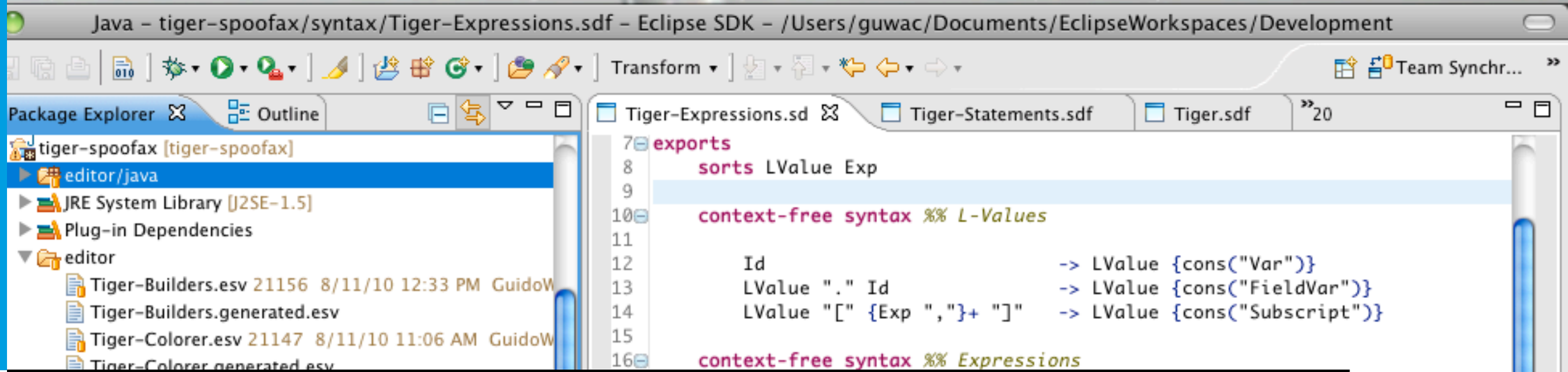
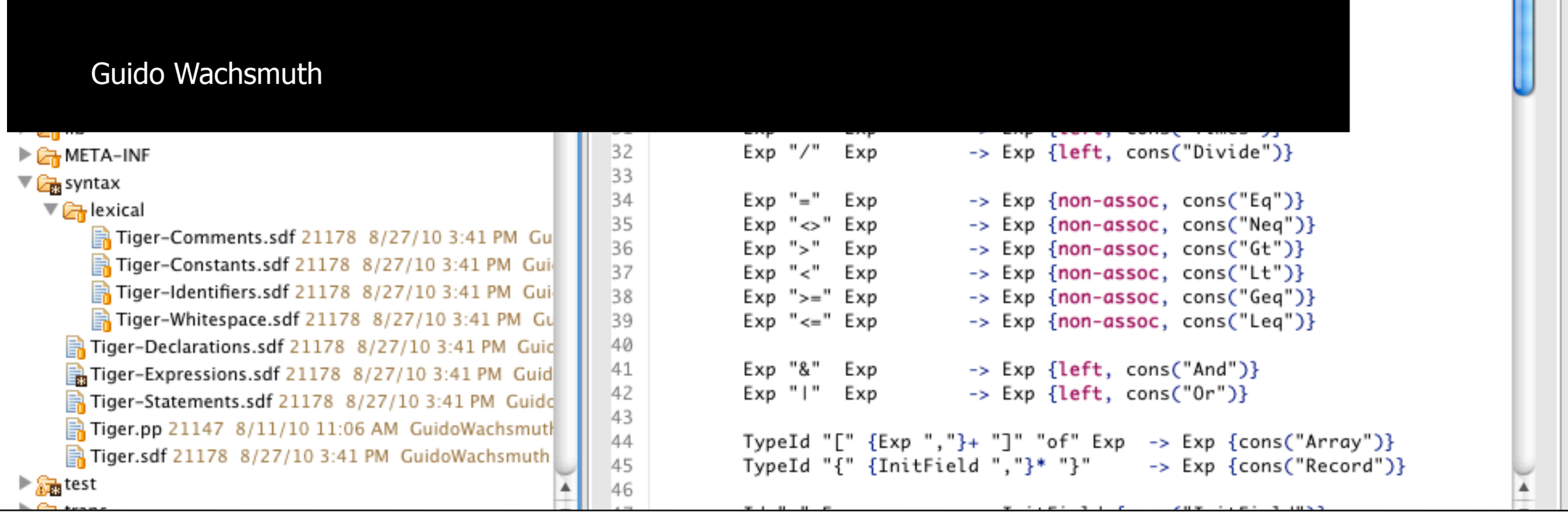


Declarative Syntax Definition

SDF and ATerms

Guido Wachsmuth



Overview
today's lecture

syntax definition formalism SDF

- lexical and context-free syntax definition
- abstract syntax definition
- disambiguation

