Traders Joe

Author: Monu Chacko (UNI: mc3574)
Course: COMS W4115 (Summer 2011)
Email: mc3574@columbia.edu

Contents

Introduction

Traders Joe is a language that provides high level technical analysis capabilities for any product like forex, security etc. This language is easy to use and has built in functions that help in technical analysis. Traders Joe understands mathematical notations and uses algorithms for modeling and simulating any trading models.

Traders Joe is written using OCaml and is optimized for technical analysis. It uses the state-of-the-art algorithms that are widely used by the technical analysis community.

Language Tutorial

Traders Joe is a Visual Basic type language that is used for designing technical analysis models. It is an easy to use language that is used to create various trading models. This quick start section helps you to get started with the basic operations.

Quick Start

Task 1 – Write 'if statement'

Steps:

1. Open your notepad and type the following code

```
main()

begin
```

```
        int intA;
        intA = 100;

        if (intA == 100)
        begin
            print(intA);
        end
    end
```

2. Save the file as “sample-if.joe” in the *test* directory.
3. To execute type: ‘*./TradersJoe -c < ./tests/sample-if.joe*’ in the program shell. Press enter.

```
$ ./TradersJoe -c < ./tests/sample-if.joe
100
```

The entry function for a program is its main() function. This is a required function and other methods can branch from it. A program file uses the ‘*.joe’* extension.

Task 2 – Calling multiple functions

Steps:

1. Open your notepad and type the following code

```
main()
begin
    print(GetValue(20));
end

GetValue(intA)
begin
    int intResult;

    [* If intA is lesser then 100 then return 50 else return 100 *]
    if (intA < 100)
    begin
        intResult = 50;
```

```
        end
        else
        begin
            intResult = 100;
        end

        return intResult;
    end
```

2. Save the file as “sample-CallingFunctions.joe” in the *test* directory.
3. To execute type: ‘*./TradersJoe -c < ./tests/ sample-CallingFunctions.joe*’ in the program shell. Press enter.

```
$ ./TradersJoe -c < ./tests/sample-callingfunctions.joe
50
```

Comments are enclosed with in *[* Comment *]* and are ignored by the program. It is a good practice to comment your code for readability.

Language Manual

Lexical Conventions

Character Set

Traders Joe uses ASCII character.

Keywords

Following are the keywords used by this language. These reserved words cannot be used as identifiers in the program. Keywords are case sensitive.

Alert	Loop
Break	Low
Case	Median
Catch	Open
Close	Sma

Else	Sum
Elseif	Try
Ema	While
For	
High	

Constants

Integer Constants

An integer constant can contains series of number from 0 to 9. For example: 2000023

Floating Point Constants

Floating point are real numbers that contain fraction part. These numbers has an integer part, a decimal point and numbers that follow the decimal points. The character ‘e’ is optionally assigned after the decimal point.

String Constants

String constant are enclosed within double quotes. It can contain escape characters that is used for formatting messages.

Operators

Operators allow users to perform various operations. The following list contains various tokens that are used to assess relationship, perform logical operation, and calculate numerical values or to assign values to a variable.

Relational operators	
Equal To	= =
Not Equal To	!=
Less Than	<
Greater Than	>
Less Than or Equal To	<=
Greater Than or Equal To	>=
Logical operators	

Not	!
And	&&
Mathematical operators	
Add	+
Subtract	-
Multiply	*
Divide	/
Power	^
Assignment operator	
Assign	=

Punctuators

Following punctuators are used to enclose or separate values or arguments. For example a string value is enclosed within double quotes “string value” and a character value is enclosed within single quotes ‘a’.

Punctuators	Description
" "	Encloses string value
' '	Encloses character value
( )	Groups arguments in a function
,	Separates arguments in a function

Comments

Comments are started with an open square bracket immediately followed by a start and terminated with a star immediately followed by a close square bracket. Any text between [* and *] are ignored. Comments helps programmers describe the function they are writing. Any code block enclosed between comments begin and end tag is also ignored.

For example:

[* *This comment is ignored by the program* ****]**

White Spaces

White spaces like tabs, line feeds etc are used to structure the code block. Developers may use them to align the code block to make their code readable.

Data Types

Simple Types

Type	Description
boolean	Boolean i.e. true or false
Int	Integer values e.g. 1, 2, 3
Float	Floating point values e.g. 1.2001
String	String value e.g. "string value"
Char	Character value e.g. 'a'

Complex Types

Objects are classes that contain types, variables, functions etc that represents a model. Trading algorithms can be organized into classes and initialized as objects at runtime. This is useful when comparing the performance of two trading models.

Statements

Statements are sequence of statements or expressions that are executed sequentially. They can be complex or simple.

Compound Statements

Compound statement is collection of statements or expressions that are enclosed within a code block. A code block can be a function that is enclosed within a '*begin*' and '*end*' keyword.

Expression Statements

An expression is a statement that is separated by an operator.

For example

```
price = 1.2005;
```

Control Statements

While loop

The ‘while loop’ allows conditional execution of the enclosed statement. The statement is executed as long as the condition in the while expression is true. Users should make sure this expression does not result in an infinite loop.

Example:

```
while (close = 10)
begin
      [* statement *]
end
```

Infinite loop:

```
while (true)
begin
      [* statement *]
end
```

For loop

```
for expression 1; expression 2; expression 3
begin
      [* statement *]
end
```

If else condition

If… else are conditional statements and has the following syntax. If the condition is true then the first statement is executed. If the condition is false then the second statement is executed.

```
if (close = 20)
begin
     [* statement *]
end
```

```
else
begin
     [* statement *]
End
```

Functions

Built-in Functions

Alert

The alert function is used to return information to the user. A statement enclosed in double quotes should follow the alert command. Alert statement without proper string statement will return “Improper syntax error”.

Example:

```
alert(“The opening price is greater than the previous closing price”);
```

Trace

Trace is used to generate valuable information about the program and is used for debugging. Trace outputs suppressed when moving the code to the production environment. A string statement should follow the trace command.

Example:

Trace(“Reached the GetPrice function”);

EMA

Exponential Moving Average (EMA) is a type of moving average that reacts faster to recent price changes. This function takes time period and calculating period as its parameter and returns a floating point number.

Example:

ema = 10; [* returns the previous days EMA that is calculated using 10 days average *]

Open

This function is used to find the opening price of a stock or currency pair. It takes time period integer as its parameter and returns an opening price floating point number.

Example:

Open = 0; [* returns the current opening price. *]

Close

This function is used to find the closing price of a stock or currency pair. It takes time period integer as its parameter and returns a closing price floating point number.

Example:

```
Close = 1; [* returns the closing price on the previous day. *]
```

High

This function is used to find the highest price of a stock or currency pair on a given day. It takes time period integer as its parameter and returns a floating point number.

Example:

```
High = 1; [* returns the highest price on the previous day. *]
```

Low

This function is used to find the lowest price of a stock or currency pair. It takes time period integer as its parameter and returns the lowest price.

Example:

```
Low = 1; [* returns the lowest price on the previous day. *]
```

Error Handling

Following are the built in errors that the system throws.

Err Code	Error Code	Description
1	FATAL_ERROR	Unexpected error occurred
2	SYNTAX_ERROR	Error in syntax
3	OUT_OF_BOUND	The data in the array you are referring does not exist
4	TYPE_MISMATCH	Data type is not in the correct format
5	MISSING_INPUT_FILE	Input file is missing
6	FORMAT_ERROR	Input file is not in the correct format

Errors can be handled using try and catch error block. The *try… catch* block without the *throw* block will suppress errors. It is a best practice to handle all errors appropriately. Following code catches error and suppresses it.

