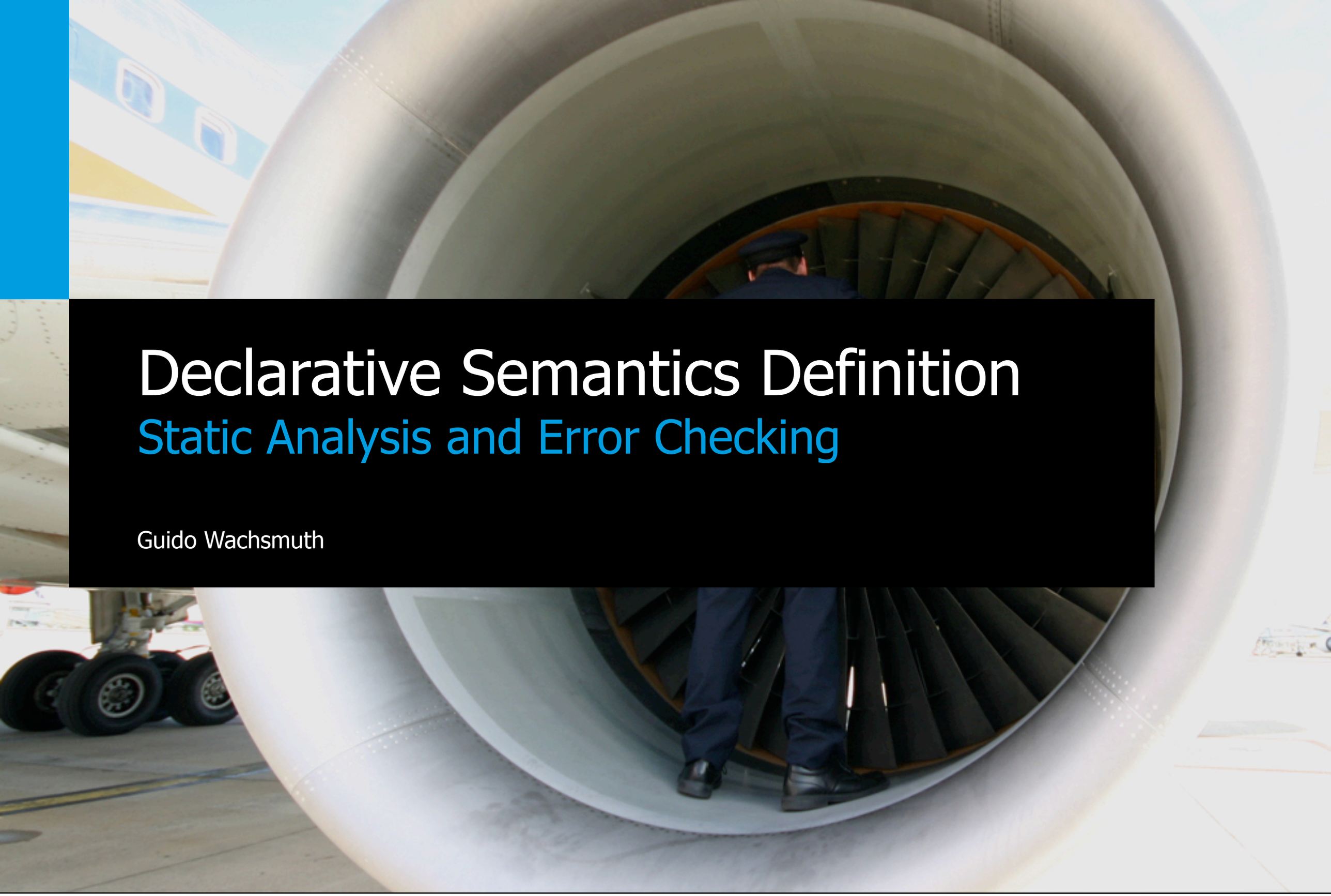


Declarative Semantics Definition

Static Analysis and Error Checking

Guido Wachsmuth

Overview
today's lecture

static analysis & error checking

- name analysis
- type analysis
- error checking
- error reporting

semantic editor services

- reference resolution
- hover help
- code completion

I

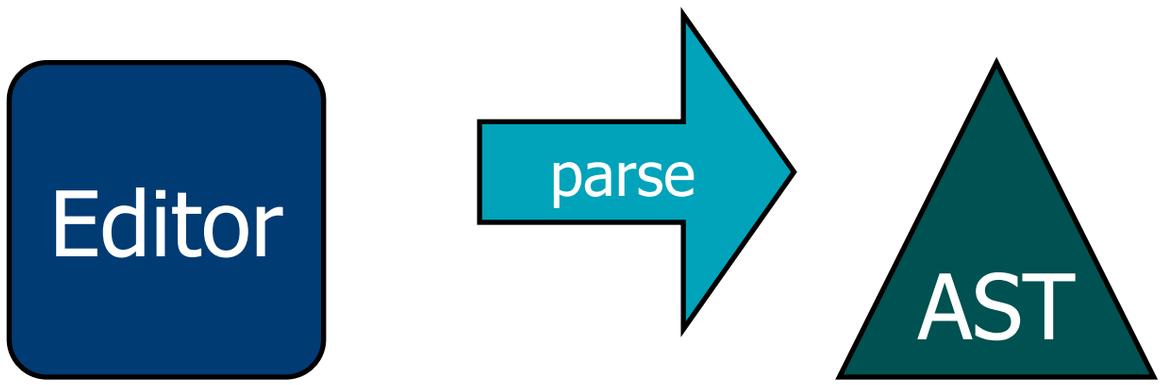
