Syntax of Hobbes

Alan Jeffrey
CTI, DePaul University
http://fpl.cs.depaul.edu/ajeffrey/

11 September 2002

1 Introduction

This is the syntactic specification of Hobbes, a strongly-typed class-based multi-threaded object-oriented language.

This is an incomplete draft: the language is missing many of its important features. These will be added as the course proceeds.

Hobbes is a 'pure' object-oriented language: everything is an object, and there are no imperative features such as variable assignment or `while` loops. Instead, recursion has to be used, for example, a recursive factorial function is written:

A program can contain any number of `class`, `interface`, `object` and `thread` declarations. Week 7 begin

Week 7 end

The Hobbes language is specified in three parts: the lexical structure, the core language and the surface language. The core language is a strict subset of the surface language, and the surface language adds useful syntax sugar:

- Infix methods such as `x+y` and prefix methods such as `-x`.
- Complex expressions such as `(x+y)*3`; in the core language, only simple expressions involving constants or variables such as `x+y` or `tmp*3` are allowed.
- Complex `if` statements using `else if` and with an optional `else` statement; in the core language, all `if` statements are of the form `if (...) { ... } else { ... }`.
- A default return type `Void` for all method headers (similar to C's `void` type) and a default return value `Nothing:Void` for all method bodies. Week 8 begin
- By default, classes, interfaces, methods and objects are non-generic: this is just sugar for a declaration with an empty list of type parameters. Week 8 end

The surface language is intended to be useful for programmers, but the core language is much simpler to provide static and dynamic semantics for.

2 Conventions used in this specification

2.1 Extended Backus Naur Form

This specification uses Extended Backus Naur Form (EBNF) declarations:

```
Nonterminal ::= Definition
```

The Definition can contain:

- Nonterminals, written Nonterminal. Each nonterminal must have a matching definition.
- Terminals, written "Terminal" or 'Terminal'. These are constant strings, and do not require a definition.
- Empty string, written ε.
- Choice, written Definition1 | Definition2.
- Sequencing, written Definition1 Definition2.
- Option, written Definition?.
- Nonempty list, written Definition+.
- Possibly empty list, written Definition*.

2.2 Notation for lists

We will use ... notation a sugar for lists, for example:

- Foo ... Foo is sugar for Foo*.
- Foo"," ..." Foo is sugar for (Foo ("," Foo)*)?

2.3 Notation for characters

We also use some syntax sugar for characters and character ranges:

- '<cr>' is carriage return and '<nl>' is new line.
- ['0'-'9'] is a character range, sugar for "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" and similarly for other character ranges.
- ~['a', 'b', 'c'] matches any character other than a, b or c.

3 Lexical issues

3.1 Comments and white space

Hobbes comments are either begun by // and ended by a line break or begun by /* and ended by */.

White space (space, tab, carriage return or new line) between tokens is unimportant.

3.2 Keywords

Hobbes treats the following symbolic strings as keywords:

```
":="   "="   "::"   ":"   ";"   "{"   "}"   "("   ")"   "["   "]"   "."   ","
```

Hobbes treats the following alphanumeric strings as keywords:

```
"block" "class" "continuation" "currentThread" "else" "extends"
"field" "if" "immutable" "implements" "infix" "interface" "let"
"method" "mutable" "new" "object" "prefix" "return" "thread" "type"
```

3.3 Identifiers

An identifier is a sequence of alphanumerics starting with an alphabetic, such as `a`, `foo37` or `Foo_8_Bar`. For simplicity, Hobbes only allows ASCII alphabetics. Hobbes distinguishes syntactically between global identifiers (such as classes) which must start with a capital, and local identifiers (such as variables) which must start with a lower case letter: this enforces the Java naming convention, and makes static and dynamic semantics much simpler.

```
LocalId ::=
  Lower ( Alpha | Digit | "_" )*

GlobalId ::=
  Upper ( Alpha | Digit | "_" )*

Lower ::=
  ['a'-'z']

Upper ::=
  ['A'-'Z']

Alpha ::=
  Lower | Upper
```

A symbolic identifier is a sequence of symbolic characters such as `+`, `<<` or `!=`. These are used as infix or prefix methods, such as `x+y` or `x<<<y`.

```
Symbolic ::=
  ( "+" | "-" | "*" | "/" | "=" | "<" | ">" | "!" | "$" | "#" | "@" )+
```