Suppress Error

```
[* Suppressing error *]
try
begin
    [* Statement *]
end
```

Catching Error

```
[* Suppressing OUT OF BOUND error *]
try
begin
    [* Statement *]
end
catch
begin
    [* Handle error *]
```

```
end
```

Custom Error

```
[* Custom error *]
try
begin
     throw(1); [* Error code 1 *]
end
catch
begin
     [* Statement *]
end
```

Time Series Inputs

Time Series data are inputs for various technical analysis operations. These inputs are stored in a file and passed to the language. This is a tab delimited file in the following format.

Symbol{tab}DataTime{tab}Open{tab}Close{tab}High{tab}Low

There is no restriction on the number of lines of data that the system can accept. Following is a sample time series data.

Symbol	DateTime	Open	Close	High	Low
EURUSD	1/5/2011	1.5095	1.5105	1.5109	1.5085
EURUSD	1/4/2011	1.5031	1.5095	1.5095	1.5021
EURUSD	1/3/2011	1.5085	1.5031	1.5085	1.5031
EURUSD	1/2/2011	1.5021	1.5085	1.5021	1.5031
EURUSD	1/1/2011	1.5001	1.5021	1.5021	1.5001

Usage:

```
 [* This will warn if the current ema 10 is less than todays close *]
if (ema < 10)
begin
     warn("Buy EURUSD");
end
```

Scope Rules

Global variables:

The global variable many be declared anywhere in the program outside the local code blocks. These variables can be accessed by functions and will retain its value. A variable that is defined with the same name within a code block will however has its own value that is initialized inside a local code block.

Local variables:

A code block can be a function or a control statement like a while loop. Variables declared within these blocks are local to its container. If a code block is a control statement then it should begin with the '*begin*' keyword and end with an '*end*' keyword. Variable scope within a nested code block is local to its container.

Example:

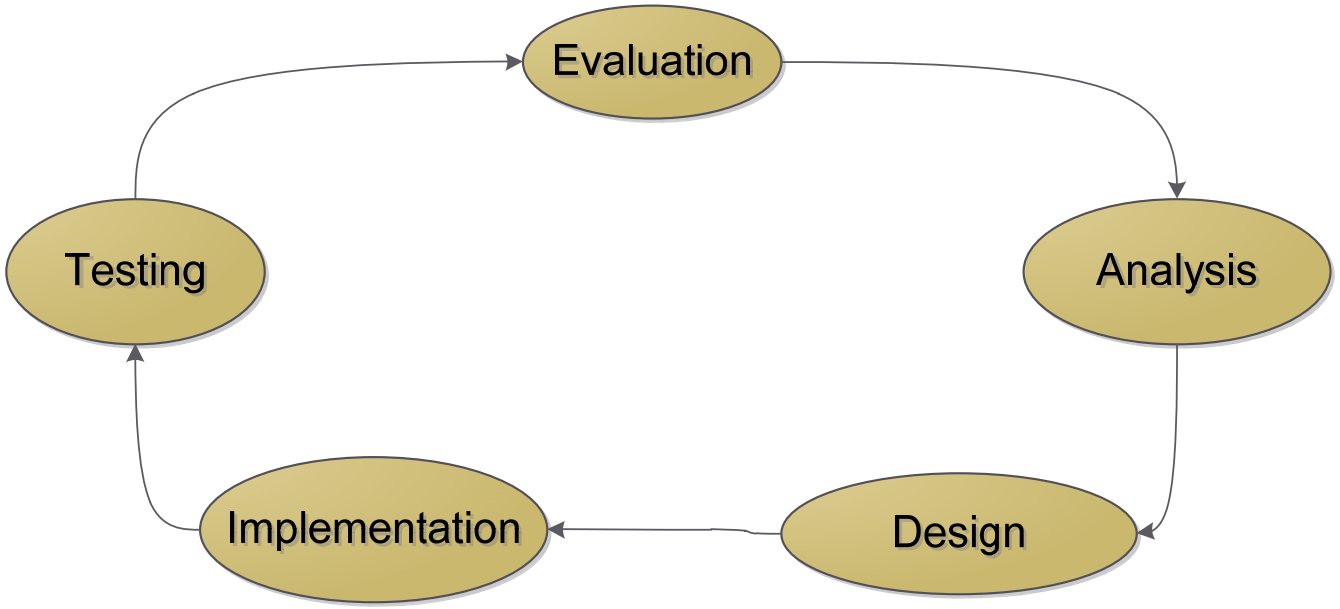
```
int targetPrice;
targetPrice = 1.2002;                    [* Global variable *]

function GetTargetPrice
begin
     int targetPrice;
     targetPrice = 1.4000;          [* Variable local to the GetTargetPrice
function *]
     while(close < ema)
     begin
          int targetPrice;
          targetPrice = 1.2500;          [* Variable local to the control
     statement *]
     end
end
```

Project Plan

The **Software Development Life Cycle (SDLC)** used in this process consists of Analysis, Design, Implementation, Testing and Evaluation. I felt it appropriate to use the **Agile software development methodology** for this process. This **iterative process** provides flexibility to the entire process.

SDLC Process



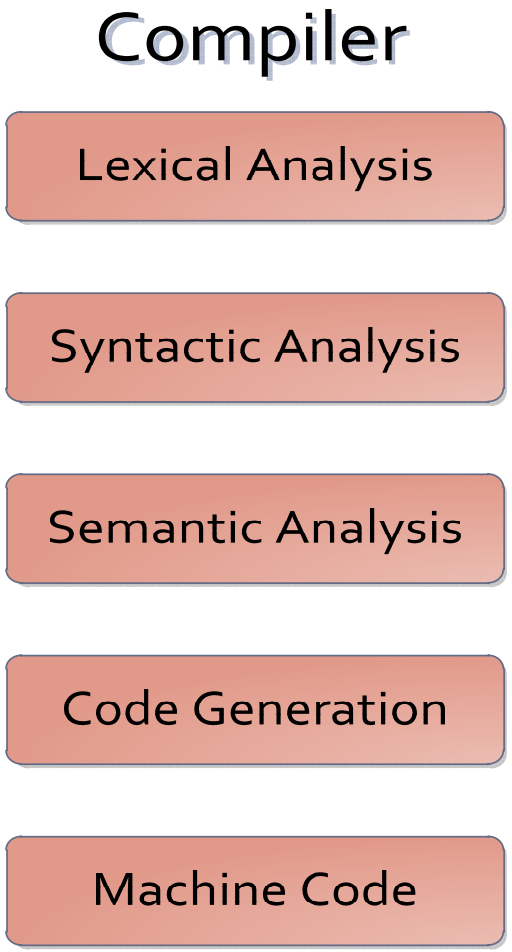
Analysis

This phase defined the broad framework of the language. We chose to create a lightweight language that can be used by traders to perform technical analysis. The first version of this language is considered lightweight that allows traders with basic technical analysis operations. The target user should have basic programing knowledge to be able to create models for trading.

Design

After defining the framework of the language I decided to design a compiler instead of an interpreter. The compiler contains its own binary code executable and does not have to rely on another program to do that. The result is faster execution of the program. This language has a front end and a back end. The **front end** checks for the syntax and semantics and make sure the grammar is correct. The **back end** does the compilation and execution. The program was broken into various units performing its own function. The scanner, parser, ast, byte code, compiler, interpreter and execution units performed its own functions.