ATerms

- trees as terms
- signatures

Spoofax

- syntactical editor services

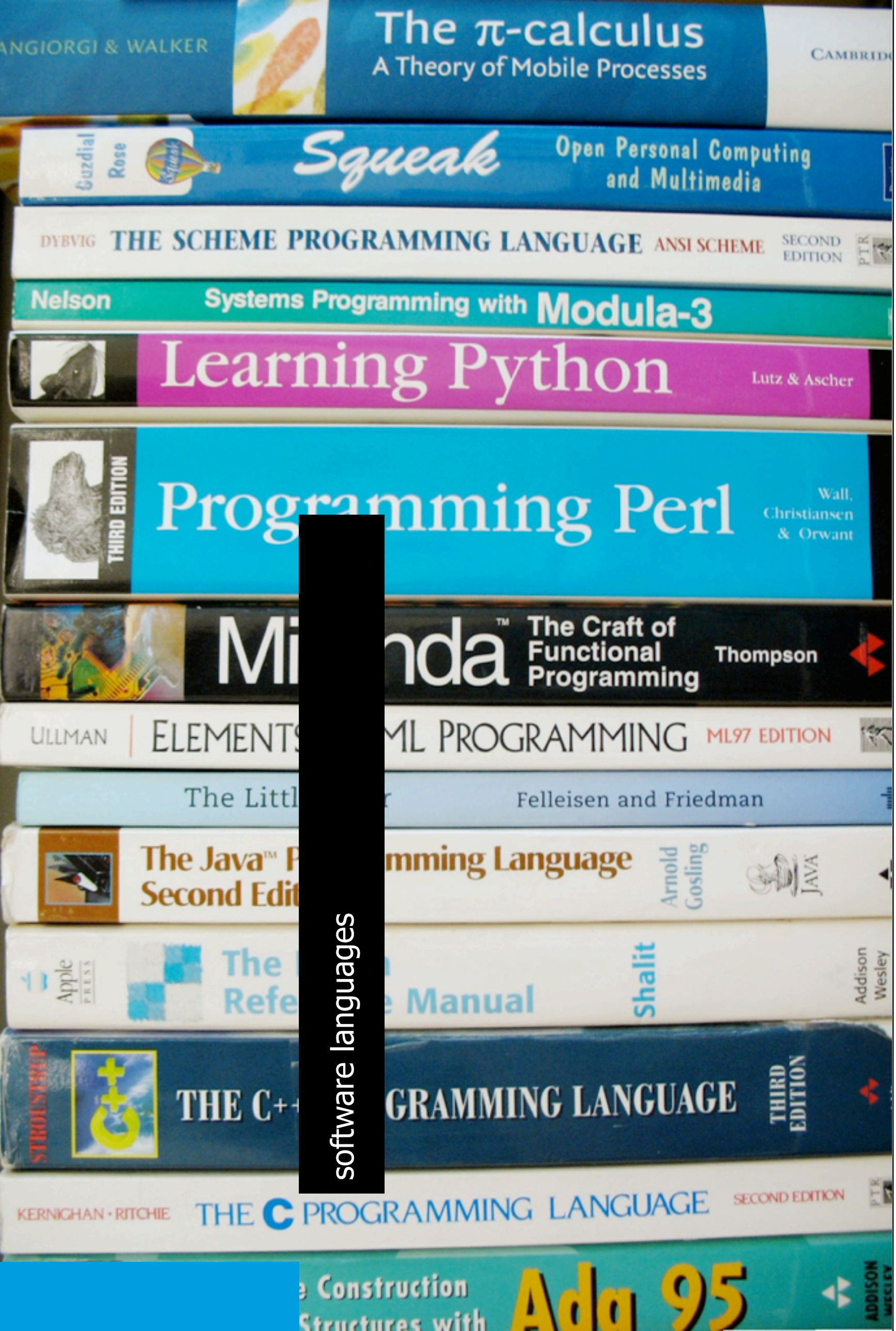
Recap: Tiger

the lecture language

```
/* factorial function */

let

   var x := 0

   function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))

in

   for i := 1 to 3 do (
      x := x + fact(i);
      printint(x);
      print(" ")
   )

end
```

I

syntax definition



software languages

LA DEUXIÈME ANNÉE

DE LATIN

declarative models

Théorie 250 pages.

227 Exercices 100 pages.

Lexiques

24 pages blanches pour notes et 4 cartes

Syntax Definition

software languages

lexical syntax

- words made from letters
- structure not relevant
- regular expressions

context-free syntax

- sentences made from words
- structure relevant
- context-free grammars
- Backus-Naur Form
- Extended Backus-Naur Form

Regular Expressions

overview

basics

- strings: `"nil"`
- character classes: `[a-zA-Z]`

combinators

- concatenation: `E1  E2`
- optional: `E?`
- zero or more: `E*`
- one or more: `E+`
- alternative: `E1 | E2`

Tiger
lexical syntax

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

```
[a-zA-Z][a-zA-Z0-9\_]*
```

Tiger
lexical syntax

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

```
[a-zA-Z][a-zA-Z0-9\_]*
```

A comment may appear between any two tokens. Comments start with /* and end with */ and may be nested.

Tiger
lexical syntax

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

```
[a-zA-Z][a-zA-Z0-9\_]*
```

A comment may appear between any two tokens. Comments start with /* and end with */ and may be nested.

Pumping Lemma

Backus-Naur Form

overview

basics

- symbols: <s>
- strings: `nil`

combinators

- concatenation
- productions: <s> ::= E1 | … | En

Binary Expressions

Backus-Naur Form

```
<exp> ::= <num>
        | <exp> + <exp>
        | <exp> - <exp>
        | <exp> * <exp>
        | <exp> / <exp>
        | ( <exp> )
```

Extended Backus-Naur Form

overview

basics

- strings: `"nil"`
- symbols: `s`

combinators

- concatenation: `E1,E2`
- optional: `[E]`
- zero or more: `{E}`
- productions: `s = E1 | … | En ;`

Binary Expressions

Extended Backus-Naur Form

```
exp = num
    | exp , "+" , exp
    | exp , "-" , exp
    | exp , "*" , exp
    | exp , "/" , exp
    | "(" , exp , ")" ;
```

II

Spoofax

Modern Compilers in IDEs

features

syntactic editor services

- syntax checking
- syntax highlighting
- outline view
- code folding
- bracket matching

semantic editor services

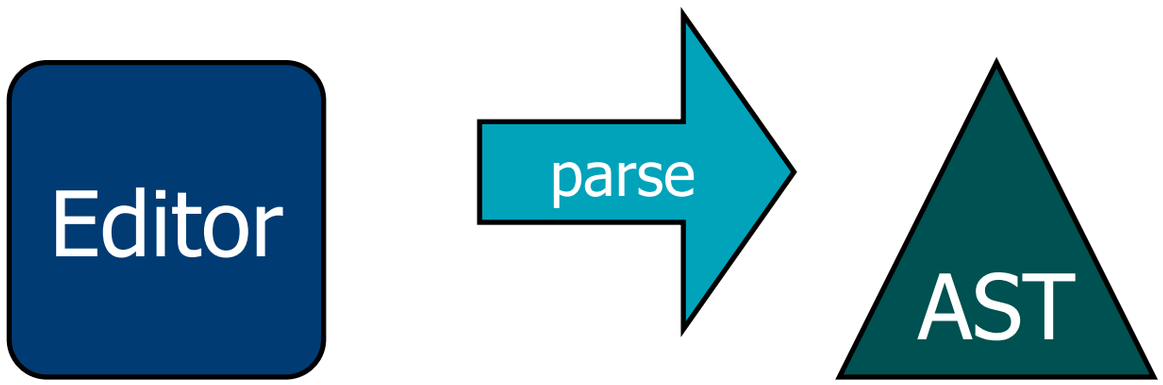
- error checking
- reference resolving
- hover help
- code completion
- refactoring

