Highly Organized & Optimized Trading In Exchanges (HOOTIE)

Final Report

HOUTAN M. FANISALEK
COLUMBIA UNIVERSITY

INTRODUCTION

Most traders have little or no technical knowledge and rely on software and human interaction to perform complex trading routines. The primary objective of this language is to provide a way for traders to create an automated black box system that is simple enough to understand and use yet robust enough for high volume trading. The language and compiler will handle all of the details leaving the user free from worrying about actual code.

The HOOTIE language is very simple and those that have the basic knowledge of trading stocks can pick it up very quickly. Each command is written on a single line with no semicolon and each item is separated by spaces. The symbol is in reference to a stock symbol.

QUICK LANGUAGE TUTORIAL

Each HOOTIE program requires at least one EXECUTE block within the .hoot file. Inside each block can be a series of instructions followed by a semi-colon for each instruction. To compile a .hoot file type (where FILE_NAME_LOCATION is the name of your source file):

```
./hooties FILE_NAME_LOCATION.hoot
```

Now you can simply type “ant run” or locate the java executable in the dist folder and run that.

A basic program:

```
// This is a comment (Comments are not used by the program) //
// Hoot_Hoot.hoot //

EXECUTE hello_world {
        PRINT “HELLO STOCK MARKET WORLD”;
        EXIT;
}
```

To run this program

```
./hooties hoot_hoot.hoot
ant run
```

LEXICAL CONVENTIONS

There are five kinds of tokens in this language: identifiers, stock symbols, keywords, strings, and operators. Tabs, blanks, and comments are ignored except when acting as separators. Newlines ignored and semicolons are used to terminate a statement.

COMMENTS

A comment or note is determined by beginning and ending with a double backslash and does not interfere with the execution of the program.

```
// This is a comment //
```

IDENTIFIERS

An identifier is a sequence of letters and digits with the first character being alphabetic including the underscore "_". Upper and lower case letters are NOT considered different.

STOCK SYMBOLS

Stock symbols are always capitalized and start with a dollar sign.

For Example: $MGM

KEYWORDS

The following capitalized keywords are reserved and may not be used as identifiers:

- LOOKUP
- BUY
- SELL
- WHEN
- OTHERWISE
- LOOP
- WAIT
- WAITEXE
- BREAKOUT
- AND
- OR
- NOT
- IS

- PRINT
- EXEUTE
- STOP
- EXIT
- TRUE
- FALSE
- PRICE
- VOLUME
- MCAP
- BETA
- RANGE

NUMBERS

Numbers can be either integer or floating point values with the following restrictions

- Numbers are a sequence of digits from 0 through 9.
  - Example: 3000000
- Floating point values must have a decimal point
  - Example: 3000000.00

STRINGS

A string is a sequence of characters starting and ending with double quotes (“ “). For Example:

“THIS IS A STRING”

OPERATORS

An operator is a specific token which specifies an operation on one or more operand and may be an expression or constant. Arithmetic and comparison operators are left associative, Assignment is right associative, and logical operators have no associativity.

COMPARISON OPERATORS

Operator	Definition	Number Example	Stock Example
<	Less than	a > b	$NTFL IS > 400.00
>	Greater than	a < b	$NTFL IS > 400.00
<=	Less than or equal to	a <= b	$NTFL IS > 400.00
>=	Greater than or equal to	a >=b	$NTFL IS > 400.00
EQUALS	Equal to	a EQUALS b	$NTFL IS > 400.00

ARITHMETIC OPERATORS

Operator	Definition	Number Example	Stock Example
+	Addition	a + b	$NTFL + 400.00
-	Subtraction	a - b	$NTFL - 400.00
*	Multiplication	a * b	$NTFL * 400.00
/	Division	a / b	$NTFL / 400.00
%	Modulus	a % b	$NTFL % 400.00

ASSIGNMENT OPERATORS

Operator	Definition	Number Example	Stock Example
=	Assignment	var = 5	N/A

LOGICAL OPERATORS

Operator	Definition	Example	Stock Example
AND	Logical AND	var1 AND var2	N/A
OR	Logical OR	var1 OR var2	N/A
NOT	Logical NOT	NOT var2	N/A

DECLARATIONS

There are two specific types of declarations. One type is the buying and selling of a stock and the other is the variable declarations.

