

# Lecture 6(a): Introduction to Static Analysis

**Eelco Visser**

**CS4200 Compiler Construction**

**TU Delft**

**October 2018**

# Why Type Checking?

# Why Type Checking? Some Discussion Points

## Dynamically Typed vs Statically Typed

- Dynamic: type checking at run-time
- Static: type checking at compile-time (before run-time)

## What does it mean to type check?

- Type safety: guarantee absence of run-time type errors

## Why static type checking?

- Avoid overhead of run-time type checking
- Fail faster: find (type) errors at compile time
- Find all (type) errors: some errors may not be triggered by testing
- But: not all errors can be found statically (e.g. array bounds checking)

# Context-Sensitive Properties

# Homework Assignment: What is the Syntax of This Language?

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
  </book>
  <book>
    <title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
</catalogue>
```

```
<languages>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <language>
    <name>Stratego</name>
    <purpose>Transformation</purpose>
    <implementedin>SDF3</implementedin>
    <implementedin>Stratego</implementedin>
    <target>Java</target>
  </language>
</languages>
```

# Syntax of Book Catalogues

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
  </book>
  <book>
    <title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
</catalogue>
```

Schema-specific syntax definition

context-free syntax

```
Document.Catalogue = [
  <catalogue>
    [Book*]
  </catalogue>
]

Book.Book = [
  <book>
    [Title]
    [Author]
    [Publisher]
  </book>
]

...
```

# A Generic Syntax of XML Documents

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
  </book>
  <book>
    <title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
</catalogue>
```

```
Doc(
  Node(
    "catalogue"
  , [ Node(
      "book"
    , [ Node("title", [Text(["Modern Compiler Implementation in ML"])], "title")
      , Node("author", [Text(["Andrew Appel"])], "author")
      , Node("publisher", [Text(["Cambridge"])], "publisher")
    ]
    , "book"
  )
  , Node(
      "book"
    , [ Node("title", [Text(["Parsing Schemata"])], "title")
      , Node("author", [Text(["Klaas Sikkel"])], "author")
      , Node("publisher", [Text(["Springer"])], "publisher")
    ]
    , "book"
  )
  , "catalogue"
)
)
```

context-free start-symbols Document

sorts Tag Word

lexical syntax

Tag =  $[a-zA-Z][a-zA-Z0-9]^*$

Word =  $\sim[\<\>]^+$

lexical restrictions

Word -/-  $\sim[\<\>]$

sorts Document Elem

context-free syntax

Document.Doc = Elem

Elem.Node = [  
 <[Tag]>  
 [Elem\*]  
 </[Tag]>  
>

Elem.Text = Word+ {longest-match}

# A Generic Syntax of XML Documents

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
  </book>
  <book>
    <title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
</catalogue>
```

What is the problem with this approach?

context-free start-symbols Document

sorts Tag Word

lexical syntax

Tag = [a-zA-Z][a-zA-Z0-9]\*

Word = ~[\<\>]+

lexical restrictions

Word -/- ~[\<\>]

sorts Document Elem

context-free syntax

Document.Doc = Elem

Elem.Node = [

<[Tag]>

[Elem\*]

</[Tag]>

]

