# K

## K

## Table of Contents


- Homepage
- K Tool Binaries
- K Tutorial
  - Section 1: Basic K Concepts
    - Lesson 1.1: Setting up a K Environment
    - Lesson 1.2: Basics of Functional K
    - Lesson 1.3: BNF Syntax and Parser Generation
    - Lesson 1.4: Disambiguating Parses
    - Lesson 1.5: Modules, Imports, and Requires
    - Lesson 1.6: Integers and Booleans
    - Lesson 1.7: Side Conditions and Rule Priority
    - Lesson 1.8: Literate Programming with Markdown
    - Lesson 1.9: Unparsing and the format and color attributes
    - Lesson 1.10: Strings
    - Lesson 1.11: Casting Terms
    - Lesson 1.12: Syntactic Lists
    - Lesson 1.13: Basics of K Rewriting
    - Lesson 1.14: Defining Evaluation Order
    - Lesson 1.15: Configuration Declarations and Cell Nesting
    - Lesson 1.16: Maps, Semantic Lists, and Sets
    - Lesson 1.17: Cell Multiplicity and Cell Collections
    - Lesson 1.18: Term Equality and the Ternary Operator
    - Lesson 1.19: Debugging with GDB
    - Lesson 1.20: K Backends and the Haskell Backend
    - Lesson 1.21: Unification and Symbolic Execution
- K PL Tutorial
  - K Overview
  - Learning K
    - Part 1: Defining LAMBDA
      - Lesson 1, LAMBDA: Syntax Modules and Basic K Commands

    - Lesson 2, LAMBDA: Module Importing, Rules, Variables
    - Lesson 3, LAMBDA: Evaluation Strategies using Strictness
    - Lesson 4, LAMBDA: Generating Documentation; Latex Attributes
    - Lesson 5, LAMBDA: Adding Builtins; Side Conditions
    - Lesson 6, LAMBDA: Selective Strictness; Anonymous Variables
    - Lesson 7, LAMBDA: Derived Constructs; Extending Predefined Syntax
    - Lesson 8, LAMBDA: Multiple Binding Constructs
    - Lesson 9, LAMBDA: A Complete and Commented Definition
* Part 2: Defining IMP
    - Lesson 1, IMP: Defining a More Complex Syntax
    - Lesson 2, IMP: Defining a Configuration
    - Lesson 3, IMP: Computations, Results, Strictness; Rules Involving Cells
    - Lesson 4, IMP: Configuration Abstraction, Part 1; Types of Rules
    - Lesson 5, IMP: Completing and Documenting IMP
* Part 3: Defining LAMBDA++
    - Lesson 1, LAMBDA++: Abrupt Changes of Control
    - Lesson 2, LAMBDA++: Semantic (Non-Syntactic) Computation Items
    - Lesson 3, LAMBDA++: Reusing Existing Semantics
    - Lesson 4, LAMBDA++: Do Not Reuse Blindly!
    - Lesson 5, LAMBDA++: More Semantic Computation Items
    - Lesson 6, LAMBDA++: Wrapping Up and Documenting LAMBDA++ (environment-based)
* Part 4: Defining IMP++
    - Lesson 1, IMP++: Extending/Changing an Existing Language Syntax
    - Lesson 2, IMP++: Configuration Refinement; Freshness
    - Lesson 3, IMP++: Tagging; Superheat/Supercool Kompilation Options
    - Lesson 4, IMP++: Semantic Lists; Input/Output Streaming
    - Lesson 5, IMP++: Deleting, Saving and Restoring Cell Contents
    - Lesson 6, IMP++: Adding/Deleting Cells Dynamically; Configuration Abstraction, Part 2
    - Lesson 7, IMP++: Everything Changes: Syntax, Configuration, Semantics
    - Lesson 8, IMP++: Wrapping up Larger Languages
* Part 5: Defining Type Systems
    - Lesson 1, Type Systems: Imperative, Environment-Based Type Systems
    - Lesson 2, Type Systems: Substitution-Based Higher-Order

Type Systems
            - Lesson 3, Type Systems: Environment-Based Higher-Order Type Systems
            - Lesson 4, Type Systems: A Naive Substitution-Based Type Inferencer
            - Lesson 5, Type Systems: A Naive Environment-Based Type Inferencer
            - Lesson 6, Type Systems: Parallel Type Checkers/Inferencers
            - Lesson 7, Type Systems: A Naive Substitution-based Polymorphic Type Inferencer
            - Lesson 8, Type Systems: A Naive Environment-based Polymorphic Type Inferencer
            - Lesson 9, Type Systems: Let-Polymorphic Type Inferencer (Damas-Hindley-Milner)
    - Learning Language Design and Semantics using K
        * Part 7: SIMPLE: Designing Imperative Programming Languages
            - Lesson 1, SIMPLE untyped
            - Lesson 2, SIMPLE typed static
            - Lesson 3, SIMPLE typed dynamic
        * Part 8: KOOL: Designing Object-Oriented Programming Languages
            - Lesson 1, KOOL untyped
            - Lesson 2, KOOL typed dynamic
            - Lesson 3, KOOL typed static
        * Part 9: FUN: Designing Functional Programming Languages
            - Lesson 1, FUN untyped, Environment-Based
            - Lesson 2, FUN untyped, Substitution-Based
            - Lesson 3, FUN polymorphic type inferencer
        * Part 10: LOGIK: Designing Logic Programming Languages
            - Lesson 1, LOGIK
- K User Manual
- K Builtins
    - domains
    - kast
    - prelude
    - ffi
    - json
    - rat
    - substitution
    - unification
- Projects using K

**K** is a rewrite-based
executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using configurations and rules. Configurations organize the state in units called cells,

which are labeled and can be nested. K rewrite rules make it explicit which parts of the term are read-only, write-only, read-write, or unused. This makes K suitable for defining truly concurrent languages even in the presence of sharing. Computations are represented as syntactic extensions of the original language abstract syntax, using a nested list structure which sequentializes computational tasks, such as program fragments. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes K suitable for defining control-intensive features such as abrupt termination, exceptions, or call/cc.

## Overview

- A ten-minute overview video slide presentation.
- A ninety-minute tutorial video, given at ETAPS'16.
- [Optional] A high-level interview about rewrite-based semantics (Wolfram Schulte interviews Grigore Rosu at ICSE'11.
- FAQ

## K Tool Download

- The provided K Tool Binaries are supported on Linux, OS X, and Windows. Other platforms may or may not work correctly. We welcome information about usability of unsupported platforms or bugs in the supported platforms.
- Try our Editor Support page for links to K syntax highlighting definitions for various popular editors/IDEs. Please feel free to contribute.
- The source code (Java) is available on GitHub, where you can also report bugs (please do so).

## Learn K

- Do the K Tutorial!
- Read some papers about K on the Formal Systems Laboratory (FSL).
- User documentation
- Builtins

## Links

- K webpage at UAIC (Romania).
- Matching logic webpage at UIUC (USA).
- Online K Discussion Channel for K users (Slack & Riot). This is the recommended way to ask questions about K and interact with the K community.

- Stackoverflow for general questions to the K user community (use the channel above if you want quick answers).

## K Tool Binaries

Download the latest stable release or prior releases.

## K Tutorial

The purpose of this series of lessons is to teach developers how to program in K. While the primary use of K is in the specification of operational semantics of programming languages, this tutorial is agnostic on how the knowledge of K is used. For a more detailed tutorial explaining the basic principles of programming language design, refer to the
K PL Tutorial. Note that that tutorial is somewhat
out of date presently.

This K tutorial is a work in progress. Many lessons are currently simply placeholders for future content.

To start the K tutorial, begin with
Section 1: Basic Programming in K.

## Section 1: Basic K Concepts

The goal of this first section of the K tutorial is to teach the basic principles of K to someone with no prior experience with K as a programming language. However, this is not written with the intended audience of someone who is a complete beginner to programming. We are assuming that the reader has a firm grounding in computer science broadly, as well as that they have experience writing code in functional programming languages before.

By the end of this section, the reader ought to be able to write specifications of simple languages in K, use these specifications to generate a fast interpreter for their programming language, as well as write basic deductive program verification proofs over programs in their language. This should give them the theoretical grounding they need to begin expanding their knowledge of K in Section 2: Intermediate K Concepts.

To begin this section, refer to
Lesson 1.1: Setting up a K Environment.

# Lesson 1.1: Setting up a K Environment

The first step to learning K is to install K on your system, and configure your editor for K development.

## Installing K

You have two options for how to install K, depending on how you intend to interact with the K codebase. If you are solely a user of K, and have no interest in developing or making changes to K, you most likely will want to install one of our binary releases of K. However, if you are going to be a K developer, or simply want to build K from source, you should follow the instructions for a source build of K.

### Installing K from a binary release

K is developed as a rolling release, with each change to K that passes our CI infrastructure being deployed on GitHub for download. The latest release of K can be downloaded here.
This page also contains information on how to install K. It is recommended that you fully uninstall the old version of K prior to installing the new one, as K does not maintain entries in package manager databases, with the exception of Homebrew on MacOS.

### Installing K from source

You can clone K from GitHub with the following Git command:

```
git clone https://github.com/runtimeverification/k --recursive
```

Instructions on how to build K from source can be found here.

## Configuring your editor

K maintains a set of scripts for a variety of text editors, including vim and emacs, in various states of maintenance. You can download these scripts with the following Git command:

```
git clone https://github.com/kframework/k-editor-support
```

Because K allows users to define their own grammars for parsing K itself, not all features of K can be effectively highlighted. However, at the cost of occasionally highlighting things incorrectly, you can get some pretty good results in many cases. With that being said, some of the editor scripts in the above repository are pretty out of date. If you manage to improve them, we welcome pull requests into the repository.

### Troubleshooting

If you have problems installing K, we encourage you to reach out to us. If you follow the above install instructions and run into a problem, you can
Create a bug report on GitHub

### Next lesson

Once you have set up K on your system to your satisfaction, you can continue to
Lesson 1.2: Basics of Functional K.

## Lesson 1.2: Basics of Functional K

The purpose of this lesson is to explain the basics of **productions** and **rules** in K. These are two types of K **sentences**. A K file consists of one or more **requires** or **modules** in K. Each module consists of one or more **imports** or sentences. For more information on requires, modules, and sentences, refer to Lesson 1.4. However, for the time being, just think of a module as a container for sentences, and don't worry about requires or imports just yet.

### Our first K program

To start with, input the following program into your editor as file `lesson-02-a.k`:

```
module LESSON-02-A

  syntax Color ::= Yellow() | Blue()
  syntax Fruit ::= Banana() | Blueberry()
  syntax Color ::= colorOf(Fruit) [function]

  rule colorOf(Banana()) => Yellow()
  rule colorOf(Blueberry()) => Blue()

endmodule
```

Save this file and then run:

```
kompile lesson-02-a.k
```

`kompile` is K's compiler. By default, it takes a program or specification written in K and compiles it into an interpreter for that input. Right now we are compiling a single file. A set of K files that are compiled together are called a **K definition**. We will cover multiple file K definitions later on. `kompile` will output a directory containing everything needed to execute programs and perform proofs using that definition. In this case, `kompile` will

(by default) create the directory `lesson-02-a-kompiled` under the current directory.

Now, save the following input file in your editor as `banana.color` in the same directory as `lesson-02-a.k`:

```
colorOf(Banana())
```

We can now evaluate this K term by running (from the same directory):

```
krun banana.color
```

`krun` will use the interpreter generated by the first call to `kompile` to execute this program.

You will get the following output:

```
<k>
  Yellow ( ) ~> .
</k>
```

For now, don't worry about the `<k>`, `</k>`, or `~> .` portions of this output file.

You can also execute small programs directly by specifying them on the command
line instead of putting them in a file. For example, the same program above could also have been executed by running the following command:

```
krun -cPGM='colorOf(Banana())'
```

Now, let's look at what this definition and program did.

## Productions, Constructors, and Functions

The first thing to realize is that this K definition contains 5 productions. Productions are introduced with the **syntax** keyword, followed by a **sort**, followed by the operator `::=` followed by the definition of one or more productions themselves, separated by the `|` operator. There are different types of productions, but for now we only care about **constructors** and **functions**. Each declaration separated by the `|` operator is individually a single production, and the `|` symbol simply groups together productions that have the same sort. For example, we could equally have written an identical K definition like so:

```
module LESSON-02-B

  syntax Color ::= Yellow()
  syntax Color ::= Blue()
  syntax Fruit ::= Banana()
  syntax Fruit ::= Blueberry()
  syntax Color ::= colorOf(Fruit) [function]
```

```
  rule colorOf(Banana()) => Yellow()
  rule colorOf(Blueberry()) => Blue()

endmodule
```

Or even:

```
module LESSON-02-C

  syntax Color ::= Yellow()
                 | Blue()
                 | colorOf(Fruit) [function]
  syntax Fruit ::= Banana()
                 | Blueberry()

  rule colorOf(Banana()) => Yellow()
  rule colorOf(Blueberry()) => Blue()

endmodule
```

Each of these types of productions named above has the same underlying syntax, but context and **attributes** are used to distinguish between the different types. Tokens, brackets, lists, macros, aliases, and anywhere productions will be covered in a later lesson, but this lesson does introduce us to constructors and functions. `Yellow()`, `Blue()`, `Banana()`, and `Blueberry()` are constructors. You can think of a constructor like a constructor for an algebraic data type, if you're familiar with a functional language. The data type itself is the sort that appears on the left of the `::=` operator. Sorts in K consist of uppercase identifiers.

Constructors can have arguments, but these ones do not. We will cover the syntax of productions in detail in the next lesson, but for now, you can write a production with no arguments as an uppercase or lowercase identifier followed by the `()` operator.

A function is distinguished from a constructor by the presence of the **function** attribute. Attributes appear in a comma separated list between square brackets after any sentence, including both productions and rules. Various attributes with built-in meanings exist in K and will be discussed throughout the tutorial.

### Exercise

Use `krun` to compute the return value of the `colorOf` function on a `Blueberry()`.

## Rules, Matching, and Variables

Functions in K are given definitions using rules. A rule begins with the `rule` keyword and contains at least one **rewrite operator**. The rewrite operator is represented by the syntax `=>`. The rewrite operator is one of the built-in productions in K, and we will discuss in more detail how it can be used in future lessons, but for now, you can think of a rule as consisting of a **left-hand side** and a **right-hand side**, separated by the rewrite operator. On the left-hand side is the name of the function and zero or more **patterns** corresponding to the parameters of the function. On the right-hand side is another pattern. The meaning of the rule is relatively simple, having defined these components. If the function is called with arguments that **match** the patterns on the left-hand side, then the return value of the function is the pattern on the right-hand side.

For example, in the above example, if the argument of the `colorOf` function is `Banana()`, then the return value of the function is `Yellow()`.

So far we have introduced that a constructor is a type of pattern in K. We will introduce more complex patterns in later lessons, but there is one other type of basic pattern: the **variable**. A variable, syntactically, consists of an uppercase identifier. However, unlike a constructor, a variable will match **any** pattern with one exception: Two variables with the same name must match the same pattern.

Here is a more complex example (`lesson-02-d.k`):

```
module LESSON-02-D

  syntax Container ::= Jar(Fruit)
  syntax Fruit ::= Apple() | Pear()

  syntax Fruit ::= contentsOfJar(Container) [function]

  rule contentsOfJar(Jar(F)) => F

endmodule
```

Here we see that `Jar` is a constructor with a single argument. You can write a production with multiple arguments by putting the sorts of the arguments in a comma-separated list inside the parentheses.

In this example, `F` is a variable. It will match either `Apple()` or `Pear()`. The return value of the function is created by **substituting** the matched values of all of the variables into the variables on the right-hand side of the rule.

To demonstrate, compile this definition and execute the following program with krun:

```
contentsOfJar(Jar(Apple()))
```

You will see when you run it that the program returns `Apple()`, because that is the pattern that was matched by `F`.

### Exercises

1. Extend the definition in `lesson-02-a.k` with the addition of blackberries and kiwis. For simplicity, blackberries are black and kiwis are green. Then compile your definition and test that your additional fruits are correctly handled by the `colorOf` function.
2. Create a new definition which defines an outfit as a multi-argument constructor consisting of a hat, shirt, pants, and shoes. Define a new sort, `Boolean`, with two constructors, true and false. Each of hat, shirt, pants, and shoes will have a single argument (a color), either black or white. Then define an `outfitMatching` function that will return true if all
the pieces of the outfit are the same color. You do not need to define the case that returns false. Write some tests that your function behaves the way
you expect.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.3: BNF Syntax and Parser Generation.

## Lesson 1.3: BNF Syntax and Parser Generation

The purpose of this lesson is to explain the full syntax and semantics of **productions** in K as well as how productions and other syntactic **sentences** can be used to define grammars for use parsing both rules as well as programs.

### K's approach to parsing

K's grammar is divided into two components: the **outer syntax** of K and the **inner syntax** of K. Outer syntax refers to the parsing of **requires**, **modules**, **imports**, and **sentences** in a K definition. Inner syntax refers to the parsing of **rules** and **programs**. Unlike the outer syntax of K, which is predetermined, much of the inner syntax of K is defined by you, the developer. When rules or programs are parsed, they are parsed within the context of a module. Rules are parsed in the context of the module in which they exist, whereas programs are parsed in the context of the **main syntax module** of a K definition. The productions and other syntactic

sentences in a module are used to construct the grammar of the module, which is then used to perform parsing.

## Basic BNF productions

To illustrate how this works, we will consider a simple K definition which defines a relatively basic calculator capable of evaluating Boolean expressions containing and, or, not, and xor.

Input the following program into your editor as file `lesson-03-a.k`:

```
module LESSON-03-A

  syntax Boolean ::= "true" | "false"
                   | "!" Boolean [function]
                   | Boolean "&&" Boolean [function]
                   | Boolean "^" Boolean [function]
                   | Boolean "||" Boolean [function]

endmodule
```

You will notice that the productions in this file look a little different than the ones from the previous lesson. In point of fact, K has two different mechanisms for defining productions. We have previously been focused exclusively on the first mechanism, where the `::=` symbol is followed by an alphanumeric identifier followed by a comma-separated list of sorts in parentheses. However, this is merely a special case of a more generic mechanism for defining the syntax of productions using a variant of
BNF Form.

For example, in the previous lesson, we had the following set of productions:

```
module LESSON-03-B
  syntax Color ::= Yellow() | Blue()
  syntax Fruit ::= Banana() | Blueberry()
  syntax Color ::= colorOf(Fruit) [function]
endmodule
```

It turns out that this is equivalent to the following definition which defines the same grammar, but using BNF notation:

```
module LESSON-03-C
  syntax Color ::= "Yellow" "(" ")" | "Blue" "(" ")"
  syntax Fruit ::= "Banana" "(" ")" | "Blueberrry" "(" ")"
  syntax Color ::= "colorOf" "(" Fruit ")" [function]
endmodule
```

In this example, the sorts of the argument to the function are unchanged, but everything else has been wrapped in double quotation marks. This is because in BNF notation, we distinguish between two types of **production items**:

**terminals** and **non-terminals**. A terminal represents simply a literal string of characters that is verbatim part of the syntax of that production. A non-terminal, conversely, represents a sort name, where the syntax of that production accepts any valid term of that sort at that position.

This is why, when we wrote the program `colorOf(Banana())`, `krun` was able to execute that program: because it represented a term of sort `Color` that was parsed and interpreted by K's interpreter. In other words, `krun` parses and interprets terms according to the grammar defined by the developer. It is automatically converted into an AST of that term, and then the `colorOf` function is evaluated using the function rules provided in the definition.

Bringing us back to the file `lesson-03-a.k`, we can see that this grammar has given a simple BNF grammar for expressions over Booleans. We have defined constructors corresponding to the Boolean values true and false, and functions corresponding to the Boolean operators for and, or, not, and xor. We have also given a syntax for each of these functions based on their syntax in the `C` programming language. As such, we can now write programs in the simple language
we have defined.

Input the following program into your editor as `and.bool` in the same directory:

```
true && false
```

We cannot interpret this program yet, because we have not given rules defining the meaning of the `&&` function yet, but we can parse it. To do this, you can run (from the same directory):

```
kast --output kore and.bool
```

`kast` is K's just-in-time parser. It will generate a grammar from your K definition on the fly and use it to parse the program passed on the command line. The `--output` flag controls how the resulting AST is represented; don't worry about the possible values yet, just use `kore`.

You ought to get the following AST printed on standard output, minus the formatting:

```
inj{SortBoolean{}, SortKItem{}}(
  Lbl'UndsAnd-And-UndsUnds'LESSON-03-A'Unds'Boolean'Unds'Boolean'Unds'Boolean{}(
    Lbltrue'Unds'LESSON-03-A'Unds'Boolean{}(),
    Lblfalse'Unds'LESSON-03-A'Unds'Boolean{}()
  )
)
```

Don't worry about what exactly this means yet, just understand that it represents the AST of the program that you just parsed. You ought to be able to recognize the basic shape of it by seeing the words `true`, `false`, and

`And` in there. This is **Kore**, the intermediate representation of K, and we will cover it in detail later.

Note that you can also tell `kast` to print the AST in other formats. For a more direct representation of the original K, while still maintaining the structure of an AST, you can say `kast --output kast and.bool`. This will yield the following output:

```
`_&&__LESSON-03-A_Boolean_Boolean_Boolean`(
  `true_LESSON-03-A_Boolean`(.KList),
  `false_LESSON-03-A_Boolean`(.KList)
)
```

Note how the first output is largely a name-mangled version of the second output. The one difference is the presence of the `inj` symbol in the KORE output. We will talk more about this in later lessons.

**Exercise**

Parse the expression `false || true` with `--output kast`. See if you can predict approximately what the corresponding output would be with `--output kore`, then run the command yourself and compare it to your prediction.

## Ambiguities

Now let's try a slightly more advanced example. Input the following program into your editor as `and-or.bool`:

```
true && false || false
```

When you try and parse this program, you ought to see the following error:

```
[Error] Inner Parser: Parsing ambiguity.
1: syntax Boolean ::= Boolean "||" Boolean [function]

`_||__LESSON-03-A_Boolean_Boolean_Boolean`(`_&&__LESSON-03-A_Boolean_Boolean_Boolean`(`true_
2: syntax Boolean ::= Boolean "&&" Boolean [function]

`_&&__LESSON-03-A_Boolean_Boolean_Boolean`(`true_LESSON-03-A_Boolean`(.KList),`_||__LESSON-0
        Source(./and-or.bool)
        Location(1,1,1,23)
```

This error is saying that `kast` was unable to parse this program because it is ambiguous. K's just-in-time parser is a GLL parser, which means it can handle the full generality of context-free grammars, including those grammars which are ambiguous. An ambiguous grammar is one where the same string can be parsed
as multiple distinct ASTs. In this example, it can't decide whether it should

be parsed as `(true && false) || false` or as `true && (false || false)`. As a
result, it reports the error to the user.

## Brackets

Currently there is no way of resolving this ambiguity, making it impossible to write complex expressions in this language. This is obviously a problem. The standard solution in most programming languages to this problem is to use parentheses to indicate the appropriate grouping. K generalizes this notion into a type of production called a **bracket**. A bracket production in K is any production with the `bracket` attribute. It is required that such a production only have a single non-terminal, and the sort of the production must equal the sort of that non-terminal. However, K does not otherwise impose restrictions on the grammar the user provides for a bracket. With that being said, the most common type of bracket is one in which a non-terminal is surrounded by terminals representing some type of bracket such as `()`, `[]`, `{}`, `<>`, etc. For example, we can define the most common type of bracket, the type used by the vast majority of programming languages, quite simply.

Consider the following modified definition, which we will save to `lesson-03-d.k`:

```
module LESSON-03-D

  syntax Boolean ::= "true" | "false"
                   | "(" Boolean ")" [bracket]
                   | "!" Boolean [function]
                   | Boolean "&&" Boolean [function]
                   | Boolean "^" Boolean [function]
                   | Boolean "||" Boolean [function]

endmodule
```

In this definition, if the user does not explicitly define parentheses, the grammar remains ambiguous and K's just-in-time parser will report an error. However, you are now able to parse more complex programs by means of explicitly
grouping subterms with the bracket we have just defined.

Consider `and-or-left.bool`:

```
(true && false) || false
```

Now consider `and-or-right.bool`:

```
true && (false || false)
```

If you parse these programs with `kast`, you will once again get a single unique AST with no error. If you look, you might notice that the bracket itself does not appear in the AST. In fact, this is a property unique to brackets: productions with the bracket attribute are not represented in the parsed AST of a term, and the child of the bracket is folded immediately into the parent term. This is the reason for the requirement that a bracket production have a single non-terminal of the same sort as the production itself.

**Exercise**

Write out what you expect the AST to be arising from parsing these two programs
above with `--output kast`, then parse them yourself and compare them to the AST you expected. Confirm for yourself that the bracket production does not appear in the AST.

## Tokens

So far we have seen how we can define the grammar of a language. However, the grammar is not the only relevant part of parsing a language. Also relevant is the lexical syntax of the language. Thus far, we have implicitly been using K's automatic lexer generation to generate a token in the scanner for each terminal in our grammar. However, sometimes we wish to define more complex lexical syntax. For example, consider the case of integers in C: an integer consists of a decimal, octal, or hexadecimal number followed by an optional suffix indicating the type of the literal.

In theory it would be possible to define this syntax via a grammar, but not only would it be cumbersome and tedious, you would also then have to deal with
an AST generated for the literal which is not convenient to work with.

Instead of doing this, K allows you to define **token** productions, where a production consists of a regular expression followed by the `token` attribute, and the resulting AST consists of a typed string containing the value recognized by the regular expression.

For example, the builtin integers in K are defined using the following production:

```
syntax Int ::= r"[\\+-]?[0-9]+" [token]
```

Here we can see that we have defined that an integer is an optional sign followed by a nonzero sequence of digits. The `r` preceding the terminal indicates that what appears inside the double quotes is a regular expression, and the `token` attribute indicates that terms which parse as this production should be converted into a token by the parser.

It is also possible to define tokens that do not use regular expressions. This can be useful when you wish to declare particular identifiers for use in your semantics later. For example:

```
syntax Id ::= "main" [token]
```

Here, we declare that `main` is a token of sort `Id`. Instead of being parsed as a symbol, it gets parsed as a token, generating a typed string in the AST. This is useful in a semantics of C because the parser generally does not treat the `main` function in C specially; only the semantics treats it specially.

Of course, languages can have more complex lexical syntax. For example, if we wish to define the syntax of integers in C, we could use the following production:

```
syntax IntConstant ::= r"(([1-9][0-9]*)|(0[0-7]*)|(0[xX][0-9a-fA-F]+))(([uU][lL]?)|([uU]((ll
```

As you may have noted above, long and complex regular expressions can be hard to read. They also suffer from the problem that unlike a grammar, they are not particularly modular.

We can get around this restriction by declaring explicit regular expressions, giving them a name, and then referring to them in productions.

Consider the following (equivalent) way to define the lexical syntax of integers in C:

```
syntax IntConstant ::= r"({DecConstant}|{OctConstant}|{HexConstant})({IntSuffix}?)" [token]
syntax lexical DecConstant = r"{NonzeroDigit}({Digit}*)"
syntax lexical OctConstant = r"0({OctDigit}*)"
syntax lexical HexConstant = r"{HexPrefix}({HexDigit}+)"
syntax lexical HexPrefix = r"0x|0X"
syntax lexical NonzeroDigit = r"[1-9]"
syntax lexical Digit = r"[0-9]"
syntax lexical OctDigit = r"[0-7]"
syntax lexical HexDigit = r"[0-9a-fA-F]"
syntax lexical IntSuffix = r"{UnsignedSuffix}({LongSuffix}?)|{UnsignedSuffix}{LongLongSuffix
syntax lexical UnsignedSuffix = r"[uU]"
syntax lexical LongSuffix = r"[lL]"
syntax lexical LongLongSuffix = r"ll|LL"
```

As you can see, this is rather more verbose, but it has the benefit of both being much easier to read and understand, and also increased modularity. Note that we refer to a named regular expression by putting the name in curly brackets. Note also that only the first sentence actually declares a new piece of syntax in the language. When the user writes `syntax lexical`, they are only declaring a regular expression. To declare an actual piece of syntax in the grammar, you still must actually declare an explicit token production.

One final note: K uses Flex to implement its lexical analysis. As a result, you can refer to the

Flex Manual
for a detailed description of the regular expression syntax supported. Note that for performance reasons, Flex's regular expressions are actually a regular language, and thus lack some of the syntactic convenience of modern "regular expression" libraries. If you need features that are not part of the syntax of Flex regular expressions, you are encouraged to express them via a grammar instead.

## Ahead-of-time parser generation

So far we have been entirely focused on K's support for just-in-time parsing, where the parser is generated on the fly prior to being used. This benefits from being faster to generate the parser, but it suffers in performance if you have to repeatedly parse strings with the same parser. For this reason, it is generally encouraged that when parsing programs, you use K's ahead-of-time parser generation. K makes use of
GNU Bison to generate parsers.

By default, you can enable ahead-of-time parsing via the `--gen-bison-parser` flag to `kompile`. This will make use of Bison's LR(1) parser generator. As such, if your grammar is not LR(1), it may not parse exactly the same as if you were to use the just-in-time parser, because Bison will automatically pick one of the possible branches whenever it encounters a shift-reduce or reduce-reduce conflict. In this case, you can either modify your grammar to be LR(1), or you can enable use of Bison's GLR support by instead passing `--gen-glr-bison-parser` to `kompile`. Note that if your grammar is ambiguous,
the ahead-of-time parser will not provide you with particularly readable error messages at this time.

If you have a K definition named `foo.k`, and it generates a directory when you run `kompile` called `foo-kompiled`, you can invoke the ahead-of-time parser you generated by running `foo-kompiled/parser_PGM <file>` on a file.

## Exercises

1. Compile `lesson-03-d.k` with ahead-of-time parsing enabled. Then compare
   how long it takes to run `kast --output kore and-or-left.bool` with how long it
   takes to run `lesson-03-d-kompiled/parser_PGM and-or-left.bool`. Confirm for
   yourself that both produce the same result, but that the latter is faster.
2. Define a simple grammar consisting of integers, brackets, addition, subtraction, multiplication, division, and unary negation. Integers should be

in decimal form and lexically without a sign, whereas negative numbers can be
represented via unary negation. Ensure that you are able to parse some basic
arithmetic expressions using a generated ahead-of-time parser. Do not worry
about disambiguating the grammar or about writing rules to implement the
operations in this definition.

3. Write a program where the meaning of the arithmetic expression based on the grammar you defined above is ambiguous, and then write programs that
express each individual intended meaning using brackets.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.4: Disambiguating Parses.

## Lesson 1.4: Disambiguating Parses

The purpose of this lesson is to teach how to use K's builtin features for disambiguation to transform an ambiguous grammar into an unambiguous one that
expresses the intended ASTs.

### Priority blocks

In practice, very few formal languages outside the domain of natural language processing are ambiguous. The main reason for this is that parsing unambiguous languages is asymptotically faster than parsing ambiguous languages. Programming language designers instead usually use the notions of operator precedence and associativity to make expression grammars unambiguous. These mechanisms work by instructing the parser to reject certain ASTs in favor of others in case of ambiguities; it is often possible to remove *all* ambiguities in a grammar with these techniques.

While it is sometimes possible to explicitly rewrite the grammar to remove these parses, because K's grammar specification and AST generation are inextricably linked, this is generally discouraged. Instead, we use the approach of explicitly expressing the relative precedence of different operators in different situations in order to resolve the ambiguity.

For example, in C, `&&` binds tighter in precedence than `||`, meaning that the expression `true && false || false` has only one valid AST: `(true && false) || false`.

