Programming Languages Final Project

Liam Twomey
Dylan Tivnan

Grammar

```
<prog> → <fun> { <fun> }

<fun> → ;<comment>; |fun <id> <lexpr> | <expr>

<expr> → let {<id> := <expr>} in <expr> | <rexpr> { ( and | or ) <rexpr>

                    | not <rexpr> | if <expr> then <expr> else <expr>

                    | switch <int> { case <int>  <expr> {case <int> <expr>}} default <expr>

                    | apply (<id> | ( <lexpr> ) ) <expr>

<rexpr> → <mexpr> [ ( < | > | >= | <= | = ) <mexpr> ]

<mexpr> → <term> { ( + | - ) <term> }

<term> → <factor> { ( * | / | ++ | U | ∩) <factor> }

<factor> → <id> | <int> | <real> | ( <expr> )

                    | hd <factor> | tl <factor>

                    | list( [(<id> | <int> | <real>) {, (<id> | <int> | <real>)}] )

                    | true | false

                    | set[(<id> | <int> | <real>) {, (<id> | <int> | <real>)}]

<lexpr> → <id> => <expr>
```

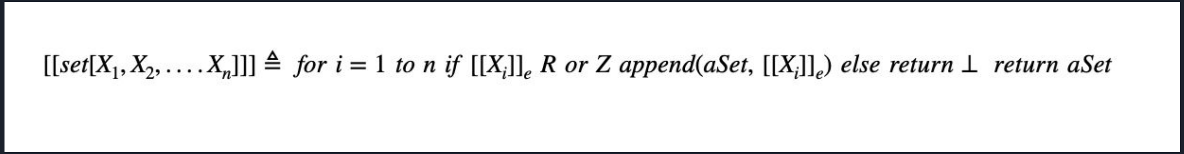
First Order Type Lambda-Expressions and Closures

```
fun name x ~

    y ~

      x * y



apply (name(3))(2)
```

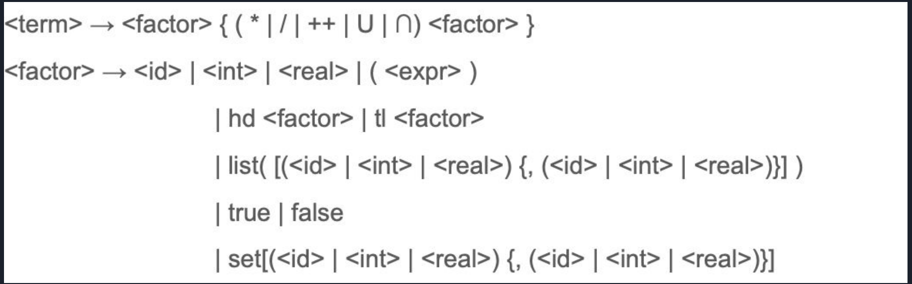
- Created a ClosureNode
- ClosureNode takes in a SyntaxNode that represents the lambda Expression
- Created a method in ClosureNode to set the environment of the closure
- Created a method in ClosureNode that will return the environment of the Closure
- The evaluate of ClosureNode will simply return the copy of the closures environment

Sets

- Sets only allow unique elements, repeated elements will not be added
- Sets have two binary operations, union and intersect which operate on two non-empty sets
- Sets can take in ids ints or reals and there is no type checking with sets as set operations are not dependent on type matching



$$[[set[X_1, X_2, \ldots. X_n]]] \triangleq \mathit{for}\ i = 1\ \mathit{to}\ n\ \mathit{if}\ [[X_i]]_e\ R\ \mathit{or}\ Z\ \mathit{append}(aSet,\ [[X_i]]_e)\ \mathit{else}\ \mathit{return}\ \perp\ \ \mathit{return}\ aSet$$



```
<term> → <factor> { ( * | / | ++ | U | ∩) <factor> }
<factor> → <id> | <int> | <real> | ( <expr> )
                  | hd <factor> | tl <factor>
                  | list( [(<id> | <int> | <real>) {, (<id> | <int> | <real>)}] )
                  | true | false
                  | set[(<id> | <int> | <real>) {, (<id> | <int> | <real>)}]
```

