

# Lecture 2: Declarative Syntax Definition

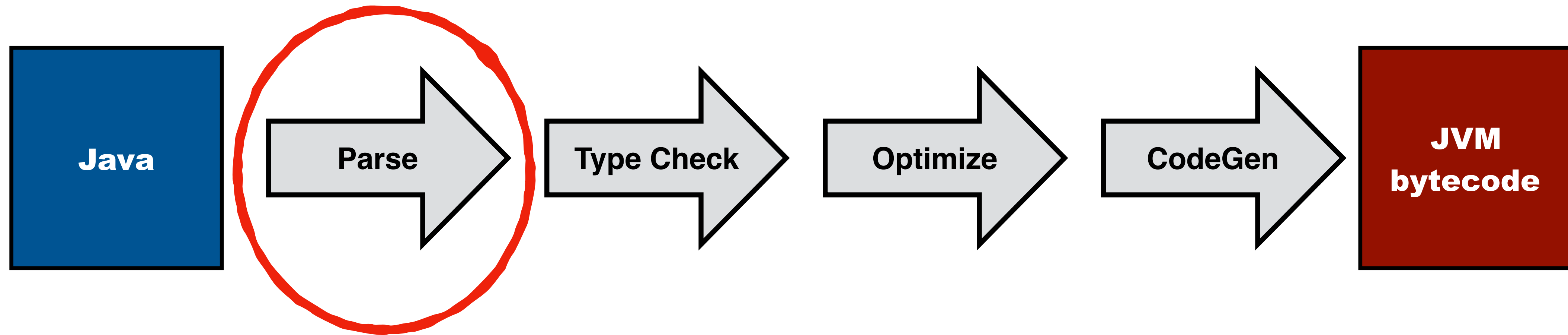
**CS4200 Compiler Construction**

**Eelco Visser**

**TU Delft**

**September 2018**

# This Lecture



Specification of syntax definition from which we can derive parsers

# Reading Material

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

$$X_1 \dots X_n \rightarrow N \{ \text{cons}(\text{“C”}) \}$$

instead of

$$N.C = X_1 \dots X_n$$

<https://doi.org/10.1145/1932682.1869535>

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

# 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 en-

gineers 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. Copyright © 2010 ACM 978-1-4503-0236-4/10/10...\$10.00  
1, 2010, Reno/Tahoe, Nevada, USA.  
1-4503-0236-4/10/10...\$10.00



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

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

<https://doi.org/10.1145/2076021.2048080>

# 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

## 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-

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



The SDF3 syntax definition formalism is documented at the [metaborg.org](http://metaborg.org) website.

 <b>Spoofax</b>	
latest	
<input type="text" value="Search docs"/>	
The Spoofax Language Workbench	
Examples	
Publications	
<b>TUTORIALS</b>	
Installing Spoofax	
Creating a Language Project	
Using the API	
Getting Support	
<b>REFERENCE MANUAL</b>	
Language Definition with Spoofax	
Abstract Syntax with ATerms	
<b>Syntax Definition with SDF3</b>	
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

- 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

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

# Syntax

# What is Syntax?

In [linguistics](#), **syntax** ([/'sɪntæks/\[1\]\[2\]](#)) 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".



# 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.<sup>[1]</sup>

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](#).

# That Govern the Structure

## 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?**

# The Structure of Programs



# What do we call the elements of programs?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

# What kind of program element is this?

```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main() {
    int i;
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
    return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n) {
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Program  
Compilation Unit

# What kind of program element is this?

```
#include <stdio.h>
```

Preprocessor Directive

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```



# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Function Declaration

Function Prototype

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {  
    int i;  
    for (i = 0; i < 10; ++i)  
        printf("%d %d %d\n", i, power(2, i), power(-3, i));  
    return 0;  
}
```

```
/* power: raise base to n-th power; n >= 0 */  
int power(int base, int n) {  
    int i, p;  
    p = 1;  
    for (i = 1; i <= n; ++i)  
        p = p * base;  
    return p;  
}
```

Comment

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {  
    int i;  
    for (i = 0; i < 10; ++i)  
        printf("%d %d %d\n", i, power(2, i), power(-3, i));  
    return 0;  
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {  
    int i, p;  
    p = 1;  
    for (i = 1; i <= n; ++i)  
        p = p * base;  
    return p;  
}
```

Function Definition



# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Variable Declaration

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Statement  
For Loop

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Statement  
Function Call



# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Expression

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Formal Function Parameter

# What kind of program element is this?

```
#include <stdio.h>
```

```
int power(int m, int n);
```

```
/* test power function */
```

```
main() {
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */
```

```
int power(int base, int n) {
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i = 1; i <= n; ++i)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Type

# Syntactic Categories

Preprocessor Directive      For Loop

Function Declaration      Compilation Unit

Function Prototype

Statement      Program      Function Call

Variable Declaration

Function Definition      Type      Formal Function  
Parameter

Expression

**Programs consist of different *kinds* of elements**

# Hierarchy of Syntactic Categories

Program

Compilation Unit

Preprocessor  
Directive

Function  
Definition

Function Declaration  
Function Prototype

Variable  
Declaration

Statement

Type

Expression

For Loop

Formal Function  
Parameter

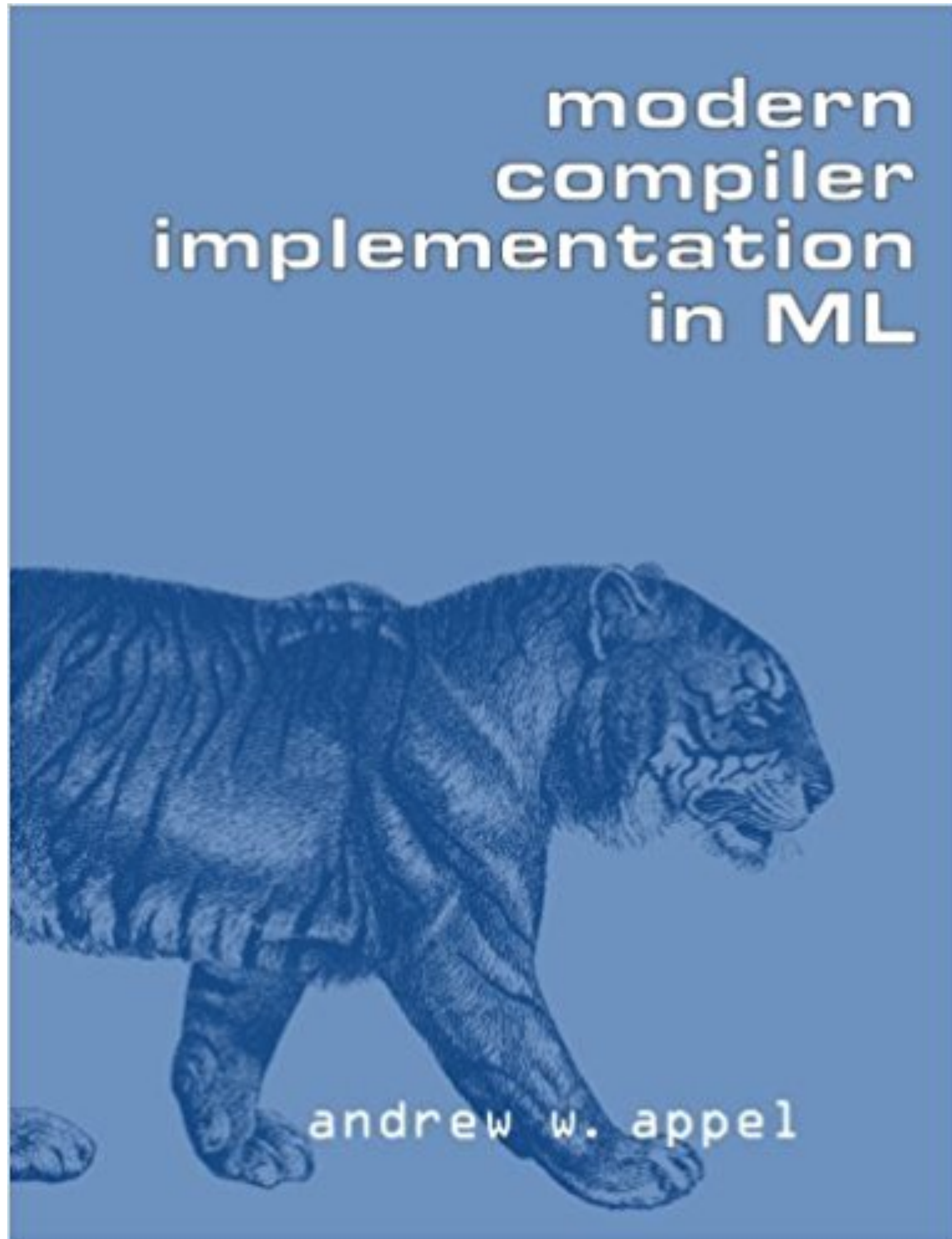
Function Call

**Some kinds of constructs are *contained* in others**



# The Tiger Language

Example language used in lectures



Documentation

<https://www.lrde.epita.fr/~tiger/tiger.html>

Spoofax project

<https://github.com/MetaBorgCube/metaborg-tiger>

```

let
  var N := 8
  type intArray = array of int
  var row := intArray[N] of 0
  var col := intArray[N] of 0
  var diag1 := intArray[N + N - 1] of 0
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
      print("\n")
    );
    print("\n")
  )
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 & diag2[r + 7 - c] = 0 then (
          row[r] := 1;
          diag1[r + c] := 1;
          diag2[r + 7 - c] := 1;
          col[c] := r;
          try(c + 1);
          row[r] := 0;
          diag1[r + c] := 0;
          diag2[r + 7 - c] := 0))
      end
    end
  )
in
  try(0)
end

```

A Tiger program that solves  
the n-queens problem

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end

```

A mutilated n-queens Tiger  
program with 'redundant'  
elements removed

What are the syntactic  
categories of Tiger?

What are the language  
constructs of Tiger called?

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
      print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end

```

Program  
Expression  
Let Binding

# Variable Declaration

```
let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
      print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end
```



## Type Declaration

```
let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end
```

## Type Expression

```
let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end
```

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end

```

## Expression

### Array Initialization

## Function Definition

```
let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end
```

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
    print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))

in
  try(0)
end

```

Function Name



```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
    function try(c : int) = (
      if c = N then
        printboard()
      else
        for r := 0 to N - 1 do
          if row[r] = 0 & diag1[r + c] = 0 then (
            diag2[r + 7 - c] := 1;
            try(c + 1);))
    in
      try(0)
  end

```

Function Body

Expression

For Loop

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
    )
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
    )
in
  try(0)
end

```

Expression

Sequential Composition

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
      print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1;
          try(c + 1);))
  in
    try(0)
end

```

Formal Parameter

```

let
  var N := 8
  type intArray = array of int
  var diag2 := intArray[N + N - 1] of 0
  function printboard() = (
    for i := 0 to N - 1 do (
      for j := 0 to N - 1 do
        print(if col[i] = j then
          " 0"
        else
          " .");
        print("\n"))
  function try(c : int) = (
    if c = N then
      printboard()
    else
      for r := 0 to N - 1 do
        if row[r] = 0 & diag1[r + c] = 0 then (
          diag2[r + 7 - c] := 1
          try(c + 1);))
  in
    try(0)
end

```

Expression  
Assignment

## Functions can be nested

```
let
...
function prettyprint(tree : tree) : string =
  let
    var output := ""
    function write(s : string) =
      output := concat(output, s)
    function show(n : int, t : tree) =
      let
        function indent(s : string) = (
          write("\n");
          for i := 1 to n do
            write(" ") ;
          output := concat(output, s))
        in
          if t = nil then
            indent(".")
          else (
            indent(t.key);
            show(n + 1, t.left);
            show(n + 1, t.right))
          end
        in
          show(0, tree);
          output
      end
    ...
  in
    ...
end
```