overview

Modern Compilers in IDEs

architecture

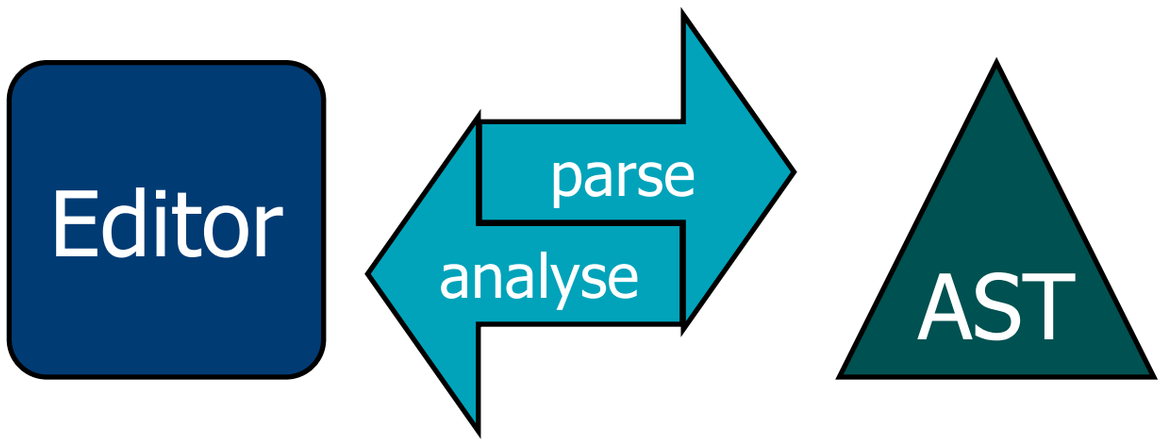
Modern Compilers in IDEs

architecture



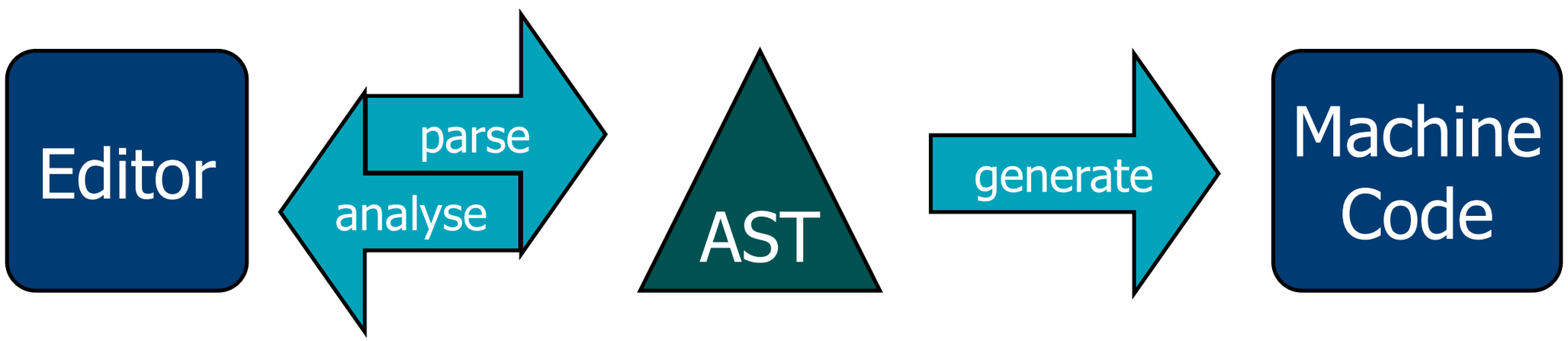
Modern Compilers in IDEs

architecture



Modern Compilers in IDEs

architecture



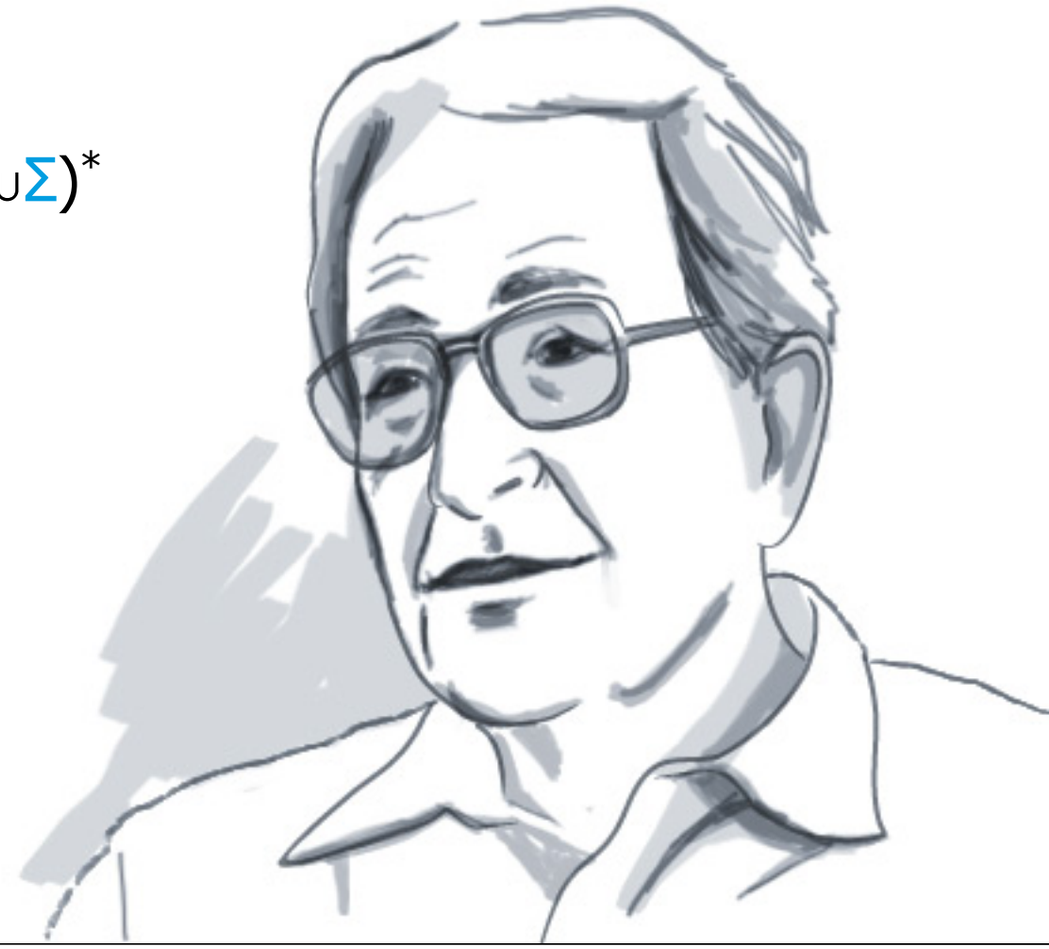
Recap: A Theory of Language
formal grammars