Modern Compilers in IDEs

architecture

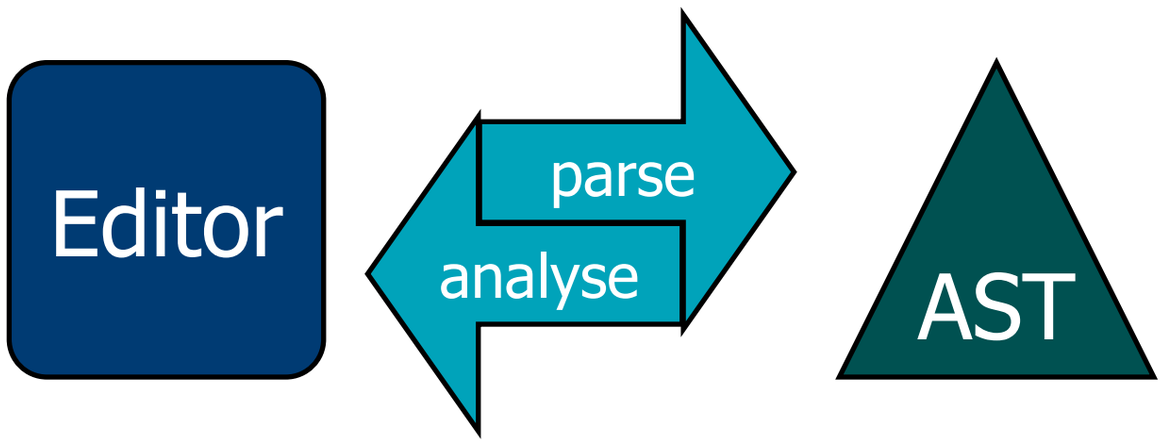
Modern Compilers in IDEs

architecture



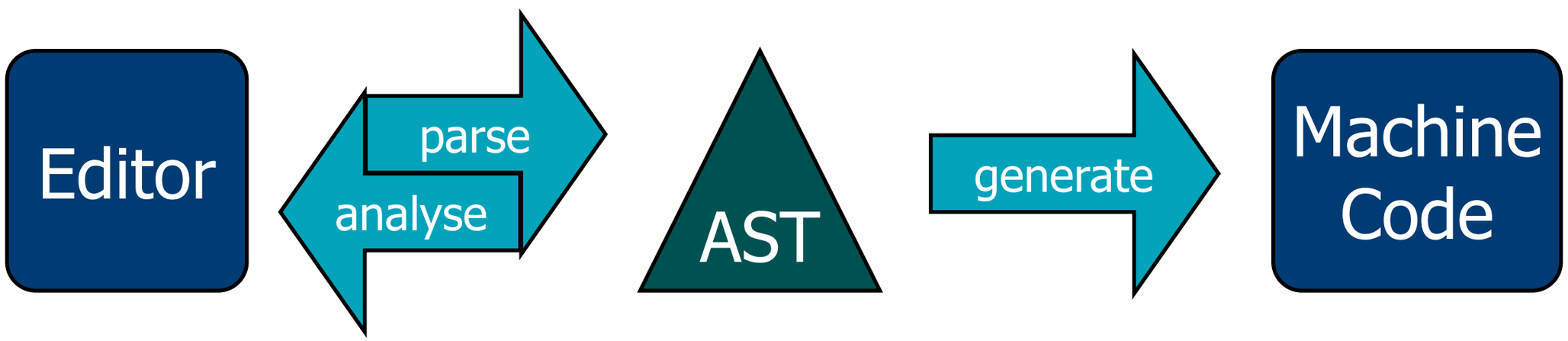
Modern Compilers in IDEs

architecture



Modern Compilers in IDEs

architecture



Language Definition
components

syntax definition

- concrete syntax
- abstract syntax

static analysis

- name resolution
- type analysis

semantics

- generation
- interpretation

Language Workbenches

architecture

Spoofax

the lecture & lab language workbench

based on IMP framework by IBM Research

syntax definition

- declarative meta-language
- modular syntax definitions

SDF

semantics

- declarative transformation language
- operating on ASTs
- rewrite rules, strategies, dynamic rules

ATerms

editor services

- declarative configuration languages

Tiger

```
module Tiger

imports

   lexical/Tiger-Whitespace
   lexical/Tiger-Comments
   Tiger-Expressions
   Tiger-Statements
   Tiger-Declarations

exports

   context-free start-symbols

      Start

   context-free syntax

      Exp -> Start
```

III

lexical syntax

Identifiers

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax

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

lexical restrictions

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

lexical syntax

   "nil" -> Id {reject}
```

Identifiers

characters not separated by layout

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax

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

   lexical restrictions

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

   lexical syntax

      "nil" -> Id {reject}
```

Identifiers

characters not separated by layout

regular expression

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax

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

   lexical restrictions

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

   lexical syntax

      "nil" -> Id {reject}
```

Identifiers

characters not separated by layout

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax
      [a-zA-Z][a-zA-Z0-9\_]* -> Id

lexical restrictions

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

lexical syntax

   "nil" -> Id {reject}
```

regular expression

grammar rule

Identifiers

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax
      [a-zA-Z][a-zA-Z0-9\_]* -> Id

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

   lexical syntax
      "nil" -> Id {reject}
```

characters not separated by layout

regular expression

grammar rule

follow restriction

Identifiers

```
module Tiger-Identifiers

exports
   sorts Id

   lexical syntax
      [a-zA-Z][a-zA-Z0-9\_]* -> Id

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

   lexical syntax
      "nil" -> Id {reject}
```

characters not separated by layout

regular expression

grammar rule

follow restriction

reserved word

Constants