3.4 Integer literals

An integer literal is either a decimal literal such as `96` or `-5` or hex literal such as `0x1AB`:

```
IntegerLiteral ::=
  "-"? ( DecimalLiteral | HexLiteral )

DecimalLiteral ::=
  Digit+

HexLiteral ::=
  "0" ("x" | "X") HexDigit
```

```
Digit ::=
  ['0'-'9']

HexDigit ::=
  ['0'-'9'] | ['a'-'f'] | ['A'-'F']
```

3.5 String literals

A string literal is a sequence of characters surrounded by quotation marks such as "abc". Strings can include escape sequences such as \n for new line, but cannot contain line breaks.

```
StringLiteral ::=
  '"' CharSeq '"'

CharSeq ::=
  ( Char | Escape )+

Char ::=
  ~[ '"', '\', '<nl>', '<cr>' ]

Escape ::=
  "\" ( "n" | "t" | "b" | "r" | "f" | "'" | '"' )
```

4 Core Hobbes

4.1 Programs

The main declaration unit is a Program which is a collection of declarations:

```
Program ::=
  Dec ... Dec
```

4.2 Declarations

Declarations are either class declarations, interface declarations, object declarations or thread declarations.

Week 7 begin
Week 7 end

```
Dec ::=
  DecClass |                                    Week 7 begin
  DecInterface |                                Week 7 end
  DecObject |
  DecThread

DecClass ::=
  "class" GlobalId                              Week 8 begin
  TypeParams                                    Week 8 end
  ExtendsClass
  ImplementsInterfaces                          Week 9 begin
  ExtendedClass                                 Week 9 end
  "{"
    FieldAbstract ... FieldAbstract             Week 5 begin
```

```
    MethodConcrete ... MethodConcrete                                    Week 5 end
  "}"

                                                                         Week 8 begin
TypeParams ::=
  "[" TypeVar "," "..." "," TypeVar "]"

ExtendsClass ::=
  "extends" TypeGlobal

ImplementsInterfaces ::=
  "implements" TypeGlobal "," "..." "," TypeGlobal

                                                                         Week 9 begin
ExtendedClass ::=
  "extended" LocalId

                                                                         Week 9 end
DecInterface ::=
  "interface" GlobalId                                                   Week 8 begin
  TypeParams                                                             Week 8 end
  ExtendsInterfaces
  "{"
    MethodAbstract ... MethodAbstract
  "}"

                                                                         Week 8 begin
ExtendsInterfaces ::=
  "extends" TypeGlobal "," "..." "," TypeGlobal

                                                                         Week 8 end
DecObject ::=
  "object" GlobalId                                                      Week 8 begin
  TypeParams
  ":" TypeGlobal                                                         Week 8 end
  "{" FieldConcrete "," "..." "," FieldConcrete "}"

DecThread ::=
  "thread" GlobalId "{" Block "}"
```

4.3 Fields

Class declarations contain *abstract fields* which have types, but no values, for example:

```
class Point {
  immutable field x : Integer;
  immutable field y : Integer;
}
```

Object declarations contain *concrete fields* which have values, but no types, for example:

```
object P : Point { x = 5, y = 37 }
```

A *field* is either an abstract field or a concrete field:

```
Field ::=
  FieldAbstract |
  FieldConcrete

FieldAbstract ::=
```

```
  Mutability "field" LocalId ":" Type ";"

Mutability ::=
  "mutable" |
  "immutable"

FieldConcrete ::=
  LocalId "=" Val
```

Week 5 begin

4.4 Methods

Class declarations contain *concrete methods* which have headers and bodies, for example:

```
class FooImpl implements Foo {
  method double (x : Integer) : Integer {
    return x * 2;
  }
}
```

Method headers provide type information, and the method body contains the code to be executed when the method is called. Methods can name their 'self' parameter (always called this in Java) for example:

```
class IntRef {
  mutable field contents : Integer;
  method self.get () : Integer {
    return self.contents;
  }
  method self.set (x : Integer) {
    self.contents := x;
  }
}
```

Interface declarations contain *abstract methods* which have headers but no bodies, for example:

```
interface Foo {
  method double (x : Integer) : Integer;
}
```

Abstract methods only have type information, so the method body has to be provided by the implementing class.

```
Method ::=
  MethodAbstract |
  MethodConcrete

MethodAbstract ::=
  "method" ThisVar MethodId
  "(" VarTyped "," ... "," VarTyped ")" ":" ReturnType ";"

MethodConcrete ::=
  "method" ThisVar MethodId
  "(" VarTyped "," ... "," VarTyped ")" ":" ReturnType
  "{" Block "}"
```

```
ThisVar ::=
  Var "."

ReturnType ::=
  Type
```