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

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* \; N \; (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$

grammar classes

- type-0, unrestricted
- type-1, context-sensitive: (a A c, a b c)
- type-2, context-free: $P \subseteq N \times (N \cup \Sigma)^*$
- type-3, regular: (A, x) or (A, xB)



Recap: Theoretical Computer Science

decidability & complexity

word problem χ_L: $\Sigma^* \rightarrow \{0,1\}$

- $w \rightarrow 1$, if $w \in L$
- $w \rightarrow 0$, else

decidability

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

complexity

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

$PSPACE \supseteq NP \supseteq P$

Recap: Language Definition
components

syntax definition

- concrete syntax
- abstract syntax

static analysis

- name resolution
- type analysis

semantics

- generation
- interpretation

Consistency Checking
ingredients

name analysis

- disambiguate names

Consistency Checking
ingredients

name analysis

- disambiguate names

reference resolving

- linking identifiers to declarations

Consistency Checking
ingredients

name analysis

- disambiguate names

reference resolving

- linking identifiers to declarations

type analysis

- computing types of expressions

Consistency Checking
ingredients

name analysis

- disambiguate names

reference resolving

- linking identifiers to declarations

type analysis

- computing types of expressions

error checking

- checking static constraints and reporting errors

Consistency Checking
ingredients

name analysis

- disambiguate names

reference resolving

- linking identifiers to declarations

type analysis

- computing types of expressions

error checking

- checking static constraints and reporting errors

editor interface

- collecting and displaying errors and warnings

Consistency Checking

generic approach

rename

- unique names

Consistency Checking
generic approach

rename

- unique names

map

- map identifiers to declarations

Consistency Checking

generic approach

rename

- unique names

map

- map identifiers to declarations

project

- compute properties of declarations and expressions

Consistency Checking

generic approach

rename

- unique names

map

- map identifiers to declarations

project

- compute properties of declarations and expressions

check

- check static constraints

II

name analysis

Name Binding and Scope

IDE components

static checking

editor services

transformation

refactoring

code generation

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      posterName = "name";
7      string posterName;
8      posterName = user.nam;
9      string posterName = user.name;
10     return posterName;
11   }
12 }
```

Name Binding and Scope

reference resolution

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

Name Binding and Scope

content completion

```
1  class Blog {
2    string post(User user, string message) {
3      string posterName;
4      posterName = user.;
5      return posterName;
6    }
7  }
8  class User {
9    string name;
10   string username;
11   string homepage;
12 }
```

homepage
name
username

```
1  class Blog {
2    string post(User user, string message) {
3      string posterName;
4      posterName = user.name;
5      return ;
6    }
7  }
8  class User
9    string na
10   string us
11   string ho
12 }
```

message
posterName
user

Bound Type Renaming

```
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
```

```
let
   type t0 = u0
   type u0 = int
   var x: u0 := 0
in
   x := 42 ;
   let
      type u1 = t0
      var y: u1 := 0
   in
      y := 42
   end
end
```

Bound Type Renaming

```
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
```

```
let
   type t0 = u0
   type u0 = int
   var x: u0 := 0
in
   x := 42 ;
   let
      type u1 = t0
      var y: u1 := 0
   in
      y := 42
   end
end
```

Bound Type Renaming

```
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
```

```
let
   type t0 = u0
   type u0 = int
   var x: u0 := 0
in
   x := 42 ;
   let
      type u1 = t0
      var y: u1 := 0
   in
      y := 42
   end
end
```

Bound Type Renaming

```
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
```

```
let
   type t0 = u0
   type u0 = int
   var x: u0 := 0
in
   x := 42 ;
   let
      type u1 = t0
      var y: u1 := 0
   in
      y := 42
   end
end
```

Bound Type Renaming

```
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
```

```
let
   type t0 = u0
   type u0 = int
   var x: u0 := 0
in
   x := 42 ;
   let
      type u1 = t0
      var y: u1 := 0
   in
      y := 42
   end
end
```

Bound Renaming

annotated terms

```
t{t1, ..., tn}
```

add additional information to a term but preserve its signature

Name Binding Language

concepts

defines

refers

namespaces

scopes

imports

Name Binding Language

definitions and references

```
TypeDec(t, _):
  defines Type t

Tid(t) :
  refers to Type t
```

```
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
```

Name Binding Language

unique and non-unique definitions

```
TypeDec(t, _):
  defines unique Type t

Tid(t) :
  refers to Type t
```

```
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
```

Name Binding Language

namespaces

```
namespaces
  Type Variable Function

TypeDec(t, _):
  defines unique Type t

FunDec(f, _, _):
  defines unique Function f
FunDec(f, _, _, _):
  defines unique Function f
Call(f, _) :
  refers to Function f

VarDec(v, _):
  defines unique Variable v
FArg(a, _):
  defines unique Variable a
Var(v):
   refers to Variable v
```

```
let
   type mt = int
   type rt = {f1: string, f2: int}
   type at = array of int

   var x       := 42
   var y: int := 42

   function p() = print("foo")
   function sqr(x: int): int = x*x
in
   …
end
```

Name Binding Language

scope

```
FunDec(f, _, _):
  defines unique Function f
  scopes Variable

FunDec(f, _, _, _):
  defines unique Function f
  scopes Variable

Let(_, _):
  scopes Type, Function, Variable

Seq(_):
  scopes 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 := 42
    end
end
```

Name Binding Language

scope