Set Semantics

- The two main operations that we have on sets are union and intersect
- Semantically some set containing elements X1, X2, ……,Xn is denoted as if the evaluation of X1 to Xn under the environment yields an allowed value, then append that evaluation to the set otherwise return null and when done return the set
- The union of two sets S1 and S2 is denoted as, if S1 and S2 are both sets, then return the union of the evaluation of S1 under the environment with the evaluation of S2 under the environment, otherwise return null
- The same goes for intersect, with the difference being the intersection of the two sets

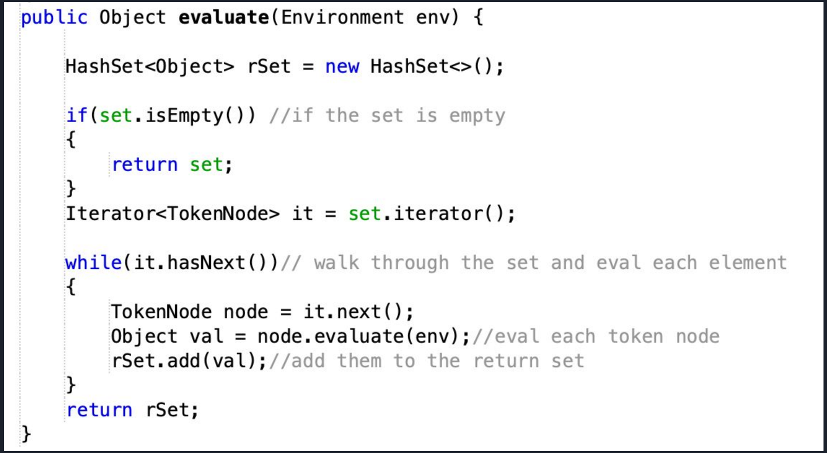
$$[[set[X_1, X_2, \dots . X_n]]] \triangleq \; for\ i = 1\ to\ n\ if\ [[X_i]]_e\ R\ or\ Z\ append(aSet,\ [[X_i]]_e)\ else\ return\ \perp\ \ return\ aSet$$

$$[[S_1\ union\ S_2]] \triangleq if\ [[S_1]]\ and\ [[S_2]]\ are\ sets\ return\ ([[S_1]]_e\ \cup\ [[S_2]]_e)\ else\ return\ \perp$$

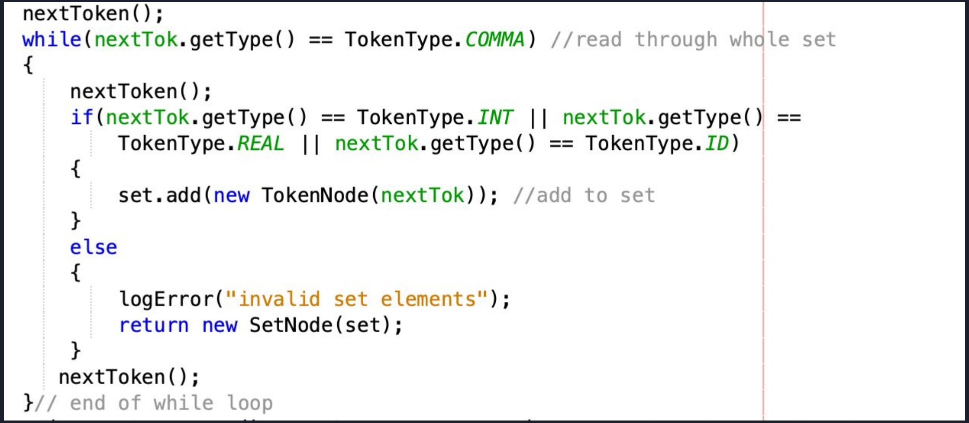
$$[[S_1\ intersect\ S_2]] \triangleq if\ [[S_1]]\ and\ [[S_2]]\ are\ sets\ return\ ([[S_1]]_e\ \cap\ [[S_2]]_e)\ else\ return\ \perp$$

