Clogic Reference Manual

Tabari Alexander
tha2105@columbia.edu
COMS W4115

1. Introduction

Clogic is loosely based on the the logic programming language Prolog. It contains many of the same principles, while at the same time adding certain restrictions and additions to the language. The main rationale behind this is to have a compiler developed in a short amount of time in addition to providing unique features not already present in Prolog.

2. 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.

2.1 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.

2.2 Identifiers

An identifier is a lowercase letter followed by a sequence of zero or more alphanumeric characters or underscores. Identifiers can represent type names, 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.

Identifiers that end 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.

2.3 Keywords

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

```
isa
type
where
when
in
while
until
```

2.4 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.

2.3 Literals

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

2.3.1 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.

2.3.2 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 compiler error. A single character may be represented by a string containing 1 character.

2.3.3 Empty list

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

3. 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***.

4. 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`.

5. 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.

6. 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.

6.1 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`.

6.2 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`.

6.3 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`.

6.4 Lists

Lists are linked lists of objects. Each object within a list can be of any type, including other lists. The empty list, `[]`, has the property of also being considered the "empty" type. Any undeclared variables or failed backtracks are given this value.

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`.

7. 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.

7.1 `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.

7.2 `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.

7.3 `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.

8. 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.

8.1 Primary expressions

All symbols used in the context of a primary expression are grouped left-to-right.

8.1.1 ***identifier***

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

8.1.2 ***literal***

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

8.1.3 ***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.

8.1.4 [***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.

8.1.5 (***expression***)

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

8.1.6 ***identifier*** (***expression-list-opt***)

A query is an identifier followed by set of parentheses containing an optional argument list of comma-separated expressions. 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. Queries can either be simply evaluated or they can be backtracked, returning an answer in a variable argument.

8.1.6.1 Evaluated queries

Queries that do not have at least one argument that is a variable bound to the empty list are evaluated according to the the algorithm below:

a. The arguments will be compared to the arguments of truths for the predicate name, in reverse order of truth declaration. If a match is found, then true will be returned.

b. The arguments will be matched to the arguments of predicate clauses for the given name, in reverse order of predicate declaration. If a match is found, then the clause will be evaluated. If the clause is evaluated to being false, the next matching clause 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.

8.1.6.2 Backtracked queries

If a query contains a variable as an argument, and the variable's value is the empty list, then the query is backtracked instead of evaluated. Backtracked queries will store a result in the variable and additionally return true if an answer was found. Otherwise, the query will store the empty list in the variable and return false. Because of this distinction, if the empty list is intended to be passed as an argument, it is best to explicitly use the literal `[]` as the argument. As a restriction, only the first variable in the argument list with this property will be treated as the return variable. The rest will be treated as if the empty list were supplied. The algorithm for backtracking is as follows:

a. The arguments will be compared to the arguments of truth declarations for the given predicate name, in reverse order of truth declaration. If a match is found for the non-variable arguments, then true will be returned and the first object seen for the truth will be returned as the variable.

b. The arguments will be matched to the arguments of predicate clauses for the given predicate name, in reverse order of predicate clause declaration. If a match is found for the non-variable arguments, then the clause will be evaluated. If the clause is evaluated to being true, and the variable argument can be resolved to a value from either a query or assignment within the clause, true will be returned. Otherwise, the next matching clause will be used until either a true result has been found or there are no more clauses to evaluate. Any side-effects caused by clause evaluation will still occur. As a means of efficiency, the compiler can choose to not attempt to evaluate a clause if it can be determined ahead of time that it will not resolve a variable.

This is best illustrated with an example. In a program containing the following predicates:

```
team(jim, john, tony);
team(jim, jason, tony);
the_odd_one(X) -> team(jim, X, tony);
```

The query `the_odd_one(jim, Y, tony)` will evaluate to true and store `jason` in `Y`.

8.1.7 ~ ***identifier*** (***expression-list***)

The undeclare expression is a primary expression which dynamically undeclares a truth, evaluating to true if it was able to do so, and false if it was not.

8.1.8 ^ ***identifier*** (***expression-list***)

The declare operation is a primary expression which dynamically declares a truth, always evaluating to true.

8.2 Unary operators

All unary operators group right-to-left. All unary operators have predicates in the form:
op (***Operand*** , ***Result***)

8.2.1 - ***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`

8.2.2 `!` ***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`.

8.3 Multiplicative operators

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

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

8.3.1 ***expression*** `*` ***expression***

The binary * operator indicates multiplication. The operation is only defined between integer types, and the resulting type after the operation is an integer. The predicate for this operation is `mult/3`.

8.3.2 ***expression*** `/` ***expression***

The binary / operator indicates division. The operation is only defined between integer types, and the resulting type after the operation is an integer. In the case where the divisor is not a factor of the dividend, only the integer part of the number will be returned. The query for this operation is `div/3`. If the divisor evaluates to zero, then an error message will be printed and the program will halt.

8.4 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*** `)`

8.4.1 ***expression*** `+` ***expression***

The binary + operator indicates addition. The operation is defined for both integers and strings. If both the operands are integers, the result will be the integer. If both operands are strings, the result will be a concatenation of the two strings. All other type combinations are undefined. The predicate for this operation is `add/3`.

8.4.2 ***expression*** `-` ***expression***

The binary – operator indicates subtraction. The operation is defined for only integers. The predicate for this operation is `sub/3`.

8.5 ***expression*** `|` ***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`.

8.5 Relation operators

The relation operators group left-to-right. All relation operators have predicates defined in the form:

op (***operand1*** , ***operand2***)
The result is the evaluation of the query.

8.5.1 ***expression*** > ***expression***

The binary > operator indicates a greater-than relation. The operator is only defined between integer types, and it will return true if the first expression is greater than the second expression. The predicate for this operation is `gt/2`.

8.5.2 ***expression*** < ***expression***

The binary < operator indicates a less-than relation. The operator is only defined between integer types, and it will return true if the first expression is less than the second expression. Otherwise the evaluation will return false. The predicate for this operation is `lt/2`.

8.5.3 ***expression*** >= ***expression***

The binary >= operator indicates a greater-than or equal relation. The operator is only defined between integer types, and it will return true if the first expression is greater-than or equal to the second expression. Otherwise, the evaluation will return false. The predicate for this operation is `ge/2`.

8.5.4 ***expression*** <= ***expression***

The binary <= operator indicates a less-than or equal relation. The operator is only defined between integer types, and it will return true if the first expression is less than or equal to the second expression. Otherwise, the evaluation will return false. The predicate for this operation is `le/2`.

8.6 Equality operators

The equality operators group left-to-right. As with the relation operators, the queries are defined in the form:
op (***operand1*** , ***operand2***)

8.6.1 ***expression*** == ***expression***

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`.

8.6.2 ***expression*** != ***expression***

The equality operator != indicates inequality. The predicate for this operation is `ne/2` and is simply defined as being `! eq/2`.

8.7 ***expression*** & ***expression***

The & operator indicates a logical and operation, which evaluates to true if both expressions evaluate to true. The operator is defined for all types, as both expressions are implicitly evaluated to boolean types using the conversion predicate `true/1`.

8.8 ***expression*** `||` ***expression***

The || operator indicates a logical or operation, which evaluates to true if either expression evaluates to true. The operator is defined for all types, as both expressions are implicitly evaluated to boolean types using the conversion predicate `true/1`.

8.9 `(` ***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.

8.10 Assignments

Assignment operators group right-to-left, and evaluate to true. The operators in this group do not have predicate equivalents.

8.10.1 ***variable*** = ***expression***

Evaluates an expression and binds the result to the variable.

8.10.2 ***variable*** `in` ***query***

If the query contains the variable in the argument list, then the query will be backtracked until all unique solutions have been found, and they will be returned in a list that will be bound by the variable. It is not an error if the query does not contain the variable in its argument list; the empty list will be returned in that case and no backtracking will be performed.

9. 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 ;***
- ***type-decl ;***
- ***isa-decl ;***
- ***when-decl***

9.1 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]`.

9.2 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 when backtracking a query.

9.3 Type declaration

A type declaration declares an atom to be a type. This has the quality that the atom cannot be returned when backtracking a query. Type declarations are written according to the syntax:

type-decl:
  `type` ***identifier***

It is an error to declare an atom as a type after it has already been used in a truth declaration.

9.4 Is-a declaration

An is-a declaration is used to easily define similar rules and truths without duplicating code. They follow the syntax:

isa-decl:
  identifier `isa` ***identifier whereclause-opt***
whereclause:
  `where` ***truth-decl-list***

Without the where clause, any truth that contains the second identifier in its literal list will be duplicated, replacing the second identifier with the first. The optional where clause can redeclare truths if the original truth is not desired. This is best explained with an example:

```
rel(a1, 1);
rel(a1, 2);
one(a1);
two(thing, a1);
```

```
a2 isa a1 where rel(a2, 0);
```

is equivalent in meaning to:

```
rel(a1, 1);
rel(a1, 2);
one(a1);
two(thing, a1);
rel(a2, 0);
one(a2);
two(thing, a2);
```

An identifier in an is-a declaration can be either a type name or an atom. It can be in multiple is-a declarations; each one will add a set of truths for the identifier.

9.5 When declaration

A when declaration defines a series of expressions to evaluate when a query evaluation changes. They follow the syntax:

when-decl:
`when` ***identifier*** `(` ***variable*** `)` ***block***
`when ~` ***identifier*** `(` ***variable*** `)` ***block***

In the first form, the block is executed when an object satisfies a backtrack query. The second form evaluates the block when the object no longer satisfies a backtrack query. The object that is returned is stored in specified variable and has scope within the following block. For simplicity, when declarations are only compatible with predicates with arity 1. Only truth declarations that are performed dynamically via the use of ~ or ^ can cause execution of the code within the block.

10 Statements

Statements are executed in sequence, except where otherwise indicated.

10.1 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.

10.2 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

10.3 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.

10.4 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***

10.5 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***

11 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

10 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.

11 Examples

All of the following examples are complete programs that are compilable in Clogic.

11.1 Hello world

```
{
     print("Hello World");
}
```

11.2 Basic example

This example shows an example of a few of the constructs in Clogic.

```
tall(jim);
heavy(jim);
footballLinebacker(Person) -> tall(Person) & heavy(Person);
{
  print(footballLinebacker(jim)); [[ prints true ]]
  print(footballLinebacker(michael)); [[ prints false ]]
  P in footballLinebacker(P);
  print(P); [[ prints jim ]]
}
```

11.3 List length

This example shows a way to obtain the length of a list.

```
length([], X) -> X = 0;
length(h | t, X) -> length(t, Y) & X = Y + 1;

{
     List = [ 1, 2, 3 ];
     length(list, L);
     print(L); [[ prints 3 ]];
}
```

11.4 Operator extension

This short example shows how operators can be extended.

```
mult(6,9,X) -> mult(7,6,X);

{
     print(6*9); [[ prints 42 ]]
     print(9*6); [[ prints 54 ]]
}
```

11.5 Loops

```
{
     I = 0
```

```
        while I < 10;
        {
                print(I);
                I = I + 1;
        }
}
```

APPENDIX I
Syntax Summary

AI.1 Operator precedence

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`
Logical And	`&`	Left-to-right	
Logical Or	`||`	Left-to-right	
Assignment	`=`	Right-to-left	
If-then-else	`?:`		

AI.2 Language summary

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***.

program:
  declaration-list-opt block

declaration-list:
  declaration
  declaration-list declaration

declaration:
  predicate-clause-decl ;
  truth-decl ;
  type-decl ;
  isa-decl ;
  when-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***

type-decl:
    `type` ***identifier***

isa-decl:
    identifier `isa` ***identifier where-opt***

where:
    `where` ***truth-decl-list***

truth-decl-list:
    truth-decl
    truth-decl-list **,** ***truth-decl***

when-decl:
    `when` ***identifier*** `(` ***variable*** `)` ***block***
    `when ~` ***identifier*** `(` ***variable*** `)` ***block***

literal:
    integer
    string
    `[]`

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

^ ***identifier*** (***expression-list***)
~ ***identifier*** (***expression-list***)

query:
identifier (***expression-list-opt***)

expression:
primary-expression
- ***expression***
! ***expression***
expression * ***expression***
expression / ***expression***
expression + ***expression***
expression - ***expression***
expression | ***expression***
expression < ***expression***
expression > ***expression***
expression <= ***expression***
expression >= ***expression***
expression == ***expression***
expression != ***expression***
expression & ***expression***
expression || ***expression***
(***expression*** ? ***expression*** : ***expression-opt***)
assignment

assignment:
variable = ***expression***
variable `in` ***query***

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
Built-in Predicate Summary

AII.1 Built-in predicates

The following section lists built-in predicates that are provided for use in any program. Assume that the predicates are defined before the optional declaration list. Definitions are in order of appearance in this section.

```
neg/2
not/2
mult/3
div/3
add/3
sub/3
cons/3
gt/2
lt/2
ge/2
le/2
eq/2
ne/2
print/1
write/1
exit/1
exit/0
true/1
true/0
false/0
integer/1
string/1
boolean/1
atom/1
list/1
tointeger/1
tostring/1
```