```
FunDec(f, _, _):
  defines unique Function f
  scopes Variable

FunDec(f, _, _, _):
  defines unique Function f
  scopes Variable

Let(_, _):
  scopes Type, Function, Variable

Seq(_):
  scopes 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 := 42
    end
end
```

Name Binding Language

scope

```
FunDec(f, _, _):
  defines unique Function f
  scopes Variable

FunDec(f, _, _, _):
  defines unique Function f
  scopes Variable

Let(_, _):
  scopes Type, Function, Variable

Seq(_):
  scopes 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 := 42
    end
end
```

Name Binding Language

scope

```
FunDec(f, _, _):
  defines unique Function f
  scopes Variable

FunDec(f, _, _, _):
  defines unique Function f
  scopes Variable

Let(_, _):
  scopes Type, Function, Variable

Seq(_):
  scopes 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 := 42
    end
end
```

Name Binding Language

scope

```
FunDec(f, _, _):
  defines unique Function f
  scopes Variable

FunDec(f, _, _, _):
  defines unique Function f
  scopes Variable

Let(_, _):
  scopes Type, Function, Variable

Seq(_):
  scopes 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 := 42
    end
end
```

Name Binding Language

definition context

```
For(v, start, end, body):
  defines Variable v in body
```

```
for x := 0 to 42 do x;
```

Name Binding Language in Spoofax

derivation of editor services

- error checking
- reference resolution
- code completion

multi-file analysis

parallel analysis

incremental analysis

III

constraint checking

Name Analysis

project

```
name-of : declaration -> name
```

map declarations to names

Name Analysis

projecting declarations to names

project

```
type-name : TypeDec(t, _)       -> t
var-name  : VarDec(x, _)        -> x
var-name  : VarDec(x, _, _)     -> x
var-name  : For(x, _, _, _)     -> x
var-name  : FArg(x, _)          -> x
fun-name  : FunDec(f, _, _)     -> f
fun-name  : FunDec(f, _, _, _) -> f
```

index library

```
index-is-resolved
index-is-unresolved
index-is-unique
index-is-nonunique
index-is-used
index-is-unused
```

Name Analysis

check

```
editor-error : expression -> ( pos, msg )
```

compute type of expression

Name Analysis

error checking

```
editor-error: e@Tid(t) -> (e, $[Type [t] is not declared.])
   where <index-is-unresolved> t
   where not ( <primitive-type> t )

editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x

editor-error: e@Call(f, _) -> (e, $[Function [f] is not declared.])
   where <index-is-unresolved> f
   where not ( <standard-lib-fun> f )

editor-error: d -> (d, $[Multiple declarations for [x].])
      where <var-name <+ fun-name> d => x
      where <index-is-nonunique> x

editor-error: d@TypeDec(t, _) -> (d, $[Multiple declarations for [t].])
      where <index-is-nonunique> t
```

Name Analysis

warnings

```
editor-warning: d -> (d, $[Type [t] is never used.])
   where <type-name> d => t
   where <index-is-unused> t



editor-warning: d -> (d, $[Variable [x] is never used.])
   where <var-name> d => x
   where not ( <?For(_, _, _, _)> d )
   where <index-is-unused> x



editor-warning: d -> (d, $[Function [f] is never called.])
   where <fun-name> d => f
   where <index-is-unused> f
```

Spoofax

collect errors and warnings

```
analyse = desugar-all

editor-analyse:
   (ast, path, project-path) -> (ast', errors, warnings, notes)
   with
      editor-init;
      <analyse> ast => ast';
      <collect-all(editor-error, conc)> ast'   => errors;
      <collect-all(editor-warning, conc)> ast' => warnings;
      <collect-all(fail, conc)> ast'           => notes
```

```
module Tiger-Builders

builders

   observer : editor-analyse
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

Spoofax

origin tracking

```
let var x := 21 in y * 2 end
```

```
Let([VarDec("x", Int("21"))], [Times(Var("y"), Int("2"))])
```

```
desugar: Times(e1, e2)  -> Bop(MUL(), e1, e2)
```

```
Let([VarDec("x", Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
Let([VarDec("x"{"…"}, Int("21"))], [Bop(MUL(), Var("y"), Int("2"))])
```

```
editor-error: e@Var(x) -> (e, $[Variable [x] is not declared.])
   where <index-is-unresolved> x
```

