# Thorn Specification

Purdue University/IBM T.J. Watson Research

*Initial Draft of Merged Spec 2009-04-03*

## Contents

1. **Classes and Objects**  
   1.1 Class Declaration .................................................. 3  
   1.2 Class Members ..................................................... 4  
   1.3 Field Declarations ................................................ 4  
   1.4 Method Declarations ................................................ 4  
   1.5 Constructors ...................................................... 6  

2. **Object Definitions** ................................................. 7  
   2.1 Object Field Declarations ........................................... 7  
   2.2 Anonymous Objects .................................................. 8  

3. **Open Declarations** .................................................. 8  

4. **Import Declarations** ............................................... 9  

5. **Expressions and Statements** ....................................... 9  
   5.1 If/Unless ............................................................. 9  
   5.2 Loops ............................................................... 11  
   5.3 Pattern Matching .................................................... 12  

6. **Exception Handling** ................................................. 12  

7. **Expressions** .......................................................... 14  
   7.1 Method Calls ......................................................... 14  
   7.2 Closures ............................................................. 14  
   7.3 Lists ................................................................. 15  
   7.4 Maps ................................................................. 15  
   7.5 ListComprehensions ................................................... 16  
   7.6 MapComprehensions ................................................... 16  
   7.7 Generators ............................................................ 17  
   7.8 Return ............................................................... 17  

8. **Operators** ........................................................... 17  

9. **Lexical Structure** .................................................... 18  

10. **Component** ........................................................ 18  

11. **Lacks Good Placement** ............................................. 18  

12. **Lost** ............................................................... 18  

13. **Th vs. Thorn** ....................................................... 18
To Do for Merged Spec

1. (JF) Explain how marshalling/demarshalling of functional values (methods, funs, objects, classes) works.

2. (JF) Explain how marshalling of other types works (based on flattening into records, perhaps?)

3. (JF) Explain how high-level communications are translated into low-level send/receives (unless we want to assert that they don’t interoperate).

4. (JF) Syntax for methods: fun or method?

5. (JF) Explain how top-level components get started up

6. (JF) Explain how environment for top-level programs gets defined

7. (JF) Are we happy with current treatment of Public/Private in modules and as annotations?

8. (JF) Where can we put type annotations (syntactically)?

9. (JF) Can type annotations be used for absolutely anything we want, or should they be restricted to “type-like” constraints? If the latter, does this conflict with using type annotations for Public/Private?

10. (JF) Clearly explain how classes and built-in values are related; e.g.:
   - are records or anonymous objects elements of a class?
   - if so, is there one class? an infinite set? can we name the classes? can we extend them?
   - how are classes for built-in types related (i.e., is there a hierarchy)?
   - how/where are methods for built-in types defined? Is there special syntax for defining the public fields/methods for built-in classes (or types, if different)?
   - can we define at least some methods for built-in classes in some kind of header file, as opposed to baking them all into the compiler?

11. (JF) Come up with a convincing story on how numeric types work.
   - Do we have a hierarchy of numerical types, or a flat space?
   - Are operations on numeric types explainable as ordinary methods, or should they be special?

12. (JF) Do we allow quoted identifiers?

13. (JF) What is character set for identifiers?

14. (JF) Are we happy with checking “confinement” dynamically before marshalling? Is there a reasonable static alternative that’s not inconsistent with a dynamic language?

15. (JF) Is it OK for tables to have methods?

16. (JF) Need to stabilize pattern matching / extractor syntax-semantics.

17. (JF) Include tuples as sugar for records?

18. (BB) Make sure that the floating point ops and libraries are adequate.

19. (BB) It might be best if the module system syntax agreed with the syntax of the rest of the language; e.g., using braces to enclose module bodies.

20. (JF) Should we define a class of identifiers which is reserved for system use?

21. (JF) Use [] or () consistently in subscripting contexts.

22. (JF) Explain limitations on parameter forwarding for class declarations.
1 Classes and Objects

1.1 Class Declaration

\[
\text{ClassDecl} ::= \text{class} \ QualName \ [\ [\ TypeParams \ ] \]
\[\ (\ Formals \ )\ ]\ [\ Extends \ ]
\[\ [\ Constraints\ ]\ (\ ClassBody \ |\ Terminator\ )\]
\]

\[
\text{Extends} ::= \text{extends} \ SuperCall \ (,\ SuperCall)^*
\]

\[
\text{SuperCall} ::= \ QualName \ [\ [\ TypeArgs \ ]\ ]\ [\ SimpleCallSuffix\ ]
\]

\[
\text{Formals} ::= [\ \text{Formal} (,\ \text{Formal})^*\ ]
\]

\[
\text{Formal} ::= [\ \text{val}\ |\ \text{var}\ ]\ \text{Id}
\]

\[
\text{Terminator} ::= ;\ |\ }\ |\ \text{eof}
\]

Examples:
1. `class Foo(val a, var b) {}`
2. `class Bar {}`
3. `class FooBar(x, y) extends Bar, Foo(y,x) {}`

Discussion

A class is a template for creating objects with an identically named type.

Syntactically, a class starts with the reserved word `class` followed by a name, an optional list of type parameters, an optional list of formal parameters, an optional list of parent classes, an optional list of type constraints, and a class body.

