr> + <Expr>> {left}
  Expr.Mul = <<Expr> * <Expr>> {left}
context-free priorities
  Expr.Mul > Expr.Add
```

Recent improvement: safe disambiguation of operator precedence [Afroozeh et al. SLE13, Onward15]

Tiger Disambiguation

```
context-free syntax

  Exp.Add = Exp "+" Exp
  Exp.Sub = Exp "-" Exp
  Exp.Mul = Exp "*" Exp
  Exp.Div = Exp "/" Exp

  Exp.Eq  = Exp "="  Exp
  Exp.Neq = Exp "<>" Exp
  Exp.Gt  = Exp ">"  Exp
  Exp.Lt  = Exp "<"  Exp
  Exp.Gte = Exp ">=" Exp
  Exp.Lte = Exp "<=" Exp

  Exp.And = Exp "&"  Exp
  Exp.Or  = Exp "|"  Exp
```

Tiger Disambiguation

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Sub = Exp "-" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Div = Exp "/" Exp {left}

  Exp.Eq  = Exp "="  Exp {non-assoc}
  Exp.Neq = Exp "<>" Exp {non-assoc}
  Exp.Gt  = Exp ">"  Exp {non-assoc}
  Exp.Lt  = Exp "<"  Exp {non-assoc}
  Exp.Gte = Exp ">=" Exp {non-assoc}
  Exp.Lte = Exp "<=" Exp {non-assoc}

  Exp.And = Exp "&"  Exp {left}
  Exp.Or  = Exp "|"  Exp {left}
```

```
context-free priorities
  { left:
      Exp.Mul
      Exp.Div
   } > { left:
      Exp.Add
      Exp.Sub
   } > { non-assoc:
      Exp.Eq
      Exp.Neq
      Exp.Gt
      Exp.Lt
      Exp.Gte
      Exp.Lte
   } > Exp.And
     > Exp.Or
```

Except where otherwise noted, this work is licensed under

Attribution

slide	title	author	license
21-22, 38-40	Noam Chomsky	Fellowsisters	CC BY-NC-SA 2.0