coffee break

IV

type analysis

Type Analysis

project

```
type-of : expression -> type
```

compute type of expression

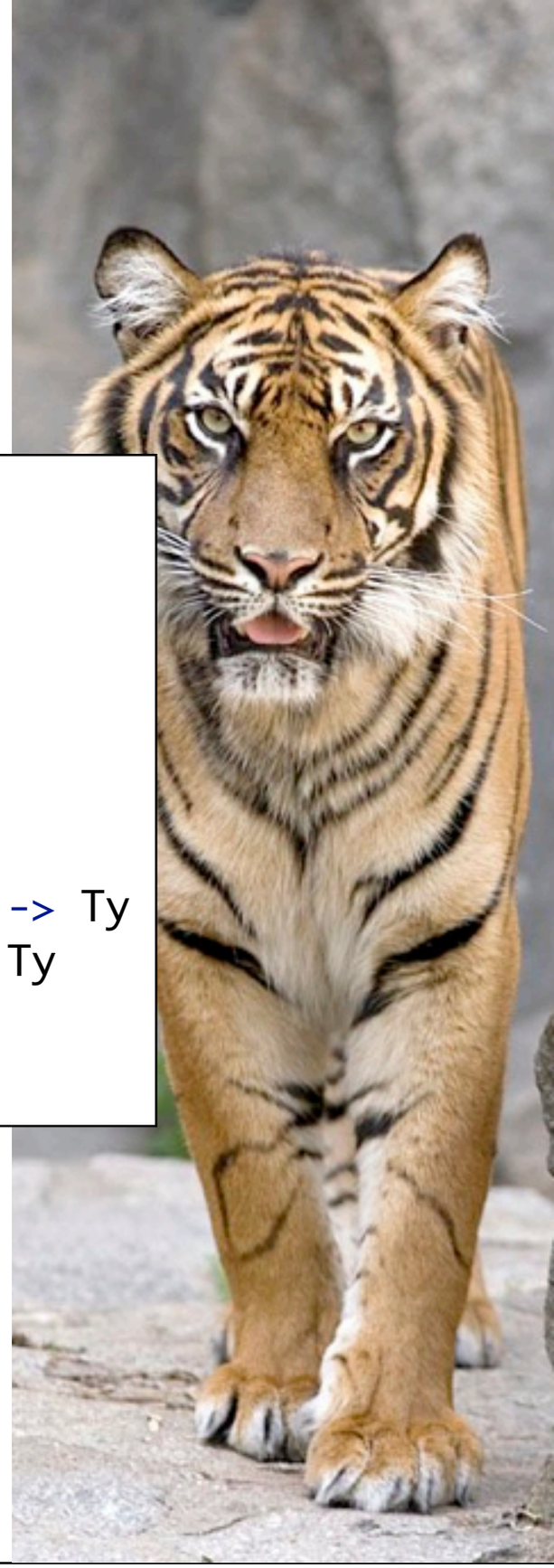
Expressions

```
type-of : Int(_)    -> INT()
type-of : String(_) -> STRING()

type-of : NilExp()  -> NIL()

type-of :
   Bop(_, e1, e2) -> INT()
   where
   <type-of> e1 => INT() ;
   <type-of> e2 => INT()

type-of : NoVal()   -> NO_VAL()
type-of : Seq(es)   -> <map(type-of) ; last> es
```

Expressions

```
type-of : Int(_)    -> INT()
type-of : String(_) -> STRING()

type-of : NilExp()  -> NIL()

type-of :
   Bop(_, e1, e2) -> INT()
   where
   <type-of> e1 => INT() ;
   <type-of> e2 => INT()

type-of : NoVal()   -> NO_VAL()
type-of : Seq(es)   -> <map(type-of) ; last> es
```

```
signature constructors

INT    : Ty
STRING : Ty
NO_VAL : Ty

NIL    : Ty
RECORD : TypeId * List(FTy) -> Ty
ARRAY  : TypeId * TypeId -> Ty

FIELD  : Id * TypeId -> FTy
```



Type Analysis

error checking

```
editor-error:
   Bop(_, e1, e2) -> errors
   where <filter(check-exp(eq))> [(e1, INT()), (e2, INT())] => errors

check-exp(eq):
   (e, t) -> (e, $[Expression is of type [<pp> t'] instead of type [<pp> t].])
   where <type-of> e => t'
   where not (<eq> (t, t'))
```

Relational Expressions

```
type-of : Rop(op, e1, e2) -> INT()
   where
      <type-of> e1 => t1 ;
      <type-of> e2 => t2 ;
      <check-rop> (op, t1, t2)

check-rop : (EQ(), t1, t2) -> <eq-or-nil> (t1, t2)
check-rop : (NE(), t1, t2) -> <eq-or-nil> (t1, t2)
check-rop : (LT(), t, t)   -> <int-or-string> t
check-rop : (LE(), t, t)   -> <int-or-string> t
```

Relational Expressions

```
type-of : Rop(op, e1, e2) -> INT()
   where
      <type-of> e1 => t1 ;
      <type-of> e2 => t2 ;
      <check-rop> (op, t1, t2)

check-rop : (EQ(), t1, t2) -> <eq-or-nil> (t1, t2)
check-rop : (NE(), t1, t2) -> <eq-or-nil> (t1, t2)
check-rop : (LT(), t, t)   -> <int-or-string> t
check-rop : (LE(), t, t)   -> <int-or-string> t
```

```
int-or-string = ?INT() <+ ?STRING()
noval-or-nil = ?NO_VAL() <+ ?NIL()

eq-or-nil : (t, t) -> t where not ( <noval-or-nil> t )

eq-or-nil : (r@RECORD(_, _), NIL()) -> r
eq-or-nil : (a@ARRAY(_, _),  NIL()) -> a
eq-or-nil : (NIL(), r@RECORD(_, _)) -> r
eq-or-nil : (NIL(), a@ARRAY(_, _))  -> a
```

Type Analysis

error checking

```
editor-error: Rop(op, e1, e2) -> error
   where <elem> (op, [EQ(), NE()])
   where <type-of> e1 => t
   where <check-exp(eq-or-nil)> (e2, t) => error

editor-error:
   Rop(op, e1, e2) -> (e1, $[Expression is of type [<pp> t] instead of type
                             [<pp> INT()] or [<pp> STRING()].])
   where <elem> (op, [LT(), LE()])
   where <type-of> e1 => t
   where not (<int-or-string> t)

editor-error: Rop(op, e1, e2) -> error
   where <elem> (op, [LT(), LE()])
   where <int-or-string> <type-of> e1 => t
   where <check-exp(eq)> (e2, t) => error
```

Record and Array Access

```
type-of :
   FieldVar(e, f) -> t
   where
      <type-of> e => RECORD(_, fields) ;
      <fetch-elem(?FIELD(f, tn))> fields ;
      <type-of> tn => t

type-of :
   Subscript(e1, e2) -> t
   where
      <type-of> e1 => ARRAY(_, tn) ;
      <type-of> tn => t ;
      <type-of> e2 => INT()
```

Type Analysis

```
editor-error:
   FieldVar(e, f) -> (e, $[Expression is of type [<pp> t] (no record type).])
   where <type-of> e => t
   where not ( <?RECORD(_, _)> t )

editor-error:
   FieldVar(e, f) -> (f, $[Field [f] not declared by type [<pp> t].])
   where <type-of> e => t@RECORD(_, fs)
   where not ( <fetch-elem(?FIELD(f, _))> fs )


editor-error:
   Subscript(e1, e2) -> (e1, $[Expression is of type [<pp> t] (no array type).])
   where <type-of> e1 => t
   where not ( <?ARRAY(_, _)> t )

editor-error:
   Subscript(e1, e2) -> error
   where <type-of> e1 => ARRAY(_, _)
   where <check-exp(eq)> (e2, INT()) => error
```