Week 5 end

4.5 Blocks

A block is executable code. It contains a series of variable assignments and if statements, followed by a return statement. In the core language, statements are only allowed to contain *values* (either constants or variables), and not arbitrary *complex expressions*. For example the following is not legal in the core language:

```
thread Main {
  return 1+2+3;
}
```

instead, you have to introduce explicit temporary variables:

```
thread Main {
  let tmp1 = 1+2;
  let tmp2 = tmp1+3;
  return tmp2;
}
```

Complex expressions are allowed in the surface syntax, but they are stripped out during the transform from the surface syntax to the core syntax; this is discussed in Section 5.

```
Block ::=
  LetBlock |
  IfBlock |
  ReturnBlock

LetBlock ::=
  "let" Var "=" Exp ";"
  Block

IfBlock ::=
  "if" "(" Val ")" "{" Block "}"
  "else" "{" Block "}"

ReturnBlock ::=
  "return" Val ";"
```

4.6 Expressions

An expression is either a method call, a field access, a field update, a new object creation, or a value, for example:

```
foo.bar (
  foo:Baz.bang (),
  $(x * 2),
```

```
  foo.fred.wilma.fish (),
  new Baz {}
)
```

These complex expressions are converted into blocks, as described in Section 5.4.

```
Exp ::=
  ExpCall |
  ExpFieldAccess |
  ExpFieldUpdate |
  ExpNewObject |
  ExpVal

ExpCall ::=
  Val ( "::" Type )? "." MethodId "(" Val "," "..." "," Val ")"

ExpFieldAccess ::=
  Val "." LocalId

ExpFieldUpdate ::=
  Val "." LocalId ":=" Val

ExpNewObject ::=
  "new" Type "{" FieldConcrete "," ... "," FieldConcrete "}"

ExpVal ::=
  Val
```

4.7 Values

A value is either an integer constant such as 37, a string constant such as "foo" or a variable such as bar.

```
Val ::=
  ValGlobal |
  ValLocal |
  ValInteger |
  ValString

ValGlobal ::=
  GlobalId                                                    Week 8 begin
  TypeArgs                                                    Week 8 end

ValLocal ::=
  LocalId

ValInteger ::=
  IntegerLiteral

ValString ::=
  StringLiteral
```

4.8 Method identifiers

A method identifier is either a regular identifier such as foo, or a prefix identifier such as prefix + or an infix identifier such as infix +.

```
MethodId ::=
  LocalId |
  "prefix" PrefixId |
  "infix" InfixId

PrefixId ::=
  Symbolic

InfixId ::=
  Symbolic | LocalId
```

4.9 Variables

A variable is an identifier with an optional type for example `foo` or `foo : Integer`.

```
Var ::=
  VarTyped |
  VarUntyped

VarTyped ::=
  LocalId ":" Type

VarUntyped ::=
  LocalId
```

4.10 Types

Week 8 begin

A type is either:

- A *local type* such as the type `a` in `class Foo[type a] { field foo : a; }`. Local types have scope given by the type variable `type a` for example in this case `a` is in scope for the declaration of `Foo`, but not anywhere else.
- A *global type* such as the type `Foo[Integer]`. Global types are in scope anywhere. They have a name (in this case `Foo`) and some type arguments (in this case `Integer`).

```
Type ::=
  TypeLocal |
  TypeGlobal

TypeLocal ::=
  LocalId

TypeGlobal ::=
  GlobalId TypeArgs

TypeArgs ::=
  "[" Type "," ... "," Type "]"
```

4.11 Type variables

Local types have scope given by type variables, for example:

```
mutable class Ref[type a] {
  field contents : a;
  method self.get () : a {
    return self.contents;
  }
  method self.set (x : a) {
    self.contents := x;
  }
}
```

In this class declaration the type variable a has scope for just the declaration of Ref. When the Ref class is instantitated (say to Ref[Integer]) the type variable a is replaced by Integer, for example:

```
thread Main {
  let x : Ref[Integer] = new Ref[Integer] { contents = 5 };
  x.set (47);
  let y : Integer = x.get ();
}
```

```
TypeVar ::=
  "type" LocalId
```

Week 8 end

5 Surface Hobbes

The surface language of Hobbes extends the core language with complex expressions such as (4*x)+(y*3).

