Clogic – A Logic Programming Language

Tabari Alexander
tha2105

COMS W4115 – PLT Summer, 2011

Table of Contents

1 Introduction

Clogic is a language designed to build and infer relationships between objects and allow a developer to easily handle situations where these relationships change. Based on concepts from logic programming languages, developers can establish facts about abstract objects, and use knowledge based on those truths in order to control their program. In addition, Clogic also allows a small set of imperative programming features for a bit of extra flexibility.

1.1 Background

Most of the logic programming concepts in Clogic are derived from Prolog, which is a declarative logic programming language. With Prolog's base datatype, an atom, a developer can create a set of facts and rules. Facts are a statement of truth about an atom, which itself is just a meaningless named abstraction. For example, the statement (not in Prolog syntax) "Jim is tall" declares a truth about the atom, Jim; Jim is tall. A rule is a statement that infers a truth from facts. An example of this would be "Football linebackers are people that are tall and heavy". If another truth were to be added, "Jim is heavy", then it could be inferred from the rule that Jim is a football linebacker.

Ultimately, the computation that is involved in Prolog is derived from a query, and the resulting answer is either a yes or no response, or a set of atoms that would make the query true. Given the rest of the already stated examples, a query in this context would be "Who is a football linebacker?", in which case the returned answer would be Jim. Additionally, if the query was "Is Michael a football linebacker?", the answer would be no. Clogic attempts to retain these concepts while at the same time allow queries and information that is obtained from the queries to be used in an imperative fashion.

1.2 Goals

Simplicity

Clogic is designed with simplicity in mind. In order to make this possible, most declarations are made implicit and any variables that are used are dynamically typed in order to make writing the code fairly easy. In addition, the language syntax is also designed to keep everything as straight-forward as possible – no messing around with include files, modules, or similar constructs.

Portable

To allow *Clogic* to be used on multiple platforms, it is designed to be compiled into bytecode to either be run via an interpreter similar to Java. The option also exists to build a standalone application, based on the bytecode.

2 Language Tutorial

In order to provide a brief overview of the language, the quicksort algorithm will be used to demonstrate majority of the language constructs. Line numbers are indicated by a comment at the beginning of the line, for the sake of clarity.

```
[[ tutorial.clg ]]
[[ 1]] append(H|T, Y, Ans) -> append(T, Y, *X) & (Ans = H | X);
[[ 2]] append([], Y, Ans) -> Ans = Y;
```

```
[[ 3]] qsort(H|T, Ans) -> qsort(H|T, H, [], [], Ans);
[[ 4]] qsort(H|[], Ans) -> Ans = [H];
[[ 5]] qsort([], []);

[[ 6]] qsort(H|T,P,L,G,Ans) -> (H > P ? G = H | G : L = H | L) &
qsort(T,P,L,G,Ans);
[[ 7]] qsort([],P,L,G,Ans) -> qsort(L,*X1) & qsort(G,*X2) & append(X1,
X2, Ans);

[[ 8]] {
[[ 9]]     qsort([3,6,1,2,8,0,9,-1,4,7],*Ans);
[[10]]     print(Ans);
[[11]] }
```

2.1 Program Overview

Although the example may look daunting, it is actually performing some relatively trivial tasks. In a nutshell, this program defines three predicates, `append/3`, `qsort/2`, and `qsort/3`, in addition to actually performing the quick sort and printing out the answer. For those unfamiliar with the algorithm, quicksort picks a pivot point and splits the list into two different lists, one containing items greater than the pivot and the other containing items less than the pivot (items equal can be placed in either). This is recursively done on each list until quicksort is performed on a list of length 1 or 0, in which case the function will simply return. As the stack unwinds, the list of larger items is appended to the list of smaller items, resulting in a sorted list.

2.2 Declarations

One of the first things that needs to be done to perform a quicksort is to define a quicksort. This is done on lines 3-5. Lines 4 and 5 define the base cases for quicksort. Both are examples of predicate declarations, with the latter being a more explicit truth operation. The first two arguments of line 4 declares the patterns that the arguments will match against. The first argument matches against lists; more specifically this matches a list with only one element. The two variables (denoted by the initial capital letter) match against anything. If the arguments of a query match these arguments then the clause will be executed, which simply performs an assignment. Line 5 is the other base case, which states that a sorted empty list is simply the empty list. A predicate declaration in this form is also known as a truth – it does not require a clause.

Line 3 is the inductive case, where the list contains more than one item. In this particular instance, the head of the list is chosen as a pivot point, and an expanded form of qsort is queried. It is important that this inductive case comes before the base cases, since queries are matched against predicate declarations in reverse order of how they are defined. Thus if this were at line 5 instead, it is possible that the code can result in an infinite loop or errors.

The pivot sorting version of qsort is defined on lines 6 and 7. As with the short form, the base case is defined after the inductive case. On line 7 is the case where all elements in the list have been sorted according to the pivot value. The asterisks in the queries denote return variables – Clogic's way of returning information from a query. Return variables will match against any literal or variable in a predicate declaration. If it matches against a literal and the query succeeds, then the variable is is given the value of the literal. If it instead matches a variable, and the variable is used as the left-hand-side of assignment or as an argument of a query, it will be given the value of the assignment or query

argument.

The weird notation on line 6 denotes an if-then-else clause, which in this case performs the sorting of the head of the input list into the greater or lesser lists.

2.3 Program Flow

The actual execution of the program is defined on lines 8-11. The curly braces on line 8 and 11 denote that the statements within are to be executed in order once the program loads. In this case, it will first perform a quicksort on the list (a random set of numbers), and then print the list using one of the language's built-in predicates.

2.4 Program Execution

This program can be compiled using the syntax below:

```
cl tutorial.clg
```

This will create a bytecode object file (tutorial.clo) which can then be run in the Clogic interpreter, using the following syntax:

```
clvm tutorial.clo
```

Upon successful completion of the program, you should be treated with the following output:

```
[-1, 0, 1, 2, 3, 4, 6, 7, 8, 9]
```

3 Language Manual

3.1 Lexical conventions

There are six kinds of tokens in Clogic: identifiers, variables, keywords, literals, operators, and separators. Whitespace in general is ignored except when it is used to separate tokens. A whitespace is required to separate adjacent tokens, keywords, and literals.

Tokens are formed based on character sequences from the input document. Priority is given to tokens that produce the largest matching sequence.

3.1.1 Comments

The characters [[start a comment, which is ended by the sequence]]. Comments cannot be nested, meaning that for several sequences of [[, the first]] seen will terminate the comment. Comments serve as a way for annotating the code, and are to be ignored by the compiler.

3.1.2 Identifiers

An identifier is a lowercase letter followed by a sequence of zero or more alphanumeric characters or underscores. Identifiers can represent predicate names, truth names, or atoms. They are case sensitive, i.e. thisname is a different identifier than thisName. Furthermore, identifiers can be any length, but it is up to the implementation to decide the length of the internal representation. In the case

of identifiers, an underscore is considered a lowercase letter.

Identifiers that start with an underscore are reserved for use by the compiler, but is not an error to use them. Instead, their use and behavior is undefined.

3.1.3 Keywords

The identifiers below are reserved for use as keywords, and may not otherwise be used as an identifier:

```
while
until
```

3.1.4 Variables

Variables are represented by an uppercase letter followed by a sequence of zero or more alphanumeric characters or underscores. As with identifiers, variables are case-sensitive. There are no reserved variable names.

3.1.5 Literals

There are three types of literals: integers, strings, and empty lists.

Integers

An integer literal is a sequence of digits, represented as a decimal number. An integer cannot start with a 0, unless it is 0.

Strings

A string literal starts with a `"`, is followed by a sequence of zero or more ASCII characters, and terminated by an ending `"`. If a `"` is intended to be in the literal, it must be escaped with a backslash. Here are all the acceptable escapes:

\n newline
\" double quote
\t horizontal tab
\\ backslash

A backslash that is not followed by a valid escape character is considered a syntax error. A single character may be represented by a string containing one character.

Empty list

An empty list is denoted by the character sequence `[]`. This token corresponds to a list containing no elements.

3.2 Syntax notation

In the following sections of this manual, ***bold italics*** will be used to identify grammar productions and a `monospace font` will be used to identify literal words, characters and symbols. Different rules for the same production will be written on separate lines, and any optional production will be given the suffix ***-opt***.

3.3 Predicates

The main way to execute code in Clogic is via the use of predicates. Predicates consist of a predicate name, some number of pattern-matching arguments, and an optional clause. If a clause is not supplied for a predicate, then all of its arguments must be literals. This is known as a truth. Predicates and truths may be referred to by their name, followed by a / and the predicate's arity, or number of arguments. For example, the predicate:

```
name(X, Y) -> X == Y
```

can be referred to as `name/2`.

3.4 Variables

Variables in Clogic do not identify mutable storage. Instead, they refer to locations in which a value is stored. The expression X = X + 3 for example does not modify the original value of X, but creates a new location in memory where the result of X + 3 is held, and X now points to that location. Compilers may optionally modify storage rather than point to a new location, as long as it does not change the behavior of the program.

3.5 Types

There are 4 different base types in Clogic: integers, booleans, atoms and lists. Clogic is dynamically typed, hence there does not exist a way to explicitly declare a variable type. Instead, the type of the variable is the type of the object it is currently referencing.

Clogic is not explicitly strongly typed, as the arguments to queries and operators can be any type. Queries can be made to be strongly typed, however, via the use of built-in type checking which is defined in this section.

3.5.1 Integers

Integers can have a maximum value of 2,147,483,647 and a minimum value of -2,147,483,648. Any overflows that may occur as a result of an operation will silently wrap, so care must be taken when performing operations near the limits of the values. The predicate for testing for an integer is `integer/1`.

3.5.2 Booleans

Booleans cannot be explicitly declared via literals. Instead, boolean values are returned from queries. There exist two built-in predicates, `true/0` and `false/0` which always evaluate to true and false, respectively. The predicate for testing for a boolean is `boolean/1`.

3.5.3 Atoms

Atoms are named abstractions. Their purpose is for building relations between different objects and types within truths and predicates. They are denoted by identifiers, and can be of any length. The predicate for testing an atom is `atom/1`.

3.5.4 Lists

Lists are linked lists of objects. Each object within a list can be of any type, including other lists. The empty list, `[]` is assigned to any undeclared variables.