# Elements of Programs

## Structure

- Programs have structure

## Categories

- Program elements come in multiple categories
- Elements cannot be arbitrarily interchanged

## Constructs

- Some categories have multiple elements

## Hierarchy

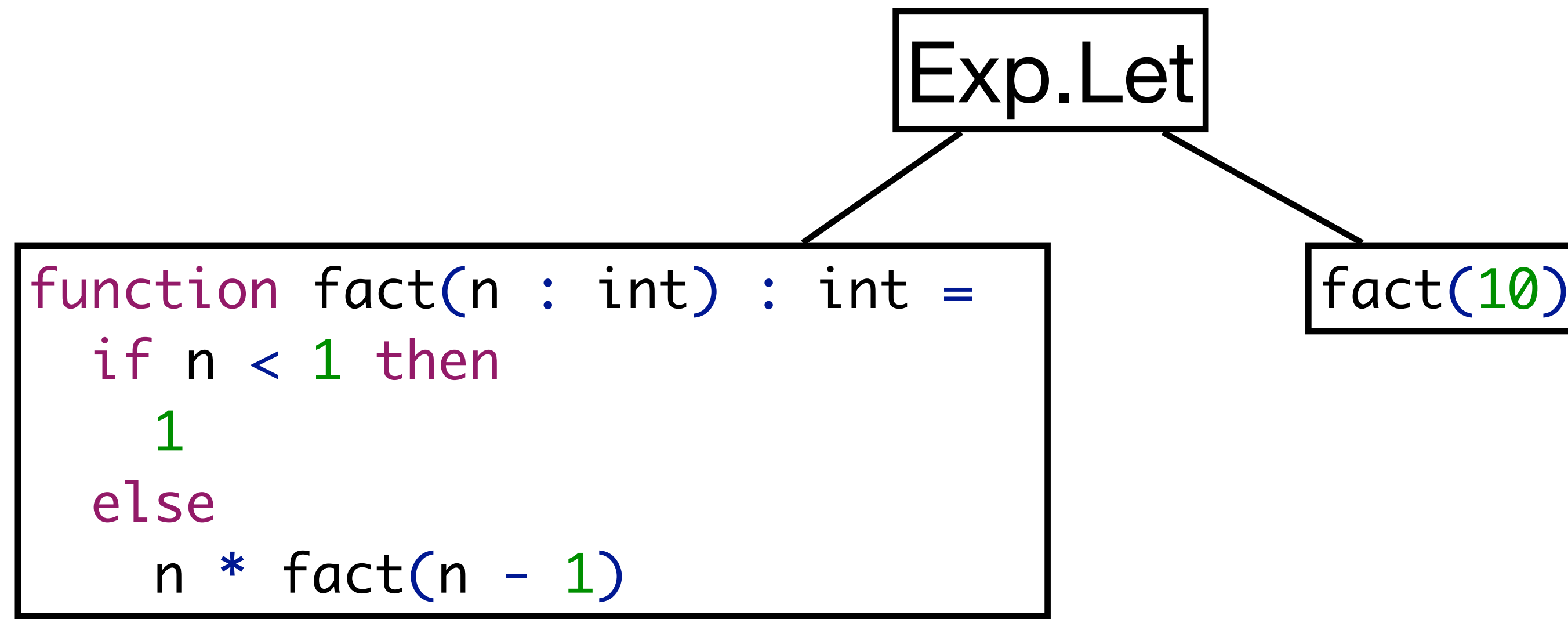
- Categories are organized in a hierarchy

# Decomposing Programs

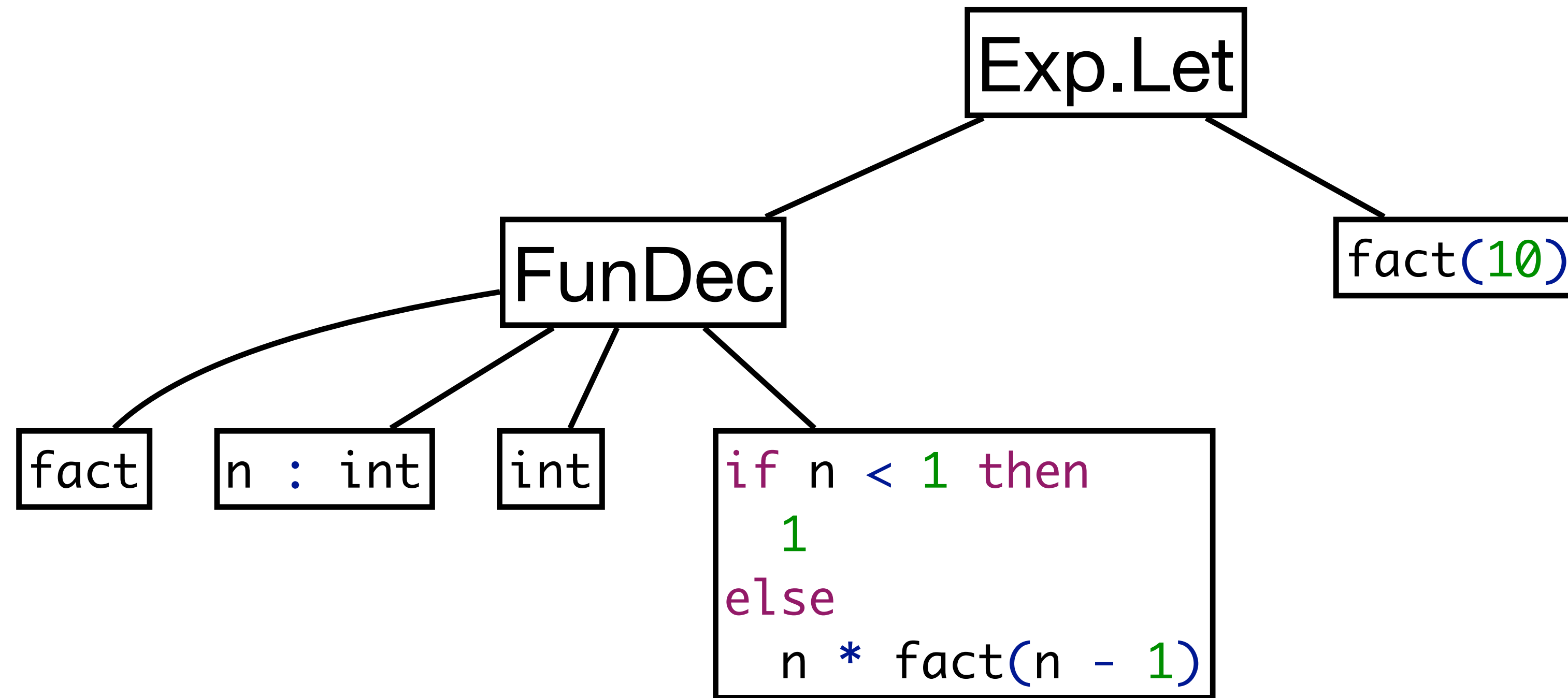
# Decomposing a Program into Elements

```
let function fact(n : int) : int =  
    if n < 1 then  
        1  
    else  
        n * fact(n - 1)  
    in  
        fact(10)  
end
```

