Names, Scopes, and Bindings

Akim Demaille   Étienne Renault   Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

January 22, 2016

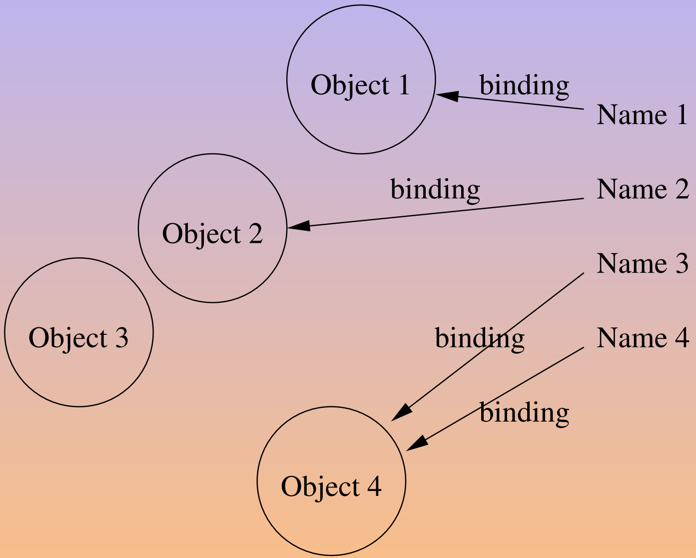
Names

Names, Identifiers, Symbols

- *Name* (*Identifiers*, *Symbols*)
- *reference*
- *address*
- *value*
- To refer to some entities: variable, type, function, namespace, constant, control structure (e.g., named `next`, `continue` in Perl), etc.

Identifiers

- usually alphanumeric and underscore, letter first, without white spaces.
- ALGOL 60, FORTRAN ignore white spaces.
- limitation on the length
  - 6 characters for the original FORTRAN (Fortran 90: 31),
  - ISO C: 31
  - no limit for most others.
- case insensitive in Modula-2 and Ada.


Object 1
binding
Name 1
binding
Name 2
Object 2
Name 3
Object 3
binding
Name 4
binding
Object 4

- When are objects created and destroyed?
  Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
  Scopes.
- When are bindings created and destroyed?
  Binding times.

Names, Objects, and Bindings

- When are objects created and destroyed?
  Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
  Scopes.
- When are bindings created and destroyed?
  Binding times.

Names, Objects, and Bindings

- When are objects created and destroyed?
  Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
  Scopes.
- When are bindings created and destroyed?
  Binding times.

Scopes

Scopes [Edwards, 2003]

When are names created, visible, and destroyed?

Scope
: The textual region in the source in which the binding is active.

Static Scoping
: The scope can be computed at compile-time.

Dynamic Scoping
: The scope depends on runtime conditions such as the function calls.

Scopes [Edwards, 2003]

When are names created, visible, and destroyed?

Scope
: The textual region in the source in which the binding is active.

Static Scoping
: The scope can be computed at compile-time.

Dynamic Scoping
: The scope depends on runtime conditions such as the function calls.

Scopes [Edwards, 2003]

When are names created, visible, and destroyed?

Scope
: The textual region in the source in which the binding is active.

Static Scoping
: The scope can be computed at compile-time.

Dynamic Scoping
: The scope depends on runtime conditions such as the function calls.

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are "useless"
- C++ namespaces are "pure scopes"

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are "useless"
- C++ namespaces are "pure scopes"

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are "useless"
- C++ namespaces are "pure scopes"

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
  (First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Declaration

Blocks determine scopes.

- local variables
- non local variables
- global variables

```c
int global;

int outer(void)
{
  int local, non_local;

  int inner(void)
  {
    return global + non_local;
  }

  return inner();
}
```

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Static Scoping

- In most languages
  (Ada, C, Tiger, FORTRAN, Scheme, Perl (`my`), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
  - safer
  - faster
  - clearer

Dynamic Scoping

- In most scripting/interpreted languages (Perl (`local`), Shell Script, $\TeX$ etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing

  An identifier may refer to different values, with different types.

Dynamic Scoping

- In most scripting/interpreted languages (Perl (`local`), Shell Script, TEX etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing
  An identifier may refer to different values, with different types.

Dynamic Scoping

- In most scripting/interpreted languages (Perl (`local`), Shell Script, TEX etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing
  An identifier may refer to different values, with different types.