Strings are a special forms of a list, where every character is in the printable ASCII range. Both strings and lists can be tested for by the predicate `list/1`. Strings can be tested for by the predicate `string/1`.

3.6 Type conversions

As mentioned earlier, Clogic is a strongly typed language. Therefore, all type conversions must be explicit. There exist a handful of built-in predicates for handling type conversions, which may be overridden by additional predicate definitions. Unless otherwise mentioned, the following predicates defined below follow the format:

identifier (***expression , result***)

where expression is the item to convert, and result is the returned conversion. If the conversion fails or the conversion is not defined for the type, result is given the value the empty lists and the query fails.

3.6.1 `tointeger/2`

This predicate is defined for strings and integers. If the string is in the form as described for integer literals, the string will be converted into an integer and result will be assigned that value, assuming that it lies within the acceptable range of integers.

If the expression is of the integer type, then the integer is returned.

3.6.2 `true/1`

This predicate converts expressions to booleans, and works on all types. Its format is defined as:

identifier (***expression)***

All non-zero integers evaluate to true. All non-empty lists, including strings, evaluate to true. All atoms evaluate to true. Everything else evaluates to false.

3.6.3 `tostring/2`

The string conversion predicate exists for integers, booleans, atoms, and strings. If the expression is an integer, the result is a string that represents the integer. If the expression is a boolean, and it evaluates to true, then the result is “true”, otherwise “false”. If the expression is an atom, then the result is the string form of the atom name.

3.7 Expressions

The operators within this section are defined in order of precedence with respect to their major sections. All operators within the same subsection have the same order of precedence. A table is defined in the appendix for quick reference.

Most operators are shorthand for predicates, and thus can be extended with predicate definitions. For example, the assignment:

```
X = 3 + 2
```

is equivalent to the query:

```
add(3, 2, *X)
```

If the predicate name is not specified, it is to be assumed that there is not an equivalent predicate for the operation.

3.7.1 Primary expressions

identifier

An identifier is a primary expression. In this context, all identifiers are atoms.

literal

Literal integers and strings can be primary expressions. They are typed as integers and strings, respectively.

variable

Variables can be primary expressions, regardless of whether or not they have been used. All variables are dynamically typed. If a previously unused variable is referenced, its value will be the empty list.

[expression-list]

A list is a set of square brackets enclosing a comma-separated list of expressions. The type of the list is list. The types of each individual expression within the list has no bearing on the type of the list.

(expression)

A parenthesized expression is a primary expression whose value and type are identical to the expression contained within.

(expression ? expression : expression-opt)

The ?: operator indicates a if-then-else statement, where the the first expression is the expression to test, the second expression is the expression to evaluate if the first is true, and the third expression is an expression to evaluate if the first expression is false.

identifier (query-args-opt)

A query is an identifier followed by set of parentheses containing an optional argument list of comma-separated query arguments. These arguments can either be expressions or when denoted by a * followed by a variable name, a return variable. The arguments of a query are first evaluated left-to-right, and then passed to the query for evaluation. The result of this expression is of the boolean type.

If a query does not have any return variables in its argument list, it is evaluated according to the following algorithm:

i. The arguments will be matched against the arguments of truths for the predicate name and arity in reverse order of truth declaration. If a match is found, then true will be returned.
ii. The arguments will be matched to the arguments of predicate clauses for the given name and arity in reverse order of predicate declaration. If a match is found, then the clause will be evaluated. If the clause is evaluated to false, the next matching predicate will be used until either a clause has been evaluated to true or there are no more rules to evaluate. Any side-effects caused by failed predicate clause evaluations will still occur.

If a query contains at least one return variable in its argument list, it is evaluated according to the same algorithm, with these additional rules:

i. When the return variable is matched against a literal in a truth, the variable is given that value.
ii. When the return variable is matched against a literal in a predicate, the variable is given that value.
iii. When the return variable is matched against a variable in a predicate, the variable will act as an alias for the return variable. Thus, if an assignment occurs with the variable at the left-hand-side, the return variable will be given the value of the assignment.
iv. If the query evaluates to false, the empty list is stored as the return value.
v. For all other instances, the empty list is stored as the return value.

It is an error for two return variables that have the same name to appear in an a query argument list. Likewise, the behavior is undetermined if a return variable and a variable with the same name to appear in a query argument list.

3.7.2 Unary operators

All unary operators have predicates in the form:
op (***Operand*** , ***Result***)

- unary-expression

The result of a negation expression is dependent on the type. When used on Integers, the expression is negated. Otherwise, the operation is undefined. The predicate for this operation is `neg/2`

! unary-expression

The result of a logical negation operator is true if the expression is false, and false if the expression is true. The predicate for this operation is `not/2`.

3.7.3 Multiplicative operators

All multiplicative operators are grouped left-to-right. All multiplicative operators have predicates defined in the form:

op (***Operand1*** , ***Operand2*** , ***Result***)

multiplicative-expression * ***unary-expression***
multiplicative-expression / ***unary-expression***

The binary * and / operators indicates multiplication and division, respectively. The operation

is only defined between integer types, and the resulting type after the operation is an integer. On a division, only the integer part of the operation will be returned. If the divisor evaluates to zero, then an error message will be printed and the program will exit. The predicate for these operations are `mult/3` and `div/3`.

3.7.4 Additive operators

All of the following additive operators are grouped left-to-right. All additive operators have predicates defined in the form:

op (***Operand1*** , ***Operand2*** , ***Result***)

additive-expression + ***multiplicative-expression***
additive-expression + ***multiplicative-expression***

The binary + and - operators indicates addition and subtraction. The operation is defined for integers, and results in an integer. The predicates for these operations are `add/3` and `sub/3`.

3.7.5 *additive-expression* | *cons-expression*

The binary | operation indicates a list construction. This operation is only defined where the first operand is any type, the second operand is a list type, and the result is of the type list. The operator groups right-to-left, and is implemented using the predicate `cons/3`.

3.7.6 Relational operators

The relation operators group left-to-right, but since the default return type is a boolean, it does not mean much in the context of these operators. All relation operators have predicates defined in the form:

op (***operand1*** , ***operand2***)

The result is the evaluation of the query.

relational-expression > cons-expression
relational-expression < cons-expression
relational-expression >= cons-expression
relational-expression <= cons-expression
relational-expression == cons-expression
relational-expression != cons-expression

The binary >, and < operators indicates a greater-than and less-than relation. The operator is only defined between integer types. These operators followed by an = denote a greater-than or equal to relation and a less-than or equal to relation. The predicates for these operations are `gt/2`, `lt/2`, `ge/2`, and `le/2`.

The equality operator == indicates equality. The operator is defined between integer types, between atoms, between booleans, and between lists. Equality between integers means that the value of both integers are equal. Equality between atoms indicates that the atom names are the same. Equality between booleans indicates that both booleans are true. Equality between lists indicates that both lists are the same length and each element is equal. The predicate for this operation is `eq/2`.

The equality operator != indicates inequality. The predicate for this operation is `ne/2`.

3.7.7 *and-expression* `&` *relational-expression*

The & operator indicates a logical and operation, which evaluates to true if both expressions evaluate to true. If the left-hand-side evaluates to false, then the right-hand-side will not be evaluated. The operator is defined for all types, as both expressions are implicitly evaluated to boolean types using the conversion predicate `true/1`.

3.7.8 *or-expression* `||` *and-expression*

The || operator indicates a logical or operation, which evaluates to true if either expression evaluates to true. If the left-hand-side evaluates to true, then the right-hand-side will not be evaluated. The operator is defined for all types, as both expressions are implicitly evaluated to boolean types using the conversion predicate `true/1`.

3.7.9 *variable* = *or-expression*

Assignments group right-to-left, and evaluate to true. They evaluate an expression and bind the result to the variable.

3.8 Declarations

Declarations are made prior to the main execution block of the program. Strictly speaking, an identifier does not need to be declared before it is used. When an undeclared query is evaluated, it instead evaluates the query `false/0`.

Declarations have the form:

declaration:

predicate-clause-decl ;

truth-decl ;

3.8.1 Predicate clause declaration

Predicate clauses are of the form:

predicate-clause-decl:

identifier `(` ***pattern-list-opt*** `)` `->` ***expression***

The pattern-list is a comma-separated list of patterns that will be matched against arguments in a query. Patterns can be literals, variables, or list patterns. Multiple predicate clauses with the same predicate name are treated as if they were joined by `||` operators, with the lowest defined predicate clause given the highest precedence.

9.1.1. List pattern

A list pattern is a pattern followed by the cons operator `|`, followed by either the empty list literal [] or a variable. When a list argument is matched against a list pattern, the left-hand-side of the cons operator is matched against the head of the list argument, and the right-hand side of the operator is matched against the tail of the list argument, and so on and so forth. A right-hand side with the value `[]` matches an empty list. For example, the expression:

```
[3, 4, 5,12]
```

matches the list pattern:

```
3 | 4 | X
```

where `X` will be given the value `[5,12]`.

3.8.2 Truth declaration

A truth is declared similarly to predicate clauses, according to the following syntax:

truth-decl:
identifier (***literal-list***)

where a literal list is a comma-separated list of literals. A truth defines a concrete relationship between literals, which in turn which defines the set of literals that can be returned as a return variable in a query.

3.9 Statements

Statements are executed in sequence, except where otherwise indicated.

3.9.1 Expression statement

Most statements are expected to be expression statements, which are in the form:

expression-statement:
expression ;

More often than not, expression statements will either be assignments or queries. Other expressions do not have much of an effect, unless their operator has been extended.

3.9.2 Block

A block is a collection of optional statements, and can be used wherever statements are allowed. They are defined in the form:

block:
{ ***statements-opt*** }
statements-list:
statement
statements statement

3.9.3 Conditional statement

A conditional statement is similar to the `?:` operator except that it can apply to statements within the main execution block. It is defined as:

conditional-statement:
expression ? ***statement*** : ***statement***

Where the first statement is evaluated if the expression is true, otherwise the second statement is evaluated.

3.9.4 While statement

A while statement executes evaluates the predicate true/1 with the specified expression. If the result is true, then the statement is evaluated, and the test and execution repeats itself until the test evaluates to false. While statements are of the form:

while-statement:
`while` ***expression ; statement***

3.9.5 Until statement

An until statement is the same as a while statement with the exception that the statement is executed before the test is performed. Until statements follow the following syntax:

until-statement:
`until` ***expression ; statement***

3.10 Programs

The input to the Clogic compiler is a program. It consists of an optional set of declarations followed by the main execution block. It is this block that is executed from the resulting compilation object. Programs are represented by the following syntax:

program:
declaration-list-opt block

3.11 Scope

There are no globally-scoped variables in Clogic. Variables have scopes within the main execution block and within each declaration.