BUY AND SELL

Buying or selling a stock begins with the BUY or SELL keyword followed by the required stock symbol, and optional conditions, and optional term. For example:

SELL *Symbol Quantity;*
Example: SELL $AIG 2000;

BUY *Symbol Quantity;*
Example: BUY $MGM 2000;

VARIABLES

To declare a double percision variable the ampersand issued with the variable identifier.

@variablename;

The assignment operator can assign values to each variable. For example:

```
@number_of_trades;
number_of_trades = 50;
```

STATEMENTS

Statements are only one per line and executed in order however statements instead the EXECUTE blocks are executed concurrently. Whitespace has no effect.

WHEN STATEMENT

The WHEN statement is in the follow two form

```
WHEN ( condition ) {
    statement
}

WHEN ( condition ) {
    statement
} OTHERWISE {
    statement
}
```

In both cases if the condition is true the first statement after the WHEN is executed. If the statement is false and there is an OTHERWISE clause then that statement will be executed.

Example:

```
WHEN ($NFLX IS > 210.00) {
    PRINT “NETFLIX IS TOO EXPENSIVE!”;
    EXIT;
} OTHERWISE {
    EXIT;
}
```

LOOP STATEMENT

The LOOP statement is in the form

```
LOOP ( condition ) {
    statement
}
```

The statement is executed repeatedly until the value of the condition becomes false.

BREAKOUT STATEMENT

The BREAKOUT statement is put within a WHEN or LOOP statement and causes the termination of that statement.

EXIT STATEMENT

The EXIT statement terminates the execution of the program completely.

PRINT STATEMENT

The print statement can be used to output strings or variables to the terminal. For example:

```
PRINT "HELLO STOCK MARKET WORLD";
```

EXECUTE STATEMENT

The execute statement begins with the EXECUTE keyword and the followed by an identifier.

```
EXECUTE identifier {

}
```

Each execute block is done concurrently and each program needs at least one execute block. For example:

```
EXECUTE a_merger {
        WHEN ($BLOAQ IS >= .09) {
                SELL NFLX 2000;
                EXIT;
        }
}
```

WAITEXE STATEMENT

The WAITEXE before a BUY or SELL and halts the execution until the BUY or SELL finishes. For Example:

```
WAITEXE SELL $NFLX 2000;
```

This will halt execution until 2000 shares of NFLX are sold.

WAIT STATEMENT

The WAIT followed by an integer will halt the execution for a specific number of milliseconds. For example:

```
WAIT 1000;
```

This will halt execution of the program for 1000 milliseconds.

LOOKUP STATEMENT

The LOOKUP statement begins with the LOOKUP keyword followed by the stock symbol and a certain characteristic.

LOOKUP *Symbol [Information]*

INFORMATION

VOLUME
BETA
MCAP
RANGE
PRICE (default)

PROJECT PLAN

The project design and development was divided into three parts.

1. The first important step was to get the Parser & Interpreter developed to a certain point in which basic functionality was implemented which resembled our Language Reference Manual.
2. Next phase we begin by creating the java application that would have core functionality of running and connecting to a firm's platform API and providing stock data.

3. The final installment would be to combine our generated code with the core java application to provide a complete application for a user.

PROJECT TIMELINE

Date	Milestone
02/09/2011	Proposal
03/20/2011	Basic Parser
03/21/2011	Language Reference Manual
04/19/2011	Java Application Development
05/02/2011	Combining Parser and Java Application
05/06/2011	Debugging and Testing
05/09/2011	Final Report

SOFTWARE DEVELOPMENT ENVIRONMENT

Since this project only involved one developer a version control system was not used however the project was maintained with numerous (daily) backups and stored on a RAID 5 drive configuration. The code was compiled and run on two different machines (Dual Core and Hex Core) and operating systems (Mac OS X x64 and Windows 7 Ultimate SP1 x64); both yielding the same results.

The interpreter was written with O'Caml and gedit and compiled with ocamlc, ocamllex, and ocamlyacc from the terminal. For the java side, Netbeans 7.0 was used to compile and debug.