Scopes in Tiger

Many different `t`, including several “variables”.

t time

```
let
  type     t = { h: int, t: t }
  function t (h: int, t: t) : t =
           t { h = h, t = t }
  var      t := t (12, nil)
  var      t := t (12, t)
in
  t.t = t
end
```

Scopes [Appel, 1998]

ML

```
structure M = struct
  structure E = struct
    val a = 5;
  end
  structure N = struct
    val b = 10;
    val a = E.a + b;
  end
  structure D = struct
    val d = E.a + N.a;
  end
end
```

Java (fwd declaration allowed)

```
package M;
class E {
  static int a = 5;
}
class N {
  static int b = 10;
  static int a = E.a + b;
}
class D {
  static int d = E.a + N.a;
}
```

Scopes [Appel, 1998]

```
structure M = struct
  structure E = struct
    val a = 5;
  end
  structure N = struct
    val b = 10;
    val a = E.a + b;
  end
  structure D = struct
    val d = E.a + N.a;
  end
end
```

$$
\begin{aligned}
\sigma_0 &= \text{Prelude} \\
\sigma_1 &= \{a : int\} \\
\sigma_2 &= \{E : \sigma_1\} \\
\sigma_3 &= \{b : int, a : int\} \\
\sigma_4 &= \{N : \sigma_3\} \\
\sigma_5 &= \{d : int\} \\
\sigma_6 &= \{D : \sigma_5\} \\
\sigma_7 &= \sigma_2 + \sigma_4 + \sigma_6
\end{aligned}
$$

$$
\begin{aligned}
\sigma_0 + \sigma_2 &\vdash N : \sigma_3 \quad \text{(ML)} \\
\sigma_0 + \sigma_2 + \sigma_4 &\vdash N : \sigma_3 \quad \text{(Java)} \\
\sigma_0 + \sigma_2 + \sigma_4 + \sigma_6 &\vdash M : \sigma_7
\end{aligned}
$$

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (`static`), ALGOL 60 `own`, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (`static`), ALGOL 60 `own`, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (`static`), ALGOL 60 `own`, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (`static`), ALGOL 60 `own`, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (`static`), ALGOL 60 `own`, PL/I. Initialization?
- Modules tend to replace this block related feature.

Binding Time

1. Bindings
   - Names
   - Scopes
   - Binding Time
2. Symbol Tables
3. Complications

Binding Time [Edwards, 2003]

When a binding from a name to an object is made.

Binding Time	Examples
language design	`if`
language implementation	data width
program writing	`foo`, `bar`
compilation	static objects, code
linkage	relative addresses
loading	shared objects
execution	heap objects

Binding Time: the moving IN

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN

```
                binding-time
early ---------------------------------> late

INflexibility                   flexibility
efficiency                    INefficiency
```

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN

```
                 binding-time
early ---------------------------------> late

INflexibility                   flexibility
efficiency                    INefficiency
```

Binding Time: the moving IN

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN

```
                binding-time
early -----------------------------------> late

INflexibility                    flexibility
efficiency                     INefficiency
```

Dynamic Binding: `virtual` in C++

Dynamic dispatch is roughly runtime overloading.

Dynamic Dispatch in C++

```
struct Shape
{
  virtual void draw() const = 0;
};

struct Square : public Shape
{
  virtual void draw() const override {};
};

struct Circle : public Shape
{
  virtual void draw() const override {};
};
```

Dynamic Binding: virtual in C++

Dynamic Dispatch in C++

```
#include <vector>
#include "shapes.hh"

using shapes_type = std::vector<Shape*>;

int main()
{
  auto ss = shapes_type{new Circle, new Square};

  for (auto s: ss)
    // Inclusion polymorphism.
    s->draw();
}
```

Late Code Binding: `eval`

- Most interpreted languages support `eval` (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
try {
  die "phooey";
} catch {
  /phooey/ and print "unphooey\n";
};
```

Late Code Binding: `eval`

- Most interpreted languages support `eval` (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
sub try (&@) {
  my ($try, $catch) = @_;
  eval { &$try };
  if ($@) {
    local $_ = $@;
    &$catch;
  }
}
sub catch (&) {
  $_[0];
}

try {
  die "phooey";
} catch {
  /phooey/ and print "unphooey\n";
};
```

Late Code Binding: eval

- Most interpreted languages support `eval` (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
sub try (&@) {
  my ($try, $catch) = @_;
  eval { &$try };   # Explicit eval.
  if ($@) {
    local $_ = $@;
    &$catch;
  }
}
sub catch (&) {
  $_[0];            # implicit eval.
}

try {
  die "phooey";
} catch {
  /phooey/ and print "unphooey\n";
};
```

Late Code Binding: eval

- Most interpreted languages support eval (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```perl
sub try (&@) {
  my ($try, $catch) = @_;
  eval { &$try };  # Explicit eval.
  if ($@) {
    local $_ = $@;
    &$catch;
  }
}
sub catch (&) {
  $_[0];            # implicit eval.
}

try {
  die "phooey";
} catch {
  /phooey/ and print "unphooey\n";
};
```

Binding Times in Tiger [Edwards, 2003]

Design	Keywords
Program	Identifiers
Compile	Function code, frames, types
Execution	Records, arrays addresses

Little dynamic behavior

Symbol Tables

Visiting an ast

For statically scoped languages

- many traversals check uses against definitions
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check uses against definitions
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check uses against definitions
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check uses against definitions
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check uses against definitions
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support `undo`: scoped symbol tables
- no: rely on automatic variables

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- `put`
- `get`

Implementation

- a list
- a tree
- a hash
- ...