5.1 Complex expressions

An expression is either a dynamic method call, a static method call, an infix expression, a prefix expression, a field access, a new object creation, or a value, for example:

```
foo.bar (
  foo:Baz.bang (),
  $(x * 2),
  foo.fred.wilma.fish (),
  new Baz {}
)
```

These complex expressions are converted into blocks, as described in Section 5.4.

```
Exp ::=
  ExpCall |
  ExpInfix |
  ExpPrefix |
  ExpFieldAccess |
  ExpFieldUpdate |
  ExpNewObject |
  ExpVal

ExpCall ::=
  Exp "." ( "::" Type )? MethodId "(" Exp "," "..." "," Exp ")"
```

```
ExpInfix ::=
  Exp (Symbolic | LocalId) Exp

ExpPrefix ::=
  Symbolic Exp

ExpFieldAccess ::=
  Exp "." LocalId

ExpFieldUpdate ::=
  Exp "." LocalId ":=" Exp

ExpNewObject ::=
  "new" Type "{" FieldConcrete "," ... "," FieldConcrete "}"

ExpVal ::=
  Val
```

Prefix expressions have precedence over infix expressions, and infix expressions are all of the same precedence, and left-associative. For example:

- `!x + y` is parsed as `(!x) + y`.
- `x + y + z` is parsed as `(x + y) + z`.
- `x + y * z` is parsed as `(x + y) * z` (note this is different from C-like languages!).

5.2 Extended syntax of blocks

We extend the grammar of blocks to allow expressions in statements; we also make the `return` statement at the end of a block optional, and extend the syntax of `if` statements:

```
Block ::=
  LetBlock |
  ExpBlock |
  IfBlock |
  ReturnBlock

LetBlock ::=
  "let" Var "=" Exp ";"
  Block

ExpBlock ::=
  Exp ";"
  Block

IfBlock ::=
  "if" "(" Exp ")" "{" Block "}"
  ( "else" ( IfExpBlock | "{" Block "}" ) )?

ReturnBlock ::=
  ( "return" ( Exp )? ";" )?
```

5.3 Optional attributes

The mutability attribute of a field declaration is optional.

```
Mutability ::=
  (
    "mutable" |
    "immutable"
  )?
```

The self variable of a method declaration is optional.

```
ThisVar ::=
  ( Var "." )?
```

The return type of a method declaration is optional.

```
ReturnType ::=
  ( Type )?
```

Week 6 begin

The extends and implements clauses of type declarations are optional:

Week 8 begin

```
ExtendsClass ::=
  ( "extends" TypeGlobal )?

ImplementsInterfaces ::=
  ( "implements" TypeGlobal ","..."," TypeGlobal )?

ExtendsInterfaces ::=
  ( "extends" TypeGlobal ","..."," TypeGlobal )?
```

Week 8 end

Week 9 begin

The extended clause of a class declaration is optional:

```
ExtendedClass ::=
  ( "extended" LocalId )?
```

Week 9 end

Generic type parameters and type arguments are optional.

```
TypeParams ::=
  ( "[" TypeVar ","..."," TypeVar "]" )?

TypeArgs ::=
  ( "[" Type "," ... "," Type "]" )?
```

Week 8 end

5.4 Removing complex expressions

We can strip complex expressions out of programs using the following definitions:

let X = $E_0.m$ (E_1, …, E_n); B
  $\stackrel{\text{def}}{=}$ let tmp = E_0; let X = tmp.m (E_1, …, E_n); B

let X = $V_0.m$ (V_1, …, V_m, E_0, E_1, …, E_n); B

$\stackrel{\text{def}}{=}$ let tmp = E_0; let X = V_0.m (V_1, …, V_m, tmp, E_1, …, E_n); B

let X = E_0::T.m (E_1, …, E_n); B
$\stackrel{\text{def}}{=}$ let tmp = E_0; let X = tmp::T.m (E_1, …, E_n); B

let X = V_0::T.m (V_1, …, V_m, E_0, E_1, …, E_n); B
$\stackrel{\text{def}}{=}$ let tmp = E_0; let X = V_0::T.m (V_1, …, V_m, tmp, E_1, …, E_n); B

let X = p E; B
$\stackrel{\text{def}}{=}$ let X = E.prefix$p$ (); B

let X = E_1 i E_2; B
$\stackrel{\text{def}}{=}$ let X = E_1.infix$i$ (E_2); B

let X = E.f; B
$\stackrel{\text{def}}{=}$ let tmp = E; let X = tmp.f; B

let X = E_0.f := E_1; B
$\stackrel{\text{def}}{=}$ let tmp = E_0; let X = tmp.f := E_1; B;

let X = V_0.f := E_1; B
$\stackrel{\text{def}}{=}$ let tmp = E_1; let X = V_0.f := tmp; B;

E; B
$\stackrel{\text{def}}{=}$ let tmp = E; B

if (E) { B_1 } else { B_2 }
$\stackrel{\text{def}}{=}$ let tmp = E; if (tmp) { B_1 } else { B_2 }

return E;
$\stackrel{\text{def}}{=}$ let tmp = E; return tmp;

where:

- B ranges over Block
- E ranges over Exp
- V ranges over Val
- X ranges over Var
- f ranges over LocalId
- i ranges over InfixId
- m ranges over MethodId
- p ranges over PrefixId
- tmp is a freshly generated temporary variable.

