Implementation Oriented Language Targeted Goto $\{((\lambda x.xx)(\lambda y.yy))\}$ Guide:

Roscoe S. Casita

University of Oregon

December 2, 2015

λ Calculus *to* $-$ *calculate*: One Axiom, One Reduction

Core Axiom: α non-naming equivalence: $(\lambda y.y\ y) == (\lambda x.x\ x)$

Core Reduction: β replacement: Replace x with a is this operator [a/x]

Identity Function $= ((\lambda x.x)a) \downarrow (x)[a/x] \downarrow (a)$

Identity applied to identity $= ((\lambda x.x)(\lambda y.y)) \Rightarrow (\lambda y.y)$

Infinite Loop $= ((\lambda y.y\ y)(\lambda x.x\ x)) \Rightarrow ((\lambda x.x\ x)(\lambda x.x\ x))^{\infty}$

Problem statement: Implement a λ expression evaluator

λ Calculus can model all computations, even evaluate itself.

Evaluate stepwise $PRED(SUCC\ ZERO) \Rightarrow ZERO$ as test.

Interpret $((\lambda x.xx)(\lambda y.yy))$ ... FOREVER!

Parser Generate : *Grammar* $\Rightarrow$ ***lexical-parser***

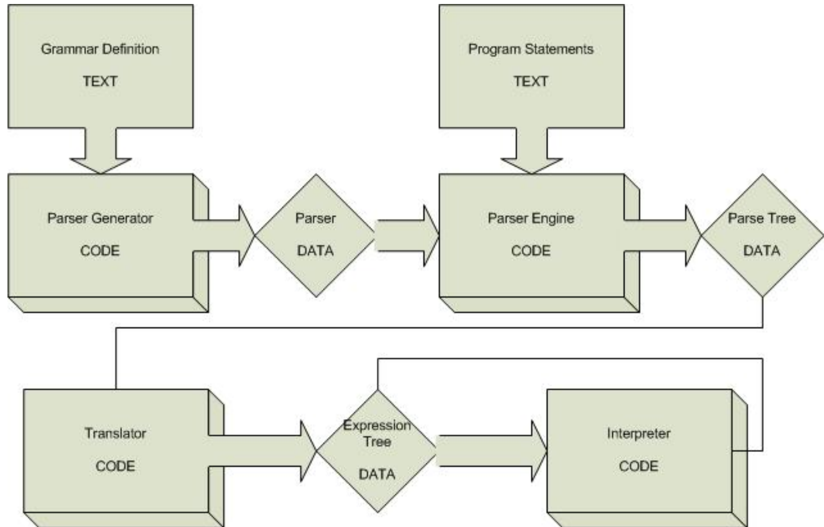
Parser Engine : ***lexical-parser*** $+ \lambda$ *expression* $\Rightarrow$ ***parse-tree***

Translater : ***parse-tree*** $\Rightarrow$ ***expression-tree***.

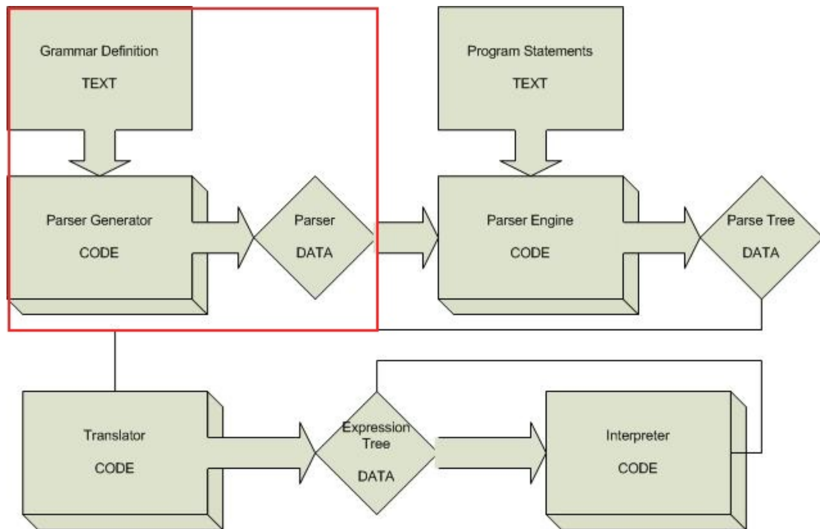
I*nterpreter* : ***expression-tree*** $\Rightarrow$ ***expression-tree*** until **done**.