Scoped Symbol Table: symbol::Table

class Table

```
template <typename Entry_T>
class Table
{
public:
  Table();

  auto put(symbol key, Entry_T& val) -> void;
  auto get(symbol key) const -> Entry_T*;

  auto scope_begin() -> void;
  auto scope_end() -> void;

  auto print(std::ostream& ostr) const -> void;
};
```

Not very C++...

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Memory Management

When do you deallocate associated data?

- scope end: deallocate everything since the latest `scope_begin`
- pass end: deallocate auxiliary data after the traversal is completed
- ast: bind the data to the ast and delegate deallocation
- by hand: thanks God for Valgrind
- never

Memory Management

When do you deallocate associated data?

- **scope end** deallocate everything since the latest `scope_begin`
- **pass end** deallocate auxiliary data after the traversal is completed
- **ast** bind the data to the ast and delegate deallocation
- **by hand** thanks God for Valgrind
- **never**

Memory Management

When do you deallocate associated data?

- **scope end** deallocate everything since the latest `scope_begin`
- **pass end** deallocate auxiliary data after the traversal is completed
- **ast** bind the data to the ast and delegate deallocation
- **by hand** thanks God for Valgrind
- **never**

Memory Management

When do you deallocate associated data?

- **scope end** deallocate everything since the latest `scope_begin`
- **pass end** deallocate auxiliary data after the traversal is completed
- **ast** bind the data to the ast and delegate deallocation
- **by hand** thanks God for Valgrind
- **never**

Memory Management

When do you deallocate associated data?

scope end	deallocate everything since the latest `scope_begin`
pass end	deallocate auxiliary data after the traversal is completed
ast	bind the data to the ast and delegate deallocation
by hand	thanks God for Valgrind and Paracetamol
never	

Memory Management

When do you deallocate associated data?

- **scope end** deallocate everything since the latest `scope_begin`
- **pass end** deallocate auxiliary data after the traversal is completed
- **ast** bind the data to the ast and delegate deallocation
- **by hand** thanks God for Valgrind and Paracetamol
- **never**

Memory Management

When do you deallocate associated data?

- **scope end** deallocate everything since the latest `scope_begin`
- **pass end** deallocate auxiliary data after the traversal is completed
- **ast** bind the data to the ast and delegate deallocation
- **by hand** thanks God for Valgrind and Paracetamol
- **never** tu sors

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice `foo`

```
let var foo := 42
    var foo := 51
in foo end
```

Two `lets`

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...
Courtesy of Arnaud Fabre.

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Bind the names/Label by definition address

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds…

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

Binder

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds... `breaks` to their loops

Complications

Overloading

Overloading

Overloading: Homonyms
: Several entities bearing the same name, but statically distinguishable, e.g., by their arity, type etc.

```
// foo is overloaded.
int foo(int);
int foo(float);
```

Aliasing: Synonyms
: One entity bearing several names.

```
// x and y are aliases.
int x;
int& y = x;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;
- : int = 3
# 1.0 + 2.0;;
Characters 0-3:
  1.0 + 2.0;;
  ^^^
This expression has type float but is here used with type int
# 1.0 +. 2.0;;
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int   a = 1   + 2;;
float b = 1.0 + 2.0;;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;
- : int = 3
# 1.0 + 2.0;;
Characters 0-3:
  1.0 + 2.0;;
  ^^^
This expression has type float but is here used with type int
# 1.0 +. 2.0;;
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int   a = 1   + 2;;
float b = 1.0 + 2.0;;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;
- : int = 3
# 1.0 + 2.0;;
Characters 0-3:
  1.0 + 2.0;;
  ^^^
This expression has type float but is here used with type int
# 1.0 +. 2.0;;
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int   a = 1   + 2;;
float b = 1.0 + 2.0;;
```

Of course this is unfair: Caml has type inference.

Function Overloading

Usually based on the arguments
(Ada, C++, Java...; not C, ALGOL 60, Fortran...).

