In Terms of Scheme Expression

Exam project, Advanced Models and Programs 2 (Preview)
Fall 2006

Mikkel Byrsø Dan-Rognlie, 030481-****
Jens Fredriksen, 250673-****
Knut Tveitane, 251055-****

Contents

1 Preface and introduction 3

2 Background 4

2.1 Goal 4

3 Problem analysis 5

3.1 Parsing the Mini Scheme syntax into to a abstract syntax tree 5
3.2 Convert the AST to meaningfull C# 3.0 Expression trees 5
3.3 Invoke lambda expressions 5
3.4 A simple winform UI 5

4 Users manual 6

4.1 Files 6
4.2 Build and run the program 6
4.3 Description of the UI 6
4.4 Error messages 7
4.5 Examples 7

5 Implementation 8

5.1 System design 8
5.2 Scanning, parsing and building an AST 8
5.3 Environment 10
5.4 Semantic analysis 10
5.5 Building arithmetic operations 11
5.5.1 Arithmetic operation lists 11
5.5.2 Free variables 11
5.6 Define keyword 12
5.6.1 Lambda body builder …a weak guy. 12
5.7 Function invocation 13

6 Unit tests 14

7 Conclusion 15

8 References 16

Appendix A: Source code 17

1 Preface and introduction

This project serves as exam in the prototype course "Advanced Models and Programs (2)" at the IT University of Copenhagen, fall 2006. We thank the SDG and PLS group for spending time to organize and teach this preview course.

The project, in short, is about using C# 3.0 expression trees to implement an interpreter for expressions written in (a limited subset of) Scheme syntax. Within this approach, we want to touch several of the subject areas we had in the course: Scheme (function-oriented, prefix notation) syntax, lambda expressions, compiler generation with Coco/R, and runtime code generation.

We have worked in a group of three, thus collaborating on finding documentation and exchanging ideas. Sometimes we have worked physically together but mostly on distance via instant messaging or phone. We have conducted an Extreme Programming approach, where we continuously refined and refactored each others code and made use of unit testing to verify our code after the changes. The code has gone through many stages, alternating between refinement and deterioration. When implementing a new feature (using these unknown tools and libraries), the code has tempted to look like a crow's nest. Through refinement, refactoring and generalizations we have managed to get a reasonably well-behaved implementation in the end.

The report writing process has furthermore been collaborative using Google Docs. Even though we have collaborated on an overall basis we will try to identify sections of the code and report that each individual has had a main responsibility for. The responsibility in code roughly reflects the responsibility of the respective sections in the report. The responsibilities are as follows:

Coco/R grammar:	Mikkel
Terms:	Knut
Visitor: Arithmetic operations	Jens
Visitor: Define	Jens
Visitor: Invocation	Knut
Environment:	Knut
GUI:	Mikkel
Unit tests:	Mikkel

The rest we have collaborated on especially the general sections of the report.

2 Background

The objective of this project is an interpreter that can take an expression written in Scheme syntax, parse it and evaluate it using C# 3.0 as target language. It has not been the purpose to implement the full Scheme syntax. The focus has been on implementing lambda expressions, and a concrete measure for success has been to correctly interpert some fundamental constructs in Scheme: Arithmetic operations (+ - * /), binding (define) and function invocation, all operating on a single datatype: Integers.

Our initial target included partial evaluation, implementing the quote, quasiquote, comma and `eval` operators in Scheme. We did not achieve this within the deadline.

The reason for using C# 3.0 is to get acquainted with (some of) the new and relevant functionality coming in this version. C# 3.0 introduces expression trees and lambda expressions. We have concentrated on these aspects of C# 3.0, and have not made use of the other new features such as extension methods or implicit type declaration.

2.1 Goal

From the start we sat up goals for defining the functionality of the interpreter in term of having Scheme syntax as the guideline.

Status	Scheme syntax (input)	Our goal (output)
X	(42)	42
OK	(+ 12 30)	42
X	‘(+ 12 30)	add(12, 30)
OK	(define x 12)	x=12
OK	(define f (+ 12 30))	f=add(12,30)
OK	(define (f y)(+ y 30))	f(y)=add(y,30)
OK	(f 12)	42
OK	(define(f z)(+ z x)) [reference to var. x]	f(z)=add(z, x)
OK	(f 30)	42

3 Problem analysis

The problem can be split into different parts:

3.1 Parsing the Mini Scheme syntax into to a abstract syntax tree

Thid is solved by using Coco/R Parser Generator [4] which is a obviuous choice since Coco/R got a C# implementation and writing a lexer and parser from scratch is meaningless in this context. We build a coarse-grained abstract syntax tree (AST) from a class heirarchi, Term.

3.2 Convert the AST to meaningfull C# 3.0 Expression trees

Building of an AST of Term seems like double work thus C# Expression trees can represent the same, even more detailed. We wented for this design because of keep the static analysis simple instead of writing a visitor for all C# 3.0 Expressions. Additionally we had in mind the possiblity of each node in the AST containing an attribut for evaluation time, like the Scheme qoute, quasiqoute and comma.

3.3 Invoke lambda expressions

A interesting part of C# 3.0 Expressions are these can be kept as data [1] which then can be compiled into assembly code at runtime. We can therefor store the lambda expressions in a dictionary and then fetch it for invocation by its name. This also gives us the possiblity of passing a lambda expression into another lambda expression as a parameter. The possiblity of lazy evaluation also opens the doors for partial evaluation for C# 3.0 Expressions.

3.4 A simple winform UI

A Winform UI is implemented for entering input and to give a quick easy overview what is going on in the interpreter instead of having a unreadable console print.