Consider, then, the third iteration on the grammar of this definition (`lesson-04-a.k`):

```
module LESSON-04-A

  syntax Boolean ::= "true" | "false"
                   | "(" Boolean ")" [bracket]
                   > "!" Boolean [function]
                   > Boolean "&&" Boolean [function]
                   > Boolean "^" Boolean [function]
                   > Boolean "||" Boolean [function]

endmodule
```

In this example, some of the `|` symbols separating productions in a single block have been replaced with `>`. This serves to describe the **priority groups** associated with this block of productions.

In this example, the first priority group consists of the atoms of the language: `true`, `false`, and the bracket operator. In general, a priority group starts either at the `::=` or `>` operator and extends until either the next `>` operator or the end of the production block. Thus, we can see that the second, third, fourth, and fifth priority groups in this grammar all consist of a single production.

The meaning of these priority groups becomes apparent when parsing programs: A symbol with a **lesser priority**, (i.e., one that **binds looser**), cannot appear as the **direct child** of a symbol with a **greater priority** (i.e., one that **binds tighter**. In this case, the `>` operator can be seen as a **greater-than** operator describing a transitive partial ordering on the productions in the production block, expressing their relative priority.

To see this more concretely, let's look again at the program `true && false || false`. As noted before, previously this program was ambiguous because the parser could either choose that `&&` was the child of `||` or vice versa. However, because a symbol with lesser priority (i.e., `||`) cannot appear as the direct child of a symbol with greater priority (i.e., `&&`), the parser will **reject** the parse where `||` is under the `&&` operator. As a result, we are left with the unambiguous parse `(true && false) || false`. Similarly, `true || false && false` parses unambiguously as `true || (false && false)`. Conversely, if the user explicitly
wants the other parse, they can express this using brackets by explicitly writing `true && (false || false)`. This still parses successfully because the `||` operator is no longer the **direct** child of the `&&` operator, but is instead the direct child of the `()` operator, and the `&&` operator is an **indirect** parent, which is not subject to the priority restriction.

Astute readers, however, will already have noticed what seems to be a

contradiction: we have defined `()` as also having greater priority than `||`. One would think that this should mean that `||` cannot appear as a direct child of `()`. This is a problem because priority groups are applied to every possible parse separately. That is to say, even if the term is unambiguous prior to this disambiguation rule, we still reject that parse if it violates the rule of priority.

In fact, however, we do not reject this program as a parse error. Why is that? Well, the rule for priority is slightly more complex than previously described. In actual fact, it applies only conditionally. Specifically, it applies in cases where the child is either the first or last production item in the parent's production. For example, in the production `Bool "&&" Bool`, the first `Bool` non-terminal is not preceded by any terminals, and the last `Bool` non-terminal is not followed by any terminals. As a result of this, we apply the priority rule to both children of `&&`. However, in the `()` operator, the sole non-terminal is both preceded by and followed by terminals. As a result, the priority rule is not applied when `()` is the parent. Because of this, the program we mentioned above successfully parses.

**Exercise**

Parse the program `true && false || false` using kast, and confirm that the AST
places `||` as the top level symbol. Then modify the definition so that you will get the alternative parse.

## Associativity

Even having broken the expression grammar into priority blocks, the resulting grammar is still ambiguous. We can see this if we try to parse the following program (`assoc.bool`):

```
true && false && false
```

Priority blocks will not help us here: the problem comes between two parses where both possible parses have a direct parent and child which is within a single priority block (in this case, `&&` is in the same block as itself).

This is where the notion of associativity comes into play. Associativity applies the following additional rules to parses:

- a left-associative symbol cannot appear as a direct rightmost child of a symbol with equal priority;
- a right-associative symbol cannot appear as a direct leftmost child of a symbol with equal priority; and
- a non-associative symbol cannot appear as a direct leftmost **or** rightmost child of a symbol with equal priority.

In C, binary operators are all left-associative, meaning that the expression `true && false && false` parses unambiguously as `(true && false) && false`,
because `&&` cannot appear as the rightmost child of itself.

Consider, then, the fourth iteration on the grammar of this definition (`lesson-04-b.k`):

```
module LESSON-04-B

  syntax Boolean ::= "true" | "false"
                   | "(" Boolean ")" [bracket]
                   > "!" Boolean [function]
                   > left: Boolean "&&" Boolean [function]
                   > left: Boolean "^" Boolean [function]
                   > left: Boolean "||" Boolean [function]

endmodule
```

Here each priority group, immediately after the `::=` or `>` operator, can be followed by a symbol representing the associativity of that priority group: either `left:` for left associativity, `right:` for right associativity, or `non-assoc:` for non-associativity. In this example, each priority group we apply associativity to has only a single production, but we could equally well write a priority block with multiple productions and an associativity.

For example, consider the following, different grammar (`lesson-04-c.k`):

```
module LESSON-04-C

  syntax Boolean ::= "true" | "false"
                   | "(" Boolean ")" [bracket]
                   > "!" Boolean [function]
                   > left:
                     Boolean "&&" Boolean [function]
                   | Boolean "^" Boolean [function]
                   | Boolean "||" Boolean [function]

endmodule
```

In this example, unlike the one above, `&&`, `^`, and `||` have the same priority. However, viewed as a group, the entire group is left associative. This means that none of `&&`, `^`, and `||` can appear as the right child of any of `&&`, `^`, or `||`. As a result of this, this grammar is also not ambiguous. However, it expresses a different grammar, and you are encouraged to think about what the differences are in practice.

**Exercise**

Parse the program `true && false && false` yourself, and confirm that the AST
places the rightmost `&&` at the top of the expression. Then modify the definition to generate the alternative parse.

## Explicit priority and associativity declarations

Previously we have only considered the case where all of the productions which you wish to express a priority or associativity relation over are co-located in the same block of productions. However, in practice this is not always feasible or desirable, especially as a definition grows in size across multiple modules.

As a result of this, K provides a second way of declaring priority and associativity relations.

Consider the following grammar, which we will name `lesson-04-d.k` and which will express the exact same grammar as `lesson-04-b.k`

```
module LESSON-04-D

  syntax Boolean ::= "true" [literal] | "false" [literal]
                   | "(" Boolean ")" [atom, bracket]
                   | "!" Boolean [not, function]
                   | Boolean "&&" Boolean [and, function]
                   | Boolean "^" Boolean [xor, function]
                   | Boolean "|" Boolean [or, function]

  syntax priorities literal atom > not > and > xor > or
  syntax left and
  syntax left xor
  syntax left or
endmodule
```

This introduces a couple of new features of K. First of all, we see a bunch of attributes we don't already recognize. These are actually not built-in attributes, but rather user-defined attributes that are used to group productions together conceptually. For example, `literal` in the `syntax priorities` sentence is used to refer to the productions with the `literal` attribute, i.e., `true` and `false`.

Once we understand this, it becomes relatively straightforward to understand the meaning of this grammar. Each `syntax priorities` sentence defines a priority relation where each `>` separates a priority group containing all the productions with at least one of the attributes in that group, and each `syntax left`, `syntax right`, or `syntax non-assoc` sentence defines an associativity relation connecting all the productions with one of the target

attributes together into a left-, right-, or non-associative grouping.
Specifically, this means that:

```
syntax left a b
```

is *different* to:

```
syntax left a
syntax left b
```

As a consequence of this, `syntax [left|right|non-assoc]` should not be used to
group together labels with different priority.

## Prefer/avoid

Sometimes priority and associativity prove insufficient to disambiguate a grammar. In particular, sometimes it is desirable to be able to choose between two ambiguous parses directly while still not rejecting any parses if the term parsed is unambiguous. A good example of this is the famous "dangling else" problem in imperative C-like languages.

Consider the following definition (`lesson-04-E.k`):

```
module LESSON-04-E

  syntax Exp ::= "true" | "false"
  syntax Stmt ::= "if" "(" Exp ")" Stmt
                | "if" "(" Exp ")" Stmt "else" Stmt
                | "{" "}"
endmodule
```

We can write the following program (`dangling-else.if`):

```
if (true) if (false) {} else {}
```

This is ambiguous because it is unclear whether the `else` clause is part of the outer `if` or the inner `if`. At first we might try to resolve this with priorities, saying that the `if` without an `else` cannot appear as a child of the `if` with an `else`. However, because the non-terminal in the parent symbol is both preceded and followed by a terminal, this will not work.

Instead, we can resolve the ambiguity directly by telling the parser to "prefer" or "avoid" certain productions when ambiguities arise. For example, when we parse this program, we see the following ambiguity as an error message:

```
[Error] Inner Parser: Parsing ambiguity.
1: syntax Stmt ::= "if" "(" Exp ")" Stmt

`if(_)__LESSON-04-E_Stmt_Exp_Stmt`(`true_LESSON-04-E_Exp`(.KList),`if(_)_else__LESSON-04-E_S
2: syntax Stmt ::= "if" "(" Exp ")" Stmt "else" Stmt
```

```
`if(_)_else__LESSON-04-E_Stmt_Exp_Stmt_Stmt`(`true_LESSON-04-E_Exp`(.KList),`if(_)__LESSON-0
        Source(./dangling-else.if)
        Location(1,1,1,30)
```

Roughly, we see that the ambiguity is between an `if` with an `else` or an `if` without an `else`. Since we want to pick the first parse, we can tell K to "avoid" the second parse with the `avoid` attribute. Consider the following modified definition (`lesson-04-f.k`):

```
module LESSON-04-F

  syntax Exp ::= "true" | "false"
  syntax Stmt ::= "if" "(" Exp ")" Stmt
                | "if" "(" Exp ")" Stmt "else" Stmt [avoid]
                | "{" "}"
endmodule
```

Here we have added the `avoid` attribute to the `else` production. As a result, when an ambiguity occurs and one or more of the possible parses has that symbol at the top of the ambiguous part of the parse, we remove those parses from consideration and consider only those remaining. The `prefer` attribute behaves similarly, but instead removes all parses which do not have that attribute. In both cases, no action is taken if the parse is not ambiguous.

## Exercises

1. Parse the program `if (true) if (false) {} else {}` using `lesson-04-f.k` and confirm that else clause is part of the innermost `if` statement. Then modify the definition so that you will get the alternative parse.
2. Modify your solution from lesson 1.3, problem 2 so that unary negation should
   bind tighter than multiplication and division, which should bind tighter than
   addition and subtraction, and each binary operator should be left associative.
   Write these priority and associativity declarations both inline and explicitly.
3. Write a simple grammar containing at least one ambiguity that cannot be resolved via priority or associativity, and then use the `prefer` attribute to resolve that ambiguity.

## Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.5: Modules, Imports, and Requires.

# Lesson 1.5: Modules, Imports, and Requires

The purpose of this lesson is to explain how K definitions can be broken into separate modules and files and how these distinct components combine into a complete K definition.

## K's outer syntax

Recall from Lesson 1.3 that K's grammar is broken into two components: the **outer syntax** of K and the **inner syntax** of K. Outer syntax, as previously mentioned, consists of **requires**, **modules**, **imports**, and **sentences**. A K semantics is expressed by the set of sentences contained in the definition. The scope of what is considered contained in that definition is determined both by the **main semantics module** of a K definition, as well as the **requires** and **imports** present in the file that contains that module.

## Basic module syntax

The basic unit of grouping sentences in K is the module. A module consists of a **module name**, an optional list of **attributes**, a list of **imports**, and a list of **sentences**.

A module name consists of one or more groups of letters, numbers, or underscores, separated by a hyphen. Here are some valid module names: `FOO`, `FOO-BAR`, `foo0`, `foo0_bar-Baz9`. Here are some invalid module names: `-`, `-FOO`, `BAR-`, `FOO--BAR`. Stylistically, modules names are usually all uppercase with hyphens separating words, but this is not strictly enforced.

Some example modules include an empty module:

```
module LESSON-05-A

endmodule
```

A module with some attributes:

```
module LESSON-05-B [attr1, attr2, attr3(value)]

endmodule
```

A module with some sentences:

```
module LESSON-05-C
  syntax Boolean ::= "true" | "false"
  syntax Boolean ::= "not" Boolean [function]
  rule not true => false
  rule not false => true
endmodule
```

## Imports

Thus far we have only discussed definitions containing a single module. Definitions can also contain multiple modules, in which one module imports others.

An import in K appears at the top of a module, prior to any sentences. It can be specified with the `imports` keyword, followed by a module name.

For example, here is a simple definition with two modules (`lesson-05-d.k`):

```
module LESSON-05-D-1
  syntax Boolean ::= "true" | "false"
  syntax Boolean ::= "not" Boolean [function]
endmodule

module LESSON-05-D
  imports LESSON-05-D-1

  rule not true => false
  rule not false => true
endmodule
```

This K definition is equivalent to the definition expressed by the single module `LESSON-05-C`. Essentially, by importing a module, we include all of the sentences in the module being imported into the module that we import from. There are a few minor differences between importing a module and simply including its sentences in another module directly, but we will cover these differences later. Essentially, you can think of modules as a way of conceptually grouping sentences in a larger K definition.

#### Exercise

Modify `lesson-05-d.k` to include four modules: one containing the syntax, two with one rule each that imports the first module, and a final module `LESSON-05-D` containing no sentences that imports the second and third module. Check to make sure the definition still compiles and that you can still evaluate the `not` function.

## Parsing in the presence of multiple modules

As you may have noticed, each module in a definition can express a distinct set of syntax. When parsing the sentences in a module, we use the syntax **of that module**, enriched with the basic syntax of K, in order to parse rules in that module. For example, the following definition is a parser error (`lesson-05-e.k`):

```
module LESSON-05-E-1
  rule not true => false
```

```
  rule not false => true
endmodule

module LESSON-05-E-2
  syntax Boolean ::= "true" | "false"
  syntax Boolean ::= "not" Boolean [function]
endmodule
```

This is because the syntax referenced in module `LESSON-05-E-1`, namely, `not`, `true`, and `false`, is not imported by that module. You can solve this problem by simply importing the modules containing the syntax you want to use in your sentences.

## Main syntax and semantics modules

When we are compiling a K definition, we need to know where to start. We designate two specific **entry point modules**: the **main syntax module** and the **main semantics module**. The main syntax module, as well as all the modules it imports recursively, are used to create the parser for programs that you use to parse programs that you execute with `krun`. The main semantics module, as well as all the modules it imports recursively, are used to determine the rules that can be applied at runtime in order to execute a program. For example, in the above example, if the main semantics module is module `LESSON-05-D-1`, then `not` is an uninterpreted function (i.e., has no rules associated with it), and the rules in module `LESSON-05-D-2` are not included.

While you can specify the entry point modules explicitly by passing the `--main-module` and `--syntax-module` flags to `kompile`, by default, if you type `kompile foo.k`, then the main semantics module will be `FOO` and the main syntax module will be `FOO-SYNTAX`.

## Splitting a definition into multiple files

So far, while we have discussed ways to break definitions into separate conceptual components (modules), K also provides a mechanism for combining multiple files into a single K definition, namely, the **requires** directive.

In K, the `requires` keyword has two meanings. The first, the **requires** statement, appears at the top of a K file, prior to any module declarations. It consists of the keyword `requires` followed by a double-quoted string. The second meaning of the `requires` keyword will be covered in a later lesson, but it is distinguished because the second case occurs only inside modules.

The string passed to the **requires** statement contains a filename. When you run
`kompile` on a file, it will look at all of the `requires` statements in that file, look up those files on disk, parse them, and then recursively process all

the **requires** statements in those files. It then combines all the modules in all of those files together, and uses them collectively as the set of modules to which `imports` statements can refer.

## Putting it all together

Putting it all together, here is one possible way in which we could break the definition `lesson-02-c.k` from Lesson 1.2 into
multiple files and modules:

`colors.k`:

```
module COLORS
  syntax Color ::= Yellow()
                 | Blue()
endmodule
```

`fruits.k`:

```
module FRUITS
  syntax Fruit ::= Banana()
                 | Blueberry()
endmodule
```

`colorOf.k`:

```
requires "fruits.k"
requires "colors.k"

module COLOROF-SYNTAX
  imports COLORS
  imports FRUITS

  syntax Color ::= colorOf(Fruit) [function]
endmodule

module COLOROF
  imports COLOROF-SYNTAX

  rule colorOf(Banana()) => Yellow()
  rule colorOf(Blueberry()) => Blue()
endmodule
```

You would then compile this definition with `kompile colorOf.k` and use it the same way as the original, single-module definition.

### Exercise

Modify the name of the `COLOROF` module, and then recompile the definition. Try to understand why you now get a compiler error. Then, resolve this compiler

error by passing the `--main-module` and `--syntax-module` flags to kompile.

### Include path

One note can be made about how paths are resolved in `requires` statements.

By default, the path you specify is allowed to be an absolute or a relative path. If the path is absolute, that exact file is imported. If the path is relative, a matching file is looked for within all of the **include directories** specified to the compiler. By default, the include directories include the current working directory, followed by the `include/kframework/builtin` directory within your installation of K. You can also pass one or more directories to `kompile` via the `-I` command line flag, in which case these directories are prepended to the beginning of the list.

### Exercises

1. Take the solution to lesson 1.4, problem 2 which included the explicit priority and associativity declarations, and modify the definition so that the syntax of integers and brackets is in one module, the syntax of addition, subtraction, and unary negation is in another module, and the syntax of multiplication and division is in a third module. Make sure you can still parse
   the same set of expressions as before. Place priority declarations in the main
   module.
2. Modify `lesson-02-d.k` from lesson 1.2 so that the rules and syntax are in
   separate modules in separate files.
3. Place the file containing the syntax from problem 2 in another directory, then recompile the definition. Observe why a compilation error occurs. Then
   fix the compiler error by passing `-I` to kompile.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.6: Integers and Booleans.

## Lesson 1.6: Integers and Booleans

The purpose of this lesson is to explain the two most basic types of builtin sorts in K, the `Int` sort and the `Bool` sort, representing **arbitrary-precision integers** and **Boolean algebra**.

## Builtin sorts in K

K provides definitions of some useful sorts in
domains.md, found in the
`include/kframework/builtin` directory of the K installation. This file is defined via a
Literate programming
style that we will discuss in a future lesson. We will not cover all of the sorts found there immediately, however, this lesson discusses some of the details surrounding integers and Booleans, as well as providing information about how to look up more detailed knowledge about builtin functions in K's documentation.

## Booleans in K

The most basic builtin sort K provides is the `Bool` sort, representing Boolean values (i.e., `true` and `false`). You have already seen how we were able to create this type ourselves using K's parsing and disambiguation features. However, in the vast majority of cases, we prefer instead to import the version of Boolean algebra defined by K itself. Most simply, you can do this by importing the module `BOOL` in your definition. For example (`lesson-06-a.k`):

```
module LESSON-06-A
  imports BOOL

  syntax Fruit ::= Blueberry() | Banana()
  syntax Bool ::= isBlue(Fruit) [function]

  rule isBlue(Blueberry()) => true
  rule isBlue(Banana()) => false
endmodule
```

Here we have defined a simple **predicate**, i.e., a function returning a Boolean value. We are now able to perform the usual Boolean operations of and, or, and not over these values. For example (`lesson-06-b.k`):"

```
module LESSON-06-B
  imports BOOL

  syntax Fruit ::= Blueberry() | Banana()
  syntax Bool ::= isBlue(Fruit) [function]

  rule isBlue(Blueberry()) => true
  rule isBlue(Banana()) => false

  syntax Bool ::= isYellow(Fruit) [function]
                | isBlueOrYellow(Fruit) [function]
```

```
  rule isYellow(Banana()) => true
  rule isYellow(Blueberry()) => false

  rule isBlueOrYellow(F) => isBlue(F) orBool isYellow(F)
endmodule
```

In the above example, Boolean **inclusive or** is performed via the `orBool` function, which is defined in the `BOOL` module. As a matter of convention, many functions over builtin sorts in K are suffixed with the name of the primary sort over which those functions are defined. This happens so that the syntax of K does not (generally) conflict with the syntax of any other programming language, which would make it harder to define that programming language in K.

### Exercise

Write a function `isBlueAndNotYellow` which computes the appropriate Boolean expression. If you are unsure what the appropriate syntax is to use, you can refer to the `BOOL` module in
domains.md. Add a term of
sort `Fruit` for which `isBlue` and `isYellow` both return true, and test that the `isBlueAndNotYellow` function behaves as expected on all three `Fruit`s.

### Syntax Modules

For most sorts in `domains.md`, K defines more than one module that can be imported by users. For example, for the `Bool` sort, K defines the `BOOL` module that has previously already been discussed, but also provides the `BOOL-SYNTAX` module. This module, unlike the `BOOL` module, only declares the values `true` and `false`, but not any of the functions that operate over the `Bool` sort. The rationale is that you may want to import this module into the main syntax module of your definition in some cases, whereas you generally do not want to do this with the version of the module that includes all the functions over the `Bool` sort. For example, if you were defining the semantics of C++, you might import `BOOL-SYNTAX` into the syntax module of your definition, because `true` and `false` are part of the grammar of C++, but you would only import the `BOOL` module into the main semantics module, because
C++ defines its own syntax for and, or, and not that is different from the syntax defined in the `BOOL` module.

Here, for example, is how we might redefine our Boolean expression calculator to use the `Bool` sort while maintaining an idiomatic structure of modules and imports, for the first time including the rules to calculate the values of expressions themselves (`lesson-06-c.k`):

```
module LESSON-06-C-SYNTAX
```

```
  imports BOOL-SYNTAX

  syntax Bool ::= "(" Bool ")" [bracket]
                > "!" Bool [function]
                > left:
                  Bool "&&" Bool [function]
                | Bool "^" Bool [function]
                | Bool "||" Bool [function]
endmodule

module LESSON-06-C
  imports LESSON-06-C-SYNTAX
  imports BOOL

  rule ! B => notBool B
  rule A && B => A andBool B
  rule A ^ B => A xorBool B
  rule A || B => A orBool B
endmodule
```

Note the encapsulation of syntax: the `LESSON-06-C-SYNTAX` module contains exactly the syntax of our Boolean expressions, and no more, whereas any other syntax needed to implement those functions is in the `LESSON-06-C` module instead.

**Exercise** Add an "implies" function to the above Boolean expression calculator, using the
`->` symbol to represent implication. You can look up K's builtin "implies" function in the `BOOL` module in `domains.md`.

## Integers in K

Unlike most programming languages, where the most basic integer type is a fixed-precision integer type, the most commonly used integer sort in K is the `Int` sort, which represents the **mathematical** integers, ie, arbitrary-precision integers.

K provides three main modules for import when using the `Int` sort. The first, containing all the syntax of integers as well as all of the functions over integers, is the `INT` module. The second, which provides just the syntax of integer literals themselves, is the `INT-SYNTAX` module. However, unlike most builtin sorts in K, K also provides a third module for the `Int` sort: the `UNSIGNED-INT-SYNTAX` module. This module provides only the syntax of **non-negative integers**, i.e., natural numbers. The reasons for this involve lexical ambiguity. Generally speaking, in most programming languages, `-1` is not a literal, but instead a literal to which the unary negation operator is

applied. K thus provides this module to ease in specifying the syntax of such languages.

For detailed information about the functions available over the `Int` sort, refer to `domains.md`. Note again how we append `Int` to the end of most of the integer operations to ensure they do not collide with the syntax of other programming languages.

### Exercises

1. Extend your solution from lesson 1.4, problem 2 to implement the rules that define the behavior of addition, subtraction, multiplication, and division. Do not worry about the case when the user tries to divide by zero
   at this time. Use `/Int` to implement division. Test your new calculator implementation by executing the arithmetic expressions you wrote as part of
   lesson 1.3, problem 2. Check to make sure each computes the value you expected.
2. Combine the Boolean expression calculator from this lesson with your solution to problem 1, and then extend the combined calculator with the `<`,
   `<=`, `>`, `>=`, `==`, and `!=` expressions. Write some Boolean expressions that combine integer and Boolean operations, and test to ensure that these expressions return the expected truth value.
3. Compute the following expressions using your solution from problem 2: `7 / 3`, `7 / -3`, `-7 / 3`, `-7 / -3`. Then replace the `/Int` function in your definition with `divInt` instead, and observe how the value of the above
   expressions changes. Why does this occur?

### Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.7: Side Conditions and Rule Priority.

## Lesson 1.7: Side Conditions and Rule Priority

The purpose of this lesson is to explain how to write conditional rules in K, and to explain how to control the order in which rules are tried.

### Side Conditions

So far, all of the rules we have discussed have been **unconditional rules**. If the left-hand side of the rule matches the arguments to the function, the

rule applies. However, there is another type of rule, a **conditional rule**. A conditional rule consists of a **rule body** containing the patterns to match, and a **side condition** representing a Boolean expression that must evaluate to true in order for the rule to apply.

Side conditions in K are introduced via the `requires` keyword immediately following the rule body. For example, here is a rule with a side condition (`lesson-07-a.k`):

```
module LESSON-07-A
  imports BOOL
  imports INT

  syntax Grade ::= "letter-A"
                 | "letter-B"
                 | "letter-C"
                 | "letter-D"
                 | "letter-F"
                 | gradeFromPercentile(Int) [function]

  rule gradeFromPercentile(I) => letter-A requires I >=Int 90
endmodule
```

In this case, the `gradeFromPercentile` function takes a single integer argument. The function evaluates to `letter-A` if the argument passed is greater than 90. Note that the side condition is allowed to refer to variables that appear on the left-hand side of the rule. In the same manner as variables appearing on the right-hand side, variables that appear in the side condition evaluate to the value that was matched on the left-hand side. Then the functions in the side condition are evaluated, which returns a term of sort `Bool`. If the term is equal to `true`, then the rule applies. Bear in mind that the side condition is only evaluated at all if the patterns on the left-hand side of the rule match the term being evaluated.

#### Exercise

Write a rule that evaluates `gradeFromPercentile` to `letter-B` if the argument to the function is in the range [80,90). Test that the function correctly evaluates various numbers between 80 and 100.

## `owise` Rules

So far, all the rules we have introduced have had the same **priority**. What this means is that K does not necessarily enforce an order in which the rules are tried. We have only discussed functions so far in K, so it is not immediately clear why this choice was made, given that a function is not considered well-defined if multiple rules for evaluating it are capable of evaluating the same arguments to different results. However, in future lessons

we will discuss other types of rules in K, some of which can be **non-deterministic**. What this means is that if more than one rule is capable of matching, then K will explore both possible rules in parallel, and consider each of their respective results when executing your program. Don't worry too much about this right now, but just understand that because of the potential later for nondeterminism, we don't enforce a total ordering on the order in which rules are attempted to be applied.

However, sometimes this is not practical; It can be very convenient to express that a particular rule applies if no other rules for that function are applicable. This can be expressed by adding the `owise` attribute to a rule. What this means, in practice, is that this rule has lower priority than other rules, and will only be tried to be applied after all the other, higher-priority rules have been tried and they have failed.

For example, in the above exercise, we had to add a side condition containing two Boolean comparisons to the rule we wrote to handle `letter-B` grades. However, in practice this meant that we compare the percentile to 90 twice. We can more efficiently and more idiomatically write the `letter-B` case for the `gradeFromPercentile` rule using the `owise` attribute (`lesson-07-b.k`):

```
module LESSON-07-B
  imports BOOL
  imports INT

  syntax Grade ::= "letter-A"
                 | "letter-B"
                 | "letter-C"
                 | "letter-D"
                 | "letter-F"
                 | gradeFromPercentile(Int) [function]

  rule gradeFromPercentile(I) => letter-A requires I >=Int 90
  rule gradeFromPercentile(I) => letter-B requires I >=Int 80 [owise]
endmodule
```

This rule is saying, "if all the other rules do not apply, then the grade is a B if the percentile is greater than or equal to 80." Note here that we use both a side condition and an `owise` attribute on the same rule. This is not required (as we will see later), but it is allowed. What this means is that the side condition is only tried if the other rules did not apply **and** the left-hand side of the rule matched. You can even use more complex matching on
the left-hand side than simply a variable. More generally, you can also have multiple higher-priority rules, or multiple `owise` rules. What this means in practice is that all of the non-`owise` rules are tried first, in any order, followed by all the `owise` rules, in any order.

**Exercise**

The grades `D` and `F` correspond to the percentile ranges [60, 70) and [0, 60) respectively. Write another implementation of `gradeFromPercentile` which handles only these cases, and uses the `owise` attribute to avoid redundant Boolean comparisons. Test that various percentiles in the range [0, 70) are evaluated correctly.

## Rule Priority

As it happens, the `owise` attribute is a specific case of a more general concept we call **rule priority**. In essence, each rule is assigned an integer priority. Rules are tried in increasing order of priority, starting with a rule with priority zero, and trying each increasing numerical value successively.

By default, a rule is assigned a priority of 50. If the rule has the `owise` attribute, it is instead given the priority 200. You can see why this will cause `owise` rules to be tried after regular rules.

However, it is also possible to directly assign a numerical priority to a rule via the `priority` attribute. For example, here is an alternative way we could express the same two rules in the `gradeFromPercentile` function (`lesson-07-c.k`):

```
module LESSON-07-C
  imports BOOL
  imports INT

  syntax Grade ::= "letter-A"
                 | "letter-B"
                 | "letter-C"
                 | "letter-D"
                 | "letter-F"
                 | gradeFromPercentile(Int) [function]

  rule gradeFromPercentile(I) => letter-A requires I >=Int 90 [priority(50)]
  rule gradeFromPercentile(I) => letter-B requires I >=Int 80 [priority(200)]
endmodule
```

We can, of course, assign a priority equal to any non-negative integer. For example, here is a more complex example that handles the remaining grades (`lesson-07-d.k`):

```
module LESSON-07-D
  imports BOOL
  imports INT

  syntax Grade ::= "letter-A"
```

```
                | "letter-B"
                | "letter-C"
                | "letter-D"
                | "letter-F"
                | gradeFromPercentile(Int) [function]

  rule gradeFromPercentile(I) => letter-A requires I >=Int 90 [priority(50)]
  rule gradeFromPercentile(I) => letter-B requires I >=Int 80 [priority(51)]
  rule gradeFromPercentile(I) => letter-C requires I >=Int 70 [priority(52)]
  rule gradeFromPercentile(I) => letter-D requires I >=Int 60 [priority(53)]
  rule gradeFromPercentile(_) => letter-F                     [priority(54)]
endmodule
```

Note that we have introduced a new piece of syntax here: `_`. This is actually just a variable. However, as a special case, when a variable is named `_`, it does not bind a value that can be used on the right-hand side of the rule, or in a side condition. Effectively, `_` is a placeholder variable that means "I don't care about this term."

In this example, we have explicitly expressed the order in which the rules of this function are tried. Since rules are tried in increasing numerical priority, we first try the rule with priority 50, then 51, then 52, 53, and finally 54.

As a final note, remember that if you assign a rule a priority higher than 200, it will be tried **after** a rule with the `owise` attribute, and if you assign a rule a priority less than 50, it will be tried **before** a rule with no explicit priority.

## Exercises

1. Write a function `isEven` that returns whether an integer is an even number.
   Use two rules and one side condition. The right-hand side of the rules should
   be Boolean literals. Refer back to
   domains.md for the relevant
   integer operations.

2. Modify the calculator application from lesson 1.6, problem 2, so that division
   by zero will no longer make `krun` crash with a "Divison by zero" exception. Instead, the `/` function should not match any of its rules if the denominator
   is zero.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.8: Literate Programming with Markdown.

## Lesson 1.8: Literate Programming with Markdown

The purpose of this lesson is to teach a paradigm for performing literate programming in K, and explain how this can be used to create K definitions that are also documentation.

### Markdown and K

The K tutorial so far has been written in
Markdown. Markdown,
for those not already familiar, is a lightweight plain-text format for styling text. From this point onward, we assume you are familiar with Markdown and how
to write Markdown code. You can refer to the above link for a tutorial if you are not already familiar.