Predicates have a global scope.

4 Project Plan

In order to develop clogic during a summer session with a team of just one student, it was crucial to have the project planned out early in development. The focus of the plan was to get as much of the fundamental framework completed as quickly as possible, allowing ample time to develop the translator and interpreter.

4.1 Design and Development

It was fortunate that there were separate due dates for the white paper and the language reference manual, as they directly led to dates in which the framework was developed. The scanner helped with development of the white paper. With it, all of the examples used in the white paper were shown to be at least parse-able by the compiler. Likewise, the parser was developed before the language reference manual was finished, as it helped to ensure that all ambiguities were resolved.

When it came to actually implementing the translator, an Agile approach was used. Using this methodology, all of the remaining development items were split into several different phases, with each phase consisting of designing, test suite improvements, translator development, interpreter

development, and final regression.

To use a concrete example, the goal of the first phase was to compile and run basic programs involving integer types and basic arithmetic expressions. Once a design was created for how the expressions would be interpreted in bytecode, several testcases were developed, testing basic expressions and operator precedence. The translator was then updated to handle the integer and arithmetic cases, and new opcodes were added for the interpreter. Once development was finished, everything was tested, with a goal of having no bugs that would not be addressed by another phase.

This approach led to rapid development and addition of new features. Some of these phases were completed in hours, while some may have taken longer. However, none of the phases took longer than 3 *working* days to complete. Unfortunately, this method of rapid development and planning did have some drawbacks. First and foremost, towards the end there was a lot of revisiting of already completed features and heavy-duty refactoring due to additional requirements in different areas of the compiler. Notably the backtracking and operator overloading work suffered from this. Additionally, a bit of stub code needed to be added in each phase in order to account for the features that had not yet been implemented. Sometimes it was just a minor annoyance; other times it proved to be a pain since it affected the test output.

4.2 Project Plan

The following is the development log, which also contains the overall project plan. Not included are any bugs that were found during development and test.

```
5/29/11 - Idea formulated for language.  Proposal started.
5/30/11 - Prototype parser that parses the language examples started
and finished.
6/4/11 - Proposal finished.
6/21/11 - Reworked example 1 from proposal and started parser and
scanner modification to parse and accept
6/22/11 - Example 1 now parses and accepts.  Started work on example
2.
6/27/11 - Finished LRM
7/5/11 - Started phase 1 work (basic expressions on ints) on bytecode
translater and interpreter; implemented arithmetic expressions
7/7/11 - Implemented variable loading and equality operators.
7/8/11 - Started phase 2 work (simple truth declarations and queries);
implemented truth declarations and started work on frames
7/9/11 - Finished stack frame work and implemented queries for truths
and predicates, with a base implementation for predefines
7/10/11 - Completed phase 4 and 5
7/12/11 - Started phase 7 (skipping 6 temporarily)
7/17/11 - Continued with phase 7 and started phase 8
7/18/11 - Finished phase 7 (may need revisiting)
7/25/11 - Finished phase 8 and 9.  Started Phases 11 and 12
7/27/11 - Started Phase 13, since it turns out that it is required for
completion of phases 11 and 12
7/30/11 - Finished Phases 13 and 14.  Added test automation
7/31/11 - Finished phase 11.  Started work on phase 12.
8/2/11 - Finished phase 12

Plan
* Phase 1 - Ints, Basic arithmetic expressions (completed 7/7)
```

```
* Phase 2 - Variable loading, logic operations, Equality, Truth
declarations, and queries (completed 7/9)
* Phase 3 - Predicate declarations (completed 7/9)
* Phase 4 - Datatype support in the interpreter and Atoms (completed
7/10)
* Phase 5 - Lists, List operations, and Strings (completed 7/10)
* Phase 6 - Revisit decl reordering (done)
* Phase 7 - Backtracking (completed 7/18)
* Phase 8 - Finish expressions (completed 7/25)
* Phase 9 - Other types of statements (completed 7/25)
* Phase 10 - removed
* Phase 11 - Builtins (completed 7/31)
* Phase 12 - Operator overloading (8/2)
* Phase 13 - Compiler cleanup and refactoring (completed 7/30)
* Phase 14 - Compiler front-end (completed 7/30)
```

4.3 Development Environment

The development environment of choice was a simple text editor, GNU Make, a hex editor and Subversion. Although a hex editor was used, it was primarily used for viewing the contents of the resulting object files. The parser, scanner, and bytecode translator are written in OCaml, while the bytecode interpreter is written in C++. This was done in part due to being more familiar with C++ and also to demonstrate how the bytecode could be interpreted with any language.

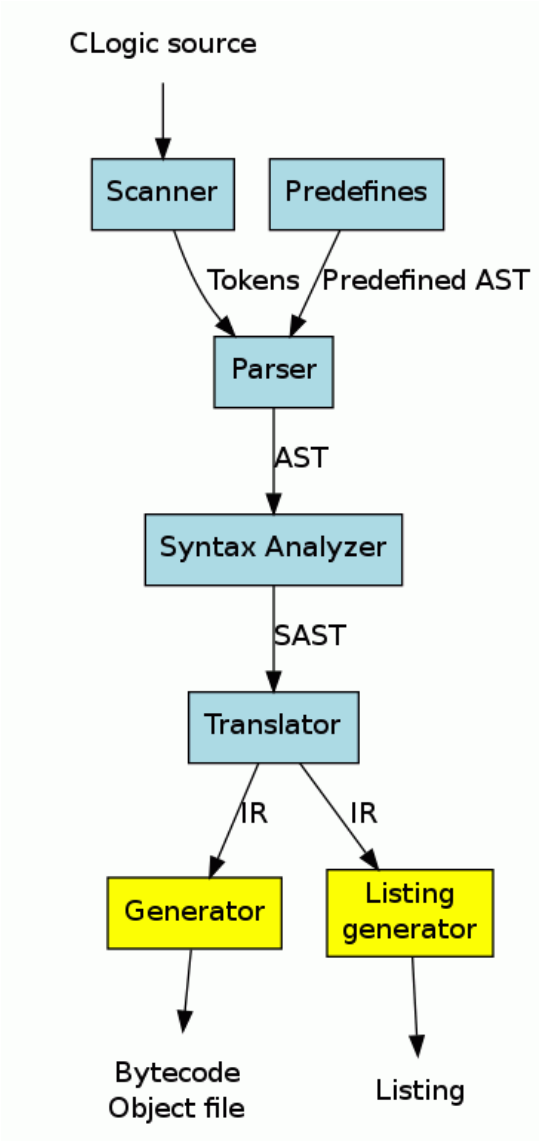
In order to have some semblance of readability, the following style was used as demonstrated by the following listings:

```
(** File description *)

(* Namespace statements *)
open Other

(* Type declarations *)
type t = Thing

(* Modules *)
module MyMap = Map.Make(Thing)

(* Declarations *)
let format_example a =
  let inner_decl_example = function
      First -> a
    | Not_first -> a - 1 in
  let last_inner = 42
  in
  inner_decl_example First + last_inner


int
func (int arg1)
{
     int rc = -1;
     switch (arg1) {
     case 1:
        if (var == 3)
```

```
            rc = 0;
        else
            rc = 12;
    }
    return rc;
}
```

5 Architectural Design

Even though a lot of the design took place while the code was developed, the overall architecture was decided upon relatively early. It follows the traditional approach of a front end and a back end, with the distinction denoted by the colors in the figure below:



Traces from the design process are evident in the architecture design, as some of the language elements, like the built-in and predefined predicates, are provided as separate “inputs” into the flow.

5.1 Scanner, Parser, and Predefined Predicates

The scanner and parser are the two oldest modules in the entire compiler. Largely implemented with ocamllex and ocamlyacc, these two modules are largely responsible for building the abstract syntax tree (AST). Early implementations created a tree largely filled with placeholder "Todo" nodes, which were coded to throw errors if they were encountered during a parse. This laid the groundwork for future work while at the same time helping to ensure that all paths were implemented.

The root of the AST that is produced by the parser consists of predicate declarations and a statement, with the statement being the main executable part of the program. Prior to any declarations from the source being added, predefined predicates are first added to the AST. This guarantees that all user defined predicates take precedence over the ones that are defined as part of the language. It was a bit of a struggle to figure out how to best incorporate the predefines with the final AST, and how to best represent them in code. Ultimately, it was decided to code the predefined predicates directly in their AST form, even though they could have been parsed by the front end. At the time, it proved too difficult to accurately parse both the predefined predicates and the input source.

5.2 Syntax Analyzer

The syntax analyzer is largely responsible for checking the nuances of the language and for massaging the AST into a form that is more recognizable by the translator. Some of the massaging that is done includes resolving query names into their internal representation, converting overloadable operations into sequences of queries, and converting groups of declarations with the same internal name into evaluable trees.

5.3 Translator and Code Generator

The translator takes the SAST as input and generates an intermediate representation (IR). It transforms the SAST into a list of instructions that operate on a stack. The translator is aided by recursive helper functions with build the IR into a list, starting from the end and working backwards. This is done to avoid the inefficiencies of list appends in OCaml. As such, the predicate declarations are translated before the main execution code.

Because of its design, the IR in large part has a one-to-one mapping with bytecode instructions that are generated by the code generator. The generator takes the IR and converts it into a file containing bytecode instructions. At this stage, the only major function that needs to take place is address resolution. All of the labels in the IR are converted to offsets from the start of the bytecode, and as the bytecode is written to file, the labels are resolved to these offsets, resulting in the final bytecode object.

Added relatively late, the listing generator takes the IR and pretty-prints it for debugging purposes. It was particularly useful for visualizing the finished product without resorting to examining the binary text file. It also exposed some oddities with the translation. Namely, labels to the same effective code segment merely stacked, even though they referred the the same location. This did not have any effect on the code itself, but specifically crafted source files could generate tons of label statements one after another.

6 Test Plan

Reiterating what was discussed in the project plan, the test approach was to rigorously test new features and add those tests to a regression bucket once the tests pass. However, this did not mean that

the tests were never touched again after initially developed. Test cases were routinely updated to account for the stubbing that was performed in the compiler. For example, the print built-in was not developed until relatively late, which created the need for an alternative means for checking testcase correctness. Until then, the interpreter would print the stack on a pop operation if the pop ended up clearing the stack.