4 Users manual

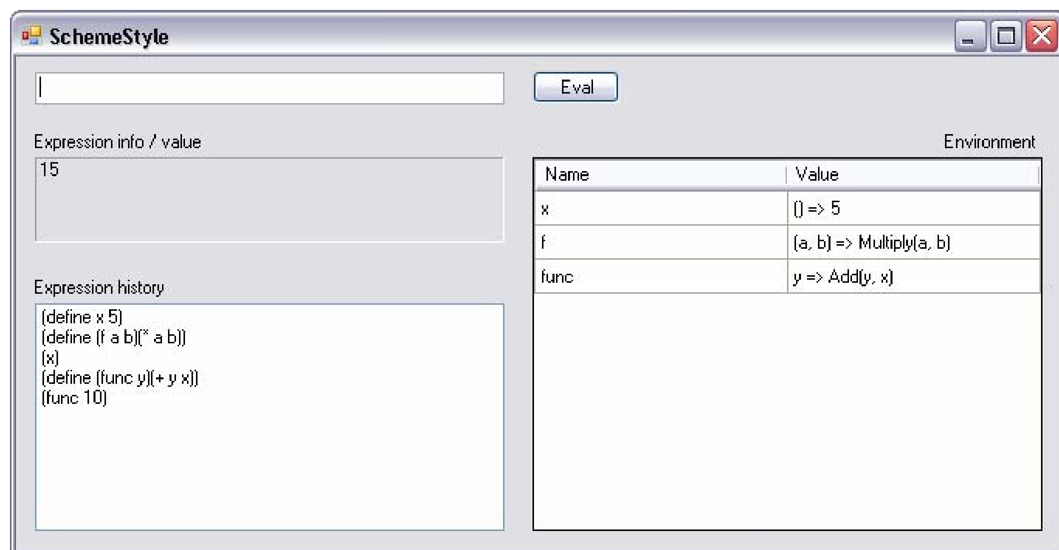
This section gives a short description of the supplied files, along with a guide to the program and some examples.

4.1 Files

The project files contain a Visual Studio Solution named SMP2.sln including all source files. The solution contains two projects: OrcasTest and Test. OrcasTest is the actual program. (Orcas is referring to the future release of Visual Studio codename “Orcas” with integrated support for C# 3.0 and LINQ). The Test project contains NUnit tests.

4.2 Build and run the program

In order to run the program the .NET Framework 2.0 has to be installed on the computer. In order to build it from Visual Studio the LINQ Preview (May 2006) package has to be installed. Both are downloadable from Microsoft. If you just want to run the program you start the OrcasTest.exe file in the folder SMP2\OrcasTest\bin\Release. The following WinForm will start:



4.3 Description of the UI

In the WinForm you can enter expressions in the text field in the upper-left corner. They are executed by clicking the Eval button or pressing <Enter>. Thus it is not possible to enter expressions that span multiple lines. If the expression is succesfully processed you will get a response in the field below. The expression is then added to

the Expression history list. If you want to execute a previous expression you can click on it in this list and it will be copied to the text field. If you declare expressions that is stored in the environment it will be displayed in the grid view to the right.

4.4 Error messages

Due to the time frame of this project it has not been possible to take care of problems in the program and nicely presenting them to the user. If you enter an expression that either can't be parsed or that the program can't build to a C# 3.0 expression tree and evaluate, you will get an exception and the program will close.

4.5 Examples

Below are some examples of user input and programs output. The ">" symbol denotes input lines, and the line below shows the output and changes made to the environment in square brackets. The " | " symbol separates the Name and Value column in the gridview.

```
> (+ 12 30)
42

> (+ 12 (* 2 15))
42

> (define x 5)
#<unspecified>     [ x | () => 5 ]

> (define (sqr a)(* a a))
#<unspecified>     [ sqr | a => Multiply(a, a) ]

> (sqr 5)
25

> (define (f a1 a2)(+ (* a1 a1) a2))
#<unspecified>     [ f | (a1, a2) => Add(Multiply(a1, a1), a2) ]

> (f 10 10)
110


> (f x 10)                          NB. x must exist in environment.
35

> (define (g y)(+ x (* 3 y)))    NB. x must exist in environment.
#<unspecified>     [ g | y => Add(x, Multiply(3, y)) ]
> (g 5)
20
```

5 Implementation

This section presents the main parts of the implementation with emphasis on design issues.

5.1 System design

The overall structuring of the system follows the tasks defined in the problem analysis. The call stack from the UI through the system is as follows

1. Get the input from the user and create a Parser that parses the expression
2. The Parser builds an AST of Term objects in the semantic actions
3. The Term hierarchy is traversed with a visitor that transforms it to C# 3.0 expression trees and saves declarations in the Environment.
4. The Expression is returned to the UI which looks at it and displays the result to the user. If the expression is a lambda declaration it just updates the environment view; otherwise it creates a new LambdaExpression from it, compiles and invokes it so that the expression is evaluated and the result of the invocation is displayed.

5.2 Scanning, parsing and building an AST

As described earlier we make use of Coco/R to generate a scanner (lexer) and a parser for our subset of the Scheme language. In the following we will describe the core features of Coco/R [4], our grammar and how the abstract syntax tree is built.

Scanner

Coco/R generates a scanner and parser from an attributed grammar – in our case SchemeStyle.ATG (Appendix A). The scanner performs the lexical analysis and is implemented as a deterministic finite automaton (DFA). The terminal token classes, e.g. identifiers and numbers is declared in EBNF.

Parser

Coco/R generates a parser that is specifed by a set of EBNF productions with attributes and semantic actions. Our productions can be seen in SchemeStyle.ATG.