Overview of the pipeline:

From *text-strings* to ***Data*** evaluating ***Data*** as **Code**:



Parser Generator: Add λ-*grammar-definition-rules* $\Rightarrow$ λ-***Parser***



Divergent directions...

Grammars are a language... thus a meta-language encoding exists

Let's define a grammar that defines a grammar.

Let's ensure the grammar definition can parse its OWN definition!

THE GRAMMAR PARSING GRAMMAR!!!!!

Form1

Parse Rules

Parse Program

Convert Parse Tree to Abstract Tree

Interpret Program Step

Step Every MS: 500

```
<PROGRAM>::==<DECLARATIONS>
<DECLARATIONS>::==<DECLARATION>
<DECLARATIONS>
<DECLARATIONS>::==<DECLARATION>
<DECLARATION>::==<REG_RULE_NAME>
<REG_RULE_EQUALITY><REG_EXP><EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME>
<GRAMMAR_RULE_EQUALITY><SEQUENCE>
<EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME>
<GRAMMAR_RULE_EQUALITY>
<REG_RULE_NAME><EOL>
<SEQUENCE>::==<GRAMMAR_RULE_NAME>
<SEQUENCE>
<SEQUENCE>::==<GRAMMAR_RULE_NAME>
<EOL>::==EOL
<REG_EXP>::==REG_EXP
<REG_RULE_NAME>::==REG_RULE_NAME
<REG_RULE_EQUALITY>::
==REG_RULE_EQUALITY
<GRAMMAR_RULE_NAME>::
==GRAMMAR_RULE_NAME
<GRAMMAR_RULE_EQUALITY>::
==GRAMMAR_RULE_EQUALITY
EOL:=^(\n)
REG_EXP:=^(\^.*)
REG_RULE_NAME:=^(\w+)
REG_RULE_EQUALITY:=^(:=)
GRAMMAR_RULE_NAME:=^(<\w+>)
GRAMMAR_RULE_EQUALITY:=^(::==)
```