ALGOL 60

```
integer I;
real X;
...
PUTSTRING("results are: ");  PUTINT(I);  PUTREAL(X);
```

Ada [ARM, 1983]

```
I :  INTEGER;
X :  REAL;
...
PUT("results are: ");  PUT(I);  PUT(X);
```

Overloading is Syntactic Sugar

Overloaded

```
#include <string>

void foo(int);
void foo(char);
void foo(const char*);
void foo(std::string);

int
main ()
{
  foo(0);
  foo('0');
  foo("0");
  foo(std::string("0"));
}
```

Un-overloaded

```
#include <string>

void foo_int(int);
void foo_char(char);
void foo_char_p(const char*);
void foo_std_string(std::string);

int
main ()
{
  foo_int(0);
  foo_char('0');
  foo_char_p("0");
  foo_std_string(std::string("0"));
}
```

Overloading is Syntactic Sugar

Overloaded

```
#include <string>

void foo(int);
void foo(char);
void foo(const char*);
void foo(std::string);

int
main ()
{
  foo(0);
  foo('0');
  foo("0");
  foo(std::string("0"));
}
```

Un-overloaded

```
#include <string>

void foo_int(int);
void foo_char(char);
void foo_char_p(const char*);
void foo_std_string(std::string);

int
main ()
{
  foo_int(0);
  foo_char('0');
  foo_char_p("0");
  foo_std_string(std::string("0"));
}
```

Overloading is Syntactic Sugar

Usually solved by renaming/mangling.

g++-2.95, como

```
f__Fi  -> int f(int);
f__FPc -> int f(char*);
```

g++-3.2, icc

```
_Z1fi  -> int f(int);
_Z1fPc -> int f(char*);
```

Overloading in Tiger

Ordering <, <=, >, and >=
overloaded for pairs of integers, or strings.

Identity = and <>
overloaded for (type coherent) pairs of integers, strings, arrays or records.

Non Local Variables

Lambda Shifting

With nested functions

```
int global;

int outer(void)
{
  int local, non_local;

  int inner(void)
  {
    return
      global + non_local;
  }

  return inner();
}
```

Without

```
int global;

int outer_inner_(int* non_local)
{
  return global + *non_local;
}

int outer(void)
{
  int local, non_local;
  return outer_inner_(&non_local);
}
```

Lambda Shifting

With nested functions

```
int global;

int outer(void)
{
  int local, non_local;

  int inner(void)
  {
    return
      global + non_local;
  }

  return inner();
}
```

Without

```
int global;

int outer_inner_(int* non_local)
{
  return global + *non_local;
}

int outer(void)
{
  int local, non_local;
  return outer_inner_(&non_local);
}
```

Non Local Variables

```
let
  function outer(): int =
    let
      non-local var outer := 0
    in
      let
        function inner() : int =
          let
            var inner := 1
          in
            inner + outer
          end
      in
        inner()
      end
    end
in
  outer ()
end
```

Non Non Local Variables

```
let

    let
      local var outer := 0
    in
      let

          let
            var inner := 1
          in
            inner + outer
          end
      in

      end
    end
in

end
```

Non Non Local Variables

```
let
  function outer(): int =
    let
      local var outer := 0
    in
      let

          let
            var inner := 1
          in
            inner + outer
          end
      in

      end
    end
in
  outer()
end
```

The Escapes and Functional Programming

```
let
  function add(non-local a: int, b: int) : int =
    let
      function add_a(x: int) : int = a + x
    in
      add_a(b)
    end
in
  print_int(add(1, 2));
  print("\n")
end
```

Closures

```
let
  function add_gen(non-local a: int) : int -> int =
    let
      function add_a(x: int) : int = a + x
    in
      add_a
    end
  incr = add_gen(1);
in
  print_int(incr(2));
  print("\n");
end
```

The Escapes & Recursion

```
let
  function one(input : int) =
    let
      function two() =
        (print("two: "); print_int(input);
         print("\n");
         one(input))
    in
      if input > 0 then
        (input := input - 1;
         two(); print("one: ");
         print_int(input); print("\n"))
    end
in
  one (3)
end
```

Escaping Variables/Arguments

Technically escaping means "cannot be stored in a register".

In C
- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger
- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically escaping means “cannot be stored in a register”.

In C
- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger
- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Technically escaping means "cannot be stored in a register".

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically escaping means "cannot be stored in a register".

In C
- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger
- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically escaping means "cannot be stored in a register".

In C
- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger
- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically escaping means "cannot be stored in a register".

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically escaping means "cannot be stored in a register".

In C
- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger
- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preleminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preleminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preleminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preleminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preleminary pass should flag non local variables

Bibliography I

Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.

ARM (1983).
Ada Reference Manual.

Edwards, S. (2003).
COMS W4115 Programming Languages and Translators.
http://www.cs.columbia.edu/~sedwards/classes/2003/w4115/.