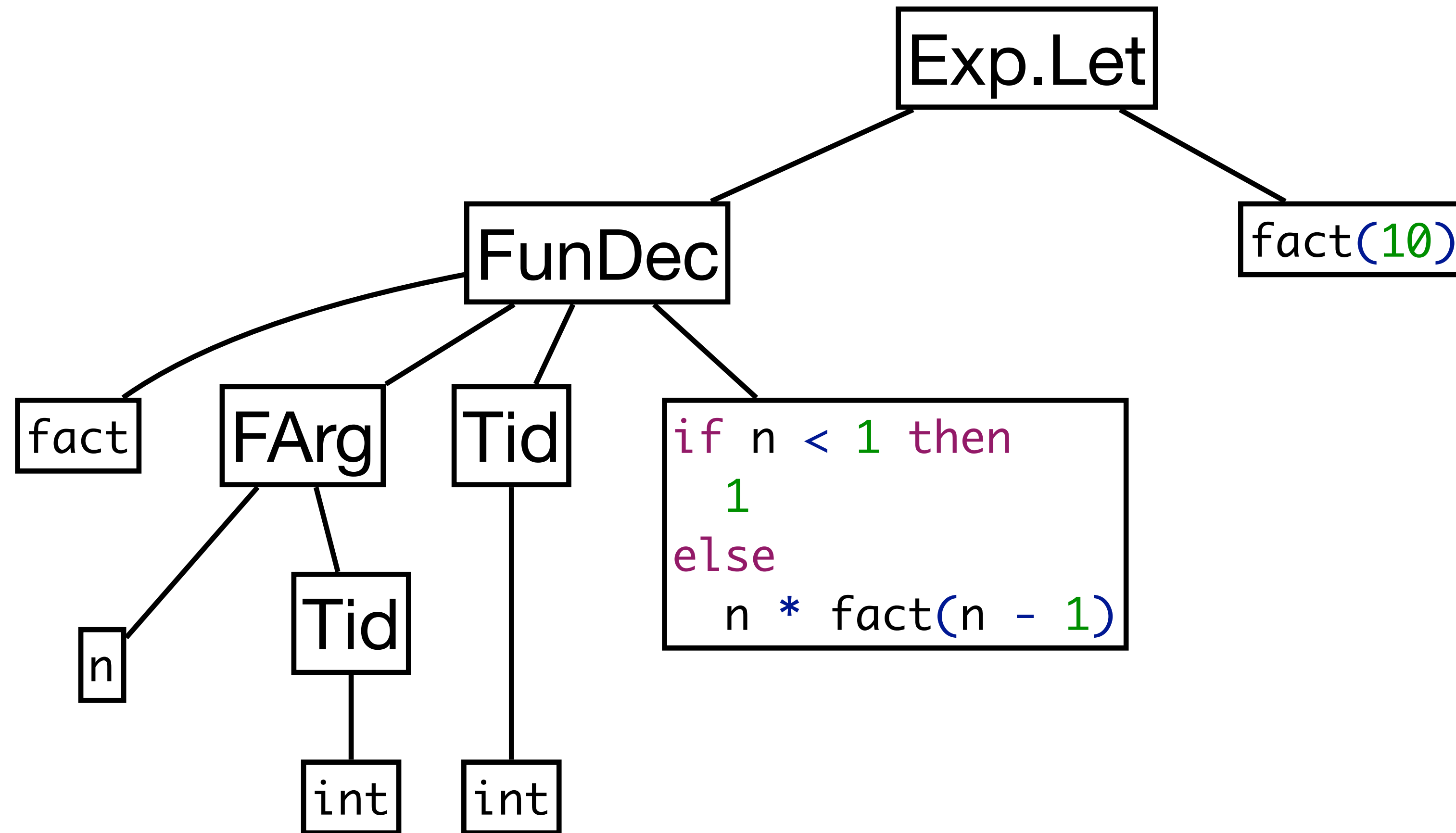
# Decomposing a Program into Elements



# Decomposing a Program into Elements

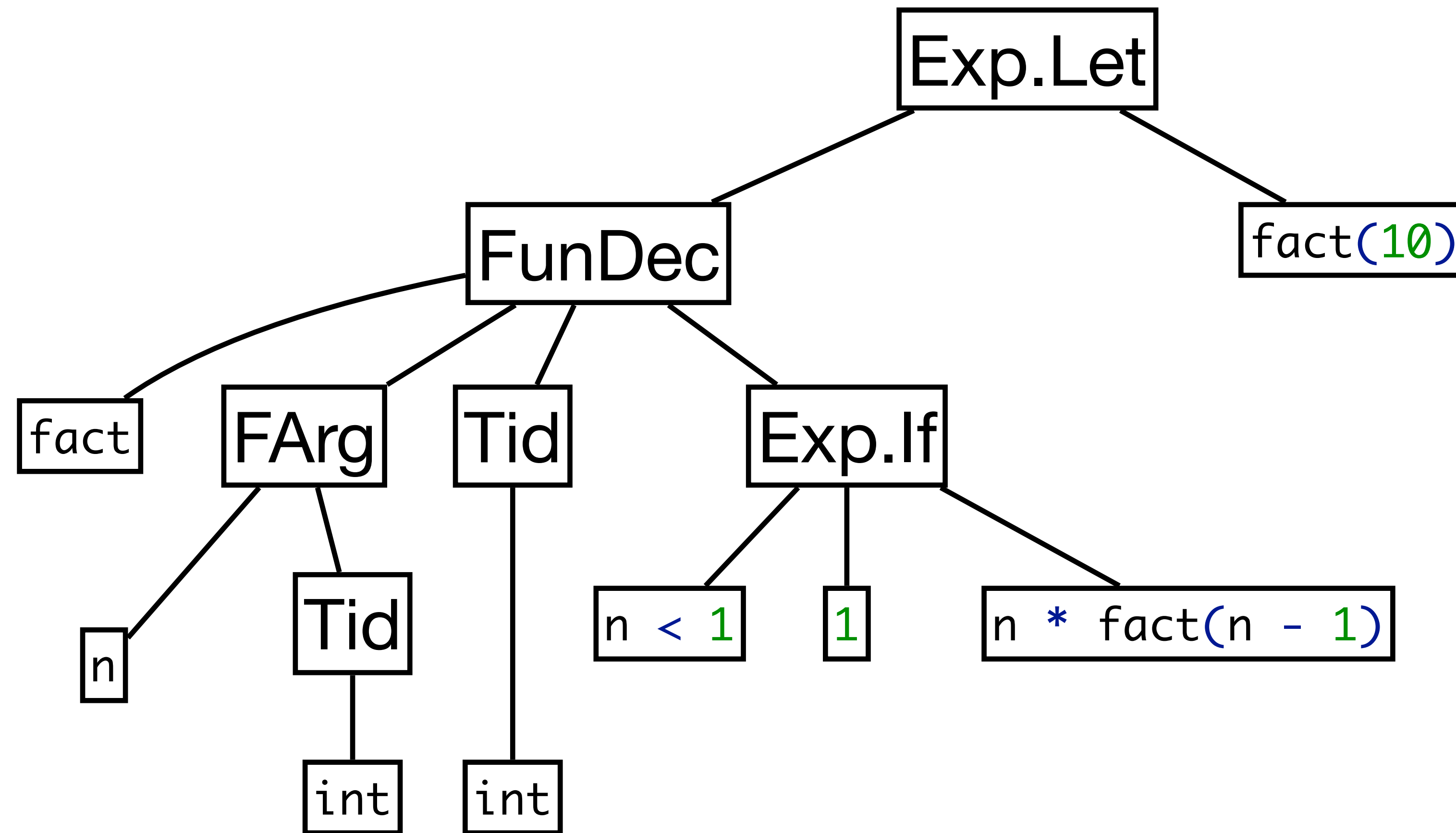


# Decomposing a Program into Elements

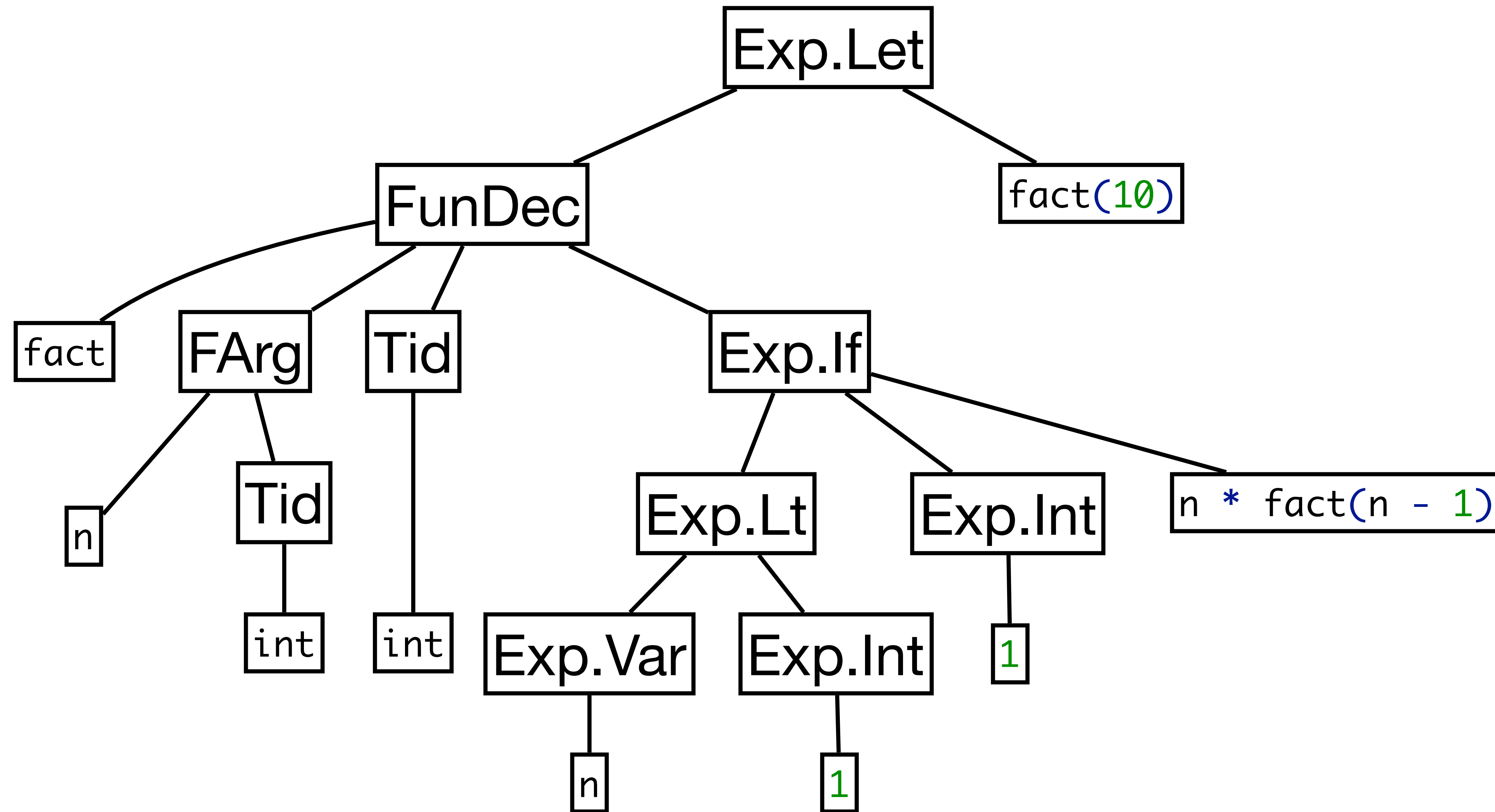




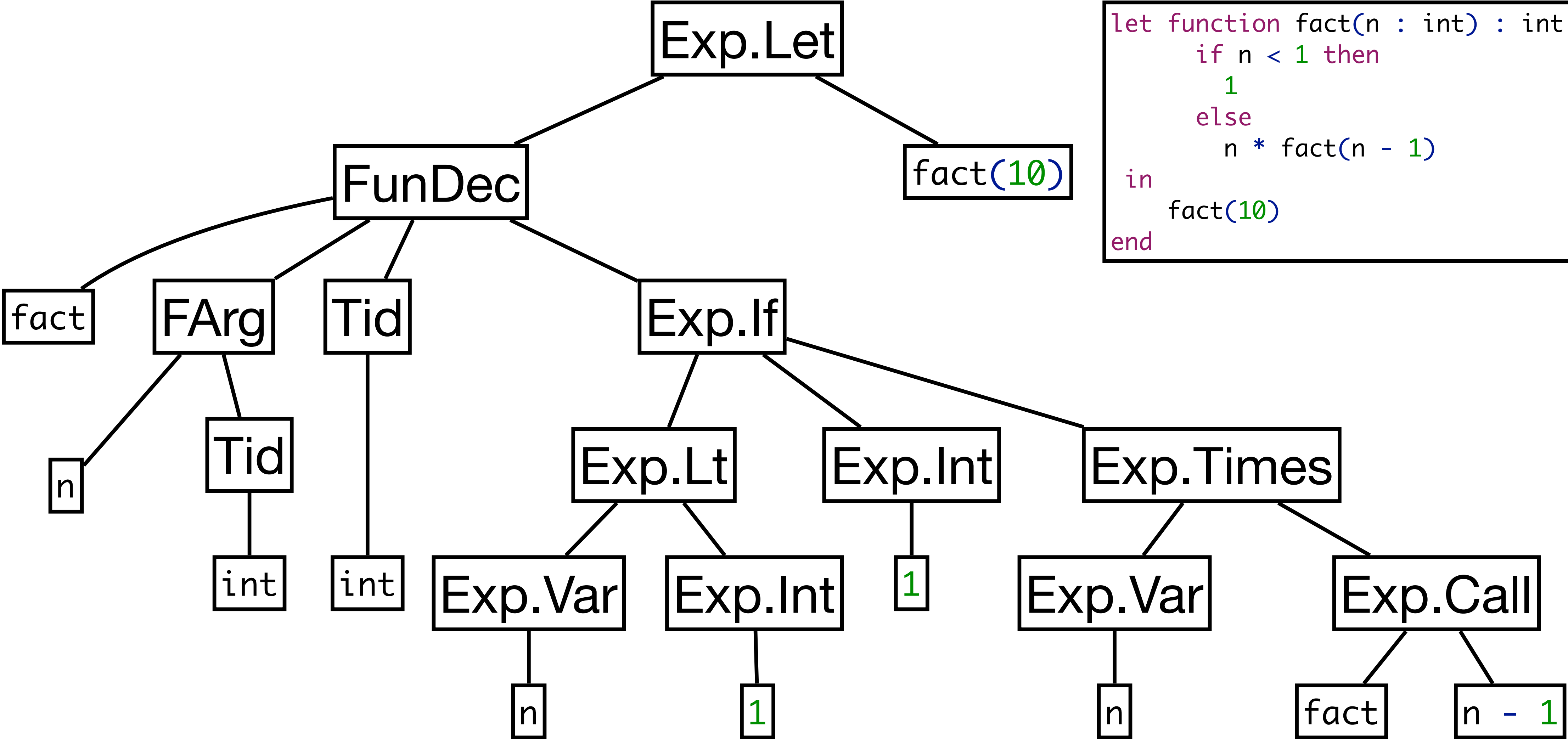
# Decomposing a Program into Elements



# Decomposing a Program into Elements



# Decomposing a Program into Elements



Etc.