OCaml was used to develop this language. Its static type system, type inference, tail recursion, pattern matching, lexical closures, exception handling, and garbage collection makes it the preferred choice for creating a language.



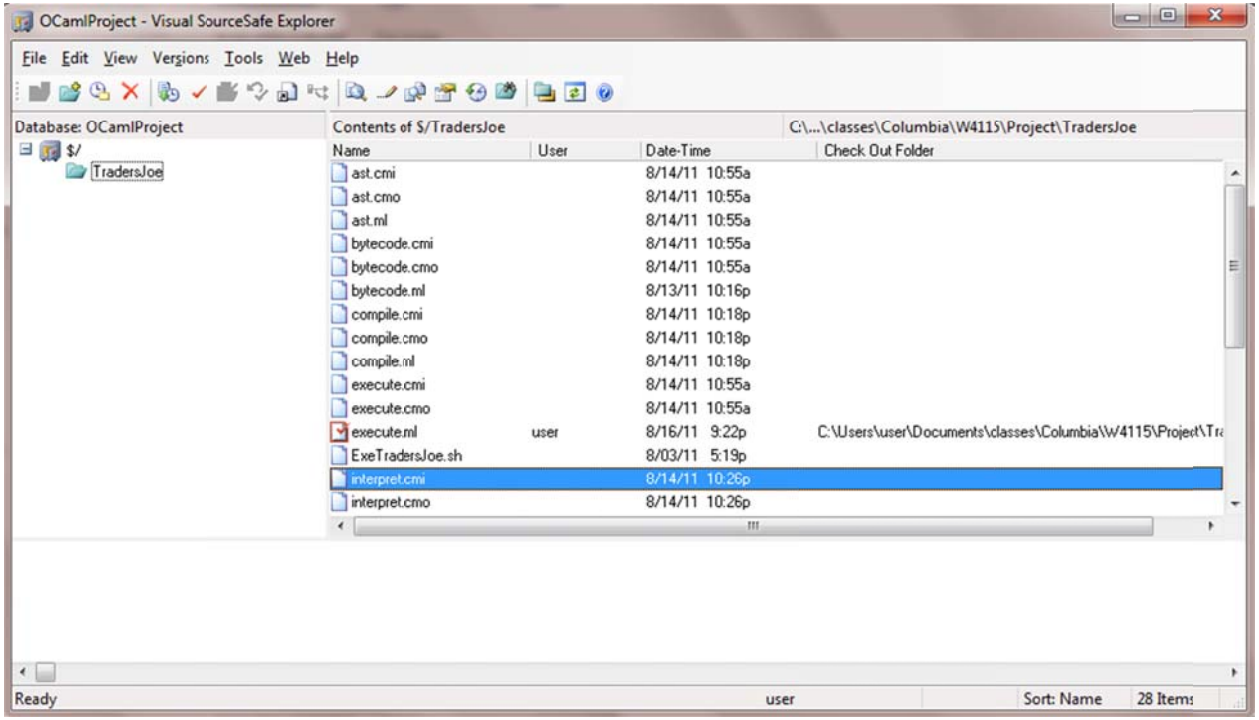
Implementation

Before implementing the above design, I had to put in place the required infrastructure like selecting the source control, testing strategy, disaster recovery strategy and project timeline etc.

Source Control

Source Control is an important part of software development. For this development I used Visual Source Safe because of it ease and version control abilities. In addition to VSS I also used branching for disaster recovery. All files were stored into additional drives like flash drives ready to be recovered in the event of a computer crash. The **Recovery Point Objective (RPO)** was one day and **Recovery Time Objective (RTO)** was ten minutes. This was adequate for me because I was a team of one.

In a larger team it makes sense to choose source controls that has project management component built into it. In addition to the above it allows for team collaboration functions like discussion, tasks and status reports.



Testing

See testing section below

Evaluation

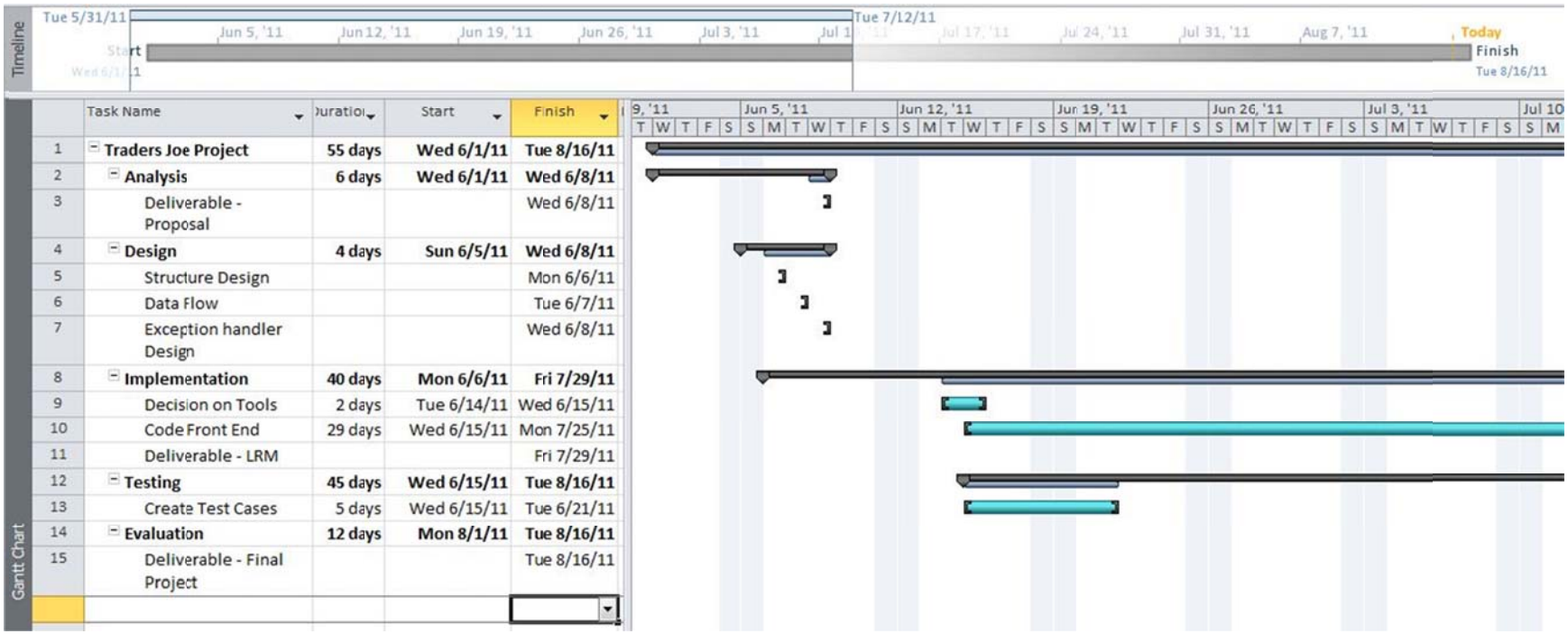
Checkpoints:

1. Make sure all the test cases are passed.
2. Check in all the files in the source control.
3. Take a backup of all files in the jump drive
4. Check the project timeline to make sure the project is on schedule

Team Members

- Army of one

Project Time Line



Development Environment

- Language (both front end and back end) was developed using OCaml.
- Program execution was done on using cygwin's bash shell.
- Visual Source Safe was used for source control.
- Microsoft Project 2010 was used for project scheduling.