The type parameter list is used to parameterize a class.

The formals list is a lightweight syntax for defining fields of a class together with default constructors and extractors. Each formal appearing in the list specifies a corresponding field. Formals can be annotated `val` (name binding similar to Java’s final) or `var` (a regular variable). Omitted annotations default to `val`.

A class with a non-empty list of formals will be provided with a default constructor with the same number of arguments as formals that initializes the fields with the value provided, in the order specified in the class declaration.

As an example, the following classes are equivalent:

1. `class C(v, var x) {}`
2. `class C { val v; var x; C(a,b) { v = a; x := b } }`

Note that assignment uses `:=` rather than `=` which is name binding.

Formals passed as arguments to parent classes will be “forwarded” to the appropriate constructors and no field will be created for them in the current class. Hence, the class on line 3 above only has fields `a` and `b` – `x` and `y` are forwarded to `Foo`.

For classes with a default constructor, an inverse default extractor method is automatically provided. This comes into play in pattern matching, and is deferred to then.

Default constructors and extractors are not provided if the class defines a constructor/extractor with the same number of arguments as forms that initializes the fields with the value provided, in the order specified in the class declaration.

Multiple Inheritance 

Thorn supports multiple inheritance but restricts its use to avoid diamond-ish problems. Notably, at most one superclass may be stateful (i.e., contain any kinds of fields) at any level of indirection. If two or more superclasses define a method with the same signature, the derived class must provide its own implementation and deal with the resolution explicitly through qualified super call(s).

Current Implementation Status

Other than support for parsing, type parameters is currently not implemented.
Known Bugs

TODO (need to write up something about the brittleness of super calls and inheritance here).

1.2 Class Members

| ClassBody | ::= | \{ ClassMember ( ; ClassMember )\* \} |
| ClassMember | ::= | FieldDef | FunDef | NewDef | MatchDef |
|            | | ObjectDef | Import ClassBody | Import Terminator |

Classes can contain fields, methods, constructors, extractors, objects and imports. These are discussed below.

Discussion

The semantics of nested objects in classes is that one object will be created for each instance of the enclosing class. This feature is implemented and seems to work.

1.3 Field Declarations

| FieldDef | ::= | \[ val \] \[ var \] VarOptInit ( , VarOptInit )\* |
| VarOptInit | ::= | Id \[ Constraints \] [ = Exp ] |

Example:

```class Foo() { val a = 10; var b = 1, c = false; }
```

Discussion

The above snippet declares a class Foo with three variables, a, b and c. The first is a value, meaning the name a is bound to the value 10 in the class body (think constant or Java final). The latter two are regular variables. Initialising val or var declarations is optional. A val field that is not initialised on declaration must be initialised in all paths in all constructors of the class. An attempt at reassigning a val field or variable throws a run-time exception or a compiler error.

Shadowing

Just like in Java, Thorn has separate name-spaces for fields and methods.

Current Implementation Status

Missing: We do not check that a val field is initialised on all paths of all constructors.

Known Bugs

No known bugs.

1.4 Method Declarations

| FunDef | ::= | fun Id \[ TypeParams \] \[\{ Formals \}\] \[ Constraints \] FunBody |
|        | | fun Id \[ TypeParams \] FunCase ( \[ FunCase \]\*) |
| FunCase | ::= | ( Patterns ) \[ Where \] => Stm |

Examples:
class Foo() { fun bar(val x, var y) { x+y } }
class Bar() {
  fun myif($false, _, action) => action.eval()
  | ($true, action, _) => action.eval()
}

Discussion

The first line declares a class Foo with method bar that takes two arguments and return their sum. Note that the value of the method is the value of its last expression (also see return).

The second to fifth line declares a class Bar with a method with several implementation under a single name. It uses pattern matching to select which implementation to dispatch to. Pattern matching is discussed in detail in Section ??.

Overloading Thorn supports overloading based on arity (signature = name + arity, e.g., foo/5). We might decide to support overloading based on types (signature = name + type constraints) for the much anticipated typed dialect of Thorn, but that is TBD. Plus the story for calling such methods “families” from untyped Thorn is not clear.

A class may not have more than one method with the same signature. Also see the discussion on multiple inheritance in Section ??.

Shadowing Just like in Java, Thorn has separate name-spaces for fields and methods.

Nested Functions Hidden in the non-terminals in the grammar above, but more visible in grammar for constructors, Thorn supports nested functions. Thus, the following snippet is valid and returns 7 when executed on the REPL.

class Nest(val a) {
  fun foo() {
    fun bar() { a + 1 };
    bar()
  }
}
Nest(6).foo() # returns 7

Note that bar/1 is not a method but a function. While it has access to the val field a in the current receiver in foo/1 it is does not belong to the Nest instance. Consequently, replacing line 4 for this.bar() would create a compile-time error: “Method Nest.bar not found”.

Nested Objects Just as for functions, methods can have nested objects declarations. These objects are created with each stack frame (i.e., created anew for each call to the method) but their life time are unbounded and they may escape the method. Example:

class Nest() {
  fun foo() {
    object Bar { var f = false; }
    Bar
  }
}
val nest = Nest();
var bar = nest.foo();
thorn.System.println(bar.f); # prints false
bar.f := true;
thorn.System.println(bar.f); # prints true
bar := nest.foo();
thorn.System.println(bar.f); # prints false
Current Implementation Status

TODO

Known Bugs

No known bugs.