# Tree Structure Represented as (First-Order) Term

```
let function fact(n : int) : int =  
  if n < 1 then  
    1  
  else  
    n * fact(n - 1)  
in  
  fact(10)  
end
```

```
Mod(  
  Let(  
    [ FunDecs(  
      [ FunDec(  
        "fact"  
        , [FArg("n", Tid("int"))]  
        , Tid("int")  
        , If(  
          Lt(Var("n"), Int("1"))  
          , Int("1")  
          , Times(  
            Var("n")  
            , Call("fact", [Minus(Var("n"), Int("1"))])  
          )  
        )  
      ]  
    )  
  )  
  , [Call("fact", [Int("10")])]  
)
```

# Decomposing Programs

## Textual representation

- Convenient to read and write (human processing)
- Concrete syntax / notation

## Structural tree/term representation

- Represents the decomposition of a program into elements
- Convenient for machine processing
- Abstract syntax

# Formalizing Program Decomposition

**What are well-formed textual programs?**

**What are well-formed terms/trees?**

**How to decompose programs automatically?**



# **Abstract Syntax: Formalizing Program Structure**

# Algebraic Signatures

```
signature
  sorts S0 S1 S2 ...
  constructors
    C : S1 * S2 * ... -> S0
    ...
```

Sorts: syntactic categories

Constructors: language constructs

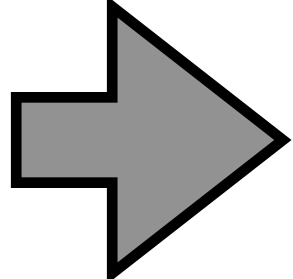
# Well-Formed Terms

The family of well-formed terms  $T(\text{Sig})$  defined by signature  $\text{Sig}$  is inductively defined as follows:

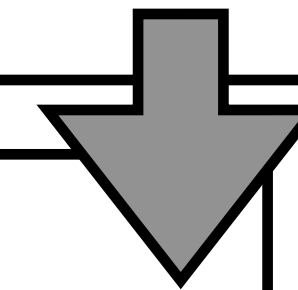
If  $C : S_1 * S_2 * \dots \rightarrow S_0$  is a constructor in  $\text{Sig}$  and  
if  $t_1, t_2, \dots$  are terms in  $T(\text{Sig})(S_1), T(\text{Sig})(S_2), \dots$ ,  
then  $C(t_1, t_2, \dots)$  is a term in  $T(\text{Sig})(S_0)$

# Well-Formed Terms: Example

```
if n < 1 then
  1
else
  n * fact(n - 1)
```

 decompose

```
If(
  Lt(Var("n"), Int("1"))
, Int("1")
, Times(
  Var("n")
  , Call("fact", [Minus(Var("n"), Int("1"))])
)
)
```

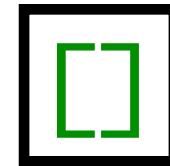


well-formed wrt

```
signature
sorts Exp
constructors
  Int    : IntConst -> Exp
  Var    : ID -> Exp
  Times  : Exp * Exp -> Exp
  Minus  : Exp * Exp -> Exp
  Lt     : Exp * Exp -> Exp
  If     : Exp * Exp * Exp -> Exp
  Call   : ID * List(Exp) -> Exp
```

# Lists of Terms

```
signature
  sorts Exp Var
  constructors
    ...
    Call : ID * List(Exp) -> Exp
```



```
[Minus(Var("n"), Int("1"))]
```

```
[Minus(Var("n"), Int("1")), Lt(Var("n"), Int("1"))]
```

# Well-Formed Terms with Lists

The family of well-formed terms  $T(\text{Sig})$  defined by signature  $\text{Sig}$  is inductively defined as follows:

If  $C : S_1 * S_2 * \dots \rightarrow S_0$  is a constructor in  $\text{Sig}$  and  
if  $t_1, t_2, \dots$  are terms in  $T(\text{Sig})(S_1), T(\text{Sig})(S_2), \dots$ ,  
then  $C(t_1, t_2, \dots)$  is a term in  $T(\text{Sig})(S_0)$

If  $t_1, t_2, \dots$  are terms in  $T(\text{Sig})(S)$ ,  
Then  $[t_1, t_2, \dots]$  is a term in  $T(\text{Sig})(\text{List}(S))$



# Abstract Syntax

## Abstract syntax of a language

- Defined by algebraic signature
- Sorts: syntactic categories
- Constructors: language constructs

## Program structure

- Represented by (first-order) term
- Well-formed with respect to abstract syntax
- (Isomorphic to tree structure)

# **From Abstract Syntax to Concrete Syntax**

# What does Abstract Syntax Abstract from?

```
signature
  sorts Exp Var
  constructors
    Int      : IntConst -> Exp
    Var      : ID -> Exp
    Times    : Exp * Exp -> Exp
    Minus    : Exp * Exp -> Exp
    Lt       : Exp * Exp -> Exp
    If       : Exp * Exp * Exp -> Exp
    Call     : ID * List(Exp) -> Exp
```

Signature does not define ‘notation’

# What is Notation?

signature

sorts Exp Var

constructors

Int : IntConst -> Exp

Var : ID -> Exp

Times : Exp \* Exp -> Exp

Minus : Exp \* Exp -> Exp

Lt : Exp \* Exp -> Exp

If : Exp \* Exp \* Exp -> Exp

Call : ID \* List(Exp) -> Exp

```
if n < 1 then
  1
else
  n * fact(n - 1)
```

```
n
x
e1 * e2
e1 - e2
e1 < e2
if e1 then e2 else e3
f(e1, e2, ...)
```

Notation: literals, keywords, delimiters, punctuation

# How can we couple notation to abstract syntax?

signature

sorts Exp Var

constructors

Int : IntConst -> Exp

Var : ID -> Exp

Times : Exp \* Exp -> Exp

Minus : Exp \* Exp -> Exp

Lt : Exp \* Exp -> Exp

If : Exp \* Exp \* Exp -> Exp

Call : ID \* List(Exp) -> Exp

```
if n < 1 then
```

```
  1
```

```
else
```

```
  n * fact(n - 1)
```

n

x

e1 \* e2

e1 - e2

e1 < e2

if e1 then e2 else e3

f(e1, e2, ...)

Notation: literals, keywords, delimiters, punctuation

# Context-Free Grammars

grammar

non-terminals  $N_0$   $N_1$   $N_2$  ...

terminals  $T_0$   $T_1$   $T_2$  ...

productions

$N_0 = S_1 S_2 \dots$

...

Non-terminals (N): syntactic categories

Terminals (T): words of sentences

Symbols (S): non-terminals and terminals

Productions: rules to create sentences



# Well-Formed Sentences

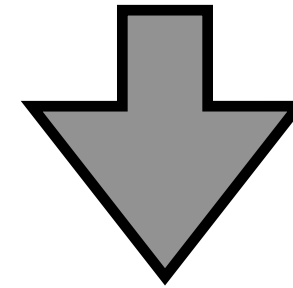
The family of sentences  $L(G)$  defined by context-free grammar  $G$  is inductively defined as follows:

A terminal  $T$  is a sentence in  $L(G)(T)$

If  $N_0 = S_1 S_2 \dots$  is a production in  $G$  and  
if  $w_1, w_2, \dots$  are sentences in  $L(G)(S_1), L(G)(S_2), \dots$ ,  
then  $w_1 w_2 \dots$  is a sentence in  $L(G)(N_0)$

# Well-Formed Sentences

```
if n < 1 then  
    1  
else  
    n * fact(n - 1)
```



grammar

non-terminals Exp Var

productions

Exp = IntConst

Exp = Id

Exp = Exp "\*" Exp

Exp = Exp "-" Exp

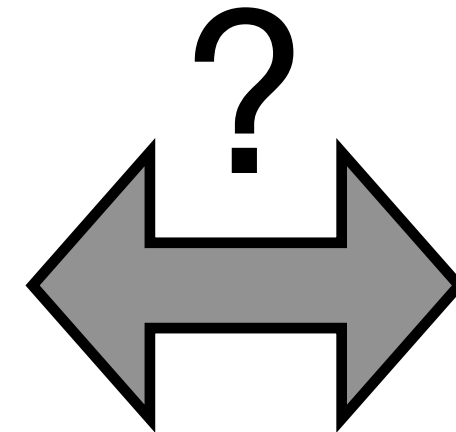
Exp = Exp "<" Exp

Exp = "if" Exp "then" Exp "else" Exp

Exp = Id "(" {Exp ","}\* ")"

# What is the relation between concrete and abstract syntax?

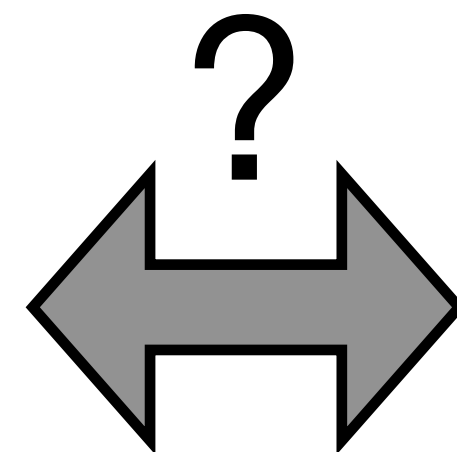
```
if n < 1 then
  1
else
  n * fact(n - 1)
```



```
If(
  Lt(Var("n"), Int("1"))
, Int("1")
, Times(
  Var("n")
  , Call("fact", [Minus(Var("n"), Int("1"))])
)
)
```

grammar  
non-terminals Exp Var  
productions

```
Exp = IntConst
Exp = Id
Exp = Exp "*" Exp
Exp = Exp "-" Exp
Exp = Exp "<" Exp
Exp = "if" Exp "then" Exp "else" Exp
Exp = Id "(" {Exp ","}* ")"
```



signature  
sorts Exp Var  
constructors

```
Int    : IntConst -> Exp
Var    : ID -> Exp
Times  : Exp * Exp -> Exp
Minus  : Exp * Exp -> Exp
Lt     : Exp * Exp -> Exp
If     : Exp * Exp * Exp -> Exp
Call   : ID * List(Exp) -> Exp
```

# Context-Free Grammars with Constructor Declarations

**sorts** Exp Var  
**context-free syntax**

Exp.Int = IntConst  
Exp.Var = Id  
Exp.Times = Exp "\*" Exp  
Exp.Minus = Exp "-" Exp  
Exp.Lt = Exp "<" Exp  
Exp.If = "if" Exp "then" Exp "else" Exp  
Exp.Call = Id "(" {Exp ","}\* ")"

**grammar**  
**non-terminals** Exp Var  
**productions**

Exp = IntConst  
Exp = Id  
Exp = Exp "\*" Exp  
Exp = Exp "-" Exp  
Exp = Exp "<" Exp  
Exp = "if" Exp "then" Exp "else" Exp  
Exp = Id "(" {Exp ","}\* ")"

**signature**  
**sorts** Exp Var  
**constructors**

Int : IntConst -> Exp  
Var : ID -> Exp  
Times : Exp \* Exp -> Exp  
Minus : Exp \* Exp -> Exp  
Lt : Exp \* Exp -> Exp  
If : Exp \* Exp \* Exp -> Exp  
Call : ID \* List(Exp) -> Exp

# Context-Free Grammars with Constructor Declarations

```
sorts Exp Var
context-free syntax
Exp.Int    = IntConst
Exp.Var    = Id
Exp.Times  = Exp "*" Exp
Exp.Minus  = Exp "-" Exp
Exp.Lt     = Exp "<" Exp
Exp.If     = "if" Exp "then" Exp "else" Exp
Exp.Call   = Id "(" {Exp ","}* ")"
```

Abstract syntax: productions define  
constructor and sorts of arguments

Concrete syntax: productions define  
notation for language constructs

# CFG with Constructors defines Abstract and Concrete Syntax

## Abstract syntax

- Production defines constructor, argument sorts, result sort
- Abstract from notation: lexical elements of productions

## Concrete syntax

- Productions define context-free grammar rules

## Some details to discuss

- Sequences
- Lexical syntax
- Ambiguities
- Converting text to tree and back (parsing, unparsing)

# Sequences (Lists)



# Encoding Sequences (Lists)

```
printlist(merge(list1,list2))
```

```
Call("printlist"  
  , [Call("merge", [Var("list1")  
                    , Var("list2")])] )
```

sorts Exp Var

context-free syntax

Exp.Int = IntConst

Exp.Var = Id

Exp.Times = Exp "\*" Exp

Exp.Minus = Exp "-" Exp

Exp.Lt = Exp "<" Exp

Exp.If = "if" Exp "then" Exp "else" Exp

Exp.Call = Id "(" ExpList ")"