```
module Tiger-Constants

exports
   sorts IntConst StringConst

   lexical syntax
      [0-9]+                          -> IntConst

      "\"" StrChar* "\""              -> StrConst
      ~[\\\"\n]                       -> StrChar
      [\\] [n]                        -> StrChar
      [\\] [t]                        -> StrChar
      [\\] [\^] [A-Z]                 -> StrChar
      [\\] [0-9][0-9][0-9]            -> StrChar
      [\\] [\"]                       -> StrChar
      [\\] [\\]                       -> StrChar
      [\\] [\ \t\n]+ [\\]             -> StrChar

lexical restrictions
   IntConst -/- [0-9]
```

Constants

```
module Tiger-Constants

exports
   sorts IntConst StringConst

   lexical syntax
      [0-9]+                          -> IntConst

      "\"" StrChar* "\""              -> StrConst
      ~[\\\"\n]                       -> StrChar
      [\\] [n]                        -> StrChar
      [\\] [t]                        -> StrChar
      [\\] [\^] [A-Z]                 -> StrChar
      [\\] [0-9][0-9][0-9]            -> StrChar
      [\\] [\"]                       -> StrChar
      [\\] [\\]                       -> StrChar
      [\\] [\ \t\n]+ [\\]             -> StrChar

lexical restrictions
   IntConst -/- [0-9]
```

context-free rule

Constants

```
module Tiger-Constants

exports
   sorts IntConst StringConst

   lexical syntax
      [0-9]+                          -> IntConst

      "\"" StrChar* "\""              -> StrConst
      ~[\\\"\n]                       -> StrChar
      [\\] [n]                        -> StrChar
      [\\] [t]                        -> StrChar
      [\\] [\^] [A-Z]                 -> StrChar
      [\\] [0-9][0-9][0-9]            -> StrChar
      [\\] [\"]                       -> StrChar
      [\\] [\\]                       -> StrChar
      [\\] [\ \t\n]+ [\\]             -> StrChar

   lexical restrictions
      IntConst -/- [0-9]
```

context-free rule

complement

Whitespace

```
module Tiger-Whitespace

exports
   lexical syntax

      [\ \t\n\r] -> LAYOUT

   context-free restrictions

      LAYOUT? -/- [\ \t\n\r]
```

Whitespace

```
module Tiger-Whitespace

exports
   lexical syntax

      [\ \t\n\r] -> LAYOUT

   context-free restrictions

      LAYOUT? -/- [\ \t\n\r]
```

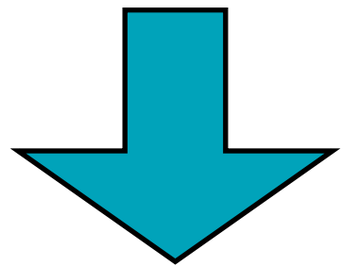
inserted between symbols in CF rules

Whitespace

```
module Tiger-Whitespace

exports
   lexical syntax

      [\ \t\n\r] -> LAYOUT

   context-free restrictions

      LAYOUT? -/- [\ \t\n\r]
```

inserted between symbols in CF rules

```
Exp "+" Exp -> Exp
```



```
<Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>
```

Comments

```
module Tiger-Comments

exports
   lexical syntax
      Comment                          -> LAYOUT
      "/*" CommentPart* "*/"           -> Comment
      ~[\*]                            -> CommentPart
      [\*]                             -> CommentPart
```

Comments

```
module Tiger-Comments

exports
   lexical syntax
      Comment                        -> LAYOUT
      "/*" CommentPart* "*/"         -> Comment
      ~[\*]                          -> CommentPart
      [\*]                           -> CommentPart

   lexical restrictions
      [\*] -/- [\/]
```

Comments

```
module Tiger-Comments

exports
   lexical syntax
      Comment                        -> LAYOUT
      "/*" CommentPart* "*/"         -> Comment
      ~[\*]                          -> CommentPart
      Asterisk                       -> CommentPart

      [\*]                           -> Asterisk

   lexical restrictions
      Asterisk -/- [\/]
```

Comments

```
module Tiger-Comments

exports
   lexical syntax
      Comment                        -> LAYOUT
      "/*" CommentPart* "*/"         -> Comment
      ~[\*]                          -> CommentPart
      Asterisk                       -> CommentPart

      [\*]                           -> Asterisk

   lexical restrictions
      Asterisk -/- [\/]

   context-free restrictions
      LAYOUT? -/- [\/].[\*]
```

IV

context-free syntax

Expressions

```
sorts LValue Exp

context-free syntax
   Id                    -> LValue
   LValue "." Id         -> LValue
   LValue "[" Exp "]"    -> LValue

   LValue                -> Exp
   "nil"                 -> Exp
   IntConst              -> Exp
   StrConst              -> Exp

   Id "(" {Exp ","}* ")" -> Exp

   TypeId "{" {InitField ","}* "}" -> Exp
   TypeId "[" Exp "]" "of" Exp     -> Exp

   Id "=" Exp            -> InitField
```

```
x
x.f1
x[i]

nil
42
"foo"

sqr(x)

at [x] of 42
rt {f1 = "foo", f2 = 42}
```

Expressions

```
sorts LValue Exp

context-free syntax
   Id                    -> LValue
   LValue "." Id         -> LValue
   LValue "[" Exp "]"    -> LValue

   LValue                -> Exp
   "nil"                 -> Exp
   IntConst              -> Exp
   StrConst              -> Exp

   Id "(" {Exp ","}* ")" -> Exp

   TypeId "{" {InitField ","}* "}" -> Exp
   TypeId "[" Exp "]" "of" Exp     -> Exp

   Id "=" Exp            -> InitField
```

repetition with separator

```
x
x.f1
x[i]

nil
42
"foo"

sqr(x)

at [x] of 42
rt {f1 = "foo", f2 = 42}
```

More Expressions

```
context-free syntax

   "-" Exp       -> Exp

   Exp "+"  Exp -> Exp
   Exp "-"  Exp -> Exp
   Exp "*"  Exp -> Exp
   Exp "/"  Exp -> Exp

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

   Exp "&"  Exp -> Exp
   Exp "|"  Exp -> Exp
```

Statements

```
sorts Exp

context-free syntax

   LValue ":=" Exp                          -> Exp

   "let" Dec* "in" {Exp ";"}* "end"         -> Exp

   "if" Exp "then" Exp                      -> Exp
   "if" Exp "then" Exp "else" Exp           -> Exp