What you may not necessarily realize, however, is that the K tutorial is also a sequence of K definitions written in the manner of
Literate Programming.
For detailed information about Literate Programming, you can read the linked Wikipedia article, but the short summary is that literate programming is a way of intertwining documentation and code together in a manner that allows executable code to also be, simultaneously, a documented description of that code.

K is provided with built-in support for literate programming using Markdown. By default, if you pass a file with the `.md` file extension to `kompile`, it will look for any code blocks containing k code in that file, extract out that K code into pure K, and then compile it as if it were a `.k` file.

A K code block begins with a line of text containing the keyword ```` ```k ````, and ends when it encounters another ```` ``` ```` keyword.

For example, if you view the markdown source of this document, this is a K code block:

```
module LESSON-08
  imports INT
```

Only the code inside K code blocks will actually be sent to the compiler. The rest, while it may appear in the document when rendered by a markdown viewer, is essentially a form of code comment.

When you have multiple K code blocks in a document, K will append each one together into a single file before passing it off to the outer parser.

For example, the following code block contains sentences that are part of the `LESSON-08` module that we declared the beginning of above:

```
syntax Int ::= Int "+" Int [function]
rule I1 + I2 => I1 +Int I2
```

**Exercise**

Compile this file with `kompile README.md --main-module LESSON-08`. Confirm
that you can use the resulting compiled definition to evaluate the `+`
function.

## Markdown Selectors

On occasion, you may want to generate multiple K definitions from a single Markdown file. You may also wish to include a block of syntax-highlighted K code that nonetheless does **not** appear as part of your K definition. It is possible to accomplish this by means of the built-in support for syntax highlighting in Markdown. Markdown allows a code block that was begun with ``` to be immediately followed by a string which is used to signify what programming language the following code is written in. However, this feature actually allows arbitrary text to appear describing that code block. Markdown parsers are able to parse this text and render the code block differently depending on what text appears after the backticks.

In K, you can use this functionality to specify one or more
**Markdown selectors** which are used to describe the code block. A Markdown selector consists of a sequence of characters containing letters, numbers, and underscores. A code block can be designated with a single selector by appending the selector immediately following the backticks that open the code block.

For example, here is a code block with the `foo` selector:

```foo
foo bar
```

Note that this is not K code. By convention, K code should have the `k`
selector on it. You can express multiple selectors on a code block by putting them between curly braces and prepending each with the `.` character. For example, here is a code block with the `foo` and `k` selectors:

```k
syntax Int ::= foo(Int) [function]
rule foo(0) => 0
```

Because this code block contains the `k` Markdown selector, by default it is included as part of the K definition being compiled.

**Exercise**

Confirm this fact by using `krun` to evaluate `foo(0)`.

## Markdown Selector Expressions

By default, as previously stated, K includes in the definition any code block with the `k` selector. However, this is merely a specific instance of a general principle, namely, that K allows you to control which selectors get included in your K definition. This is done by means of the `--md-selector` flag to `kompile`. This flag accepts a **Markdown selector expression**, which you can essentially think of as a kind of Boolean algebra over Markdown selectors. Each selector becomes an atom, and you can combine these atoms via the `&`, `|`, `!`, and `()` operators.

Here is a grammar, written in K, of the language of Markdown selector expressions:

```
syntax Selector ::= r"[0-9a-zA-Z_]+" [token]
syntax SelectorExp ::= Selector
                     | "(" SelectorExp ")" [bracket]
                     > right:
                       "!" SelectorExp
                     > right:
                       SelectorExp "&" SelectorExp
                     > right:
                       SelectorExp "|" SelectorExp
```

Here is a selector expression that selects all the K code blocks in this definition except the one immediately above:

```
k & (! selector)
```

## Addendum

This code block exists in order to make the above lesson a syntactically valid K definition. Consider why it is necessary.

```
endmodule
```

**Exercises**

1. Compile this lesson with the selector expression `k & (! foo)` and confirm that you get a parser error if you try to evaluate the `foo` function with the
   resulting definition.
2. Compile Lesson 1.3 as a K definition. Identify
   why it fails to compile. Then pass an appropriate `--md-selector` to the compiler in order to make it compile.

3. Modify your calculator application from lesson 1.7, problem 2, to be written
   in a literate style. Consider what text might be appropriate to turn the resulting markdown file into documentation for your calculator.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.9: Unparsing and the format and color attributes.

## Lesson 1.9: Unparsing and the format and color attributes

The purpose of this lesson is to teach the user about how terms are pretty-printed in K, and how the user can make adjustments to the default settings for how to print specific terms.

### Parsing, Execution, and Unparsing

When you use `krun` to interpret a program, the tool passes through three major phases. In the first, parsing, the program itself is parsed using either `kast` or an ahead-of-time parser generated via Bison, and the resulting AST becomes the input to the interpreter. In the second phase, execution, K evaluates functions and (as we will discuss in depth later) performs rewrite steps to iteratively transform the program state. The third and final phase is called **unparsing**, because it consists of taking the final state of the application after the program has been interpreted, and converting it from an AST back into
text that (in theory, anyway) could be parsed back into the same AST that was the output of the execution phase.

In practice, unparsing is not always precisely reversible. It turns out (although we are not going to cover exactly why this is here), that constructing a sound algorithm that takes a grammar and an AST and emits text
that could be parsed via that grammar to the original AST is an **NP-hard problem**. As a result, in the interests of avoiding exponential time algorithms when users rarely care about unparsing being completely sound, we take certain shortcuts that provide a linear-time algorithm that *approximates* a sound solution to the problem while sacrificing the notion that the result can be parsed into the exact original term in all cases.

This is a lot of theoretical explanation, but at root, the unparsing process is fairly simple: it takes a K term that is the output of execution and pretty prints it according to the syntax defined by the user in their K definition. This is useful because the original AST is not terribly user-readable, and it

is difficult to visualize the entire term or decipher information about the final state of the program at a quick glance. Of course, in rare cases, the pretty-printed configuration loses information of relevance, which is why K allows you to obtain the original AST on request.

As an example of all of this, consider the following K definition (`lesson-09-a.k`):

```
module LESSON-09-A
  imports BOOL

  syntax Exp ::= "(" Exp ")" [bracket]
               | Bool
               > "!" Exp
               > left:
                 Exp "&&" Exp
               | Exp "^" Exp
               | Exp "||" Exp

  syntax Exp ::= id(Exp) [function]
  rule id(E) => E
endmodule
```

This is similar to the grammar we defined in `LESSON-06-C`, with the difference that the Boolean expressions are now constructors of sort `Exp` and we define a trivial function over expressions that returns its argument unchanged.

We can now parse a simple program in this definition and use it to unparse some Boolean expressions. For example (`exp.bool`):

```
id(true&&false&&!true^(false||true))
```

Here is a program that is not particularly legible at first glance, because all extraneous whitespace has been removed. However, if we run `krun exp.bool`, we
see that the result of the unparser will pretty-print this expression rather nicely:

```
<k>
  true && false && ! true ^ ( false || true ) ~> .
</k>
```

Notably, not only does K insert whitespace where appropriate, it is also smart enough to insert parentheses where necessary in order to ensure the correct parse. For example, without those parentheses, the expression above would parse
equivalent to the following one:

```
(((true && false) && ! true) ^ false) || true
```

Indeed, you can confirm this by passing that exact expression to the `id` function and evaluating it, then looking at the result of the unparser:

```
<k>
  true && false && ! true ^ false || true ~> .
</k>
```

Here, because the meaning of the AST is the same both with and without parentheses, K does not insert any parentheses when unparsing.

**Exercise**

Modify the grammar of `LESSON-09-A` above so that the binary operators are right associative. Try unparsing `exp.bool` again, and note how the result is different. Explain the reason for the difference.

## Custom unparsing of terms

You may have noticed that right now, the unparsing of terms is not terribly imaginative. All it is doing is taking each child of the term, inserting it into the non-terminal positions of the production, then printing the production with a space between each terminal or non-terminal. It is easy to see why this might not be desirable in some cases. Consider the following K definition (`lesson-09-b.k`):

```
module LESSON-09-B
  imports BOOL

  syntax Stmt ::= "{" Stmt "}" | "{" "}"
                > right:
                  Stmt Stmt
                | "if" "(" Bool ")" Stmt
                | "if" "(" Bool ")" Stmt "else" Stmt [avoid]
endmodule
```

This is a statement grammar, simplified to the point of meaninglessness, but still useful as an object lesson in unparsing. Consider the following program in this grammar (`if.stmt`):

```
if (true) {
  if (true) {}
  if (false) {}
  if (true) {
    if (false) {} else {}
  } else {
    if (false) {}
  }
}
```

This is how that term would be unparsed if it appeared in the output of krun:

```
if ( true ) { if ( true ) { } if ( false ) { } if ( true ) { if ( false ) { } else { } } els
```

This is clearly much less legible than we started with! What are we to do? Well, K provides an attribute, `format`, that can be applied to any production, which controls how that production gets unparsed. You've seen how it gets unparsed by default, but via this attribute, the developer has complete control over how the term is printed. Of course, the user can trivially create ways to print terms that would not parse back into the same term. Sometimes this is even desirable. But in most cases, what you are interested in is controlling the line breaking, indentation, and spacing of the production.

Here is an example of how you might choose to apply the `format` attribute to improve how the above term is unparsed (`lesson-09-c.k`):

```
module LESSON-09-C
  imports BOOL

  syntax Stmt ::= "{" Stmt "}" [format(%1%i%n%2%d%n%3)] | "{" "}" [format(%1%2)]
                > right:
                  Stmt Stmt [format(%1%n%2)]
                | "if" "(" Bool ")" Stmt [format(%1 %2%3%4 %5)]
                | "if" "(" Bool ")" Stmt "else" Stmt [avoid, format(%1 %2%3%4 %5 %6 %7)]
endmodule
```

If we compile this new definition and unparse the same term, this is the result we get:

```
if (true) {
  if (true) {}
  if (false) {}
  if (true) {
    if (false) {} else {}
  } else {
    if (false) {}
  }
}
```

This is the exact same text we started with! By adding the `format` attributes, we were able to indent the body of code blocks, adjust the spacing of if statements, and put each statement on a new line.

How exactly was this achieved? Well, each time the unparser reaches a term, it looks at the `format` attribute of that term. That `format` attribute is a mix of characters and **format codes**. Format codes begin with the `%` character. Each character in the `format` attribute other than a format code is appended verbatim to the output, and each format code is handled according to
its meaning, transformed (possibly recursively) into a string of text, and

spliced into the output at the position the format code appears in the format string.

Provided for reference is a table with a complete list of all valid format codes, followed by their meaning:

| Format Code | Meaning |
|---|---|
| n | Insert '\n' followed by the current indentation level |
| i | Increase the current indentation level by 1 |
| d | Decrease the current indentation level by 1 |
| c | Move to the next color in the list of colors for this production (see next section) |
| r | Reset color to the default foreground color for the terminal (see next section) |
| an integer | Print a terminal or non-terminal from the production. The integer is treated as a 1-based index into the terminals and non-terminals of the production.<br><br>If the offset refers to a terminal, move to the next color in the list of colors for this production, print the value of that terminal, then reset the color to the default foreground color for the terminal.<br><br>If the offset refers to a regular expression terminal, it is an error.<br><br>If the offset refers to a non-terminal, unparse the corresponding child of the current term (starting with the current indentation level) and print the resulting text, then set the current color and indentation level to the color and indentation level following unparsing that term. |
| other char | Print that character verbatim |

**Exercise**

Change the format attributes for `LESSON-09-C` so that `if.stmt` will unparse as follows:

```
if (true)
{
  if (true)
  {
  }
  if (false)
  {
  }
  if (true)
  {
    if (false)
    {
    }
    else
    {
    }
  }
  else
  {
    if (false)
    {
    }
  }
}
```

## Output coloring

When the output of unparsing is displayed on a terminal supporting colors, K is capable of coloring the output, similar to what is possible with a syntax highlighter. This is achieved via the `color` and `colors` attributes.

Essentially, both the `color` and `colors` attributes are used to construct a list of colors associated with each production, and then the format attribute is used to control how those colors are used to unparse the term. At its most basic level, you can set the `color` attribute to color all the terminals in the production a certain color, or you can use the `colors` attribute to specify a comma-separated list of colors for each terminal in the production. At a more advanced level, the `%c` and `%r` format codes control how the formatter interacts with the list of colors specified by the `colors` attribute. You can essentially think of the `color` attribute as a way of specifying that you want all the colors in the list to be the same color.

Note that the `%c` and `%r` format codes are relatively primitive in nature.

The `color` and `colors` attributes merely maintain a list of colors, whereas the `%c` and `%r` format codes merely control how to advance through that list and how individual text is colored.

It is an error if the `colors` attribute does not provide all the colors needed by the terminals and escape codes in the production. `%r` does not change the position in the list of colors at all, so the next `%c` will advance to the following color.

As a complete example, here is a variant of LESSON-09-A which colors the various boolean operators:

```
module LESSON-09-D
  imports BOOL

  syntax Exp ::= "(" Exp ")" [bracket]
               | Bool
               > "!" Exp [color(yellow)]
               > left:
                 Exp "&&" Exp [color(red)]
               | Exp "^" Exp [color(blue)]
               | Exp "||" Exp [color(green)]

  syntax Exp ::= id(Exp) [function]
  rule id(E) => E
endmodule
```

For a complete list of allowed colors, see here.

## Exercises

1. Use the color attribute on `LESSON-09-C` to color the keywords `true` and `false` one color, the keywords `if` and `else` another color, and the operators `(`, `)`, `{`, and `}` a third color.
2. Use the `format`, `color`, and `colors` attributes to tell the unparser to style the expression grammar from lesson 1.8, problem 3 according to your own personal preferences for syntax highlighting and code formatting. You can view the result of the unparser on a function term without evaluating that function by means of the command `kparse <file> | kore-print -`.

## Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.10: Strings.

# Lesson 1.10: Strings

The purpose of this lesson is to explain how to use the `String` sort in K to represent **sequences of characters**, and explain where to find additional information about builtin functions over strings.

## The `String` Sort

In addition to the `Int` and `Bool` sorts covered in Lesson 1.6, K provides, among others, the `String` sort to represent sequences of characters. You can import this functionality via the `STRING-SYNTAX` module, which contains the syntax of string literals in K, and the `STRING` module, which contains all the functions that operate over the `String` type.

Strings in K are double-quoted. The following list of escape sequences is supported:

| Escape Sequence | Meaning |
|---|---|
| `\"` | The literal character " |
| `\\` | The literal character \ |
| `\n` | The newline character (ASCII code 0x0a) |
| `\r` | The carriage return character (ASCII code 0x0d) |
| `\t` | The tab character (ASCII code 0x09) |
| `\f` | The form feed character (ASCII code 0x0c) |
| `\x00` | \x followed by 2 hexadecimal digits indicates a code point between 0x00 and 0xFF |
| `\u0000` | \u followed by 4 hexadecimal digits indicates a code point between 0x0000 and 0xFFFF |
| `\U00000000` | \U followed by 8 hexadecimal digits indicates a code point between 0x000000 and 0x10FF |

Please note that as of the current moment, K's unicode support is not fully complete, so you may run into errors using code points greater than 0xff.

As an example, you can construct a string literal containing the following block of text:

```
This is an example block of text.
Here is a quotation: "Hello world."
    This line is indented.
ÁÉÍÓÚ
```

Like so:

```
"This is an example block of text.\nHere is a quotation: \"Hello world.\"\n\tThis line is in
```

## Basic String Functions

The full list of functions provided for the `String` sort can be found in domains.md, but here we
describe a few of the more basic ones.

### String concatenation

The concatenation operator for strings is `+String`. For example, consider the following K rule that constructs a string from component parts (`lesson-10.k`):

```
module LESSON-10
  imports STRING

  syntax String ::= msg(String) [function]
  rule msg(S) => "The string you provided: " +String S +String "\nHave a nice day!"
endmodule
```

Note that this operator is `O(N)`, so repeated concatenations are inefficient. For information about efficient string concatenation, refer to
Lesson 2.14.

### String length

The function to return the length of a string is `lengthString`. For example, `lengthString("foo")` will return 3, and `lengthString("")` will return 0. The return value is the length of the string in **code points**.

### Substring computation

The function to compute the substring of a string is `substrString`. It takes two string indices, starting from 0, and returns the substring within the range [start..end). It is only defined if `end >= start`, `start >= 0`, and `end <= length of string`. Here, for example, we return the first 5 characters of a string:

```
substrString(S, 0, 5)
```

Here we return all but the first 3 characters:

```
substrString(S, 3, lengthString(S))
```

## Exercises

1. Write a function that takes a paragraph of text (i.e., a sequence of sentences, each ending in a period), and constructs a new (nonsense) sentence
   composed of the first word of each sentence, followed by a period. Do not worry about capitalization or periods within the sentence which do not

end the
sentence (e.g. "Dr."). You can assume that all whitespace within the paragraph
are spaces. For more information about the functions over strings required to
implement such a function, refer to `domains.md`.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.11: Casting Terms.

## Lesson 1.11: Casting Terms

The purpose of this lesson is to explain how to use **cast** expressions in order to disambiguate terms using sort information. We also explain how the variable sort inference algorithm works in K, and how to change the default behavior by casting variables to a particular sort.

### Casting in K

Sometimes the grammar you write for your rules in K can be a little bit ambiguous on purpose. While grammars for programming languages may be unambiguous when considered in their entirety, K allows you to write rules involving arbitrary **fragments** of that grammar, and those fragments can sometimes be ambiguous by themselves, or similar enough to other fragments of the grammar to trigger ambiguity. As a result, in addition to the tools covered in Lesson 1.4, K provides one
additional powerful tool for disambiguation: cast expressions.

K provides three main types of casts: the semantic cast, the strict cast, and the projection cast. We will cover each of them, and their similarities and differences, in turn.

#### Semantic casts

The most basic, and most common, type of cast in K is called the **semantic cast**. For every sort `S` declared in a module, K provides the following (implicit) production for use in sentences:

```
syntax S ::= S ":S"
```

Note that `S` simply represents the name of the sort. For example, if we defined a sort `Exp`, the actual production for that sort would be:

```
syntax Exp ::= Exp ":Exp"
```

At runtime, this expression will not actually exist; it is merely an annotation to the compiler describing the sort of the term inside the cast. It is telling the compiler that the term inside the cast must be of sort `Exp`. For example, if we had the following grammar:

```
module LESSON-11-A
  imports INT

  syntax Exp ::= Int | Exp "+" Exp
  syntax Stmt ::= "if" "(" Exp ")" Stmt | "{" "}"
endmodule
```

Then we would be able to write `1:Exp`, or `(1 + 2):Exp`, but not `{}:Exp`.

You can also restrict the sort that a variable in a rule will match by casting it. For example, consider the following additional module:

```
module LESSON-11-B
  imports LESSON-11-A
  imports BOOL

  syntax Term ::= Exp | Stmt
  syntax Bool ::= isExpression(Term) [function]

  rule isExpression(_E:Exp) => true
  rule isExpression(_) => false [owise]
endmodule
```

Here we have defined a very simple function that decides whether a term is an expression or a statement. It does this by casting the variable inside the `isExpression` rule to sort `Exp`. As a result, that variable will only match terms of sort `Exp`. Thus, `isExpression(1)` will return true, as will `isExpression(1 + 2)`, but
`isExpression({})` will return false.

**Exercise** Verify this fact for yourself by running `isExpression` on the above examples. Then
write an `isStatement` function, and test that it works as expected.

### Strict casts

On occasion, a semantic cast is not strict enough. It might be that you want to, for disambiguation purposes, say **exactly** what sort a term is. For example, consider the following definition:

```
module LESSON-11-C
  imports INT

  syntax Exp ::= Int | Exp "+" Exp [exp]
```

```
  syntax Exp2 ::= Exp | Exp2 "+" Exp2 [exp2]
endmodule
```

This grammar is a little ambiguous and contrived, but it serves to demonstrate how a semantic cast might be insufficient to disambiguate a term. If we were to write the term `(I1:Int + I2:Int):Exp2`, the term would be ambiguous, because the cast is not sufficiently strict to determine whether you mean to derive the "+" production tagged `exp`, or the one tagged `exp2`.

In this situation, there is a solution: the **strict cast**. For every sort `S` in your grammar, K also defines the following production:

```
  syntax S ::= S "::S"
```

This may at first glance seem the same as the previous cast. And indeed, from the perspective of the grammar and from the perspective of rewriting, they are in fact identical. However, the second variant has a unique meaning in the **type system** of K: namely, the term inside the cast cannot be a **subsort**, i.e., a term of another sort `S2` such that the production `syntax S ::= S2` exists.

As a result, if we were to write in the above grammar the term `(I1:Int + I2:Int)::Exp2`, then we would know that the second derivation above
should be chosen, whereas if we want the first derivation, we could write `(I1:Int + I2:Int)::Exp`.

### Projection casts

Thus far we have focused entirely on casts which exist solely to inform the compiler about the sort of terms. However, sometimes when dealing with grammars
containing subsorts, it can be desirable to reason with the subsort production itself, which **injects** one sort into another. Remember from above that such a production looks like `syntax S ::= S2`. This type of production, called a **subsort production**, can be thought of as a type of inheritance involving constructors. If we have the above production in our grammar, we say that `S2` is a subsort of `S`, or that any `S2` is also an `S`. K implicitly maintains a symbol at runtime which keeps track of where such subsortings occur; this symbol is called an **injection**.

Sometimes, when one sort is a subsort of another, it can be the case that a function returns one sort, but you actually want to cast the result of calling that function to another sort which is a subsort of the first sort. This is similar to what happens with inheritance in an object-oriented language, where you might cast a superclass to a subclass if you know for sure the object at runtime is in fact an instance of that class.

K provides something similar for subsorts: the **projection cast**.

For each pair of sorts `S` and `S2`, K provides the following production:

```
  syntax S ::= "{" S2 "}" ":>S"
```

What this means is that you take any term of sort `S2` and **cast** it to sort `S`. If the term of sort S2 consists of an injection containing a term of sort `S`, then this will return that term. Otherwise, an error occurs and rewriting fails, returning the projection function which failed to apply. The sort is not actually checked at compilation time; rather, it is a runtime check inserted into the code that runs when the rule applies.

For example, here is a module that makes use of projection casts:

```
module LESSON-11-D
  imports INT
  imports BOOL

  syntax Exp ::= Int | Bool | Exp "+" Exp | Exp "&&" Exp

  syntax Exp ::= eval(Exp) [function]
  rule eval(I:Int) => I
  rule eval(B:Bool) => B
  rule eval(E1 + E2) => {eval(E1)}:>Int +Int {eval(E2)}:>Int
  rule eval(E1 && E2) => {eval(E1)}:>Bool andBool {eval(E2)}:>Bool
endmodule
```

Here we have defined constructors for a simple expression language over Booleans and integers, as well as a function `eval` that evaluates these expressions to a value. Because that value could be an integer or a Boolean, we need the casts in the last two rules in order to meet the type signature of `+Int` and `andBool`. Of course, the user can write ill-formed expressions like `1 && true` or `false + true`, but these will cause errors at runtime, because the projection cast will fail.

## Exercises

1. Extend the `eval` function in `LESSON-11-D` to include Strings and add a `.` operator which concatenates them.
2. Modify your solution from lesson 1.9, problem 2 by using an `Exp` sort to express the integer and Boolean expressions that it supports, in the same style
   as `LESSON-11-D`. Then write an `eval` function that evaluates all terms of sort `Exp` to either a `Bool` or an `Int`.

## Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.12: Syntactic Lists.

# Lesson 1.12: Syntactic Lists

The purpose of this lesson is to explain how K provides support for syntactic repetition through the use of the `List{}` and `NeList{}` constructs, generally called **syntactic lists**.

## The `List{}` construct

Sometimes, when defining a grammar in K, it is useful to define a syntactic construct consisting of an arbitrary-length sequence of items. For example, you might wish to define a function call construct, and need to express a way of passing arguments to the function. You can in theory simply define these productions using ordinary constructors, but it can be tricky to get the syntax exactly right in K without a lot of tedious glue code.

For this reason, K provides a way of specifying that a non-terminal represents a syntactic list (`lesson-12-a.k`):

```
module LESSON-12-A-SYNTAX
  imports INT-SYNTAX

  syntax Ints ::= List{Int,","}
endmodule

module LESSON-12-A
  imports LESSON-12-A-SYNTAX
endmodule
```

Note that instead of a sequence of terminals and non-terminals, the right hand side of the `Ints` production contains the symbol `List` followed by two items in curly braces. The first item is the non-terminal which is the element type of the list, and the second item is a terminal representing the separator of the list. As a special case, lists which are separated only by whitespace can be specified with a separator of `""`.

This `List{}` construct is roughly equivalent to the following definition (`lesson-12-b.k`):

```
module LESSON-12-B-SYNTAX
  imports INT-SYNTAX

  syntax Ints ::= Int "," Ints | ".Ints"
endmodule

module LESSON-12-B
  imports LESSON-12-B-SYNTAX
endmodule
```

As you can see, the `List{}` construct represents a cons-list with an element

at the head and another list at the tail. The empty list is represented by a `.` followed by the sort of the list.

However, the `List{}` construct provides several key syntactic conveniences over the above definition. First of all, when writing a list in a rule, explicitly writing the terminator is not always required. For example, consider the following additional module (`lesson-12-c.k`):

```
module LESSON-12-C
  imports LESSON-12-A
  imports INT

  syntax Int ::= sum(Ints) [function]
  rule sum(I:Int) => I
  rule sum(I1:Int, I2:Int, Is:Ints) => sum(I1 +Int I2, Is)
endmodule
```

Here we see a function that sums together a non-empty list of integers. Note in particular the first rule. We do not explicitly mention `.Ints`, but in fact, the rule in question is equivalent to the following rule:

```
  rule sum(I:Int, .Ints) => I
```

The reason for this is that K will automatically insert a list terminator anywhere a syntactic list is expected, but an element of that list appears instead. This works even with lists of more than one element:

```
  rule sum(I1:Int, I2:Int) => I1 +Int I2
```

This rule is redundant, but here we explicitly match a list of exactly two elements, because the `.Ints` is implicitly added after `I2`.

**Exercise**

Write a function `concat` which takes a list of `String` and concatenates them all together. Do not worry if the function is O(n^2).

## Parsing Syntactic Lists in Programs

An additional syntactic convenience takes place when you want to express a syntactic list in the input to `krun`. In this case, K will automatically transform the grammar in `LESSON-12-B-SYNTAX` into the following (`lesson-12-d.k`):

```
module LESSON-12-D
  imports INT-SYNTAX

  syntax Ints ::= #NonEmptyInts | #IntsTerminator
  syntax #NonEmptyInts ::= Int "," #NonEmptyInts
                         | Int #IntsTerminator
```

```
  syntax #IntsTerminator ::= ""
endmodule
```

This allows you to express the usual comma-separated list of arguments where an empty list is represented by the empty string, and you don't have to explicitly terminate the list. Because of this, we can write the syntax of function calls in C very easily (`lesson-12-e.k`):

```
module LESSON-12-E
  syntax Id ::= r"[a-zA-Z_][a-zA-Z0-9_]*" [token]
  syntax Exp ::= Id | Exp "(" Exps ")"
  syntax Exps ::= List{Exp,","}
endmodule
```

**Exercise**

Write some function call expressions using identifiers in C and verify with `kast` that the above grammar captures the intended syntax. Make sure to test with function calls with zero, one, and two or more arguments.

## The `NeList{}` construct

One limitation of the `List{}` construct is that it is always possible to write a list of zero elements where a `List{}` is expected. While this is desirable in a number of cases, it is sometimes not what the grammar expects.

For example, in C, it is not allowable for an `enum` definition to have zero members. In other words, if we were to write the grammar for enumerations like
so (`lesson-12-f.k`):

```
module LESSON-12-F
  syntax Id ::= r"[a-zA-Z_][a-zA-Z0-9_]*" [token]
  syntax Exp ::= Id

  syntax EnumSpecifier ::= "enum" Id "{" Ids "}"
  syntax Ids ::= List{Id,","}
endmodule
```

Then we would be syntactically allowed to write `enum X {}`, which instead, ought to be a syntax error.

For this reason, we introduce the additional `NeList{}` construct. The syntax is identical to `List{}`, except with `NeList` instead of `List` before the curly braces. When parsing rules, it behaves identically to the `List{}` construct. However, when parsing inputs to `krun`, the above grammar, if we replaced `syntax Ids ::= List{Id,","}` with `syntax Ids ::= NeList{Id,","}`, would become equivalent to the following (`lesson-12-g.k`):

```
module LESSON-12-G
  syntax Id ::= r"[a-zA-Z_][a-zA-Z0-9_]*" [token]
  syntax Exp ::= Id

  syntax EnumSpecifier ::= "enum" Id "{" Ids "}"
  syntax Ids ::= Id | Id "," Ids
endmodule
```

In other words, only non-empty lists of `Id` would be allowed.

### Exercises

1. Modify the `sum` function in `LESSON-12-C` so that the `Ints` sort is an `NeList{}`. Verify that calling `sum()` with no arguments is now a syntax error.
2. Write a modified `sum` function with the `List` construct that can also sum up an empty list of arguments. In such a case, the sum ought to be 0.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.13: Basics of K Rewriting.

## Lesson 1.13: Basics of K Rewriting

The purpose of this lesson is to explain how rewrite rules that are not the definition of a function behave, and how, using these rules, you can construct a semantics of programs in a programming language in K.

### Recap: Function rules in K

Recall from Lesson 1.2 that we have, thus far,
introduced two types of productions in K: **constructors** and **functions**.
A function is identified by the **function** attribute placed on the production. As you may recall, when we write a rule with a function on the left-hand side of the `=>` operator, we are defining the meaning of that function for inputs which match the patterns on the left-hand side of the rule. If the argument to the function match the patterns, then the function is evaluated to the value constructed by substituting the bindings for the variables into the right-hand side of the rule.

### Top-level rules

However, function rules are not the only type of rule permissible in K, nor even the most frequently used. K also has a concept of a
**top-level rewrite rule**. The simplest way to ensure that a rule is treated

as a top-level rule is for the left-hand side of the rule to mention one or more **cells**. We will cover how cells work and are declared in more detail in a later lesson, but for now, what you should know is that when we ran `krun` in our very first example in Lesson 1.2 and got the following output:

```
<k>
  Yellow ( ) ~> .
</k>
```

`<k>` is a cell, known by convention as the **K cell**. This cell is available by default in any definition without needing to be explicitly declared.

The K cell contains a single term of sort `K`. `K` is a predefined sort in K with two constructors, that can be roughly represented by the following grammar:

```
syntax K ::= KItem "~>" K
           | "."
```

As a syntactic convenience, K allows you to treat `~>` like it is an associative list (i.e., as if it were defined as `syntax K ::= K "~>" K`), but when a definition is compiled, it will automatically transform the rules you write so that they treat the `K` sort as a cons-list. Another syntactic convenience is that, for disambiguation purposes, you can write `.K` anywhere you would otherwise write `.` and the meaning is identical.

Now, you may notice that the above grammar mentions the sort `KItem`. This is another built-in sort in K. For every sort `S` declared in a definition (with the exception of `K` and `KItem`), K will implicitly insert the following production:

```
syntax KItem ::= S
```

In other words, every sort is a subsort of the sort `KItem`, and thus a term of any sort can be injected as an element of a term of sort `K`, also called a **K sequence**.

By default, when you `krun` a program, the AST of the program is inserted as the sole element of a K sequence into the `<k>` cell. This explains why we saw the output we did in Lesson 1.2.