ExpList.Nil =

ExpList = ExpListNE

ExpListNE.One = Exp

ExpListNE.Snoc = ExpListNE "," Exp

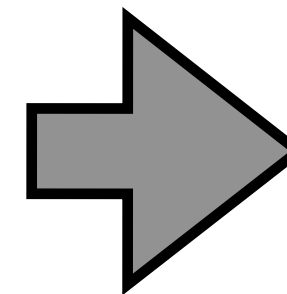
# Sugar for Sequences and Optionals

```
printlist(merge(list1,list2))
```

```
Call("printlist"  
  , [Call("merge", [Var("list1")  
                    , Var("list2")])] )
```

context-free syntax

```
Exp.Call = Id "(" {Exp ","}* ")"
```



context-free syntax

// automatically generated

```
{Exp ","}* . Nil = // empty list  
{Exp ","}*      = {Exp ","}+
```

```
{Exp ","}+ . One = Exp  
{Exp ","}+ . Snoc = {Exp ","}+ "," Exp
```

```
Exp* . Nil = // empty list  
Exp*       = Exp+
```

```
Exp+ . One = Exp  
Exp+ . Snoc = Exp+ Exp
```

```
Exp? . None = // no expression  
Exp? . Some = Exp // one expression
```

# Normalizing Lists

## rules

```
Snoc(Nil(), x) -> Cons(x, Nil())
Snoc(Cons(x, xs), y) -> Cons(x, Snoc(xs, y))

One(x) -> Cons(x, Nil())

Nil() -> []
Cons(x, xs) -> [x | xs]
```

## context-free syntax

// automatically generated

```
{Exp “,”}* .Nil = // empty list
{Exp “,”}*      = {Exp “,”}+
```

```
{Exp “,”}+ .One  = Exp
{Exp “,”}+ .Snoc = {Exp “,”}+ “,” Exp
```

```
Exp* .Nil  = // empty list
Exp*      = Exp+
```

```
Exp+ .One  = Exp
Exp+ .Snoc = Exp+ Exp
```

```
Exp? .None = // no expression
Exp? .Some = Exp // one expression
```

# 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**

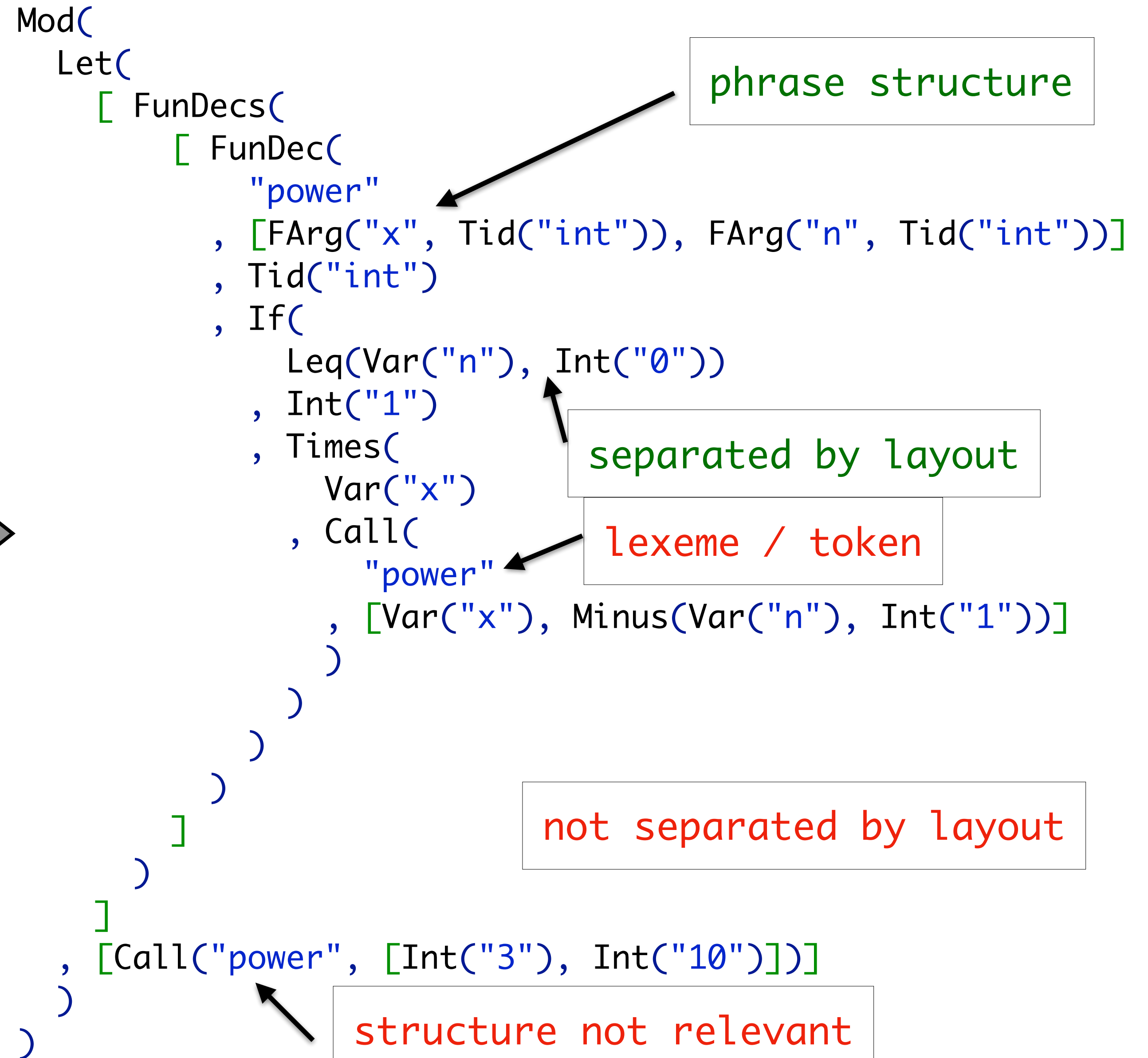
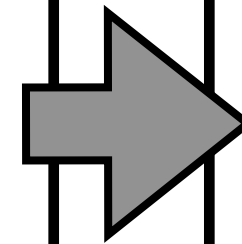
Exp = "let" Dec\* "in" {Exp ";"}\* "end"

Declarations = "declarations" Dec\*

# Lexical Syntax

# Context-Free Syntax vs Lexical Syntax

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



# Character Classes

lexical syntax // character codes

Character =  $[\backslash 65]$

Range =  $[\backslash 65 - \backslash 90]$

Union =  $[\backslash 65 - \backslash 90] \vee [\backslash 97 - \backslash 122]$

Difference =  $[\backslash 0 - \backslash 127] / [\backslash 10 \backslash 13]$

Union =  $[\backslash 0 - \backslash 9 \backslash 11 - \backslash 12 \backslash 14 - \backslash 255]$

Character class represents choice from a set of characters

# Sugar for Character Classes

lexical syntax // sugar

CharSugar = [a]  
= [\97]

CharClass = [abcdefghijklmnopqrstuvwxyz]  
= [\97-\122]

SugarRange = [a-z]  
= [\97-\122]

Union = [a-z] ∨ [A-Z] ∨ [0-9]  
= [\48-\57\65-\90\97-\122]

RangeCombi = [a-z0-9\\_]  
= [\48-\57\95\97-\122]

Complement = ~[\n\r]  
= [\0-\255] / [\10\13]  
= [\0-\9\11-\12\14-\255]

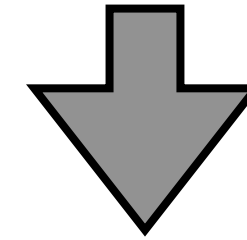


# Literals are Sequences of Characters

lexical syntax // literals

Literal = "then" // case sensitive sequence of characters

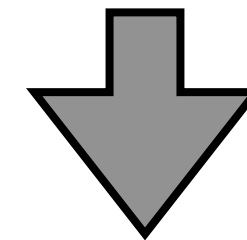
CaseInsensitive = 'then' // case insensitive sequence of characters



syntax

"then" = [t] [h] [e] [n]

'then' = [Tt] [Hh] [Ee] [Nn]



syntax

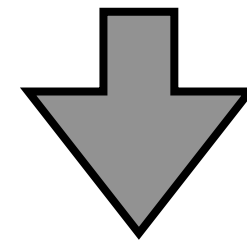
"then" = [\116] [\104] [\101] [\110]

'then' = [\84\116] [\72\104] [\69\101] [\78\110]

# Identifiers

lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*



```
Id = a
Id = B
Id = cD
Id = xyz10
Id = internal_
Id = CamelCase
Id = lower_case
Id = ...
```

# Lexical Ambiguity: Longest Match

lexical syntax

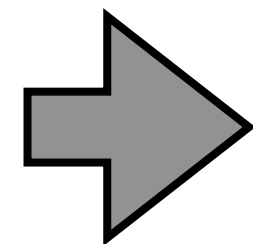
Id = [a-zA-Z][a-zA-Z0-9\\_]\*

context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left} // curried function call

ab



```
Mod(  
  amb(  
    [Var("ab"),  
      Call(Var("a"), Var("b"))]  
  )  
)
```

# Lexical Ambiguity: Longest Match

lexical syntax

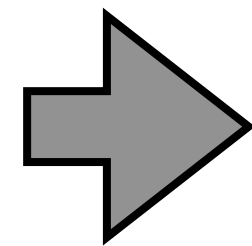
Id = [a-zA-Z][a-zA-Z0-9\\_]\*

context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left} // curried function call

abc



```
Mod(  
  amb(  
    [ amb(  
      [ Var("abc")  
        , Call(  
          amb(  
            [Var("ab"), Call(Var("a"), Var("b"))]  
          )  
          , Var("c")  
        )  
      ]  
    )  
    , Call(Var("a"), Var("bc"))  
  ]  
)
```

# Lexical Restriction => Longest Match

lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

lexical restrictions

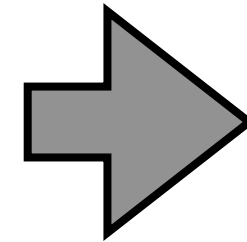
Id -/- [a-zA-Z0-9\\_]  
// longest match for identifiers

context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left}  
// curried function call

abc def ghi



Call(Call(Var("abc"), Var("def")), Var("ghi"))

Lexical restriction: phrase cannot be followed by character in character class

# Lexical Ambiguity: Keywords overlap with Identifiers

lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

lexical restrictions

Id -/- [a-zA-Z0-9\\_]  
// longest match for identifiers

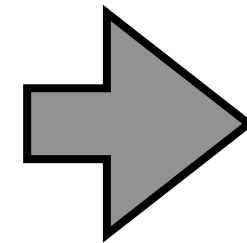
context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left}

Exp.IfThen = "if" Exp "then" Exp

if def then ghi



```
amb(  
  [ Mod(  
    Call(  
      Call(Call(Var("if"), Var("def")), Var("then"))  
      , Var("ghi")  
    )  
    , Mod(IfThen(Var("def"), Var("ghi")))  
  ]  
)
```

# Lexical Ambiguity: Keywords overlap with Identifiers

lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

lexical restrictions

Id -/- [a-zA-Z0-9\\_]

// longest match for identifiers

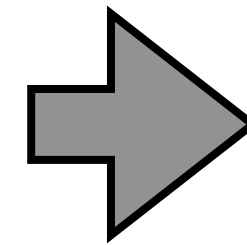
context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left}

Exp.IfThen = "if" Exp "then" Exp

ifdef then ghi



```
amb(  
  [ Mod(  
    Call(Call(Var("ifdef"), Var("then")), Var("ghi"))  
  )  
  , Mod(IfThen(Var("def"), Var("ghi")))  
  ]  
)
```

# Reject Productions => Reserved Words

## lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

Id = "if" {reject}

Id = "then" {reject}

## lexical restrictions

Id -/- [a-zA-Z0-9\\_]  
// longest match for identifiers

"if" "then" -/- [a-zA-Z0-9\\_]

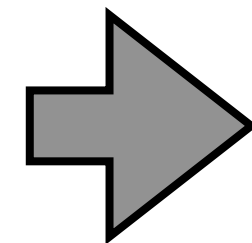
## context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left}