1.5 Constructors

Also see the discussion on default constructors in Section ??.

<table>
<thead>
<tr>
<th>NewDef</th>
<th>::=</th>
<th>Id [ [ TypeParams ] ][ ( Formals ) ][ Constraints ] NewBody</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewBody</td>
<td>::=</td>
<td>[ = ] { [ NewBlockBody ] } = ConstructorStm Terminator</td>
</tr>
<tr>
<td>NewBlockBody</td>
<td>::=</td>
<td>(BlockDef;</td>
</tr>
<tr>
<td>ConstructorStm</td>
<td>::=</td>
<td>SuperCallExp</td>
</tr>
<tr>
<td>BlockDef</td>
<td>::=</td>
<td>LocalDef</td>
</tr>
</tbody>
</table>

Example:

```plaintext
1 class Foo(var a) {
2   Foo(val b) { a := b }
3   Foo(val b, c) = this(b + c); # semantically equivalent to { this(b + c) }
4 }
```

Discussion

The example class above has two constructors, Foo/1 and Foo/2. The first is identical (modulo variable names) to the default constructor that would otherwise have been automagically inserted by the compiler from the class’ formals declaration.

The second constructor exemplifies chaining constructor calls (this(Exp)). It also shows that Thorn allows omitting the brackets when a function body is a single expression.

Current Implementation Status

TODO (Need to revisit the constructor design—seems like there is no limit in what they can do anymore as Stm is valid at every position. Is there a good enough syntactic restriction to make constructors more safe or not?)

Known Bugs

No known bugs.
2 Object Definitions

ObjectDef ::= \texttt{object QualName [ Extends ] [ Constraints ]}
\hspace{1em} ObjectBody \hspace{1em} \texttt{Terminator}

ObjectDef ::= \texttt{object ObjectDefHeaders Terminator}

ObjectDefHeaders ::= ObjectDefHeader (, ObjectDefHeader)*

ObjectDefHeader ::= Id [ Extends ] [ Constraints ]

ObjectBody ::= { [ ObjectMember ( ; ObjectMember )* ] } Correct?

ObjectMember ::= ObjectFieldDef | FunDef |
\hspace{1em} MatchDef \hspace{1em} ObjectDef \hspace{1em} OpenDef \hspace{1em} ClassDef |
\hspace{1em} Import ObjectBody \hspace{1em} \texttt{Import Terminator} |

Examples:

1. \texttt{object Foo;}
2. \texttt{object Bar \{ a = 1; val b = 1; var c; \}}
3. \texttt{class A(x); object FooBar extends A(1) \{ fun foo() = x; \};}

Discussion

An object definition is syntactically very similar to a class definition except the \texttt{object} keyword, but while a class is a template describing its instances the object definition is self-describing. There is no dynamic instantiation operation for objects. Instead objects are singleton entities created and initialized statically. Objects may be defined in different surrounding scopes. Top level objects are called stable objects. These are globally accessible singleton objects (used much in the same way static is used in Java). However, objects may also be defined as members of classes and functions. In these cases a new object will be created for each class instance and stack frame, respectively. Stable objects may be opened and extended using the \texttt{open} keyword (see section below).

TODO: Need to say something about import here.

Objects may extend classes, but cannot be be extended themselves.

Current Implementation Status

Seems to be complete.

Known Bugs

Perhaps the bug described in Anonymous Objects really belongs here?

2.1 Object Field Declarations

ObjectFieldDef ::= \texttt{\texttt{var} VarOptInit (, VarOptInit)*}
\hspace{1em} \texttt{val} ValInit (, ValInit)*
\hspace{1em} \texttt{ValInit (, ValInit)*}

ValInit ::= Id [ Constraints ] [ = Exp ]

As objects are initialized automatically (statically) and are singleton entities they have no constructors. It is therefore required that all value fields are initialized as they are declared.

Current Implementation Status

Complete.
Known Bugs

No known bugs.

2.2 Anonymous Objects

\[ AnonObject ::= object [ Extends ] [ Constraints ] ObjectBody \]

An object definition with no provided name is an expression and can be used without restriction in any expression context. Anonymous objects have identity and can thus be distinguished from each other.

Examples:

1. object Foo { fun bar(x) { object { val z = 1; }; } }  
2. object {} != object {}

Current Implementation Status

Seems to be complete.

Known Bugs

1. object Foo { val y = 1; fun bar(x) { object { val z = x+y; }; }; }; Foo.bar(2).z

This crashes with a java.lang.VerifyError.

stable objects open and import variable declarations and shadowing method shadowing and super calls, @-specified calls Note that super calls must be specified when there is multiple inheritance, even if the method can only come from one parent anonymous objects

talk about namespaces

3 Open Declarations

\[ OpenDecl ::= open QualName \]

Discussion

An open declaration states that the remainder of the scope is compiled as if it were contained within the body of the named singleton object. A compilation unit may begin with an open declaration. If the open declaration is absent, the compilation unit is implicitly open this, where this is a reference to the current component.