With these preliminaries in mind, we can now explain how top-level rewrite rules work in K. Put simply, any rule where there is a cell (such as the K cell) at the top on the left-hand side will be a top-level rewrite rule. Once the initial program has been inserted into the K cell, the resulting term, called the **configuration**, will be matched against all the top-level rewrite rules in the definition. If only one rule matches, the substitution generated by the matching will be applied to the right-hand side of the rule and the resulting term is **rewritten** to be the new configuration. Rewriting proceeds by iteratively applying rules, also called **taking steps**, until

no top-level rewrite rule can be applied. At this point the configuration becomes the **final configuration** and is output by `krun`.

If more than one top-level rule applies, by default, `K` will pick just one of those rules, apply it, and continue rewriting. However, it is **non-deterministic** which rule applies. In theory, it could be any of them. By passing the `--search` flag to `krun`, you are able to tell `krun` to explore all possible non-deterministic choices, and generate a complete list of all possible final configurations reachable by each nondeterminstic choice that can be made. Note that the `--search` flag to krun only works if you pass `--enable-search` to kompile first.

**Exercise**

Pass a program containing no functions to `krun`. You can use a term of sort `Exp` from `LESSON-11-D`. Observe the output and try to understand why you get the output you do. Then write two rules that rewrite that program to another. Run `krun --search` on that program and observe both results. Then add a third
rule that rewrites one of those results again. Test that that rule applies as well.

## Using top-level rules to evaluate expressions

Thus far, we have focused primarily on defining functions over constructors in K. However, now that we have a basic understanding of top-level rules, it is possible to introduce a **rewrite system** to our definitions. A rewrite system is a collection of top-level rewrite rules which performs an organized transformation of a particular program into a result which expresses the meaning of that program. For example, we might rewrite an expression in a programming language into a value representing the result of evaluating that expression.

Recall in Lesson 1.11, we wrote a simple grammar of Boolean and integer expressions that looked roughly like this (`lesson-13-a.k`):

```
module LESSON-13-A
  imports INT

  syntax Exp ::= Int
               | Bool
               | Exp "+" Exp
               | Exp "&&" Exp
endmodule
```

In that lesson, we defined a function `eval` which evaluated such expressions to either an integer or Boolean.

However, it is more idiomatic to evaluate such expressions using top-level rewrite rules. Here is how one might do so in K (`lesson-13-b.k`):

```
module LESSON-13-B-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Val ::= Int | Bool
  syntax Exp ::= Val
               > left: Exp "+" Exp
               > left: Exp "&&" Exp
endmodule

module LESSON-13-B
  imports LESSON-13-B-SYNTAX
  imports INT
  imports BOOL

  rule <k> I1:Int + I2:Int ~> K:K </k> => <k> I1 +Int I2 ~> K </k>
  rule <k> B1:Bool && B2:Bool ~> K:K </k> => <k> B1 andBool B2 ~> K </k>

  syntax KItem ::= freezer1(Val) | freezer2(Exp)
                 | freezer3(Val) | freezer4(Exp)

  rule <k> E1:Val + E2:Exp ~> K:K </k> => <k> E2 ~> freezer1(E1) ~> K </k> [priority(51)]
  rule <k> E1:Exp + E2:Exp ~> K:K </k> => <k> E1 ~> freezer2(E2) ~> K </k> [priority(52)]
  rule <k> E1:Val && E2:Exp ~> K:K </k> => <k> E2 ~> freezer3(E1) ~> K </k> [priority(51)]
  rule <k> E1:Exp && E2:Exp ~> K:K </k> => <k> E1 ~> freezer4(E2) ~> K </k> [priority(52)]

  rule <k> E2:Val ~> freezer1(E1) ~> K:K </k> => <k> E1 + E2 ~> K </k>
  rule <k> E1:Val ~> freezer2(E2) ~> K:K </k> => <k> E1 + E2 ~> K </k>
  rule <k> E2:Val ~> freezer3(E1) ~> K:K </k> => <k> E1 && E2 ~> K </k>
  rule <k> E1:Val ~> freezer4(E2) ~> K:K </k> => <k> E1 && E2 ~> K </k>
endmodule
```

This is of course rather cumbersome currently, but we will soon introduce syntactic convenience which makes writing definitions of this type considerably easier. For now, notice that there are roughly 3 types of rules here: the first matches a K cell in which the first element of the K sequence is an `Exp` whose arguments are values, and rewrites the first element of the sequence to the result of that expression. The second also matches a K cell with an `Exp` in the first element of its K sequence, but it matches when one or both arguments of the `Exp` are not values, and replaces the first element of the K sequence with two new elements: one being an argument to evaluate, and the other being a special constructor called a **freezer**. Finally, the third matches a K sequence where a `Val` is first, and a freezer is second, and replaces them with a partially evaluated expression.

This general pattern is what is known as **heating** an expression, **evaluating** its arguments, **cooling** the arguments into the expression again, and **evaluating** the expression itself. By repeatedly performing this sequence of actions, we can evaluate an entire AST containing a complex expression down into its resulting value.

**Exercise**

Write an addition expression with integers. Use `krun --depth 1` to see the result of rewriting after applying a single top-level rule. Gradually increase the value of `--depth` to see successive states. Observe how this combination of rules is eventually able to evaluate the entire expression.

## Simplifying the evaluator: Local rewrites and cell ellipses

As you saw above, the definition we wrote is rather cumbersome. Over the remainder of Lessons 1.13 and 1.14, we will greatly simplify it. The first step in doing so is to teach a bit more about the rewrite operator, `=>`. Thus far, all the rules we have written look like `rule LHS => RHS`. However, this is not the only way the rewrite operator can be used. It is actually possible to place a constructor or function at the very top of the rule, and place rewrite operators inside that term. While a rewrite operator cannot appear nested inside another rewrite operator, by doing this, we can express that some parts of what we are matching are not **changed** by the rewrite operator. For example, consider the following rule from above:

```
rule <k> I1:Int + I2:Int ~> K:K </k> => <k> I1 +Int I2 ~> K </k>
```

We can equivalently write it like following:

```
rule <k> (I1:Int + I2:Int => I1 +Int I2) ~> _:K </k>
```

When you put a rewrite inside a term like this, in essence, you are telling the rule to only rewrite **part** of the left-hand side to the right-hand side. In practice, this is implemented by lifting the rewrite operator to the top of the rule by means of duplicating the surrounding context.

There is a way that the above rule can be simplified further, however. K provides a special syntax for each cell containing a term of sort K, indicating that we want to match only on some prefix of the K sequence. For example, the above rule can be simplified further like so:

```
rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
```

Here we have placed the symbol `...` immediately prior to the `</k>` which ends the cell. What this tells the compiler is to take the contents of the cell, treat it as the prefix of a K sequence, and insert an anonymous variable of sort `K` at the end. Thus we can think of `...` as a way of saying we **don't care** about the part of the K sequence after the beginning, leaving it unchanged.

Putting all this together, we can rewrite `LESSON-13-B` like so (`lesson-13-c.k`):

```
module LESSON-13-C-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Val ::= Int | Bool
  syntax Exp ::= Val
               > left: Exp "+" Exp
               > left: Exp "&&" Exp
endmodule

module LESSON-13-C
  imports LESSON-13-C-SYNTAX
  imports INT
  imports BOOL

  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
  rule <k> B1:Bool && B2:Bool => B1 andBool B2 ...</k>

  syntax KItem ::= freezer1(Val) | freezer2(Exp)
                 | freezer3(Val) | freezer4(Exp)

  rule <k> E1:Val + E2:Exp => E2 ~> freezer1(E1) ...</k> [priority(51)]
  rule <k> E1:Exp + E2:Exp => E1 ~> freezer2(E2) ...</k> [priority(52)]
  rule <k> E1:Val && E2:Exp => E2 ~> freezer3(E1) ...</k> [priority(51)]
  rule <k> E1:Exp && E2:Exp => E1 ~> freezer4(E2) ...</k> [priority(52)]

  rule <k> E2:Val ~> freezer1(E1) => E1 + E2 ...</k>
  rule <k> E1:Val ~> freezer2(E2) => E1 + E2 ...</k>
  rule <k> E2:Val ~> freezer3(E1) => E1 && E2 ...</k>
  rule <k> E1:Val ~> freezer4(E2) => E1 && E2 ...</k>
endmodule
```

This is still rather cumbersome, but it is already greatly simplified. In the next lesson, we will see how additional features of K can be used to specify heating and cooling rules much more compactly.

## Exercises

1. Modify `LESSON-13-C` to add rules to evaluate integer subtraction.

## Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.14: Defining Evaluation Order.

# Lesson 1.14: Defining Evaluation Order

The purpose of this lesson is to explain how to use the `heat` and `cool` attributes, `context` and `context alias` sentences, and the `strict` and `seqstrict` attributes to more compactly express heating and cooling in K, and to express more advanced evaluation strategies in K.

## The `heat` and `cool` attributes

Thus far, we have been using rule priority and casts to express when to heat an expression and when to cool it. For example, the rules for heating have lower priority, so they do not apply if the term could be evaluated instead, and the rules for heating are expressly written only to apply if the argument of the expression is a value.

However, K has built-in support for deciding when to heat and when to cool. This support comes in the form of the rule attributes `heat` and `cool` as well as the specially named function `isKResult`.

Consider the following definition, which is equivalent to `LESSON-13-C` (`lesson-14-a.k`):

```
module LESSON-14-A-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int
               | Bool
               > left: Exp "+" Exp
               > left: Exp "&&" Exp
endmodule

module LESSON-14-A
  imports LESSON-14-A-SYNTAX
  imports INT
  imports BOOL

  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
  rule <k> B1:Bool && B2:Bool => B1 andBool B2 ...</k>

  syntax KItem ::= freezer1(Exp) | freezer2(Exp)
                 | freezer3(Exp) | freezer4(Exp)

  rule <k> E:Exp + HOLE:Exp => HOLE ~> freezer1(E) ...</k>
    requires isKResult(E) [heat]
  rule <k> HOLE:Exp + E:Exp => HOLE ~> freezer2(E) ...</k> [heat]
  rule <k> E:Exp && HOLE:Exp => HOLE ~> freezer3(E) ...</k>
```

```
    requires isKResult(E) [heat]
  rule <k> HOLE:Exp && E:Exp => HOLE ~> freezer4(E) ...</k> [heat]

  rule <k> HOLE:Exp ~> freezer1(E) => E + HOLE ...</k> [cool]
  rule <k> HOLE:Exp ~> freezer2(E) => HOLE + E ...</k> [cool]
  rule <k> HOLE:Exp ~> freezer3(E) => E && HOLE ...</k> [cool]
  rule <k> HOLE:Exp ~> freezer4(E) => HOLE && E ...</k> [cool]

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_:Bool) => true
  rule isKResult(_) => false [owise]
endmodule
```

We have introduced three major changes to this definition. First, we have removed the `Val` sort. We replace it instead with a function `isKResult`. The function in question must have the same signature and attributes as seen in
this example. It ought to return `true` whenever a term should not be heated (because it is a value) and `false` when it should be heated (because it is not a value). We thus also insert `isKResult` calls in the side condition of two of the heating rules, where the `Val` sort was previously used.

Second, we have removed the rule priorities on the heating rules and the use of the `Val` sort on the cooling rules, and replaced them with the `heat` and `cool` attributes. These attributes instruct the compiler that these rules are heating and cooling rules, and thus should implicitly apply only when certain terms on the LHS either are or are not a `KResult` (i.e., `isKResult` returns `true` versus `false`).

Third, we have renamed some of the variables in the heating and cooling rules to the special variable `HOLE`. Syntactically, `HOLE` is just a special name for a variable, but it is treated specially by the compiler. By naming a variable `HOLE`, we have informed the compiler which term is being heated or cooled. The compiler will automatically insert the side condition `requires isKResult(HOLE)` to cooling rules and the side condition `requires notBool isKResult(HOLE)` to heating rules.

**Exercise**

Modify `LESSON-14-A` to add rules to evaluate integer subtraction.

## Simplifying further with Contexts

The above example is still rather cumbersome to write. We must explicitly write both the heating and the cooling rule separately, even though they are essentially inverses of one another. It would be nice to instead simply

indicate which terms should be heated and cooled, and what part of them to operate on.

To do this, K introduces a new type of sentence, the **context**. Contexts begin with the `context` keyword instead of the `rule` keyword, and usually do not contain a rewrite operator.

Consider the following definition which is equivalent to `LESSON-14-A` (`lesson-14-b.k`):

```
module LESSON-14-B-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int
               | Bool
               > left: Exp "+" Exp
               > left: Exp "&&" Exp
endmodule

module LESSON-14-B
  imports LESSON-14-B-SYNTAX
  imports INT
  imports BOOL

  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
  rule <k> B1:Bool && B2:Bool => B1 andBool B2 ...</k>

  context <k> E:Exp + HOLE:Exp ...</k>
    requires isKResult(E)
  context <k> HOLE:Exp + _:Exp ...</k>
  context <k> E:Exp && HOLE:Exp ...</k>
    requires isKResult(E)
  context <k> HOLE:Exp && _:Exp ...</k>

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_:Bool) => true
  rule isKResult(_) => false [owise]
endmodule
```

In this example, the `heat` and `cool` rules have been removed entirely, as have been the productions defining the freezers. Don't worry, they still exist under the hood; the compiler is just generating them automatically. For each context sentence like above, the compiler generates a `#freezer` production, a `heat` rule, and a `cool` rule. The generated form is equivalent to the rules we wrote manually in `LESSON-14-A`. However, we are now starting to considerably simplify the definition. Instead of 3 sentences, we just have one.

### `context alias` sentences and the `strict` and `seqstrict` attributes

Notice that the contexts we included in `LESSON-14-B` still seem rather similar in form. For each expression we want to evaluate, we are declaring one context for each operand of that expression, and they are each rather similar to one another. We would like to be able to simplify further by simply annotating each expression production with information about how it is to be evaluated instead. We can do this with the `seqstrict` attribute.

Consider the following definition, once again equivalent to those above (`lesson-14-c.k`):

```
module LESSON-14-C-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int
               | Bool
               > left: Exp "+" Exp [seqstrict(exp; 1, 2)]
               > left: Exp "&&" Exp [seqstrict(exp; 1, 2)]
endmodule

module LESSON-14-C
  imports LESSON-14-C-SYNTAX
  imports INT
  imports BOOL

  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
  rule <k> B1:Bool && B2:Bool => B1 andBool B2 ...</k>

  context alias [exp]: <k> HERE ...</k>

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_:Bool) => true
  rule isKResult(_) => false [owise]
endmodule
```

This definition has two important changes from the one above. The first is that the individual `context` sentences have been removed and have been replaced with a single `context alias` sentence. You may notice that this sentence begins with an identifier in square brackets followed by a colon. This syntax is a way of naming individual sentences in K for reference by the tool or by other sentences. The context alias sentence also has a special variable `HERE`.

The second is that the productions in `LESSON-14-C-SYNTAX` have been given a

`seqstrict` attribute. The value of this attribute has two parts. The first is the name of a context alias sentence. The second is a comma-separated list of integers. Each integer represents an index of a non-terminal in the production, counting from 1. For each integer present, the compiler implicitly generates a new `context` sentence according to the following rules:

1. The compiler starts by looking for the `context alias` sentence named. If there is more than one, then one `context` sentence is created per `context alias` sentence with that name.
2. For each `context` created, the variable `HERE` in the context alias is substituted with an instance of the production the `seqstrict` attribute is attached to. Each child of that production is a variable. The non-terminal indicated by the integer offset of the `seqstrict` attribute is given the name `HOLE`.
3. For each integer offset prior in the list to the one currently being processed, the predicate `isKResult(E)` is conjuncted together and included
   as a side condition, where `E` is the child of the production term with that offset, starting from 1. For example, if the attribute lists `1, 2`, then the rule generated for the `2` will include `isKResult(E1)` where `E1` is the first child of the production.

As you can see if you work through the process, the above code will ultimately generate the same contexts present in `LESSON-14-B`.

Finally, note that there are a few minor syntactic conveniences provided by the `seqstrict` attribute. First, in the special case of the `context alias` sentence being `<k> HERE ...</k>`, you can omit both the `context alias` sentence and the name from the `seqstrict` attribute.

Second, if the numbered list of offsets contains every non-terminal in the production, it can be omitted from the attribute value.

Thus, we can finally produce the idiomatic K definition for this example (`lesson-14-d.k`):

```
module LESSON-14-D-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int
               | Bool
               > left: Exp "+" Exp [seqstrict]
               > left: Exp "&&" Exp [seqstrict]
endmodule

module LESSON-14-D
  imports LESSON-14-D-SYNTAX
  imports INT
```

```
  imports BOOL

  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>
  rule <k> B1:Bool && B2:Bool => B1 andBool B2 ...</k>

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_:Bool) => true
  rule isKResult(_) => false [owise]
endmodule
```

**Exercise**

Modify `LESSON-14-D` to add a production and rule to evaluate integer subtraction.

## Nondeterministic evaluation order with the `strict` attribute

Thus far, we have focused entirely on deterministic evaluation order. However, not all languages are deterministic in the order they evaluate expressions. For example, in C, the expression `a() + b() + c()` is guaranteed to parse to `(a() + b()) + c()`, but it is not guaranteed that `a` will be called before `b` before `c`. In fact, this evaluation order is non-deterministic.

We can express non-deterministic evaluation orders with the `strict` attribute. Its behavior is identical to the `seqstrict` attribute, except that step 3 in the above list (with the side condition automatically added) does not take place. In other words, if we wrote `syntax Exp ::= Exp "+" Exp [strict]` instead of `syntax Exp ::= Exp "+" Exp [seqstrict]`, it would generate the following two contexts instead of the ones found in `LESSON-14-B`:

```
  context <k> _:Exp + HOLE:Exp ...</k>
  context <k> HOLE:Exp + _:Exp ...</k>
```

As you can see, these contexts will generate heating rules that can both apply to the same term. As a result, the choice of which heating rule applies first is non-deterministic, and as we saw in Lesson 1.13, we can get all possible behaviors by passing `--search` to krun.

## Exercises

1. Add integer division to `LESSON-14-D`. Make division and addition `strict` instead of `seqstrict`, and write a rule evaluating integer division with a side condition that the denominator is non-zero. Run `krun --search` on the
   program `1 / 0 + 2 / 1` and observe all possible outputs of the program.

How
many are there total, and why?

2. Modify your solution from lesson 1.11 problem 2 to remove the `eval` function and instead evaluate expressions from left to right using the `seqstrict` attribute.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.15: Configuration Declarations and Cell Nesting.

## Lesson 1.15: Configuration Declarations and Cell Nesting

The purpose of this lesson is to explain how to store additional information about the state of your interpreter by declaring **cells** using the `configuration` sentence, as well as how to add additional inputs to your definition.

### Cells and Configuration Declarations

We have already covered the absolute basics of cells in K by looking at the `<k>` cell. As explained in Lesson 1.13, the
`<k>` cell is available without being explicitly declared. It turns out this is because, if the user does not explicitly specify a `configuration` sentence anywhere in the main module of their definition, the `configuration` sentence from the `DEFAULT-CONFIGURATION` module of
kast.md is imported
automatically. Here is what that sentence looks like:

```
configuration <k> $PGM:K </k>
```

This configuration declaration declares a single cell, the `<k>` cell. It also declares that at the start of rewriting, the contents of that cell should be initialized with the value of the `$PGM` **configuration variable**. Configuration variables function as inputs to `krun`. These terms are supplied to `krun` in the form of ASTs parsed using a particular module. By default, the `$PGM` configuration variable uses the main syntax module of the definition.

The cast on the configuration variable also specifies the sort that is used as the entry point to the parser, in this case the `K` sort. It is often
useful to cast to other sorts there as well for better control over the accepted language. The sort used for the `$PGM` variable is referred to as the start symbol. During parsing, the default start symbol `K` subsumes all user-defined sorts except for syntactic lists. These are excluded because they will always produce an ambiguity error when parsing a single element.

Note that we did not explicitly specify the `$PGM` configuration variable when we invoked `krun` on a file. This is because `krun` handles the `$PGM` variable specially, and allows you to pass the term for that variable via a file passed as a positional argument to `krun`. We did, however, specify the `PGM` name explicitly when we called `krun` with the `-cPGM` command line argument in Lesson 1.2. This is the other, explicit, way of specifying an input to krun.

This explains the most basic use of configuration declarations in K. We can, however, declare multiple cells and multiple configuration variables. We can also specify the initial values of cells statically, rather than dynamically via krun.

For example, consider the following definition (`lesson-15-a.k`):

```
module LESSON-15-A-SYNTAX
  imports INT-SYNTAX

  syntax Ints ::= List{Int,","}
endmodule

module LESSON-15-A
  imports LESSON-15-A-SYNTAX
  imports INT

  configuration <k> $PGM:Ints </k>
                <sum> 0 </sum>

  rule <k> I:Int, Is:Ints => Is ...</k>
       <sum> SUM:Int => SUM +Int I </sum>
endmodule
```

This simple definition takes a list of integers as input and sums them together. Here we have declared two cells: `<k>` and `<sum>`. Unlike `<k>`, `<sum>` does not get initialized via a configuration variable, but instead is initialized statically with the value `0`.

Note the rule in the second module: we have explicitly specified multiple cells in a single rule. K will expect each of these cells to match in order for the rule to apply.

Here is a second example (`lesson-15-b.k`):

```
module LESSON-15-B-SYNTAX
  imports INT-SYNTAX
endmodule

module LESSON-15-B
  imports LESSON-15-B-SYNTAX
```

```
  imports INT
  imports BOOL

  configuration <k> . </k>
                <first> $FIRST:Int </first>
                <second> $SECOND:Int </second>

  rule <k> . => FIRST >Int SECOND </k>
       <first> FIRST </first>
       <second> SECOND </second>
endmodule
```

This definition takes two integers as command-line arguments and populates the
`<k>` cell with a Boolean indicating whether the first integer is greater than the second. Notice that we have specified no `$PGM` configuration variable here. As a result, we cannot invoke `krun` via the syntax `krun $file`. Instead, we must explicitly pass values for each configuration variable via the `-cFIRST` and `-cSECOND` command line flags. For example, if we invoke `krun -cFIRST=0 -cSECOND=1`, we will get the value `false` in the K cell.

You can also specify both a `$PGM` configuration variable and other configuration variables in a single configuration declaration, in which case you would be able to initialize `$PGM` with either a positional argument or the `-cPGM` command line flag, but the other configuration variables would need to be explicitly initialized with `-c`.

**Exercise**

Modify your solution to Lesson 1.14, Problem 2 to add a new cell with a configuration variable of sort `Bool`. This variable should determine whether the `/` operator is evaluated using `/Int` or `divInt`. Test that by specifying different values for this variable, you can change the behavior of rounding on division of negative numbers.

## Cell Nesting

It is possible to nest cells inside one another. A cell that contains other cells must contain **only** other cells, but in doing this, you are able to create a hierarchical structure to the configuration. Consider the following definition which is equivalent to the one in `LESSON-15-B` (`lesson-15-c.k`):

```
module LESSON-15-C-SYNTAX
  imports INT-SYNTAX
endmodule

module LESSON-15-C
  imports LESSON-15-C-SYNTAX
```

```
  imports INT
  imports BOOL

  configuration <T>
                  <k> . </k>
                  <state>
                    <first> $FIRST:Int </first>
                    <second> $SECOND:Int </second>
                  </state>
                </T>

  rule <k> . => FIRST >Int SECOND </k>
       <first> FIRST </first>
       <second> SECOND </second>
endmodule
```

Note that we have added some new cells to the configuration declaration: the `<T>` cell wraps the entire configuration, and the `<state>` cell is introduced around the `<first>` and `<second>` cells.

However, we have not changed the rule in this definition. This is because of a concept in K called **configuration abstraction**. K allows you to specify any number of cells in a rule (except zero) in any order you want, and K will compile the rules into a form that matches the structure of the configuration specified by the configuration declaration.

Here then, is how this rule would look after the configuration abstraction has been resolved:

```
  rule <T>
         <k> . => FIRST >Int SECOND </k>
         <state>
           <first> FIRST </first>
           <second> SECOND </second>
         </state>
       </T>
```

In other words, K will complete cells to the top of the configuration by inserting parent cells where appropriate based on the declared structure of the configuration. This is useful because as a definition evolves, the configuration may change, but you don't want to have to modify every single rule each time. Thus, K follows the principle that you should only mention the cells in a rule that are actually needed in order to accomplish its specific goal. By following this best practice, you can significantly increase the modularity of the definition and make it easier to maintain and modify.

**Exercise**

Modify your definition from the previous exercise in this lesson to wrap the two cells you have declared in a top cell `<T>`. You should not have to change any other rules in the definition.

## Cell Variables

Sometimes it is desirable to explicitly match a variable against certain fragments of the configuration. Because K's configuration is hierarchical, we can grab subsets of the configuration as if they were just another term. However, configuration abstraction applies here as well.
In particular, for each cell you specify in a configuration declaration, a unique sort is assigned for that cell with a single constructor (the cell itself). The sort name is taken by removing all special characters, capitalizing the first letter and each letter after a hyphen, and adding the word `Cell` at the end. For example, in the above example, the cell sorts are `TCell`, `KCell`, `StateCell`, `FirstCell`, and `SecondCell`. If we had declared a cell as `<first-number>`, then the cell sort name would be `FirstNumberCell`.

You can explicitly reference a variable of one of these sorts anywhere you might instead write that cell. For example, consider the following rule:

```
rule <k> true => S </k>
     (S:StateCell => <state>... .Bag ...</state>)
```

Here we have introduced two new concepts. The first is the variable of sort `StateCell`, which matches the entire `<state>` part of the configuration. The second is that we have introduced the concept of `...` once again. When a cell contains other cells, it is also possible to specify `...` on either the left, right or both sides of the cell term. Each of these three syntaxes are equivalent in this case. When they appear on the left-hand side of a rule, they indicate that we don't care what value any cells not explicitly named might have. For example, we might write `<state>... <first> 0 </first> ...</state>` on
the left-hand side of a rule in order to indicate that we want to match the rule when the `<first>` cell contains a zero, regardless of what the `<second>` cell contains. If we had not included this ellipsis, it would have been a syntax error, because K would have expected you to provide a value for each of the child cells.

However, if, as in the example above, the `...` appeared on the right-hand side of a rule, this instead indicates that the cells not explicitly mentioned under the cell should be initialized with their default value from the configuration declaration. In other words, that rule will set the value of `<first>` and `<second>` to zero.

You may note the presence of the phrase `.Bag` here. You can think of this as the empty set of cells. It is used as the child of a cell when you want to

indicate that no cells should be explicitly named. We will cover other uses of this term in later lessons.

### Exercises

1. Modify the definition from the previous exercise in this lesson so that the Boolean cell you created is initialized to false. Then add a production `syntax Stmt ::= Bool ";" Exp`, and a rule that uses this `Stmt` to set the
value of the Boolean flag. Then add another production
`syntax Stmt ::= "reset" ";" Exp` which sets the value of the Boolean flag back
to its default value via a `...` on the right-hand side. You will need to add an additional cell around the Boolean cell to make this work.

### Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.16: Maps, Semantic Lists, and Sets.

## Lesson 1.16: Maps, Semantic Lists, and Sets

The purpose of this lesson is to explain how to use the data structure sorts provided by K: maps, lists, and sets.

### Maps

The most frequently used type of data structure in K is the map. The sort provided by K for this purpose is the `Map` sort, and it is provided in domains.md in the `MAP`
module. This type is not (currently) polymorphic. All `Map` terms are maps that map terms of sort `KItem` to other terms of sort `KItem`. A `KItem` can contain any sort except a `K` sequence. If you need to store such a term in a map, you can always use a wrapper such as `syntax KItem ::= kseq(K)`.

A `Map` pattern consists of zero or more map elements (as represented by the symbol `syntax Map ::= KItem "|->" KItem`), mixed in any order, separated by
whitespace, with zero or one variables of sort `Map`. The empty map is represented by `.Map`. If all of the bindings for the variables in the keys of the map can be deterministically chosen, these patterns can be matched in `O(1)` time. If they cannot, then each map element that cannot be deterministically constructed contributes a single dimension of polynomial time to the cost of the matching. In other words, a single such element is linear, two are quadratic, three are cubic, etc.

Patterns like the above are the only type of `Map` pattern that can appear on the left-hand-side of a rule. In other words, you are not allowed to write a `Map` pattern on the left-hand-side with more than one variable of sort `Map` in it. You are, however, allowed to write such patterns on the right-hand-side of a rule. You can also write a function pattern in the key of a map element so long as all the variables in the function pattern can be deterministically chosen.

Note the meaning of matching on a `Map` pattern: a map pattern with no variables of sort `Map` will match if the map being matched has exactly as many bindings as `|->` symbols in the pattern. It will then match if each binding in the map pattern matches exactly one distinct binding in the map being matched. A map pattern with one `Map` variable will also match any map that contains such a map as a subset. The variable of sort `Map` will be bound to whatever bindings are left over (`.Map` if there are no bindings left over).

Here is an example of a simple definition that implements a very basic variable declaration semantics using a `Map` to store the value of variables (`lesson-16-a.k`):

```
module LESSON-16-A-SYNTAX
  imports INT-SYNTAX
  imports ID-SYNTAX

  syntax Exp ::= Id | Int
  syntax Decl ::= "int" Id "=" Exp ";" [strict(2)]
  syntax Pgm ::= List{Decl,""}
endmodule

module LESSON-16-A
  imports LESSON-16-A-SYNTAX
  imports BOOL

  configuration <T>
                  <k> $PGM:Pgm </k>
                  <state> .Map </state>
                </T>

  // declaration sequence
  rule <k> D:Decl P:Pgm => D ~> P ...</k>
  rule <k> .Pgm => . ...</k>

  // variable declaration
  rule <k> int X:Id = I:Int ; => . ...</k>
       <state> STATE => STATE [ X <- I ] </state>

  // variable lookup
```

```
  rule <k> X:Id => I ...</k>
       <state>... X |-> I ...</state>

  syntax Bool ::= isKResult(K) [symbol, function]
  rule isKResult(_:Int) => true
  rule isKResult(_) => false [owise]
endmodule
```

There are several new features in this definition. First, note we import the module `ID-SYNTAX`. This module is defined in `domains.md` and provides a basic syntax for identifiers. We are using the `Id` sort provided by this module in this definition to implement the names of program variables. This syntax is only imported when parsing programs, not when parsing rules. Later in
this lesson we will see how to reference specific concrete identifiers in a rule.

Second, we introduce a single new function over the `Map` sort. This function, which is represented by the symbol
`syntax Map ::= Map "[" KItem "<-" KItem "]"`, represents the map update operation. Other functions over the `Map` sort can be found in `domains.md`.

Finally, we have used the `...` syntax on a cell containing a Map. In this case, the meaning of `<state>... Pattern ...</state>`,
`<state>... Pattern </state>`, and `<state> Pattern ...</state>` are the same:
it is equivalent to writing `<state> (Pattern) _:Map </state>`.

Consider the following program (`a.decl`):

```
int x = 0;
int y = 1;
int a = x;
```

If we run this program with `krun`, we will get the following result:

```
<T>
  <k>
    .
  </k>
  <state>
    a |-> 0
    x |-> 0
    y |-> 1
  </state>
</T>
```

Note that `krun` has automatically sorted the collection for you. This doesn't happen at runtime, so you still get the performance of a hash map, but it will help make the output more readable.

**Exercise**

Create a sort `Stmt` that is a subsort of `Decl`. Create a production of sort `Stmt` for variable assignment in addition to the variable declaration production. Feel free to use the syntax `syntax Stmt ::= Id "=" Exp ";"`. Write
a rule that implements variable assignment using a map update function. Then write the same rule using a map pattern. Test your implementations with some programs to ensure they behave as expected.