Productions

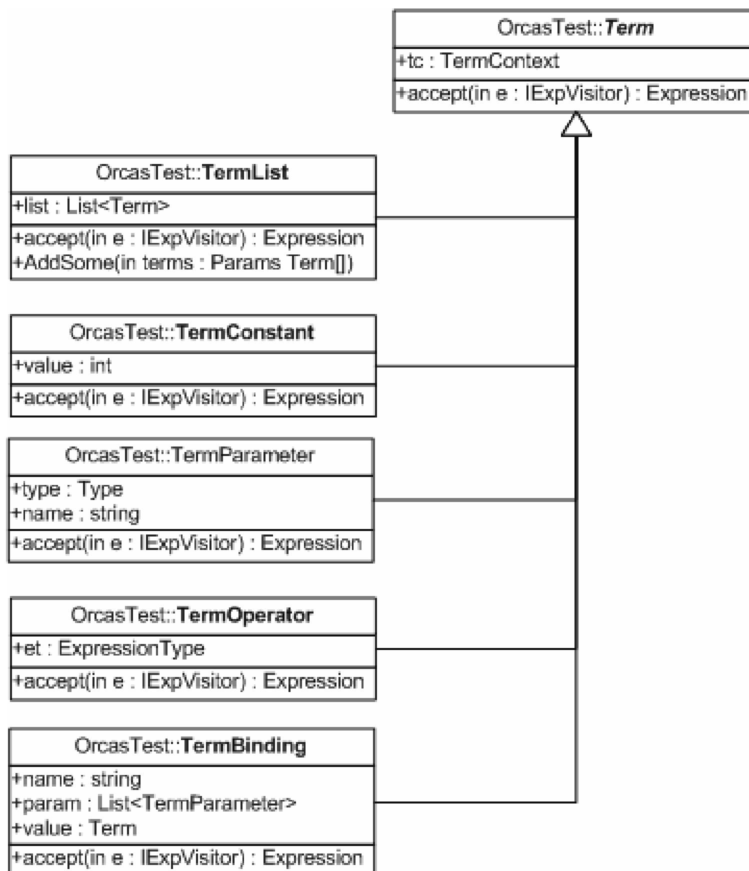
A production specifies the syntactical structure of a non-terminal symbol. They are translated into a recursive descent parser and every production is translated into a parsing method.

Semantic actions

The semantic actions are regular C# code that is inserted in the generated parser in the different parsing methods. Since semantic actions simply are copied to the parser without being checked by Coco/R, it is also possible to include C# 3.0 specific syntax. In the final version of the solution we do not use this possibility.

Parsing to an AST

In order to represent the different parts of an expression we have designed a small hierarchy of classes, as shown below.



We use the different Term classes for representing the abstract syntax tree. Term is the abstract super class for all types of terms. In general, an integer number is parsed to a TermConstant, a character string to a TermParameter, an arithmetic operator to a TermOperator and the keyword `define` to a TermBinding. A paranthesised expression is (recursively) parsed to a TermList containing other Term descendants. Nested parentesis are represented as TermLists within TermLists.

Further processing of the Term object to build the C# 3.0 Expressions will be done by a visitor traversing the produced term classes as described in sections 5.4 to 5.6

Future work

The grammar was designed with a larger set of functionality in mind. Therefore, the grammar can parse several Scheme syntax constructs that the rest of the program does not support. In particular, the grammar (and the Term hierarchy) identifies quote, quasiquote and comma operators (by means of the `TermContext` property), but in the further processing these are ignored. As syntax check and error handling are very sparsely implemented, other legal Scheme expressions, even though correctly parsed, might even crash the program.

5.3 Environment

An environment was needed to store bound variables and their values. Different considerations were made when the design og the environment was decided. It was discussed wether the environment should have notion of scope, facilitating declaration of local variables (within closures). It was decided that this was not a thing we would spend valuable time on. For the purpose of our demonstration application, there was neither any need to be able to remove entries from the environment, so it was decided to make a clean and minimalistic implementation, just consisting of a dictionary of global variables, implemented as a singleton, and with only two methods, SetValue and GetValue.

For each environment entry a string (name) and a LambdaExpression are stored, the name being the key. No distinction is made between variables storing lambda expressions and variables representing simple values. They are all stored as lambda ecpressions.

In particular, neither scoping nor function overloading is possible with this approach. An entry in the environment is identified by its name only. Entering a new item with the same name as an existing, effectively overwrites the existing entry.

5.4 Semantic analysis

After the parser has build a AST of Term classes the next step is to transform it to Expression trees by using a visitor, IExpVisitor.

```
public interface IExpVisitor{
          Expression Build(Term exp);
          Expression Build(TermList exp);
          Expression Build(TermParameter exp);
          Expression Build(TermConstant exp);
          Expression Build(TermBinding exp);
}
```

Scheme usage of parenthesise makes TermList to a central part of the semantic analysis thus all expressions are wrapped in parenthesises either if it is arithmetic operations, bindings or invocations. For seperate the actions condition statements

tests which kind of action should be carried out be checking the first element in the TermList and then delegating the TermList on to a specific method for the action. This functionality is in the visitor Build(TermList) method.

5.5 Building arithmetic operations

When entering an arithmetic operation, e.g (+ 12 30), the system have to return the result immediately to the UI. An arithmetic operation is represented in the AST as TermList starting with an arithmetic operator as the first element in the list. This operator represented by an Expressions.ExpressionType of either Add, Subtract, Multiply or Divide and can therefore be used directly in building the BinaryExpression.

An Arithmetic operation must contain at least one operand thus the first Expression operand can be assigned directly when entering the method by using the visitor at the TermLists 2nd. element.

5.5.1 Arithmetic operation lists

Scheme has the possibility of evaluate lists operands and nested lists in an arithmetic operation evaluated from left to right.

Arithmetic operation	Scheme evaluation order	Binary operation format
(- 90 45 3)	((90 - 45) – 3)	Subtract(Subtract(90, 45), 3)
(+ 2 (/ (* 8 10) 2))	(2 + ((8 * 10) / 2))	Add(2, Divide(Multiply(8, 10), 2))

The algorithm used for building this lefty binary tree bottom-up for representing the left-associative operations.

```
BuildOperationExpression(list)
    SoFar ← list[1]
    For each i ← list[2] to list.length
        SoFar ← MakeBinaryOperation(SoFar, list[i])
```

The implementation of the building of arithmetic operation from a TermList specified by having a ExpressionType as the first element which is used in the implementation of MakeBinaryOperation thus this have to choose which kind of the Expression binary operators should be picked.