Set Implementation

- Created a new class called SetNode, which takes in a HashSet of TokenNodes
- In the parser sets are handled very similarly to lists, after the left brackets a while loop goes through each token until the right bracket and adds each new element to the set, and at the end it creates the new SetNode passing in the HashSet
- The evaluation of SetNode iterates through the set and evaluates each TokenNode, then adds that to a return set, which is returned once all TokenNodes are evaluated



```java
public Object evaluate(Environment env) {

    HashSet<Object> rSet = new HashSet<>();

    if(set.isEmpty()) //if the set is empty
    {
        return set;
    }
    Iterator<TokenNode> it = set.iterator();

    while(it.hasNext())// walk through the set and eval each element
    {
        TokenNode node = it.next();
        Object val = node.evaluate(env);//eval each token node
        rSet.add(val);//add them to the return set
    }
    return rSet;
}
```



```java
nextToken();
while(nextTok.getType() == TokenType.COMMA) //read through whole set
{
    nextToken();
    if(nextTok.getType() == TokenType.INT || nextTok.getType() ==
        TokenType.REAL || nextTok.getType() == TokenType.ID)
    {
        set.add(new TokenNode(nextTok)); //add to set
    }
    else
    {
        logError("invalid set elements");
        return new SetNode(set);
    }
   nextToken();
}// end of while loop
```

Two New Features

- N-way Selection
  - Switch Statement
- Multiple declaration lets

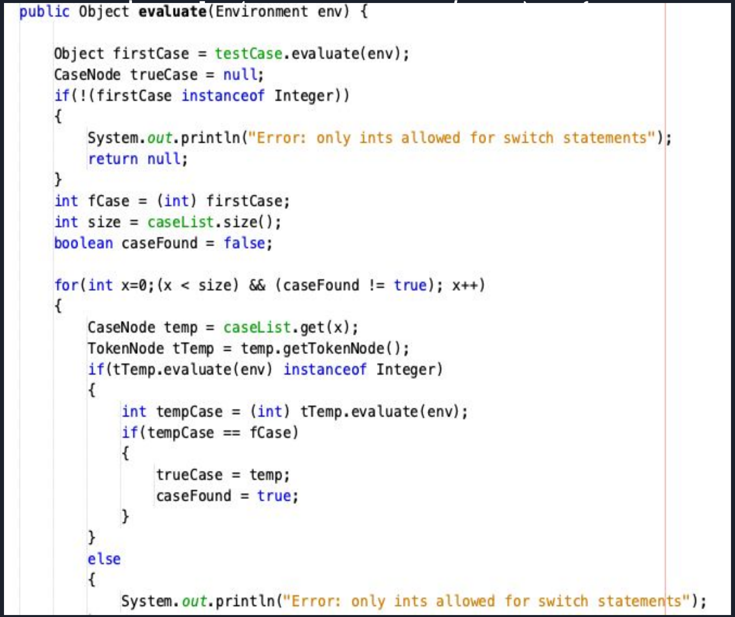
N-Way selection

```
switch(expression) {
  case(1)
    // code block
  case(2)
    // code block
  default
    // code block
 }
```

- Created two new classes called SwitchNode and CaseNode
- Switch node Takes in the test case (TokenNode), a linked list of all CaseNodes , and the default case
- CaseNode will take in the case condition , and the branch of the case
- Created new token types SWITCH, CASE, DEFAULT
- Created a new method in the parser called handleSwitch() that parses the Switch statement

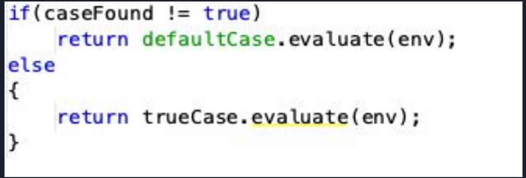
$$[[\mathit{switch}\ E_n\ \mathit{case}\ E_1\ E_2 \ldots\ \mathit{case}\ E_n\ E_{n+1}\ \mathit{default}\ E_k]]_e \triangleq \mathit{if}\ \mathit{switch}\ E_1 = \mathit{case}\ E_1\ \mathit{then}\ [[E_2]]_e\ \mathit{else}\ \mathit{if} \ldots \mathit{else}\ \mathit{if}\ \mathit{switch}\ E_1 = \mathit{case}\ E_m\ \mathit{then}\ [[E_{n+1}]]_e\ \mathit{else}\ \mathit{default}\ [[E_k]]_e$$