## Semantic Lists

In a previous lesson, we explained how to represent lists in the AST of a program. However, this is not the only context where lists can be used. We also frequently use lists in the configuration of an interpreter in order to represent certain types of program state. For this purpose, it is generally useful to have an associative-list sort, rather than the cons-list sorts provided in Lesson 1.12.

The type provided by K for this purpose is the `List` sort, and it is also provided in `domains.md`, in the `LIST` module. This type is also not (currently) polymorphic. Like `Map`, all `List` terms are lists of terms of the `KItem` sort.

A `List` pattern in K consists of zero or more list elements (as represented by the `ListItem` symbol), followed by zero or one variables of sort `List`, followed by zero or more list elements. An empty list is represented by `.List`. These patterns can be matched in `O(log(N))` time. This is the only type of `List` pattern that can appear on the left-hand-side of a rule. In other words, you are not allowed to write a `List` pattern on the left-hand-side with more than one variable of sort `List` in it. You are, however, allowed to write such patterns on the right-hand-side of a rule.

Note the meaning of matching on a `List` pattern: a list pattern with no variables of sort `List` will match if the list being matched has exactly as many elements as `ListItem` symbols in the pattern. It will then match if each element in sequence matches the pattern contained in the `ListItem` symbol. A list pattern with one variable of sort `List` operates the same way, except that it can match any list with at least as many elements as `ListItem` symbols, so long as the prefix and suffix of the list match the patterns inside the `ListItem` symbols. The variable of sort `List` will be bound to whatever elements are left over (`.List` if there are no elements left over).

The `...` syntax is allowed on cells containing lists as well. In this case, the meaning of `<cell>... Pattern </cell>` is the same as `<cell> _:List (Pattern) </cell>`, the meaning of `<cell> Pattern ...</cell>`
is the same as `<cell> (Pattern) _:List</cell>`. Because list patterns with

multiple variables of sort `List` are not allowed, it is an error to write `<cell>... Pattern ...</cell>`.

Here is an example of a simple definition that implements a very basic function-call semantics using a `List` as a function stack (`lesson-16-b.k`):

```
module LESSON-16-B-SYNTAX
  imports INT-SYNTAX
  imports ID-SYNTAX

  syntax Exp ::= Id "(" ")" | Int
  syntax Stmt ::= "return" Exp ";" [strict]
  syntax Decl ::= "fun" Id "(" ")" "{" Stmt "}"
  syntax Pgm ::= List{Decl,""}
  syntax Id ::= "main" [token]
endmodule

module LESSON-16-B
  imports LESSON-16-B-SYNTAX
  imports BOOL
  imports LIST

  configuration <T>
                  <k> $PGM:Pgm ~> main () </k>
                  <functions> .Map </functions>
                  <fstack> .List </fstack>
                </T>

  // declaration sequence
  rule <k> D:Decl P:Pgm => D ~> P ...</k>
  rule <k> .Pgm => . ...</k>

  // function definitions
  rule <k> fun X:Id () { S } => . ...</k>
       <functions>... .Map => X |-> S ...</functions>

  // function call
  syntax KItem ::= stackFrame(K)
  rule <k> X:Id () ~> K => S </k>
       <functions>... X |-> S ...</functions>
       <fstack> .List => ListItem(stackFrame(K)) ...</fstack>

  // return statement
  rule <k> return I:Int ; ~> _ => I ~> K </k>
       <fstack> ListItem(stackFrame(K)) => .List ...</fstack>

  syntax Bool ::= isKResult(K) [function, symbol]
```

```
  rule isKResult(_:Int) => true
  rule isKResult(_) => false [owise]
endmodule
```

Notice that we have declared the production `syntax Id ::= "main" [token]`. Since we use the `ID-SYNTAX` module, this declaration is necessary in order to be able to refer to the `main` identifier directly in the configuration declaration. Our `<k>` cell now contains a `K` sequence initially: first we process all the declarations in the program, then we call the `main` function.

Consider the following program (`foo.func`):

```
fun foo() { return 5; }
fun main() { return foo(); }
```

When we `krun` this program, we should get the following output:

```
<T>
  <k>
    5 ~> .
  </k>
  <functions>
    foo |-> return 5 ;
    main |-> return foo ( ) ;
  </functions>
  <fstack>
    .List
  </fstack>
</T>
```

Note that we have successfully put on the `<k>` cell the value returned by the `main` function.

**Exercise**

Add a term of sort `Id` to the `stackFrame` operator to keep track of the name of the function in that stack frame. Then write a function `syntax String ::= printStackTrace(List)` that takes the contents of the `<fstack>` cell and pretty prints the current stack trace. You can concatenate strings with `+String` in the `STRING` module in `domains.md`, and you can convert an `Id` to a `String` with the `Id2String` function in the `ID` module. Test this function by creating a new expression that returns the current stack trace as a string. Make sure to update `isKResult` and the `Exp` sort as appropriate to allow strings as values.

## Sets

The final primary data structure sort in K is a set, i.e., an idempotent unordered collection where elements are deduplicated. The sort provided by K

for this purpose is the `Set` sort and it is provided in `domains.md` in the `SET` module. Like maps and lists, this type is not (currently) polymorphic. Like `Map` and `List`, all `Set` terms are sets of terms of the `KItem` sort.

A `Set` pattern has the exact same restrictions as a `Map` pattern, except that its elements are treated like keys, and there are no values. It has the same performance characteristics as well. However, syntactically it is more similar to the `List` sort: An empty `Set` is represented by `.Set`, but a set element is represented by the `SetItem` symbol.

Matching behaves similarly to the `Map` sort: a set pattern with no variables of sort `Set` will match if the set has exactly as many bindings as `SetItem` symbols, and if each element pattern matches one distinct element in the set. A set with a variable of sort `Set` also matches any superset of such a set. As with map, the elements left over will be bound to the `Set` variable (or `.Set` if no elements are left over).

Like `Map`, the `...` syntax on a set is syntactic sugar for an anonymous variable of sort `Set`.

Here is an example of a simple modification to `LESSON-16-A` which uses a `Set` to ensure that variables are never declared more than once. In practice, you would likely just use the `in_keys` symbol over maps to test for this, but it's still useful as an example of sets in practice:

```
module LESSON-16-C-SYNTAX
  imports LESSON-16-A-SYNTAX
endmodule

module LESSON-16-C
  imports LESSON-16-C-SYNTAX
  imports BOOL
  imports SET

  configuration <T>
                  <k> $PGM:Pgm </k>
                  <state> .Map </state>
                  <declared> .Set </declared>
                </T>

  // declaration sequence
  rule <k> D:Decl P:Pgm => D ~> P ...</k>
  rule <k> .Pgm => . ...</k>

  // variable declaration
  rule <k> int X:Id = I:Int ; => . ...</k>
       <state> STATE => STATE [ X <- I ] </state>
       <declared> D => D SetItem(X) </declared>
```

```
    requires notBool X in D

  // variable lookup
  rule <k> X:Id => I ...</k>
       <state>... X |-> I ...</state>
       <declared>... SetItem(X) ...</declared>

  syntax Bool ::= isKResult(K) [symbol, function]
  rule isKResult(_:Int) => true
  rule isKResult(_) => false [owise]
endmodule
```

Now if we `krun` a program containing duplicate declarations, it will get stuck on the declaration.

### Exercises

1. Modify your solution to Lesson 1.14, Problem 2 and introduce the sorts `Decls`, `Decl`, and `Stmt` which include variable and function declaration (without function parameters), and return and assignment statements, as well
as call expressions. Use `List` and `Map` to implement these operators, making
sure to consider the interactions between components, such as saving and restoring the environment of variables at each call site. Don't worry about local function definitions or global variables for now. Make sure to test the
resulting interpreter.

### Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.17: Cell Multiplicity and Cell Collections.

## Lesson 1.17: Cell Multiplicity and Cell Collections

The purpose of this lesson is to explain how you can create optional cells and cells that repeat multiple times in a configuration using a feature called **cell multiplicity**.

### Cell Multiplicity

K allows you to specify attributes for cell productions as part of the syntax of configuration declarations. Unlike regular productions, which use the `[]` syntax for attributes, configuration cells use an XML-like attribute syntax:

```
configuration <k color="red"> $PGM:K </k>
```

This configuration declaration gives the `<k>` cell the color red during unparsing using the `color` attribute as discussed in Lesson 1.9.

However, in addition to the usual attributes for productions, there are some other attributes that can be applied to cells with special meaning. One such attribute is the `multiplicity` attribute. By default, each cell that is declared occurs exactly once in every configuration term. However, using the `multiplicity` attribute, this default behavior can be changed. There are two values that this attribute can have: `?` and `*`.

## Optional cells

The first cell multiplicity we will discuss is `?`. Similar to a regular expression language, this attribute tells the compiler that this cell can appear 0 or 1 times in the configuration. In other words, it is an **optional cell**. By default, K does not create optional cells in the initial configuration, unless that optional cell has a configuration variable inside it. However, it is possible to override the default behavior and create that cell initially by adding the additional cell attribute `initial=""`.

K uses the `.Bag` symbol to represent the absence of any cells in a particular rule. Consider the following module:

```
module LESSON-17-A
  imports INT

  configuration <k> $PGM:K </k>
                <optional multiplicity="?"> 0 </optional>

  syntax KItem ::= "init" | "destroy"

  rule <k> init => . ...</k>
       (.Bag => <optional> 0 </optional>)
  rule <k> destroy => . ...</k>
       (<optional> _ </optional> => .Bag)

endmodule
```

In this definition, when the `init` symbol is executed, the `<optional>` cell is added to the configuration, and when the `destroy` symbol is executed, it is removed. Any rule that matches on that cell will only match if that cell is present in the configuration.

**Exercise**

Create a simple definition with a `Stmts` sort that is a `List{Stmt,""}` and a `Stmt` sort with the constructors `syntax Stmt ::= "enable" | "increment" | "decrement" | "disable"`. The configuration should have an optional cell that contains an integer that is created with the `enable` command, destroyed with the `disable` command, and its value is incremented or decremented by the `increment` and `decrement` command.

## Cell collections

The second type of cell multiplicity we will discuss is `*`. Simlar to a regular expression language, this attribute tells the compiler that this cell can appear 0 or more times in the configuration. In other words, it is a **cell collection**. Cells with multiplicity `*` must be the only child of their parent cell. As a convention, the inner cell is usually named with the singular form of what it contains, and the outer cell with the plural form, for example, "thread" and "threads".

All cell collections are required to have the `type` attribute set to either `Set` or `Map`. A `Set` cell collection is represented as a set and behaves internally the same as the `Set` sort, although it actually declares a new sort. A `Map` cell collection is represented as a `Map` in which the first subcell of the cell collection is the key and the remaining cells are the value.

For example, consider the following module:

```
module LESSON-17-B
  imports INT
  imports BOOL
  imports ID-SYNTAX

  syntax Stmt ::= Id "=" Exp ";" [strict(2)]
                | "return" Exp ";" [strict]
  syntax Stmts ::= List{Stmt,""}
  syntax Exp ::= Id
               | Int
               | Exp "+" Exp [seqstrict]
               | "spawn" "{" Stmts "}"
               | "join" Exp ";" [strict]

  configuration <threads>
                  <thread multiplicity="*" type="Map">
                    <id> 0 </id>
                    <k> $PGM:K </k>
```

```
                </thread>
              </threads>
              <state> .Map </state>
              <next-id> 1 </next-id>

  rule <k> X:Id => I:Int ...</k>
       <state>... X |-> I ...</state>
  rule <k> X:Id = I:Int ; => . ...</k>
       <state> STATE => STATE [ X <- I ] </state>
  rule <k> S:Stmt Ss:Stmts => S ~> Ss ...</k>
  rule <k> I1:Int + I2:Int => I1 +Int I2 ...</k>

  rule <thread>...
         <k> spawn { Ss } => NEXTID ...</k>
       ...</thread>
       <next-id> NEXTID => NEXTID +Int 1 </next-id>
       (.Bag =>
       <thread>
         <id> NEXTID </id>
         <k> Ss </k>
       </thread>)

  rule <thread>...
         <k> join ID:Int ; => I ...</k>
       ...</thread>
       (<thread>
         <id> ID </id>
         <k> return I:Int ; ...</k>
       </thread> => .Bag)

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_) => false [owise]
endmodule
```

This module implements a very basic fork/join semantics. The `spawn` expression spawns a new thread to execute a sequence of statements and returns a thread id, and the `join` statement waits until a thread executes `return` and then returns the return value of the thread.

Note something quite novel here: the `<k>` cell is inside a cell of multiplicity `*`. Since the `<k>` cell is just a regular cell (mostly), this is perfectly allowable. Rules that don't mention a specific thread are automatically completed to match any thread.

When you execute programs in this language, the cells in the cell collection get sorted and printed like any other collection, but they still display like

cells. Rules in this language also benefit from all the structural power of cells, allowing you to omit cells you don't care about or complete the configuration automatically. This allows you to have the power of cells while still being a collection under the hood.

### Exercises

1. Modify the solution from Lesson 1.16, Problem 1 so that the cell you use to
   keep track of functions in a `Map` is now a cell collection. Run some programs and compare how they get unparsed before and after this change.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.18: Term Equality and the Ternary Operator.

## Lesson 1.18: Term Equality and the Ternary Operator

The purpose of this lesson is to introduce how to compare equality of terms in K, and how to put conditional expressions directly into the right-hand side of rules.

### Term Equality

One major way you can compare whether two terms are equal in K is to simply match both terms with a variable with the same name. This will only succeed in matching if the two terms are equal structurally. However, sometimes this is impractical, and it is useful to have access to a way to actually compare whether two terms in K are equal. The operator for this is found in domains.md in the `K-EQUAL`
module. The operator is `==K` and takes two terms of sort `K` and returns a `Bool`. It returns true if they are equal. This includes equality over builtin types such as `Map` and `Set` where equality is not purely structural in nature. However, it does not include any notion of semantic equality over user-defined syntax. The inverse symbol for inequality is `=/=K`.

### Ternary Operator

One way to introduce conditional logic in K is to have two separate rules, each with a side condition (or one rule with a side condition and another with the `owise` attribute). However, sometimes it is useful to explicitly write a conditional expression directly in the right-hand side of a rule. For this purpose, K defines one more operator in the `K-EQUAL` module, which corresponds

to the usual ternary operator found in many languages. Here is an example of its
usage (`lesson-18.k`):

```
module LESSON-18
  imports INT
  imports BOOL
  imports K-EQUAL

  syntax Exp ::= Int | Bool | "if" "(" Exp ")" Exp "else" Exp [strict(1)]

  syntax Bool ::= isKResult(K) [function, symbol]
  rule isKResult(_:Int) => true
  rule isKResult(_:Bool) => true

  rule if (B:Bool) E1:Exp else E2:Exp => #if B #then E1 #else E2 #fi
endmodule
```

Note the symbol on the right-hand side of the final rule. This symbol is polymorphic: `B` must be of sort `Bool`, but `E1` and `E2` could have been any sort so long as both were of the same sort, and the sort of the entire expression becomes equal to that sort. K supports polymorphic built-in operators, but does not yet allow users to write their own polymorphic productions.

The behavior of this function is to evaluate the Boolean expression to a Boolean, then pick one of the two children and return it based on whether the Boolean is true or false. Please note that it is not a good idea to use this symbol in cases where one or both of the children is potentially undefined (for example, an integer expression that divides by zero). While the default implementation is smart enough to only evaluate the branch that happens to be
picked, this will not be true when we begin to do program verification. If you need short circuiting behavior, it is better to use a side condition.

## Exercises

1. Write a function in K that takes two terms of sort `K` and returns an `Int`: the `Int` should be 0 if the terms are equal and 1 if the terms are unequal.
2. Modify your solution to Lesson 1.16, Problem 1 and introduce an `if Stmt` to the syntax of the language, then implement it using the `#if` symbol.
   Make sure to write tests for the resulting interpreter.

### Next lesson

Once you have completed the above exercises, you can continue to Lesson 1.19: Debugging with GDB.

## Lesson 1.19: Debugging with GDB

The purpose of this lesson is to teach how to debug your K interpreter using the K-language support provided in GDB.

### Getting started

You will need GDB in order to complete this lesson. If you do not already have GDB installed, then do so. Steps to install GDB are outlined in this GDB Tutorial.

The first thing neccessary in order to debug a K interpreter in GDB is to build the interpreter with full debugging support enabled. This can be done relatively simply. First, make sure you have not passed `-O1`, `-O2`, or `-O3` to `kompile`. Second, simply add the command line flags `-ccopt -g -ccopt -O1` to `kompile`. The resulting compiled K definition will be ready to support debugging.

Note: the 'O' in `-O1` is the letter 'O' not the number 0!

Once you have a compiled K definition and a program you wish to debug, you can start the debugger by passing the `--debugger` flag to `krun`. This will automatically load the program you are executing into GDB and drop you into a GDB shell ready to start executing the program.

As an example, consider the following K definition (`lesson-19-a.k`):

```
module LESSON-19-A
  imports INT

  rule I => I +Int 1
    requires I <Int 100
endmodule
```

If we compile this definition with `kompile lesson-19-a.k -ccopt -g -ccopt -O1`, and run the program `0` in the debugger with `krun -cPGM=0 --debugger`, we will see the following output (roughly):

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./lesson-19-a-kompiled/interpreter...
warning: File "/home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_debugg
To enable execution of this file add
        add-auto-load-safe-path /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/
line to your configuration file "/home/dwightguth/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/dwightguth/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

To make full advantage of the GDB features of K, you should follow the first command listed in this output message and add the corresponding `add-auto-load-safe-path` command to your `~/.gdbinit` file as prompted. Please note that the path will be different on your machine than the one listed above. Adding directories to the ”load safe path” effectively tells GDB to trust those directories. All content under a given directory will be recursively trusted, so if you want to avoid having to add paths to the ”load safe path” every time you kompile a different `K` definition, then you can just trust a minimal directory containing all your kompiled files; however, do not choose a top-level directory containing arbitrary files as this amounts to trusting arbitrary files and is a security risk. More info on the load safe path can be found here.

## Basic commands

The most basic commands you can execute in the K GDB session are to run your program or to step through it. The first can be accomplished using GDB's built-in `run` command. This will automatically start the program and begin executing it. It will continue until the program aborts or finishes, or the debugger is interrupted with Ctrl-C.

Sometimes you want finer-grained control over how you proceed through the program you are debugging. To step through the rule applications in your program, you can use the `k start` and `k step` GDB commands.

`k start` is similar to the built-in `start` command in that it starts the program and then immediately breaks before doing any work. However, unlike the `start` command which will break immediately after the `main` method of a program is executed, the `K start` program will initialize the rewriter, evaluate the initial configuration, and break immediately prior to applying any rewrite steps.

In the example above, here is what we see when we run the `k start` command:

```
Temporary breakpoint 1 at 0x239210
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x0000000000239210 in main ()
0x0000000000231890 in step (subject=<k>
  0 ~> .
</k>)
(gdb)
```

As you can see, we are stopped at the `step` function in the interpreter. This function is responsible for taking top-level rewrite steps. The `subject` parameter to this function is the current K configuration.

We can step through K rewrite steps one at a time by running the `k step` command. By default, this takes a single rewrite step (including any function rule applications that are part of that step).

Here is what we see when we run that command:

```
Continuing.

Temporary breakpoint -22, 0x0000000000231890 in step (subject=<k>
  1 ~> .
</k>)
(gdb)
```

As we can see, we have taken a single rewrite step. We can also pass a number to the `k step` command which indicates the number of rewrite steps to take.

Here is what we see if we run `k step 10`:

```
Continuing.

Temporary breakpoint -23, 0x0000000000231890 in step (subject=<k>
  11 ~> .
```

```
</k>)
(gdb)
```

As we can see, ten rewrite steps were taken.

## Breakpoints

The next important step in debugging an application in GDB is to be able to set breakpoints. Generally speaking, there are three types of breakpoints we are interested in in a K semantics: Setting a breakpoint when a particular function is called, setting a breakpoint when a particular rule is applied, and setting a breakpoint when a side condition of a rule is evaluated.

The easiest way to do the first two things is to set a breakpoint on the line of code containing the function or rule.

For example, consider the following K definition (`lesson-19-b.k`):

```
module LESSON-19-B
  imports BOOL

  syntax Bool ::= isBlue(Fruit) [function]
  syntax Fruit ::= Blueberry() | Banana()
  rule isBlue(Blueberry()) => true
  rule isBlue(Banana()) => false

  rule F:Fruit => isBlue(F)
endmodule
```

Once this program has been compiled for debugging, we can run the program `Blueberry()`. We can then set a breakpoint that stops when the `isBlue` function is called with the following command:

```
break lesson-19-b.k:4
```

Here is what we see if we set this breakpoint and then run the interpreter:

```
(gdb) break lesson-19-b.k:4
Breakpoint 1 at 0x231040: file /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1
(gdb) run
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, LblisBlue'LParUndsRParUnds'LESSON-19-B'Unds'Bool'Unds'Fruit (_1=Blueberry ( ))
4         syntax Bool ::= isBlue(Fruit) [function]
(gdb)
```

As we can see, we have stopped at the point where we are evaluating that function. The value `_1` that is a parameter to that function shows the

value passed to the function by the caller.

We can also break when the `isBlue(Blueberry()) => true` rule applies by simply
changing the line number to the line number of that rule:

```
(gdb) break lesson-19-b.k:6
Breakpoint 1 at 0x2af710: file /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1
(gdb) run
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, apply_rule_138 () at /home/dwightguth/kframework-5.0.0/k-distribution/k-tutori
6         rule isBlue(Blueberry()) => true
(gdb)
```

We can also do the same with a top-level rule:

```
(gdb) break lesson-19-b.k:9
Breakpoint 1 at 0x2aefa0: lesson-19-b.k:9. (2 locations)
(gdb) run
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, apply_rule_107 (Var'Unds'DotVar0=<generatedCounter>
  0
</generatedCounter>, Var'Unds'DotVar1=., VarF=Blueberry ( )) at /home/dwightguth/kframework-
9         rule F:Fruit => isBlue(F)
(gdb)
```

Unlike the function rule above, we see several parameters to this function. These are the substitution that was matched for the function. Variables only appear in this substitution if they are actually used on the right-hand side of the rule.

## Advanced breakpoints

Sometimes it is inconvenient to set the breakpoint based on a line number.

It is also possible to set a breakpoint based on the rule label of a particular rule. Consider the following definition (`lesson-19-c.k`):

```
module LESSON-19-C
  imports INT
  imports BOOL

  syntax Bool ::= isEven(Int) [function]
```

```
  rule [isEven]: isEven(I) => true requires I %Int 2 ==Int 0
  rule [isOdd]: isEven(I) => false requires I %Int 2 =/=Int 0

endmodule
```

We will run the program `isEven(4)`. We can set a breakpoint for when a rule applies by means of the `MODULE-NAME.label.rhs` syntax:

```
(gdb) break LESSON-19-C.isEven.rhs
Breakpoint 1 at 0x2afda0: file /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1
(gdb) run
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, LESSON-19-C.isEven.rhs () at /home/dwightguth/kframework-5.0.0/k-distribution/
6         rule [isEven]: isEven(I) => true requires I %Int 2 ==Int 0
(gdb)
```

We can also set a breakpoint for when a rule's side condition is evaluated by means of the `MODULE-NAME.label.sc` syntax:

```
(gdb) break LESSON-19-C.isEven.sc
Breakpoint 1 at 0x2afd70: file /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1
(gdb) run
Starting program: /home/dwightguth/kframework-5.0.0/k-distribution/k-tutorial/1_basic/19_deb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, LESSON-19-C.isEven.sc (VarI=4) at /home/dwightguth/kframework-5.0.0/k-distribu
6         rule [isEven]: isEven(I) => true requires I %Int 2 ==Int 0
(gdb) finish
Run till exit from #0  LESSON-19-C.isEven.sc (VarI=4) at /home/dwightguth/kframework-5.0.0/k
0x00000000002b2662 in LblisEven'LParUndsRParUnds'LESSON-19-C'Unds'Bool'Unds'Int (_1=4) at /h
5         syntax Bool ::= isEven(Int) [function]
Value returned is $1 = true
(gdb)
```

Here we have used the built-in GDB command `finish` to tell us whether the side condition returned true or not. Note that once again, we see the substitution that was matched from the left-hand side. Like before, a variable will only appear here if it is used in the side condition.

## Debugging rule matching

Sometimes it is useful to try to determine why a particular rule did or did not apply. K provides some basic debugging commands which make it easier to determine this.

Consider the following K definition (`lesson-19-d.k`):

```
module LESSON-19-D

  syntax Foo ::= foo(Bar)
  syntax Bar ::= bar(Baz) | bar2(Baz)
  syntax Baz ::= baz() | baz2()

  rule [baz]: foo(bar(baz())) => .K

endmodule
```

Suppose we try to run the program `foo(bar(baz2()))`. It is obvious from this example why the rule in this definition will not apply. However, in practice, such cases are not always obvious. You might look at a rule and not immediately
spot why it didn't apply on a particular term. For this reason, it can be useful to get the debugger to provide a log about how it tried to match that term. You can do this with the `k match` command. If you are stopped after having run `k start` or `k step`, you can obtain this log for any rule after any step by running the command `k match MODULE.label subject` for a particular
top-level rule label.

For example, with the `baz` rule above, we get the following output:

```
(gdb) k match LESSON-19-D.baz subject
Subject:
baz2 ( )
does not match pattern:
baz ( )
```

As we can see, it provided the exact subterm which did not match against the rule, as well as the particular subpattern it ought to have matched against.

This command does not actually take any rewrite steps. In the event that matching actually succeeds, you will still need to run the `k step` command to advance to the next step.

## Final notes

In addition to the functionality provided above, you have the full power of GDB at your disposal when debugging. Some features are not particularly well-adapted to K code and may require more advanced knowledge of the term representation or implementation to use effectively, but anything that can be done in GDB can in theory be done using this debugging functionality. We suggest you refer to the
GDB Documentation if you
want to try to do something and are unsure as to how.

### Exercises

1. Compile your solution to Lesson 1.18, Problem 2 with debugging support enabled and step through several programs you have previously used to test.
   Then set a breakpoint on the `isKResult` function and observe the state of the
   interpreter when stopped at that breakpoint. Set a breakpoint on the rule for
   addition and run a program that causes it to be stopped at that breakpoint.
   Finally, step through the program until the addition symbol is at the top of the K cell, and then use the `k match` command to report the reason why
   the subtraction rule does not apply. You may need to modify the definition
   to insert some rule labels.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.20: K Backends and the Haskell Backend.

## Lesson 1.20: K Backends and the Haskell Backend

The purpose of this lesson is to teach about the multiple backends of K, in particular the Haskell Backend which is the complement of the backend we have been using so far.

### K Backends

Thus far, we have not discussed the distinction between the K frontend and the K backends at all. We have simply assumed that if you run `kompile` on a K definition, there will be a compiler backend that will allow you to execute the K definition you have compiled.

K actually has multiple different backends. The one we have been using so far implicitly, the default backend, is called the **LLVM Backend**. It is designed to support efficient, optimized concrete execution and search. It does this by compiling your K definition to LLVM bitcode and then using LLVM to generate machine code for it that is compiled and linked and executed. However, K is a formal methods toolkit at the end of the day, and the primary goal many people have when defining a programming language in K is to ultimately be able to perform more advanced verification on programs in their programming language.

It is for this purpose that K also provides the **Haskell Backend**, so called because it is implemented in Haskell. While we will cover the features of the Haskell Backend in more detail in the next two lessons, the important thing to understand is that it is a separate backend which is optimized for more formal reasoning about programming languages. While it is capable of performing concrete execution, it does not do so as efficiently as the LLVM Backend. In exchange, it provides more advanced features.

## Choosing a backend

You can choose which backend to use to compile a K definition by means of the `--backend` flag to `kompile`. By default, if you do not specify this flag, it is equivalent to if you had specified `--backend llvm`. However, to use the Haskell Backend instead, you can simply say `kompile --backend haskell` on a particular K definition.

As an example, here is a simple K definition that we have seen before in the previous lesson (`lesson-20.k`):

```
module LESSON-20
  imports INT

  rule I => I +Int 1
    requires I <Int 100
endmodule
```

Previously we compiled this definition using the LLVM Backend, but if we instead execute the command `kompile lesson-20.k --backend haskell`, we will get an interpreter for this K definition that is implemented in Haskell instead. Unlike the default LLVM Backend, the Haskell Backend is not a compiler per se. It does not generate new Haskell code corresponding to your programming language and then compile and execute it. Instead, it is an interpreter which reads the generated IR from `kompile` and implements in Haskell an interpreter that is capable of interpreting any K definition.

#### Exercise

Try running the program `0` in this K definition on the Haskell Backend and compare the final configuration to what you would get compiling the same definition with the LLVM Backend.

## Legacy backends

As a quick note, K does provide one other backend, which exists primarily as legacy code which should be considered deprecated. This is the **Java Backend**. The Java Backend is essentially a precursor to the Haskell

Backend. We will not cover this backend in any detail since it is deprecated, but we still mention it here for the purposes of understanding.

### Exercises

1. Compile your solution to Lesson 1.18, Problem 2 with the Haskell Backend and execute some programs. Compare the resulting configurations with the
output of the same program on the LLVM Backend. Note that if you are getting
different behaviors on the Haskell backend, you might have some luck debugging
by passing `--search` to `krun` when using the LLVM backend.

### Next lesson

Once you have completed the above exercises, you can continue to
Lesson 1.21: Unification and Symbolic Execution.

## Lesson 1.21: Unification and Symbolic Execution

The purpose of this lesson is to teach the basic concepts of symbolic execution in order to introduce the unique capabilities of the Haskell Backend at a conceptual level.

### Symbolic Execution

Thus far, all of the programs we have run using K have been **concrete** configurations. What this means is that the configuration we use to initialize the K rewrite engine is concrete; in other words, contains no logical variables. The LLVM Backend is a **concrete execution** engine, meaning that it is only capable of rewriting concrete configurations.

By contrast, the Haskell Backend performs **symbolic execution**, which is capable of rewriting any configuration, including those where parts of the configuration are **symbolic**, ie, contain variables or uninterpreted functions.

### Unification

Previously, we have introduced the concept that K rewrite rules operate by means of pattern matching: the current configuration being rewritten is pattern matched against the left-hand side of the rewrite rule, and the substitution is used in order to construct a new term from the right-hand side. In symbolic execution, we use
unification

instead of pattern matching. To summarize, unification behaves akin to a two-way pattern matching where both the configuration and the left-hand side of the rule can contain variables, and the algorithm generates a **most general unifier** containing substitutions for the variables in both which will make both terms equal.

## Feasibility

Unification by itself cannot completely solve the problem of symbolic execution. One task symbolic execution must perform is to identify whether a particular symbolic term is **feasible**, that is to say, that there actually exists a concrete instantiation of that term such that all the logical constraints on that term can actually be satisfied. The Haskell Backend delegates this task to Z3, a
SMT solver.
This solver is used to periodically trim configurations that are determined to be mathematically infeasible.

## Symbolic terms

The final component of symbolic execution consists of the task of introducing symbolic terms into the configuration. This can be done one of two different ways. First, the term being passed to `krun` can actually be symbolic. This is less frequently used because it requires the user to construct an AST that contains variables, something which our current parsing capabilities are not well-equipped to do. The second, more common, way of introducing symbolic
terms into a configuration consists of writing rules where there exists an existentially qualified variable on the right-hand side of the rule that does not exist on the left-hand side of the rule.

In order to prevent users from writing such rules by accident, K requires that such variables begin with the `?` prefix. For example, here is a rule that rewrites a constructor `foo` to a symbolic integer:

```
rule <k> foo => ?X:Int ...</k>
```

When this rule applies, a fresh variable is introduced to the configuration, which then is unified against the rules that might apply in order to symbolically execute that configuration.