QualName can be a path where all names refer to singleton objects.

Current Implementation Status

Completed.

Known Bugs

No known bugs.
4 Import Declarations

An import declaration adds names of objects, classes, or class members to the current scope. The declaration `import QualName . Id` allows the name `QualName . Id` to be referred to as `Id`. If the `import` has an optional as clause, the given short name `Id` can be used to refer to the fully-qualified `QualName . Id`. The declaration `import QualName . { NameWOptRename ( , NameWOptRename )* }` adds the named members of `QualName` to the scope using their short names (with optional renaming). The declaration `import QualName . *` adds all members of `QualName` to the scope using their short names and uses the optional except clause `except { Id ( , Id)* }` to exclude those listed in the brackets.

The import statement:

```plaintext
import Foo.Bar { ... }
```

imports `Foo.Bar` under the name `Bar` for the scope of the block.

Current Implementation Status

The current implementation support `import Id = QualName` as an alternative to `import QualName as Id`. Tobias would like to drop this—there is no point in having two different syntaxes for the same thing.

Known Bugs

No known bugs.

5 Expressions and Statements

5.1 If/Unless

```plaintext
If ::= ( if | unless ) ( Exp ) Exp [ else Exp ]
```
Discussion

An **if (unless)** expressions conditionally evaluates an expression. Any value other than true is treated as false.

The **if** expression `if (e₁) e₂` first evaluates the condition `e₁`; if `true`, `e₂` is evaluated and the **if** expression evaluates to the result of `e₂`; if `false`, `e₂` is skipped and the **if** expression evaluates to unit (see known bugs below).

The **if** expression `if (e₁) e₂ else e₃` first evaluates the condition `e₁`; if `true`, `e₂` is evaluated and the **if** expression evaluates to the result of `e₂`; if `false`, `e₃` is evaluated and the **if** expression evaluates to the result of `e₃`.

The **unless** expressions are sugar for **if**-expressions with negated conditions (`!e₁` in both cases above).

Variable declarations in conditionals

Declaring variables in boolean expressions of conditionals is supported:

```plaintext
if (x,y=Person.findByName("Elvis"))
    print(x);  # entered iff y is true above, x and y are defined here
else
    throw new HasLeftTheBuildingException();  # x and y are defined here
```

The first line above declares `x` and `y` for the remainder of the current scope. If `Person.findByName()` does not return two values, an exception is thrown which will exit the current scope, making the undefined variables inaccessible.

Note that the right-most value is the only one tested by the if-statement (see current implementation status and known bugs below).

Current Implementation Status

Currently, there seems to be no support for trailing if/unless tests, like

```plaintext
import thorn.System.exit as die;
die(-1) unless (success);  # Common PERL pattern
```

Also, support for multiple return values is not working properly. `return 1,2` boxes the values into `[1,2]`, but unpacking that array into multiple variables is not supported.

Known Bugs

Return value of an if-expression whose conditional evaluates to false is `()`, “unit”. A better design (Tobias thinks) should be to return null. Example:

```plaintext
var x = if (false) 2;  # assigns () to x
```

Last, the story on multiple return values is shaky. Line one defines a function `f/0` used on the subsequent lines.

```plaintext
val f = fun () => { 1,2 }; f();  # returns [1, 2]
var a,b = f();  # works, a = null, b = [1,2]
# Unexpected character ','; expected '=' while parsing 'ValInit'
val a,b = f();
# Parses and executes, but a and b are undefined
a,b = f();  # a,b = 1,2 has same semantics
```

Lines 2, 5 and 8 should clearly be equivalent (modulo val/varness).
5.2 Loops

Thorn loops are optionally labelled in the standard fashion.

| Labelled | ::= | [ Id : ] Loop |
| Loop | ::= | While | DoWhile | ForEach |
| While | ::= | ( while | until ) ( Exp ) Stm |
| DoWhile | ::= | do Stm until CondExp |
| ForEach | ::= | for ( [ val | ForTake ( , ForTake )* ) Stm |
| ForTake | ::= | Id <- Exp |
| Break | ::= | break [ Id ][ ( if | unless ) CondExp ] |
| Continue | ::= | continue [ Id ][ ( if | unless ) CondExp ] |

While/Until and DoWhile Loops

A while (until) loop evaluate an expression repeatedly until a condition is false (true). Any value other than true is treated as false.

A while loop of the form while (e₁) e₂ first evaluates the expression e₁; if e₁ evaluates to true, e₂ is evaluated. Control then returns to e₁. If e₁ evaluates to false, the loop is exited. The value of the entire loop is the last value of e₁. Thus, if the loop exited normally, it evaluates to false. If it exited due to a break, its value is true.

An until loop until (e₁) e₂ is sugar for the while loop while (!e₁) e₂.

do e₁ while (e₂) first evaluates e₁, then evaluates e₂. If e₂ evaluates to true, the loop repeats at e₁. Its return value is the same as for while/until loops.

ForEach Loops

```plaintext
1  for (val a <- [1,2,3,4])
2    print(a);
```

A for loop iterates over the elements produced by a generator. Under the hood, the above invokes the iterator/0 method on the array and each iteration in the loop binds val a to the value returned by calling take/0 on the iterator which returns the next value and eventually null at which point the loop terminates. Return values for ForEach loops are the same as for while/until loops.