Record and Array Initialising

```
type-of :
   Record(tn, inits) -> t
   where
      <type-of> tn => t@RECORD(tn, fields) ;
      <zip(check-field)> (inits, fields)

check-field :
   (InitField(fn, e), FIELD(fn, tn)) -> t
   where
      <type-of> tn => t1 ; <type-of> e => t2 ;
      <eq-or-nil> (t1, t2) => t

type-of :
   Array(tn, e1, e2) -> t
   where
      <type-of> tn => t@ARRAY(tn, vtn) ;
      <type-of> vtn => vt1 ;
      <type-of> e1 => INT() ;
      <type-of> e2 => vt2 ;
      <eq-or-nil> (vt1, vt2)
```

Type Analysis

```
editor-error:
   e@Record(t, fs) -> (t, $[Type [<pp> t] is not a record type.])
   where <type-of> t => t'
   where not ( <?RECORD(_, _)> t' )

editor-error: Record(t, fs) -> errors
   where <type-of> t => RECORD(_, fdefs)
   where <filter(editor-error)> <zip> (fs, fdefs) => errors

editor-error:
   (InitField(f, e), FIELD(g, t)) -> (f, $[Field [g] expected.])
   where not ( <eq> (f, g) )

editor-error: (InitField(f, e), FIELD(g, t)) -> error
   where <type-of> t => t'
   where <filter(check-exp(eq-or-nil))> (e, t') => error
```

Type Analysis

```
editor-error:
   Array(t, e1, e2) -> (t, $[Type [<pp> t] is not an array type.])
   where <type-of> t => t'
   where not ( <?ARRAY(_, _)> t' )

editor-error:
   Array(t, e1, e2) -> errors
   where <type-of> t => ARRAY(_, tn)
   where <type-of> tn => t2
   where <filter(check-exp(eq-or-nil))> [(e1, INT()), (e2, t2)] => errors
```

Statements

```
type-of :
   Assign(l, e) -> NO_VAL()
   where
      <type-of> l => t1 ;
      <type-of> e => t2 ;
      <eq-or-nil> (t1, t2)

type-of :
   IfThenElse(e1, e2, e3) -> t
   where
      <type-of> e1 => INT() ;
      <type-of> e2 => t1 ;
      <type-of> e3 => t2 ;
      <eq-or-nil <+ noval-noval> (t1, t2) => t

type-of :
   While(e1, e2) -> NO_VAL()
   where
      <type-of> e1 => INT() ;
      <type-of> e2 => NO_VAL()
```

More Statements

```
type-of :
   For(x, e1, e2, e3) -> NO_VAL()
   where
      <type-of> e1 => INT() ;
      <type-of> e2 => INT() ;
      <type-of> e3 => NO_VAL()

type-of :
   Break() -> NO_VAL()

type-of :
   Let(_, []) -> NO_VAL()

type-of :
   Let(_, es@[_|_]) -> <map(type-of) ; last> es
```

Type Analysis

```
editor-error: Assign(e1, e2) -> error
   where <type-of> e1 => t
   where <check-exp(eq-or-nil)> (e2, t) => error

editor-error: IfThenElse(e1, e2, e3) -> errors
   where <type-of> e2 => t
   where <filter(check-exp(eq-or-nil <+ noval-noval))>
            [(e1, INT()), (e3, t)] => errors

editor-error: While(e1, e2) -> errors
   where <filter(check-exp(eq))> [(e1, INT()), (e2, NO_VAL())] => errors

editor-error: For(v, e1, e2, e3) -> errors
   where <filter(check-exp(eq))>
            [(e1, INT()), (e2, INT()), (e3, NO_VAL())] => errors
```

Name Binding Language

interaction with type system

```
FArg(a, t):
  defines unique Variable a of type t

FunDec(f, a*, t, _):
  defines unique Function f of type (t*, t)
  where a* has type t*

FunDec(f, a*, e):
  defines unique Function f of type (t*, NoVal())
  where a* has type t*

Call(f, a*) :
  refers to Function f of type (t*, _)
  where a* has type t*
```

Variables and Function Calls

```
type-of :
   Var(x) -> <index-type-of> x

type-of :
   Call(f, es) -> t
   where
      <index-type-of> f => (ts, t) <+
      <standard-lib-fun> f => (ts, t) ;
      <zip(eq-or-nil)> (ts, <map(type-of)> es)
```

Type Analysis

```
editor-error: VarDec(_, t, e) -> error
   where <type-of> t => t'
   where <check-exp(eq-or-nil)> (e, t') => error

editor-error: VarDec(_, e) ->(e, $[Expression is of type [<pp> t].])
   where <type-of> e => t
   where <elem> (t, [ NO_VAL(), NIL() ])

editor-error: e@Call(f, es) -> (e, "Wrong number of arguments.")
   where <index-type-of <+ stdlib-fun> f => (ts, t)
   where not( <eq> (<length> es, <length> ts) )

editor-error: Call(f, es) -> errors
   where <index-type-of <+ stdlib-fun> f => (ts, t)
   where <zip(check-exp(eq-or-nil))> (es, ts) => errors
```

V

semantic editor services

Semantic Editor Services

reference resolution

```
editor-resolve:
    (node, position, ast, path, project-path) -> target
    where
      index-setup(|<language>, [project-path], $[[project-path]/[path]]);
      index-transaction(
        target := <index-lookup> node
      )
```

```
module Tiger-References

references

   reference _ : editor-resolve
```