Exp.IfThen = "if" Exp "then" Exp

if def then ghi



IfThen(Var("def"), Var("ghi"))



# Reject Productions => Reserved Words

## lexical syntax

Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

Id = "if" {reject}

Id = "then" {reject}

## lexical restrictions

Id -/- [a-zA-Z0-9\\_]

// longest match for identifiers

"if" "then" -/- [a-zA-Z0-9\\_]

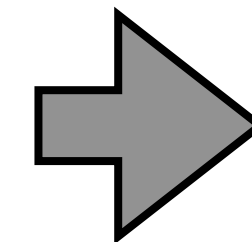
## context-free syntax

Exp.Var = Id

Exp.Call = Exp Exp {left}

Exp.IfThen = "if" Exp "then" Exp

ifdef then ghi



parse error

# Real Numbers

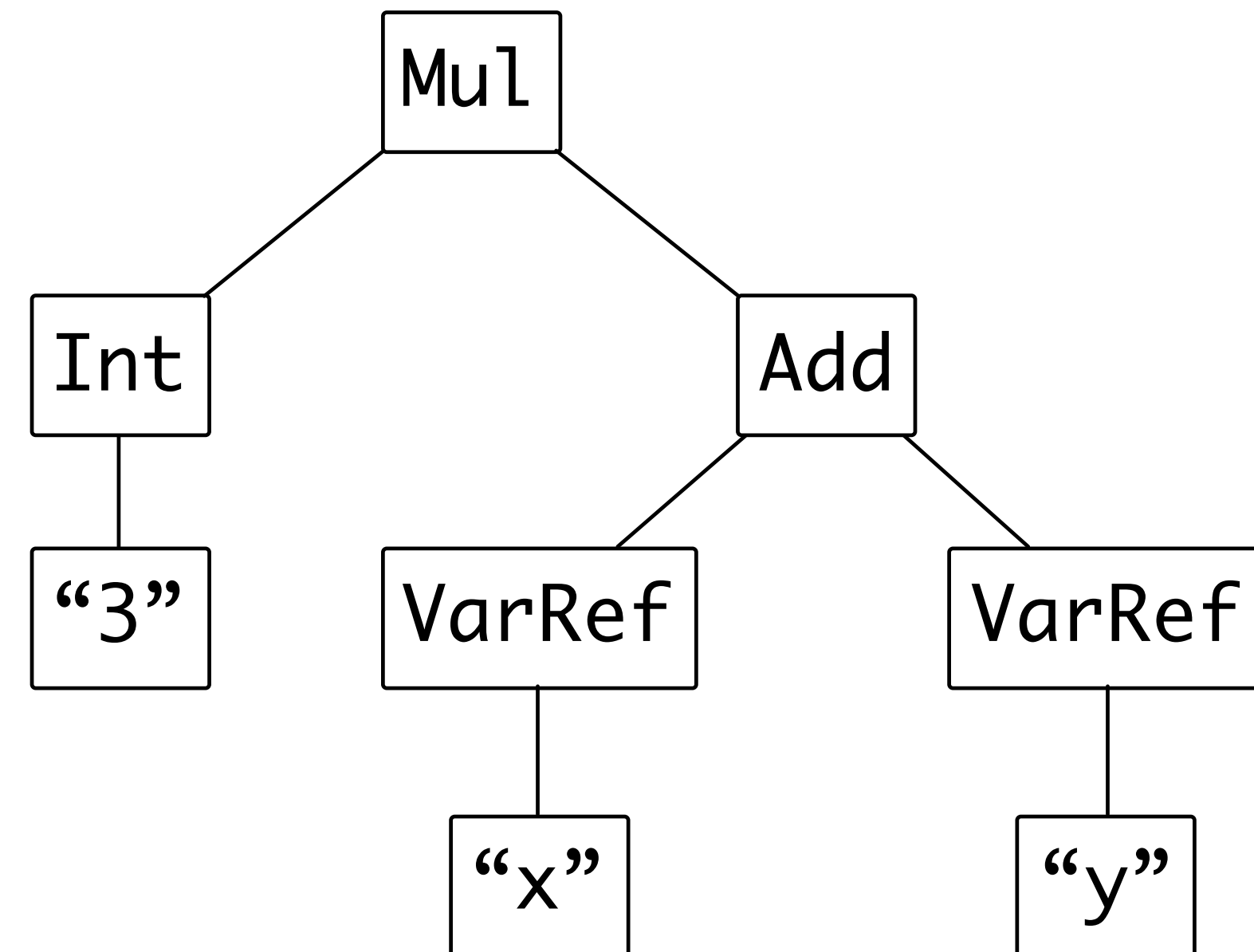
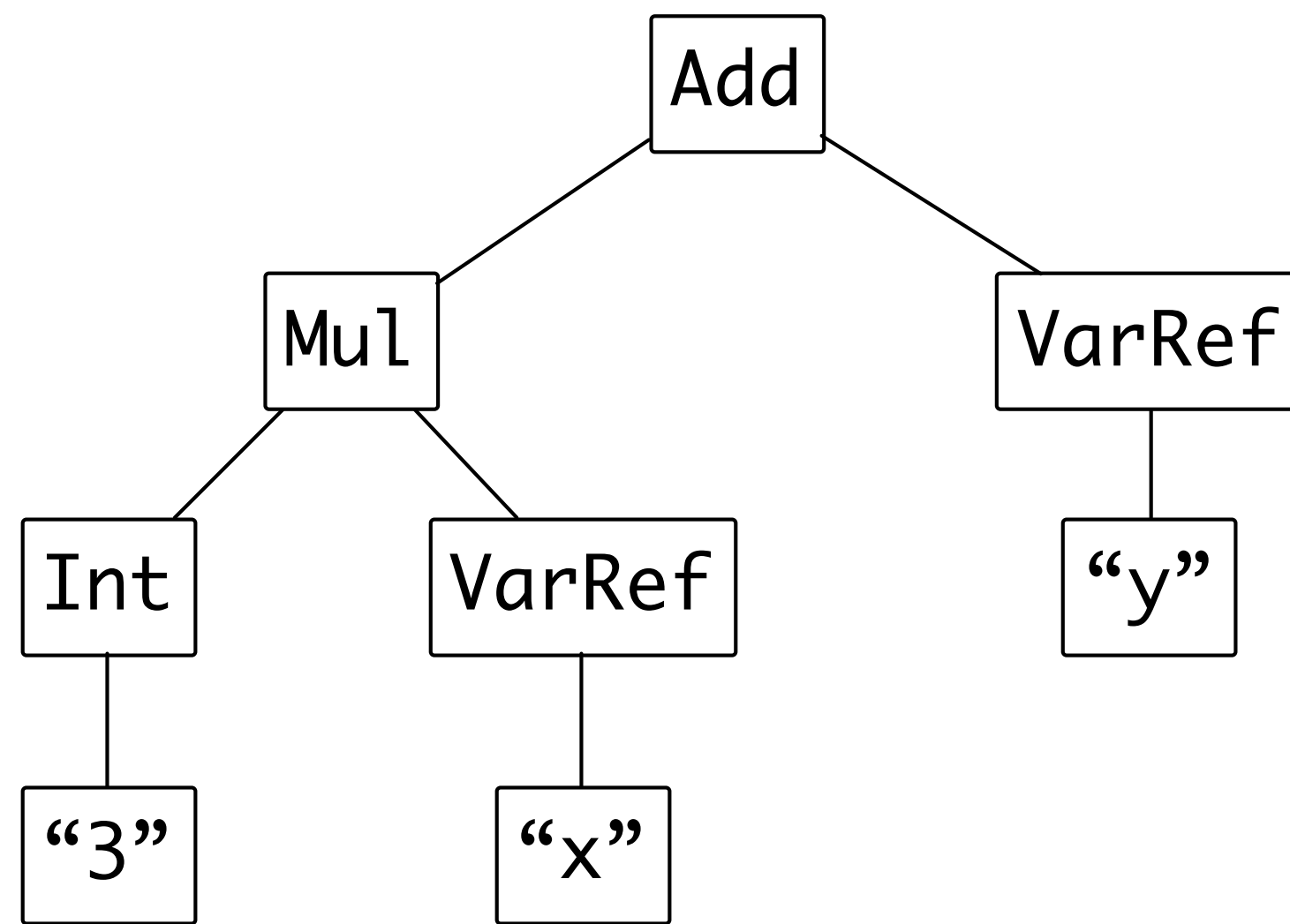
lexical syntax

```
RealConst.RealConstNoExp = IntConst "." IntConst  
RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst  
Sign = "+"  
Sign = "-"
```

# Ambiguity & Disambiguation

# Ambiguity

3 \* x + y



$t1 \neq t2 \wedge \text{format}(t1) = \text{format}(t2)$

ExpressionsAmb.sdf3 Records.sdf3 LMR.sdf3

```

10 Expr.True  = <true>
11 Expr.False = <false>
12
13 Expr       = <<VarRef>>
14
15 Expr.Add   = <<Expr> + <Expr>>
16 Expr.Sub   = <<Expr> - <Expr>>
17 Expr.Mul   = <<Expr> * <Expr>>
18 Expr.Div   = <<Expr> / <Expr>>
19 Expr.And   = <<Expr> & <Expr>>
20 Expr.Or    = <<Expr> | <Expr>>
21 Expr.Eq    = <<Expr> == <Expr>>
22 Expr.App   = <<Expr> <Expr>>
23
24 Expr.If    = <
25   if <Expr> then
26     <Expr>
27   else
28     <Expr>
29 > {longest-match}
30
31 Expr.Fun    = <fun (<ArgDecl>) { <Expr> }>
32 ArgDecl.ArgDecl = <<VarId> : <Type>>
33
34 Expr.Let    = <let <DefBind+> in <Expr>>
35 Expr.LetRec = <letrec <DefBind+> in <Expr>>
36 Expr.LetPar = <letpar <DefBind+> in <Expr>>
37
38 DefBind.DefBind      = <<VarId> = <Expr>>
39 DefBind.DefBindTyped = <<VarId> : <Type> = <Expr>>
40

```

\*amb01.lmr test01.lmr

 1 a + b \* x - 1

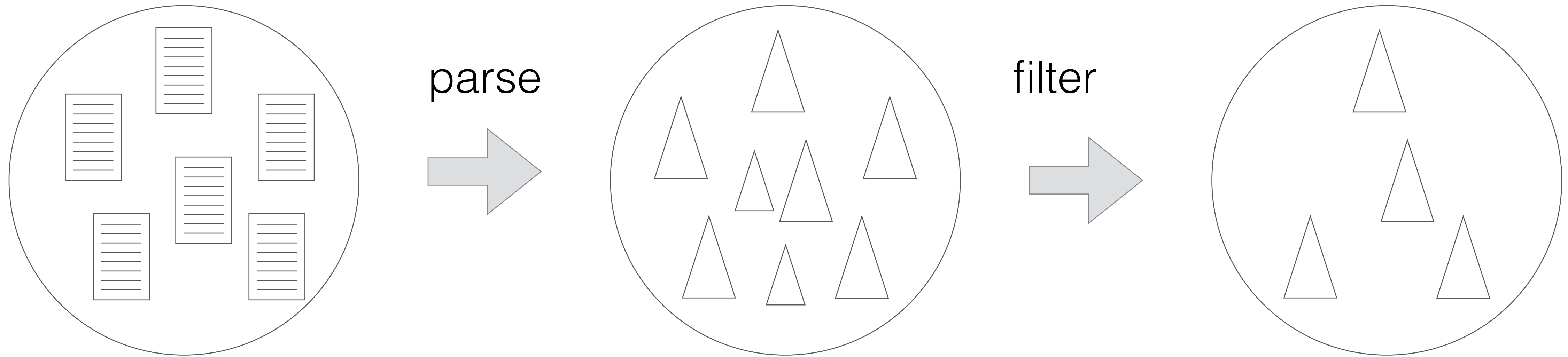
test01.aterm amb01.aterm

```

1 amb(
2   [ amb(
3     [ Sub(
4       amb(
5         [ Mul(Add(VarRef("a"), VarRef("b")), VarRef("x"))
6           , Add(VarRef("a"), Mul(VarRef("b"), VarRef("x")))
7         ]
8       )
9     , Int("1")
10  )
11  , Add(
12    VarRef("a")
13  , amb(
14    [ Sub(Mul(VarRef("b"), VarRef("x")), Int("1"))
15      , Mul(VarRef("b"), Sub(VarRef("x"), Int("1")))
16    ]
17  )
18  )
19  ]
20  )
21  , Mul(
22    Add(VarRef("a"), VarRef("b"))
23    , Sub(VarRef("x"), Int("1"))
24  )
25  ]
26 )