For example:

```
import "Base.hob";

thread Main {
  let x : Integer = 1 + 2 + 3;
  Out.println ("x = " + $x);
}
```

is desugared to:

```
import "Base.hob";

thread Main {
  let tmp1 = 1.infix+ (2);
  let x : Integer = tmp1.infix+ (3);
  let tmp2 = x.prefix$ ();
  let tmp3 : String = "x = ".infix+ (tmp2);
  let tmp4 : Void = Out.println (tmp3);
  return Nothing;
}
```

Note that this desugaring process results in a left-to-right evaluation order.

5.5 Removing complex if blocks

We remove complex if blocks, generating multiple simple if blocks in their place:

if (E) { B }
$\stackrel{\text{def}}{=}$ if (E) { B } else { }

if (E_1) { B_1 } else if (E_2) { B_2 } else { B_3 }
$\stackrel{\text{def}}{=}$ if (E_1) { B_1 } else { if (E_2) { B_2 } else { B_3 } }

For example

```
if (x) { foo (); }
else if (y) { bar (); }
else if (z) { baz (); }
```

is desugared to:

```
if (x) {
  foo ();
} else {
  if (y) {
    bar ();
  } else {
    if (z) {
      baz ();
    } else {
    }
  }
}
```

5.6 Default values for optional attributes

The default ReturnBlock is return Nothing;.

The default ThisVar is this..

The default ReturnType is Void.

The default Mutability is immutable. Week 6 begin

The default ExtendsClass is extends Object.

The default ImplementsInterfaces is implements.

The default ExtendsInterfaces is extends. Week 9 begin

The default ExtendedClass is extended myType. Week 9 end

The default TypeParams is [].

The default TypeArgs is []. Week 8 end

For example:

```
class Foo {
  field x : Integer;
  method bar () {}
}
```

is syntax sugar for:

Week 8 begin

```
class Foo[] extends Object[] implements {
  immutable field x : Integer[];
  method this.bar () : Void[] { return Nothing[]; }
}
```

Week 8 end

6 Native Classes

In order to provide access to the operating system, and for base types such as integers and booleans, Hobbes supports *native class* and *native object* declarations.

```
Dec ::=
  {as before} |
  DecNativeClass |
  DecNativeObject

DecNativeClass ::=
  "native" "class" GlobalId "{"
    MethodAbstract ... MethodAbstract
  "}"

DecNativeObject ::=
  "native" "object" GlobalId ":" Type ";"
```

As syntax sugar, surface Hobbes allows native methods and objects to have bodies containing arbitrary brace-balanced material, typically this consists of assembly instructions for an appropriate architecture.