Elem.Text = Word+ {longest-match}



# Generic Syntax is Too Liberal!

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <book>
    //<title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
  <book>
    </koob>
  </catalogue>
```

Year is not a valid element of book

Only books in catalogue

Book should have a title

Closing tag is not consistent with starting tag

context-free start-symbols Document

sorts Tag Word

lexical syntax

Tag =  $[a-zA-Z][a-zA-Z0-9]^*$

Word =  $\sim[\<\>]^+$

lexical restrictions

Word  $-/- \sim[\<\>]$

sorts Document Elem

context-free syntax

Document.Doc = Elem

Elem.Node = [

<[Tag]>

[Elem\*]

</[Tag]>

]

Elem.Text = Word+ {longest-match}

# Context-Sensitive Properties

**Context-free grammar is ... context-free!**

- Cannot express alignment

**Languages have context-sensitive properties**

**How can we have our cake and eat it too?**

- Generic (liberal) syntax
- Forbid programs/documents that are not well-formed

# Checking Context-Sensitive Properties

# Approach: Checking Context-Sensitive Properties

## Generic (liberal) syntax

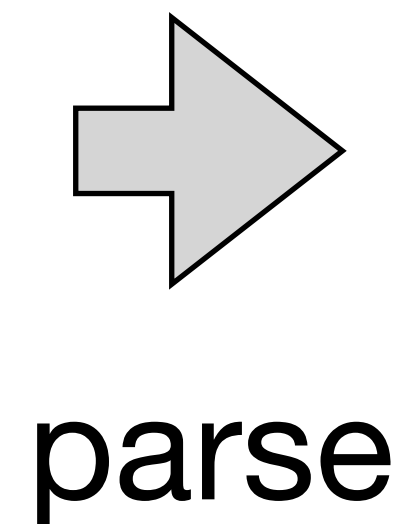
- Allow more programs/documents

## Check properties on AST

- Reject programs/documents that are not well-formed

# Checking Context-Context-Sensitive Properties in Spoofax

```
errors.xml
1 <catalogue>
2   <book>
3     <title>Modern Compiler Implementation in ML</title>
4     <author>Andrew Appel</author>
5     <publisher>Cambridge</publisher>
6     <year></year>
7   </book>
8   <language>
9     <name>SDF3</name>
10    <purpose>Syntax Definition</purpose>
11    <implementedin>SDF3</implementedin>
12  </language>
13  <book>
14    //<title>Parsing Schemata</title>
15    <author>Klaas Sikkel</author>
16    <publisher>Springer</publisher>
17  </book>
18  <book>
19  </koob>
20 </catalogue>
21
```



```
Doc(
  Node(
    "catalogue"
    , [ Node(
        "book"
        , [ Node("title", [Text(["Modern Compiler Implementation in ML"])], "title")
          , Node("author", [Text(["Andrew Appel"])], "author")
          , Node("publisher", [Text(["Cambridge"])], "publisher")
          , Node("year", [], "year")
        ]
        , "book"
      )
    , Node(
        "language"
        , [ Node("name", [Text(["SDF3"])], "name")
          , Node("purpose", [Text(["Syntax Definition"])], "purpose")
          , Node("implementedin", [Text(["SDF3"])], "implementedin")
        ]
        , "language"
      )
    , Node(
        "book"
        , [ Node("author", [Text(["Klaas Sikkel"])], "author")
          , Node("publisher", [Text(["Springer"])], "publisher")
        ]
        , "book"
      )
    , Node("book", [], "koob")
  ]
)
```

errors

rules // Analysis

editor-analyze:

(ast, path, project-path) -> (ast', error\*, warning\*, info\*)

with

ast' := <id> ast

; error\* := <collect-all(constraint-error)> ast'

; warning\* := <collect-all(constraint-warning)> ast'

; info\* := <collect-all(constraint-info)> ast'

# Check Violations using Constraint-Error Rules

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <book>
    //<title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
  <book>
  </book>
</catalogue>
```

Find all sub-terms that are not  
consistent with the context-  
sensitive rules

collect-all: type-unifying  
generic traversal

```
rules // Analysis
```

```
editor-analyze:
```

```
(ast, path, project-path) -> (ast', error*, warning*, info*)
```

```
with
```

```
  ast'      := <id> ast
```

```
; error*    := <collect-all(constraint-error)> ast'
```

```
; warning*  := <collect-all(constraint-warning)> ast'
```

```
; info*     := <collect-all(constraint-info)> ast'
```



# Check Violations using Constraint-Error Rules

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <book>
    //<title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
  <book>
    </koob>
  </book>
</catalogue>
```

Closing tag does not match starting tag

Origin

Error message

rules

constraint-error :

Node(tag1, elems, tag2) -> (tag2, \$[Closing tag does not match starting tag])

where <not(eq)>(tag1, tag2)

# Check Violations using Constraint-Error Rules

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <book>
    //<title>Parsing Schemata</title>
    <author>Klaas Sikkels</author>
    <publisher>Springer</publisher>
  </book>
  <book>
  </book>
</catalogue>
```