PROGRAMMING SYTLE

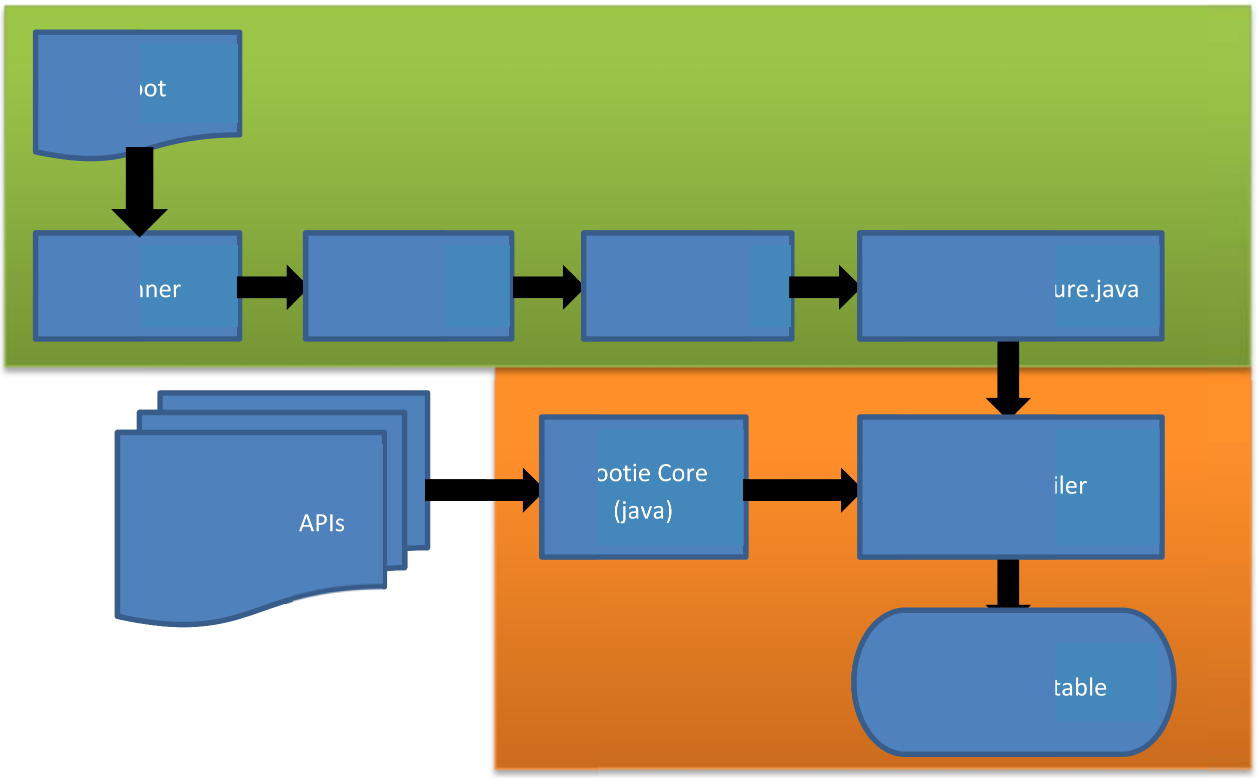
The programming style for HOOTIE is very simple. Each block should be indented to provide for readability. Identifiers should begin with a lowercase and be descriptive of what they are representing whenever possible.

For Example:

```
EXECUTE inRange {
     LOOP($NFLX IS > 222.50 AND $NFLX IS < 500) {
          BUY $NFLX 200;
          BREAKOUT;
     }
}
```

ARCHITECHTURAL DESIGN

The architecture of the HOOTIE compiler contains three major components.



The .hoot file is first run through the scanner, parser, and translator in which a LanguageStructure.java file is generated that contains the java code that will run with the Hootie Core. Each EXECUTE block is transformed into a separate thread that will work with the API that is currently implemented to buy and sell stocks based on given conditions. Each trading firm platform needs to have StockTradingAPI interface implemented. For the purposes of this project a sample TestTradingFirm was used with basic information. Combining both will give us the Hootie java executable that can be run on any machine with JRE 6 or JDK 1.6 installed.