Project Logs

Sno	Items	Target Date	Member	Status
1	Plan for the language	6/1/2011	Monu	
2	Make Proposal document	6/8/2011	Monu	
3	**Architectural Design**		Monu	
4	Structure Design	6/6/2011	Monu	
5	Data Flow Design	6/7/2011	Monu	
6	Exception Handler Design	6/8/2011	Monu	
7	**Coding**		Monu	
8	Scanner	7/25/2011	Monu	
9	Parser	7/25/2011	Monu	
10	Ast	7/25/2011	Monu	
11	Bytecode	7/25/2011	Monu	
12	Compile	7/25/2011	Monu	
13	Execute	7/25/2011	Monu	
14	Interpret	7/25/2011	Monu	
15	Makefile	7/25/2011	Monu	
16	Test Cases	7/25/2011	Monu	
17	LRM	7/29/2011	Monu	
18	QA	8/10/2011	Monu	
19	Final Evaluation	8/14/2011	Monu	
20	Final Proposal	8/16/2011	Monu	

Architectural Design

Structure

This language was divided into front end and back end. The scanner, parser, ast, byte code, compiler, interpreter, execution and traders joe formed the structure of the program. Organizing the language into various functional modular units gave clarity to the structure of the language. Following is the structure of the language.

Structure

Scanner

Parser

Ast

Byte Code

Compiler

Interpreter

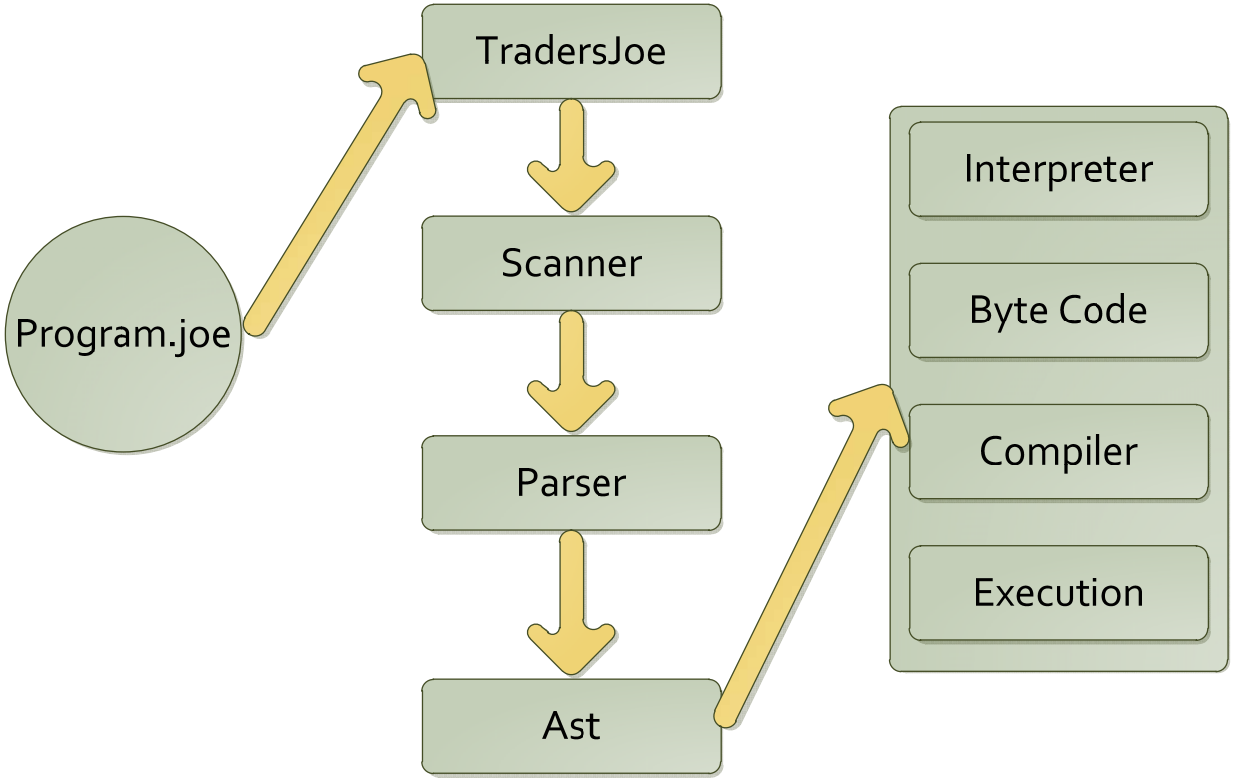
Execution

TradersJoe

Data Flow

The language is written into a file with .joe extension. The TradersJoe executable reads the character stream from the .joe file and performs lexical analysis and then passes it to the parser to do a syntactic analysis. The abstract syntax tree checks for the structure of the program. The program is now ready to be interpreted and executed. The byte code is used for various stack operations. The language is executed.

Data Flow

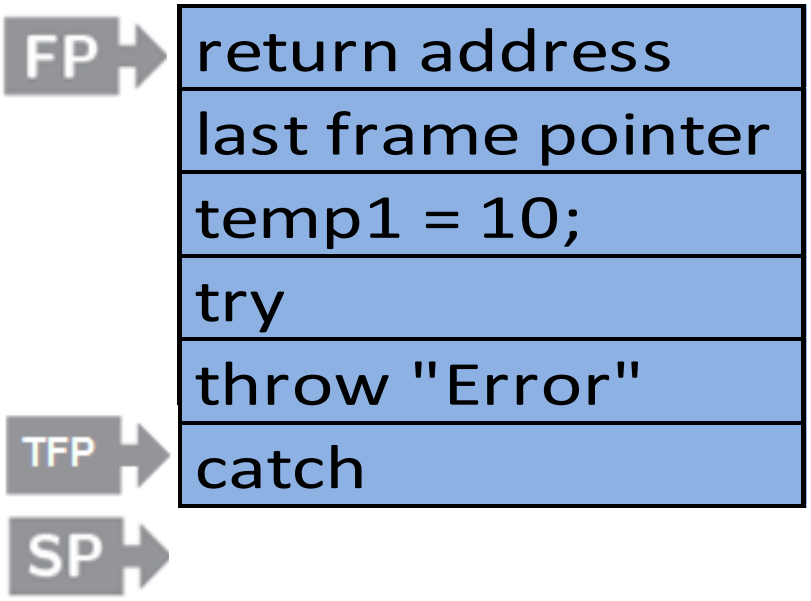

TradersJoe
Program.joe
Scanner
Parser
Ast
Interpreter
Byte Code
Compiler
Execution

Exception Handling Design

Designing the exception handler was an interesting part of this project. A try… throw… catch structure will require new Try Function Pointer (TFP). When the program sees a try block it will execute the block till it sees a throw statement or reaches the end of the try block. If there is a throw statement then the try block is terminated and the program jumps to the catch block. If there is no throw statement then it ignores any catch blocks.

To design this exception handling model the program has to remember the TFP to perform a jump. The diagram below shows the structure of a try…catch design.

Try Catch Pointer



Test Plan

White box testing with **code coverage** and **fault injection** was used for this development. After developing a piece of code all test cases were ran to make sure codes were not broken. This approach helped me in stopping the development and rolling back to the previous working code before it is too late to figure out the cause of the problem. This was true when working with byte codes and reference pointers. In addition **individual tests** were also conducted to perform targeted Quality Assurance (QA).

Following are the test cases for performing all tests.