Containment checks

Book should have a title

rules

```
constraint-error :
  n@Node(tag@"book", elems, _) -> (tag, $[Book should have title])
  where <not(has(l"title"))> elems
```

```
has(ltag) = fetch(?Node(tag, _, _))
```



# Check Violations using Constraint-Error Rules

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
```

```
<language>
  <name>SDF3</name>
  <purpose>Syntax Definition</purpose>
  <implementedin>SDF3</implementedin>
</language>
```

```
<book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
</book>
<book>
</koob>
</catalogue>
```

Containment checks

Catalogue can only have books

rules

constraint-error :

```
Node("catalogue", elems, _) -> error
where <filter(not-a-book)> elems => [error | _]
```

not-a-book :

```
Node(tag, _, _) -> (tag, $[Catalogue can only have books])
where <not(eq)> (tag, "book")
```

# Check Violations using Constraint-Error Rules

```
<catalogue>
  <book>
    <title>Modern Compiler Implementation in ML</title>
    <author>Andrew Appel</author>
    <publisher>Cambridge</publisher>
    <year></year>
  </book>
  <language>
    <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
  </language>
  <book>
    //<title>Parsing Schemata</title>
    <author>Klaas Sikkel</author>
    <publisher>Springer</publisher>
  </book>
  <book>
  </book>
</catalogue>
```

Book can only have title, author, publisher

Containment checks

rules

constraint-error :

Node("book", elems, \_) -> error

where <filter(not-a-book-elm)> elems => [error | \_]

not-a-book-elm :

Node(tag, \_, \_) -> (tag, \$[Book can only have title, author, publisher])

where <not(elem())> (tag, ["title", "author", "publisher"])

# Approach: Checking Context-Sensitive Properties

## Generic (liberal) syntax

- Allow more programs/documents
- ‘permissive syntax’

## Check properties on AST

- Reject programs/documents that are not well-formed

## Advantage

- Smaller syntax definition
- Parser does not fail (so often)
- Better error messages than parser can give

# **How are programming languages different from XML?**

# Programming Languages vs XML

## XML checking

- Tag consistency
- Schema consistency
- These are structural properties

## Programming languages

- Type consistency (similar to schema?)
- Name consistency: declarations and references should correspond
- Name dependent type consistency: names carry types

# Types

# Why types?

## Why types?

- "guarantee absence of run-time type errors"

## What is a type system?

- A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pierce2002]

## Discuss using a series of untyped examples

- Do you consider the example correct or not, and why?
  - ▶ That is, do you think it should type-check?
- If incorrect: what types will disallow this program?
- If correct: what types will allow this program?

# Preliminaries

```
class A {  
  B b;  
  int m(int i) {  
    return i + b.f  
  }  
}  
  
class B {  
  int f;  
}
```

## How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types ( $x : T$ )
- Expressions have types ( $e : T$ )
  - Including all sub-expressions



# Types Example

4 / "four"

4 : number  
"four" : string  
/ : number \* number → number

simple types

typing prevents undefined runtime behavior

# Types Example

```
7 + (if (true) { 5 } else { "four" })
```

```
7 : number
```

```
5 : number
```

```
"four" : string
```

```
if      : ?
```

no simple type

- typing (over)approximates runtime behavior
- programs without runtime errors can be rejected

# Types Example

```
function id(x) { return x; }  
id(4); id(true);
```

```
4      : number  
true   : boolean  
id     :  $\forall T. T \rightarrow T$ 
```

*polymorphic type*

- richer types approximate behavior better
- depends on runtime representation of values

# Types Example

```
if (a < 5) { 5 } else { "four" }
```

```
5      : number  
"four" : string  
if     : number|string
```

*union type*

- richer types approximate behavior better
- depends on runtime representation of values

# Types Example

