Fifth Language Quick Reference

Syntax, types, keywords, and idioms (verified against learnfifth examples)

Declarations

Module import

```
import Math;
import IO;
import Net;
```

Alias (IRI namespaces)

```
alias ex as <http://example.org/>;
alias foaf as <http://xmlns.com/foaf/0.1/>;
```

Store declaration (SPARQL endpoint)

```
myStore : store = sparql_store(<http://localhost:
8080/graphdb>);
```

Variable declarations

```
x: int;
y: float;
s: string;

z: int;
z = 100;
```

Literals

Integers

```
x = 42;
x = 0b101010;
x = 0o52;
x = 0x2A;
x = 42i;
```

Floats, bool, string, null

```
pi = 3.14159;
pi = 3.14e0;

t = true;
f = false;

plain = "Hello";
raw = `Raw\nstring`;

ptr = null;
```

Types

Built-in (core + KG) numeric: int, float bool, string graph, triple, store

Arrays, lists, generics

```
arr: int[10];
dynamicArr: int[];
matrix: int[5][5];

numbers: [int];
optional: Maybe<int>;
```

Operators

Arithmetic, comparison, logical, bitwise

```
x = 10 + 5;
x = 10 - 5;
x = 10 * 5;
x = 10 / 5;
x = 10 % 3;
x = 2 ** 8;
x = 2 ^ 8;

result = 5 == 5;
result = 5 != 3;
result = 5 < 10;
result = 5 <= 5;
result = 10 > 5;
result = 10 >= 5;

result = true && false;
result = true || false;
result = !true;
result = true ~ false;

x = 5 | 3;
x = 5 & 3;
x = 5 << 2;
x = 20 >> 2;
```

Increment / compound assignment

```
x++;
x--;
++x;
--x;

x += 5;
x -= 3;
```

Functions

Declaration + call

```
add(x: int, y: int): int {
    return x + y;
}

sum = add(5, 3);
```

Constrained overloads + base case

```
positive(x: int | x > 0): int { return x * 2; }
positive(x: int): int { return 0; }

classify(x: int | x < 0): string { return
"negative"; }
classify(x: int | x == 0): string { return "zero"; }
classify(x: int | x > 0): string { return
"positive"; }
classify(x: int): string { return "unknown"; }
```

Parameter destructuring

```
class Person {
    FirstName: string;
    LastName: string;
}

greetPerson(p: Person { first: FirstName, last:
LastName }): string {
    return first;
}
```

Functional Programming

Function types

```
f: [int] -> int;
noop: [] -> int;
```

Lambda expressions (fun)

```
inc: [int] -> int;
inc = fun(x: int): int {
    return x + 1;
};
```

Higher-order function usage

```
applyTwice(f: [int] -> int, x: int): int {
    return f(f(x));
}
```

```
result = applyTwice(fun(x: int): int { return x +
1; }, 5);
```

Control Constructs

If / else, while

```
if (x > 5) {
    x = 1;
}

if (x < 5) {
    x = 0;
} else {
    x = 1;
}

while (i < 10) {
    i++;
}
```

Blocks + expression statements

```
{
    y: int;
    y = 5;
}

5 + 3;
x = 10;
x > 5;
```

Lists & Comprehensions

```
empty = [];
numbers = [1, 2, 3, 4, 5];

ys = [x from x in xs where x > 2];

first = numbers[0];
```

Classes & Objects

Classes, methods, inheritance

```
class Rectangle {
    Width: float;
    Height: float;
}

class Calculator {
    Value: int;
    Add(x: int): int { return Value + x; }
    Multiply(x: int): int { return Value * x; }
}

class Shape { Color: string; }
class Circle extends Shape { Radius: float; }
```

Instantiation + member access

```
rect = new Rectangle();
rect = new Rectangle() { Width = 10.0, Height =
5.0 };
calc = new Calculator() { Value = 100 };

w = rect.Width;
rect.Width = 15.0;
```

Knowledge Graphs

Triple literals

```
<ex:john, foaf:name, "John Doe">;
<ex:john, foaf:age, 30>;
<ex:john, ex:knows, ex:jane>;
```

Graphs + stores

```
g : graph in <ex:> = KG.CreateGraph();
g += <ex:subject1, ex:predicate1, ex:object1>;

myStore += g;
myStore -= g;
```

Semantic classes

```
class Entity in <http://example.org/ontology#> {
    Name: string;
    Value: int;
}
```

Idioms

Constrained overloads must include a base-case (unconstrained) function.
Use object initializers for readable construction.
Use list comprehensions for projection + filtering.
Graphs are mutable: add/remove with += and -=.

Keywords (reserved)

alias, as, base, break, case, catch, class, const, continue, default, defer, else, extends, export, fallthrough, finally, for, from, func, fun, go, goto, graph, if, import, in, interface, new, namespace, package, range, return, select, sparql_store, store, struct, switch, throw, try, type, use, var, when, where, while, with, true, false, null