TEST PLAN

It is impossible to test every single path that a compiler can generate. However, the most important paths and basic control structures can be tested to provide a certain level of sanity. Two sample test cases are shown below:

TEST 1

INPUT

```
// THIS IS AN OPERATOR TEST //

EXECUTE testerA {
      LOOP($NFLX IS > 222.50 AND $NFLX IS < 500) {
            BUY $NFLX 200;
            BREAKOUT;
      }
}

EXECUTE testerB {
      LOOP($NFLX IS > 222.50 OR $NFLX IS < 500) {
            BUY $NFLX 200;
            BREAKOUT;
      }
}

EXECUTE testerC {
      @temp;
      temp = temp + 1;
      PRINT "RUNNING NOW";
      WHEN(temp EQUALS 10) {
            EXIT;
      }
      WHEN($NFLX IS > 222.50 != $NFLX IS < 500) {
            BUY $NFLX 200;
            BREAKOUT;
      }
}
```

OUTPUT

```
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
      [java] RUNNING NOW
```

```
     [java] RUNNING NOW
     [java] hootie.exceptions.InsufficientFunds: Insufficient funds in
the account for the transaction!

     [java] hootie.exceptions.InsufficientFunds: Insufficient funds in
the account for the transaction!
```

TEST 2

INPUT

```
// THIS IS A EXIT AND WAIT TEST //

EXECUTE tester {
      @test;
      @bools;
      LOOP($NFLX IS > 222.50) {
            WAIT 2000;
            test = 50;
      }
}

EXECUTE testerB {
      PRINT "THIS IS A TEST";
      WAITEXE SELL $NFLX 200;
      LOOP($NFLX IS < 500.50) {
            EXIT;
      }
}
```

OUTPUT

```
     [java] THIS IS A TEST

     [java] hootie.exceptions.NotInPortfolio: Symbol: $NFLX was not
found in the users portfolio!
```

TEST CASES

A test case developed for each of the specific types of blocks as well as individual operators. Certain test cases were created to cause runtime errors. There were also test cases created to cause compiling errors (not included with tests but can be easily reproduced). The output of each test case was compared with what should have been logically expected.

LESSONS LEARNED

Writing a compiler is a difficult and complex task. There should be ample time to design and think through all possible outcomes. Although it is very hard to keep on schedule because of work and school, I tried my best to keep a consistent level work throughout the time period. There is not enough time in a term to create a fully functional compiler that tests and runs every path. Going through the process, however, helps understand the process in which your source is generated into machine code. As a recommendation to future teams, it is important to start the design early so you have enough time to go back and change things that may not necessarily work.

CODE LISTING

The following is a complete listing of the source code used by the HOOTIE Project:

SCANNER.MLL

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "//"          { comment lexbuf }           (* Comments *)
| '@'      { AMP }
| '('           { LPAREN }
| ')'           { RPAREN }
| '{'           { LBRACE }
| '}'           { RBRACE }
| ';'           { SEMI }
| ','           { COMMA }
| '+'           { PLUS }
| '-'           { MINUS }
| '*'           { TIMES }
| '/'           { DIVIDE }
| '='           { ASSIGN }
| "AND"         { AND }
| "OR"          { OR }
| "NOT"         { NOT }
| "EQUALS"      { EQ }
| "!="          { NEQ }
| '<'           { LT }
| "<="          { LEQ }
| ">"           { GT }
| ">="          { GEQ }
| "WHEN"        { IF }
| "OTHERWISE"  { ELSE }
| "LOOP"        { WHILE }
| "BREAKOUT"    { RETURN }
| "BUY"         { BUY }
| "SELL"   { SELL }
| "EXECUTE"     { EXECUTE }
| "IS"          { IS }
| "PRICE"  { PRICE }
| "VOLUME" { VOLUME }
| "MCAP"   { MCAP }
| "BETA"   { BETA }
| "RANGE"  { RANGE }
| "LOOKUP" { LOOKUP }
| "WAIT"   { WAIT }
| "WAITEXE"     { WAITEXE }
| "EXIT"   { EXIT }
| "PRINT"  { PRINT }
| "TRUE"   { TRUE }
| "FALSE"  { FALSE }
| ['"']['a'-'z' 'A'-'Z' '0'-'9' '_' ' ']*['"'] as lxm { TEXT(lxm) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['0'-'9']*['.']['0'-'9']* as lxm { FLOAT (float_of_string lxm) }
| '$'['A'-'Z']+ as lxm { SYMBOL(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

and comment = parse
  "//" { token lexbuf }
| _    { comment lexbuf }
```

PARSER.MLY

```
%{ open Ast %}

%token BUY SELL EXECUTE LOOKUP EXIT PRINT
%token WAIT WAITEXE
%token IS PRICE VOLUME MCAP BETA RANGE
%token <string> SYMBOL
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AND OR NOT TRUE FALSE
%token RETURN IF ELSE FOR WHILE AMP
%token <float> FLOAT
%token <int> LITERAL
%token <string> ID
%token <string> TEXT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
```