   "while" Exp "do" Exp                     -> Exp
   "for" Id ":=" Exp "to" Exp "do" Exp      -> Exp

   "break"                                  -> Exp

   "(" ")"                 -> Exp
   "(" Exp ")"             -> Exp
   "(" {Exp ";"}+ ")"      -> Exp
```

Declarations

```
sorts Dec TypeId

context-free syntax
   "type" Id "=" Type    -> Dec

   TypeId                -> Type
   "{" {Field ","}* "}"  -> Type
   "array" "of" TypeId   -> Type

   Id                    -> TypeId

   Id ":" TypeId         -> Field

   "var" Id            ":=" Exp -> Dec
   "var" Id ":" TypeId ":=" Exp -> Dec

   "function" Id "(" {FArg ","}* ")"            "=" Exp -> Dec
   "function" Id "(" {FArg ","}* ")" ":" TypeId "=" Exp -> Dec

   Id ":" TypeId                                -> FArg
```

```
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
```

metalanguage facility

SDF in SDF

meta-grammar

```
sorts Sort Symbol Production Grammar

context-free syntax
   Id -> Sort

   Sort                          -> Symbol
   String                        -> Symbol
   Symbol "?"                    -> Symbol
   Symbol "*"                    -> Symbol
   Symbol "+"                    -> Symbol
   "{" Symbol Symbol "}" "*"     -> Symbol
   "{" Symbol Symbol "}" "+"     -> Symbol

   Symbol* "->" Symbol           -> Production

   "sorts" Sort* "context-free" "syntax" Production* -> Grammar
```

V

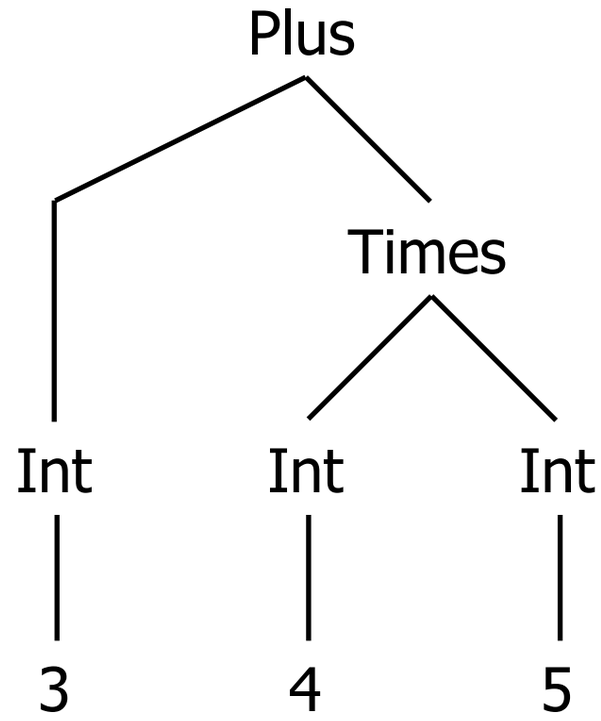
abstract syntax

ATerms

trees as terms

tree

- leaf
- parent node
- children (trees)



ATerms

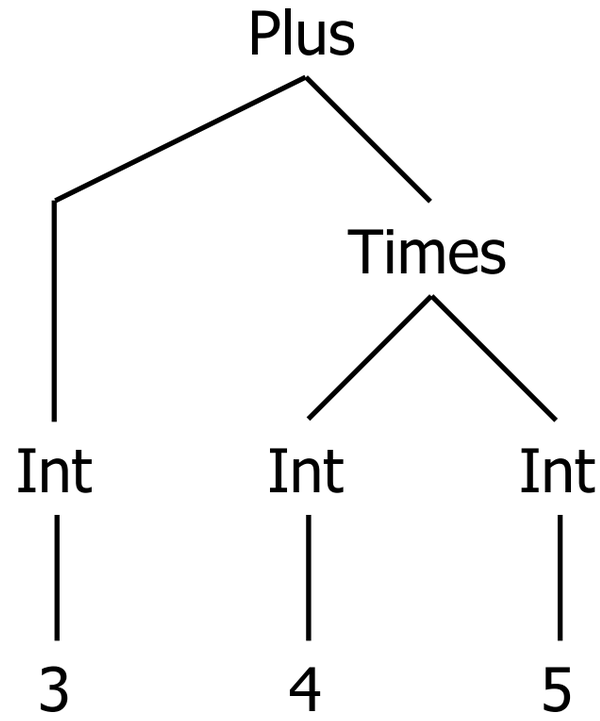
trees as terms

tree

- leaf
- parent node
  - children (trees)

terms

- constant
- constructor
  - arguments (terms)




```
Plus(
   Int ("3")
,  Times(
      Int("4")
   ,  Int("5")
   )
)
```

AST Construction

```
context-free syntax
   "-" Exp       -> Exp {cons("Uminus")}
   Exp "+"  Exp -> Exp {cons("Plus")}
   Exp "-"  Exp -> Exp {cons("Minus")}
   Exp "*"  Exp -> Exp {cons("Times")}
   Exp "/"  Exp -> Exp {cons("Divide")}

   Exp "="  Exp -> Exp {cons("Eq")}
   Exp "<>" Exp -> Exp {cons("Neq")}
   Exp ">"  Exp -> Exp {cons("Gt")}
   Exp "<"  Exp -> Exp {cons("Lt")}
   Exp ">=" Exp -> Exp {cons("Geq")}
   Exp "<=" Exp -> Exp {cons("Leq")}

   Exp "&"  Exp -> Exp {cons("And")}
   Exp "|"  Exp -> Exp {cons("Or")}

   "(" ")"              -> Exp {cons("NoVal")}
   "(" Exp ")"          -> Exp {bracket}
   "(" {Exp ";"}+ ")" -> Exp {cons("Seq")}
```

```
(x + 5) * f(x)
```

```
Times(
   Plus(
      Var("x")
   ,  Int("5")
   )
,  Call(
      "f"
   ,  [Var("x")]
   )
)
```

Parsing & AST construction

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

Parsing & AST construction

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

Parsing & AST construction

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

Parsing & AST construction