5.5.2 Free variables

We wanted lambda expression be able of using free variables thus a declaration of a lambda expression should contain a name for the variable in the environment and

the value. Without having a flag for indicating whether the visitor is evaluating a arithmetic operation or binding variables or lambda expressions to the environment we had to chose to prevent arithmetic operation from using free variables. This is because of the implementation of MakeBinaryOperation serves both evaluating and binding. When a lambda expression have a free variable the lambda expressions body should not contain the value of free variable at binding time but when evaluation of the lambda expression. Hence the lambda body have to contain the free variable in somekind of format that it can be find in the environment when evaluating. For this purpose we store the name of the variable in a LINQ ParameterExpression.
This does curreny not work:

```
(define x 12)
(+ x 30)
```

Subconclusion
The MakeBinaryOperation should only serve one of it purpose properly hence a refactoring could give one of the building action its own visitor consisting of a subclass to the ordinary visitor with MakeBinaryOperation as a template method.

5.6 Define keyword

Binding a variable or lambda expression to the environment the visitor implementation, BldExpVisitor.Build(TermBinding), have to transform a Term-Binding into an C# 3.0 LambdaExpression for storing in the environment.

5.6.1 Lambda body builder …a weak guy.

Building a C# 3.0 LambdaExpression it is important to ensure parameters inside the body are references to the matching ParameterExpression in the LambdaExpression.

```
ParameterExpression p = Expression.Parameter(typeof(int), 21);
Expression.Lambda<T>(Expression.Add(p, p), p);
```

Hence our decision on having both variables and lambda expression saved as C# 3.0 LambdaExpression our method for creating the LambdaExpression in the environment:

1. Build Expression for either a variable, ConstantExpression, or lambda expression, BinaryExpression.
2. Stuff the Expression from step 1 into a LambdaExpression followed with the parameters

This way seems backwards since the LambdaExpression body needs instance references to the parameters. The reason is our desire of lambda expressions can use

free variables which first are evaluated when invoking the lambda expression. Thus this can be viewed as parameters and free variables inside a lambda expression are only represented by names. At invocation time these parameters on varialbes are called by name for fetching the right instance or value.
A lambda function can not have more than 3 parameters. This is because of the static declaration of delegate definitions in a C# 3.0 LambdaExpression. We only implement lambda expressions for delegates with 0, 1, 2 or 3 parameters. More could be implemented, but we have not found a way to make this more dynamic.

The way of do the binding got two weakness which affect the possiblity of making alias'. When binding a variable we assign the integer value in a C# 3.0 ConstantExpression and since a parameterless lambda expression is interpretated as a variable definition this trick does not work.

5.7 Function invocation

A Scheme expression like (f x...) represents a function invocation inquiry on the lambda expression bound to the variable f, where x... is a list of arguments (constants or variables) in accordance with the parameter list in the function declaration. The program looks up the function name in the environment (getting a lambda expression back), checks that the number of arguments is correct, and then binds the arguments to the parameters, resolving any variable references among the arguments (recursively invoking the functions they contain). `BuildInvocationExpression` method in the visitor class is providing this functionality. "InvocationExpression" might sound like a contradiction in terms, but due to the main design, all Scheme sentences are converted to expressions, also the invocations. An invocation returns a simple ConstantExpression that can readily be evaluated to its integer value.

The body of the LambdaExpression found in the environment has to be a ConstantExpression.

6 Unit tests

During the process we have used unit tests to ensure that changes and additions to the code didn't introduce new errors or change the behavior of the program. Especially in the refactoring phase these tests have proved valuable.

There are two kinds of tests. We have some tests that verify the output from the parser. They check whether the produced AST of Term objects for a given input expression is equal to coding the AST by hand.

The other tests are at a higher level. They test different expressions like the ones in the User manual to see if the whole program behaves correctly. They reside in EvalTest.cs in the Test project. These tests behave like if the input is typed directly in the WinForm. In this way the whole call stack is tested from the building of the AST in the parser to the building of the C# 3.0 expressions trees (the visitor implementation) and the compilation and invocation of the expressions.

7 Conclusion

It often happens, especially when experimenting with new technologies, that the first version of a program is completely discarded and a new version is made from scratch with a better design - a design one could not overview before one had tried and erred. Not that the first fraft was useless: It was the machinery for learning the new technology. We would very much have liked to present a second approach to the subject. Unfortunately, we did not have the time for that. This implementation is very much suffering from our fumbling in the dark with the new tools. However, the learning effect has been very good.

The Term class hierarchy used for the AST is very coarse-grained. In some cases it gives an ambigious representation, and this causes a lot of extra testing and type casting in the program. Given better time, we would have solved this differently. One solution is a more fine-grained Term hierarchy, another approach is to build C# 3.0 expression trees directly in the grammar file. The latter approach became clear to use after we read Tomas Petricek's blog [3] which introduced us to C# 3.0 ExpressionVisitor.

In general we have used a lot of time experimenting with C# 3.0 Expression trees and trying to find information about it. C# 3.0. Even though it has been extremely frustrating at times, we have learned a lot, and hopefully this will be useful to us in the future.