### `ensures` clauses

We also introduce here a new feature of K rules that applies when a rule has this type of variable on the right-hand side: the `ensures` clause.
An `ensures` clause is similar to a `requires` clause and can appear after a rule body, or after a `requires` clause. The `ensures` clause is used to introduce constraints that might apply to the variable that was introduced by

that rule. For example, we could write the rule above with the additional constraint that the symbolic integer that was introduced must be less than five, by means of the following rule:

```
rule <k> foo => ?X:Int ...</k> ensures ?X <Int 5
```

## Putting it all together

Putting all these pieces together, it is possible to use the Haskell Backend to perform symbolic reasoning about a particular K module, determining all the possible states that can be reached by a symbolic configuration.

For example, consider the following K definition (`lesson-21.k`):

```
module LESSON-21
    imports INT

    rule <k> 0 => ?X:Int ... </k> ensures ?X =/=Int 0
    rule <k> X:Int => 5  ... </k> requires X >=Int 10
endmodule
```

When we symbolically execute the program `0`, we get the following output from the Haskell Backend:

```
    <k>
      5 ~> .
    </k>
  #And
    {
      true
    #Equals
      ?X:Int >=Int 10
    }
  #And
    #Not ( {
      ?X:Int
    #Equals
      0
    } )
#Or
    <k>
      ?X:Int ~> .
    </k>
  #And
    #Not ( {
      true
    #Equals
      ?X:Int >=Int 10
```

```
  } )
#And
  #Not ( {
    ?X:Int
  #Equals
    0
  } )
```

Note some new symbols introduced by this configuration: `#And`, `#Or`, and `#Equals`. While `andBool`, `orBool`, and `==K` represent functions of sort `Bool`, `#And`, `#Or`, and `#Equals` are **matching logic connectives**. We will discuss matching logic in more detail later in the tutorial, but the basic idea is that these symbols represent Boolean operators over the domain of configurations and constraints, as opposed to over the `Bool` sort.

Notice that the configuration listed above is a disjunction of conjunctions. This is the most common form of output that can be produced by the Haskell Backend. In this case, each conjunction consists of a configuration and a set of constraints. What this conjunction describes, essentially, is a configuration and a set of information that was derived to be true while rewriting that configuration.

Similar to how we saw `--search` in a previous lesson, the reason we have multiple disjuncts is because there are multiple possible output states for this program, depending on whether or not the second rule applied. In the first case, we see that `?X` is greater than or equal to 10, so the second rule applied, rewriting the symbolic integer to the concrete integer 5. In the second case, we see that the second rule did not apply because `?X` is less than 10. Moreover, because of the `ensures` clause on the first rule, we know that `?X` is not zero, therefore the first rule will not apply a second time. If we had omitted this constraint, we would have ended up infinitely applying the first rule, leading to `krun` not terminating.

In the next lesson, we will cover how symbolic execution forms the backbone of deductive program verification in K and how we can use K to prove programs correct against a specification.

## Exercises

1. Create another rule in `LESSON-21` that rewrites odd integers greater than ten to a symbolic even integer less than 10 and greater than 0. This rule will
   now apply nondeterministically along with the existing rules. Predict what the
   resulting output configuration will be from rewriting `0` after adding this rule. Then run the program and see whether your prediction is correct.

Once you have completed the above exercises, you can continue to

Lesson 1.22: Basics of Deductive Program Verification using K.

# K PL Tutorial

Here you will learn how to use the K tool to define languages by means of a series of screencast movies. It is recommended to do these in the indicated order, because K features already discussed in a previous language definition will likely not be rediscussed in latter definitions. The screencasts follow quite closely the structure of the files under the tutorial folder in the K tool distribution. If you'd rather follow the instructions there and do the tutorial exercises yourself, then go back to https://kframework.org and download the K tool, if you have not done it already. Or, you can first watch the screencasts below and then do the exercises, or do them in parallel.

## K Overview

Make sure you watch the K overview video before you do the K tutorial:

- [9'59"] K Overview

## Learning K

### [34'46"] Part 1: Defining LAMBDA

Here you will learn how to define a very simple functional language in K and the basics of how to use the K tool. The language is a call-by-value variant of lambda calculus with builtins and mu, and its definition is based on substitution.

- [04'07"] Lesson 1, LAMBDA: Syntax Modules and Basic K Commands
- [04'03"] Lesson 2, LAMBDA: Module Importing, Rules, Variables
- [02'20"] Lesson 3, LAMBDA: Evaluation Strategies using Strictness
- [03'13"] Lesson 4, LAMBDA: Generating Documentation; Latex Attributes
- [04'52"] Lesson 5, LAMBDA: Adding Builtins; Side Conditions
- [02'14"] Lesson 6, LAMBDA: Selective Strictness; Anonymous Variables
- [05'10"] Lesson 7, LAMBDA: Derived Constructs; Extending Predefined Syntax
- [02'40"] Lesson 8, LAMBDA: Multiple Binding Constructs (uncommented)
- [06'07"] Lesson 9, LAMBDA: A Complete and Commented Definition (commented)

### [37'07"] Part 2: Defining IMP

Here you will learn how to define a very simple, prototypical textbook C-like imperative language, called IMP, and several new features of the K tool.

- [09'15"] Lesson 1, IMP: Defining a More Complex Syntax

- [04'21"] Lesson 2, IMP: Defining a Configuration
- [10'30"] Lesson 3, IMP: Computations, Results, Strictness; Rules Involving Cells
- [09'16"] Lesson 4, IMP: Configuration Abstraction, Part 1; Types of Rules
- [03'45"] Lesson 5, IMP: Completing and Documenting IMP

### [33'10"] Part 3: Defining LAMBDA++

Here you will learn how to define constructs which abruptly change the execution control, as well as how to define functional languages using environments and closures. LAMBDA++ extends the LAMBDA language above with a callcc construct.

- [06'28"] Lesson 1, LAMBDA++: Abrupt Changes of Control (substitution-based, uncommented)
- [08'02"] Lesson 2, LAMBDA++: Semantic (Non-Syntactic) Computation Items
- [03'21"] Lesson 3, LAMBDA++: Reusing Existing Semantics
- [03'37"] Lesson 4, LAMBDA++: Do Not Reuse Blindly!
- [05'19"] Lesson 5, LAMBDA++: More Semantic Computation Items
- [06'23"] Lesson 6, LAMBDA++: Wrapping Up and Documenting LAMBDA++ (environment-based)

### [46'46"] Part 4: Defining IMP++

Here you will learn how to refine configurations, how to generate fresh elements, how to tag syntactic constructs and rules, how to exhaustively search the space of non-deterministic or concurrent program executions, etc. IMP++ extends the IMP language above with increment, blocks and locals, dynamic threads, input/output, and abrupt termination.

- [07'47"] Lesson 1, IMP++: Extending/Changing an Existing Language Syntax
- [04'06"] Lesson 2, IMP++: Configuration Refinement; Freshness
- [06'56"] Lesson 3, IMP++: Tagging; Superheat/Supercool Kompilation Options
- [05'21"] Lesson 4, IMP++: Semantic Lists; Input/Output Streaming
- [04'30"] Lesson 5, IMP++: Deleting, Saving and Restoring Cell Contents
- [11'40"] Lesson 6, IMP++: Adding/Deleting Cells Dynamically; Configuration Abstraction, Part 2
- [??'??"] Lesson 7, IMP++: Everything Changes: Syntax, Configuration, Semantics
- [06'26"] Lesson 8, IMP++: Wrapping up Larger Languages

### [17'03"] Part 5: Defining Type Systems

Here you will learn how to define various kinds of type systems following various approaches or styles using K.

- [10'11"] Lesson 1, Type Systems: Imperative, Environment-Based Type Systems (IMP++ type checker)
- [06'52"] Lesson 2, Type Systems: Substitution-Based Higher-Order Type Systems (LAMBDA type checker, substitution, uncommented)
- [??'??"] Lesson 3, Type Systems: Environment-Based Higher-Order Type Systems (LAMBDA type checker, environment, uncommented)
- [??'??"] Lesson 4, Type Systems: A Naive Substitution-Based Type Inferencer (for LAMBDA, uncommented)
- [??'??"] Lesson 5, Type Systems: A Naive Environment-Based Type Inferencer (for LAMBDA, uncommented)
- [??'??"] Lesson 6, Type Systems: Parallel Type Checkers/Inferencers (for LAMBDA, uncommented)
- [??'??"] Lesson 7, Type Systems: A Naive Substitution-based Polymorphic Type Inferencer (for LAMBDA, uncommented)
- [??'??"] Lesson 8, Type Systems: A Naive Environment-based Polymorphic Type Inferencer (for LAMBDA, uncommented)
- [??'??"] Lesson 9, Type Systems: Let-Polymorphic Type Inferencer (Damas-Hindley-Milner) (for LAMBDA, uncommented)

#### [??'??"] Part 6: Miscellaneous Other K Features

Here you will learn a few other K features, and better understand how features that you have already seen work.

- [??'??"] ...

## Learning Language Design and Semantics using K

#### [??'??"] Part 7: SIMPLE: Designing Imperative Programming Languages

Here you will learn how to design imperative programming languages using K. SIMPLE is an imperative language with functions, threads, pointers, exceptions, multi-dimensional arrays, etc. We first define an untyped version of SIMPLE, then a typed version. For the typed version, we define both a static and a dynamic semantics.

- [??'??"] Lesson 1, SIMPLE untyped
- [??'??"] Lesson 2, SIMPLE typed static
- [??'??"] Lesson 3, SIMPLE typed dynamic

#### [??'??"] Part 8: KOOL: Designing Object-Oriented Programming Languages

Here woul will learn how to design object-oriented programming languages using K. KOOL is an object-oriented language that extends SIMPLE with classes and objects. We first define an untyped version of KOOL, then a typed version, with both a dynamic and a static semantics.

- [??'??"] Lesson 1, KOOL untyped
- [??'??"] Lesson 2, KOOL typed dynamic
- [??'??"] Lesson 3, KOOL typed static

**[??'??"] Part 9: FUN: Designing Functional Programming Languages**

H
ere woul will learn how to design functional programming languages using K. FUN is a higher-order functional language with general let, letrec, pattern matching, references, lists, callcc, etc. We first define an untyped version of FUN, then a let-polymorphic type inferencer.

- [??'??"] Lesson 1, FUN untyped, Environment-Based
- [??'??"] Lesson 2, FUN untyped, Substitution-Based
- [??'??"] Lesson 3, FUN polymorphic type inferencer

**[??'??"] Part 10: LOGIK: Designing Logic Programming Languages**

Here you will learn how to design a logic programming language using K.

- [??'??"] Lesson 1, LOGIK

## K overview

## An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Go to Youtube mirror, if the above does not work.

Go back to https://kframework.org for further links, the K tool and contact information.

## Learning K

We start by introducing the basic features of K by means of a series of very simple languages. The objective here is neither to learn those languages nor to study their underlying paradigm, but simply to learn K.

- LAMBDA: Lambda calculus defined.
- IMP: A simple imperative language.
- LAMBDA++: LAMBDA extended with control flow.
- IMP++: IMP extended with threads and IO.
- TYPES: LAMBDA type system.

### Part 1: Defining LAMBDA

Here you will learn how to define a very simple language in K and the basics of how to use the K tool. The language is a variant of call-by-value lambda calculus and its definition is based on substitution. Specifically, you will learn the following:

- How to define a module.
- How to define a language syntax.
- How to use the defined syntax to parse programs.
- How to import predefined modules.
- How to define evaluation strategies using strictness attributes.
- How to define semantic rules.
- How the predefined generic substitution works.
- How to generate PDF and HTML documentation from ASCII definitions.
- How to include builtins (integers and Booleans) into your language.
- How to define derived language constructs.

This folder contains several lessons, each adding new features to LAMBDA.

## Syntax Modules and Basic K Commands

Here we define our first K module, which contains the initial syntax of the LAMBDA language, and learn how to use the basic K commands.

Let us create an empty working folder, and open a terminal window (to the left) and an editor window (to the right). We will edit our K definition in the right window in a file called `lambda.k`, and will call the K tool commands in the left window.

Let us start by defining a K module, containing the syntax of LAMBDA.

K modules are introduced with the keywords `module` ... `endmodule`.

The keyword `syntax` adds new productions to the syntax grammar, using a BNF-like notation.

Terminals are enclosed in double-quotes, like strings.

You can define multiple productions for the same non-terminal in the same syntax declaration using the `|` separator.

Productions can have attributes, which are enclosed in square brackets.

The attribute `left` tells the parser that we want the lambda application to be left associative. For example, `a b c d` will then parse as `(((a b) c) d)`.

The attribute `bracket` tells the parser to not generate a node for the parenthesis production in the abstract syntax trees associated to programs. In other words, we want to allow parentheses to be used for grouping, but we do not want to bother to give them their obvious (ignore) semantics.

In our variant of lambda calculus defined here, identifiers and lambda abstractions are meant to be irreducible, that is, are meant to be values. However, so far `Val` is just another non-terminal, just like `Exp`, without any semantic meaning. It will get a semantic meaning later.

After we are done typing our definition in the file `lambda.k`, we can kompile it with the command:

```
kompile lambda.k
```

If we get no errors then a parser has been generated. This parser will be called from now on by default by the krun tool. To see whether and how the parser works, we are going to write some LAMBDA programs and store them in files with the extension `.lambda`.

Let us create a file `identity.lambda`, which contains the identity lambda abstraction:

```
lambda x . x
```

Now let us call `krun` on `identity.lambda`:

```
krun identity.lambda
```

Make sure you call the `krun` command from the folder containing your language definition (otherwise type `krun --help` to learn how to pass a language definition as a parameter to `krun`). The krun command produces the output:

```
<k>
  lambda x . x
</k>
```

If you see such an output it means that your program has been parsed (and then pretty printed) correctly. If you want to see the internal abstract syntax tree (AST) representation of the parsed program, which we call the K AST, then type `kast` in the command instead of `krun`:

```
kast identity.lambda
```

You should normally never need to see this internal representation in your K definitions, so do not get scared (yes, it is ugly for humans, but it is very convenient for tools).

Note that `krun` placed the program in a `<k> ... </k>` cell. In K, computations happen only in cells. If you do not define a configuration in your definition, like we did here, then a configuration will be created automatically for you which contains only one cell, the default `k` cell, which holds the program.

Next, let us create a file `free-variable-capture.lambda`, which contains an expression which, in order to execute correctly in a substitution-based

semantics of LAMBDA, the substitution operation needs to avoid variable-capture:

```
a (((lambda x.lambda y.x) y) z)
```

Next, file `closed-variable-capture.lambda` shows an expression which also requires a capture-free substitution, but this expression is closed (that is, it has no free variables) and all its bound variables are distinct (I believe this is the smallest such expression):

```
(lambda z.(z z)) (lambda x.lambda y.(x y))
```

Finally, the file `omega.lambda` contains the classic omega combinator (or closed expression), which is the smallest expression which loops forever (not now, but after we define the semantics of LAMBDA):

```
(lambda x.(x x)) (lambda x.(x x))
```

Feel free to define and parse several other LAMBDA programs to get a feel for how the parser works. Parse also some incorrect programs, to see how the parser generates error messages.

In the next lesson we will see how to define semantic rules that iteratively rewrite expressions over the defined syntax until they evaluate to a result. This way, we obtain our first programming language defined using K.

Go to Lesson 2, LAMBDA: Module Importing, Rules, Variables

MOVIE (out of date) [4'07"]

## Module Importing, Rules, Variables

We here learn how to include a predefined module (SUBSTITUTION), how to use it to define a K rule (the characteristic rule of lambda calculus), and how to make proper use of variables in rules.

Let us continue our `lambda.k` definition started in the previous lesson.

The `requires` keyword takes a `.k` file containing language features that are needed for the current definition, which can be found in the k-distribution/include/kframework/builtin folder. Thus, the command

```
require "substitution.k"
```

says that the subsequent definition of LAMBDA needs the generic substitution, which is predefined in file `substitution.k` under the folder k-distribution/include/kframework/builtin. Note that substitution can be defined itself in K,
although it uses advanced features that we have not discussed yet in this tutorial, so it may not be easy to understand now.

Using the `imports` keyword, we can now modify LAMBDA to import the module SUBSTITUTION, which is defined in the required `substitution.k` file.

Now we have all the substitution machinery available for our definition. However, since our substitution is generic, it cannot know which language constructs bind variables, and what counts as a variable; however, this information is critical in order to correctly solve the variable capture problem. Thus, you have to tell the substitution that your lambda construct is meant to be a binder, and that your `Id` terms should be treated as variables for substitution. The former is done using the attribute `binder`.
By default, `binder` binds all the variables occurring anywhere in the first argument of the corresponding syntactic construct within its other arguments; you can configure which arguments are bound where, but that will be discussed in subsequent lectures. To tell K which terms are meant to act as variables for binding and substitution, we have to explicitly subsort the desired syntactic categories to the builtin `KVariable` sort.

Now we are ready to define our first K rule. Rules are introduced with the keyword `rule` and make use of the rewrite symbol, `=>`. In our case, the rule defines the so-called lambda calculus *beta-reduction*, which makes use of substitution in its right-hand side, as shown in `lambda.k`.

By convention, variables that appear in rules start with a capital letter (the current implementation of the K tool may even enforce that).

Variables may be explicitly tagged with their syntactic category (also called *sort*). If tagged, the matching term will be checked at run-time for membership to the claimed sort. If not tagged, then no check will be made. The former is safer, but involves the generation of a side condition to the rule, so the resulting definition may execute slightly slower overall.

In our rule in `lambda.k` we tagged all variables with their sorts, so we chose the safest path. Only the `V` variable really needs to be tagged there, because we can prove (using other means, not the K tool, as the K tool is not yet concerned with proving) that the first two variables will always have the claimed sorts whenever we execute any expression that parses within our original grammar.

Let us compile the definition and then run some programs. For example,

```
krun closed-variable-capture.lambda
```

yields the output

```
<k>
  lambda y . ((lambda x . (lambda y . (x  y))) y)
</k>
```

Notice that only certain programs reduce (some even yield non-termination, such as `omega.lambda`), while others do not. For example,

`free-variable-capture.lambda` does not reduce its second argument expression
to `y`, as we would expect. This is because the K rewrite rules between syntactic terms do not apply anywhere they match. They only apply where they have been
given permission to apply by means of appropriate evaluation strategies of language
constructs, which is done using strictness attributes, evaluation contexts, heating/cooling rules, etc., as discussed in the next lessons.

The next lesson will show how to add LAMBDA the desired evaluation strategies using strictness attributes.

Go to Lesson 3, LAMBDA: Evaluation Strategies using Strictness

MOVIE (out of date) [4'03"]

## Evaluation Strategies using Strictness

Here we learn how to use the K `strict` attribute to define desired evaluation strategies. We will also learn how to tell K which terms are already evaluated, so it does not attempt to evaluate them anymore and treats them internally as results of computations.

Recall from the previous lecture that the LAMBDA program `free-variable-capture.lambda` was stuck, because K was not given permission
to evaluate the arguments of the lambda application construct.

You can use the attribute `strict` to tell K that the corresponding construct has a strict evaluation strategy, that is, that its arguments need to be evaluated before the semantics of the construct applies. The order of argument evaluation is purposely unspecified when using `strict`, and indeed the K tool allows us to detect all possible non-deterministic behaviors that result from such intended underspecification of evaluation strategies. We will learn how to do that when we define the IMP language later in this tutorial; we will also learn how to enforce a particular order of evaluation.

In order for the above strictness declaration to work effectively and efficiently, we need to tell the K tool which expressions are meant to be results of computations, so that it will not attempt to evaluate them anymore. One way to do it is to make `Val` a syntactic subcategory of the builtin `KResult` syntactic category. Since we use the same K parser to also parse the semantics, we use the same `syntax` keyword to define additional syntax needed exclusively for the semantics (like `KResult`s). See `lambda.k`.

Compile again and then run some programs. They should all work as expected. In particular, `free-variable-capture.lambda` now evaluates to `a y`.

We now got a complete and working semantic definition of call-by-value lambda-calculus. While theoretically correct, our definition is not easy to use and disseminate. In the next lessons we will learn how to generate formatted documentation for LAMBDA and how to extend LAMBDA in order to write human readable and interesting programs.

Go to Lesson 4, LAMBDA: Generating Documentation; Latex Attributes.

MOVIE (out of date) [2'20"]

## Generating Documentation; Latex Attributes

In this lesson we learn how to generate formatted documentation from K language definitions. We also learn how to use Latex attributes to control the formatting of language constructs, particularly of ones which have a mathematical flavor and we want to display accordingly.

To enhance readability, we may want to replace the keyword `lambda` by the mathematical lambda symbol in the generated documentation. We can control the way we display language constructs in the generated documentation by associating them Latex attributes.

This is actually quite easy. All we have to do is to associate a `latex` attribute to the production defining the construct in question, following the Latex syntax for defining new commands (or macros).

In our case, we associate the attribute `latex(\lambda{#1}.{#2})` to the production declaring the lambda abstraction (recall that in Latex, `#n` refers to the n-th argument of the defined new command).

We will later see, in Lesson 9, that we can add arbitrarily complex Latex comments and headers to our language definitions, which give us maximum flexibility in formatting our language definitions.

Now we have a simple programming language, with a nice documentation. However,
it is not easy to write interesting programs in this language. Almost all programming languages build upon existing data-types and libraries. The K tool provides a few of these (and you can add more).

In the next lesson we show how we can add builtin integers and Booleans to LAMBDA, so we can start to evaluate meaningful expressions.

Go to Lesson 5, LAMBDA: Adding Builtins; Side Conditions.

MOVIE (out of date) [3'13"]

## Adding Builtins; Side Conditions

We have already added the builtin identifiers (sort `Id`) to LAMBDA expressions, but those had no operations on them. In this lesson we add integers and Booleans to LAMBDA, and extend the builtin operations on them into corresponding operations on LAMBDA expressions. We will also learn how to add
side conditions to rules, to limit the number of instances where they can apply.

The K tool provides several builtins, which are automatically included in all definitions. These can be used in the languages that we define, typically by including them in the desired syntactic categories. You can also define your own builtins in case the provided ones are not suitable for your language (e.g., the provided builtin integers and operations on them are arbitrary precision).

For example, to add integers and Booleans as values to our LAMBDA, we have to
add the productions

```
syntax Val ::= Int | Bool
```

`Int` and `Bool` are the nonterminals that correspond to these builtins.

To make use of these builtins, we have to add some arithmetic operation constructs to our language. We prefer to use the conventional infix notation for these, and the usual precedences (i.e., multiplication and division bind tighter than addition, which binds tighter than relational operators). Inspired from SDF, we use `>` instead of
`|` to state that all the previous constructs bind tighter than all the subsequent ones. See `lambda.k`.

The only thing left is to link the LAMBDA arithmetic operations to the corresponding builtin operations, when their arguments are evaluated. This can be easily done using trivial rewrite rules, as shown in `lambda.k`. In general, the K tool attempts to uniformly add the corresponding builtin name as a suffix to all the operations over builtins. For example, the addition over integers is an infix operation named `+Int`.

Compile the new `lambda.k` definition and evaluate some simple arithmetic expressions. For example, if `arithmetic.lambda` is `(1+2*3)/4 <= 1`, then

```
krun arithmetic.lambda
```

yields, as expected, `true`. Note that the parser took the desired operation precedence into account.

Let us now try to evaluate an expression which performs a wrong computation, namely a division by zero. Consider the expression `arithmetic-div-zero.lambda` which is `1/(2/3)`. Since division is strict and `2/3` evaluates to `0`, this

expression reduces to `1/0`, which further reduces to `1 /Int 0` by the rule for division, which is now stuck (with the current back-end to the K tool).

In fact, depending upon the back-end that we use to execute K definitions and in particular to evaluate expressions over builtins, `1 /Int 0` can evaluate to anything. It just happens that the current back-end keeps it as an irreducible term. Other K back-ends may reduce it to an explicit error element, or issue a segmentation fault followed by a core dump, or throw an exception, etc.

To avoid requesting the back-end to perform an illegal operation, we may use a side condition in the rule of division, to make sure it only applies when the denominator is non-zero.

Like in other operational formalisms, the role of the K side conditions is to filter the number of instances of the rule. The notion of a *side condition* comes from logics, where a sharp distinction is made between a side condition (cheap) and a *premise* (expensive). Premises are usually resolved using further (expensive) logical derivations, while side conditions are simple (cheap) conditions over the rule meta-variables within the underlying mathematical domains (which in K can be extended by the user, as we will see in future lessons). Regarded as a logic, K derives rewrite rules from other rewrite rules; therefore, the K side conditions cannot contain other rewrites in them (using `=>`). This contrasts other rewrite engines, for example Maude, which
allow conditional rules with rewrites in conditions.
The rationale behind this deliberate restriction in K is twofold:

- On the one hand, general conditional rules require a complex, and thus slower
  rewrite engine, which starts recursive (sometimes exhaustive) rewrite sessions
  to resolve the rewrites in conditions. In contrast, the side conditions in K can be evaluated efficiently by back-ends, for example by evaluating builtin expressions and/or by calling builtin functions.
- On the other hand, the semantic definitional philosophy of K is that rule premises are unnecessary, so there is no need to provide support for them.

Having builtin arithmetic is useful, but writing programs with just lambda and arithmetic constructs is still a pain. In the next two lessons we will add conditional (`if_then_else`) and binding (`let` and `letrec`) constructs, which will allow us to write nicer programs.

Go to Lesson 6, LAMBDA: Selective Strictness; Anonymous Variables.

MOVIE (out of date) [4'52"]

## Selective Strictness; Anonymous Variables

We here show how to define selective strictness of language constructs, that is, how to state that certain language constructs are strict only in some arguments. We also show how to use anonymous variables.

We next define a conditional `if` construct, which takes three arguments, evaluates only the first one, and then reduces to either the second or the third, depending on whether the first one evaluated to true or to false.

K allows to define selective strictness using the same `strict` attribute, but passing it a list of numbers. The numbers correspond to the arguments in which we want the defined construct to be strict. In our case,

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp   [strict(1)]
```

states that the conditional construct is strict in the first argument.

We can now assume that its first argument will eventually reduce to a value, so we only write the following two semantic rules:

```
rule if true  then E else _ => E
rule if false then _ else E => E
```

Thus, we assume that the first argument evaluates to either `true` or `false`.

Note the use of the anonymous variable `_`. We use such variables purely for structural reasons, to state that something is there but we don't care what. An anonymous variable is therefore completely equivalent to a normal variable which is unsorted and different from all the other variables in the rule. If you use `_` multiple times in a rule, they will all be considered distinct.

Compile `lambda.k` and write and execute some interesting expressions making use of the conditional construct. For example, the expression

```
if 2<=1 then 3/0 else 10
```

evaluates to `10` and will never evaluate `3/0`, thus avoiding an unwanted division-by-zero.

In the next lesson we will introduce two new language constructs, called `let` and `letrec` and conventionally found in functional programming languages, which will allow us to already write interesting LAMBDA programs.

Go to Lesson 7, LAMBDA: Derived Constructs; Extending Predefined Syntax.

MOVIE (out of date) [2'14"]

# Derived Constructs, Extending Predefined Syntax

In this lesson we will learn how to define derived language constructs, that is, ones whose semantics is defined completely in terms of other language constructs. We will also learn how to add new constructs to predefined syntactic categories.

When defining a language, we often want certain language constructs to be defined in terms of other constructs. For example, a let-binding construct of the form

```
let x = e in e'
```

is nothing but syntactic sugar for

```
(lambda x . e') e
```

This can be easily achieved with a rule, as shown in `lambda.k`.

As a side point, which is not very relevant here but good to know, we may want the *desugaring* of `let` to not even count as a computational step, but as a mere *structural rearrangement* of the program so that other semantic rules (beta reduction, in our case) can match and apply.

The K tool allows us to tag rules with the attribute `structural`, with precisely the intuition above. You can think of structural rules as a kind of light rules, almost like macros, or like ones which apply *under the hood*, instantaneously. There are several other uses for structural rules in K, which we will discuss later in this tutorial.

Compile `lambda.k` and write some programs using `let` binders.

For example, consider a `lets.lambda` program which takes `arithmetic.lambda` and replaces each integer by a let-bound variable. It should evaluate to `true`, just like the original `arithmetic.lambda`.

Let us now consider a more interesting program, namely one that calculates the factorial of 10:

```
let f = lambda x . (
        (lambda t . lambda x . (t t x))
        (lambda f . lambda x . (if x <= 1 then 1 else (x * (f f (x + -1)))))
        x
      )
in (f 10)
```

This program follows a common technique to define fixed points in untyped lambda calculus, based on passing a function to itself.

We may not like to define fixed-points following the approach above, because it requires global changes in the body of the function meant to be recursive,

basically to pass it to itself (`f f` in our case above). The approach below isolates the fixed-point aspect of the function in a so-called *fixed-point combinator*, which we call `fix` below, and then apply it to the function defining the body of the factorial, without any changes to it:

```
let fix = lambda f . (
          (lambda x . (f (lambda y . (x x y))))
          (lambda x . (f (lambda y . (x x y))))
        )
in let f = fix (lambda f . lambda x .
                (if x <= 1 then 1 else (x * (f (x + -1)))))
   in (f 10)
```

Although the above techniques are interesting and powerful (indeed, untyped lambda calculus is in fact Turing complete), programmers will probably not like to write programs this way.

We can easily define a more complex derived construct, called `letrec` and conventionally encountered in functional programming languages, whose semantics
captures the fixed-point idea above. In order to keep its definition simple and intuitive, we define a simplified variant of `letrec`, namely one which only allows to define one recursive one-argument function. See `lambda.k`.

There are two interesting observations here.

First, note that we have already in-lined the definition of the `fix` combinator in the definition of the factorial, to save one application of the beta reduction rule (and the involved substitution steps). We could have in-lined the definition of the remaining `let`, too, but we believe that the current definition is easier to read.

Second, note that we extended the predefined `Id` syntactic category with two new constants, `$x` and `$y`. The predefined identifiers cannot start with `$`, so programs that will be executed with this semantics cannot possibly contain the identifiers *xandx*andy. In other words, by adding them to Id they become indirectly reserved for the semantics. This is indeed desirable, because any possible uses of *xinthebodyofthefunctiondefinedusing*‘*letrec*‘*wouldbecapturedbythe*‘*lambda*xinth
`in the definition of`letrec‘.

Using `letrec`, we can now write the factorial program as elegantly as it can be written in a functional language:

```
letrec f x = if x <= 1 then 1 else (x * (f (x + -1)))
in (f 10)
```

In the next lesson we will discuss an alternative definition of `letrec`, based on another binder, `mu`, specifically designed to define fixed points.

Go to Lesson 8, LAMBDA: Multiple Binding Constructs.

MOVIE (out of date) [5'10"]

## Multiple Binding Constructs

Here we learn how multiple language constructs that bind variables can coexist. We will also learn about or recall another famous binder besides `lambda`, namely `mu`, which can be used to elegantly define all kinds of interesting fixed-point constructs.

The `mu` binder has the same syntax as lambda, except that it replaces `lambda` with `mu`.

Since `mu` is a binder, in order for substitution to know how to deal with variable capture in the presence of `mu`, we have to tell it that `mu` is a binding construct, same like lambda. We take advantage of being there and also add `mu` its desired latex attribute.

The intuition for

```
mu x . e
```

is that it reduces to `e`, but each free occurrence of `x` in `e` behaves like a pointer that points back to `mu x . e`.

With that in mind, let us postpone the definition of `mu` and instead redefine `letrec F X = E in E'` as a derived construct, assuming `mu` available. The idea is to simply regard `F` as a fixed-point of the function

