Name Analysis

Viktor Kunčak, EPFL

Compiler phases

characters `var arg = 2;   res = 14 + arg * 3`

↓ lexical analyzer

words $\boxed{\texttt{var}}\ \boxed{\texttt{arg}}\ \boxed{\texttt{=}}\ \boxed{\texttt{2}}\ \boxed{\texttt{;}}\ \boxed{\texttt{res}}\ \boxed{\texttt{=}}\ \boxed{\texttt{14}}\ \boxed{\texttt{+}}\ \boxed{\texttt{arg}}\ \boxed{\texttt{*}}\ \boxed{\texttt{3}}$

↓ parser

trees (AST) Let(Def("arg", "Int", 2),
Assign("res", Plus(C(14), Times(V("arg"),C(3)))))

↓ **name analyzer**

AST + symbol table (graphs) Variables are mapped to declarations.

↓ type checker

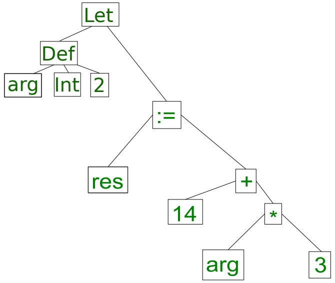
typed AST

↓ intermediate code generator

intermediate code

↓ JIT compiler or platform-specific back end

machine code



Examples of errors detected in compiler phases

characters	
↓ lexical analyzer	
words	Unrecognized token, line 12 col 4 in 'Arithmetic.scala'
↓ parser	
trees (AST)	Syntax error: '(' expected, line 12, col 4 in 'Arithmetic.scala'
↓ **name analyzer**	
ast + symbol table	**Unknown identifier: 'ammount'**
↓ type checker	
typed trees	Type error: '+' expects argument of type 'Int'; found 'x' : String
↓ . . .	

Parser did not differentiate different identifiers, just stored them.
Later phases know which identifier is which.

Further examples of errors that name analysis can report

- An identifier is used but not declared:

```
def p(amount: Int) { total = total + ammount }
```

- Multiple method arguments have the same name

```
def p(x:Int, y:Int, x:Int) { ... }
```

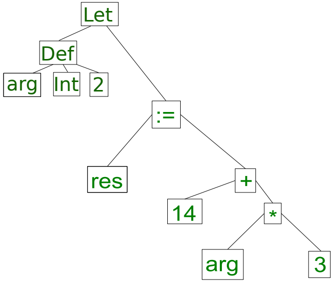
- Multiple functions with the same name

```
object Program {
  def m(x: Int) = { x + 1 }
  def m(x: Int) = { x + 3 } }
```

- Ill-formed type definition (e.g. circular)

```
class List extends Expr
class Cons extends List
class Expr extends Cons
```

Symbol tables: how to map uses to declarations?



Possibilities:

- at each occurrence search the tree for the declaration (very inefficient and clumsy)
- maintain a map from identifiers to declaration information (symbol) at each point in the tree: **symbol table**

Symbol tables can be computed every time, cached, or integrated partly or fully into trees as symbol references.
An efficient simple compiler: only symbol table, generate code while parsing (no AST).

What is in the symbol table?

Symbol table provides efficient access to information that gives meaning to identifiers:

- Declaration of a value or variable typically introduces its type and initial value.
- Variable inside a pattern introduces the name for part of the value in pattern matching
- Definition of a function specifies its signature (arguments and their types) and its body.
- Definition of an algebraic data type (e.g., case class) specifies its alternatives and fields of each alternative.

Symbol table is a map data structure

It maps identifiers (at a given position in the tree) to additional information such as types, memory locations, and their internal structure.
Map data structure implementations:

- mutable map: hash table or mutable search tree
  - Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Chapter 11: Hash Tables". Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 221–252. ISBN 978-0-262-53196-2.
- functional map: purely functional (persistent) search tree
  - Driscoll JR, Sarnak N, Sleator DD, Tarjan RE (1986). Making data structures persistent. Proceeding STOC '86. Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. pp. 109–121. doi:10.1145/12130.12142. ISBN 978-0-89791-193-1.
  - Chris Okasaki: Red-Black Trees in a Functional Setting. J. Funct. Program. 9(4): 471-477 (1999)
  - Chris Okasaki: Purely functional data structures. Cambridge University Press 1999, ISBN 978-0-521-66350-2, pp. I–X, 1–220

Scope and scoping rules

From Wikipedia: Scope (computer science): “In computer programming, the scope of a name binding–an association of a name to an entity, such as a variable–is the part of a program where the name binding is valid, that is where the name can be used to refer to the entity. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).”

We are only going to consider **static (lexical) scoping**, which is used in compiled languages today (Scala, Java, C, . . .). A specification of a programming language should clearly describe the scoping rules (shadowing, scopes for types vs values, . . .).

When we introduce an identifier (value, variable, function, type), scoping rules tell us where it is visible.
For example, **local variables** are only visible inside the function or block where they are introduced.
Scoping rules specify, given an occurrence of an identifier to which definition or memory location it corresponds.

Static scoping rule in an example

```scala
object World {
 var sum: Int = 0
 var value: Int = 0
 def add(): Unit = {
  sum = sum + value
  value = 0
 }
 def main(): Unit = {
  sum = 0
  value = 10
  add()
  if (sum % 3 == 1) {
   var value: Int = 1
   add()
   println("inner value: " + value)
   println("sum: " + sum)
  }
  println("outer value: " + value)
 }
}
```

Static scoping rule in an example

```scala
object World {
  var sum: Int = 0
  var value: Int = 0
  def add(): Unit = {
    sum = sum + value
    value = 0
  }
  def main(): Unit = {
    sum = 0
    value = 10
    add()
    if (sum % 3 == 1) {
      var value: Int = 1
      add()
      println("inner value: " + value)
      println("sum: " + sum)
    }
    println("outer value: " + value)
  }
}
```

```scala
object World {
  var sum: Int = 0
  var value: Int = 0
  def add(): Unit = {
    sum = sum + value
    value = 0
  }
  def main(): Unit = {
    sum = 0
    value = 10
    add()
    if (sum % 3 == 1) {
      var value1: Int = 1 // renamed
      add()
      println("inner value: " + value1)
      println("sum: " + sum)
    }
    println("outer value: " + value)
  }
}
```

With static scoping, we can rename identifiers to make them unique per binding.