<PROGRAM>

<DECLARATIONS>

<DECLARATION>

<SEQUENCE> <EOL> <REG_EXP> <REG_RULE_NAME> <REG_RULE_EQUALITY> <GRAMMAR_RULE_EQUALITY>

<GRAMMAR_RULE_NAME> EOL REG_EXP REG_RULE_NAME REG_RULE_EQUALITY GRAMMAR_RULE_EQUALITY

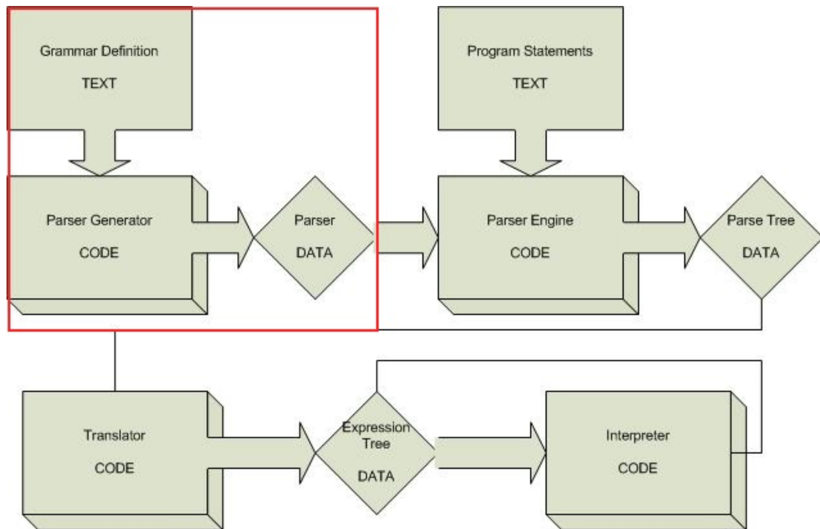
GRAMMAR_RULE_NAME

Grammar Parser

```
<PROGRAM>::==<DECLARATIONS>
<DECLARATIONS>::==<DECLARATION><DECLARATIONS>
<DECLARATIONS>::==<DECLARATION>
<DECLARATION>::==<REG_RULE_NAME><REG_RULE_EQUALITY><REG_EXP><EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME><GRAMMAR_RULE_EQUALITY><SEQUENCE><EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME><GRAMMAR_RULE_EQUALITY><REG_RULE_NAME><EOL>
<SEQUENCE>::==<GRAMMAR_RULE_NAME><SEQUENCE>
<SEQUENCE>::==<GRAMMAR_RULE_NAME>
<EOL>::==EOL
<REG_EXP>::==REG_EXP
<REG_RULE_NAME>::==REG_RULE_NAME
<REG_RULE_EQUALITY>::==REG_RULE_EQUALITY
<GRAMMAR_RULE_NAME>::==GRAMMAR_RULE_NAME
<GRAMMAR_RULE_EQUALITY>::==GRAMMAR_RULE_EQUALITY
```

```
Parsed Program: <PROGRAM>::==<DECLARATIONS>
<DECLARATIONS>::==<DECLARATION><DECLARATIONS>
<DECLARATIONS>::==<DECLARATION>
<DECLARATION>::==<REG_RULE_NAME><REG_RULE_EQUALITY><REG_EXP><EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME><GRAMMAR_RULE_EQUALITY><SEQUENCE>
<EOL>
<DECLARATION>::==<GRAMMAR_RULE_NAME><GRAMMAR_RULE_EQUALITY>
<REG_RULE_NAME><EOL>
<SEQUENCE>::==<GRAMMAR_RULE_NAME><SEQUENCE>
<SEQUENCE>::==<GRAMMAR_RULE_NAME>
<EOL>::==EOL
<REG_EXP>::==REG_EXP
<REG_RULE_NAME>::==REG_RULE_NAME
<REG_RULE_EQUALITY>::==REG_RULE_EQUALITY
<GRAMMAR_RULE_NAME>::==GRAMMAR_RULE_NAME
<GRAMMAR_RULE_EQUALITY>::==GRAMMAR_RULE_EQUALITY
EOL:=^(\n)
```

Parser Generator: Add λ-*grammar-definition-rules* $\Rightarrow$ λ-***Parser***



Pit Falls in “commonly repeated” λ-grammars:

<exp>::==<var> |(λ <var> . <exp>)|(<exp><exp>)

This is not an adequate grammar unless you LOVE lots of “()”

$\lambda x.\lambda y.y(xx)$ must be written as $(\lambda x.(\lambda y.((y(xx)))))$

Nested variable capture is an advanced topic: $\lambda x.\lambda y.x((\lambda x.x)y)$

When frustration sets in, ask for help.

Don't fix a bad grammar in the parser, translator, or interpreter.

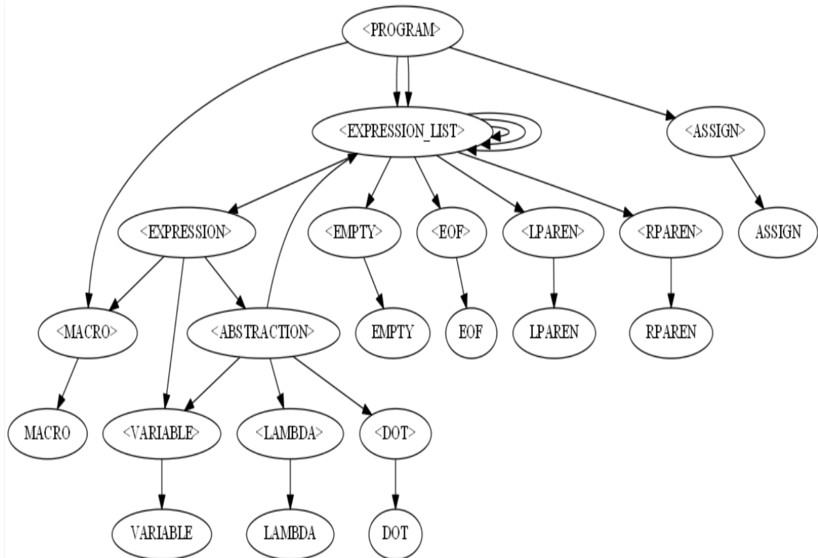
Left-hand associativity MUST be built into the grammar itself:

<exp-list>::== (<exp-list>) <exp-list> | <exp><exp-list> |ϵ
<exp>::==<var> |λ <var> . <exp-list>

< *exp-list* > will be 'caught' in a loop in the **Translator** later.

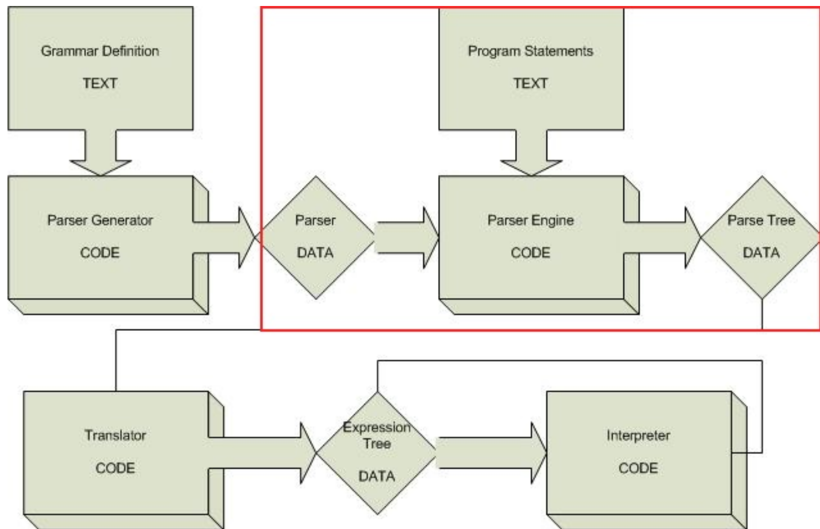
Credit, explanation, and thanks to Dr. Boyana Norris.

Generated λ-*Parser* for **Parser Engine**



Grammar Parser

Parser Engine: Use λ-*parser* and λ-*text* $\Rightarrow$ λ-*syntax-tree*



Parser Engine

LL(*) Parser Engine: $O(n^2)$ Memory & Runtime.

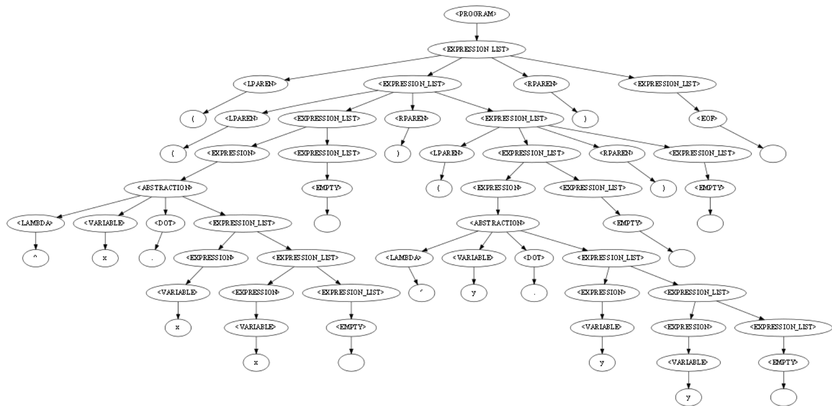
Two mutually recursive functions: Parse & Parse Segment.

Deterministic recursive decent following *Grammar Rules*.

Pros: Saves input on each call, thus stops at first 'bad' character.

Cons: Infinite loop on bad grammar. Slow & Big on 'weird' input.