No	Program File	Output File	Status
1	test-arith1.joe	test-arith1.out	
2	test-arith2.joe	test-arith2.out	
3	test-closeValue.joe	test-closeValue.out	
4	test-comments.joe	test-comments.out	
5	test-errorhandler.joe	test-errorhandler.out	
6	test-fib.joe	test-fib.out	
7	test-for1.joe	test-for1.out	
8	test-func1.joe	test-func1.out	
9	test-func2.joe	test-func2.out	
10	test-func3.joe	test-func3.out	
11	test-functioncall.joe	test-functioncall.out	
12	test-gcd.joe	test-gcd.out	
13	test-global1.joe	test-global1.out	
14	test-hello.joe	test-hello.out	
15	test-highValue.joe	test-highValue.out	
16	test-if1.joe	test-if1.out	
17	test-if2.joe	test-if2.out	
18	test-if3.joe	test-if3.out	
19	test-if4.joe	test-if4.out	
20	test-lowValue.joe	test-lowValue.out	
21	test-minValue.joe	test-minValue.out	
22	test-openValue.joe	test-openValue.out	
23	test-ops1.joe	test-ops1.out	
24	test-throwErrAfter.joe	test-throwErrAfter.out	
25	test-throwErrBefore.joe	test-throwErrBefore.out	
26	test-tryblock.joe	test-tryblock.out	
27	test-var1.joe	test-var1.out	
28	test-while1.joe	test-while1.out	

Command: sh testall.sh

```
$ sh testall.sh
testall.sh: line 6: ulimit: cpu time: cannot modify limit: Invalid argument
test-arith1...OK
test-arith2...OK
test-closeValue...OK
test-comments...OK
test-errorhandler...OK
test-fib...OK
test-for1...OK
test-func1...OK
test-func2...OK
test-func3...OK
test-functioncall...OK
test-gcd...OK
test-global1...OK
test-hello...OK
test-highValue...OK
test-if1...OK
test-if2...OK
test-if3...OK
test-if4...OK
test-lowValue...OK
test-minValue...OK
test-openValue...OK
test-ops1...OK
test-throwErrAfter...OK
test-throwErrBefore...OK
test-tryblock...OK
test-var1...OK
test-while1...OK
```

Testing throw error

Test File: test-throwErrAfter.joe

main()
begin

```
    try
    begin
        ThrowErr();
    end
end

ThrowErr()
begin
        throw(1);
        print(42); [* Should not reach here *]
end
```

Output File: test-throwErrAfter.out
Value: Empty file

Testing function call

Test File: test-functioncall.joe

```
main()
begin
```

```
    min minA;
    minA= 100;

    PrintValue(minA);
end

PrintValue(minValue)
begin
    print(minValue);
end
```

Output File: test-functioncall.out
Value: 100

Testing while loop

Test File: test-while1.joe

```
main()
begin
    int i;
```

```
    i = 5;

    while (i > 0)
    begin
        print(i);
        i = i - 1;
    end
    print(42);
end
```

Output File: test-while1.out
Value:
5
4
3
2
1
42

Lessons Learned

1. Debugging the code is a life saver. Although there is an OCamldebug, I was not able to use this for this program. Instead I used print_endline to get a handle of the program.

2. Thank God for the microC samples. It gives you the depth you need for creating a compiler from start to finish.

3. The programmer spirit is important. Be systematic, patient and understand the code.

4. Always run a full test suite before going to bed. If you ignore broken codes then it will haunt you in future.

5. Have a working code in hand early on. This will help you rapidly develop the program.

6. Listen to the Professors advice on the project. I realized that some of my proposal did not keep up with the spirit of this language and had to abandon it.

7. If you are working in a group then:
    a. Assign a project manager
    b. Select a project management tool that has collaboration and source control.

Appendix

Scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)

| "[*"     { comment lexbuf }           (* Comments *)

| '('      { LEFT_PAREN }
```

```
| ')'      { RIGHT_PAREN }

| "begin"  { BEGIN_STATEMENT }

| "end"    { END_STATEMENT }

| ';'      { SEMICOLON }

| '['    { LEFT_SQ_BRACKET }
| ']'    { RIGHT_SQ_BRACKET }
| '^'        { MATRIX_POWER }
| '.'    { DECIMAL_POINT }
| ','      { COMMA }

| '+'      { PLUS }

| '-'      { MINUS }

| '*'      { TIMES }
```

```
| '/'      { DIVIDE }

| '='      { ASSIGNMENT }

| "=="     { EQUAL_TO }

| "!="     { NOT_EQUAL_TO }

| '<'      { LESS_THAN }

| "<="     { LESS_THAN_EQUAL_TO }

| ">"      { GREATER_THAN }

| ">="     { GREATER_THAN_EQUAL_TO }
| "var"    { VARIABLE_ASSIGNMENT }

| "if"     { IF_CONDITION }

| "else"   { ELSE_CONDITION }
```

```
| "for"    { FOR }

| "while"  { WHILE }

| "return" { RETURN_STATEMENT }

| "begin"  { BEGIN_LOOP }
| "end"    { END_LOOP }
| "try"    { ERROR_TRY }
| "catch"  { ERROR_CATCH }
| "throw"  { ERROR_THROW }
| "errno"  { ERROR_NUM }
| "int"    { INT }

| "float"  { FLOATLIT }
| ['0'-'9']+ ['.'] ['0'-'9']* ('e' ['-' '+']? ['0'-'9']+)? as lxm { FLOAT
(float_of_string lxm) }
| "min"    { MINIMUM }
| "max"    { MAXIMUM }
| "median" { MEDIAN_PRICE }
| "ema"    { EMA_AVERAGE }
```

```
| "sma"    { SMA_AVERAGE }
| "open"   { OPEN_PRICE }
| "close"  { CLOSE_PRICE }
| "high"   { HIGH_PRICE }
| "low"    { LOW_PRICE }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }

| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

| eof { EOF }

| _ as char { raise (Failure("Error in command. Please refer TradersJoe
language reference manual " ^ Char.escaped char)) }

 and comment = parse
  "*]" { token lexbuf }

| _    { comment lexbuf }
```

Parser.mly

```
%{ open Ast %}

%token SEMICOLON LEFT_PAREN RIGHT_PAREN BEGIN_STATEMENT
END_STATEMENT LEFT_SQ_BRACKET RIGHT_SQ_BRACKET COMMA

%token MATRIX_POWER DECIMAL_POINT
%token PLUS MINUS TIMES DIVIDE ASSIGNMENT IS
%token EQUAL_TO NOT_EQUAL_TO LESS_THAN LESS_THAN_EQUAL_TO
GREATER_THAN GREATER_THAN_EQUAL_TO

%token RETURN_STATEMENT IF_CONDITION ELSE_CONDITION FOR WHILE
INT
 FLOATLIT VARIABLE_ASSIGNMENT
%token <int> LITERAL

%token <string> ID
%token <float> FLOAT
%token MINIMUM MAXIMUM MEDIAN_PRICE
```

```
%token EMA_AVERAGE SMA_AVERAGE OPEN_PRICE CLOSE_PRICE
HIGH_PRICE LOW_PRICE
%token ERROR_TRY ERROR_CATCH ERROR_THROW ERROR_NUM
%token BEGIN_LOOP END_LOOP
%token EOF