```
float distance = 12.0, time = 4.0  
float velocity = time / distance
```

```
distance : float<m>  
time      : float<s>  
velocity  : float<m/s>
```

*unit-of-measure type*

- no runtime problems, but not correct ( $v = d / t$ )
- types can enforce other correctness properties

# What kind of types?

- Simple `int, float → float, bool`
- Named `class A, newtype Id`
- Polymorphic `List<X>, ∀a. a → a`
- Union/sum (one of) `string | string[]`
- Unit-of-measure `float<m>, float<m/s>`
- Structural `{ x: number, y: number }`
- Intersection (all of) `Comparable & Serializable`
- Recursive `μT. int | T * T` (binary int tree)
- Ownership `&mut data`
- Dependent – values in types `Vector 3`
- ... many more ...

# Why types?

## Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - ▶ “Well-typed programs cannot go wrong.” [Reynolds1985]
  - ▶ Also logical properties beyond runtime problems

## What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

## What is the difference between typing and testing?

- Typing is an over-approximation of runtime behavior (proof of absence)
- Testing is an under-approximation of runtime behavior (proof of presence)



# Types and language design

## Types influence language design

- Types abstract over implementation
  - ▶ Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - ▶ As long as the public interface is implemented, dependent modules do not change

## Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
  - ▶ The exact set of values computed for a given input?
- Expressive typing problems become hard to compute
- Many are undecidable, if they imply solving the halting problem
- Designing type systems always involves trade-offs



# Relations between Types

# Comparing Types

```
interface Point2D { x: number, y: number }  
interface Vector2D { x: number, y: number }  
var p1: Point2D = { x: 5, y: -11 }  
var p2: Vector2D = p1
```

**Is this program correct?**

- No, if types are compared by name
- Yes, if types are compared based on structure

# Comparing Types

```
interface Point2D { x: number, y: number }  
interface Point3D { x: number, y: number, z: number }  
var p1: Point3D = { x: 5, y: -11, z: 0 }  
var p2: Point2D = p1
```

## Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is T a subtype of U?
  - ▶ When a value of type T can be used when a value of U is expected
- What about nominal subtypes?
  - ▶ `interface Point3D extends Point2D`

# Combination example: generics and subtyping

```
class A {}  
class B extends A {}  
  
B[] bs = new B[1];  
A[] as = bs;  
as[0] = new A();  
B b = bs[0];
```

*subtyping on arrays &  
mutable updates is unsound*

- unsound = under-approximation of runtime behavior
- feature combinations are not trivial

# Comparing Types

```
int i = 12  
float f = i
```

## Is this program correct?

- No, floats and integers have different runtime representations
- Yes, possible by coercion
  - ▶ Coercion requires insertion of code to convert between representations
- How is this different than subtyping?
  - ▶ Subtyping says that the use of the unchanged value is safe

# Type Relations

## What kind of relations between types?

- Equality  $T = T$  – syntactic or structural
- Subtyping  $T < : T$  – nominal or structural
- Coercion – requires code insertion

# **Implementing a Type Checker with Rewrite Rules**

# Compute Type of Expression

## rules

type-check :  
Mod(e) -> t  
where <type-check> e => t

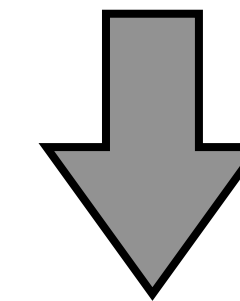
type-check :  
String(\_) -> STRING()

type-check :  
Int(\_) -> INT()

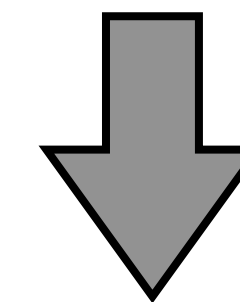
type-check :  
Plus(e1, e2) -> INT()  
where  
    <type-check> e1 => INT();  
    <type-check> e2 => INT()

type-check :  
Times(e1, e2) -> INT()  
where  
    <type-check> e1 => INT();  
    <type-check> e2 => INT()

1 + 2 \* 3