N-Way selection Evaluation



```
public Object evaluate(Environment env) {

    Object firstCase = testCase.evaluate(env);
    CaseNode trueCase = null;
    if(!(firstCase instanceof Integer))
    {
        System.out.println("Error: only ints allowed for switch statements");
        return null;
    }
    int fCase = (int) firstCase;
    int size = caseList.size();
    boolean caseFound = false;

    for(int x=0;(x < size) && (caseFound != true); x++)
    {
        CaseNode temp = caseList.get(x);
        TokenNode tTemp = temp.getTokenNode();
        if(tTemp.evaluate(env) instanceof Integer)
        {
            int tempCase = (int) tTemp.evaluate(env);
            if(tempCase == fCase)
            {
                trueCase = temp;
                caseFound = true;
            }
        }
        else
        {
            System.out.println("Error: only ints allowed for switch statements");
```

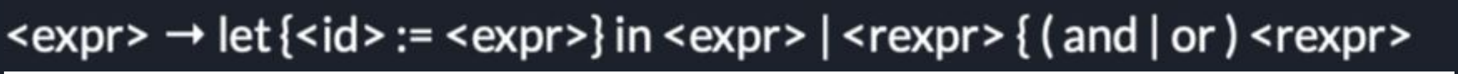
- If the first token is “Switch”, the handleSwitch() method will be called in evalExpr() (similar to the Let, If and Apply statements)
- Presented a while loop that checks all the cases (with the bodies) and added them to the case linked list until the default body shows up
- The case linked list is all CaseNodes
- Once default is reached, it will return a SwitchNode that includes the initial case, a syntax node, and the linked list of all the cases
- SwitchNode is responsible for finding the correct case.
- If the case is found, evaluate the correct branch, else evaluate the default branch



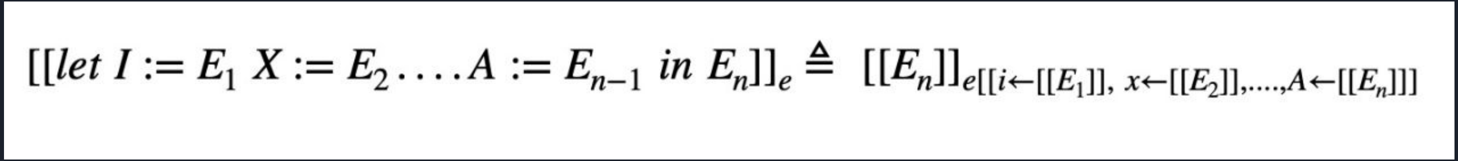
```
if(caseFound != true)
    return defaultCase.evaluate(env);
else
{
    return trueCase.evaluate(env);
}
```

Multiple variable declaration lets

- This allows a single let to declare and evaluate statements with multiple variables
- Like normal let statements, these multiple variable lets have their own scope which is defined by let.
- In terms of the grammar, after the let keyword any amount of id assignments to expressions more than one are allowed and will all be handled with a single in followed by an expression
- An example : let x := 4 y:=6 z:=5 in x + y + z
- Semantically the evaluation under the environment of a let expression that assigns I to E1 X to E2 all the way through A to En-1 in En is denoted as the evaluation of En under the environment where E1 maps to I, E2 maps to X,........, En maps to A



