Degc Language

Jonathan Clark

April 11, 2013

1 Language Specification

1.1 Lexical Grammar

1.1.1 Overview

Lexical grammar is similar to the *Python* language.

The Degc tokenizer always returns the longest possible token which can be produced from the input. Tokens are delimited by non-matching characters and by whitespace. An unrecognized character or sequence is treated as an error.

In Degc, whitespace is syntactically significant. Four (4) spaces are used to indent blocks, and other whitespace characters are ignored. Spaces are also ignored inside parenthesized (,) and bracketed [,] code.

Line comments are indicated with a hash # symbol. All input from the hash sign to the end of the current line is ignored.

1.1.2 Identifiers

Types and functions are designated by a string identifier. Valid identifiers may begin with an underscore, or any upper- or lower-case English letter. Identifiers may not begin with numbers, but numbers are permitted in the remainder of the identifier.

Identifiers which begin with a number, or contain any symbols other than English letters, numbers, and underscores, are invalid.

1.1.3 Literals

Degc includes the Boolean literal values *true* and *false*.

Numeric literals denote fractional decimal numbers, and may contain any Arabic number from 0 to 9, and optionally a single decimal . which must be followed by additional numbers. Degc supports fixed-precision numbers with $\varepsilon = 0.0001$.

1.1.4 Keywords and Punctuators

The Degc lexicon contains the following keywords: *all, and, any, assert, as, best, by, either, else, embed, enum, exists, extends, for, from, function, if, import, intersect, in, limit, module, not, or, panic, program, record, setminus, set, take, union.*

The Degc lexicon also contains the following punctuators: *@ : , . -> + - * / < <= > >= = != ( ) [ ] { } |*

1.2 Syntactic Grammar

1.2.1 Overview

Degc is a declarative language for describing academic programs. As such, Degc programs are written to model the given problem, rather than written to execute specific computations.

Degc programs operate over an input *ontology* - a set of facts, which are described using records. Degc programs do not have direct access to the contents of records. Instead, Degc programs contain assertions about the ontology. A Degc program evaluates *true* when all of its assertions are true, and *false* otherwise.

1.2.2 Declarations

1.2.2.1 Records

A record is a quantum of the input ontology. Records contain relevant facts, such as courses or certifications taken, student status, etc.

A record may optionally have one *quantity* field. A record type must have a quantity field to use *take* and *limit* statements.

Example:

```
record TakenCourse:
    Faculty faculty
    Subject subject
    number level
    quantity credits
```

1.2.2.2 Enumerations

Enumerations are a list of options.

Example:

```
enum Subject:
    CMPUT
    MATH
    ENGL
    PHIL
```

1.2.2.3 Functions

Degc contains an embedded functional programming language which can be accessed by writing functions. Degc supports higher-order functions, but not lexical closures.

Example:

```
function SubjectLevelCourse(Subject sub, number lev) -> set(TakenCourse):
    { TakenCourse | subject = sub and level >= lev and level < (lev + 100) }
```

1.2.2.4 Programs

Degc programs describe statements about the ontology. Programs may inherit from other programs. Programs may be parameterized, but are restricted to enumeration arguments.

Example:

```
program ExampleProgram(Subject majorSubject):
    take 6 in { TakenCourse | subject = majorSubject }
```

1.2.3 Expressions

1.2.3.1 Overview

Deg expressions are similar to those in other languages. As is typical, function calls have the highest precedence, followed by member access, unary operators (arithmetic and boolean negation), multiplication, addition, relations, equality, logical and, and logical or. A higher precedence may be imposed with parentheses.

1.2.3.2 Logical and Arithmetic

Degc supports the usual arithmetic operators for number types: *+*,*-*,***,*/*.

Degc also supports the following logical operations: *and, or, not.*

1.2.3.3 Set

Set expressions are used to describe a portion of the ontology. They are constructed using a typical set builder notation.

Example:

```
{ TakenCourse }
```

This code describes a set of all records with the type *TakenCourse*.

```
{ TakenCourse | subject = Subject.CMPUT and level < 200 }
```

This code describes a set of all records with the type *`TakenCourse`*`, a subject of` *`CMPUT`*`, and a level which is below` 200`.`

It must be possible to represent the constructed set as a union of orthogonal ranges. As a result, set clause relations must each depend on a single record member, and that member must be isolated on a single side of the relational operator.

Sets support the *exists, union, intersect,* and *setminus* operators. The *exists* operator returns true if and only if there is at least one record matching the input set. The other operators function as expected.

1.2.3.4 Panic

Degc supports a rudimentary exception mechanism. Panic will immediately cause the executing program to terminate with a result of *false*.

1.2.4 Statements

1.2.4.1 Overview

Program statements are used to describe facts about the ontology. In the following, an asterisk (*) indicates that the statement type is planned but not currently supported by the reference implementation.

1.2.4.2 Assertions*

Assertions execute a boolean expression. If the expression evaluates as *false*, the current execution is rejected.

Example:

```
assert exists { RoyalConservatoryPiano | grade >= 6 }
```

1.2.4.3 Embed

Embed statements include, in-line, the contents of another program.

Example:

```
embed MajorProgramMap(majorSubject)
```

The embed keyword may also be used to enclose program statements in a nested block:

Example:

```
take 6 in { TakenCourse | subject = majorSubject }
embed:
    assert exists { StudentRegistered }
    take 6 in { TakenCourse | subject = minorSubject }
take 3 in { TakenCourse }
```

1.2.4.4 Disjunction

Disjunctions are used to indicate a non-discriminated branch. Program execution is successful if at least one branch may be taken.

Example:

```
either:
    assert exists { TakenCourse | subject = Subject.MUS and level = 110 }
or:
    assert exists { PianoExemptionExam | grade >= 0.8 }
or:
    assert exists { RoyalConservatoryPiano | grade >= 6 }
```

1.2.4.5 For All*

1.2.4.6 For Any*

1.2.4.7 For Best*

1.2.4.8 Conditional*

1.2.4.9 Take

Take statements are used to allocate a specified quantity from a set. Once taken, the quantity cannot be used to fulfill another *take* requirement.

For example, an academic program may require 6 credits from art courses and 3 additional credits in free options. Although art credits may be applied toward the free option requirement, art credits may not simultaneously be applied to both requirements.

```
take 6 in { TakenCourse | faculty = Faculty.AR }
take 3 in { TakenCourse }
```

In this example, the student always requires at least 9 credits in taken courses.

1.2.4.10 Limit

Limit statements are used to constrain the quantity which may be taken from a set.

For example, a typical academic program will apply a restriction on the number of introductory-level courses which may be applied to a program of study. Limit statements allow these types of requirements to be imposed while still maximizing the assignment of credits.

```
limit 42 in { TakenCourse | level < 200 }
```

Due to the algorithm used for solving programs, limit sets must either be point-wise disjoint or a sub/super-set. Compliant Degc compilers must produce an error when a program contains limit statements which do not satisfy this restriction.

2 Execution Model

2.1 Overview

TODO: Explain at a high level how the execution model works and why.

2.2 Proof of Correspondence

Note 1. The *take* and *limit* statements of a Degc program describe an integer program of a particular form.

Any requirement is of the form

Suppose, for example, we have the following simple program:

```
take 6 from { TakenCourse | faculty=Faculty.SC and subject=Subject.CMPUT }     # R1
take 6 from { TakenCourse | faculty=Faculty.SC }                               # R2
limit 3 from { TakenCourse | level<200 }                                      # L1
```

After some work, the compiler will reduce this program to some relevant pair-disjoint subsets:

```
{ TakenCourse | subject=Subject.CMPUT and faculty=Faculty.SC and level<200 }   # A1
{ TakenCourse | subject=Subject.CMPUT and faculty=Faculty.SC and level>=200 }  # A2
{ TakenCourse | subject!=Subject.CMPUT and faculty=Faculty.SC and level<200 }  # A3
{ TakenCourse | subject!=Subject.CMPUT and faculty=Faculty.SC and level>=200 } # A4
{ TakenCourse | faculty!=Faculty.SC and level<200 }                           # A5
```

Suppose the input ontology has quantities of $c_1, \ldots, c_5$ for sets $A_1, \ldots, A_5$, respectively. Then, the following integer program is equivalent:

maximize $\sum_i \alpha_i$ subject to:

$\alpha_1 + \alpha_2 \leq c_1$ # credits assigned from A_1 and A_2 to requirement R_1.

$\alpha_3 + \alpha_4 + \alpha_5 + \alpha_6 \leq c_2$ # credits assigned from A_1, A_2, A_3, and A_4 to requirement R_2.

$\alpha_5 + \alpha_6$

Theorem 2. *Any integer program of the form:*
maximize $\alpha_1 + \alpha_2 + \cdots + \alpha_n + \beta_1 + \beta_2 + \cdots + \beta_n$
subject to