```
Mod(  
  Plus(Int("1"),  
        Times(Int("2"),  
                Int("3")))  
)
```



INT()



# Compute Type of Variable?

rules

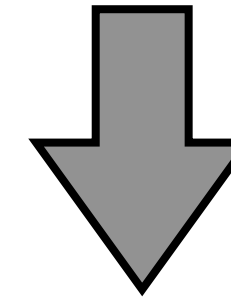
type-check :  
Let([VarDec(x, e)], e\_body) -> t

where

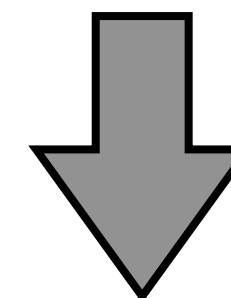
<type-check> e => t\_e;  
<type-check> e\_body => t

type-check :  
Var(x) -> t // ???

```
let  
  var x := 1  
in  
  x + 1  
end
```



```
Mod(  
  Let(  
    [VarDec("x", Int("1"))]  
    , [Plus(Var("x"), Int("1"))]  
  )  
)
```



?

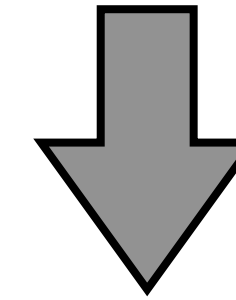
# Type Checking Variable Bindings

rules

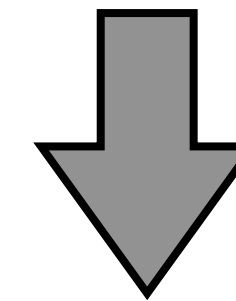
```
type-check(lenv) :  
  Let([VarDec(x, e)], e_body) -> t  
  where  
    <type-check(lenv)> e => t_e;  
    <type-check(l[(x, t_e) | lenv])> e_body => t
```

```
type-check(lenv) :  
  Var(x) -> t  
  where  
    <fetch?(x, t)> env
```

```
let  
  var x := 1  
in  
  x + 1  
end
```



```
Mod(  
  Let(  
    [VarDec("x", Int("1"))]  
    , [Plus(Var("x"), Int("1"))]  
  )  
)
```



Store association between variable and type in type environment

INT()

# Pass Environment to Sub-Expressions

## rules

type-check :

Mod(e) -> t

where <type-check(l[])> e => t

type-check(lenv) :

String(\_) -> STRING()

type-check(lenv) :

Int(\_) -> INT()

type-check(lenv) :

Plus(e1, e2) -> INT()

where

<type-check(lenv)> e1 => INT();

<type-check(lenv)> e2 => INT()

type-check(lenv) :

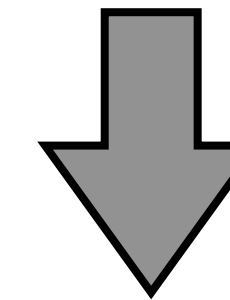
Times(e1, e2) -> INT()

where

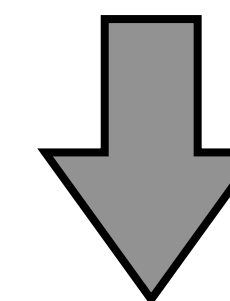
<type-check(lenv)> e1 => INT();

<type-check(lenv)> e2 => INT()

```
let
  var x := 1
in
  x + 1
end
```



```
Mod(
  Let(
    [VarDec("x", Int("1"))],
    [Plus(Var("x"), Int("1"))]
  )
)
```



INT()

# But what about?

## Type checking ill-typed/named programs?

- add rules for ‘bad’ cases

## More complicated name binding patterns?

- Hoisting of variables in JavaScript functions
- Mutually recursive bindings
- Possible approaches
  - ▶ Multiple traversals over program
  - ▶ Defer checking until entire scope is processed
  - ▶ ...

**Name Binding  
Complicates  
Type Checking**

# **Intermezzo: Testing Static Analysis**

# Testing Name Resolution