It is possible to iterate over several lists in the same loop. This has the same semantics as if the loops were nested. For example:

```plaintext
1  for (val a <- [1], b <- [2,3], c <- [4,5,6])
2    print("${a},${b},${c} ");
```

prints

```
{1,2,4} {1,2,5} {1,2,6} {1,3,4} {1,3,5} {1,3,6}
```

and has the same semantics as

```plaintext
1  for (val a <- [1])
2  for (val b <- [2,3])
3    for (val c <- [4,5,6])
4      print("${a},${b},${c} ");
```

Break and Continue

A break statement with a label immediately exits the enclosing loop with that label when evaluated. If the break is unlabeled, the innermost enclosing loop is exited.

A continue statement with a label immediately restarts the enclosing loop with that label when evaluated. If unlabeled, the innermost enclosing loop is restarted.

It is a static error if an unlabeled break or continue has no enclosing loop or if a labeled break or continue is not enclosed within a loop with that label.
Current Implementation Status

Finished.

Known Bugs

None.

5.3 Pattern Matching

\[
\begin{array}{ll}
\text{Where} & ::= \text{where } EqExp \\
\text{Patterns} & ::= \text{Pattern} \ (, \text{Pattern})* \\
\text{Pattern} & ::= [ \text{Identifier} - ] \text{CallPattern} \ (: \ : \text{CallPattern})* \\
\text{CallPattern} & ::= \text{Identifier} \ (\text{Patterns}) \\
& \mid \text{Literal} \\
& \mid \text{ListConstructorPattern} \\
& \mid \text{MapConstructorPattern} \\
& \mid \text{RecordPattern} \\
& \mid \text{Identifier} \\
& \mid (\text{Exp}) \\
\text{Cases} & ::= [ | ] \text{Case} \ [ ; ] ( | \text{Case} [ ; ] )* \\
\text{Case} & ::= [ \text{Formal} - ] \text{Pattern} \ [ \text{Where} ] => \text{Stm} \\
\text{Match} & ::= \text{match}(\text{Exp}) \ \text{Case} \\
& \mid \text{match}(\text{Exp}) \ \{ \text{Cases} \}
\end{array}
\]

Discussion

TODO: match order, variable scoping, exhaustiveness, type matching, extractors, other semantic

Current Implementation Status

Currently there is a Where rule added throughout the grammar (reflected above), but it is never used and unparseable. The right-hand side is simply where EqExp, and EqExp is not defined in the core grammar (wishful thinking?)

MapConstructorPattern and RecordPattern are unimplemented/obsolete. Perhaps the constructor should be into the Record plugin, along with an override for the PrimaryPattern rule.

Known Bugs

TODO.

6 Exception Handling

\[
\begin{array}{ll}
\text{Try} & ::= \text{try} \ (\text{CondExp} \mid \text{Block} \mid \text{Stm} [ ; ] ) \ \text{catch} \ (\text{Case} \mid \{ \text{Cases} \} ) \ [ \text{finally} \ \text{Stm} ] \\
& \mid \text{try} \ (\text{CondExp} \mid \text{Block} \mid \text{Stm} [ ; ] ) \ \text{finally} \ \text{Stm} \\
\text{Throw} & ::= \text{throw} \ \text{Exp} \ [ \text{ifunless} \mid \text{CondExp} ] \\
\text{Suppress} & ::= (\text{CondExp} \mid \text{Block}) \ \text{suppress} \ \text{Pattern} \mid \{ \text{Patterns} \} \\
\text{OrElse} & ::= \text{Exp} \ \text{orelse} \ \text{Exp}
\end{array}
\]

Examples:
try ... catch ThornRuntimeException => { ... }

try { throw 5 } catch { 5 => 7 | n ~ Int() => n }
try ... finally { file.close(); false }

// Suppress exceptions:
1/0 suppress DivisionByZero
... suppress { Foo | Bar }
throw bathWather unless bath.contains?(baby)
1/0 orelse -1

Discussion

Thorn try/catch/finally expressions work much like the Java equivalents with a number of important differences:

1. They are expressions and may thus be used e.g., on the RHS of an assignment or a return
2. They use pattern matching to determine how to handle an exception
3. Any object can be thrown as an exception, not just subclasses of ThornRuntimeException
4. The expression following a try mustn’t be a block

If no exception is thrown in the block or expression e following the try, the “return value” of the entire expression is the value of e. If an exception is thrown in e, the catch clause is searched top-down for a pattern matching the raised value. If a matching pattern is found, its attached block or expression is evaluated and the return value is the last expression in the block. If there is a finally clause, it is evaluated last and the return value of the entire expression is the last expression in the finally block.

As an example, the second example above evaluates to 7. Notably, the second pattern n ~ Int() matches if the thrown value is an integer (it will, however, not match 5 as that case is covered by a previous pattern) and binds the matching value to the name n defined for the scope of the case.

The expression e suppress P (where P is a pattern) is semantically equivalent to try e catch P => null, i.e., it suppresses any exception that matches P and instead returns null. Similarly, the expression e suppress { P1 | P2 } is semantically equivalent to try e catch { P1 => null | P2 => null }. As an example, the 4th example above evaluates to null.