8 References

[1] Don Box blog "Code and Data in C#"
URL: http://pluralsight.com/blogs/dbox/archive/2006/05/12/23354.aspx
Last accessed: 2006.12.20

[2] Don Box blog "New drop of C# 3.0/LINQ"
URL: http://pluralsight.com/blogs/dbox/archive/2006/05/11/23258.aspx
Last accessed: 2006.12.20

[3] Tomas Petricek blog
URL: http://tomasp.net/articles/linq-expand.aspx
Last accessed: 2006.12.20

[4] Coco/R website and User Manual
URL: http://www.ssw.uni-linz.ac.at/coco/
Last accessed: 2006.12.20

[5] Hands-On Lab Manual, C# 3.0 (Part of VS LINQ Preview (May 2006))
URL:http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c- 4d13-9f04-70eb5e3dceea&displaylang=en
Last accessed: 2006.12.20

[6] C# version 3.0 Specification
URL:http://download.microsoft.com/download/9/5/0/9503e33e-fde6-4aed-b5d0-ffe749822f1b/csharp%203.0%20specification.doc
Last accessed: 2006.12.20

[7] The Scheme Programming Language
URL: http://www.scheme.com/tspl2d/
Last accessed: 2006.12.20

[8] SCM Scheme System
URL: http://www-swiss.ai.mit.edu/~jaffer/SCM.html
Last accessed: 2006.12.20

Appendix A: Source code

The source code presented on the following pages can also be downloaded from a CVS repository named "SMP2" found on this address:

:ext:knut@ssh.itu.dk:/import/home/knut/public_html/CVS

```
/* Coco/R lexer and parser specification for Scheme-like expressions. */
/* 2006-12-20 */

using System.Expressions;

COMPILER SchemeStyle
//$L
  public Term term;

/*-------------------------------------------------------------------------------------*/
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit = "0123456789".
  cr  = '\r'.
  lf  = '\n'.
  tab = '\t'.

TOKENS
  ident  = letter {letter | digit}.
  number = digit {digit}.

IGNORE cr + lf + tab

PRODUCTIONS

/*-------------------------------------------------------------------------------------*/
ExTerm<out Term t>                (. t=null; TermContext tc=TermContext.regular; .)
=   [ "'"                         (. tc=TermContext.quote; .)
    | "`"                         (. tc=TermContext.backquote; .)
    | ","                         (. tc=TermContext.comma; .)
    ]

    (Oper<out t>
    | Number<out t>
    | Ident<out t>
    | Liste<out t>
    | Binding<out t>
    )                             (.  t.tc=tc; .)
.

/*-------------------------------------------------------------------------------------*/
Binding<out Term t>               (. Term sign; Term value; .)
=
    "define"                      (. t=null; .)
    Signature<out sign>
    Value<out value>              (. t=new TermBinding(sign,value); .)
.

/*-------------------------------------------------------------------------------------*/
Signature<out Term sign>
=   ExTerm<out sign>
.

/*-------------------------------------------------------------------------------------*/
Value<out Term value>
=   ExTerm<out value>
.

/*-------------------------------------------------------------------------------------*/
Liste<out Term l>                 (. Term t; l = new TermList(); .)
=
    "(" { ExTerm<out t>           (. (l as TermList).list.Add(t); .)
        }
    ")"
.

/*-------------------------------------------------------------------------------------*/
Oper<out Term op>                 (. op = new TermOperator(); .)
=
    ( "+"                         (. (op as TermOperator).et = ExpressionType.Add; .)
    | "-"                         (. (op as TermOperator).et = ExpressionType.Subtract; .)
    | "*"                         (. (op as TermOperator).et = ExpressionType.Multiply; .)
    | "/"                         (. (op as TermOperator).et = ExpressionType.Divide; .)
    | ">"                         (. (op as TermOperator).et = ExpressionType.GT; .)
```

```
    | "eq?"                    (. (op as TermOperator).et = ExpressionType.EQ; .)
    | "if"                     (. (op as TermOperator).et = ExpressionType.Conditional; .↙
    )
    )
.

/*------------------------------------------------------------------------------*/
Ident<out Term name>           (. name = new TermParameter(); .)
=   ident                      (.
                                  (name as TermParameter).name = t.val;
                                  (name as TermParameter).type = typeof(int);
                               .)
.

/*------------------------------------------------------------------------------*/
Number<out Term val>           (. val = new TermConstant(); .)
=   number                     (. (val as TermConstant).value = Convert.ToInt32(t.val); .↙
    )
.

/*------------------------------------------------------------------------------*/
SchemeStyle                    (. Term t; .)
=   ExTerm<out t>              (. term = t; .)
.

END SchemeStyle.
```

```
using System;
using System.Collections.Generic;
using System.Text;

using System.Expressions;

namespace OrcasTest
{
    public enum TermContext {regular, quote, backquote, comma}

    public abstract class Term
    {
        public TermContext tc;

        public virtual Expression accept(IExpVisitor e){
            return e.Build(this);
        }
        public override string ToString()
        {
            return base.ToString();
        }
    }

    public class TermList : Term {
        public List<Term> list = new List<Term>();

        public void AddSome(params Term[] terms){
            foreach(Term t in terms){list.Add(t);}
        }
        public override Expression accept(IExpVisitor e){
            return e.Build(this);
        }
        // Not a very readable ToString() method (missing parenthesis in nested lists),
        // but it is good enough for comparison in tests
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            foreach (Term t in list)
            {
                sb.Append(t);
            }
            return sb.ToString();
        }
    }

    public class TermConstant : Term {
        public int value;

        public TermConstant(){}
        public TermConstant(int v):this(){
            value = v;
        }
        public override Expression accept(IExpVisitor e){
            return e.Build(this);
        }
        public override string ToString()
        {
            return value.ToString();
        }
    }

    public class TermParameter : Term {
        public Type type;
        public String name;
        public TermParameter(){}
        public TermParameter(Type t, String n):this(){
            type = t;
            name = n;
        }
        public override Expression accept(IExpVisitor e){
            return e.Build(this);
        }
        public override string ToString()
        {
```

```
            return "[type: " + type.ToString() + " | name: " + name + "]";
        }
    }

    public class TermOperator : Term {
        public ExpressionType et;
        public TermOperator(){}
        public TermOperator(ExpressionType e):this(){
            et = e;
        }
        public override Expression accept(IExpVisitor e){
            return e.Build(this);
        }
        public override string ToString()
        {
            return et.ToString();
        }
    }

    public class TermBinding : Term {
        public string name;
        public List<TermParameter> param = new List<TermParameter>();
        public Term value;

        public TermBinding() {}

        public TermBinding(Term sign, Term value) {
            if (sign is TermList) {
                TermList s = (TermList)sign;
                this.name = ((TermParameter)s.list[0]).name;
                for (int i = 1; i < s.list.Count; i++) {
                    this.param.Add((TermParameter)s.list[i]);
                }
            }
            else {
                this.name = ((TermParameter)sign).name;
                this.param = null;
            }
            this.value = value;
        }

        public override Expression accept(IExpVisitor e) {
            return e.Build(this);
        }
    }

    /*
    public class TermLambda : Term {
        public string name;
        public List<TermParameter> param;
        public Term body;

        public TermLambda(){
            param = new List<TermParameter>();
        }
        public TermLambda(string n, Term b, params TermParameter[] p) : this(){
            name = n;
            param.AddRange(p);
            body = b;
        }
        public override Expression accept(IExpVisitor e){
            return e.Build(this);
        }
    }
     * */
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Expressions;
using System.Query;

namespace OrcasTest {
    public class Environment {

        private static Environment env;

        public static Environment GetEnv() {
            if (env == null) env = new Environment();
            return env;
        }

        public readonly Dictionary<string, LambdaExpression> locals;

        private Environment() {
            locals = new Dictionary<string, LambdaExpression>();
        }

        public void SetValue(string name, LambdaExpression value) {
            if (locals.ContainsKey(name)) locals[name] = value;
            else locals.Add(name, value);
        }

        public LambdaExpression GetValue(string name) {
            if (locals.ContainsKey(name)) return locals[name];
            return null;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

using System.Expressions;

namespace OrcasTest
{
    public interface IExpVisitor
    {
        Expression Build(Term exp);
        Expression Build(TermList exp);
        Expression Build(TermParameter exp);
        Expression Build(TermConstant exp);
        Expression Build(TermBinding exp);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Expressions;
using System.Query;

namespace OrcasTest {