7 Example Programs

7.1 Base

```
native class Boolean {

  method prefix $ () : String {
    if (this) { return "True"; } else { return "False"; }
  }

  method prefix ! () : Boolean {
    if (this) { return False; } else { return True; }
  }

  method infix & (x : Boolean) : Boolean {
    if (this) { return x; } else { return False; }
  }

  method infix | (x : Boolean) : Boolean {
    if (this) { return True; } else { return x; }
  }

}

native class Integer {

  /*
   * Note that there's nothing special about the Integer class
   * in Hobbes, it's just another native class.
   */

  method infix + (x : Integer) : Integer {
    return native Integer (this, x) {
      register %1;
      hint %1 = #0;
      hint %1 = #1;
      asm "movl #1, %1";
      asm "addl #2, %1";
      asm "movl %1, #0";
    };
  }
  method infix - (x : Integer) : Integer {
    return native Integer (this, x) {
      register %1;
      hint %1 = #0;
      hint %1 = #1;
      asm "movl #1, %1";
      asm "subl #2, %1";
      asm "movl %1, #0";
    };
  }
  method infix * (x : Integer) : Integer {
    return native Integer (this, x) {
      register %1;
      hint %1 = #0;
      hint %1 = #1;
```

```
    asm "movl #1, %1";
    asm "imull #2, %1";
    asm "movl %1, #0";
  };
}
method infix == (x : Integer) : Boolean {
  return native Boolean (this, x, False, True) {
    register %1, %2;
    hint %1 = #0;
    hint %1 = #1;
    asm "movl #1, %1";
    asm "cmpl #2, %1";
    asm "movl #3, %1";
    asm "movl #4, %2";
    asm "cmove %2, %1";
    asm "movl %1, #0";
  };
}
method infix <= (x : Integer) : Boolean {
  return native Boolean (this, x, False, True) {
    register %1, %2;
    hint %1 = #0;
    hint %1 = #1;
    asm "movl #1, %1";
    asm "cmpl #2, %1";
    asm "movl #3, %1";
    asm "movl #4, %2";
    asm "cmovle %2, %1";
    asm "movl %1, #0";
  };
}
method infix < (x : Integer) : Boolean {
  return native Boolean (this, x, False, True) {
    register %1, %2;
    hint %1 = #0;
    hint %1 = #1;
    asm "movl #1, %1";
    asm "cmpl #2, %1";
    asm "movl #3, %1";
    asm "movl #4, %2";
    asm "cmovl %2, %1";
    asm "movl %1, #0";
  };
}
method infix != (x : Integer) : Boolean {
  return native Boolean (this, x, False, True) {
    register %1, %2;
    hint %1 = #0;
    hint %1 = #1;
    asm "movl #1, %1";
    asm "cmpl #2, %1";
    asm "movl #3, %1";
    asm "movl #4, %2";
    asm "cmovne %2, %1";
    asm "movl %1, #0";
  };
}
```

```
  method infix > (x : Integer) : Boolean {
    return x < this;
  }
  method infix >= (x : Integer) : Boolean {
    return x <= this;
  }
  method prefix $ () : String {
    let result : String = native String "malloc" (15);
    native Integer "snprintf" (result, 15, "%d", this);
    return result;
  }
}

native class String {
  method infix + (x : String) : String {
    let thisSize : Integer = #this;
    let xSize : Integer = #x;
    if (thisSize == 0) { return x; }
    else if (xSize == 0) { return this; }
    else {
      let result : String = native String "malloc" (thisSize + xSize + 1);
      native String "strncpy" (result, this, thisSize + 1);
      native String "strncat" (result, x, xSize + 1);
      return result;
    }
  }
  method prefix # () : Integer {
    return native Integer "strlen" (this);
  }
}

native class Void {
}

native class Thread {
  method infix == (other : Thread) : Boolean {
    return native Boolean (this, x, False, True) {
      register %1, %2;
      hint %1 = #0;
      hint %1 = #1;
      asm "movl #1, %1";
      asm "cmpl #2, %1";
      asm "movl #3, %1";
      asm "movl #4, %2";
      asm "cmove %2, %1";
      asm "movl %1, #0";
    };
  }
  method infix != (other : Thread) : Boolean {
    return native Boolean (this, x, False, True) {
      register %1, %2;
      hint %1 = #0;
      hint %1 = #1;
      asm "movl #1, %1";
      asm "cmpl #2, %1";
      asm "movl #3, %1";
      asm "movl #4, %2";
```

```
      asm "cmovne %2, %1";
      asm "movl %1, #0";
    };
  }
}

native class StdOutStream {
  method print (msg : String) {
    native Integer "printf" ("%s", msg);
  }
  method println (msg : String) {
    native Integer "printf" ("%s\n", msg);
  }
  method printInteger (msg : Integer) {
    native Integer "printf" ("%d", msg);
  }
}

native class ErrorClass {
  method infix ! (msg : String) : Empty {
    native Integer "printf" ("%s\n", msg);
    return native Empty "exit" (1);
  }
}

native class Empty {
}

native object Error : ErrorClass { 0 }

native object Out : StdOutStream { 0 }

native object Nothing : Void { 0 }

native object True : Boolean { 1 }
native object False : Boolean { 0 }
```

7.2 Simple arithmetic

```
import "Base.hob";

thread Main {
  Out.println ("1 + 2 = " + $(1+2));
}
```

7.3 More simple arithmetic

```
import "Base.hob";

thread Main {
  let x : Integer = 1 + 2;
  let y : Integer = x * 3;
  Out.println ($x + "+" + $y + " = " + $(x + y));
  Out.println ($x + "-" + $y + " = " + $(x - y));
  Out.println ($x + "*" + $y + " = " + $(x * y));
```

```
  Out.println ($x + "<=" + $y + " = " + $(x <= y));
  Out.println ($x + "<" + $y + " = " + $(x < y));
  Out.println ($x + ">=" + $y + " = " + $(x >= y));
  Out.println ($x + ">" + $y + " = " + $(x > y));
  Out.println ($x + "==" + $y + " = " + $(x == y));
  Out.println ($x + "!=" + $y + " = " + $(x != y));
}
```