```
%left IS
%left EQ NEQ
%left AND OR NOT
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   EXECUTE ID LBRACE vdecl_list stmt_list RBRACE
     { { fname = $2;
       locals = List.rev $4;
       body = List.rev $5 } }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   AMP ID SEMI { $2 }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

cmpr:
         { ".price"  }
  | VOLUME { ".volume" }
  | PRICE  { ".price"  }
  | MCAP   { ".mcap"   }
  | BETA   { ".beta"   }
  | RANGE  { ".range"  }

stmt:
    expr SEMI { Expr($1) }
  | RETURN SEMI { Return }
  | WAITEXE stmt { Waitexe($2) }
  | WAIT LITERAL SEMI { Wait($2) }
  | EXIT SEMI    { Exit }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | BUY SYMBOL LITERAL SEMI { Buy($2, $3) }
  | SELL SYMBOL LITERAL SEMI { Sell($2, $3) }
  | LOOKUP SYMBOL cmpr SEMI { Lookup($2, $3) }
  | PRINT TEXT SEMI { Print($2) }

expr:
    LITERAL          { Literal($1) }
  | FLOAT       { Price($1) }
  | ID               { Id($1) }
  | TRUE        { True }
  | FALSE       { False }
  | SYMBOL IS   cmpr { Symbol($1, $3) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND     expr { Binop($1, And,   $3) }
  | expr OR      expr { Binop($1, Or,    $3) }
  | expr NOT     expr { Binop($1, Not,   $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | LPAREN expr RPAREN { $2 }
```

AST.MLI

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater
| Geq | And | Or | Not

type expr =
    Literal of int
  | Price of float
  | Symbol of string * string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | True
  | False
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return
  | Wait of int
  | Waitexe of stmt
  | Exit
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Buy of string * int
  | Sell of string * int
  | Lookup of string * string
  | Print of string

type func_decl = {
    fname : string;
    locals : string list;
    body : stmt list;
  }

type program = string list * func_decl list
```

PRINTER.ML

```
open Ast

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Price(f) -> string_of_float f
  | Symbol(s, c) -> "api.lookup(\"" ^ s ^ "\")" ^ c
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
     Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||" | Not -> "!") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | True -> "true"
  | False -> "false"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return -> "break;\n";
  | Wait(i) -> "sleep(" ^ string_of_int i ^ ");"
  | Waitexe(s) -> "\nnew Thread( \n new Runnable() { \npublic void
run() {" ^
   " try { " ^ string_of_stmt s ^ " } \ncatch (Exception e) {
\ne.printStackTrace();" ^
   " } \n } \n}).start();"
  | Exit -> "System.exit(0);"
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
  | Buy(e, s) -> "api.buy(\"" ^ e ^ "\", " ^ string_of_int s ^ ");\n"
  | Sell(e, s) -> "api.sell(\"" ^ e ^ "\", " ^ string_of_int s ^
");\n"
  | Lookup(s, c) -> "api.lookup(\"" ^ s ^ "\")" ^ c
  | Print(s) -> "api.print(" ^ s ^ ");"


let string_of_vdecl id = "private double " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  " class " ^ fdecl.fname ^ " extends Thread {\n" ^
  "   private boolean running = true; \n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  "    public void run() { \n" ^
  " while(running) {\n" ^
  " try { \n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "} catch (Exception e) { \n System.out.println(e); \n running =
false; " ^
  "}\n   }\n  }\n }\n"

let string_of_threads tdecl =
  let rec helper i = function
     [] -> "\n";
     | h::t -> (helper(i - 1) t) ^
  "     threads[" ^ string_of_int i ^ "] = new " ^ h.fname ^ "();\n"
  in helper ((List.length tdecl) - 1) tdecl


let string_of_program (vars, funcs) =
  "package hootie.generated;\n" ^
  "import hootie.connector.StockTradeAPI;\n\n" ^
  "public class LanguageStructure { \n" ^
  "    private StockTradeAPI api;\n" ^
  "    public Thread[] threads = new Thread[" ^ string_of_int
(List.length funcs) ^ "];\n" ^
  "    public LanguageStructure(StockTradeAPI api) {\n" ^
  (string_of_threads funcs) ^
  "         this.api = api; "^
  "\n    }" ^
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs) ^
  "}\n"
```