    public class BldExpVisitor : IExpVisitor {
        private Environment env;
        public Expression e;
        private List<ParameterExpression> parmList;

        public BldExpVisitor(Environment env) {
            this.env = env;
            parmList = new List<ParameterExpression>();
        }

        public Expression Build(Term exp) {
            return exp.accept(this);
        }

        public Expression Build(TermList exp) {
            if (exp.list.ElementAt<Term>(0) is TermOperator) {
                return BuildOperationExpression(exp);
            }
            else if (exp.list.ElementAt<Term>(0) is TermBinding) {
                return BuildBindingExpression(exp);
            }
            else if (exp.list.ElementAt<Term>(0) is TermParameter) {
                return BuildInvocationExpression(exp);
            }
            return null;
        }

        private Expression BuildInvocationExpression(TermList exp) {
            LambdaExpression e1;
            if (env.GetValue((exp.list.ElementAt<Term>(0) as TermParameter).name) != null) ↙
   {
                e1 = env.GetValue((exp.list.ElementAt<Term>(0) as TermParameter).name);

                ExpressionExpander ex = new ExpressionExpander(env, e1);
                e1 = LambdaExpression.Lambda(e1.Type, ex.Visit(e1.Body), e1.Parameters);

                if (exp.list.Count - 1 == e1.Parameters.Count) {
                    object[] ol = new object[e1.Parameters.Count];
                    for (int i = 1; i < exp.list.Count; i++) {
                        if (exp.list.ElementAt<Term>(i) is TermConstant) {
                            ol[i - 1] = ((ConstantExpression)exp.list.ElementAt<Term>(i). ↙
    accept(this)).Value;
                        }
                        else if (exp.list.ElementAt<Term>(i) is TermParameter) {
                            ol[i - 1] = ((ConstantExpression)(env.GetValue((exp.list.     ↙
    ElementAt<Term>(i) as TermParameter).name) as LambdaExpression).Body).Value;
                        }

                        else if (exp.list.ElementAt<Term>(i) is TermList) {
                            ol[i - 1] = ((ConstantExpression)(BuildInvocationExpression    ↙
    (exp.list.ElementAt<Term>(i) as TermList))).Value;
                        }
                    }

                    return Expression.Constant(e1.Compile().DynamicInvoke(ol));
                }
            }
            return null;
        }

        private Expression BuildBindingExpression(TermList exp) {
            return exp.list.ElementAt<Term>(0).accept(this);
        }

        private Expression BuildOperationExpression(TermList exp) {
            ExpressionType et = ((TermOperator)exp.list.ElementAt<Term>(0)).et;
            Expression left = exp.list.ElementAt<Term>(1).accept(this);
```

```
            for (int i = 2; i < exp.list.Count; i++) {
                Expression right = exp.list.ElementAt<Term>(i).accept(this);
                left = this.MakeBinaryOperation(et, left, right);
            }
            return left;
        }

        public Expression Build(TermParameter exp) {
            return Expression.Parameter(exp.type, exp.name);
        }

        public Expression Build(TermConstant exp) {
            return Expression.Constant(exp.value);
        }

        public Expression Build(TermBinding exp) {
            Expression body;
            this.ExtractParameterExp(exp.param);
            if (exp.value is TermList) {

                body = this.Build(exp.value as TermList);
            }
            else {
                body = Expression.Constant((exp.value as TermConstant).value);
            }

            LambdaExpression e1 = this.correctLambdaTypes(body, parmList);
            env.SetValue(exp.name, e1);

            return e1;
        }

        /*******************************************************
         Help Methods
         ****************************************************/
        public LambdaExpression correctLambdaTypes(Expression body, List
    <ParameterExpression> pe) {
            switch (pe.Count) {
                case 1:
                    return Expression.Lambda<Func<int, int>>(body, pe);
                case 2:
                    return Expression.Lambda<Func<int, int, int>>(body, pe);
                case 3:
                    return Expression.Lambda<Func<int, int, int, int>>(body, pe);
                default:
                    return Expression.Lambda<Func<int>>(body, pe);
            }
        }

        private Expression neutralValueForOperations(ExpressionType et) {
            switch (et) {
                case ExpressionType.Add:
                    return Expression.Constant(0);
                case ExpressionType.Multiply:
                    return Expression.Constant(1);
                case ExpressionType.Subtract:
                    return Expression.Constant(0);
                case ExpressionType.Divide:
                    return Expression.Constant(1);
                default:
                    return Expression.Constant(-1);
            }
        }

        private Expression MakeSubstitutionAndSwapExp_1_2(ExpressionType et, ref
    Expression e1) {
            Expression eTemp = e1;
            e1 = this.neutralValueForOperations(et);
            return eTemp;
        }

        private Expression MakeBinaryOperation(ExpressionType et, Expression e1,
    Expression e2) {
            Console.WriteLine("Exp type: " + et + " e1:" + e1 + " e2:" + e2);
```

```
            foreach (ParameterExpression pe in parmList) {
                Console.Write(pe.GetHashCode() + " - ");
            }

            foreach (ParameterExpression pe in parmList) {
                if (e1.GetType() == typeof(ParameterExpression) && pe.Name == (e1 as
    ParameterExpression).Name) {
                    e1 = pe;
                }

                if (e2.GetType() == typeof(ParameterExpression) && pe.Name == (e2 as
    ParameterExpression).Name) {
                    e2 = pe;
                }
            }

            switch (et) {
                case ExpressionType.Add:
                    return Expression.Add(e1, e2);
                case ExpressionType.Multiply:
                    return Expression.Multiply(e1, e2);
                case ExpressionType.Subtract:
                    return Expression.Subtract(e1, e2);
                case ExpressionType.Divide:
                    return Expression.Divide(e1, e2);
                default:
                    return Expression.Constant(-1);
            }
        }

        public void ExtractParameterExp(List<TermParameter> tp) {
            if (tp != null) {
                foreach (TermParameter t in tp) {
                    ParameterExpression pex = ((Build(t) as ParameterExpression));
                    parmList.Add(pex);
                }

            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Query;
using System.Xml.XLinq;
using System.Data.DLinq;
using System.Expressions;

namespace OrcasTest {

    public partial class Form1 : Form {

        public Form1() {
            InitializeComponent();
            textBox1.Text = "(define x 5)";
        }

        private void HandleUserInput()
        {
            Term term = Parse(textBox1.Text);
            Expression exy = CreateExpression(term);

            exprHistory.Items.Add(textBox1.Text);
            textBox1.Text = "";

            if (exy.NodeType.Equals(ExpressionType.Lambda)){
                label1.Text = @"#<unspecified>";
            } else {
                label1.Text = Expression.Lambda<Func<int>>(exy, new List
    <ParameterExpression>()).Compile().DynamicInvoke(new object[] { }).ToString();
            }

            dataGridView1.Rows.Clear();
            Environment env = Environment.GetEnv();
            foreach (string o in env.locals.Keys)
                dataGridView1.Rows.Add(o, env.locals[o]);
        }

        public Term Parse(string s)
        {
            Parser parser = new Parser(s);
            parser.Parse();
            return parser.term;
        }

        public Expression CreateExpression(Term term)
        {
            return term.accept(new BldExpVisitor(Environment.GetEnv()));
        }


        #region Event handlers

        private void button1_Click(object sender, EventArgs e)
        {
            HandleUserInput();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)Keys.Return)
            {
                e.Handled = true;
                HandleUserInput();