```

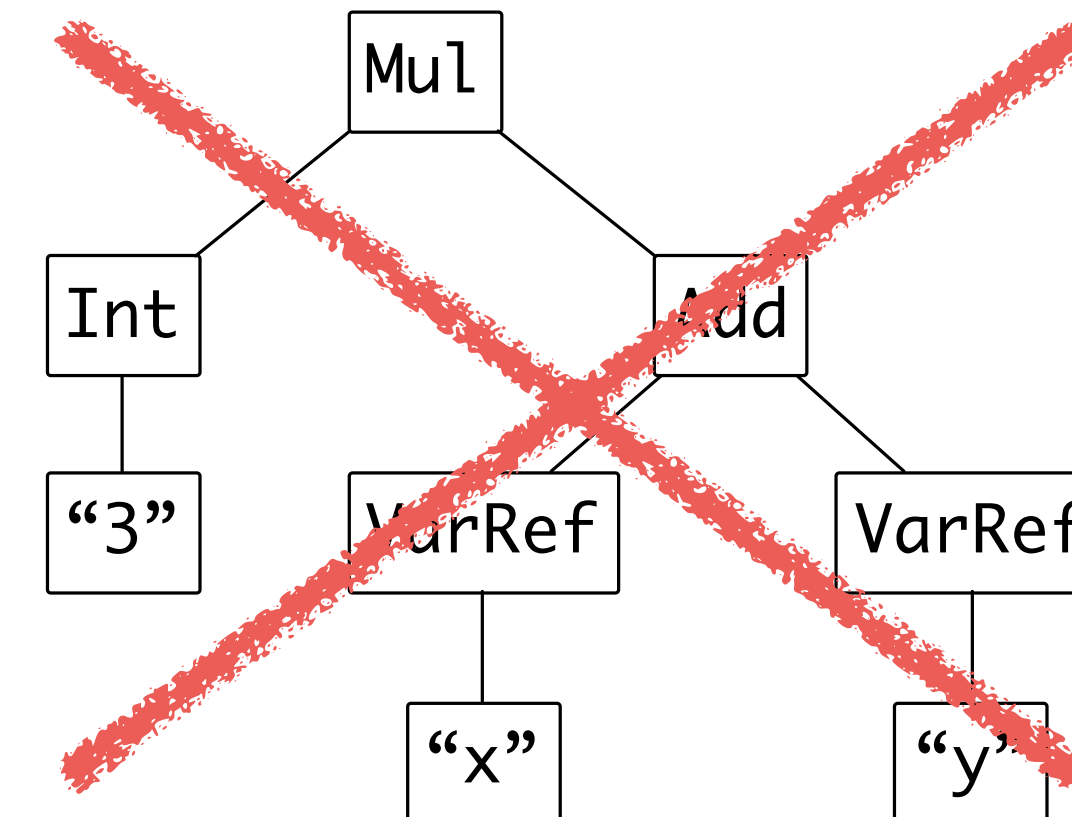
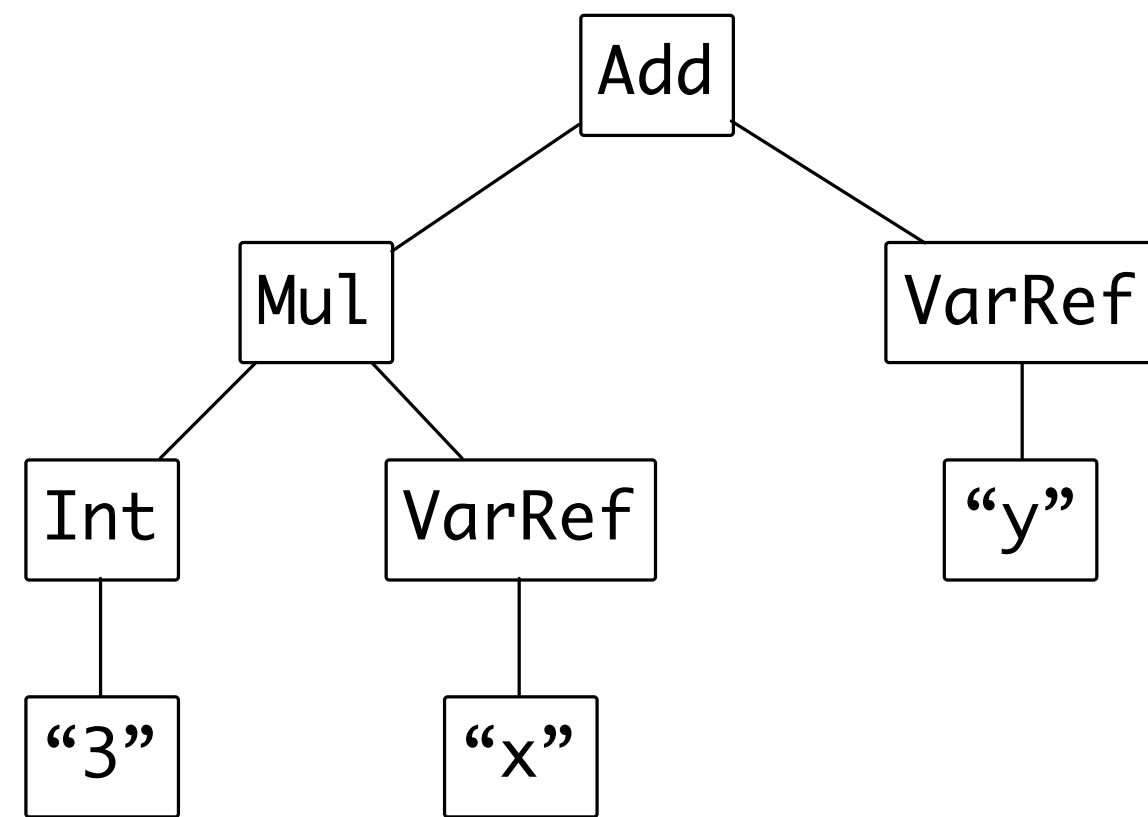
# Disambiguation Filters



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

# Associativity and Priority

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 [SLE13, Onward15, Programming18]



# Ambiguous Operator Syntax

## 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



# Disambiguation by Associativity and Priority Rules

## 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
```

# Disambiguation by Associativity and Priority Rules

## context-free syntax

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

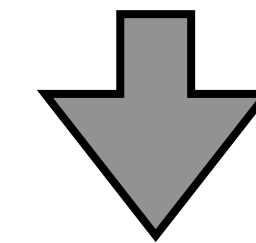
```
Exp.Eq  = Exp "="  Exp {non-assoc}
Exp.Neq = Exp "<>" Exp {non-assoc}
Exp.Gt  = Exp ">"  Exp {non-assoc}
Exp.Lt  = Exp "<"  Exp {non-assoc}
Exp.Gte = Exp ">=" Exp {non-assoc}
Exp.Lte = Exp "<=" Exp {non-assoc}
```

```
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
```

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



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

# Testing Syntax Definitions with SPT

# 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

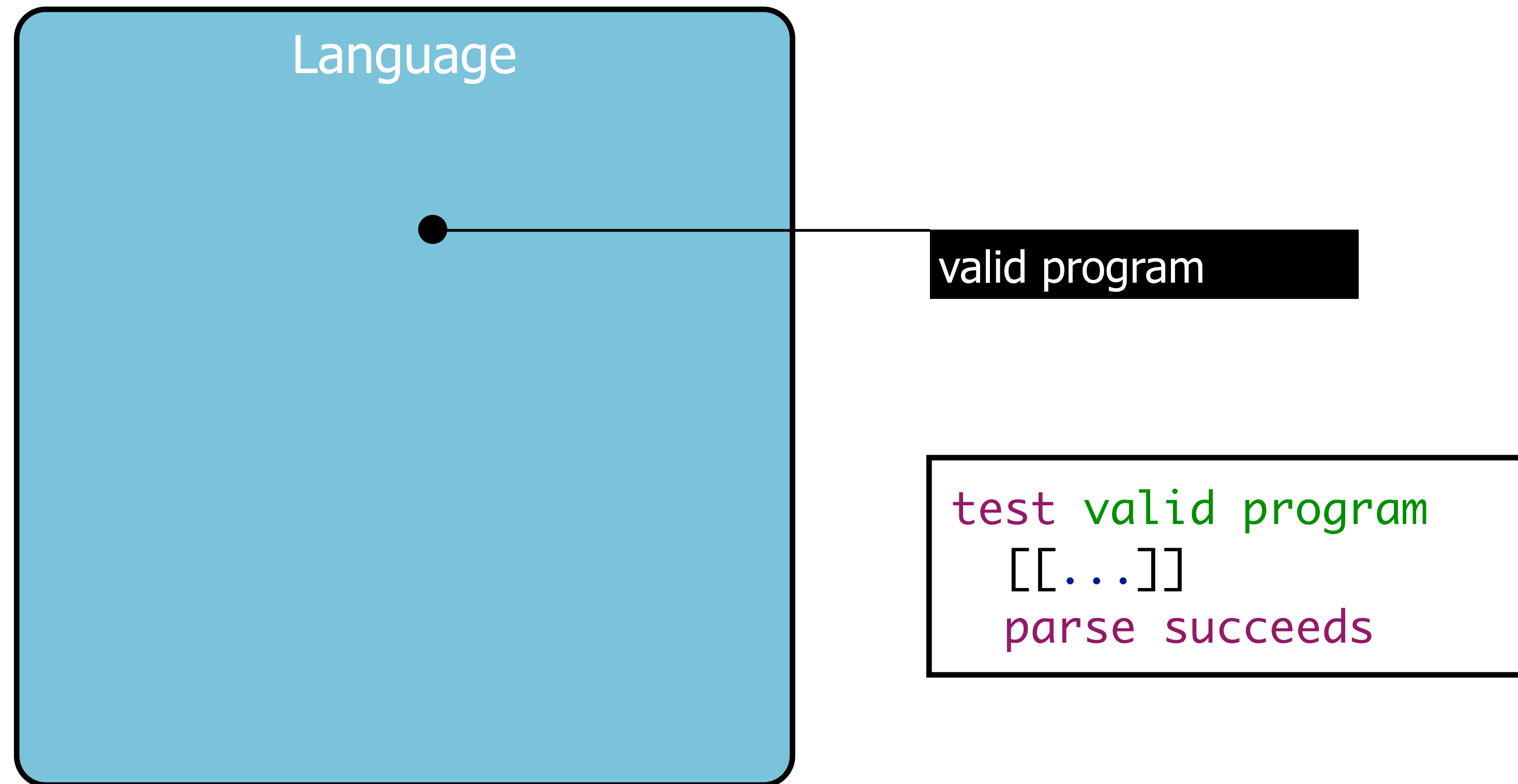
test this test fails [[
  1 + 3 * 4
]] parse fails

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

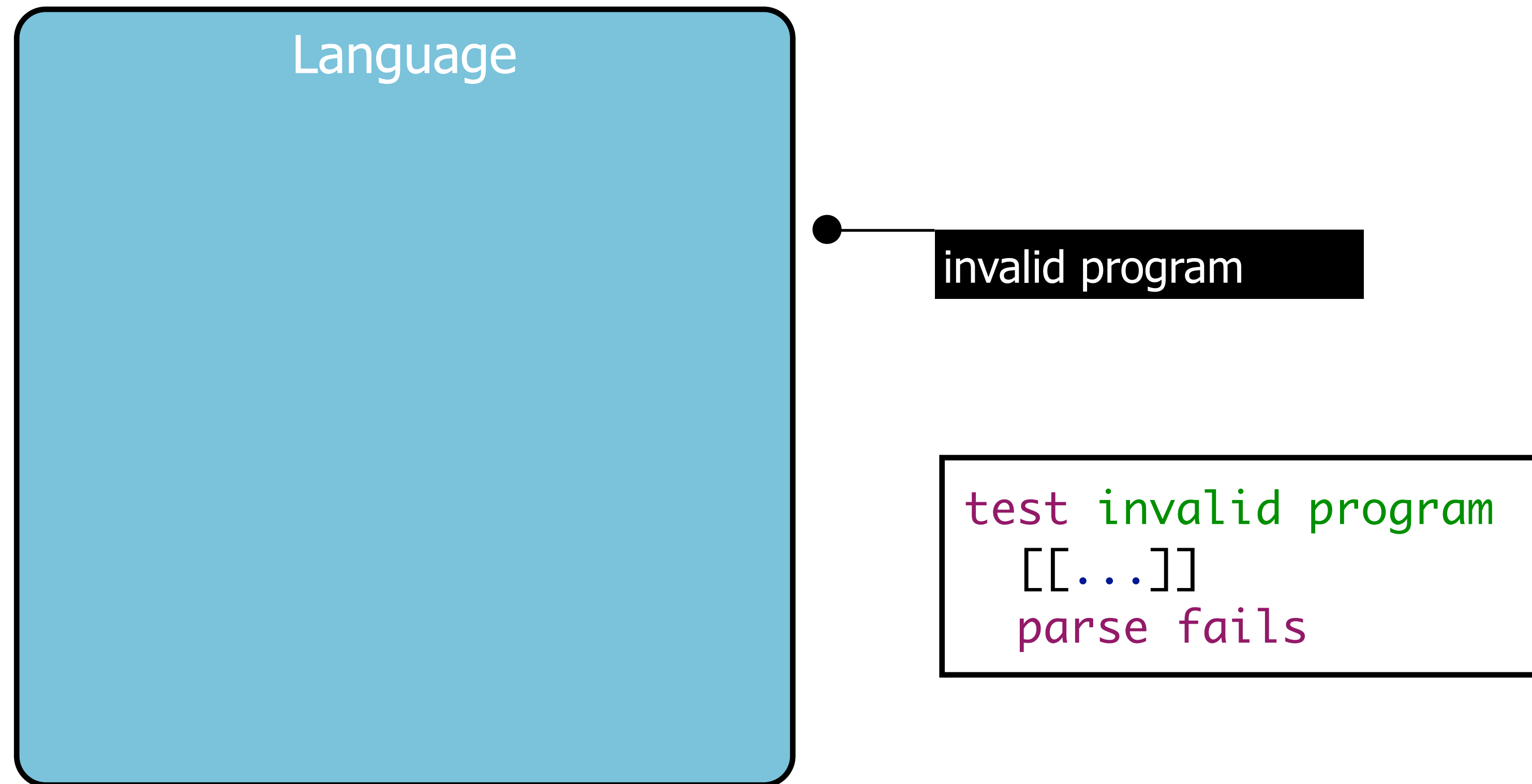
test this test succeeds [[
  1 + 3 *
]] parse fails
```

```
1 module success-failure
2
3 language Tiger
4
5 test this test succeeds [[
6   1 + 3 * 4
7 ]] parse succeeds
8
9  test this test fails [[
10   1 + 3 * 4
11 ]] parse fails
12
13  test this test fails [[
14   1 + 3 *
15 ]] parse succeeds
16
17 test this test succeeds [[
18   1 + 3 *
19 ]] parse fails
```