The expression e1 orelse e2 first tries to evaluate e1. If e1 throws an exception, e2 is evaluated and the entire expression evaluates to the result of e2. Otherwise, the expression evaluates to the result of e1.

Note that Thorn does not use checked exceptions.

Internal Thorn Exceptions

The standard internal Thorn runtime exception is ThornRuntimeException. For now, the special exception CCE is used to signal calls to undefined methods. The exception ValErr is thrown as a result of an attempt at rebinding the value of a val variable or field. All these exceptions are internal and cannot be thrown by user code in Thorn land.

There is currently no Thorn exception hierarchy and current Thorn code either defines own exception classes or uses string literals.

Current Implementation Status

TODO. We need to work out a simple Thorn exception hierarchy.

Known Bugs

The keywords list of ThornGrammar.th defines a keyword rethrow but this is never used anywhere.
7 Expressions

7.1 Method Calls

```
| Call ::= | Call . Id TypeArgs SimpleCallSuffix |
| super @ QualName [ TypeArgs ] SimpleCallSuffix |
| super @ Id [ TypeArgs ] SimpleCallSuffix |
| super . Id [ TypeArgs ] SimpleCallSuffix |
| Call TypeArgs SimpleCallSuffix |
| Call . Id CallSuffix |
| Call CallSuffix |
| Call . Id What to do with this? |
| Primary What to do with this? |

| SimpleCallSuffix ::= | ( [ Exp ( , Exp )^* ] ) |
| CallSuffix ::= | SimpleCallSuffix |
| Comprehension |
| ListConstructor |
| MapConstructor |
| Record |
```

Examples:
1. `a.foo(1)`
2. `a.foo(a, 1)`
3. `foo().a.foo()`

Discussion

A method call has the general form of $e_0.m(a_0, \ldots, a_n)$. If $e_0$ evaluates to an object and $m$ is a method of that object then $m$ is invoked with that object as receiver and $a_0, \ldots, a_n$ are evaluated and their result passed as actual arguments. If $m$ is not a method of the receiver or the actual arguments do not match the arity of $m$ an exception is raised. If $m$ evaluates to an object, then its apply method is called with the result of evaluating $a_0, \ldots, a_n$ as actual arguments.

7.2 Closures

```
| ClosureExp ::= | fun Formal => Exp |
| fun ( [ Formal ( , Formal )^* ] ) => Exp |
| fun => Exp |
| fun [ [ TypeParams ] ] FunCase ( | FunCase )^* |
```

Examples:
1. `fun x => { x+2 }`
2. `(fun x => { x+2 })(1)`
3. `fun(x, y) => { x+y }`
4. `val a = fun x => { x+2 }; a(3)`

Discussion

Closures are anonymous functions. Closures differ from function definitions in that they are expressions and may be used wherever expressions are allowed. Closures are represented at runtime as objects with an apply method of the appropriate arity. A closure call $a(i)$ is sugar for $a.apply(i)$. 
### 7.3 Lists

| ListConstructor | ::= [ [ Exp , Exp ]* ] |
| ListConstructorPattern | ::= [ [ Pattern , Pattern ]* ] |
| TODO: Need to add Pattern somewhere... |

#### Examples:

1. `[1, 2, 3]`
2. `[a, 1, ‘Hello, World!’]`
3. `val a = [1, ‘s’, 2]; match(a) { [i1 ~ Int, String, i2 ~ Int] => i1+i2 | _ => -1}`
4. `val a = [1, 2, 3, 4, 5, 6]; a.get([2, 4])`

#### Discussion

List literals are enclosed in brackets.

Lists have a `get` method used to index into the collection. The list `get` method takes an integer index and returns the element at that index. The `get` method can also take a list or generator of indexes and returns a list of elements at those indexes.

Collections also have an `apply` method which simply invokes `get`. As with closures, `a(i)` is sugar for `a.apply(i)`.

Mutable collections may be updated with the `set` method. `a(i) = e` is sugar for `a.set(i,e)`. Like `apply`, the `set` method can take an index or a list or generator of indexes.

Strings are lists of characters and so support the `apply` and `set` methods.

#### Current Implementation Status

Seems to be complete.

#### Known Bugs

In the pattern matching example on Line 1 and 2 (below) the implementation seems to only care about the length of the list and not about the types.

```plaintext
val a = [1, ‘s’, 2];
match(a) { [i1 ~ Int, String, i2 ~ Int] => i1+i2 | _ => -1}
# returns 3, which is correct

val a = [1, ‘s’, 2];
match(a) { [i1 ~ String, String, i2 ~ String] => i1+i2 | _ => -1}
# returns 3, which is an error

val a = [1, ‘s’, 2];
match(a) { [i1 ~ String, String] => i1 | _ => -1}
# returns -1, which is correct
```

### 7.4 Maps

| MapConstructor | ::= [ [ MapElement , MapElement ]* ] |
| MapConstructorPattern | ::= [ [ Pattern => Pattern , Pattern => Pattern ]* ] |
| MapElement | ::= Exp => Exp |

#### Examples:

...
Map literals are enclosed in braces.