Complete λ-***syntax-tree*** of $((\lambda x.xx)(\lambda y.yy))$



AbstractParse Tree

Parser Engine

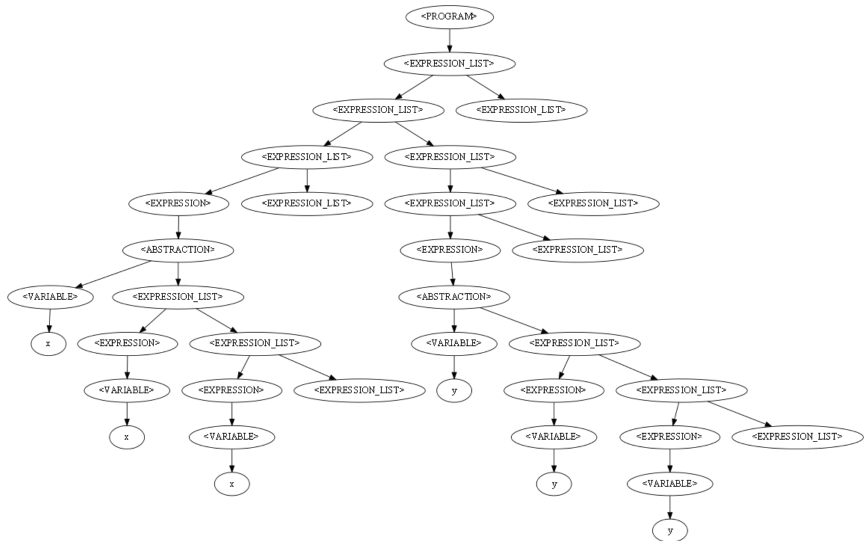
The **Parser Engine** is the least external-knowledge subsystem.

The **Parser Generator** encodes a ***Parser*** for the **Parser Engine**.

The **Parser Engine** uses ***Parser*** on *input* creating a ***syntax-tree***.

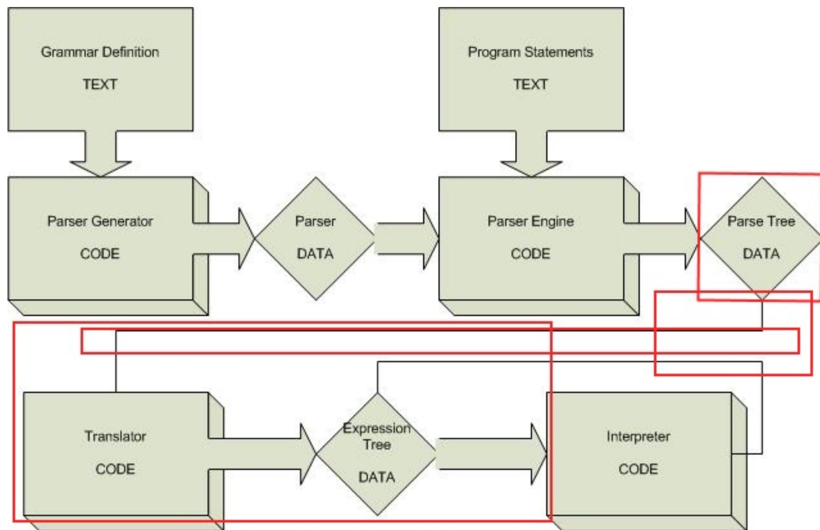
The **Translater** decodes the ***syntax-tree*** into ***expression-tree***.

Trimmed λ-***syntax-tree*** of $((\lambda x.xx)(\lambda y.yy))$



Abstract Parse Tree

Translater: Use λ-*syntax-tree* and λ-**Types** $\Rightarrow$ λ-*expression-tree*



Translate deep quantum entanglement...

The **Translator** embeds knowledge of λ-*grammar* structure.

The **Translator** embeds knowledge of λ-*expression-tree* structure.

The **Translator** matches *string* to ***syntax-tree*** *rule*.

The **Translator** builds the ***expression-tree*** nodes.