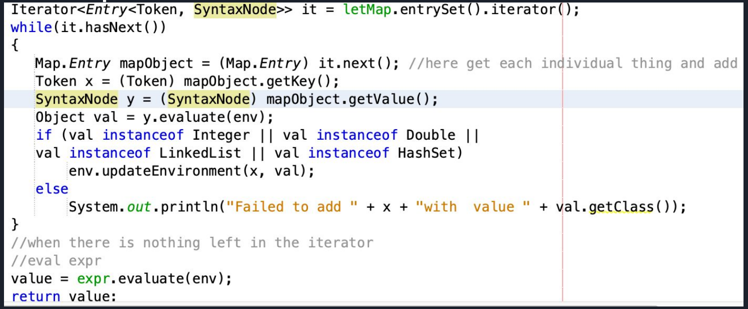
<expr> → let {<id> := <expr>} in <expr> | <rexpr> { (and | or) <rexpr>



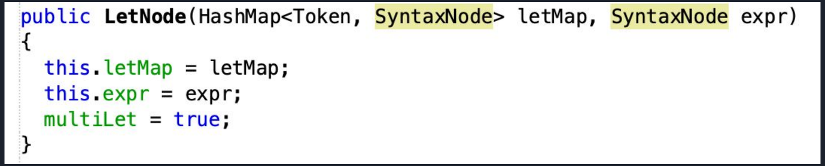
$$[[let\ I := E_1\ X := E_2 \ldots . A := E_{n-1}\ in\ E_n]]_e \triangleq [[E_n]]_{e[[i \leftarrow [[E_1]],\ x \leftarrow [[E_2]], \ldots ., A \leftarrow [[E_n]]]]}$$

Implementation

- Created a new constructor in LetNode
- Takes in a HashMap of Tokens as keys with the syntax nodes that give them value as values and the expression to evaluate, it also sets a boolean multiLet to true
- In the parser, if the next token after the first variable assignment is not IN, then a while loop will run until IN which adds each new token as a key and assignment as a value to a Hashmap, this HashMap is then passed into LetNode along with the expression
- In the evaluation of LetNode if multiLet is true, the HashMap is iterated through
- For every key, we get the value, which is the syntaxNode that gives value, then as long as it's of the allowed type we add each key and value pair to the environment and evaluate the expression when done



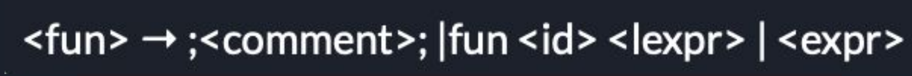
```java
Iterator<Entry<Token, SyntaxNode>> it = letMap.entrySet().iterator();
while(it.hasNext())
{
   Map.Entry mapObject = (Map.Entry) it.next(); //here get each individual thing and add
   Token x = (Token) mapObject.getKey();
   SyntaxNode y = (SyntaxNode) mapObject.getValue();
   Object val = y.evaluate(env);
   if (val instanceof Integer || val instanceof Double ||
   val instanceof LinkedList || val instanceof HashSet)
       env.updateEnvironment(x, val);
   else
       System.out.println("Failed to add " + x + "with  value " + val.getClass());
}
//when there is nothing left in the iterator
//eval expr
value = expr.evaluate(env);
return value;
```



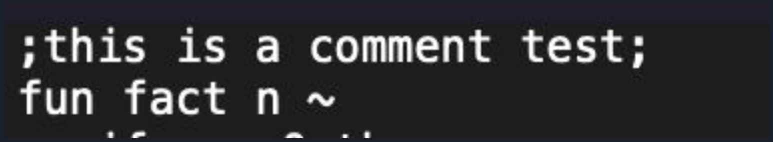
```java
public LetNode(HashMap<Token, SyntaxNode> letMap, SyntaxNode expr)
{
  this.letMap = letMap;
  this.expr = expr;
  multiLet = true;
}
```

Comments

- Comments have a high priority so that they don't get mixed in with other operations
- Comments must be at the top of test case, before function declarations and any other operations otherwise they won't be read, with them at the top they will always be displayed even if parsing fails
- We chose semicolon as the comment indicator
- Comments function as block comments and anything in between the semicolons will be parsed as part of the comment
- In the parser, there is now a while loop in evalFun() where if the first token is a semicolon the while loop will read tokens until the closing semicolon and outputs the string it read as a comment



<fun> → ;<comment>; |fun <id> <lexpr> | <expr>



```
;this is a comment test;
fun fact n ~
```