            }
        }

        private void exprHistory_SelectedIndexChanged(object sender, EventArgs e)
        {
```

```
            textBox1.Text = ((ListBox)sender).SelectedItem.ToString();
        }

        #endregion
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Query;
using System.Xml.XLinq;
using System.Data.DLinq;

namespace OrcasTest {
    static class Program {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main() {
            Application.EnableVisualStyles();
            Application.Run(new Form1());
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Query;
using System.Expressions;
using NUnit.Framework;
using OrcasTest;

namespace Test
{
    [TestFixture]
    public class ParserTest
    {
        Parser p;

        [SetUp]
        public void BeforeEachTest()
        {

        }

        [TearDown]
        public void AfterEachTest()
        {
            p = null;
        }

        [Test]
        public void TestAddition()
        {
            p = new Parser("(+ 5 10)");
            p.Parse();

            var op = new TermOperator(ExpressionType.Add);
            var tc1 = new TermConstant(5);
            var tc2 = new TermConstant(10);
            var tl = new TermList();
            tl.AddSome(op, tc1, tc2);

            TermList tlParser = (TermList) p.term;

            Console.WriteLine(tl);
            Console.WriteLine(tlParser);

            Assert.AreEqual(tl.ToString(), tlParser.ToString());
            Assert.AreEqual(tl.list.Count, tlParser.list.Count);

        }

        [Test]
        public void TestNestedArithmetic()
        {
            p = new Parser("(+ (* 5 2) 10)");
            p.Parse();

            var add = new TermOperator(ExpressionType.Add);
            var mul = new TermOperator(ExpressionType.Multiply);
            var tc1 = new TermConstant(5);
            var tc2 = new TermConstant(2);
            var tc3 = new TermConstant(10);
            var tl = new TermList();
            tl.AddSome(add, mul, tc1, tc2, tc3);

            TermList tlParser = (TermList)p.term;

            Console.WriteLine(tl);
            Console.WriteLine(tlParser);

            Assert.AreEqual(tl.ToString(), tlParser.ToString());
        }


        private void WriteLineTest(object o)
        {
```

```
            Console.WriteLine("\n\n*** TEST OUTPUT ***      " + o.ToString() + "\n");
        }
    }


}
```

```
using System;
using System.Collections.Generic;
using System.Text;

using System.Query;
using System.Expressions;
using NUnit.Framework;
using OrcasTest;

namespace Test
{
    [TestFixture]
    public class EvalTest
    {
        Parser p;
        OrcasTest.Environment env;
        BldExpVisitor visitor;
        Expression exp;
        LambdaExpression lambda;
        int result;

        [SetUp]
        public void BeforeEachTest()
        {
            env = OrcasTest.Environment.GetEnv();
            visitor = new BldExpVisitor(env);
        }

        [TearDown]
        public void AfterEachTest()
        {
            p = null;
            env = null;
            visitor = null;
            exp = null;
            lambda = null;
        }

        [Test]
        public void TestSimpleAddition()
        {
            result = InvokeExpr("(+ 5 5)");
            Assert.AreEqual(10, result);
            Assert.AreEqual("Add(5, 5)", exp.ToString());
        }

        [Test]
        public void TestSimpleSubtraction()
        {
            result = InvokeExpr("(- 10 5)");
            Assert.AreEqual(5, result);
            Assert.AreEqual("Subtract(10, 5)", exp.ToString());
        }

        [Test]
        public void TestSimpleMultiplication()
        {
            result = InvokeExpr("(* 5 5)");
            Assert.AreEqual(25, result);
            Assert.AreEqual("Multiply(5, 5)", exp.ToString());
        }

        [Test]
        public void TestSimpleDivision()
        {
            result = InvokeExpr("(/ 10 5)");
            Assert.AreEqual(2, result);
            Assert.AreEqual("Divide(10, 5)", exp.ToString());
        }


        [Test]
        public void TestLongAddition()
        {
```

```
            result = InvokeExpr("(+ 1 2 3 4 5 6 7 8 9)");
            Assert.AreEqual(45, result);
            Assert.AreEqual("Add(Add(Add(Add(Add(Add(Add(Add(1, 2), 3), 4), 5), 6), 7), 8)↙
    , 9)", exp.ToString());
        }

        [Test]
        public void TestNestedArithmeticOperations()
        {
            result = InvokeExpr("(- 5 (/ 1000 (+ ( *10 10) 100)))");
            WriteLineTest(exp);
            Assert.AreEqual(0, result);
            Assert.AreEqual("Subtract(5, Divide(1000, Add(Multiply(10, 10), 100)))", exp. ↙
    ToString());
        }

        [Test]
        public void TestLambdaDeclWithOneParameter()
        {
            p = new Parser("(define(f x)(* x x))");
            p.Parse();
            exp = p.term.accept(visitor);
            Assert.AreEqual("x => Multiply(x, x)", exp.ToString());
        }

        [Test]
        public void TestLambdaDeclWithTwoParameters()
        {
            p = new Parser("(define(f x y)(+ x y))");
            p.Parse();
            exp = p.term.accept(visitor);
            WriteLineTest(exp);
            Assert.AreEqual("(x, y) => Add(x, y)", exp.ToString());
        }

        [Test]
        public void TestLambdaDeclWithThreeParameters()
        {
            p = new Parser("(define(f x y z)(+ x y z))");
            p.Parse();
            exp = p.term.accept(visitor);
            WriteLineTest(exp);
            Assert.AreEqual("(x, y, z) => Add(Add(x, y), z)", exp.ToString());
        }

        [Test]
        public void TestLambdaInvocationWithOneParameter()
        {
            Expression func = ParseAndBuild("(define(f x)(* x x))");
            WriteLineTest("func: " + func);

            exp = ParseAndBuild("(f 5)");
            WriteLineTest("exp: " + exp);

            lambda = Expression.Lambda<Func<int>>(exp, new ParameterExpression[] { });
            result = (int)lambda.Compile().DynamicInvoke(new object[] { });

            Assert.AreEqual("x => Multiply(x, x)", func.ToString());
            Assert.AreEqual(25, result);
        }

        [Test]
        public void TestLambdaInvocationWithThreeParameters()
        {
            Expression func = ParseAndBuild("(define(f a b c)(- a b c))");
            WriteLineTest("func: " + func);

            exp = ParseAndBuild("(f 100 50 40)");
            WriteLineTest("exp: " + exp);

            lambda = Expression.Lambda<Func<int>>(exp, new ParameterExpression[] { });
            result = (int)lambda.Compile().DynamicInvoke(new object[] { });

            Assert.AreEqual("(a, b, c) => Subtract(Subtract(a, b), c)", func.ToString());
```

```
            Assert.AreEqual(10, result);
        }

        [Test]
        public void TestLambdaDeclAndInvokeWhichUseAnotherDeclVariable()
        {
            Expression xExp = ParseAndBuild("(define x 1000)");
            WriteLineTest("eExp: " + xExp);

            Expression func = ParseAndBuild("(define(f a b)(+ x (* a b)))");
            WriteLineTest("func: " + func);

            exp = ParseAndBuild("(f 100 100)");
            WriteLineTest("exp: " + exp);

            lambda = Expression.Lambda<Func<int>>(exp, new ParameterExpression[] { });
            result = (int)lambda.Compile().DynamicInvoke(new object[] { });

            Assert.AreEqual("(a, b) => Add(x, Multiply(a, b))", func.ToString());
            Assert.AreEqual(11000, result);
        }

        [Test]
        public void TestOneOperand()
        {
            result = InvokeExpr("(+ 15");
            Assert.AreEqual(15, result);
        }



        // *********************************
        // UTILITY METHODS
        // *********************************

        private int InvokeExpr(string expression)
        {
            p = new Parser(expression);
            p.Parse();
            exp = p.term.accept(visitor);
            lambda = Expression.Lambda<Func<int>>(exp, new ParameterExpression[] { });
            return (int)lambda.Compile().DynamicInvoke(new object[] { });
        }

        private Expression ParseAndBuild(string expression)
        {
            p = new Parser(expression);
            p.Parse();
            return p.term.accept(visitor);
        }

        private void WriteLineTest(object o)
        {
            Console.WriteLine("\n\n*** TEST OUTPUT ***     " + o.ToString() + "\n");
        }


    }
}
```