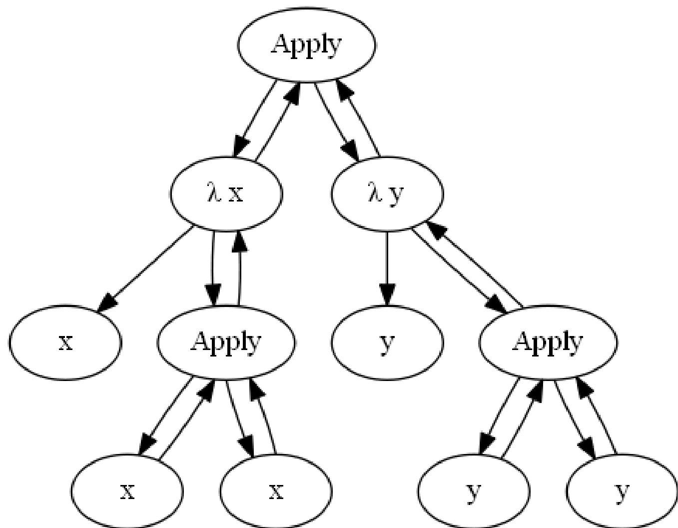
Remember the <*exp-list*> ? The loop handler is next.

Grammar *text* strings found in the **Translator**

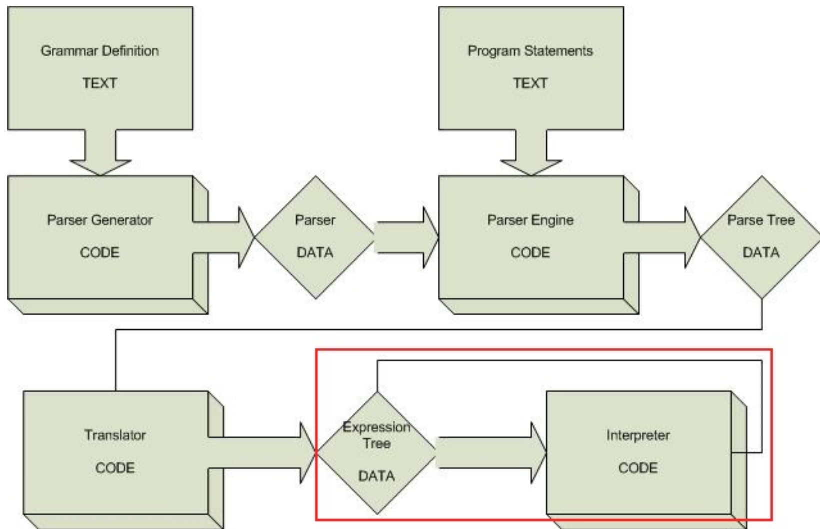
```
switch (tree.Rule.Name)
{
    case "<EXPRESSION_LIST>":
        {
            ParseTree ptr = tree;
            Expression return_value = null;
            do
            {
                if(ptr.Decendants.Count == 0)
                {
                    ptr = null;
                }
                else if (return_value == null)
                {
                    return_value = ConvertTree(ptr.Decendants[0]);
                    ptr = ptr.Decendants[1];
                }
                else
                {
                    Expression new_return = ConvertTree(ptr.Decendants[0]);
                    ptr = ptr.Decendants[1];
                    return_value = new Apply(return_value,new_return);
                }
            }while(ptr != null);

            if(return_value == null)
            {
                throw new Exception("This also should never happen.");
            }
             return return_value;
        }
    case "<EXPRESSION>":
        return ConvertTree(tree.Decendants[0]);
```

```
case "<ABSTRACTION>":
    {
        Variable v = new Variable(tree.Decendants[0].Decendants[0].Token);

        if (!NamedCaptures.ContainsKey(v.Name))
        {
            NamedCaptures.Add(v.Name, new List<Variable>());
        }

        NamedCaptures[v.Name].Insert(0, v);

        Expression exp = ConvertTree(tree.Decendants[1]);

        NamedCaptures[v.Name].RemoveAt(0);

        if (NamedCaptures[v.Name].Count == 0)
        {
            NamedCaptures.Remove(v.Name);
        }

        Abstraction a = new Abstraction(v, exp);

        return a;
    }
case "<VARIABLE>":
    {
        if (NamedCaptures.ContainsKey(tree.Decendants[0].Token))
        {
            return NamedCaptures[tree.Decendants[0].Token][0].Copy();
        }
        else
        {
            return new Variable(tree.Decendants[0].Token);
        }
    }
case "<MACRO>":
    {
        return new Macro(tree.Decendants[0].Token);
    }
```

Translater: λ-*expression-tree* of $((\lambda x.xx)(\lambda y.yy))$



Interpreter : Use λ-*expression-trees* $\Rightarrow$ λ-*expression-trees* until **done**



Enough with the syntax, where is the semantic action?