%nonassoc NOELSE

%nonassoc NOTRY
%nonassoc ELSE_CONDITION


%right ASSIGNMENT

%left EQUAL_TO NOT_EQUAL_TO

%left LESS_THAN GREATER_THAN LESS_THAN_EQUAL_TO
GREATER_THAN_EQUAL_TO

%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:
   ID LEFT_PAREN formals_opt RIGHT_PAREN BEGIN_STATEMENT vdecl_list
stmt_list END_STATEMENT
     { { fname = $1;
       formals = $3;
       locals = List.rev $6;
       body = List.rev $7 } }



formals_opt:
    /* nothing */ { [] }

| formal_list   { List.rev $1 }



formal_list:
```

```
  ID                { [$1] }
| formal_list COMMA ID { $3 :: $1 }


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


vdecl:
        INT ID SEMICOLON { $2 }
      | FLOATLIT ID SEMICOLON { $2 }
```

```
    | MINIMUM ID SEMICOLON { $2 }
    | MAXIMUM ID SEMICOLON { $2 }
    | MEDIAN_PRICE ID SEMICOLON { $2 }
    | EMA_AVERAGE ID SEMICOLON { $2 }
    | SMA_AVERAGE ID SEMICOLON { $2 }
    | OPEN_PRICE ID SEMICOLON { $2 }
    | CLOSE_PRICE ID SEMICOLON { $2 }
    | HIGH_PRICE ID SEMICOLON { $2 }
    | LOW_PRICE ID SEMICOLON { $2 }
    | ERROR_NUM ID SEMICOLON { $2 }
```

```
stmt_list:
   /* nothing */  { [] }

| stmt_list stmt { $2 :: $1 }
```

```
stmt:

  expr SEMICOLON { Expr($1) }

| RETURN_STATEMENT expr SEMICOLON { Return($2) }

| BEGIN_STATEMENT stmt_list END_STATEMENT { Block(List.rev $2) }

| IF_CONDITION LEFT_PAREN expr RIGHT_PAREN stmt %prec NOELSE { If($3,
$5, Block([])) }

| IF_CONDITION LEFT_PAREN expr RIGHT_PAREN stmt ELSE_CONDITION stmt
{ If($3, $5, $7) }

| FOR LEFT_PAREN expr_opt SEMICOLON expr_opt SEMICOLON expr_opt
RIGHT_PAREN stmt
    { For($3, $5, $7, $9) }
```

```
| WHILE LEFT_PAREN expr RIGHT_PAREN stmt { While($3, $5) }


| ERROR_TRY BEGIN_STATEMENT stmt_list END_STATEMENT { Block(List.rev
$3) }

| ERROR_CATCH BEGIN_STATEMENT stmt_list END_STATEMENT {
Block(List.rev $3) }

| ERROR_THROW expr SEMICOLON { Throw($2) }


expr_opt:
   /* nothing */ { Noexpr }

| expr        { $1 }



expr:
```

```
  LITERAL                 { Literal($1) }
| ID                      { Id($1) }
| 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 EQUAL_TO           expr    { Binop($1, Equal, $3) }
| expr NOT_EQUAL_TO           expr    { Binop($1, Neq,   $3) }
| expr LESS_THAN          expr    { Binop($1, Less,  $3) }
| expr LESS_THAN_EQUAL_TO      expr    { Binop($1, Leq,   $3) }
| expr GREATER_THAN     expr    { Binop($1, Greater,  $3) }
```

```
| expr GREATER_THAN_EQUAL_TO    expr    { Binop($1, Geq,   $3) }

| ID ASSIGNMENT           expr    { Assign($1, $3) }

| ID LEFT_PAREN actuals_opt RIGHT_PAREN { Call($1, $3) }

| LEFT_PAREN               expr RIGHT_PAREN { $2 }
```

```
actuals_opt:
   /* nothing */ { [] }

| actuals_list  { List.rev $1 }


actuals_list:
   expr { [$1] }
```

```
| actuals_list COMMA expr { $3 :: $1 }
```

Ast.mli

```
type ope = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq


type expr =
    Literal of int

| Id of string

| Binop of expr * ope * expr

| Assign of string * expr

| Call of string * expr list

| Noexpr
```

```
type stmt =
    Block of stmt list

| Expr of expr

| Return of expr

| If of expr * stmt * stmt

| For of expr * expr * expr * stmt

| While of expr * stmt


| Try of stmt list
| Catch of stmt list
| Throw of expr


type func_decl = {
```

```
    fname : string;

    formals : string list;

    locals : string list;

    body : stmt list;
  }


(* Exception declarations *)
type func_excdecl = {

    excfname : string;

    excformals : string list;

    exclocals : string list;
```

```
    excbody : stmt list;
  }


type program = string list * func_decl list

type excprogram = string list * func_excdecl list


let rec string_of_expr = function
  Literal(l) -> string_of_int l
| Id(s) -> s
| Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^
```

```
    (match o with

          Add -> "+"
        | Sub -> "-"
        | Mult -> "*"
        | Div -> "/"

        | Equal -> "=="
        | Neq -> "!="

        | Less -> "<"
        | Leq -> "<="
        | Greater -> ">"
        | Geq -> ">=")
     ^ " " ^
    string_of_expr e2

| Assign(v, e) -> v ^ " = " ^ string_of_expr e

| Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
```

```
| 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(expr) -> "return " ^ string_of_expr expr ^ ";\n";

| 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
```

```
| For(e1, e2, e3, s) ->
     "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
     string_of_expr e3  ^ ") " ^ string_of_stmt s

| While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s


| Throw(expr) -> "throw " ^ string_of_expr expr ^ ";\n";

| Try(stmts) ->
 "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"

| Catch(stmts) ->
 "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"


let string_of_vdecl
     id = "int " ^ id ^ ";\n"
```

```
let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"



let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

ByteCode.ml

```
type bstmt =
      Lit of int    (* Push a literal *)
    | Drp           (* Discard a value *)
    | Bin of Ast.ope (* Perform arithmetic on top of stack *)
    | Lod of int    (* Fetch global variable *)
    | Str of int    (* Store global variable *)
    | Lfp of int    (* Load frame pointer relative *)
```

```
| Sfp of int    (* Store frame pointer relative *)

| Jsr of int    (* Call function by absolute address *)

| Ent of int    (* Push FP, FP -> SP, SP += i *)

| Rts of int    (* Restore FP, SP, consume formals, push result *)

| Beq of int    (* Branch relative if top-of-stack is zero *)

| Bne of int    (* Branch relative if top-of-stack is non-zero *)

| Bra of int    (* Branch relative *)

| Fld of int    (* Fetch global variable *)

| Flt of int    (* Load frame pointer relative *)

(* try/ catch block *)
| Ltp of int    (* Load frame pointer relative for try/catch block *)
```

```
    | Stp of int    (* Store frame pointer relative for try/catch block *)

    | Jtr of int    (* Call function by absolute address for try/catch block *)

    | Ett of int    (* Push FP, FP -> SP, SP += i for try/catch block *)

    | Rtb of int    (* Restore FP, SP, consume formals, push result for try/catch
block *)

    | Hlt          (* Terminate *)



(* Frame Pointer, Stack Pointer, Program Counter *)
type prog = {

    num_globals : int;   (* Number of global variables *)

    text : bstmt array; (* Code for all the functions *)
```

```
}


let string_of_stmt = function
        Lit(i) -> "Lit " ^ string_of_int i
      | Drp -> "Drp"
      | Bin(Ast.Add) -> "Add"
      | Bin(Ast.Sub) -> "Sub"
      | Bin(Ast.Mult) -> "Mul"
      | Bin(Ast.Div) -> "Div"
      | Bin(Ast.Equal) -> "Eql"
      | Bin(Ast.Neq) -> "Neq"
```

```
| Bin(Ast.Less) -> "Lt"

| Bin(Ast.Leq) -> "Leq"

| Bin(Ast.Greater) -> "Gt"

| Bin(Ast.Geq) -> "Geq"

| Lod(i) -> "Lod " ^ string_of_int i

| Str(i) -> "Str " ^ string_of_int i

| Lfp(i) -> "Lfp " ^ string_of_int i

| Sfp(i) -> "Sfp " ^ string_of_int i

| Jsr(i) -> "Jsr " ^ string_of_int i

| Ent(i) -> "Ent " ^ string_of_int i
```

```
| Rts(i) -> "Rts " ^ string_of_int i

| Bne(i) -> "Bne " ^ string_of_int i

| Beq(i) -> "Beq " ^ string_of_int i

| Bra(i) -> "Bra " ^ string_of_int i

| Fld(i) -> "Fld " ^ string_of_int i

| Flt(i) -> "Flt " ^ string_of_int i

(* try/ catch block *)
| Ltp(i) -> "Ltp " ^ string_of_int i
| Stp(i) -> "Stp " ^ string_of_int i
| Jtr(i) -> "Jtr " ^ string_of_int i
| Ett(i) -> "Ett " ^ string_of_int i
| Rtb(i) -> "Rtb " ^ string_of_int i
| Hlt    -> "Hlt"
```

```
let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^


let funca = Array.mapi

          (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text

      in String.concat "\n" (Array.to_list funca)
```