# Test Cases: Valid



# Test Cases: Invalid



# 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
```

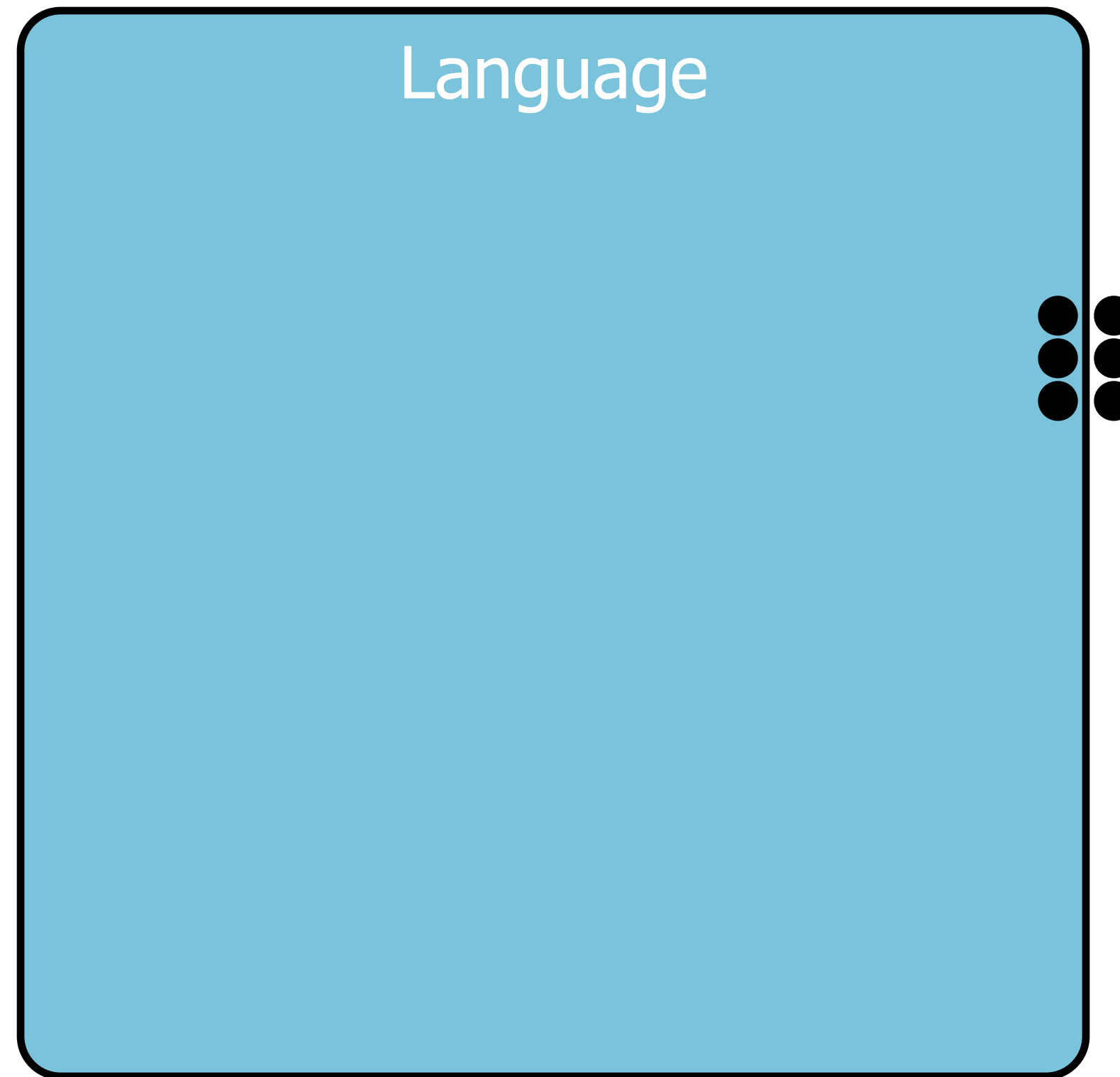
```
test single uc digit   [[X1]] parse succeeds
```

```
test single digit uc   [[1X]] parse fails
```

```
test double digit      [[11]] parse fails
```



# Test Corner Cases



# Testing Structure

```
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")))
```

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))))

# 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 Spoofox

workspace - Java - org.metaborg.lang.tiger/syntax/Numbers.sdf3 - Eclipse

Package Explorer

- Alarms.sar3
- 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.featu
- org.metaborg.lang.tiger.eclipse.site
- > org.metaborg.lang.tiger.example [1]
  - 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

```
1 module Numbers
2
3 lexical syntax
4
5 IntConst = [0-9]+
6
7 lexical syntax
8
9 RealConst.RealConstNoExp = IntConst "." IntConst
10 RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
11 Sign = "+"
12 Sign = "-"
13
14 context-free syntax
15
16 Exp.Int      = IntConst
17
18 Exp.Uminus   = [- [Exp]]
19 Exp.Times    = [[Exp] * [Exp]] {left}
20 Exp.Divide   = [[Exp] / [Exp]] {left}
21 Exp.Plus     = [[Exp] + [Exp]] {left}
22 Exp.Minus    = [[Exp] - [Exp]] {left}
23
24 Exp.Eq       = [[Exp] = [Exp]] {non-assoc}
25 Exp.Neq      = [[Exp] <> [Exp]] {non-assoc}
26 Exp.Gt       = [[Exp] > [Exp]] {non-assoc}
27 Exp.Lt       = [[Exp] < [Exp]] {non-assoc}
28 Exp.Geq      = [[Exp] >= [Exp]] {non-assoc}
29 Exp.Leq      = [[Exp] <= [Exp]] {non-assoc}
30
31 Exp.And      = [[Exp] & [Exp]] {left}
32 Exp.Or       = [[Exp] | [Exp]] {left}
33
34 //Exp = [[Exp]] {bracket, avoid}
35
36 context-free priorities
37
38 // Precedence of operators: Unary minus has the highest
39 // precedence. The operators *, / have the next highest
40 // (tightest binding) precedence, followed by +, -, then
41 // by =, <>, >, <, >=, <=, then by &, then by |.
42
43 // Associativity of operators: The operators *, /, +, -
44 // are all left associative. The comparison operators do
45 // not associate, so a = b = c is not a legal expression,
46 // a = (b = c) is legal.
47
48 {Exp.Uminus}
49 > {left :
50   Exp.Times
51   Exp.Divide}
```

power.tig

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

power.aterm

```
1 Mod(
2   Let(
3     [ FunDecs(
4       [ FunDec(
5         "power"
6         , [FArg("x", Tid("int")), FArg("n", Tid("int"))]
7         , Tid("int")
8         , If(
9           Leq(Var("n"), Int("0"))
10          , Int("1")
11          , Times(
12            Var("x")
13            , Call(
14              "power"
15              , [Var("x"), Minus(Var("n"), Int("1"))]
16            )
17          )
18        )
19      )
20    ]
21  )
22 , [Call("power", [Int("3"), Int("10")])]
23 )
24 )
25 )
```

structure.spt

```
1 module structure
2
3 language Tiger
4
5 test if [[
6   if x then 3 else 4
7 ]] parse to Mod(If(Var("x"), Int("3"), Int("4")))
8
9 test add times [[
10  21 + 14 + 7
11 ]] parse to Mod(Plus(Int("21"), Times(Int("14"), Int("7"))))
12
13 test times add [[
14  3 * 7 + 21
15 ]] parse to Mod(Plus(Times(Int("3"), Int("7")), Int("21")))
16
```

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

# **Declarative Syntax**

## **Definition: Summary**



# Declarative Syntax Definition

## Syntax definition

- Define structure (decomposition) of programs
- Define concrete syntax: notation
- Define abstract syntax: constructors

## Using syntax definitions (next)

- Parsing: converting text to abstract syntax term
- Pretty-printing: convert abstract syntax term to text
- Editor services: syntax highlighting, syntax checking, completion

# Next: Parsing

Except where otherwise noted, this work is licensed under