```
let
   function fact(n : int): int =
      if Var("n") < Int("1") then
         Int("1")
      else
         Var("n") * fact(Minus(Var("n"), Int("1")))
in
   printint(fact(Int("10")))
end
```

Parsing & AST construction

```
let
   function fact(n : int): int =
      if Var("n") < Int("1") then
         Int("1")
      else
         Var("n") *
            Call("fact", [Minus(Var("n"), Int("1"))])
in
   Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
   function fact(n : int): int =
      if Var("n") < Int("1") then
         Int("1")
      else
         Times(
            Var("n"),
            Call("fact", [Minus(Var("n"), Int("1"))])
         )
in
   Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
   function fact(n : int): int =
      if Lt(Var("n"), Int("1")) then
         Int("1")
      else
         Times(
            Var("n"),
            Call("fact", [Minus(Var("n"), Int("1"))])
         )
in
   Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
    function fact(n : int): int =
        IfThenElse(
            Lt(Var("n"), Int("1")),
            Int("1"),
            Times(
                Var("n"),
                Call("fact", [Minus(Var("n"), Int("1"))])
            )
        )
in
    Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
    function fact(n : Tid("int")): Tid("int") =
        IfThenElse(
            Lt(Var("n"), Int("1")),
            Int("1"),
            Times(
                Var("n"),
                Call("fact", [Minus(Var("n"), Int("1"))])
            )
        )
in
    Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
    function fact(FArg("n", Tid("int")): Tid("int") =
        IfThenElse(
            Lt(Var("n"), Int("1")),
            Int("1"),
            Times(
                Var("n"),
                Call("fact", [Minus(Var("n"), Int("1"))])
            )
        )
in
    Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

```
let
    FunDec(
        "fact",
        [FArg("n", Tid("int"))],
        Tid("int"),
        IfThenElse(
            Lt(Var("n"), Int("1")),
            Int("1"),
            Times(
                Var("n"),
                Call("fact", [Minus(Var("n"), Int("1"))])
            )
        )
    )
in
    Call("printint", [Call("fact", [Int("10")])])
end
```

Parsing & AST construction

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

Parsing & AST construction

Parsing & AST construction

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

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

Signatures

```
context-free syntax

   "-" Exp      -> Exp {cons("Uminus")}
   Exp "+"  Exp -> Exp {cons("Plus")}
   Exp "-"  Exp -> Exp {cons("Minus")}
   Exp "*"  Exp -> Exp {cons("Times")}
   Exp "/"  Exp -> Exp {cons("Divide")}

   Exp "="  Exp -> Exp {cons("Eq")}
   Exp "<>" Exp -> Exp {cons("Neq")}
   Exp ">"  Exp -> Exp {cons("Gt")}
   Exp "<"  Exp -> Exp {cons("Lt")}
   Exp ">=" Exp -> Exp {cons("Geq")}
   Exp "<=" Exp -> Exp {cons("Leq")}

   Exp "&"  Exp -> Exp {cons("And")}
   Exp "|"  Exp -> Exp {cons("Or")}
```

```
signature
   constructors
      Uminus : Exp       -> Exp
      Plus   : Exp * Exp -> Exp
      Minus  : Exp * Exp -> Exp
      Times  : Exp * Exp -> Exp
      Divide : Exp * Exp -> Exp

      Eq     : Exp * Exp -> Exp
      Neq    : Exp * Exp -> Exp
      Gt     : Exp * Exp -> Exp
      Lt     : Exp * Exp -> Exp
      Geq    : Exp * Exp -> Exp
      Leq    : Exp * Exp -> Exp

      And    : Exp * Exp -> Exp
      Or     : Exp * Exp -> Exp
```

metalanguage facility

SDF in SDF

meta-grammar

```
sorts Sort Symbol Production Grammar

context-free syntax
   Id -> Sort

   Sort                          -> Symbol
   String                        -> Symbol
   Symbol "?"                    -> Symbol
   Symbol "*"                    -> Symbol
   Symbol "+"                    -> Symbol
   "{" Symbol Symbol "}" "*"     -> Symbol
   "{" Symbol Symbol "}" "+"     -> Symbol

   Symbol* "->" Symbol                          -> Production
   Symbol* "->" Symbol "{" Annotation* "}"      -> Production

   "sorts" Sort* "context-free" "syntax" Production* -> Grammar
```

ATerms in SDF

grammar

```
sorts Term

lexical syntax
   Id     -> Cons
   String -> Cons

context-free syntax
   Cons                        -> Term {cons("Constant")}
   Cons "(" {Term ","}* ")"    -> Term {cons("Application")}
   "[" {Term ","}* "]"         -> Term {cons("List")}
   "(" {Term ","}* ")"         -> Term {cons("Tuple")}
```

ATerms in Stratego

signature

```
signature
   constructors
      Constructor : Cons                -> Term
      Application : Cons * List(Term)   -> Term
      List        : List(Term)          -> Term
      Tuple       : List(Term)          -> Term
```

```
signature
   constructors
      Nil  :                        -> List(a)
      Cons : a * List(a)            -> List(a)
      Conc : List(a) * List(a)      -> List(a)
```

coffee break

VI

disambiguation

Associativity

```
context-free syntax

   "-" Exp       -> Exp {cons("Uminus")}
   Exp "+"  Exp -> Exp {left, cons("Plus")}
   Exp "-"  Exp -> Exp {left, cons("Minus")}
   Exp "*"  Exp -> Exp {left, cons("Times")}
   Exp "/"  Exp -> Exp {left, cons("Divide")}

   Exp "="  Exp -> Exp {non-assoc, cons("Eq")}
   Exp "<>" Exp -> Exp {non-assoc, cons("Neq")}
   Exp ">"  Exp -> Exp {non-assoc, cons("Gt")}
   Exp "<"  Exp -> Exp {non-assoc, cons("Lt")}
   Exp ">=" Exp -> Exp {non-assoc, cons("Geq")}
   Exp "<=" Exp -> Exp {non-assoc, cons("Leq")}

   Exp "&"  Exp -> Exp {left, cons("And")}
   Exp "|"  Exp -> Exp {left, cons("Or")}
```