6.1 Test Suite

The test suite consisted of several testcases from each phase that proved successful implementation of the content of that particular phase. It would take ages to come up with a fully comprehensive set of tests, but an attempt was made to exercise each path allowable by the language. For illustrative purposes, here is an example testcase for the expression phase of development which tests operator precedence:

```
{
    print(5 + 6 * 2 + 3);
}
```

Some testcases in the suite ran multiple tests at once. Rather than having multiple different testcases that tested variations of the same thing, a single testcase would include all of these variations with the thought that if one test failed, they were likely to all fail. This helped to reduce the amount of time needed for debugging testcase failures after a failed run. Here is an example testcase demonstrating multiple similar tests:

```
[[ Test arithmetic operations. ]]
{
    print(3 + 4);
    print(90 - 30);
    print(0 - 90);
    print(-90);
    print(4 * 5);
    print(1000000 * 0);
    print(100 / 20);
    print(5 / 2);
}
```

Some of the final testcases were more complex, combining multiple operations into one. These were designed to be more expressive of an actual program developed by the language and thus more prone to breakage due to regression. These were mostly various implementations of different algorithms.

6.2 Automation

The test suite was driven by a makefile which could be driven once the compiler was built. As an initial step, it would compile all of the testcases in one pass. This would identify the testcases which failed due to incorrect grammar rules or bad translation. If it was the first time the testcases were executed, a generation target would be executed which would run each of the testcases, saving the output to files. These would then be verified manually. After that, all future runs could be executed with a test target, which would recompile the testcases and diff the output with the verified results.

This entire process culminates in a log file that lists the successes and failures of the latest run.

7 Lessons Learned

Personally I have found this project rather fun and interesting, but it has not been without its difficulties. One of the biggest problems that I had was not with the project itself, but with its timing. Given only two months to implement a compiler by myself was an extreme exercise in time management. Implementing this during the summer, when free time on the weekend is relatively scarce, did not help. Fortunately it was not a problem that could not be overcome, as by the end I was able to come up with a working compiler (and interpreter!).

Another difficulty that I constantly ran into was in developing the language itself. There were many other features that I wanted to add. However, when writing out examples, the new features seemed either too complicated or confusing. In the long run, removing some of these features proved to be beneficial as they would have taken too long to develop.

7.1 Advice to Future Teams

Overall, I think that the project went quite smoothly (ultimately that is up for the instructor to decide), so I can mostly offer suggestions based on things that went well, as opposed to things that went wrong.

- *Start and plan early*: I was only able to complete as much as I did by starting the project as soon as it was assigned. With dates planned out until the day this report was due, I was able to allow for any setbacks.
- *Implement features in small batches*: Had I tried to implement everything at once, there is no telling if I would have been able to finish as easily. By working in small batches, I was able to flesh out bugs early on as well as fix any design issues. Had I implemented everything at once, I could have easily been set back by crippling amounts of bugs and rework.
- *Always have a means of testing*: Even if a print-like function takes a long time to develop, a good substitute is crucial in the early stages of development.
- *When compiling to bytecode, a pretty-printer is helpful*: I did not realize this until very late, but a pretty-printer for the bytecode is really useful, as combing through binary data is both tedious and time-consuming.

Appendix I

Below is the complete grammar of Clogic, compilable by yacc without any changes except for the precedence rules above. For productions that have the suffix ***-opt***, there is an unwritten definition that chooses either no tokens or the production without ***-opt***.

Precedence table:

Operator type	Operator	Grouping	Queries
Unary	`- !`	Right-to-left	`neg/2 not/2`
Multiplicative	`* /`	Left-to-right	`mult/3 div/3`
Additive	`+ -`	Left-to-right	`add/3 sub/3`
Cons	`|`	Right-to-left	`cons/3`
Relation	`> < >= <=`	Left-to-right	`gt/2 lt/2 ge/2 le/2`
Equality	`== !=`	Left-to-right	`eq/2 ne/2`
Assignment	`=`	Right-to-left	
Logical And	`&`	Left-to-right	
Logical Or	`||`	Left-to-right	

program:
declaration-list-opt block

declaration-list:
declaration
declaration-list declaration

declaration:
predicate-clause-decl ;
truth-decl ;

truth-decl:
identifier (***literal-list***)

literal-list:
literal
literal-list , literal

predicate-clause-decl:
identifier (***literal-list***) -> ***expression***
identifier (***pattern-list-opt***) -> ***expression***

pattern-list:
pattern
literal-list , pattern
pattern-list , pattern
pattern-list , literal

pattern:
variable
variable | ***pattern***
literal | ***pattern***

literal:
integer
string
[]

primary-expression:
identifier
literal
variable
(***expression***)
[***expression-list***]
query
(***expression*** ? ***expression*** : ***expression-opt***)

query:
identifier (***query-args-opt***)

query-args:
query-arg
query-args , query-arg

query-arg:
expression
* ***variable***

unary-expression:
primary-expression
- ***unary-expression***
! ***unary-expression***

multiplicative-expression:
unary-expression
multiplicative-expression * ***unary-expression***
multiplicative-expression / ***unary-expression***

additive-expression:
    multiplicative-expression
    additive-expression + ***multiplicative-expression***
    additive-expression − ***multiplicative-expression***

cons-expression:
    additive-expression
    additive-expression | ***cons-expression***

relational-expression:
    cons-expression
    relational-expression < ***cons-expression***
    relational-expression > ***cons-expression***
    relational-expression <= ***cons-expression***
    relational-expression >= ***cons-expression***
    relational-expression == ***cons-expression***
    relational-expression != ***cons-expression***

assignment:
    relational-expression
    variable = ***assignment***

and-expression:
    assignment
    and-expression & ***assignment***

or-expression:
    and-expression
    or-expression || ***and-expression***

expression:
    or-expression

expression-list:
    expression
    expression-list , ***expression***

block:
    { ***statements-opt*** }

statements:
    statement
    statements statement

statement:

expression ;
expression `?` ***statement*** `:` ***statement***
block
`while` ***expression ; statement***
`until` ***expression ; statement***

Appendix II – Predefined Predicates

This section lists built-in predicates that are provided for use in any program. Assume that the predicates are defined before the optional declaration list, and that any return variable, if any is the last argument.

`neg/2` – Arithmetically negate an expression.
`not/2` – Logically negate an expression.
`mult/3` – Multiply two expressions.
`div/3` – Divide one expression by another.
`add/3` – Add two expressions.
`sub/3` – Subtract two expressions.
`cons/3` – Prepend an expression to a list.
`gt/2` – Test if one expression is greater than another.
`lt/2` – Test if one expression is less than another.
`ge/2` – Test if one expression is greater than or equal to another.
`le/2` – Test if one expression is less than another.
`eq/2` – Test two expressions for equality.
`ne/2` – Test two expressions for inequality.
`print/1` – Print an item, adding a newline at the end.
`write/1` – Print an item.
`exit/1` – Exit the program with the supplied return code.
`exit/0` – Exit the program with a 0 return code.
`true/1` – Test to see if an expression is true.
`true/0` – True.
`false/0` – False.
`integer/1` – Test if an item is an integer.
`string/1` – Test if an item is a string.
`boolean/1` – Test if an item is a boolean.
`atom/1` – Test if an item is an atom.
`list/1` – Test if an item is a list.
`tointeger/1` – Convert a string into an integer.
`tostring/1` – Convert an item into a string.

Appendix III – Complete Compiler Code Listing

Attached is a complete listing of the compiler, with exception of any redundant interfaces and the interpreter.

clogic.ml

```
(** clogic.ml - The compiler front-end.
   Tabari Alexander tha2105
   COMS W4115
*)
open Ast

let _ =
try
  (* The following are references used when parsing the options in the
compiler*)
  let print_ir = ref false in
  let compile_only = ref true in
  let infile = ref None in
  let outfile = ref None in
  (* Parse the options.  The return value is a Unit, it can be
ignored. *)
  let _ =
    let usage = "Usage: cl [options] file\nOptions:" in
    let options =
      [("-c", Arg.Unit(fun () -> compile_only := true), "\tCompile and
assemble the code to be interpreted by clvm.");
       ("-S", Arg.Unit(fun () -> print_ir := true), "\tPrint the IR to
a file.");
       ("-o", Arg.String(fun o -> outfile := Some o),
"<filename>\tPlace the output into <filename>.")]
    in
    Arg.parse options (fun fname -> infile := Some(fname)) usage in
  (* Determine the input source.  If no file is specified, then stdin
is used. *)
  let instream = match !infile with Some(f) -> open_in f | _ -> stdin
in
  let prefix = match !infile with
      Some(f) -> String.sub f 0 (String.rindex f '.')
    | None -> "a" in
  let outfile = match !outfile with
      Some(f) -> f
    | None -> prefix ^ ".clo" in
  let irfile = prefix ^ ".ir" in
  let lexbuf = Lexing.from_channel instream in
  (* Finally, perform front-end operations.  This will tokenize,
parse, and compile
     the input *)
  let ir = Translator.compile(Sast.translate(Parser.program
Scanner.token lexbuf))
  in
  (* Perform back-end processing.  If it was specified to print the IR
to a file,
```

```
     then it will be done before compiling the IR into bytecode. *)
  (if !print_ir then Opcodes.print_ir irfile ir);
  if !compile_only then Opcodes.assemble outfile ir
with
  Scanner.ScannerError x -> prerr_endline x; exit 2
| Sast.SyntaxError x -> prerr_endline x; exit 2
| Failure x -> prerr_endline("Unexpected error: " ^ x); exit 2
```