Compile.ml

```
open Ast

open Bytecode



module StringMap = Map.Make(String)
```

```
(* Symbol table: Information about all the names in scope *)

type env =
{
    function_index : int StringMap.t; (* Index for each function *)

    global_index   : int StringMap.t; (* "Address" for global variables *)

    local_index    : int StringMap.t; (* FP offset for args, locals *)

    trycatch_index : int StringMap.t; (* Index for try/catch block *)
```

```
}



(* val enum : int int -> 'a list -> (int * 'a) list *)

let rec enum stride n = function

      [] -> []

    | hd::tl -> (n, hd) :: enum stride (n+stride) tl
```

```
(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)

let string_map_pairs map pairs =

    List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs



(** Translate a program in AST form into a bytecode program.  Throw an

    exception if something is wrong, e.g., a reference to an unknown
```

```
      variable or function *)

let translate (globals, functions) =



(* Allocate "addresses" for each global variable *)

let global_indexes =
      string_map_pairs StringMap.empty (enum 1 0 globals) in
```

```
(* Assign indexes to function names; built-in "print" is special *)

let built_in_functions =
    StringMap.add "print" (-1) StringMap.empty in


let function_indexes =
    string_map_pairs built_in_functions
 (enum 1 1 (List.map (fun f -> f.fname) functions)) in


let trycatch_indexes =
    string_map_pairs function_indexes (enum 1 2 (List.map (fun f -> f.fname)
functions)) in
```

```
(* Translate a function in AST form into a list of bytecode statements *)

let translate env fdecl =

(* Bookkeeping: FP offsets for locals and arguments *)

    let num_formals = List.length fdecl.formals

    and num_locals = List.length fdecl.locals

    and local_offsets = enum 1 1 fdecl.locals
```

```
and formal_offsets = enum (-1) (-2) fdecl.formals in

let env = { env with local_index = string_map_pairs

    StringMap.empty (local_offsets @ formal_offsets) } (* ERR *) in


let rec expr = function

    Literal i -> [Lit i]

  | Id s ->
```

```
        (try [Lfp (StringMap.find s env.local_index)]

         with Not_found -> try
              [Lod (StringMap.find s env.global_index)]

        with Not_found ->
              raise (Failure ("undeclared variable " ^ s)))

| Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]

| Assign (s, e) -> expr e @

        (try [Sfp (StringMap.find s env.local_index)]
```

```
        with Not_found -> try
            [Str (StringMap.find s env.global_index)]

        with Not_found ->
            raise (Failure ("undeclared variable " ^ s)))

    | Call (fname, actuals) -> (try

            (List.concat (List.map expr (List.rev actuals))) @

            [Jsr (StringMap.find fname env.function_index) ]

        with Not_found ->
            raise (Failure ("undefined function " ^ fname)))
```

```
      | Noexpr -> []


in
let rec stmt = function

       Block sl     -> List.concat (List.map stmt sl)

      | Expr e       -> expr e @ [Drp]

      | Return e     -> expr e @ [Rts num_formals]

      | If (p, t, f) -> let t' = stmt t and f' = stmt f in
```

```
        expr p @ [Beq(2 + List.length t')] @
        t' @ [Bra(1 + List.length f')] @ f'
    | For (e1, e2, e3, b) ->
        stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
    | While (e, b) ->
        let b' = stmt b and e' = expr e in
        [Bra (1+ List.length b')] @ b' @ e' @
```

```
            [Bne (-(List.length b' + List.length e'))]


        | Try sl     ->  List.concat (List.map stmt sl)

        | Throw e     -> expr e @ [Rts num_formals]

        | Catch sl   ->
                (* List.iter print_endline; *)
                (try (List.concat (List.map stmt sl))
@
                    [Lfp (StringMap.find "catch" env.trycatch_index)]
```

```
                with Not_found -> try
                    [Lod (StringMap.find "catch" env.global_index)]

                with Not_found ->
                    (* print_endline "Debug: Catch Not found"; *)
                    raise (Failure ("undefined catch block")))

in [Ent num_locals] @      (* Entry: allocate space for locals *)

stmt (Block fdecl.body) @  (* Body *)

[Lit 0; Rts num_formals]   (* Default = return 0 *)
```

```
in let env = { function_index = function_indexes;

      global_index = global_indexes;

      local_index = StringMap.empty;
      trycatch_index = trycatch_indexes } in



(* Code executed to start the program: Jsr main; halt *)

let entry_function = try

            [Jsr (StringMap.find "main" function_indexes); Hlt]
```

```
        with Not_found -> raise (Failure ("no \"main\" function"))

in



(* Compile the functions *)

let func_bodies = entry_function ::
        List.map (translate env) functions in
```

```
(* Calculate function entry points by adding their lengths *)

let (fun_offset_list, _) = List.fold_left

    (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0)
    func_bodies in

let func_offset = Array.of_list (List.rev fun_offset_list) in



{ num_globals = List.length globals;

    (* Concatenate the compiled functions and replace the function
```

```
indexes in Jsr statements with PC values *)

    text = Array.of_list (List.map (function

        Jsr i when i > 0 -> Jsr func_offset.(i)

      | _ as s -> s) (List.concat func_bodies))

}
```