Operator Precedence

```
context-free priorities
   {
      "-" Exp       -> Exp
   } > { left:
      Exp "*"  Exp -> Exp
      Exp "/"  Exp -> Exp
   } > { left:
      Exp "+"  Exp -> Exp
      Exp "-"  Exp -> Exp
   } > { non-assoc:
      Exp "="  Exp -> Exp
      Exp "<>" Exp -> Exp
      Exp ">"  Exp -> Exp
      Exp "<"  Exp -> Exp
      Exp ">=" Exp -> Exp
      Exp "<=" Exp -> Exp
   }
   >  Exp "&"  Exp -> Exp
   >  Exp "|"  Exp -> Exp
```

Dangling Else

```
context-free syntax

   LValue ":=" Exp                        -> Exp {cons("Assign")}

   "let" Dec* "in" {Exp ";"}* "end" -> Exp {cons("Let")}

   "if" Exp "then" Exp "else" Exp   -> Exp {cons("IfThenElse")}
   "if" Exp "then" Exp              -> Exp {prefer, cons("IfThen")}

   "while" Exp "do" Exp                   -> Exp {cons("While")}
   "for" Id ":=" Exp "to" Exp "do" Exp -> Exp {cons("For")}
   "break"                                -> Exp {cons("Break")}
```

Dangling Else

```
context-free syntax

   LValue ":=" Exp                        -> Exp {cons("Assign")}

   "let" Dec* "in" {Exp ";"}* "end" -> Exp {cons("Let")}

   "if" Exp "then" Exp "else" Exp   -> Exp {cons("IfThenElse")}
   "if" Exp "then" Exp              -> Exp {prefer, cons("IfThen")}

   "while" Exp "do" Exp                   -> Exp {cons("While")}
   "for" Id ":=" Exp "to" Exp "do" Exp -> Exp {cons("For")}
   "break"                                -> Exp {cons("Break")}
```

```
context-free syntax

   "(" ")"            -> Exp {cons("NoVal")}
   "(" Exp ")"        -> Exp {bracket}
   "(" {Exp ";"}+ ")" -> Exp {avoid, cons("Seq")}
```

VII

syntactic editor services

Spoofax Editor Description

```
module Tiger.main

imports
   Tiger-Builders Tiger-Colorer Tiger-Completions
   Tiger-Folding Tiger-Outliner Tiger-References
   Tiger-Syntax

language General properties

   name     : Tiger
   id       : org.strategoxt.spoofax.tiger
   extends  : Root

   description : "Spoofax/IMP-generated Tiger compiler"
   url         : http://strategoxt.org

   extensions     : tig
   table          : include/Tiger.tbl
   start symbols  : Start
```

Bracket Matching

```
module Tiger-Syntax.generated

language Syntax properties (static defaults)

   line comment  : "//"
   block comment : "/*" * "*/"

   fences : [ ]
            ( )
            { }

indent after : "="
               ":"

identifier lexical : "[A-Za-z0-9_]+"
```

Outline View

```
module Tiger-Outliner

imports Tiger-Outliner.generated

outliner

    Dec*
    Dec
    Field
    FArg
```

```
module Tiger-Outliner.generated

outliner Default outliner

    Exp.Call
    Exp.Record
    Dec.FunDec
    Dec.FunDec
```

Content Folding

```
module Tiger-Folding

imports Tiger-Folding.generated

folding

    Dec*
    Dec.FunDec
```

```
module Tiger-Folding.generated

folding Default folding definitions

    Exp
    Dec.TypeDec
    Type.RecordTy
```

Syntax Highlighting

```
module Tiger-Colorer

imports Tiger-Colorer.generated

colorer TU Delft colours

TUDlavender = 123 160 201

colorer token-based highlighting

    layout    : TUDlavender
    StrConst  : darkgreen
    TypeId    : blue
```

Syntax Highlighting

```
module Tiger-Colorer.generated

colorer Default, token-based highlighting

   keyword    : 127 0 85 bold
   identifier : default
   string     : blue
   number     : darkgreen
   var        : 255 0 100 italic
   operator   : 0 0 128
   layout     : 100 100 0 italic

colorer System colors

   darkred    = 128 0 0
   red        = 255 0 0
   darkgreen  = 0 128 0
   green      = 0 255 0
   darkblue   = 0 0 128
   …
```

Code Completion

```
module Tiger-Completions

imports Tiger-Completions.generated

completions

   completion keyword  : "function"

   completion template Exp:

        "if " <e> " then\n\t" <s> (blank)

completion proposer : editor-complete
```

VIII

summary

Summary
lessons learned

Summary
lessons learned

How do you define the syntax of a software language with SDF?

- lexical syntax
- context-free syntax
- abstract syntax
- disambiguation

Summary
lessons learned

How do you define the syntax of a software language with SDF?

- lexical syntax
- context-free syntax
- abstract syntax
- disambiguation

How can you represent abstract syntax trees as terms?

Summary
lessons learned

How do you define the syntax of a software language with SDF?

- lexical syntax
- context-free syntax
- abstract syntax
- disambiguation

How can you represent abstract syntax trees as terms?

How do you specify syntactic editor services with Spoofax?