Maps have the same set of get, set and apply methods with similar semantics as lists. The difference is that the the keys can be arbitrary values and not just integers.

**Current Implementation Status**

MapConstructorPattern is not implemented.

**Known Bugs**

No known bugs.

### 7.5 ListComprehensions

```plaintext
ListComprehension ::= [[ [ Exp ( , Exp)* ] ]]
| [[ Exp SuchThat ]]
SuchThat ::= | ! Case [ Take | CondExp ( , Take | CondExp)* ]
```

Examples:

1. `[[ x | x <- [1, 2, 3] ]]
2. `[[ x | x <- [1, 2, 3], x != 2 ]]
3. `[[ x+y | x <- [1, 2, 3], y <- [7, 8, 9] ]]
4. `[[ x+y | x <- [3, 2, 1], y <- [7, 8, 9], x+y == 10]]`

**Discussion**

List comprehensions are lazily evaluated.

**Current Implementation Status**

Status of laziness unknown.

**Known Bugs**

Multiple takes doesn’t seem to work. The example below blows up.

1. `[[ x+y | x <- [1, 2, 3], y <- [7, 8, 9] ]]`

### 7.6 MapComprehensions

```plaintext
MapComprehension ::= {{ Exp => Exp SuchThat }}
```

Examples:

1. `{{ x => x.length() | x <- ["a", "aa", "aaa"] }}`
Discussion

The example above (Line 1) returns a map where each string is mapped to its length.

Current Implementation Status

Seems very shaky. The above example says x is undefined. We had the same problem with list comprehensions... maybe same solution?

Known Bugs

TODO

7.7 Generators

Generators are objects implementing the following interface:

```java
class Generator[T] {
    fun take(): T & abstract & throws(End);
}
```

Generator objects may be used as sources in take expressions. E.g. in the following example a generator is used as the source of the numbers printed to the console.

```java
class A {
    var i = 0;
    fun take() {
        if (i < 5) {
            i := i+1;
            i
        } else
            throw End;
    }
}

for (val x <- A()) {
    thorn.System.println(x);
}
```

Current Implementation Status

Unknown. Using a generator object in a list comprehension doesn’t seem to work.

Known Bugs

No known bugs.

7.8 Return

TODO

8 Operators

operators explained precedence
9  Lexical Structure
Comments Literals Strings and characters Integers and reals Booleans AST literals Identifiers Blocks

10  Component

11  Lacks Good Placement
A Thorn program Source and class paths? Compilation units Multiple return values

12  Lost
Access control modifiers Object identity as extending ID class or not? Equality Types, constraints and type system Plugin architecture

13  Th vs. Thorn

14  Standard Plugin Set
assert deepcopy binaryop metathorn

A  Complete Syntax
add standard plugin syntax

B  Equality and Identity

B.1  Identity and GlobalIdentity
As a default, objects in thorn have no object identity and the standard definition of equality is structural equality. Objects can get local identity by extending the class Identity. This class (logically) defines a field id with a unique identifier for the objects. Identity implements the method same?/1 that compares the identity of two objects. The identity is not accessible and cannot be stored somewhere.\footnote{As an implementation aside, this allows us to use the object’s address as its local identity, even in the presence of moving GC’s.}

The class GlobalIdentity is a subclass of Identity and gives an object a globally, weakly unique identifier. Weakly unique means that it may actually not be unique—exploding and restarting Thorn VM:s might accidentally reuse a unique identifier. The class GlobalIdentity might implement convenience methods like, getOriginatingNode/0 and overrides same?.

The reasons for splitting global and local identity into separate concerns are purely pragmatical. Guaranteeing globally unique ids is extremely difficult and will be performance degrading. Presumably, most objects will only need local identity, and the refactoring step for a class that needs to evolve into a distributed system is a simple one.

B.2  Structural Equality
Every thorn class implicitly defines a method ==/1 with slightly different behaviour depending on the nature of the class. The implicit ==/1 method cannot be overridden or redefined in subclasses, modulo Identity and GlobalIdentity.\footnote{Caveat: is GlobalIdentity is a subclass of Identity, will that not preclude subclassing a class with identity and “mixin in” a global identity. If so, then that’s bad.}
For Identity and GlobalIdentity, \( ==/1 \) returns true if the argument has the same unique id. For value classes and records, it returns true if the receiver is structural equivalent to the argument. For all other classes, it throws a `NoNotionOfEquality` error, a subclass of `NotImplemented`.

The structural equivalence is defined thus:

An object \( A \) is structurally equal to an object \( B \) iff \( \text{fields}(A) = \text{fields}(B) \) and for every field \( A.f, A.f \) is structurally equivalent to \( B.f \).

Notably, structural equivalence is symmetric, i.e., \( A == B \) implies \( B == A \) and vice versa.

**Discussion**

Library classes should be implemented using comparators. We need a standard comparator library (mixin style?).

For classes that have key fields, \( ==/1 \) only looks at the key fields.

**Current Implementation Status**

Not implemented yet. Moreover, there is no design draft of GlobalIdentity.

**Known Bugs**

None.

### C Records

Thorn records are simple, non-recursive (modulo circular definitions in objects contained in records) data structures. Records are objects without an identity, and with the exception of comparison operators, only define fields.