Execute.ml

```
open Ast

open Bytecode



(* Stack layout just after "Ent":

          <-- SP
  Local n
  ...

    Local 0
```

```
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)


let execute_prog prog =

     let stack = Array.make 1024 0

     and globals = Array.make prog.num_globals 0
     and trycatchstack = Array.make 1024 0 in



let rec exec fp sp pc = match prog.text.(pc) with

      Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)

     | Drp    -> exec fp (sp-1) (pc+1)
```

```
| Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
    stack.(sp-2) <- (let boolean i = if i then 1 else 0 in
    match op with
      Add    -> op1 + op2
    | Sub    -> op1 - op2
    | Mult   -> op1 * op2
    | Div    -> op1 / op2
    | Equal  -> boolean (op1 =  op2)
    | Neq    -> boolean (op1 != op2)
    | Less   -> boolean (op1 <  op2)
```

```
        | Leq    -> boolean (op1 <= op2)
        | Greater -> boolean (op1 >  op2)
        | Geq    -> boolean (op1 >= op2)) ;

         exec fp (sp-1) (pc+1)
        | Lod i  -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
        | Str i  -> globals.(i)  <- stack.(sp-1) ; exec fp sp    (pc+1)
        | Lfp i  -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
        | Sfp i  -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp    (pc+1)
        | Jsr(-1) -> print_endline (string_of_int stack.(sp-1)) ; exec fp sp
(pc+1)
        | Jsr i  -> stack.(sp)   <- pc + 1       ; exec fp (sp+1) i
```

```
        | Ent i   -> stack.(sp)   <- fp          ; exec sp (sp+i+1) (pc+1)

        | Rts i   -> let new_fp = stack.(fp) and new_pc = stack.(fp-1) in

            stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc

        | Beq i   -> exec fp (sp-1) (pc + if stack.(sp-1) =  0 then i else 1)

        | Bne i   -> exec fp (sp-1) (pc + if stack.(sp-1) != 0 then i else 1)

        | Bra i   -> exec fp sp (pc+i)

        | Fld j   -> stack.(sp)   <- globals.(j)  ; exec fp (sp+1) (pc+1)

        | Flt j   -> exec fp sp (pc+j)

        (* try/ catch block *)
        | Ltp i   -> trycatchstack.(sp)   <- trycatchstack.(fp+i) ; exec fp
(sp+1) (pc+1)
```

```
        | Stp i   -> trycatchstack.(fp+i) <- trycatchstack.(sp-1) ; exec fp sp
(pc+1)

        | Jtr(-2) -> print_endline (string_of_int stack.(sp-1)) ; exec fp sp
(pc+1)

        | Jtr i   -> trycatchstack.(sp)   <- pc + 1      ; exec fp (sp+1) i

        | Ett i   -> trycatchstack.(sp)   <- fp          ; exec sp (sp+i+1) (pc+1)

        | Rtb i   -> let new_fp = trycatchstack.(fp) and new_pc =
trycatchstack.(fp-1) in

            trycatchstack.(fp-i-1) <- trycatchstack.(sp-1) ; exec new_fp (fp-i)
new_pc

        | Hlt     -> ()



        in exec 0 0 0
```

Interpret.ml

```
open Ast


module NameMap = Map.Make(struct
      type t = string
      let compare x y = Pervasives.compare x y
end)

exception ReturnException of int * int NameMap.t
```

```
(* Main entry point: run a program *)

let run (vars, funcs) =

(* Put function declarations in a symbol table *)

let func_decls = List.fold_left

    (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)

    NameMap.empty funcs

in


(* Invoke a function and return an updated global symbol table *)

let rec call fdecl actuals globals =
```

```
(* Evaluate an expression and return (value, updated environment) *)

let rec eval env = function

    Literal(i) -> i, env

    | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)

    | Id(var) ->

        let locals, globals = env in

        if NameMap.mem var locals then

            (NameMap.find var locals), env

        else if NameMap.mem var globals then

            (NameMap.find var globals), env
```

```
        else raise (Failure ("undeclared identifier " ^ var))
    | Binop(e1, op, e2) ->
        let v1, env = eval env e1 in
        let v2, env = eval env e2 in
        let boolean i = if i then 1 else 0 in
        (match op with
              Add -> v1 + v2
            | Sub -> v1 - v2
            | Mult -> v1 * v2
            | Div -> v1 / v2
            | Equal -> boolean (v1 = v2)
```

```
| Neq -> boolean (v1 != v2)

| Less -> boolean (v1 < v2)

| Leq -> boolean (v1 <= v2)

| Greater -> boolean (v1 > v2)

| Geq -> boolean (v1 >= v2)), env

| Assign(var, e) ->

    let v, (locals, globals) = eval env e in

    if NameMap.mem var locals then

        v, (NameMap.add var v locals, globals)

    else if NameMap.mem var globals then
```

```
                v, (locals, NameMap.add var v globals)
            else raise (Failure ("undeclared identifier " ^ var))
        | Call("print", [e]) ->
            let v, env = eval env e in
            print_endline (string_of_int v);
            0, env
        | Call(f, actuals) ->
            let fdecl =
                try NameMap.find f func_decls
                with Not_found -> raise (Failure ("undefined function " ^
f))
```

```
in
let actuals, env = List.fold_left
    (fun (actuals, env) actual ->
        let v, env = eval env actual in v :: actuals, env)
    ([], env) (List.rev actuals)
in
let (locals, globals) = env in
try
    let globals = call fdecl actuals globals
    in 0, (locals, globals)
with ReturnException(v, globals) -> v, (locals, globals)
```

```
    in


    (* Execute a statement and return an updated environment *)
    let rec exec env = function
        Block(stmts) -> List.fold_left exec env stmts
        | Expr(e) -> let _, env = eval env e in env
        | If(e, s1, s2) ->
            let v, env = eval env e in
exec env (if v != 0 then s1 else s2)
        | While(e, s) ->
            let rec loop env =
```

```
            let v, env = eval env e in
            if v != 0 then loop (exec env s) else env
        in loop env
    | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
            let v, env = eval env e2 in
            if v != 0 then
                let _, env = eval (exec env s) e3 in
                loop env
```

```
                    else

                        env

                in loop env

            | Try(stmts
) -> List.fold_left exec env stmts

            | Throw(e) ->
    let v, (locals, globals) = eval env e in
     (* Jump to catch block *)
                raise (ReturnException(v, globals))

            | Catch(stmts
) -> List.fold_left exec env stmts

            | Return(e) ->
    let v, (locals, globals) = eval env e in

                raise (ReturnException(v, globals))
```

```
in
    (* Enter the function: bind actual values to formal arguments *)
let locals =
      try List.fold_left2
          (fun locals formal actual -> NameMap.add formal actual locals)
          NameMap.empty fdecl.formals actuals
      with Invalid_argument(_) ->
          raise (Failure ("wrong number of arguments passed to " ^
fdecl.fname))
in
    (* Initialize local variables to 0 *)
```

```
let locals = List.fold_left

      (fun locals local -> NameMap.add local 0 locals)
      locals fdecl.locals

in
   (* Execute each statement in sequence, return updated global symbol table
*)

snd (List.fold_left exec (locals, globals) fdecl.body)



(* Run a program: initialize global variables to 0, find and run "main" *)

in let globals = List.fold_left

      (fun globals vdecl -> NameMap.add vdecl 0 globals) NameMap.empty vars

in try
```

```
        call (NameMap.find "main" func_decls) [] globals

with Not_found ->
        raise (Failure ("did not find the main() function"))
```

TradersJoe.ml

```
type action = Ast | Interpret | Bytecode | Compile


let _ =

        let action = if Array.length Sys.argv > 1 then

                List.assoc Sys.argv.(1)
                        [ ("-a", Ast);

                        ("-i", Interpret);
```

```
            ("-b", Bytecode);

            ("-c", Compile) ]

else Compile in

let lexbuf = Lexing.from_channel stdin in

let program = Parser.program Scanner.token lexbuf in

match action with

      Ast -> let listing = Ast.string_of_program program

                  in print_string listing

            | Interpret -> ignore (Interpret.run program)
```

```
| Bytecode -> let listing =

        Bytecode.string_of_prog
        (Compile.translate program)

    in print_endline listing

| Compile -> Execute.execute_prog (Compile.translate program)
```

Bibliography

- Book: Compilers Principles, Techniques & Tools – Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman
- Web: OCaml: http://www.cs.jhu.edu/~scott/pl/lectures/caml-intro.html
- Course Materials: Micro C
- Video: http://www.youtube.com/watch?v=1XZoKXJpbVg