7.4 Variable rebinding

```
import "Base.hob";

thread Main {
  // What happens if we redefine variables?
  let x : Integer = 1 + 2;
  let x : Integer = x + 3;
  Out.println ("x = " + $x);
}
```

7.5 Syntax sugar

```
import "Base.hob";

thread Main {
  let x : Integer = 1 + 2 + 3;
  Out.println ("x = " + $x);
}
```

7.6 Removal of syntax sugar

```
import "Base.hob";

thread Main {
  let tmp1 = 1.infix+ (2);
  let x : Integer = tmp1.infix+ (3);
  let tmp2 = x.prefix$ ();
  let tmp3 : String = "x = ".infix+ (tmp2);
  let tmp4 : Void = Out.println (tmp3);
  return Nothing;
}
```

Week 5 begin

7.7 Factorial

```
import "Base.hob";

class Factorial {
  // The non-tail-recursive implementation of factorial.
  method fact (n : Integer) : Integer {
    if (n <= 1) {
      return 1;
    } else {
      return n * this.fact (n-1);
```

```
    }
  }
}
object F : Factorial {
}
thread Main {
  let x : Integer = F.fact (5);
  Out.println ("5! = " + $x);
}
```

7.8 Integer references

```
import "Base.hob";

class IntRef {
  mutable field contents : Integer;
  method prefix * () : Integer { return this.contents; }
  method infix <- (n : Integer) { this.contents := n; }
}
class IntRefFactory {
  method build (n : Integer) : IntRef { return new IntRef { contents=n }; }
}
object Factory : IntRefFactory {}
thread Main {
  let x : IntRef = Factory.build (17);
  Out.println ("Before: *x = " + $(*x));
  x <- (*x + 5);
  Out.println ("After: *x = " + $(*x));
}
```

Week 6 begin

7.9 String lists

```
import "Base.hob";

class List {
  // Note that in "full" Hobbes we can make List an interface
  // rather than a class, which saves having lots of error
  // methods.
  method cons (hd : String) : List {
    let sz : Integer = this.size () + 1;
    return new ListCons { hd=hd, tl=this, sz=sz };
  }
  method reverse () : List { return this.revApp (Empty); }
  // If we had a proper exception mechanism we could do
  // a better job of error reporting!
  method head () : String { return Error! "Oops head!"; }
  method tail () : List { return Error! "Oops tail!"; }
  method size () : Integer { return Error! "Oops size!"; }
  method revApp (l : List) : List { return Error! "Oops revApp!"; }
  method prefix $ () : String { return Error! "Oops $!"; }
}

class ListEmpty extends List {
  method size () : Integer { return 0; }
```

```
  method revApp (l : List) : List { return l; }
  method prefix $ () : String { return "EMPTY"; }
}

class ListCons extends List {
  field hd : String;
  field tl : List;
  field sz : Integer;
  method tail () : List { return this.tl; }
  method head () : String { return this.hd; }
  method size () : Integer { return this.sz; }
  method revApp (l : List) : List { return this.tl.revApp (l.cons (this.hd)); }
  method prefix $ () : String { return this.hd + "::" + $this.tl; }
}

object Empty : ListEmpty {
}

thread Main {
  let l : List = Empty.cons ("Fred").cons ("Wilma").cons ("Barney").cons ("Betty");
  let m : List = l.reverse ();
  Out.println ("l = " + $l);
  Out.println ("m = " + $m);
}
```