scanner.mll

```
(** scanner.mll - the lexical analyzer.
   Tabari Alexander tha2105
   COMS W4115
*)
{ open Parser
  exception ScannerError of string
}
let ucletter = ['A'-'Z']
let lcletter = ['a'-'z' '_']
let letter = ucletter|lcletter
let digit = ['0'-'9']
rule token = parse
      [' ' '\t' '\r' '\n']+ { token lexbuf }
  | '+'                  { PLUS }
  | '-'                  { MINUS }
  | '*'                  { TIMES }
  | '/'                  { DIVIDE }
  | '='                  { ASSIGN }
  | '('                  { LPAREN }
  | ')'                  { RPAREN }
  | '{'                  { LBRACE }
  | '}'                  { RBRACE }
  | '['                  { LBRACK }
  | ']'                  { RBRACK }
  | '|'                  { CONS }
  | '!'                  { NOT }
  | '?'                  { THEN }
  | ':'                  { ELSE }
  | '>'                  { GT }
  | '<'                  { LT }
  | '&'                  { AND }
  | ';'                  { SEMICOLON }
  | ','                  { COMMA }
  | ">="                 { GTEQ }
  | "<="                 { LTEQ }
  | "||"                 { OR }
  | "=="                 { EQ }
  | "!="                 { NE }
  | "[]"                 { EMPTYLIST }
  | "[["                 { comment lexbuf } (* Comment *)
  | "->"                 { IMPLY }
  | "in"                 { IN }
  | "isa"                { ISA }
  | "type"               { TYPE }
  | "when"               { WHEN }
  | "until"              { UNTIL }
```

```
  | "where"             { WHERE }
  | "while"             { WHILE }
  | ucletter(letter|digit)* as lit { VARIABLE lit }
  | lcletter(letter|digit)* as lit { ID lit }
  | ['1'-'9']digit*|'0' as lit { INTLIT(int_of_string lit) }
  | '"' (([^ '"' '\\'] | '\\'['\\' '"' 't' 'n'])* as s) '"'
{ STRINGLIT s }
  | eof                 { EOF }
  | _ as ch             { raise(ScannerError("Unexpected character: "
^ (Char.escaped ch))) }
(* Comment - it just gobbles everything up and tosses it. *)
and comment = parse
      "]]"                    { token lexbuf }
  | eof                 { raise(ScannerError "Unexpected end of file
while handling a comment!") }
  | _                       { comment lexbuf }
```

parser.mly

```
/** parser.mly - grammar rules.
    Tabari Alexander tha2105
    COMS W4115 */

%{
  open Ast

  module StringMap = Map.Make(String)

  type decl = {nargs:int; truths:pred list; preds:pred list}

  let rec ptrn_of_lit = function
      IntLit(x) -> IntPtrn(x)
    | AtomLit(x) -> AtomPtrn(x)
    | Var(x) -> VarPtrn(x)
    | EmptyList -> EmptyPtrn
    | _ -> raise(Failure "Unexpected literal in pattern")

  let predefines =
    [("true", 0, Pred([],
       Some(Call("__true", [])))));
     ("false", 0, Pred([],
       Some(Call("__false", [])))));
     ("neg", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
       Some(LogOp(Call("__integer", [Var "X"]), And, Call("__neg",
[Var "X"; Var "Y"])))));
     ("tointeger", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
       Some(LogOp(Call("__string", [Var "X"]), And,
Call("__tointeger",[Var "X"; Var "Y"])))));
     ("tostring", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
       Some(Call("__neg",[Var "X"; Var "Y"])))));
     ("exit", 1, Pred([VarPtrn "X"],
       Some(Call("__exit",[Var "X"])))));
     ("exit", 0, Pred([],
       Some(Call("__exit",[IntLit 0])))));
     ("true", 1, Pred([VarPtrn "X"],
       Some(Call("__true",[Var "X"])))));
```

```
     ("integer", 1, Pred([VarPtrn "X"],
        Some(Call("__integer",[Var "X"]))));
     ("string", 1, Pred([VarPtrn "X"],
        Some(Call("__string",[Var "X"]))));
     ("boolean", 1, Pred([VarPtrn "X"],
        Some(Call("__boolean",[Var "X"]))));
     ("atom", 1, Pred([VarPtrn "X"],
        Some(Call("__atom",[Var "X"]))));
     ("list", 1, Pred([VarPtrn "X"],
        Some(Call("__list",[Var "X"]))));
     ("list", 1, Pred([EmptyPtrn], None));
     ("not", 1, Pred([VarPtrn "X"],
        Some(Call("__not",[Call("__true",[Var "X"])])))));
     ("gt", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__gt",[Var "X"; Var
"Y"])))));
     ("lt", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__lt",[Var "X"; Var
"Y"])))));
     ("ge", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__ge",[Var "X"; Var
"Y"])))));
     ("le", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__le",[Var "X"; Var
"Y"])))));
     ("eq", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(Call("__eq",[Var "X"; Var "Y"]))));
     ("ne", 2, Pred([VarPtrn "X"; VarPtrn "Y"],
        Some(Call("__ne",[Var "X"; Var "Y"]))));
     ("cons", 3, Pred([VarPtrn "X"; VarPtrn "Y"; VarPtrn "Z"],
        Some(LogOp(LogOp(RelOp(Var "Y",Eq,
EmptyList),Or,Call("__list", [Var "Y"])), And, Call("__cons",[Var "X";
Var "Y"; Var "Z"])))));
     ("mult", 3, Pred([VarPtrn "X"; VarPtrn "Y"; VarPtrn "Z"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__mult",[Var "X"; Var "Y";
Var "Z"])))));
     ("div", 3, Pred([VarPtrn "X"; VarPtrn "Y"; VarPtrn "Z"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__div",[Var "X"; Var "Y";
Var "Z"])))));
     ("div", 3, Pred([VarPtrn "X"; IntPtrn 0; VarPtrn "Z"],
        Some(LogOp(Call("__print", [StrLit "Divide by zero error"]),
And, Call("__exit", [IntLit 1])))));
     ("add", 3, Pred([VarPtrn "X"; VarPtrn "Y"; VarPtrn "Z"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__add",[Var "X"; Var "Y";
Var "Z"])))));
     ("sub", 3, Pred([VarPtrn "X"; VarPtrn "Y"; VarPtrn "Z"],
        Some(LogOp(LogOp(Call("__integer", [Var "X"]), And,
Call("__integer", [Var "Y"])), And, Call("__sub",[Var "X"; Var "Y";
Var "Z"])))));
```

```
     ("print", 1, Pred([VarPtrn "X"],
        Some(Call("__print",[Var "X"]))));
     ("write", 1, Pred([VarPtrn "X"],
        Some(Call("__write", [Var "X"]))));
     ("write", 1, Pred([ConsPtrn(VarPtrn "H", VarPtrn "T")],
        Some(LogOp(Call("write", [Var "H"]), And, Call("write", [Var
"T"])))));
     ("write", 1, Pred([VarPtrn "X"],
        Some(LogOp(Call("__string", [Var "X"]), And, Call("__write",
[Var "X"])))))]

  let add_decl sm = function
    (n, a, p) ->
      let name = n ^ "/" ^ (string_of_int a) in
      let r =
        try
          StringMap.find name sm
        with
          Not_found -> {nargs = a; truths=[]; preds=[]} in
      let r =
        (match p with Pred(_, Some e) -> { r with preds = p::r.preds }
          | Pred(_, None) -> { r with truths = p::r.truths })
      in
      StringMap.add name r sm

%}
%token EOF LPAREN RPAREN LBRACE RBRACE SEMICOLON COMMA IMPLY IN LBRACK
RBRACK
%token CONS ASSIGN ISA PLUS MINUS TIMES DIVIDE CONS TYPE THEN ELSE NOT
AND OR
%token LT GT LTEQ GTEQ EQ NE EMPTYLIST WHERE UNTIL WHEN WHILE DEFINE
UNDEFINE
%token <string> VARIABLE ID STRINGLIT
%token <int> INTLIT

%start program
%type <Ast.prog> program

%%

program:
    decls_opt block
      { Prog(StringMap.fold (fun n r l -> {name=n; decls =
List.rev_append r.truths r.preds; arity = r.nargs}::l) $1 [],$2) }

literal:
    INTLIT { IntLit $1 }
  | ID { AtomLit $1 }
  | STRINGLIT { StrLit $1 }
  | EMPTYLIST { EmptyList }

primary_expr:
    literal { $1 }
  | VARIABLE { Var $1  }
  | LPAREN expr RPAREN { $2 }
  | LBRACK expr_list RBRACK { List.fold_left (fun e li -> BinOp(li,
```

```
Cons, e)) EmptyList $2 }
  | query { $1 }
  | LPAREN expr THEN expr ELSE expr_opt RPAREN { IfExpr($2, $4, $6) }

query:
    ID LPAREN query_args_opt RPAREN { Call($1, List.rev $3) }

unary_expr:
    primary_expr { $1 }
  | MINUS unary_expr { Neg $2 }
  | NOT unary_expr { Not $2 }

mult_expr:
    unary_expr { $1 }
  | mult_expr TIMES unary_expr { BinOp($1, Mul, $3) }
  | mult_expr DIVIDE unary_expr { BinOp($1, Div, $3) }

add_expr:
    mult_expr { $1 }
  | add_expr PLUS mult_expr { BinOp($1, Add, $3) }
  | add_expr MINUS mult_expr { BinOp($1, Sub, $3) }

cons_expr:
    add_expr { $1 }
  | add_expr CONS cons_expr { BinOp($1, Cons, $3) }

rel_expr:
    cons_expr { $1 }
  | rel_expr LT cons_expr { RelOp($1, Lt, $3) }
  | rel_expr GT cons_expr { RelOp($1, Gt, $3) }
  | rel_expr LTEQ cons_expr { RelOp($1, LtEq, $3) }
  | rel_expr GTEQ cons_expr { RelOp($1, GtEq, $3) }
  | rel_expr EQ cons_expr { RelOp($1, Eq, $3) }
  | rel_expr NE cons_expr { RelOp($1, Ne, $3) }

assignment:
    rel_expr { $1 }
  | VARIABLE ASSIGN rel_expr { Asn($1, $3) }

and_expr:
    assignment { $1 }
  | and_expr AND assignment { LogOp($1, And, $3) }

or_expr:
    and_expr { $1 }
  | or_expr OR and_expr { LogOp($1, Or, $3) }

expr:
    or_expr { $1 }

query_args_opt:
      { [] }
  | query_args { $1 }

query_args:
    query_arg { [$1] }
```

```
  | query_args COMMA query_arg { $3::$1 }

query_arg:
    expr { $1 }
  | TIMES VARIABLE { RetVar $2 }

expr_list:
      expr { [$1] }
  | expr_list COMMA expr { $3::$1 }

expr_opt:
    { EmptyList }
  | expr { $1 }

decls_opt:
    { List.fold_left add_decl StringMap.empty predefines }
  | decls { $1 }

decls:
    decl { add_decl (List.fold_left add_decl StringMap.empty
predefines) $1 }
  | decls decl { add_decl $1 $2 }

decl:
    rule_decl SEMICOLON { $1 }
  | truth_decl SEMICOLON { $1 }

truth_decl:
    ID LPAREN literal_list RPAREN
      { ($1, List.length $3, Pred(List.rev (List.map ptrn_of_lit $3),
None)) }

literal_list:
    literal { [$1] }
  | literal_list COMMA literal { $3::$1 }

rule_decl:
    ID LPAREN literal_list RPAREN IMPLY expr
     { ($1, List.length $3, Pred(List.rev (List.map ptrn_of_lit $3),
Some $6)) }
  | ID LPAREN pattern_list_opt RPAREN IMPLY expr
     { ($1, List.length $3, Pred(List.rev $3, Some $6)) }

pattern_list_opt:
    { [] }
  | pattern_list { $1 }

pattern_list:
    pattern { [$1] }
  | literal_list COMMA pattern { $3::(List.map ptrn_of_lit $1) }
  | pattern_list COMMA pattern { $3::$1 }
  | pattern_list COMMA literal { (ptrn_of_lit $3)::$1 }

pattern:
    VARIABLE { VarPtrn($1) }
  | VARIABLE CONS pattern { ConsPtrn(VarPtrn($1), $3) }
```

```
  | VARIABLE CONS EMPTYLIST { ConsPtrn(VarPtrn($1), EmptyPtrn) }
  | literal CONS pattern { ConsPtrn(ptrn_of_lit $1,$3) }
  | literal CONS EMPTYLIST { ConsPtrn(ptrn_of_lit $1, EmptyPtrn) }

block:
    LBRACE statements RBRACE { Block(List.rev $2) }
  | LBRACE RBRACE { Block([]) }

statements:
    statement { [$1] }
  | statements statement { $2::$1 }

statement:
    expr SEMICOLON { ExprStmt $1 }
  | expr THEN statement ELSE statement { IfStmt($1, $3, $5) }
  | block { $1 }
  | WHILE expr SEMICOLON statement { While($2,$4) }
  | UNTIL expr SEMICOLON statement { Until($2,$4) }
```

ast.mli

```
(** ast.mli - This file defines the types for an Abstract Syntax Tree
that is produced by
   the parser.
   Tabari Alexander tha2105
   COMS W4115
*)

(** The set of operators for an arithmetic op *)
type operator = Add | Sub | Mul | Div | Cons

(** The set of operators for a relation op *)
type relop = Eq | Ne | Lt | Gt | LtEq | GtEq

(** The set of operators for a logic op *)
type logicop = And | Or

(** Expressions that mostly map to the expressions in the language's
grammar. *)
type expr =
    IntLit of int
  | AtomLit of string
  | EmptyList
  | StrLit of string
  | Call of string * expr list
  | BinOp of expr * operator * expr
  | RelOp of expr * relop * expr
  | LogOp of expr * logicop * expr
  | Var of string
  | RetVar of string
  | Asn of string * expr
  | IfExpr of expr * expr * expr
  | Not of expr
  | Neg of expr

(** Patterns are used in predicate definition arguments. *)
```

```
type ptrn =
    IntPtrn of int
  | AtomPtrn of string
  | VarPtrn of string
  | ConsPtrn of ptrn * ptrn
  | EmptyPtrn

type pred = Pred of ptrn list * expr option

type stmt =
    ExprStmt of expr
  | IfStmt of expr * stmt * stmt
  | Block of stmt list
  | While of expr * stmt
  | Until of expr * stmt

type decl = {name : string; arity : int; decls : pred list}

type prog =
    Prog of decl list * stmt
```

sast.ml

```
(** This file defines the types for an Abstract Syntax Tree that is
produced by
   the parser.
*)

(** The set of operators for an arithmetic op *)
type operator = Add | Sub | Mul | Div | Cons

(** The set of operators for a relation op *)
type relop = Eq | Ne | Lt | Gt | LtEq | GtEq

(** The set of operators for a logic op *)
type logicop = And | Or

(** Expressions that mostly map to the expressions in the language's
grammar. *)
type expr =
    IntLit of int
  | AtomLit of string
  | BoolLit of int
  | EmptyList
  | StrLit of string
  | Call of string * expr list
  | BinOp of expr * Ast.operator * expr
  | RelOp of expr * Ast.relop * expr
  | LogOp of expr * Ast.logicop * expr
  | AsnEq of expr * expr
  | Var of string
  | RetVar of string
  | Asn of string * expr
  | SetAsn of string * string
  | IfExpr of expr * expr * expr
  | Not of expr
```

```
  | Seq of expr list
  | Clear of string
  | TestListExpr of expr
  | HeadExpr of expr
  | TailExpr of expr
  | Ignore of expr

type decl = Decl of string * int * expr

type stmt =
    ExprStmt of expr
  | IfStmt of expr * stmt * stmt
  | Block of stmt list
  | While of expr * stmt
  | Until of expr * stmt

type prog =
    Prog of decl list * stmt

exception SyntaxError of string

module StringSet = Set.Make(String)

(** var translate : Ast.prog -> prog *)
let translate =
  (* Helper function for providing overrides for operators. *)
  let rec create_override name exprs =
    Seq [Ignore(Call(name, List.map (fun e -> doexpr e) exprs)); Var
"_ans"]
  (* Helper function for changing binary op types to predicate names.
*)
  and getbinop = function
      Ast.Add -> "add/3"
    | Ast.Sub -> "sub/3"
    | Ast.Mul -> "mult/3"
    | Ast.Div -> "div/3"
    | Ast.Cons -> "cons/3"
  (* Function for translating and syntax checking an expression *)
  and doexpr = function
      Ast.IntLit(x) -> IntLit x
    | Ast.AtomLit(x) -> AtomLit x
    | Ast.EmptyList -> EmptyList
    | Ast.StrLit(x) ->
      (* Check string literals first before continuing onwards *)
      let len = String.length x in
      let rec check_escape(b,i) =
        if i == len then
          raise(SyntaxError "Bad escape found in string literal.")
        else
          (match x.[i] with
              '\\' -> Buffer.add_char b '\\'; check_string(b, i+1)
            | 't' -> Buffer.add_char b '\t'; check_string(b, i+1)
            | 'n' -> Buffer.add_char b '\n'; check_string(b, i+1)
            | '"' -> Buffer.add_char b '"'; check_string(b, i+1)
            | _ -> raise(SyntaxError "Bad escape sequence found in
string literal."))
```

```
      and check_string(b, i) =
        if i == len then
          Buffer.contents b
        else
          (match x.[i] with
              '\\' -> check_escape(b, i+1)
            | ch -> Buffer.add_char b ch; check_string(b, i+1))
      in
      StrLit(check_string(Buffer.create len,0))
    | Ast.Call(name, exprs) ->
      (* Transform the query name into a predicate name in addition to
providing
         additional logic to perform the return variable setting.*)
      let callexpr = Call(name ^ "/" ^ (string_of_int(List.length
exprs)), List.map doexpr exprs) in
      let retvars =
        let addvar vars = function
            Ast.RetVar(v) -> if List.mem v vars then
                raise(SyntaxError("Error - Return variable \"" ^ v ^
"\" used multiple times in a query"))
              else
                v::vars
          | _ -> vars
        in
        List.fold_left addvar [] exprs
      in
      if retvars = [] then
        callexpr
      else
        LogOp(callexpr, Ast.Or, Seq([Seq(List.map (fun v -> Clear v)
retvars);BoolLit 0]))
    | Ast.BinOp(e1, op, e2) -> create_override (getbinop op) [e1; e2;
Ast.RetVar "_ans"]
    | Ast.RelOp(e1, op, e2) -> RelOp(doexpr e1, op, doexpr e2)
    | Ast.LogOp(e1, op, e2) -> LogOp(doexpr e1, op, doexpr e2)
    | Ast.Var(x) -> Var x
    | Ast.RetVar(x) -> RetVar x
    | Ast.Asn(var, e) -> Asn(var, doexpr e)
    | Ast.IfExpr(e1, e2, e3) -> IfExpr(doexpr e1, doexpr e2, doexpr
e3)
    | Ast.Not(e) -> Not(doexpr e)
    | Ast.Neg(e) -> BinOp(IntLit 0, Ast.Sub, doexpr e)
  (* Function that translates the AST of a statement into a checked
AST *)
  and dostmt = function
      Ast.ExprStmt(x) -> ExprStmt(doexpr x)
    | Ast.IfStmt(e, s1, s2) -> IfStmt(doexpr e, dostmt s1, dostmt s2)
    | Ast.Block(stmts) -> Block(List.map dostmt stmts)
    | Ast.While(e, s) -> While(doexpr e, dostmt s)
    | Ast.Until(e, s) -> Until(doexpr e, dostmt s)
  (* Converts declarations from their multiple definition form into
long expressions. *)
  and dodecl decls =
    let map_default f o d = match o with Some(x) -> f x | _ -> d in
    let get o = match o with Some(x) -> x | _ -> raise Not_found in
    let join op o e =
```

```
      Some(map_default (fun v -> LogOp(v, op, e)) o e) in
    let dopred arity = function
      Ast.Pred(ptrns, clause) ->
        let rec check_arg (i, (seen,e)) p =
          let var = string_of_int i in
          (* Helper function for providing the additional logic to
pattern match against a list pattern. *)
          let rec check_cons ((seen, e), v) = function
              Ast.IntPtrn x -> (seen, join Ast.And e (RelOp(v, Ast.Eq,
IntLit x)))
            | Ast.AtomPtrn x -> (seen, join Ast.And e (RelOp(v,
Ast.Eq, AtomLit x)))
            | Ast.EmptyPtrn -> (seen, join Ast.And e (RelOp(v, Ast.Eq,
EmptyList)))
            | Ast.VarPtrn x -> if StringSet.mem x seen then
                (seen, join Ast.And e (RelOp(Var x, Ast.Eq, v)))
              else
                (StringSet.add x seen, join Ast.And e (Asn(x, v)))
            | Ast.ConsPtrn(p1, p2) ->
              let e = join Ast.And e (TestListExpr(Var "_tmp")) in
              let (seen, e) = check_cons((seen, e), HeadExpr(Var
"_tmp")) p1
              in
              check_cons ((seen, join Ast.And e (Asn("_tmp",
TailExpr(Var "_tmp")))), Var "_tmp") p2
          in
          (i + 1, (match p with
              Ast.IntPtrn x -> (seen, join Ast.And e (AsnEq(Var var,
IntLit x)))
            | Ast.AtomPtrn x -> (seen, join Ast.And e (AsnEq(Var var,
AtomLit x)))
            | Ast.EmptyPtrn -> (seen, join Ast.And e (AsnEq(Var var,
EmptyList)))
            | Ast.VarPtrn x -> if StringSet.mem x seen then
                (seen, join Ast.And e (RelOp(Var x, Ast.Eq, Var var)))
              else
                (StringSet.add x seen, join Ast.And e (SetAsn(x,
var)))
            | Ast.ConsPtrn(_, _) -> check_cons ((seen, join Ast.And e
(SetAsn("_tmp",var))), Var "_tmp") p)) in
        let argexp = snd(snd(List.fold_left check_arg (0,
(StringSet.empty, None)) ptrns))
        in
        map_default (fun e -> join Ast.And argexp (doexpr e)) clause
argexp
    in
    Decl(decls.Ast.name, decls.Ast.arity,
           get(List.fold_left
                (fun e p -> join Ast.Or e (get(dopred decls.Ast.arity
p)))
                None decls.Ast.decls))
in
function Ast.Prog(d, s) -> Prog(List.map dodecl d, dostmt s)
```

builtins.ml

```
(** builtins.ml - defines the IR for built-in predicates
   Tabari Alexander tha2105
   COMS W4115
*)
module StringMap = Map.Make(String)

let builtins =
  (* This is a list of all the builtin predicates and their IR
statements. *)
  let builtin_list = [("__neg/2", [PushInt 0; Load 0; SubOp; Store 1;
PushBool 1]);
                      ("__not/1", [Load 0; NotOp]);
                      ("__mult/3", [Load 0; Load 1; MulOp; Store 2;
PushBool 1]);
                      ("__div/3", [Load 0; Load 1; DivOp; Store 2;
PushBool 1]);
                      ("__add/3", [Load 0; Load 1; AddOp; Store 2;
PushBool 1]);
                      ("__sub/3", [Load 0; Load 1; SubOp; Store 2;
PushBool 1]);
                      ("__cons/3", [Load 0; Load 1; ConsOp; Store 2;
PushBool 1]);
                      ("__gt/2", [Load 0; Load 1; GtOp]);
                      ("__lt/2", [Load 0; Load 1; LtOp]);
                      ("__ge/2", [Load 0; Load 1; GtEqOp]);
                      ("__le/2", [Load 0; Load 1; LtEqOp]);
                      ("__eq/2", [Load 0; Load 1; EqOp]);
                      ("__ne/2", [Load 0; Load 1; NeOp]);
                      ("__print/1", [Load 0; BuiltIn(1);]);
                      ("__write/1", [Load 0; BuiltIn(2);]);
                      ("__exit/1", [Load 0; Halt]);
                      ("__exit/0", [PushInt 0; Halt]);
                      ("__true/1", [Load 0; BuiltIn(3)]);
                      ("__true/0", [PushBool(1)]);
                      ("__false/0", [PushBool(0)]);
                      ("__integer/1", [Load 0; TestInt]);
                      ("__string/1", [Load 0; TestString]);
                      ("__boolean/1", [Load 0; TestBool]);
                      ("__atom/1", [Load 0; TestAtom]);
                      ("__list/1", [Load 0; TestList]);
                      ("__tointeger/2", [Load 0; BuiltIn(4); Store 1;
PushBool 1]);
                      ("__tostring/2", [Load 0; BuiltIn(5); Store 1;
PushBool 1])]
in
List.fold_left (fun m (n,c) -> StringMap.add n c m) StringMap.empty
builtin_list
```

translator.ml

```
(** translator.ml - defines the functions responsible for compiling
from SAST to
   IR
   Tabari Alexander tha2105
```

```
   COMS W4115
*)
open Opcodes
open Sast
open Builtins

module Globals = Set.Make(String)
module StringMap = Map.Make(String)

type env = {locals : int StringMap.t; globals : Globals.t; labelind :
int}

(** var compile : prog -> opcode list *)
let compile =
  (* creates a new label, incrementing the label index *)
  let getlabel env =
    ({env with labelind = env.labelind + 1}, "L" ^ (string_of_int
env.labelind)) in
  (* returns the offset into the stack for the local variable.  If not
found, a
     new entry is created. *)
  let getlocal (i, sm) name=
    if StringMap.mem name sm then (i, sm) else (i+1, StringMap.add
name i sm) in
  (* Finds and allocates space for all variables within an expression.
*)
  let rec exprlocals t = function
      Call(_, exprs) -> List.fold_left exprlocals t exprs
    | Seq(exprs) -> List.fold_left exprlocals t exprs
    | BinOp(e1, _, e2) -> List.fold_left exprlocals t [e2;e1]
    | RelOp(e1, _, e2) -> List.fold_left exprlocals t [e2;e1]
    | LogOp(e1, _, e2) -> List.fold_left exprlocals t [e2;e1]
    | Var s -> getlocal t s
    | RetVar s -> getlocal t s
    | Asn(s, e) -> getlocal (exprlocals t e) s
    | SetAsn(s1, s2) -> List.fold_left getlocal t [s2;s1]
    | IfExpr(e1, e2, e3) -> List.fold_left exprlocals t [e3;e2;e1]
    | Not e -> exprlocals t e
    | TestListExpr e -> exprlocals t e
    | HeadExpr e -> exprlocals t e
    | TailExpr e -> exprlocals t e
    | Ignore e -> exprlocals t e
    | _ -> t in
  (* Finds and allocates all locals within a statement *)
  let rec stmtlocals t = function
      ExprStmt(e) -> exprlocals t e
    | IfStmt(e1, s1, s2) -> exprlocals (List.fold_left stmtlocals t
[s2;s1]) e1
    | Block(stmts) -> List.fold_left stmtlocals t (List.rev stmts)
    | While(e, s) -> exprlocals (stmtlocals t s) e
    | Until(e, s) -> exprlocals (stmtlocals t s) e in
  (* Transforms an expression into an IR.  It is built bottom up to
ease the
     burden on the code generation. *)
  let rec expr (env, code) = function
      IntLit(i) -> (env, PushInt(i)::code)
```

```
    | AtomLit(s) -> (env, PushAtom(s)::code)
    | BoolLit(i) -> (env, PushBool(i)::code)
    | EmptyList -> (env, PushEmpty::code)
    | StrLit(s) -> (env, PushString(s)::code)
    | Call(s, exprs) ->
      let frameop = PushFrame(StringMap.fold (fun _ _ i -> i + 1)
env.locals 0) in
      let code =
        List.fold_left
         (fun c e -> match e with
           RetVar s -> StopBT(StringMap.find s env.locals)::c | _ ->
c) code exprs in
      (* Resolve the predicate name.  If it is a builtin, use the
builtin code.
         If found, generate a call.  Otherwise, generate a call to
false/0 *)
      let code =
        try
          (StringMap.find s builtins) @ (PopFrame::code)
        with
          Not_found -> let s = if Globals.mem s env.globals then s
else "false/0" in
            CallOp(s)::code in
      let rec storeargs i code =
        if i >= 0 then
          storeargs (i-1) (StoreVal(i)::code)
        else
          code in
      (* Generate code for storing arguments *)
      let (env, code) =
        List.fold_left expr
          (env, frameop::(storeargs (List.length exprs - 1) code))
exprs
      in
      (env, List.fold_left
        (fun c e -> match e with RetVar s -> StartBT(StringMap.find s
env.locals)::c | _ -> c) code exprs)
    | Seq(exprs) -> List.fold_left expr (env, code) (List.rev exprs)
    | BinOp(e1, op, e2) ->
      let oper = (match op with
          Ast.Add  -> AddOp
        | Ast.Sub  -> SubOp
        | Ast.Mul  -> MulOp
        | Ast.Div  -> DivOp
        | Ast.Cons -> ConsOp)
      in
      expr (expr (env, oper::code) e2) e1
    | RelOp(e1, op, e2) ->
      let oper = (match op with
          Ast.Eq   -> EqOp
        | Ast.Ne   -> NeOp
        | Ast.Lt   -> LtOp
        | Ast.Gt   -> GtOp
        | Ast.LtEq -> LtEqOp
        | Ast.GtEq -> GtEqOp)
      in
```

```
      expr (expr (env, oper::code) e2) e1
    | LogOp(e1, op, e2) ->
      let (env, l0) = getlabel env in
      let (env, code) = expr (env, Label(l0)::code) e2
      in
      (match op with
          Ast.And -> expr (env, Jz(l0)::Pop::code) e1
        | Ast.Or  -> expr (env, Jnz(l0)::Pop::code) e1)
    | Var(s) -> (env, Load(StringMap.find s env.locals)::code)
    | RetVar(s) -> (env, PushRef(StringMap.find s env.locals)::code)
    | Asn(s, e) -> expr (env, Store(StringMap.find s
env.locals)::PushBool(1)::code) e
    | SetAsn(s1, s2) -> (env, Load(StringMap.find s2
env.locals)::StoreVal(StringMap.find s1
env.locals)::PushBool(1)::code)
    | AsnEq(e1, e2) -> List.fold_left expr (env, AsnEqOp::code)
[e2;e1]
    | Clear(s) -> (env, ClearVal(StringMap.find s env.locals)::code)
    | IfExpr(e1, e2, e3) ->
      let (env, l0) = getlabel env in
      let (env, l1) = getlabel env in
      let (env, code) = expr (env, Label(l1)::code) e3 in
      let (env, code) = expr (env, Goto(l1)::Label(l0)::code) e2
      in
      expr (env, Jz(l0)::Pop::code) e1
    | Not(e) -> expr (env, NotOp::code) e
    | TestListExpr e -> expr (env, TestList::code) e
    | HeadExpr e -> expr (env, Head::code) e
    | TailExpr e -> expr (env, Tail::code) e
    | Ignore e -> expr (env, Pop::code) e
  and stmt (env, code) = function
      ExprStmt(e) -> expr (env, Pop::code) e
    | IfStmt(e, s1, s2) ->
      let (env, l0) = getlabel env in
      let (env, l1) = getlabel env in
      let (env, code) = stmt (env, Label(l1)::code) s2 in
      let (env, code) = stmt (env, Goto(l1)::Label(l0)::Pop::code) s1
      in
      expr (env, Jz(l0)::Pop::code) e
    | Block(stmts) -> List.fold_left stmt (env, code) (List.rev stmts)
    | While(e, s) ->
      let (env, l0) = getlabel env in
      let (env, l1) = getlabel env in
      let (env, code) = stmt (env, Goto(l0)::Label(l1)::Pop::code) s
in
      let (env, code) = expr (env, Jz(l1)::Pop::code) e
      in
      (env, Label(l0)::code)
    | Until(e, s) ->
      let (env, l0) = getlabel env in
      let (env, l1) = getlabel env in
      let (env, code) = expr (env, Jnz(l1)::Pop::code) e in
      let (env, code) = stmt (env, code) s
      in
      (env, Goto(l0)::Label(l1)::Pop::Label(l0)::code)
  and dodecl (env, code) = function
```

```
    Decl(name, arity, e) ->
      let rec get_arg_vars i sm =
        if i >= 0 then
          get_arg_vars (i-1) (StringMap.add (string_of_int i) i sm)
        else
          sm in
      let (env, code) = expr ({env with locals = snd(exprlocals
(arity, get_arg_vars (arity-1) StringMap.empty) e)},
PopFrame::Ret::code) e
      in
      (env, Label(name)::code)
  in
function Prog(decls, s) ->
  let env =
    {globals = List.fold_left
       (fun g d -> match d with Decl(n,_,_) -> Globals.add n g)
Globals.empty decls;
     locals = StringMap.empty;
     labelind = 0} in
  let (env, code) =
    List.fold_left dodecl (env, []) decls
  in
  snd(stmt ({env with locals = snd(stmtlocals (0,StringMap.empty) s)},
PushInt(0)::Halt::code) s)
```

opcodes.ml

```
(** opcodes.ml - defines the instructions in the IR and their bytecode
   equivalents
   Tabari Alexander tha2105
   COMS W4115
*)
module OffsetMap = Map.Make(String)
module StringMap = Map.Make(String)

type opcode =
    AddOp
  | SubOp
  | MulOp
  | DivOp
  | ConsOp
  | NotOp
  | PushInt of int
  | PushAtom of string
  | PushEmpty
  | PushBool of int
  | PushString of string
  | PushRef of int
  | Head
  | Tail
  | Pop
  | PushFrame of int
  | Store of int
  | StoreVal of int
  | ClearVal of int
  | CallOp of string
```

```
  | BuiltIn of int
  | Ret
  | Load of int
  | EqOp
  | AsnEqOp
  | NeOp
  | LtOp
  | LtEqOp
  | GtOp
  | GtEqOp
  | TestInt
  | TestAtom
  | TestBool
  | TestList
  | TestString
  | Jnz of string
  | Jz of string
  | Goto of string
  | PopFrame
  | TodoOp
  | Halt
  | Label of string
  | StartBT of int
  | StopBT of int

type asm_env = { consts : int StringMap.t; nextconst : int; offsets :
int OffsetMap.t }
let op_length = function
    PushInt _     -> 5
  | PushAtom _    -> 5
  | PushRef _     -> 2
  | PushString _ -> 5
  | Store _       -> 2
  | StoreVal _    -> 2
  | ClearVal _    -> 2
  | CallOp _      -> 5
  | BuiltIn _     -> 2
  | Load _        -> 2
  | Jnz _         -> 5
  | Jz _          -> 5
  | Goto _        -> 5
  | PushBool _    -> 2
  | PushFrame _   -> 2
  | Label _       -> 0
  | StartBT _     -> 0 (* Not really necessary *)
  | StopBT _      -> 2
  | _             -> 1

let int_of_op = function
    AddOp         -> 10
  | SubOp         -> 11
  | MulOp         -> 12
  | DivOp         -> 13
  | ConsOp        -> 14
  | NotOp         -> 15
  | Pop           -> 2
```

```
    | PushInt _     -> 1
    | PushAtom _    -> 3
    | PushEmpty     -> 4
    | PushBool _    -> 7
    | PushString _  -> 98
    | PushRef _     -> 99
    | Head          -> 8
    | Tail          -> 9
    | PushFrame _   -> 5
    | Store _       -> 50
    | CallOp _      -> 60
    | BuiltIn _     -> 64
    | Ret           -> 70
    | Load _        -> 51
    | StoreVal _    -> 52
    | ClearVal _    -> 53
    | EqOp          -> 30
    | AsnEqOp       -> 31
    | NeOp          -> 32
    | LtOp          -> 33
    | LtEqOp        -> 34
    | GtOp          -> 35
    | GtEqOp        -> 29
    | TestInt       -> 36
    | TestAtom      -> 37
    | TestBool      -> 38
    | TestList      -> 39
    | TestString    -> 40
    | Jnz _         -> 61
    | Jz _          -> 62
    | Goto _        -> 63
    | PopFrame      -> 6
    | TodoOp        -> 42
    | Halt          -> 0
    | StartBT _     -> 80
    | StopBT _      -> 81
    | Label _       -> raise(Failure "Unexpected instruction seen")

let string_of_op = function
    AddOp         -> "    ADD"
  | SubOp         -> "    SUB"
  | MulOp         -> "    MUL"
  | DivOp         -> "    DIV"
  | ConsOp        -> "    CONS"
  | NotOp         -> "    NOT"
  | Pop           -> "    POP"
  | PushInt x     -> "    PUSHINT " ^ (string_of_int x)
  | PushAtom x    -> "    PUSHATOM " ^ x
  | PushEmpty     -> "    PUSHEMPTY"
  | PushBool x    -> "    PUSHBOOL " ^ (string_of_int x)
  | PushString x  -> "    PUSHSTRING " ^ x
  | PushRef x     -> "    PUSHREF " ^ (string_of_int x)
  | Head          -> "    HEAD"
  | Tail          -> "    TAIL"
  | PushFrame x   -> "    PUSHFRAME " ^ (string_of_int x)
  | Store x       -> "    STORE " ^ (string_of_int x)
```

```
  | StoreVal x   -> "    STOREVAL " ^ (string_of_int x)
  | ClearVal x   -> "    CLEARVAL " ^ string_of_int x
  | CallOp x     -> "    CALLOP " ^ x
  | BuiltIn x    -> "    BUILTIN " ^ (string_of_int x)
  | Ret          -> "    RET"
  | Load x       -> "    LOAD " ^ (string_of_int x)
  | EqOp         -> "    EQ"
  | AsnEqOp      -> "    ASNEQ"
  | NeOp         -> "    NE"
  | LtOp         -> "    LT"
  | LtEqOp       -> "    LTEQ"
  | GtOp         -> "    GT"
  | GtEqOp       -> "    GTEQ"
  | TestInt      -> "    TESTINT"
  | TestAtom     -> "    TESTATOM"
  | TestBool     -> "    TESTBOOL"
  | TestList     -> "    TESTLIST"
  | TestString   -> "    TESTSTRING"
  | Jnz x        -> "    JNZ " ^ x
  | Jz x         -> "    JZ " ^ x
  | Goto x       -> "    GOTO " ^ x
  | PopFrame     -> "    POPFRAME"
  | Halt         -> "    HALT"
  | Label x      -> x
  | StartBT x    -> "    STARTBT " ^ string_of_int x
  | StopBT x     -> "    STOPBT " ^ string_of_int x
  | TodoOp       -> "    TODO"

(** Iterate through the code, printing out a string form of the
instructions *)
let print_ir fname code =
let file = open_out_bin fname
in
List.iter (fun op -> output_string file (string_of_op op); output_char
file '\n') code; close_out_noerr file

(** Convert the IR into consumable code. *)
let assemble fname code =
  let file = open_out_bin fname in
  let get_env (i, env) op =
    (op_length op + i,
     match op with
         Label s -> { env with offsets = OffsetMap.add s i env.offsets
}
       | PushAtom s -> if StringMap.mem s env.consts then env
         else
           {env with consts = StringMap.add s env.nextconst
env.consts;
            nextconst = env.nextconst + 1}
       | PushString s -> if StringMap.mem s env.consts then env
         else
           {env with consts = StringMap.add s env.nextconst
env.consts;
            nextconst = env.nextconst + 1}
       | _ -> env) in
  let (code_len, env) = List.fold_left get_env
```

```
                  (0, {offsets=OffsetMap.empty;
                       consts = StringMap.empty; nextconst = 1}) code
in
  let consts =
    let unsorted = StringMap.fold (fun k v l -> (k,v)::l) env.consts
[]
    in
    Sort.list (fun (_,v1) (_,v2) -> v1 <= v2) unsorted in
  let consts_len = (List.length consts * 4) +
    List.fold_left (fun l (k,_) -> l + String.length k) 0 consts in
  let header_len = 8 + 8 + 8 in
  let print_header () =
    (* Write the eyecatcher *)
    output_string file "OCAMLOBJ";
    (* Write the offset and length of the code *)
    output_binary_int file header_len; output_binary_int file
code_len;
    (* Write the offset and length of the constants *)
    output_binary_int file (header_len + code_len); output_binary_int
file consts_len in
  let print_opcode op = match op with
    (* These instructions don't need to be printed, so don't do
anything *)
      Label(_) -> ()
    | StartBT(_) -> ()
    | _ -> output_byte file (int_of_op op); (match op with
        PushInt(x) -> output_binary_int file x
      | PushAtom(x) -> output_binary_int file (StringMap.find x
env.consts)
      | PushString(x) -> output_binary_int file (StringMap.find x
env.consts)
      | PushRef(x) -> output_byte file x
      | Store(x) -> output_byte file x
      | StoreVal(x) -> output_byte file x
      | ClearVal(x) -> output_byte file x
      | CallOp(x) -> output_binary_int file (OffsetMap.find x
env.offsets)
      | BuiltIn(x) -> output_byte file x
      | Load(x) -> output_byte file x
      | Jnz(x) -> output_binary_int file (OffsetMap.find x
env.offsets)
      | Jz(x) -> output_binary_int file (OffsetMap.find x env.offsets)
      | Goto(x) -> output_binary_int file (OffsetMap.find x
env.offsets)
      | PushBool(x) -> output_byte file x
      | PushFrame(x) -> output_byte file x
      | StopBT x -> output_byte file x
      | _ -> ()) in
  let print_consts () =
    List.iter (fun (k,_) -> output_binary_int file (String.length k);
output_string file k) consts
  in
  print_header(); List.iter print_opcode code; print_consts();
close_out_noerr file
```