Semantic Editor Services

hover help

```
editor-hover:
   (node, position, ast, path, project-path) -> $[type: [<pp> t]]
   where
      <type-of> node => t

pp: NIL()     -> $["nil"]          pp: Tid(i)    -> $["[i]"]
pp: NO_VAL() -> $["no-val"]

pp: INT()     -> $["int"]          pp: RECORD(Tid(i), _) -> $["record [i]"]
pp: STRING() -> $["string"]        pp: ARRAY(_, Tid(i))  -> $["array [i]"]
```

```
module Tiger-References

references

   hover _ : editor-hover
```

Semantic Editor Services

code completion

```
editor-complete:
    (node, position, ast, path, project-path) -> proposals'
    where
      editor-init;
      (ast', _) := <analyze-top(|<language>)> (ast, path, project-path);
      item@COMPLETION(name) := <collect-one(?COMPLETION(_))> ast';
      index-transaction(
        (<index-lookup-all(|name)> item <+ ![]) => proposals
      );
      proposals' := <map(index-uri; index-uri-name)> proposals
```

```
module Tiger-Completions

completions

   completions proposer : editor-complete
```

VI

second milestone

Assignment: Static Analysis
schedule

Lab 1

- import MiniJava project
- write test cases

Lab 2

- desugar expressions
- analyse names

Lab 3

- analyse types
- check constraints

Assignment: Static Analysis
requirements

duplicate class, field, method, variable names

missing class, field, method, variable declarations

cyclic inheritance

unused classes, fields, methods, variables

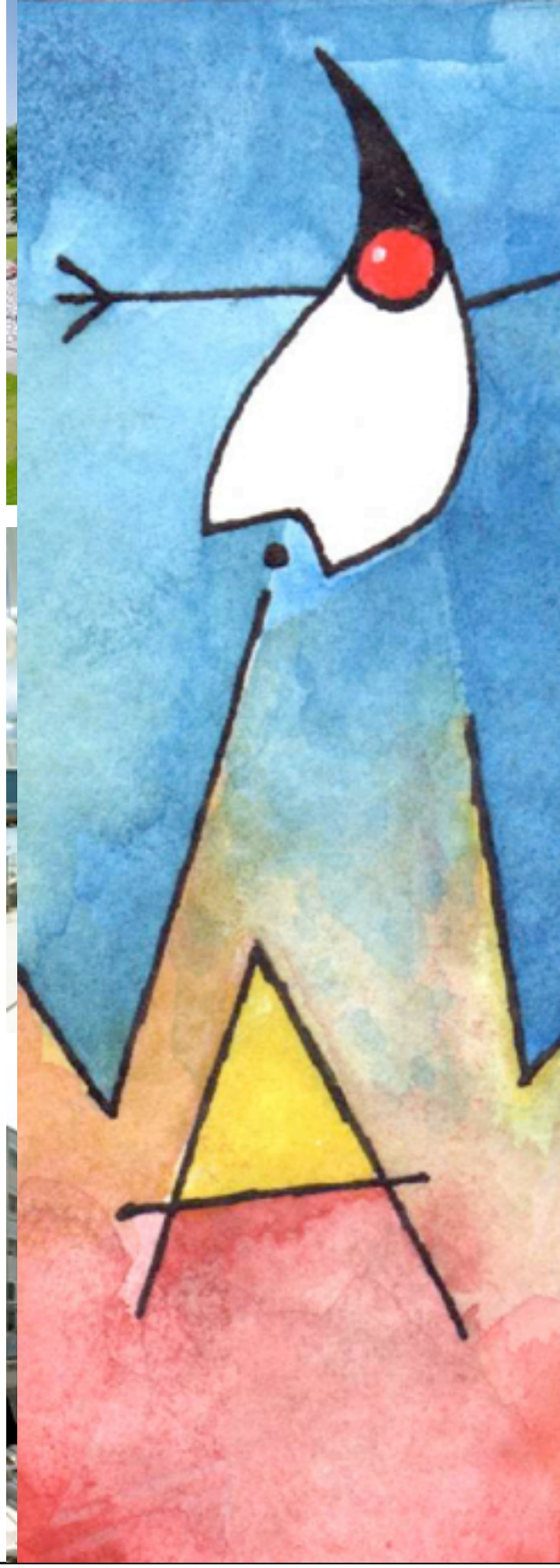
subclassing main class

instantiating main class

overriding field declarations

overriding method declarations

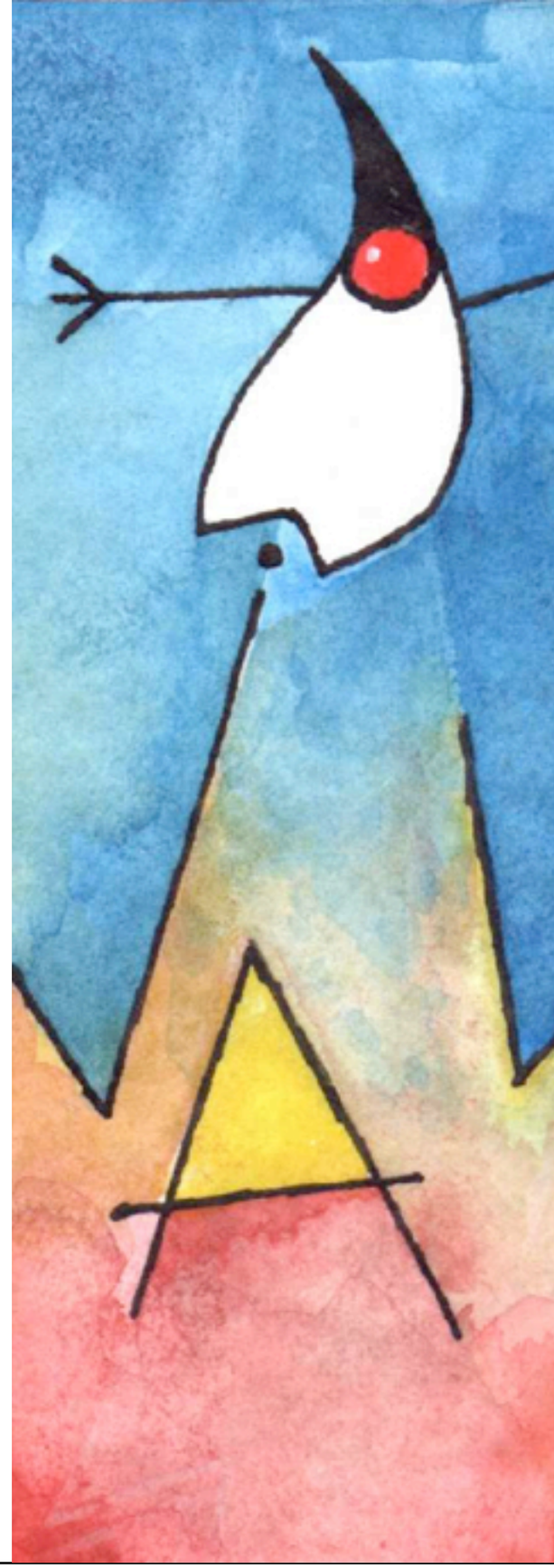
overloading method declarations



Assignment: Name Analysis
requirements

editor services

- reference resolution
  - test cases
  - classes, fields, parameters, local variables
- content completion
  - parent class, class types, object creation, field & variable references
  - trigger
- hover help
  - types, definition sites



VII

summary

Summary