7.10 Shape lists

```
import "Base.hob";

// An example of many patterns in action, to check that
// we are doing OO "properly" :-)

class Shape {
  field isRed : Boolean;
  method isRed () : Boolean { return this.isRed; }
  method accept (v : ShapeVisitor) { v.visitShape (this); }
}
class Rectangle extends Shape {
  method isSquare () : Boolean { return False; }
  method accept (v : RectangleVisitor) { v.visitRectangle (this); }
}
class Square extends Rectangle {
  method isSquare () : Boolean { return True; }
  method accept (v : SquareVisitor) { v.visitSquare (this); }
}

class SquareVisitor {
  method visitSquare (x : Square) { return Error! "Oops"; }
}
class RectangleVisitor extends SquareVisitor {
  method visitRectangle (x : Rectangle) { return Error! "Oops"; }
}
class ShapeVisitor extends RectangleVisitor {
  method visitShape (x : Shape) { return Error! "Oops"; }
}
```

```
class ShapeList {
  method head () : Shape { return Error! "Oops"; }
  method tail () : ShapeList { return Error! "Oops"; }
  method size () : Integer { return Error! "Oops"; }
  method iterate (v : ShapeVisitor) { return Error! "Oops"; }
  method consShape (x : Shape): ShapeList { return new ConsShapeList { hd=x, tl=this }; }
}
class RectangleList extends ShapeList {
  method head () : Rectangle { return Error! "Oops"; }
  method tail () : RectangleList { return Error! "Oops"; }
  method iterate (v : RectangleVisitor) { return Error! "Oops"; }
  method consRectangle (x : Rectangle): RectangleList { return new ConsRectangleList { hd=x, tl=this }; }
}
class EmptyShapeList extends ShapeList {
  method size () : Integer { return 0; }
  method iterate (v : ShapeVisitor) { }
}
class ConsShapeList extends ShapeList {
  field hd : Shape;
  field tl : ShapeList;
  method head () : Shape { return this.hd; }
  method tail () : ShapeList { return this.tl; }
  method size () : Integer { return this.tl.size () + 1; }
  method iterate (v : ShapeVisitor) { this.hd.accept (v); this.tl.iterate (v); }
}
// It would be nice if EmptyRectangleList <: EmptyShapeList but that requires multiple inheritance
class EmptyRectangleList extends RectangleList {
  method size () : Integer { return 0; }
  method iterate (v : RectangleVisitor) { }
}
// It would be nice if ConsRectangleList <: ConsShapeList but that requires multiple inheritance
class ConsRectangleList extends RectangleList {
  field hd : Rectangle;
  field tl : RectangleList;
  method head () : Rectangle { return this.hd; }
  method tail () : RectangleList { return this.tl; }
  method size () : Integer { return this.tl.size () + 1; }
  method iterate (v : RectangleVisitor) {
    this.hd.accept (v);
    this.tl.iterate (v);
  }
}
class CountShapes extends ShapeVisitor {
  mutable field rectangles : Integer;
  mutable field squares : Integer;
  mutable field total : Integer;
  method visitSquare (x : Square) {
    this.squares := this.squares + 1;
    this.total := this.total + 1;
  }
  method visitRectangle (x : Rectangle) {
    this.rectangles := this.rectangles + 1;
    this.total := this.total + 1;
  }
  method visitShape (x : Shape) {
    this.total := this.total + 1;
  }
```

```
  method go (l : ShapeList) {
    this.rectangles := 0;
    this.squares := 0;
    this.total := 0;
    l.iterate (this);
    Out.println ("number of rectangles = " + $this.rectangles);
    Out.println ("number of squares = " + $this.squares);
    Out.println ("total = " + $this.total);
  }
}
class PrintShapes extends ShapeVisitor {
  method visitSquare (x : Square) { Out.print ("square "); }
  method visitRectangle (x : Rectangle) { Out.print ("rectangle "); }
  method visitShape (x : Shape) { Out.print ("shape "); }
}
object Tri : Shape { isRed = False }
object Rect : Rectangle { isRed = True }
object Sq : Square { isRed = False }
object Nil : EmptyRectangleList { }
object Counter : CountShapes { rectangles = 0, squares = 0, total = 0 }
object Printer : PrintShapes { }
thread Main {
  let xs : ShapeList = Nil.consShape (Tri).consShape (Rect).consShape (Sq);
  let ys : RectangleList = Nil.consRectangle (Rect).consRectangle (Sq).consRectangle (Sq);
  Counter.go (xs);
  Counter.go (ys);
}
```

Week 8 begin

7.11 Generic lists

```
import "Base.hob";
interface List [type a] {
  method cons (hd : a) : List[a];
  method reverse () : List[a];
  method revApp (l : List[a]) : List[a];
}
class ListEmpty [type a] implements List[a] {
  method cons (hd : a) : List[a] { return new ListCons[a] {hd=hd,tl=this}; }
  method reverse () : List[a] { return this; }
  method revApp (l : List[a]) : List[a] { return l; }
}
class ListCons [type a] implements List[a] {
  field hd : a;
  field tl : List[a];
  method cons (hd : a) : List[a] { return new ListCons[a] {hd=hd,tl=this}; }
  method reverse () : List[a] { return this.revApp (new ListEmpty[a] {}); }
  method revApp (l : List[a]) : List[a] { return this.tl.revApp (l.cons (this.hd)); }
}
object EmptyString : ListEmpty[String] {}
object EmptyInteger : ListEmpty[Integer] {}
thread Main {
  EmptyString.cons ("Fred").cons ("Wilma").reverse ();
  EmptyInteger.cons (1).cons (2).reverse ();
}
```

Week 9 begin

7.12 Generic sorted lists

Week 9 end