HOOTIE.ML

```
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let listing = Printer.string_of_program program in
  print_string listing
```

MAIN.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie;

import hootie.firms.TestTradingFirm;
import hootie.generated.LanguageStructure;

/**
 *
 * @author Houtan Fanisalek
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        TestTradingFirm etrade = new TestTradingFirm();
        LanguageStructure executor = new LanguageStructure(etrade);
        Thread[] execute = executor.threads;
        try {
            for (int i = 0; i < execute.length; i++) {
                execute[i].start();
            }
            for (int i = 0; i < execute.length; i++) {
                execute[i].join();
            }
        } catch (InterruptedException ignore) {
            //Do nothing
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

STOCK.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.connector;

/**
 *
 * @author Houtan Fanisalek
 */
public class Stock {
    public double price;
    public int volume;
    public double beta;
    public double range;
    public double mcap;
    public String Name;

    public Stock () {
        Name = "$EMPTY";
        price = 0.0;
        volume = 0;
        range = 0.0;
        mcap = 0.0;
    }
}
```

STOCKTRADEAPI.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.connector;

import hootie.exceptions.BadSymbol;
import hootie.exceptions.InsufficientFunds;
import hootie.exceptions.NotInPortfolio;

/**
 *
 * @author Houtan Fanisalek
 */
public class StockTradeAPI {

    public StockTradeAPI ()
    {

    }

    public synchronized void print(String output) {
        System.out.println(output);
    }

    public synchronized Stock lookup(String stock) throws BadSymbol
    {
        return new Stock();
    }

    public synchronized boolean buy(String stock, int quant)
            throws BadSymbol, InsufficientFunds {
```

```
        return true;
    }

    public synchronized boolean sell(String stock, int quant)
            throws BadSymbol, NotInPortfolio {

        return true;
    }

}
```

TESTTRADINGFIRM.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.firms;

import hootie.connector.Stock;
import hootie.connector.StockTradeAPI;
import hootie.exceptions.BadSymbol;
import hootie.exceptions.InsufficientFunds;
import hootie.exceptions.NotInPortfolio;

/**
 *
 * @author Houtan Fanisalek
 */
public class TestTradingFirm extends StockTradeAPI {

    @Override
    public synchronized void print(String output) {
        System.out.println(output);
    }

    @Override
    public synchronized Stock lookup(String stock) throws BadSymbol {
        Stock s = new Stock();
        if (stock.equalsIgnoreCase("$NFLX")) {
            s.price = 250.0;
            s.volume = 20000000;
        } else if (stock.equalsIgnoreCase("$AIG")) {
            s.price = 24.0;
            s.volume = 44444444;
        } else {
            throw new BadSymbol(stock);
        }
        return s;
    }
    @Override
    public synchronized boolean buy(String stock, int quant)
            throws BadSymbol, InsufficientFunds {

        if (stock.equalsIgnoreCase("$NFLX")) {
            throw new InsufficientFunds();
        } else if (stock.equalsIgnoreCase("$AIG")) {
            return true;
        } else {
            throw new BadSymbol(stock);
        }
    }

    @Override
    public synchronized boolean sell(String stock, int quant)
            throws BadSymbol, NotInPortfolio {

         if (stock.equalsIgnoreCase("$NFLX")) {
            throw new NotInPortfolio(stock);
        } else if (stock.equalsIgnoreCase("$AIG")) {
            return true;
        } else {
            throw new BadSymbol(stock);
        }
    }
}
```

BADSYMBOL.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.exceptions;

/**
 *
 * @author Houtan Fanisalek
 */
public class BadSymbol extends Exception {

    public BadSymbol(String Symbol) {
        super("Symbol: " + Symbol + "does not exsist");
    }
}
```

INSUFFICIENTFUNDS.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.exceptions;

/**
 *
 * @author Houtan Fanisalek
 */
public class InsufficientFunds extends Exception {

    public InsufficientFunds() {
        super("Insufficient funds in the account for the
transaction!");
    }

}
```

NOTINPORTFOLIO.JAVA

```
/*
 * Houtan Fanisalek
 * Copyright (c) 2011
 */
package hootie.exceptions;

/**
 *
 * @author Houtan Fanisalek
 */
public class NotInPortfolio extends Exception {

    public NotInPortfolio(String Symbol) {
        super("Symbol: " + Symbol + " was not found in the users
portfolio!");
    }
}
```