lessons learned

Summary
lessons learned

How do you define static analysis and error checking in Stratego?

- name analysis with NBL
- index map names to properties
- project declarations and expressions to properties
- check static constraints
- report errors

Summary
lessons learned

How do you define static analysis and error checking in Stratego?

- name analysis with NBL
- index map names to properties
- project declarations and expressions to properties
- check static constraints
- report errors

How do you define semantic editor services in Spoofax?

- reference resolution
- hover help
- content completion

Literature

learn more

Literature

learn more

Spoofax

Lennart C. L. Kats, Eelco Visser: The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. OOPSLA 2010

http://www.spoofax.org

Literature

learn more

Spoofax

Lennart C. L. Kats, Eelco Visser: The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. OOPSLA 2010

http://www.spoofax.org

Name Binding Language

Gabriël Konat, Lennart Kats, Guido Wachsmuth, Eelco Visser: Language-Parametric Name Resolution Based on Declarative Name Binding and Scope Rules. SLE 2012

Outlook
coming next

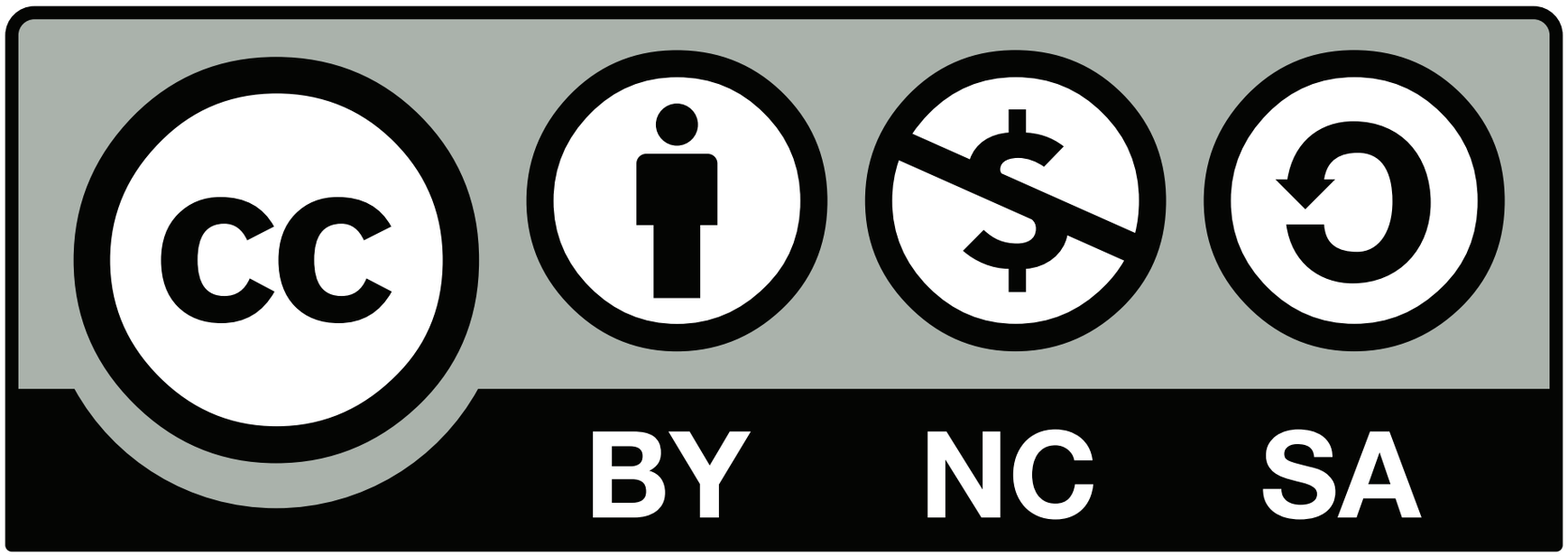
declarative semantics definition

- Code Generation

Lab Oct 11

- import MiniJava project
- explore ASTs
- write test cases

copyrights


BY NC SA

Pictures

attribution & copyrights

Slides 1, 8, 9: Inspection by Fly For Fun, some rights reserved

Slide 5: Noam Chomsky by Fellowsisters, some rights reserved

Slide 30: Kenco by Dominica Williamson, some rights reserved

Slides 33, 35, 37, 39, 42, 43: Tiger by Bernard Landgraf, some rights reserved

Slide 52: Students, TU Delft, Media Solutions, all rights reserved

Slides 53-56: Italian Java book cover by unknown artist, some rights reserved

Slide 57: Envelopes by benchilada, some rights reserved

Slide 58: Report card by Carosaurus, some rights reserved

Slide 62: Portal BUGA 2009 by Harald Thiele, some rights reserved