```
test outer name [[
  let type t = u
    type [[u]] = int
    var x: [[u]] := 0
  in
    x := 42 ;
    let type u = t
      var y: u := 0
    in
      y := 42
    end
  end
]] resolve #2 to #1
```

```
test inner name [[
  let type t = u
    type u = int
    var x: u := 0
  in
    x := 42 ;
    let type [[u]] = t
      var y: [[u]] := 0
    in
      y := 42
    end
  end
]] resolve #2 to #1
```

# Testing Type Checking

```
test integer constant [[
  let type t = u
    type u = int
    var x: u := 0
  in
    x := 42 ;
    let type u = t
      var y: u := 0
    in
      y := [[42]]
    end
  end
]] run get-type to INT()
```

```
test variable reference [[
  let type t = u
    type u = int
    var x: u := 0
  in
    x := 42 ;
    let type u = t
      var y: u := 0
    in
      y := [[x]]
    end
  end
]] run get-type to INT()
```

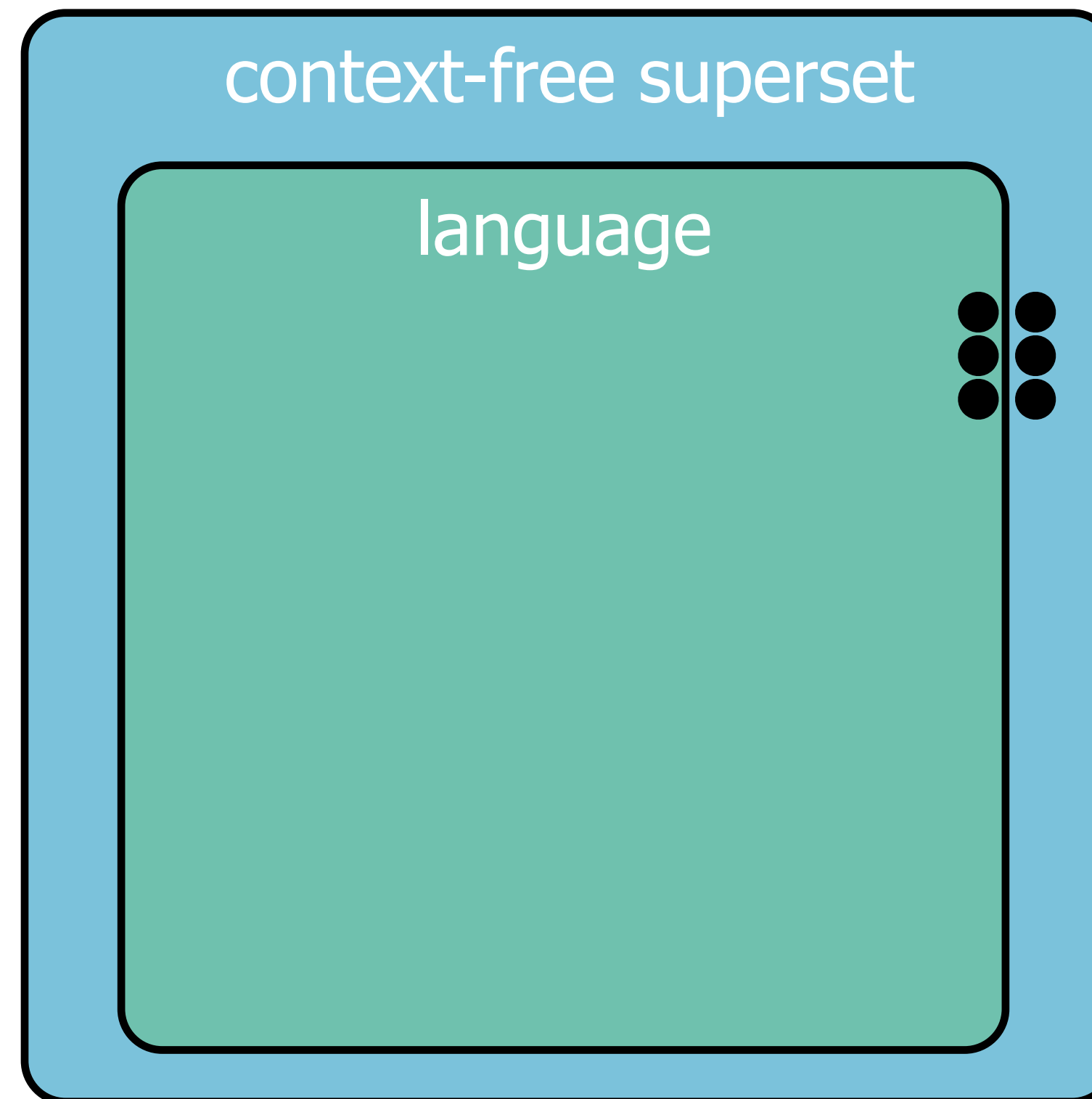


# Testing Errors