β reduction from λ calculus semantics in C#:

```
private Expression Replace(Expression tree, Variable v, Expression value)
{
    if (tree is Variable)
    {
        Variable var = tree as Variable;

        if (var.ID == var.ID)
        {
            return value.Copy();
        }
        return tree;
    }
    else if (tree is Macro)
    {
        return tree;
    }
    else if (tree is Abstraction)
    {
        Abstraction l = tree as Abstraction;
        l.Body = Replace(l.Body, v, value);
        l.Body.Parent = l;
        return tree;
    }
    else if (tree is Apply)
    {
        Apply ex = tree as Apply;
        ex.Execute = Replace(ex.Execute, v, value);
        ex.Execute.Parent = ex;
        ex.Parameter = Replace(ex.Parameter, v, value);
        ex.Parameter.Parent = ex;
        return tree;
    }
    throw new Exception("Unknown Expression Type. Bad Replacement operation.");
}
```

Well that was rather unsatisfying.

The entire semantic implementation is '3' switch statements.

The majority of the case matches are the "boring" cases.

There is one interesting statement in Replace.

The absence of one line interestingly implements Lazy evaluation.

Semantics of implemented λ-*language*

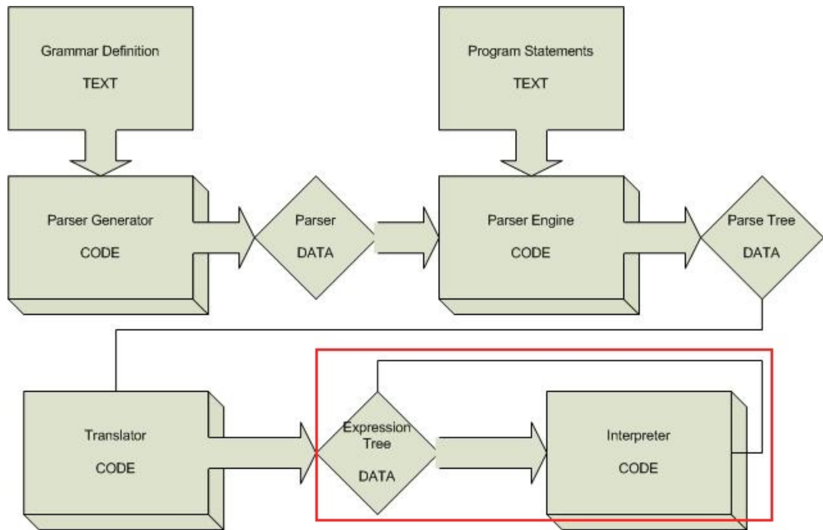
$\dfrac{\text{Var Done}}{x \to x}$	$\dfrac{\text{Abstraction Done}}{\lambda \to \lambda}$	$\dfrac{\text{Apply Done}}{e1', e2 \to e1', e2}$
$\dfrac{\text{Apply Reduce } \beta}{\lambda x.e1, e2 \to e1[e2/x]}$	$\dfrac{\text{Apply Left}}{e1, e2 \to e1', e2}$	$\dfrac{\text{Macro Expand}}{X.e \to e}$

"Solves" the Halting problem by always halting.

Just make the problem go away... to somebody else.

Now we can't solve the **done** or **notdone** problem.

User Loop Control: Pass output into input until **done**



Abstract groupings of related things...

Grammars are meta-languages for language decomposition.

Parse Trees are meta-language encoding of language statements.

Translators implement the meta-language to language transform.

Interpreters perform semantics operations on the language types.

Review:

From *text-string* to ***Data*** evaluating ***Data*** as **Code**: