Degc Language

Jonathan Clark

April 18, 2013

1 Language Specification

1.1 Lexical Grammar

111 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, with.

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

1.2 Syntactic Grammar

121 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:

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 }
```

123 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 Relational

Degc supports the following relational operators: =,!=,>,>=,<,<=.

Nominal types - enumerations and booleans - support the identity relations.

The ordinal numeric type supports all relational operators.

1.2.3.4 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.

1.2.3.5 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

1241 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 ComputingScienceMajor

Parameterized programs may also be embedded with included arguments. Example:

```
embed MajorProgramMap(majorSubject) with (minorSubject,)
```

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*

For All loops execute the statement body for each record in the matching set. This allows programs to make assertions about all records in the ontology.

For All loops may not contain take or limit statements.

Example:

```
for all course in { TakenCourse | subject = Subject.CMPUT }:
    assert course.grade >= 2.7
```

1.2.4.6 For Any*

For Any loops execute the statement body for any one, arbitrarily-chosen record in the ontology.

For Any loops may not contain take or limit statements.

Example:

```
for any status in { RegistrationStatus }:
    assert status.registered
```

1.2.4.7 For Best*

For Best loops execute the statement body for the best matching record, where the best record is identified with an order predicate.

For Best loops may not contain take or limit statements.

Example:

```
function HighestGrade(TakenCourse a, TakenCourse b)->bool:
    a.grade >= b.grade
for best course by HighestGrade in { TakenCourse }:
    assert course.grade >= 4.0
```

1.2.4.8 Conditional

Conditional statements are used to indicate a discriminated branch. Conditional statements are used to branch program execution predicated upon the value of a boolean expression. Example:

```
if exists { RoyalConservatoryPiano | grade >= 6 }:
    assert not exists { PianoExemptionExam }
else:
    assert exists { PianoExemptionExam }
```

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 }</pre>
```

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

Degc uses a hybrid execution model which combines an integer program with assertions defined using a programming language.

Due to the specific form of integer programs written in Degc, it is possible to describe the integer program portions as an instance of network max-flow. This correspondence allows a larger amount of preprocessing and a guaranteed polynomial execution time. The disadvantage to this approach is a greater demand for storage space. Since branches may produce significantly different networks, each possible branch and combination of program parameters must be enumerated into a separate network.

The accompanying bytecode program serves to prune the set of feasible networks which must be tested. Any branch which contains a failing assertion is flagged as infeasible, and it is excluded during program execution.

Programs are thus executed in three major stages:

- 1. The fully specified program is selected based on the input parameters.
- 2. The bytecode program is executed, masking impossible branches.
- 3. The max-flow is computed for each possible branch.
 - a) Credit quantities are computed from the ontology and assigned to the network inputs.
 - b) The max-flow is computed with the Edmonds-Karp algorithm.
 - c) If the max-flow is sufficient, the program has been satisfied and returns immediately.

2.2 Integer Program - Network 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

2 Execution Model

```
take 6 from { TakenCourse | faculty=Faculty.SC } # R2
limit 3 from { TakenCourse | level<200 } # L1</pre>
```

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
```

3 Implementation

3.1 Compiling Degc

3.1.1 Requirements

Degc has been built and tested using Ubuntu 12.10. This is currently the only supported platform.

Degc requires the following additional libraries and tools:

- GCC 4.7 or Clang 3.3
- Make 3.8
- Bison 2.5
- Flex 2.5
- Boost 1.49
- NullUnit 0.3 (available here: http://code.google.com/p/nullunit)

3.1.2 Compiling

Navigate to the directory where Degc has been extracted and run the following command:

\$ make

This command will automatically compile Degc and run the test cases.

3.2 Development Guide

3.2.1 Overview

This portion of the document contains a high-level description of the major namespaces and components in the Degc implementation.

The Degc compiler is a multi-pass design, translating the input text to executable code over 8 self-contained stages. This design simplifies the implementation of new features, as the behavior of each stage can be implemented and tested separately.

The recommended approach for introducing new features is to write failing test cases. The compiler will automatically generate unimplemented feature errors, letting you know what files must be modified to support your feature.

3.2.2 compiler::ast

The ast namespace contains the Abstract Syntax Tree implementation. The parser output consists of AST nodes, which encode the language in a tree structure.

- FACTORY An AST node factory, used by the parser rules to generate AST nodes.
- Node Contains declarations of AST node types.
- VISITOR An abstract visitor interface used by later compiler stages to inspect the AST.

3.2.3 compiler::diagnostics

The diagnostics namespace contains utilities and helper functions for error reporting.

- Error Container object describing a single error.
- ERRORCODE Names error codes.
- ERRORLEVEL Names error levels.
- ERRORLOCATION Container object describing the filename and position of an error.
- REPORT Abstract error report interface.
- STOREDREPORT Error reporter which stores Error objects.
- STREAMREPORT Error reporter which prints errors to a provided std::ostream&.

3.2.4 compiler::grammar

The grammar namespace contains the parser and lexical analyzer.

- INSTANCE Encapsulates an instance of the parser.
- LEXER.LEX GNU Flex lexer rules.
- PARSER.Y GNU Bison parser rules.

3.2.5 compiler::ir

The *ir* namespace contains helper classes for generating bytecode.

- PRINTER Abstract interface for a bytecode printer.
- CODEPRINTER Prints executable bytecode.
- TEXTPRINTER Prints human-readable text.
- SPLITPRINTER Prints to multiple printer implementations.

3.2.6 compiler::sg

The sg namespace contains the Semantic Graph implementation. The first compiler stages convert the Abstract Syntax Tree to the Semantic Graph data structure, which is more useful for performing semantic analysis.

- Error Helper Contains helper functions for reporting semantic errors.
- MODULE Records all of the symbols contained within a single module.
- Node Contains declarations of Semantic Graph node types.
- SCOPE_STACK Contains a stack of scopes, allowing argument look-up within nested scopes.
- SCOPE A single scope containing symbols (e.g. function arguments, record members).
- Table Semantic Graph root node.
- VISITOR An abstract visitor interface used by later compiler stages to inspect the SG.

3.2.7 compiler::stages

The *stages* namespace contains the transformation rules used for translating the input program into the bytecode and network output.

3.2.7.1 generate ast

Constructs an Abstract Syntax Tree from an input source file.

3.2.7.2 generate sg

Constructs a basic Semantic Graph from the Abstract Syntax Tree generated in the previous stage.

The basic SG contains built-in types, along with user-defined modules and symbols, although the contents of type, function, and program symbols are not yet initialized.

3.2.7.3 resolve_imports

Associates named imports with the actual symbol nodes generated in the previous stage.

3.2.7.4 generate members

Populates user-defined type symbols - records, enumerations, and function arguments.

3.2.7.5 generate expr

Generates expressions from the AST, replacing names with SG node references, and performs semantic analysis.

3.2.7.6 constant folding

Performs any possible off-line computation, converting constant expressions into constant values.

3.2.7.7 generate set expr

Normalizes set filter expressions, verifying that set filters can be expressed as a union of orthogonal ranges.

3.2.7.8 generate code

Generates networks and bytecode for programs and functions.

3.2.8 runtime::code

The *code* namespace contains data structures used to record facts about an executable program.

- CODEBUFFER Contains an executable bytecode stream.
- CODEBUFFERREADSTREAM Reads from a CodeBuffer at a specified offset.
- CODEBUFFERWRITESTREAM Appends to a CodeBuffer, used by COMPILER::IR::CODEPRINTER.
- FUNCTIONTABLE Uniquely associates function names with a CodeBuffer offset.
- PROGRAMTABLE Uniquely associates program names with a program data structure.
- RECORDTYPETABLE Uniquely associates record names with an integer identifier.

3.2.9 runtime::math

The math namespace contains utility classes used for compiling and solving programs.

- FIXED Defines a fixed-point decimal data type.
- INTERVAL Defines an interval on a data type. Used by SET.
- RELATION Defines the relations used by SET to construct intervals.
- SET Defines a set on the ontology as a union of orthogonal ranges.

3 Implementation

3.2.10 runtime::solver

The solver namespace contains utility classes used for solving programs.

- RECORD Describes the contents of a single record field.
- RECORDTABLE Abstract interface for a collection of records of a particular type.
- LINEARRECORDTABLE Implementation of RecordTable which uses linear search.
- RECORDINDEX Collection of RecordTable instances, one for each type in the ontology.
- NETWORK Network data structure and implementation of Edmonds-Karp algorithm.
- PROGRAM Program data structure.

3.2.11 runtime::vm

The vm namespace contains the bytecode virtual machine implementation.

- OPCODE Defines the opcodes which can be executed by the virtual machine.
- VIRTUALMACHINE The virtual machine implementation. Executes programs via bytecode interpretation.