- outline view
- content folding
- syntax highlighting
- code completion

Literature

learn more

Literature

learn more

Spoofax and SDF

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

Eelco Visser: A Family of Syntax Definition Formalisms. ASF+SDF 1995

Literature

learn more

Spoofax and SDF

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

Eelco Visser: A Family of Syntax Definition Formalisms. ASF+SDF 1995

MiniJava

Andrew W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd edition. 2002

Literature

learn more

Spoofax and SDF

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

Eelco Visser: A Family of Syntax Definition Formalisms. ASF+SDF 1995

MiniJava

Andrew W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd edition. 2002

Tiger

Andrew W. Appel: Modern Compiler Implementation in ML. 1998

Outlook
coming next

declarative semantics definition

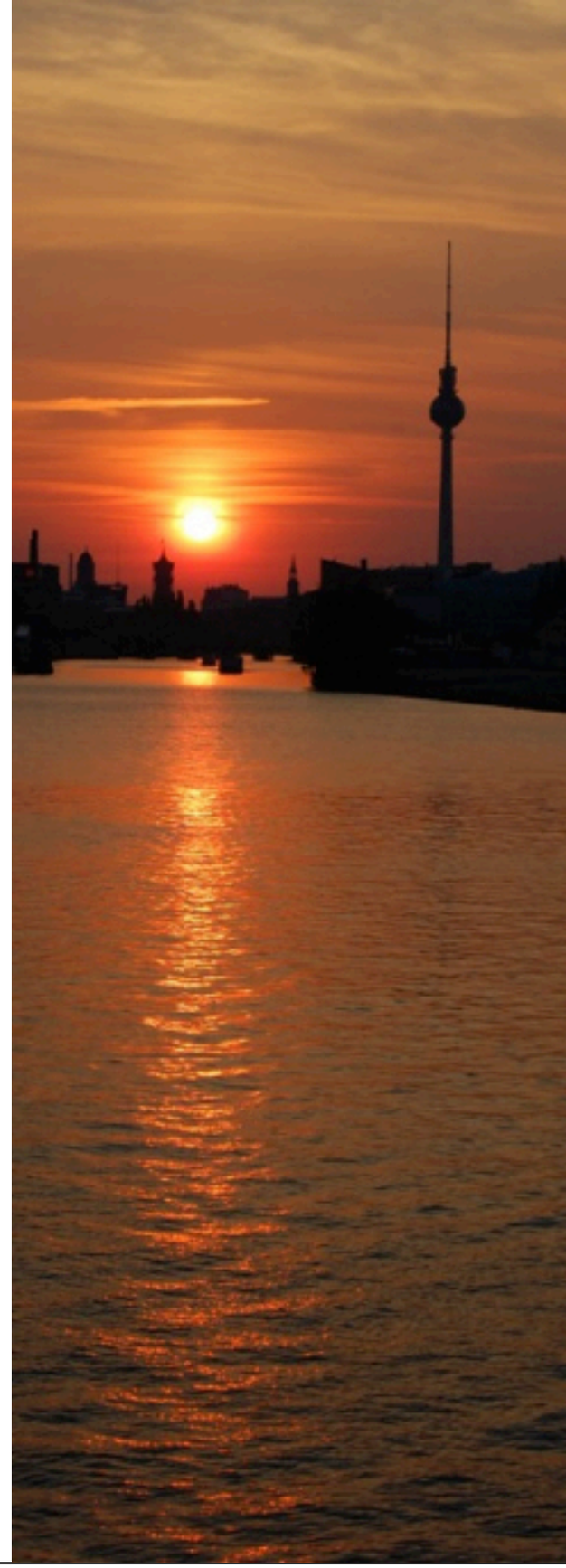
- Lecture 3: Term Rewriting
- Lecture 4: Static Analysis and Error Checking
- Lecture 5: Code Generation

Lab Sep 22

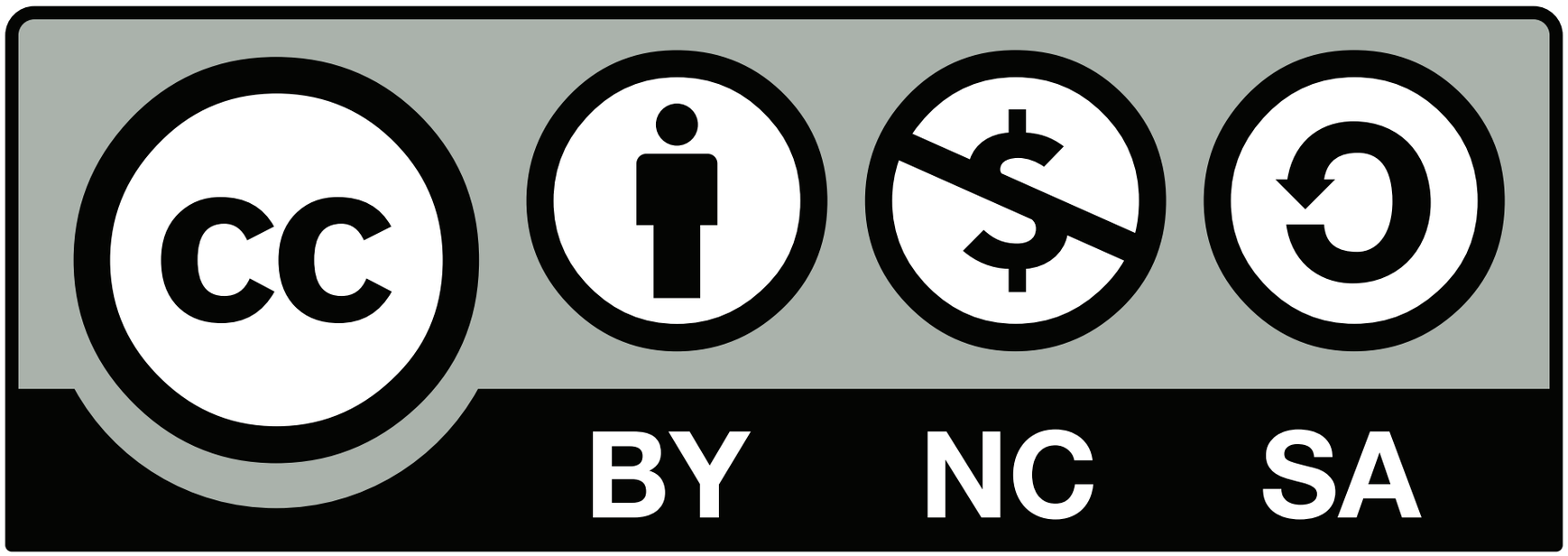
- extend test cases for ASTs and disambiguation
- complete syntax definition

Assignment Oct 2: Eclipse editor for MiniJava

- syntax definition in SDF
- syntactic editor services in Spoofax



copyrights


BY
NC
SA

Pictures
attribution & copyrights

Slide 1:
Screenshot of the Tiger syntax definition in Spoofax, public domain

Slides 2, 9, 11, 13, 22-28, 30-32, 38, 48, 49, 52-58:
Tiger by Bernard Landgraf, some rights reserved

Slide 5:
Programming language textbooks by K.lee, public domain

Slide 6:
Latin Grammar by Anthony Nelzin, some rights reserved

Slides 34, 42:
Thesaurus by Enoch Lau, some rights reserved

Slide 46:
Millie's Classic Coffee by Dominica Williamson, some rights reserved

Slide 62:
Sunset by ciaron, some rights reserved