Records are created through record literals:

\[
\{a:1, b:2, c:3\}
\]

This creates a record with val fields \( a, b, \) and \( c \) with the values \( 1, 2 \) and \( 3 \) respectively. The record also has a “secret” val field, \( \$\text{fields} \) which contains the array \( ["a","b","c"] \). This array is used internally by the record plugin and should intended to be used by clients.

Fields of records are accessed through regular field accesses:

\[
\text{val record} = \{a:1, b:2, c:3\};
\]

\[
\text{record.a} \; \# \; 1
\]

Records can be arbitrarily nested:

\[
\text{val foo} = \{a:1, b:{\text{foo}:"Hello"}}\};
\]

\[
\text{val bar} = \{x:foo\}; \; \# \; \text{eq to} \; \{x:{a:1, b:{foo:"Hello"}}\}
\]

Furthermore, there is a convenience syntax for creating records from existing variables:

\[
\text{val x = 42;}
\]

\[
\{x\}; \; \# \; \text{eq to} \; \{x:x\} \; \text{or} \; \{x:42\}
\]

Notably, record structures are tree-shaped, i.e., the following record is not valid \( a:b:a \), and will result in a compile-time error.
C.1 Operations on Records

Records can be non-destructively combined:

```
val a = {a:1};
val b = {b:2};
val c = a + b;  # eq {a:1, b:2}
val d = c + {a:4,c:3};  # eq {a:4,b:2,c:3}
```

Notably, values in right-hand side records overwrite values in the record on the right hand side. The operation returns a new record and does not modify its operands.

Records also define an projection operation:

```
val record = {a:1, b:2, c:3};
val subset = record | a,b;  # eq. {a:1, b:2}
val subset = record | z;  # raises an error
```

In addition to the above, records support the following operations:

`==` structural equality as defined in Section B.2.

`<` s.t. \( r_1 < r_2 \) iff \( \text{dom}(r_1) \subseteq \text{dom}(r_2) \).

`>` s.t. \( r_1 > r_2 \) iff \( \text{dom}(r_2) \subseteq \text{dom}(r_1) \).

\( \overset{\text{s.t.}}{='} \) s.t. \( r_1 \overset{\text{r}}{=} r_2 \) iff \( r_1.f = r_2.f \) for all \( f \in \text{dom}(r_1) \) or \( r_2.f \) is a matching extractor pattern. Notably, \( c \overset{\text{r}}{=} \text{Record()} \) returns true for all records.

`toString` returns the record literal as a string, modulo whitespace and `toString/0` implementations of its contents.

Discussion

There is currently no convenient way of programmatically building up a record whose compartments are unknown at compile-time. For this, a dictionary/map is clearly a better data structure.

Current Implementation Status

Finished. The record plugin is called `RecordPlugin` and is enabled in `Bootstrap.th`.

Known Bugs

None.

D Value Classes, Pure Methods and Constants

This section introduces three new terms: value classes, pure methods and constants.

- A variable or field is a constant iff
  - it is a val field
  - it is initialized from a value class, i.e., (effectively) has a value type

- A method or constructor is pure iff
  - All fields it accesses are constants
  - All methods it overrides are pure
  - It only calls pure methods or constructors

- A class is a value class iff
  - All its fields are constants
  - All its methods are pure
Discussion

The current implementation of pure methods creates a duplicate copy for every method in a class prefixed Pure__. If the method is pure, all method calls of the bodies of both methods are translated into calling the Pure__ method. If the method was not pure, the body of the duplicate copy is replaced with throw CallToImpureMethodFromPureContext().

In this implementation, the constraints in the above list are automagically satified, modulo fields being val and public field accesses, which is now deprecated.

Nevertheless, the value class plugin requires the following to hold for a class with a value constraint:

1. All (implicit) fields are val fields
2. All (implicit) fields have value constraints
3. All methods have a pure constraint
4. All method parameters and return have val constraints
5. Module state accessed is constant

Current Implementation Status

The current value type plugin is thorn.plugins.ValueTypePlugin. It currently checks items 1), 2), 3) and 4). It does not however verify that the constraints are satisfied by instantiations, assignments and method calls. It does however elaborate in a ==/1 method for value classes and duplicates method entries according to the above.

Known Bugs

None.

E Atoms

Atoms or symbols are basically internalised strings with special syntax. Their intended use is as enum or special constant values, and as keys in dictionaries. Atoms literals are written thus:

:foo

and can be used as a right-hand side, e.g., in a variable assignment:

val mySymbol = :foo;

The class Atom is a value class.

Discussion

The chosen syntax overloads :, which might be confusing, seeing as how we use it for constraints on declarations. In particular, one could end up writing:

val x :foo = :bar;

which is not very readable. OTOH, saving an atom in a variable should rarely happen, and the syntax follows Ruby's.

As a more serious aside, we currently use ‘foo’ as an escape syntax in the compiler to allow certain special characters in identifiers. Maybe these could be unified for the sake of having fewer concepts.

Current Implementation Status

Done. The record plugin is called AtomPlugin and is enabled in Bootstrap.th.

One feature that is currently missing from the atom plugin is string interpolation in atoms, which is a relatively oft-used feature in Ruby. This is an easy extension.
Known Bugs

None.