```
test undefined variable [[
  let type t = u
    type u = int
    var x: u := 0
  in
    x := 42 ;
    let type u = t
      var y: u := 0
    in
      y := [[z]]
    end
  end
]] 1 error
```

```
test type error [[
  let type t = u
    type u = string
    var x: u := 0
  in
    x := 42 ;
    let type u = t
      var y: u := 0
    in
      y := [[x]]
    end
  end
]] 1 error
```

# Test Corner Cases



# Static Analysis for Tiger

# Scope

```
let
  var x : int := 0 + z
  var y : int := x + 1
  var z : int := x + y + 1
in
  x + y + z
end
```

# Scope: Definition before Use

```
let
  var x : int := 0 + z // z not in scope
  var y : int := x + 1
  var z : int := x + y + 1
in
  x + y + z
end
```

# Mutual Recursion

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
in
  even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
in
  even(34)
end
```

# Mutually Recursive Functions should be Adjacent

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
in
  even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
in
  even(34)
end
```

# Name Spaces

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
in foo(56) + foo
end
```



# Functions and Variables in Same Name Space

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
in foo(56) + foo // both refer to the variable foo
end
```

Functions and variables are in the same namespace

# Type Dependent Name Resolution

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
in origin.x
end
```

# Type Dependent Name Resolution

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
in origin.x
end
```

Resolving origin.x requires the type of origin

# Name Correspondence

```
let
  type point = {x : int, y : int}
  type errpoint = {x : int, x : int}
  var p : point
  var e : errpoint
in
  p := point{ x = 3, y = 3, z = "a" }
  p := point{ x = 3 }
end
```

# Name Set Correspondence

Duplicate Declaration of Field “x”

```
let
  type point = {x : int, y : int}
  type errpoint = {x : int, x : int}
  var p : point
  var e : errpoint
in
  p := point{ x = 3, y = 3, z = "a" }
  p := point{ x = 3 }
end
```

Field “y” not initialized

Reference “z” not resolved

# Recursive Types

```
let
  type intlist = {hd : int, tl : intlist}
  type tree = {key : int, children : treelist}
  type treelist = {hd : tree, tl : treelist}
  var l : intlist
  var t : tree
  var tl : treelist
in
  l := intlist { hd = 3, tl = l };
  t := tree {
    key = 2,
    children = treelist {
      hd = tree{ key = 3, children = 3 },
      tl = treelist{ }
    }
  };
  t.children.hd.children := t.children
end
```

# Recursive Types

```
let
  type intlist = {hd : int, tl : intlist}
  type tree = {key : int, children : treelist}
  type treelist = {hd : tree, tl : treelist}
  var l : intlist
  var t : tree
  var tl : treelist
in
  l := intlist { hd = 3, tl = l };
  t := tree {
    key = 2,
    children = treelist {
      hd = tree{ key = 3, children = 3 },
      tl = treelist{ }
    }
  };
  t.children.hd.children := t.children
end
```

type mismatch

Field "tl" not initialized  
Field "hd" not initialized

Except where otherwise noted, this work is licensed under