```
lambda X . E
```

that is, to first calculate

```
mu F . lambda X . E
```

and then to evaluate `E'` where `F` is bound to this fixed-point:

```
let F = mu F . lambda X . E in E'
```

This new definition of `letrec` may still look a bit tricky, particularly because `F` is bound twice, but it is much simpler and cleaner than our previous definition. Moreover, now it is done in a type-safe manner (this aspect goes beyond our objective in this tutorial).

Let us now define the semantic rule of `mu`.

The semantics of `mu` is actually disarmingly simple. We just have to substitute `mu X . E` for each free occurrence of `X` in `E`:

```
mu X . E => E[(mu X . E) / X]
```

Compile `lambda.k` and execute some recursive programs. They should be now several times faster. Write a few more recursive programs, for example ones

for calculating the Ackermann function, for calculating the number of moves needed to solve the Hanoi tower problem, etc.

We have defined our first programming language in K, which allows us to write interesting functional programs. In the next lesson we will learn how to fully document our language definition, in order to disseminate it, to ship it to colleagues or friends, to publish it, to teach it, and so on.

Go to Lesson 9, LAMBDA: A Complete and Commented Definition.

MOVIE (out of date) [2'40"]

# A Complete and Documented K Definition

In this lesson you will learn how to add formal comments to your K definition, in order to nicely document it. The generated document can be then used for various purposes: to ease understanding the K definition, to publish it, to send it to others, etc.

The K tool allows a literate programming style, where the executable language definition can be documented by means of annotations. One such annotation is the `latex(_)` annotation, where you can specify how to format the given production when producing Latex output via the `--output latex` option to `krun`, `kast`, and `kprove`.

There are three types of comments, which we discuss next.

## Ordinary comments

These use `//` or `/* ... */`, like in various programming languages. These comments are completely ignored.

## Document annotations

Use the `@` symbol right after `//` or `/*` in order for the comment to be considered an annotation and thus be processed by the K tool when it generates documentation.

As an example, we can go ahead and add such an annotation at the beginning of the LAMBDA module, explaining how we define the syntax of this language.

## Header annotations

Use the `!` symbol right after `//` or `/*` if you want the comment to be considered a header annotation, that is, one which goes before `\begin{document}` in the generated Latex. You typically need header annotations to include macros, or to define a title, etc.

As an example, let us set a Latex length and then add a title and an author to this K definition.

Compile the documentation and take a look at the results. Notice the title.

Feel free to now add lots of annotations to `lambda.k`.

Then compile and check the result. Depending on your PDF viewer, you may also see a nice click-able table of contents, with all the sections of your document. This could be quite convenient when you define large languages, because it helps you jump to any part of the semantics.

Tutorial 1 is now complete. The next tutorial will take us through the definition of a simple imperative language and will expose us to more feature of the K framework and the K tool.

MOVIE (out of date) [6'07"]

### Part 2: Defining IMP

Here you will learn how to define a very simple imperative language in K and the basics of how to work with configurations, cells, and computations. Specifically, you will learn the following:

- How to define languages using multiple modules.
- How to define sequentially strict syntactic constructs.
- How to use K's syntactic lists.
- How to define, initialize and configure configurations.
- How the language syntax is swallowed by the builtin K syntactic category.
- The additional syntax of the K syntactic category.
- How the strictness annotations are automatically desugared into rules.
- The first steps of the configuration abstraction mechanism.
- The distinction between structural and computational rules.

Like in the previous tutorial, this folder contains several lessons, each adding new features to IMP. Do them in order. Also, make sure you completed and understood the previous tutorial.

## Defining a More Complex Syntax

Here we learn how to define a more complex language syntax than LAMBDA's, namely the C-like syntax of IMP. Also, we will learn how to define languages using multiple modules, because we are going to separate IMP's syntax from its semantics using modules. Finally, we will also learn how to use K's builtin support for syntactic lists.

The K tool provides modules for grouping language features. In general, we can organize our languages in arbitrarily complex module structures. While there are no rigid requirements or even guidelines for how to group

language features in modules, we often separate the language syntax from the language semantics in different modules.

In our case here, we start by defining two modules, IMP-SYNTAX and IMP, and
import the first in the second, using the keyword `imports`. As their names suggest, we will place all IMP's syntax definition in IMP-SYNTAX and all its semantics in IMP.

Note, however, that K does no more than simply includes all the contents of the imported module in the one which imports it (making sure that everything is only kept once, even if you import it multiple times). In other words, there is currently nothing fancy in K tool's module system.

IMP has six syntactic categories, as shown in `imp.k`: `AExp` for arithmetic expressions, `BExp` for Boolean expressions, `Block` for blocks, `Stmt` for statements, `Pgm` for programs and `Ids` for comma-separated lists of identifiers. Blocks are special statements, whose role is to syntactically constrain the conditional statement and the while loop statement to only take blocks as branches and body, respectively.

There is nothing special about arithmetic and Boolean expressions. They are given the expected strictness attributes, except for `<=` and `&&`, for demonstration purposes.

The `<=` is defined to be `seqstrict`, which means that it evaluates its arguments in order, from left-to-right (recall that the `strict` operators can evaluate their arguments in any, fully interleaved, orders). Like `strict`, the `seqstrict` annotation can also be configured; for example, one can specify in which arguments and in what order. By default, `seqstrict` refers to all the arguments, in their left-to-right order. In our case here, it is equivalent with `seqstrict(1 2)`.

The `&&` is only strict in its first argument, because we will give it a short-circuited semantics (its second argument will only be evaluated when the first evaluates to true). Recall the K tool also allows us to associate LaTex attributes to constructs, telling the document generator how to display them. For example, we associate `<=` the attribute `latex({#1}\leq{#2})`, which makes it be displayed $\leq$ everywhere in the generated LaTex documentation.

In this tutorial we take the freedom to associate the various constructs parsing precedences that we have already tested and we know work well, so that we can focus on the semantics here instead of syntax. In practice, though, you typically need to experiment with precedences until you obtain the desired parser.

Blocks are defined using curly brackets, and they can either be empty or hold a statement.

Nothing special about the IMP statements. Note that `;` is an assignment statement terminator, not a statement separator. Note also that blocks are special statements.

An IMP program declares a comma-separated list of variables using the keyword `int` like in C, followed by a semicolon `;`, followed by a statement. Syntactically, the idea here is that we can wrap any IMP program within a `main(){...}` function and get a valid C program. IMP does not allow variable declarations anywhere else except through this construct, at the top-level of the program. Other languages provided with the K distribution (see, e.g., the IMP++ language also discussed in this tutorial) remove this top-level program construct of IMP and add instead variable declaration as a statement construct, which can be used anywhere in the program, not only at the top level.

Note how we defined the comma-separated list of identifiers using `List{Id,","}`. The K tool provides builtin support for generic syntactic lists. In general,

```
syntax B ::= List{A,T}
```

declares a new non-terminal, `B`, corresponding to `T`-separated sequences of elements of `A`, where `A` is a non-terminal and `T` is a terminal. These lists can also be empty, that is, IMP programs declaring no variable are also allowed (e.g., `int; {}` is a valid IMP program). To instantiate and use the K builtin lists, you should alias each instance with a (typically fresh) non-terminal in your syntax, like we do with the `Ids` nonterminal.

Like with other K features, there are ways to configure the syntactic lists, but we do not discuss them here.

Recall from Tutorial 1 (LAMBDA) that in order for strictness to work well we also need to tell K which computations are meant to be results. We do this as well now, in the module IMP: integers and Booleans are K results.

Kompile `imp.k` and test the generated parser by running some programs. Since IMP is a fragment of C, you may want to select the C mode in your editor when writing these programs. This will also give your the feel that you are writing programs in a real programming language.

For example, here is `sum.imp`, which sums in `sum` all numbers up to `n`:

```
int n, sum;
n = 100;
sum=0;
while (!(n <= 0)) {
  sum = sum + n;
  n = n + -1;
}
```

Now krun it and see how it looks parsed in the default `k` cell.

The program `collatz.imp` tests the Collatz conjecture for all numbers up to `m` and accumulates the total number of steps in `s`:

```
int m, n, q, r, s;
m = 10;
while (!(m<=2)) {
  n = m;
  m = m + -1;
  while (!(n<=1)) {
    s = s+1;
    q = n/2;
    r = q+q+1;
    if (r<=n) {
      n = n+n+n+1;          // n becomes 3*n+1 if odd
    } else {n=q;}           //        of   n/2 if even
  }
}
```

Finally, program `primes.imp` counts in `s` all the prime numbers up to `m`:

```
int i, m, n, q, r, s, t, x, y, z;
m = 10;  n = 2;
while (n <= m) {
  // checking primality of n and writing t to 1 or 0
  i = 2;  q = n/i;  t = 1;
  while (i<=q && 1<=t) {
    x = i;
    y = q;
    // fast multiplication (base 2) algorithm
    z = 0;
    while (!(x <= 0)) {
      q = x/2;
      r = q+q+1;
      if (r <= x) { z = z+y; } else {}
      x = q;
      y = y+y;
    } // end fast multiplication
    if (n <= z) { t = 0; } else { i = i+1;  q = n/i; }
  } // end checking primality
  if (1 <= t) { s = s+1; } else {}
  n = n+1;
}
```

All the programs above will run once we define the semantics of IMP. If you want to execute them now, wrap them in a `main(){...}` function and compile them and run them with your favorite C compiler.

Before we move to the K semantics of IMP, we would like to make some

clarifications regarding the K builtin parser, `kast`. Although it is quite powerful, you should not expect magic from it! While the K parser can parse many non-trivial languages (see, for example, the KOOL language in pl-tutorial/2_languages) in the K distribution), it was never meant to be a substitute for real parsers. We often call the syntax defined in K *the syntax of the semantics*, to highlight the fact that its role is to serve as a convenient notation when writing the semantics, not necessarily as a means to define concrete syntax of arbitrarily complex programming languages. See the KERNELC language for an example on how to connect an external parser for concrete syntax to the K tool.

The above being said, we strongly encourage you to strive to make the builtin parser work with your desired language syntax! Do not give up simply because you don't want to deal with syntactic problems. On the contrary, fight for your syntax! If you really cannot define your desired syntax because of tool limitations, we would like to know. Please tell us.

Until now we have only seen default configurations. In the next lesson we will learn how to define a K custom configuration.

Go to Lesson 2, IMP: Defining a Configuration.

MOVIE (out of date) [09'15"]

## Defining a Configuration

Here we learn how to define a configuration in K. We also learn how to initialize and how to display it.

As explained in the overview presentation on K, configurations are quite important, because all semantic rules match and apply on them. Moreover, they are the backbone of *configuration abstraction*, which allows you to only mention the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically. The importance of configuration abstraction will become clear when we define more complex languages (even in IMP++). IMP does not really need it. K configurations are constructed making use of cells, which are labeled and can be arbitrarily nested.

Configurations are defined with the keyword `configuration`. Cells are defined using an XML-ish notation stating clearly where the cell starts and where it ends.

While not enforced by the tool, we typically like to put the entire configuration in a top-level cell, called `T`. So let's define it:

```
configuration <T>...</T>
```

Cells can have other cells inside. In our case of IMP, we need a cell to hold the remaining program, cell which we typically call `k`, and a cell to hold the program state. Let us add them:

```
configuration <T> <k>...</k> <state>...</state> </T>
```

K allows us to also specify how to initialize a configuration at the same time with declaring the configuration. All we have to do is to fill in the contents of the cells with some terms. The syntactic categories of those terms will also indirectly define the types of the corresponding cells.

For example, we want the `k` cell to initially hold the program that is passed to `krun`. K provides a builtin configuration variable, called `$PGM`, which is specifically designed for this purpose: `krun` will place its program there (after it parses it, or course). The K tool allows users to define their own configuration variables, too, which can be used to develop custom initializations of program configurations with the help of `krun`; this can be quite useful when defining complex languages, but we do not discuss it in this tutorial.

```
configuration <T> <k> $PGM </k> <state>...</state>  </T>
```

Moreover, we want the program to be a proper `Pgm` term (because we do not want to allow `krun` to take fragments of programs, for example, statements). Therefore, we tag `$PGM` with the desired syntactic category, `Pgm`:

```
configuration <T> <k> $PGM:Pgm </k> <state>...</state>  </T>
```

Like for other variable tags in K, a run-time check will be performed and the semantics will get stuck if the passed term is not a well-formed program.

We next tell K that the state cell should be initialized with the empty map:

```
configuration <T> <k> $PGM:Pgm </k> <state> .Map </state>  </T>
```

Recall that in K `.` stands for *nothing*. However, since there are various types of nothing, to avoid confusion we can suffix the `.` with its desired type. K has several builtin data-types, including lists, sets, bags, and maps. `.Map` is the empty map.

Kompile `imp.k` and run several programs to see how the configuration is initialized as desired.

When configurations get large, and they do when defining large programming languages, you may want to color the cells in order to more easily distinguish them. This can be easily achieved using the `color` cell attribute, following again an XML-ish style:

```
configuration <T color="yellow">
                <k color="green"> $PGM:Pgm </k>
```

```
    <state color="red"> .Map </state>
  </T>
```

In the next lesson we will learn how to write rules that involve cells.

Go to Lesson 3, IMP: Computations, Results, Strictness; Rules Involving Cells.

MOVIE (out of date) [04'21"]

# Computations, Results, Strictness; Rules Involving Cells

In this lesson we will learn about the syntactic category `K` of computations, about how strictness attributes are in fact syntactic sugar for rewrite rules over computations, and why it is important to tell the tool which computations are results. We will also see a K rule that involves cells.

## K Computations

Computation structures, or more simply *computations*, extend the abstract syntax of your language with a list structure using `~>` (read *followed by* or *and then*, and written $\curvearrowright$ in Latex) as a separator.
K provides a distinguished sort, `K`, for computations. The extension of the abstract syntax of your language into computations is done automatically by the K tool when you declare constructs using the `syntax` keyword, so the K semantic rules can uniformly operate only on terms of sort `K`. The intuition for computation structures of the form

```
t1 ~> t2 ~> ... ~> tn
```

is that the listed tasks are to be processed in order. The initial computation typically contains the original program as its sole task, but rules can then modify it into task sequences, as seen shortly.

## Strictness in Theory

The strictness attributes, used as annotations to language constructs, actually correspond to rules over computations. For example, the `strict(2)` attribute of the assignment statement corresponds to the following two opposite rules (`X` ranges over `Id` and `A` over `AExp`):

```
X=A; => A ~> X=[];
A ~> X=[]; => X=A;
```

The first rule pulls `A` from the syntactic context `X=A;` and schedules it for processing. The second rule plugs `A` back into its context.
Inspired from the chemical abstract machine, we call rules of the first type above *heating* rules and rules of the second type *cooling* rules.

Similar rules are generated for other arguments in which operations are strict. Iterative applications of heating rules eventually bring to the top of the computation atomic tasks, such as a variable lookup, or a builtin operation, which then make computational progress by means of other rules. Once progress is made, cooling rules can iteratively plug the result back into context, so that heating rules can pick another candidate for reduction, and so on and so forth.

When operations are strict only in some of their arguments, the corresponding positions of the arguments in which they are strict are explicitly enumerated in the argument of the `strict` attribute, e.g., `strict(2)` like above, or `strict(2 3)` for an operation strict in its second and third arguments, etc. If an operation is simply declared `strict` then it means that it is strict in all its arguments. For example, the strictness of addition yields:

```
A1+A2 => A1 ~> []+A2
A1 ~> []+A2 => A1+A2
A1+A2 => A2 ~> A1+[]
A2 ~> A1+[] => A1+A2
```

It can be seen that such heating/cooling rules can easily lead to non-determinism, since the same term may be heated many different ways; these different evaluation orders may lead to different behaviors in some languages (not in IMP, because its expressions do not have side effects, but we will experiment with non-determinism in its successor, IMP++).

A similar desugaring applies to sequential strictness, declared with the keyword `seqstrict`. While the order of arguments of `strict` is irrelevant, it matters in the case of `seqstrict`: they are to be evaluated in the specified order; if no arguments are given, then they are assumed by default to be evaluated from left-to-right. For example, the default heating/cooling rules associated to the sequentially strict `<=` construct above are (`A1`, `A2` range over `AExp` and `I1` over `Int`):

```
A1<=A2 => A1 ~> []<=A2
A1 ~> []<=A2 => A1<=A2
I1<=A2 => A2 ~> I1<=[]
A2 ~> I1<=[] => I1<=A2
```

In other words, `A2` is only heated/cooled after `A1` is already evaluated.

While the heating/cooling rules give us a nice and uniform means to define all the various allowable ways in which a program can evaluate, all based on rewriting, the fact that they are reversible comes with a serious practical problem: they make the K definitions unexecutable, because they lead to non-termination.

## Strictness in Practice; K Results

To break the reversibility of the theoretical heating/cooling rules, and, moreover, to efficiently execute K definitions, the current implementation of the K tool relies on users giving explicit definitions of their languages' results.

The K tool provides a predicate `isKResult`, which is automatically defined as we add syntactic constructs to `KResult` (in fact the K tool defines such predicates for all syntactic categories, which are used, for example, as rule side conditions to check user-declared variable memberships, such as `V:Val` stating that `V` belongs to `Val`).

The `kompile` tool, depending upon what it is requested to do, changes the reversible heating/cooling rules corresponding to evaluation strategy definitions (e.g., those corresponding to strictness attributes) to avoid non-termination. For example, when one is interested in obtaining an executable model of the language (which is the default compilation mode of `kompile`), then heating is performed only when the to-be-pulled syntactic fragment is not a result, and the corresponding cooling only when the to-be-plugged fragment is a result. In this case, e.g., the heating/cooling rules for assignment are modified as follows:

```
X=A; => A ~> X=[];  requires notBool isKResult(A)
A ~> X=[]; => X=A;  requires isKResult(A)
```

Note that non-termination of heating/cooling is avoided now. The only thing lost is the number of possible behaviors that a program can manifest, but this is irrelevant when all we want is one behavior.

As will be discussed in the IMP++ tutorial, the heating/cooling rules are modified differently by `kompile` when we are interested in other aspects of the language definition, such us, for example, in a search-able model that comprises all program behaviors. This latter model is obviously more general from a theoretical perspective, but, in practice, it is also slower to execute. The `kompile` tool strives to give you the best model of the language for the task you are interested in.

## Can't Results be Inferred Automatically?

This is a long story, but the short answer is: *No!*. Maybe in some cases it is possible, but we prefer to not attempt it in the K tool. For example, you most likely do not want any stuck computation to count as a result, since some of them can happen simply because you forgot a semantic rule that could have further reduce it! Besides, in our experience with defining large languages, it is quite useful to take your time and think of what the results of your language's computations are. This fact in itself may help you improve your overall language design. We typically do it at the same time with defining the evaluation strategies of our languages. Although in theory K

could infer the results of your language as the stuck computations, based on the above we have deliberately decided to not provide this feature, in spite of requests from some users. So you currently do have to explicitly define your K results if you want to effectively use the K tool. Note, however, that theoretical definitions, not meant to be executed, need not worry about defining results (that's because in theory semantic rules apply *modulo* the reversible heating/cooling rules, so results are not necessary).

## A K Rule Involving Cells

All our K rules so far in the tutorial were of the form

```
rule left => right requires condition
```

where `left` and `right` were syntactic, or more generally computation, terms.

Here is our first K rule explicitly involving cells:

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

Recall that the `k` cell holds computations, which are sequences of tasks separated by `~>`. Also, the `state` cell holds a map, which is a set of bindings, each binding being a pair of computations (currently, the K builtin data-structures, like maps, are untyped; or, said differently, they are all over the type of computations, `K`).

Therefore, the two cells mentioned in the rule above hold collections of things, ordered or not. The `...`s, which we also call cell *frames*, stand for *more stuff there, which we do not care about*.

The rewrite relation `=>` is allowed in K to appear anywhere in a term, its meaning being that the corresponding subterm is rewritten as indicated in the shown context. We say that K's rewriting is *local*.

The rule above says that if the identifier `X` is the first task in the `k` cell, and if `X` is bound to `I` somewhere in the `state`, then `X` rewrites to `I` *locally* in the `k` cell. Therefore, IMP variables need to be already declared when looked up.

Of course, the K rule above can be translated into an ordinary rewrite rule of the form

```
rule <k> X ~> Rest </k> <state> Before (X |-> I) After </state>
  => <k> I ~> Rest </k> <state> Before (X |-> I) After </state>
```

Besides being more verbose and thus tedious to write, this ordinary rule is also more error-prone; for example, we may forget the `Rest` variable in the right-hand-side, etc. Moreover, the concurrent semantics of K allows for its rules to be interpreted as *concurrent transactions*, where the context is the read-only component of the transaction, while the subterms which are rewritten are read/write component of the transaction;

thus, K rule instances can apply concurrently if they only overlap on read-only parts, while they cannot if regarded as ordinary rewrite logic rules. Note: our current implementation of the K tool is not concurrent, so K rules are in fact desugared as normal rewrite rules in the K tool.

Kompile `imp.k` using a documentation option and check out how the K rule looks in the generated document. The `...` frames are displayed as cell tears, metaphorically implying that those parts of the cells that we do not care about are *torn away*. The rewrite relation is replaced by a horizontal line: specifically, the subterm which rewrites, `X`, is underlined, and its replacement is written underneath the line.

In the next lesson we define the complete K semantics of IMP and run the programs we parsed in the first lesson.

Go to Lesson 4, IMP: Configuration Abstraction, Part 1; Types of Rules.

MOVIE (out of date) [10'30"]

# Configuration Abstraction, Part 1; Types of Rules

Here we will complete the K definition of IMP and, while doing so, we will learn the very first step of what we call *configuration abstraction*, and the semantic distinction between structural and computational rules.

## The IMP Semantic Rules

Let us add the remaining rules, in the order in which the language constructs were defined in IMP-SYNTAX.

The rules for the arithmetic and Boolean constructs are self-explanatory. Note, however, that K will infer the correct sorts of all the variables in these rules, because they appear as arguments of the builtin operations (`_+Int_`, etc.). Moreover, the inferred sorts will be enforced dynamically. Indeed, we do not want to apply the rule for addition, for example, when the two arguments are not integers. In the rules for `&&`, although we prefer to not do it here for simplicity, we could have eliminated the dynamic check by replacing `B` (and similarly for `_`) with `B:K`. Indeed, it can be shown that whenever any of these rules apply, `B` (or `_`) is a `BExp` anyway. That's because there is no rule that can touch such a `B` (or `_`); this will become clearer shortly, when we discuss the first step of configuration abstraction. Therefore, since we know that `B` will be a `BExp` anyway, we could save the time it takes to check its sort; such times may look minor, but they accumulate, so some designers may prefer to avoid run-time checks whenever possible.

The block rules are trivial. However, the rule for non-empty blocks is semantically correct only because we do not have local variable declarations in IMP. We will have to change this rule in IMP++.

The assignment rule has two `=>`: one in the `k` cell dissolving the assignment statement, and the other in the `state` cell updating the value of the assigned variable. Note that the one in the `state` is surrounded by parentheses: `(_ => I)`. That is because `=>` is greedy: it matches as much as it can to the left and to the right, until it reaches the cell boundaries (closed or open). If you want to limit its scope, or for clarity, you can use parentheses like here.

The rule for sequential composition simply desugars `S1 S2` into `S1 ~> S2`. Indeed, the two have exactly the same semantics. Note that statements *evaluate* to nothing (`.`), so once `S1` is processed in `S1 ~> S2`, then the next task is automatically `S2`, without wasting any step for the transition.

The rules for the conditional and while statements are clear. One thing to keep in mind now is that the `while` unrolling rule will not apply indefinitely in the positive branch of the resulting conditional, because of K's configuration abstraction, which will be discussed shortly.

An IMP program declares a set of variables and then executes a statement in the state obtained after initializing all those variables to `0`. The rules for programs initialize the declared variables one by one, checking also that there are no duplicates. We check for duplicates only for demonstration purposes, to illustrate the `keys` predefined operation that returns the set of keys of a map, and the set membership operation `in`. In practice, we typically define a static type checker for our language, which we execute before the semantics and reject inappropriate programs.

The use of the `.Ids` in the second rule is not necessary. We could have written `int; S` instead of `int .Ids; S` and the K tool would parse it and kompile the definition correctly, because it uses the same parser used for parsing programs also to parse the semantics. However, we typically prefer to explicitly write the *nothing* values in the semantics, for clarity; the parser has been extended to accept these. Note that the first rule matches the entire `k` cell, because `int_;_` is the top-level program construct in IMP, so there is nothing following it in the computation cell. The anonymous variable stands for the second argument of this top-level program construct, not for the rest of the computation. The second rule could have also been put in a complete `k` cell, but we preferred not to, for simplicity.

Our IMP semantics is now complete, but there are a few more things that we need to understand and do.

## Configuration Abstraction, Part 1

First, let us briefly discuss the very first step of configuration abstraction. In K, all semantic rules are in fact rules between configurations. As soon explained in the IMP++ tutorial, the declared configuration cell structure is used to automatically complete the missing configuration parts in rules. However, many rules do not involve any cells, being rules between syntactic terms (of sort `K`); for example, we had only three rules involving cells in our IMP semantics. In this case, the `k` cell will be added automatically and the actual rewrite will happen on top of the enclosed computation. For example, the rule for the `while` loop is automatically translated into the following:

```
rule <k> while (B) S => if (B) {S while (B) S} else {} ...</k>
```

Since the first task in computations is what needs to be done next, the intuition for this rule completion is that the syntactic transition only happens when the term to rewrite is ready for processing. This explains, for example, why the while loop unrolling does not indefinitely apply in the positive branch of the conditional: the inner while loop is not ready for evaluation yet. We call this rule completion process, as well as other similar ones, *configuration abstraction*. That is because the incomplete rule abstracts away the configuration structure, thus being easier to read. As seen soon when we define IMP++, configuration abstraction is not only a user convenience; it actually significantly increases the modularity of our definitions. The k-cell-completion is only the very first step, though.

If you really want certain rewrites over syntactic terms to apply anywhere they match, then you should tag the rule with the attribute `anywhere`, which was discussed in Tutorial 1, Lesson 2.5.

## Structural vs. Computational Rules

The K rules are of two types: structural and computational. Intuitively, structural rules rearrange the configuration so that computational rules can apply. Structural rules therefore do not count as computational steps. A K semantics can be thought of as a generator of transition systems, one for each program. It is only the computational rules that create steps, or transitions, in the corresponding transition system, the structural rules being unobservable at this level. By default, rules are all assumed computational, except for the implicit heating/cooling rules that define evaluation strategies of language constructs, which are assumed structural. If you want to explicitly make a rule structural, then you should include the tag (or attribute) `structural` in square brackets right after the rule. These attributes may be taken into account by different K tools, so it is highly recommended to spend a moment or two after each rule and think whether you want it to be structural or computational.

Let us do it. We want the lookup and the arithmetic and Boolean construct

rules to be computational, because they make computational progress whenever they apply. However, the block rules can be very well structural, because we can regard them simply as syntactic grouping constructs. In general, we want to have as few computational rules as possible, because we want the resulting transition systems to be smaller for analysis purposes, but not too few to lose behaviors. For example, making the block rules structural loses no meaningful behaviors. Similarly, the sequential composition, the while loop unrolling, and the no-variable declaration rules can all safely be structural.

Kompile and then krun the programs that you only parsed in Lesson 1. They should all execute as expected. The state cell shows the final state of the program. The `k` cell shows the final code contents, which should be empty whenever the IMP program executes correctly.

Kompile also with the documentation option and take a look at the generated documentation. The assignment rule should particularly be of interest, because it contains two local rewrites.

In the next lesson we comment the IMP definition and conclude this tutorial.

Go to Lesson 5, IMP: Completing and Documenting IMP.

MOVIE (out of date) [09'16"]

## Completing and Documenting IMP

We here learn no new concepts, but it is a good moment to take a break and contemplate what we learned so far.

Let us add lots of formal annotations to `imp.k`.

Once we are done with the annotations, we kompile with the documentation option and then take a look at the produced document. We often call these documents *language posters*. Depending on how much information you add to these language posters, they can serve as standalone, formal presentations of your languages. For example, you can print them as large posters and post them on the wall, or in poster sessions at conferences.

This completes our second tutorial. The next tutorials will teach us more features of the K framework, such as how to define languages with complex control constructs (like `callcc`), languages which are concurrent, and so on.

MOVIE (out of date) [03'45"]

# Part 3: Defining LAMBDA++

Here you will learn how to define language constructs which abruptly change the execution control flow, and how to define language semantics following

and environment/store style. Specifically, you will learn the following:

- How to define constructs like `callcc`, which allow you to take snapshots of
  program executions and to go *back in time* at any moment.
- How to define languages in an environment/store style.
- Some basic notions about the use of closures and closure-like semantic structures to save and restore execution environments.
- Some basic intuitions about reusing existing semantics in new languages, as well as some of the pitfalls in doing so.

## Abrupt Changes of Control

Here we add *call-with-current-continuation* (`callcc`) to the definition of LAMBDA completed in Tutorial 1, and call the resulting language LAMBDA++.
While doing so, we will learn how to define language constructs that abruptly change the execution control flow.

Take over the `lambda.k` definition from Lesson 8 in Part 1 of this Tutorial, which is the complete definition of the LAMBDA language, but without the comments.

`callcc` is a good example for studying the capabilities of a framework to support abrupt changes of control, because it is one of the most control-intensive language constructs known. Scheme is probably the first programming language that incorporated the `callcc` construct, although similar constructs have been recently included in many other languages in one form or another.

Here is a quick description: `callcc e` passes the remaining computation context, packaged as a function `k`, to `e` (which is expected to be a function); if during its evaluation `e` passes any value to `k`, then the current execution context is discarded and replaced by the one encoded by `k` and the value is passed to it; if `e` evaluates normally to some value `v` and passes nothing to `k` in the process, then `v` is returned as a result of `callcc e` and the execution continues normally. For example, we want the program `callcc-jump.lambda`:

```
(callcc (lambda k . ((k 5) + 2))) + 10
```

to evaluate to `15`, not `17`! Indeed, the computation context `[] + 10` is passed to `callcc`'s argument, which then sends it a `5`, so the computation resumes to `5 + 10`. On the other hand, the program `callcc-not-jump.lambda`

```
(callcc (lambda k . (5 + 2))) + 10
```

evaluates to `17`.

If you like playing games, you can metaphorically think of `callcc e` as *saving your game state in a file and passing it to your friend* `e`. Then `e` can decide at some moment to drop everything she was doing, load your game and continue to play it from where you were.

The behavior of many popular control-changing constructs can be obtained using `callcc`. The program `callcc-return.lambda` shows, for example, how to obtain the behavior of a `return` statement, which exits the current execution context inside a function and returns a value to the caller's context:

```
letrec f x = callcc (lambda return . (
  f (if (x <= 0) then ((return 1) / 0) else 2)
))
in (f -3)
```

This should evaluate to `1`, in spite of the recursive call to `f` and of the division by zero! Note that `return` is nothing but a variable name, but one which is bound to the current continuation at the beginning of the function execution. As soon as `1` is passed to `return`, the computation jumps *back in time* to where `callcc` was defined! Change `-3` to `3` and the program will loop forever.

`callcc` is quite a powerful and beautiful language construct, although one which is admittedly hard to give semantics to in some frameworks. But not in K  Here is the entire K syntax and semantics of `callcc`:

```
syntax Exp ::= "callcc" Exp  [strict]
syntax Val ::= cc(K)
rule <k> (callcc V:Val => V cc(K)) ~> K </k>
rule <k> cc(K) V ~> _ =>  V ~> K </k>
```

Let us first discuss the annotated syntax. We declared `callcc` strict, because its argument may not necessarily be a function yet, so it may need to be evaluated. As explained above, we need to encode the remaining computation somehow and pass it to `callcc`'s argument. More specifically, since LAMBDA is call-by-value, we have to encode the remaining computation as a value. We do not want to simply subsort computations to `Val`, because there are computations which we do not want to be values. A simple solution to achieve our goal here is to introduce a new value construct, say `cc` (from *current-continuation*), which holds any computation.

Note that, inspired from SDF, K allows you to define the syntax of helping semantic operations, like `cc`, more compactly. Typically, we do not need a fancy syntax for such operators; all we need is a name, followed by open parenthesis, followed by a comma-separated list of arguments, followed by closed parenthesis. If this is the syntax that you want for a particular construct, then K allows you to

drop all the quotes surrounding the terminals, as we did above for `cc`.

The semantic rules do exactly what the English semantics of `callcc` says. Note that here, unlike in our definition of LAMBDA in Tutorial 1, we had to mention the cell `<k/>` in our rules. This is because we need to make sure that we match the entire remaining computation, not only a fragment of it! For example, if we replace the two rules above with

```
rule (callcc V:Val => V cc(K)) ~> K
rule cc(K) V ~> _ =>  V ~> K
```

then we get a `callcc` which is allowed to non-deterministically pick a prefix of the remaining computation and pass it to its argument, and then when invoked within its argument, a non-deterministic prefix of the new computation is discarded and replaced by the saved one. Wow, that would be quite a language! Would you like to write programs in it?

Consequently, in K we can abruptly change the execution control flow of a program by simply changing the contents of the `<k/>` cell. This is one of the advantages of having an explicit representation of the execution context, like in K or in reduction semantics with evaluation contexts. Constructs like `callcc` are very hard and non-elegant to define in frameworks such as SOS, because those implicitly represent the execution context as proof context, and the latter cannot be easily changed.

Now that we know how to handle cells in configurations and use them in rules, in the next lesson we take a fresh look at LAMBDA and define it using an environment-based style, which avoids the complexity of substitution (e.g., having to deal with variable capture) and is closer in spirit to how functional languages are implemented.

Go to Lesson 2, LAMBDA++: Semantic (Non-Syntactic) Computation Items.

MOVIE (out of date) [6'28"]

## Semantic (Non-Syntactic) Computation Items

In this lesson we start another semantic definition of LAMBDA++, which follows a style based on environments instead of substitution. In terms of K, we will learn how easy it is to add new items to the syntactic category of computations `K`, even ones which do not have a syntactic nature.

An environment binds variable names of interest to locations where their values are stored. The idea of environment-based definitions is to maintain a global *store* mapping locations to values, and then have environments available when we evaluate expressions telling where the variables are located in the store. Since LAMBDA++ is a relatively simple language, we only need to maintain one global environment. Following a similar style like in IMP, we place all cells into a top cell `T`:

```
configuration <T>
                <k> $PGM:Exp </k>
                <env> .Map </env>
                <store> .Map </store>
              </T>
```

Recall that `$PGM` is where the program is placed by `krun` after parsing. So the program execution starts with an empty environment and an empty store.

In environment-based definitions of lambda-calculi, lambda abstractions evaluate to so-called *closures*:

```
rule <k> lambda X:Id . E => closure(Rho,X,E) ...</k>
     <env> Rho </env>
```

A closure is like a lambda abstraction, but it also holds the environment in which it was declared. This way, when invoked, a closure knows where to find in the store the values of all the variables that its body expression refers to. We will define the lookup rule shortly.

Therefore, unlike in the substitution-based definitions of LAMBDA and LAMBDA++, neither the lambda abstractions nor the identifiers are values anymore here, because they both evaluate further: lambda abstractions to closures and identifiers to their values in the store. In fact, the only values at this moment are the closures, and they are purely semantic entities, which cannot be used explicitly in programs. That's why we modified the original syntax of the language to include no `Val` syntactic category anymore, and that's why we need to add closures as values now; same like before, we add a `Val` syntactic category which is subsorted to `KResult`. In general, whenever you have any strictness attributes, your should also define some K results.

Invoking a closure is a bit more involved than the substitution-based beta-reduction: we need to switch to the closure's environment, then create a new, or fresh, binding for the closure's parameter to the value passed to the closure, then evaluate the closure's body, and then switch back to the caller's environment, which needs to be stored somewhere in the meanwhile. We can do all these with one rule:

```
rule <k> closure(Rho,X,E) V:Val => E ~> Rho' ...</k>
     <env> Rho' => Rho[X <- !N] </env>
     <store>... .Map => (!N:Int |-> V) ...</store>
```

Therefore, we atomically do all the following:

- switch the computation to the closure's body, `E`, followed by a caller-environment-recovery task `Rho'` (note that `Rho'` is the current environment),
- generate a fresh location `!N` (the `!` is important, we discuss it below), bind `X` to `!N` in closure's environment and switch the current environment

`Rho'` to that one,
- write the value passed to the closure, `V`, at location `!N`.

This was the most complex K rule we've seen so far in the tutorial. Note, however, that this one rule achieves a lot. It is, in fact, quite compact considering how much it does. Note also that everything that this K rule mentions is needed also conceptually in order to achieve this task, so it is minimal from that point of view. That would not be the case if we used, instead, a conventional rewrite rule, because we would have had to mention the remaining store, say `Sigma`, in both sides of the rule, to say it stays unchanged. Here we just use `...`.

The declaration of the *fresh* variable above, `!N`, is new and needs some explanation. First, note that `!N` appears only in the right-hand-side terms in the rule, that is, it is not matched when the rule is applied. Instead, a fresh `Nat` element is generated each time the rule is applied. In K, we can define syntactic categories which have the capability to generate fresh elements like above, using unbound variables whose name starts with a `!`. The details of how to do that are beyond the scope of this tutorial (see Tutorial 6). All we need to know here is that an arbitrary fresh element of that syntactic category is generated each time the rule is applied. We cannot rely on the particular name or value of the generated element, because that can change with the next version of the K tool, or even from execution to execution with the same version. All you can rely on is that each newly generated element is distinct from the previously generated elements for the same syntactic category.

Unlike in the substitution-based definition, we now also need a lookup rule:

```
rule <k> X => V ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> V ...</store>
```

This rule speaks for itself: replace `X` by the value `V` located in the store at `X`'s location `N` in the current environment.

The only thing left to define is the auxiliary environment-recovery operation:

rule _:Val ~> (Rho => .) ... _ => Rho

When the item preceding the environment recovery task `Rho` in the computation becomes a value, replace the current environment with `Rho` and dissolve `Rho` from the computation.

Before we kompile, let us make this rule and the lambda evaluation rule structural, because we do not want these to count as transitions.

Let us kompile and ... fail:

```
kompile lambda
```

gives a parsing error saying that `V:Val` does not fit there in the closure invocation rule. That's because `Val` and `Exp` are currently completely disconnected, so K rightfully complains that we want to apply a value to another one, because application was defined to work with expressions, not values. What we forgot here was to state that `Exp` includes `Val`:

```
syntax Exp ::= Val
```

Now everything works, but it is a good time to reflect a bit.

So we added closures, which are inherently semantic entities, to the syntax of expressions. Does that mean that we can now write LAMBDA programs with closures in them? Interestingly, with our current definition of LAMBDA, which purposely did not follow the nice organization of IMP into syntax and semantic modules, and with K's default parser, `kast`, you can. But you are not supposed to speculate this! In fact, if you use an external parser, that parser will reject programs with explicit closures. Also, if we split the LAMBDA definition into two modules, one called LAMBDA-SYNTAX containing
exclusively the desired program syntax and one called LAMBDA importing the former and defining the syntax of the auxiliary operations and the semantics, then even K's default parser will reject programs using auxiliary syntactic constructs.

Indeed, when you kompile a language, say `lang.k`, the tool will by default attempt to find a module LANG-SYNTAX and generate the program parser from
that. If it cannot find it, then it will use the module LANG instead. There are also ways to tell kompile precisely which syntax module you want to use for the program parser if you don't like the default convention.
See `kompile --help`.

Another insightful thought to reflect upon, is the relationship between your language's values and other syntactic categories. It is often the case that values form a subset of the original language syntax, like in IMP (Part 2 of the tutorial), but sometimes that is not true, like in our case here. When that happens, in order for the semantics to be given smoothly and uniformly using the original syntax, you need to extend your language's original syntactic categories with the new values. The same holds true in other semantic approaches, not only in K, even in ones which are considered purely syntactic. As it should be clear by now, K does not enforce you to use a purely syntactic style in your definitions; nevertheless, K does allow you to develop purely syntactic definitions, like LAMBDA in Part 1 of the tutorial, if you prefer those.

`krun` some programs, such as those provided in Lesson 1 of the LAMBDA tutorial (Part 1). Note the closures, both as results in the `<k/>` cell, and as values in the store. Also, since variables are not values anymore, expressions that contain free variables may get stuck with one of those on

top of their computation. See, for example, `free-variable-capture.lambda`, which gets stuck on `z`, because `z` is free, so it cannot evaluate it.
If you want, you can go ahead and manually provide a configuration with `z` mapped to some location in the environment and that location mapped to some value in the store, and then you can also execute this program. The program `omega.lambda` should still loop.

Although we completely changed the definitional style of LAMBDA, the semantics
of the other constructs do not need to change, as seen in the next lesson.

Go to Lesson 3, LAMBDA++: Reusing Existing Semantics.

MOVIE (out of date) [8'02"]

## Reusing Existing Semantics

In this lesson we will learn that, in some cases, we can reuse existing semantics of language features without having to make any change!

Although the definitional style of the basic LAMBDA language changed quite radically in our previous lesson, compared to its original definition in Part 1 of the tutorial, we fortunately can reuse a large portion of the previous definition. For example, let us just cut-and-paste the rest of the definition from Lesson 7 in Part 1 of the tutorial.

Let us `kompile` and `krun` all the remaining programs from Part 1 of the tutorial. Everything should work fine, although the store contains lots of garbage. Garbage collection is an interesting topic, but we do not do it here. Nevertheless, much of this garbage is caused by the intricate use of the fixed-point combinator to define recursion. In a future lesson in this tutorial we will see that a different, environment-based definition of fixed-points will allocate much less memory.

One interesting question at this stage is: how do we know when we can reuse an existing semantics of a language feature? Well, I'm afraid the answer is: we don't. In the next lesson we will learn how reuse can fail for quite subtle reasons, which are impossible to detect statically (and some non-experts may fail to even detect them at all).

Go to Lesson 4, LAMBDA++: Do Not Reuse Blindly!.

MOVIE (out of date) [3'21"]

## Do Not Reuse Blindly!

It may be tempting to base your decision to reuse an existing semantics of a language feature solely on syntactic considerations; for example, to reuse

whenever the parser does not complain. As seen in this lesson, this could be quite risky.

Let's try (and fail) to reuse the definition of `callcc` from Lesson 1:

```
syntax Exp ::= "callcc" Exp  [strict]
syntax Val ::= cc(K)
rule <k> (callcc V:Val => V cc(K)) ~> K </k>
rule <k> cc(K) V ~> _ =>  V ~> K </k>
```

The `callcc` examples that we tried in Lesson 1 work, so it may look it works.

However, the problem is that `cc(K)` should also include an environment, and that environment should also be restored when `cc(K)` is invoked. Let's try to illustrate this bug with `callcc-env1.lambda`

```
let x = 1 in
  ((callcc lambda k . (let x = 2 in (k x))) + x)
```

where the second argument of `+`, `x`, should be bound to the top `x`, which is 1. However, since `callcc` does not restore the environment, that `x` should be looked up in the wrong, callcc-inner environment, so we should see the overall result 4.

Hm, we get the right result, 3 ... (Note: you may get 4, depending on your version of K and platform; but both 3 and 4 are possible results, as explained below and seen in the tests). How can we get 3? Well, recall that `+` is strict, which means that it can evaluate its arguments in any order. It just happened that in the execution that took place above its second argument was evaluated first, to 1, and then the `callcc` was evaluated, but its `cc` value K had already included the 1 instead of `x` ... In Part 4 of the tutorial we will see how to explore all the non-deterministic behaviors of a program; we could use that feature of K to debug semantics, too. For example, in this case, we could search for all behaviors of this program and we would indeed get two possible value results: 3 and 4.

One may think that the problem is the non-deterministic evaluation order of `+`, and thus that all we need to do is to enforce a deterministic order in which the arguments of + are evaluated. Let us follow this path to see what happens. There are two simple ways to make the evaluation order of `+`'s arguments deterministic. One is to make `+ seqstrict` in the semantics, to enforce its evaluation from left-to-right. Do it and then run the program above again; you should get only one behavior for the program above, 4, which therefore shows that copying-and-pasting our old definition of `callcc` was incorrect. However, as seen shortly, that only fixed the problem for the particular example above, but not in general. Another conventional approach to enforce the desired evaluation order is to modify the program to enforce the left-to-right evaluation order using let binders, as we do in `callcc-env2.lambda`:

```
let x = 1 in
  let a = callcc lambda k . (let x = 2 in (k x)) in
    let b = x in
      (a + b)
```

With your installation of K you may get the "expected" result 4 when you execute this program, so it may look like our non-deterministic problem is fixed. Unfortunately, it is not. Using the K tool to search for all the behaviors in the program above reveals that the final result 3 is still possible. Moreover, both the 3 and the 4 behaviors are possible regardless of whether `+` is declared to be `seqstrict` or just `strict`. How is that possible? The problem is now the non-deterministic evaluation strategy of the function application construct. Indeed, recall that the semantics of the let-in construct is defined by desugaring to lambda application:

```
rule let X = E in E' => (lambda X . E') E     [macro]
```

With this, the program above eventually reduces to

```
(lambda a . ((lambda b . a + b) x))
(callcc lambda k . (let x = 2 in (k x)))
```

in an environment where `x` is 1. If the first expression evaluates first, then it does so to a closure in which `x` is bound to a location holding 1, so when applied later on to the `x` inside the argument of `callcc` (which is 2), it will correctly lookup `x` in its enclosed environment and thus the program will evaluate to 3. On the other hand, if the second expression evaluates first, then the `cc` value will freeze the first expression as is, breaking the relationship between its `x` and the current environment in which it is bound to 1, being inadvertently captured by the environment of the let-in construct inside the `callcc` and thus making the entire expression evaluate to 4.

So the morale is: Do not reuse blindly. *Think!*

In the next lesson we fix the environment-based semantics of `callcc` by having `cc` also wrap an environment, besides a computation. We will also give a more direct semantics to recursion, based on environments instead of fixed-point combinators.

Go to Lesson 5, LAMBDA++: More Semantic Computation Items.

MOVIE (out of date) [3'37"]

## More Semantic Computation Items

In this lesson we see more examples of semantic (i.e., non-syntactic) computational items, and how useful they can be. Specifically, we fix the

environment-based definition of `callcc` and give an environment-based definition of the `mu` construct for recursion.

Let us first fix `callcc`. As discussed in Lesson 4, the problem that we noticed there was that we only recovered the computation, but not the environment, when a value was passed to the current continuation. This is quite easy to fix: we modify `cc` to take both an environment and a computation, and its rules to take a snapshot of the current environment with it, and to recover it at invocation time:

```
syntax Val ::= cc(Map,K)
rule <k> (callcc V:Val => V cc(Rho,K)) ~> K </k> <env> Rho </env>
rule <k> cc(Rho,K) V:Val ~> _ =>  V ~> K </k> <env> _ => Rho </env>
```

Let us kompile and make sure it works with the `callcc-env2.lambda` program, which should evaluate to 3, not to 4.

Note that the `cc` value, which can be used as a computation item in the `<k/>` cell, is now quite semantic in nature, pretty much the same as the closures.

Let us next add one more closure-like semantic computational item, for `mu`. But before that, let us reuse the semantics of `letrec` in terms of `mu` that was defined in Lesson 8 of Part 1 of the tutorial on LAMBDA:

```
syntax Exp ::= "letrec" Id Id "=" Exp "in" Exp
             | "mu" Id "." Exp      [latex(\mu{#1}.{#2})]
rule letrec F:Id X = E in E' => let F = mu F . lambda X . E in E'    [macro]
```

We removed the `binder` annotation of `mu`, because it is not necessary anymore (since we do not work with substitutions anymore).

To save the number of locations needed to evaluate `mu X . E`, let us replace it with a special closure which already binds `X` to a fresh location holding the closure itself:

```
syntax Exp ::= muclosure(Map,Exp)

rule <k> mu X . E => muclosure(Rho[X <- !N], E) ...</k>
     <env> Rho </env>
     <store>... .Map => (!N:Int |-> muclosure(Rho[X <- !N], E)) ...</store>
  [structural]
```

Since each time `mu X . E` is encountered during the evaluation it needs to evaluate `E`, we conclude that `muclosure` cannot be a value. We can declare it as either an expression or as a computation. Let's go with the former.

Finally, here is the rule unrolling the `muclosure`:

rule muclosure(Rho,E) => E ~> Rho' ...
Rho' => Rho

Note that the current environment `Rho'` needs to be saved before and restored after `E` is executed, because the fixed point may be invoked from a context with a completely different environment from the one in which `mu X . E` was declared.

We are done. Let us now `kompile` and `krun factorial-letrec.lambda` from Lesson 7 in Part 1 of the tutorial on LAMBDA. Recall that in the previous lesson this program generated a lot of garbage into the store, due to the need to allocate space for the arguments of all those lambda abstractions needed to run the fixed-point combinator. Now we need much fewer locations, essentially only locations for the argument of the factorial function, one at each recursive call. Anyway, much better than before.

In the next lesson we wrap up the environment definition of LAMBDA++ and generate its documentation.

Go to Lesson 6, LAMBDA++: Wrapping Up and Documenting LAMBDA++.

MOVIE (out of date) [5'19"]

## Wrapping Up and Documenting LAMBDA++

In this lesson we wrap up and nicely document LAMBDA++. In doing so, we also
take the freedom to reorganize the semantics a bit, to make it look better.

See the `lambda.k` file, which is self-explanatory.

Part 3 of the tutorial is now complete. Part 4 will teach you more features of the K framework, in particular how to exhaustively explore the behaviors of non-deterministic or concurrent programs.

MOVIE (out of date) [6'23"]

# Part 4: Defining IMP++

IMP++ extends IMP, which was discussed in Part 2 of this tutorial, with several new syntactic constructs. Also, some existing syntax is generalized, which requires non-modular changes of the existing IMP semantics. For example, global variable declarations become local declarations and can occur anywhere a statement can occur. In this tutorial we will learn the following:

- That (and how) existing syntax/semantics may change as a language evolves.
- How to refine configurations as a language evolves.
- How to define and use fresh elements of desired sorts.
- How to tag syntactic constructs and rules, and how to use such tags with the `superheat`/`supercool`/`transition` options of `kompile`.

- How the `search` option of `krun` works.
- How to stream cells holding semantic lists to the standard input/output, and thus obtain interactive interpreters for the defined languages.
- How to delete, save and restore cell contents.
- How to add/delete cells dynamically.
- More details on how the configuration abstraction mechanism works.

Like in the previous tutorials, this folder contains several lessons, each adding new features to IMP++. Do them in order and make sure you completed and understood the previous tutorials.

## Extending/Changing an Existing Language Syntax

Here we learn how to extend the syntax of an existing language, both with new syntactic constructs and with more general uses of existing constructs. The latter, in particular, requires changes of the existing semantics.

Consider the IMP language, as defined in Lesson 4 of Part 2 of the tutorial.

Let us first add the new syntactic constructs, with their precedences:

- variable increment, `++`, which increments an integer variable and evaluates to the new value;
- `read`, which reads and evaluates to a new integer from the input buffer;
- `print`, which takes a comma-separated list of arithmetic expressions and evaluates and prints each of them in order, from left to right, to the output buffer; we therefore define a new list syntactic category, `AExps`, which we pass as an argument to `print`; note we do not want to declare `print` to be `strict`, because we do not want to first evaluate the arguments and then print them (for example, if the second argument performs
  an illegal operation, say division by zero, we still want to print the first argument); we also go ahead and add strings as arithmetic expressions, because we intend print to also take strings, in order to print nice messages to the user;
- `halt`, which abruptly terminates the program; and
- `spawn`, which takes a statement and creates a new concurrent thread executing it and sharing its environment with the parent thread.

Also, we want to allow local variable declarations, which can appear anywhere a statement can appear. Their scope ranges from the place they are defined until the end of the current block, and they can shadow previous declarations, both inside and outside the current block. The simplest way to define the syntax of the new variable declarations is as ordinary statements, at the same time removing the previous `Pgm` syntactic category and its construct. Programs are now just statements.

We are now done with adding the new syntax and modifying the old one. Note that the old syntax was modified in a way which makes the previous IMP programs still parse, but this time as statements. Let us then modify the configuration variable `$PGM` to have the sort `Stmt` instead of `Pgm`, and let us try to run the old IMP programs, for example `sum.imp`.

Note that they actually get stuck with the *global* declaration on the top of their computations. This is because variable declarations are now treated like any statements, in particular, the sequential composition rule applies. This makes the old IMP rule for global variable declarations not match anymore. We can easily fix it by replacing the anonymous variable `_`, which matched the program's statement that now turned into the remaining computation in the `<k/>` cell, with the cell frame variable `...`, which matches the remaining computation. Similarly, we have to change the rule for the case where there are no variables left to declare into one that dissolves itself.

We can now run all the previous IMP programs, in spite of the fact that our IMP++ semantics is incomplete and, more interestingly, in spite of the fact that our current semantics of blocks is incorrect in what regards the semantics of local variable declarations (note that the old IMP programs do not declare block-local variables, which is why they still run correctly).

Let us also write some proper IMP++ programs, which we would like to execute once we give semantics to the new constructs.

`div.imp` is a program manifesting non-deterministic behaviors due to the desired non-deterministic evaluation strategy of division and the fact that expressions will have side effects once we add variable increment. We will be able to see all the different behaviors of this program. Challenge: can you identify the behavior where the program performs a division-by-zero?

If we run `div.imp` now, it will get stuck with the variable increment construct on top of the computation cell. Once we give it a semantics, `div.imp` will execute completely (all the other constructs in `div.imp` already have their semantics defined as part of IMP).

Note that some people prefer to define all their semantics in a *by need* style, that is, they first write and parse lots of programs, and then they add semantics to each language construct on which any of the programs gets stuck, and so on and so forth until they can run all the programs.

`io.imp` is a program which exercises the input/output capabilities of the language: reads two integers and prints three strings and an integer. Note that the variable declaration is not the first statement anymore.

`sum-io.imp` is an interactive variant of the sum program.

`spawn.imp` is a program which dynamically creates two threads that interact with the main thread via the shared variable x. Lots of behaviors will be seen here once we give spawn the right semantics.

Finally, `locals.imp` tests whether variable shadowing/unshadowing works well.

In the next lesson we will prepare the configuration for the new constructs, and will see what it takes to adapt the semantics to the new configuration. Specifically, we will split the state cell into an environment cell and a store cell, like in LAMBDA++ in Part 3 of the tutorial.

Go to Lesson 2, IMP++: Configuration Refinement; Freshness.

MOVIE (out of date) [07'47"]

## Configuration Refinement; Freshness

To prepare for the semantics of threads and local variables, in this lesson we split the state cell into an environment and a store. The environment and the store will be similar to those in the definition of LAMBDA++ in Part 3 of the Tutorial. This configuration refinement will require us to change some of IMP's rules, namely those that used the state.

To split the state map, which binds program variables to values, into an environment mapping program variables to locations and a store mapping locations to values, we replace in the configuration declaration the cell

```
<state color="red"> .Map </state>
```

with two cells

```
<env color="LightSkyBlue"> .Map </env>
<store color="red"> .Map </store>
```

Structurally speaking, this split of a cell into other cells is a major semantic change, which, unfortunately, requires us to revisit the existing rules that used the state cell. One could, of course, argue that we could have avoided this problem if we had followed from the very beginning the good-practice style to work with an environment and a store, instead of a monolithic state. While that is a valid argument, highlighting the fact that modularity is not only a feature of the framework alone, but one should also follow good practices to achieve it, it is also true that if all we wanted in Part 2 of the tutorial was to define IMP as is, then the split of the state in an environment and a store is unnecessary and not really justified.

The first rule which used a state cell is the lookup rule:

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

We modify it as follows:

```
rule <k> X:Id => I ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> I ...</store>
```

So we first match the location `N` of `X` in the environment, then the value `I` at location `N` in the store, and finally we rewrite `X` to `I` into the computation. This rule also shows an instance of a more complex multiset matching, where two variables (`X` and `N`) are matched each twice.

The assignment rule is modified quite similarly.

The variable declaration rule is trickier, though, because we need to allocate a fresh location in the store and bind the newly declared variable to it. This is quite similar to the way we allocated space for variables in the environment-based definition of LAMBDA++ in Part 3 of the tutorial.

```
rule <k> int (X,Xs => Xs); ...</k>
     <env> Rho => Rho[X <- !N:Int] </env>
     <store>... .Map => !N |-> 0 ...</store>
```

Note the use of the fresh (`!N`) variable notation above. Recall from the LAMBDA++ tutorial that each time the rule with fresh (`!`) variables is applied, fresh elements of corresponding sorts are generated for the fresh variables, distinct from all the previously generated elements; also, we cannot and should not assume anything about the particular element that is being generated, except that it is different from the previous ones.

`kompile` and `krun sum.imp` to see how the fresh locations have been generated and used. There were two fresh locations needed, for the two variables. Note also that a cell holding the counter has been added to the configuration.

In the next lesson we will add the semantics of variable increment, and see how that yields non-deterministic behaviors in programs and how to explore those behaviors using the K tool.

Go to Lesson 3, IMP++: Tagging; Superheat/Supercool Kompilation Options.

MOVIE (out of date) [04'06"]

## Tagging; Transition Kompilation Option

In this lesson we add the semantics of variable increment. In doing so, we learn how to tag syntactic constructs and rules and then use such tags to instruct the `kompile` tool to generate the desired language model that is amenable for exhaustive analysis.

The variable increment rule is self-explanatory:

```
rule <k> ++X => I +Int 1 ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> (I => I +Int 1) ...</store>
```

We can now run programs like our `div.imp` program introduced in Lesson 1. Do it.

The addition of increment makes the evaluation of expressions have side effects. That, in combination with the non-determinism allowed by the strictness attributes in how expression constructs evaluate their arguments, makes expressions in particular and programs in general have non-deterministic behaviors. One possible execution of the `div.imp` program assigns `1` to `y`'s location, for example, but this program manifests several other behaviors, too.

To see all the (final-state) behaviors that a program can have, you can call the `krun` tool with the option `--search`. For example:

```
krun div.imp --search
```

Oops, we see only one solution, the same as when we ran it without search.

Here is what happens. `krun` can only explore as much of the transition system associated to a program as `kompile` allowed the generated language model to yield. Since most of the K users are interested in language models that execute efficiently, that is, in faster interpreters for the defined languages, by default `kompile` optimizes the generated language model for execution. In particular, it inserts no backtracking markers, which `krun` uses when called with the `--search` option in order to systematically generate the entire transition system associated to a program. This is why `krun` showed us only one solution when run with the `--search` option on `div.imp`.

We next explain how to tell `kompile` what kind of language model we are interested in for analysis purposes. When you experiment with non-determinism in a language semantics, you should keep it in mind that the `--transition` option of `kompile` allows you to configure what counts as a transition in your language model. We here only discuss transitions due to the non-deterministic evaluation strategies of language constructs, but we will see in future lectures (see Lesson 6 of IMP++, where we add concurrency) that we can also have transitions due to non-deterministic applications of rewrite rules.

If you want to explore the entire behavior space due to non-deterministic evaluation strategies, then you should include all the language constructs in the `--transition` option. This may sound like the obvious thing to always do, but as soon as you do it you soon realize that it is way too much in practice when you deal with large languages or programs. There are simply too many program behaviors to consider, and `krun` will likely hang on you or crush. For example, a small ten-statement program where each statement uses one strict expression construct already has 1000+ behaviors for `krun` to explore! Driven by practical needs of its users, the K tool therefore allows you to finely tune the generated language models using the `--transition` option.

To state which constructs are to be considered to generate transitions in the generated language model, and for other reasons, too, the K tool allows you to tag any production and any rule. You can do this the same way we tagged rules with the `structural` keyword in earlier tutorials: put the tag in brackets. You can associate multiple tags to the same construct or rule, and more than one construct or rule can have the same tag. As an example, let us tag the division construct with `division`, the lookup rule with `lookup` and the increment rule with `increment`. The tags of the rules are not needed in this lesson, we do it only to demonstrate that rules can also be tagged.

The least intrusive way to enforce our current language to explore the entire space of behaviors due to the strictness of division is to `kompile` it with the following option:

```
kompile imp.k --transition "division"
```

It is interesting to note that the `lookup` and `increment` rules are the only two rules which can trigger non-deterministic behaviors for division, because no other rule but these two can ever apply while a division operation is heated. Previous versions of K allowed you to also specify which rules could trigger non-deterministic behaviors of operator evaluation strategies, but that option was rarely used and is not available anymore.

Note that it is highly non-trivial to say precisely whether a strict language construct may yield non-deterministic behaviors. For example, division's strictness would yield no non-determinism if the language had no side effects. It is even harder to say so for a particular program. Consequently, our K implementation makes no attempt to automatically detect which operations should be tagged as transitions. Instead, it provides the functionality to let you decide it.

Now the command

```
krun div.imp --search
```

shows us all five behaviors of this program. Interestingly, one of the five behaviors yields a division by zero!

The `--transition` option can be quite useful when you experiment with your language designs or when you formally analyze programs for certain kinds of errors. Please let us know if you ever need more finer-grained control over the non-determinism of your language models.

Before we conclude this lesson, we'd like to let you know one trick, which you will hopefully not overuse: you can tag elements in your K definition with `kompile` option names, and those elements will be automatically included in their corresponding options. For example, if you tag the division production with `transition` then the command

```
kompile imp
```

is completely equivalent to the previous `kompile` command.

Please use this default behavior with caution, or even better, try to avoid using it! You may be tempted to add the `transition` tag to lots of elements and then forget about them; your language models will then be increasingly slower
when you execute them and you may wonder why ... This convention is typically convenient when you want to quickly experiment with non-determinism and do not
want to bother inventing tag names and calling `kompile` with options.

In the next lesson we add input/output to our language and learn how to generate a model of it which behaves like an interactive interpreter!

Go to Lesson 4, IMP++: Semantic Lists; Input/Output Streaming.

MOVIE (out of date) [06'56"]

## Semantic Lists; Input/Output Streaming

In this lesson we add semantics to the `read` and `print` IMP++ constructs. In doing so, we also learn how to use semantic lists and how to connect cells holding semantic lists to the standard input and standard output. This allows us to turn the K semantics into an interactive interpreter.

We start by adding two new cells to the configuration,

```
<in color="magenta"> .List </in>
<out color="Orchid"> .List </out>
```

each holding a semantic list, initially empty. Semantic lists are space-separated sequences of items, each item being a term of the form `ListItem(t)`, where `t` is a term of sort `K`. Recall that the semantic maps, which we use for states, environments, stores, etc., are sets of pairs `t1 |-> t2`, where `t1` and `t2` are terms of sort K. The `ListItem` wrapper is currently needed, to avoid parsing ambiguities.

Since we want the `print` statement to also print strings, we need to tell K that strings are results. To make it more interesting, let us also overload the `+` symbol on arithmetic expressions to also take strings and, as a result, to concatenate them. Since `+` is already strict, we only need to add a rule reducing the IMP addition of strings to the builtin operation `+String` which concatenates two strings.

The semantics of `read` is immediate: read and consumes the first integer item from the `<in/>` cell; note that our read only reads integer values (it gets stuck if the first item in the `<in/>` cell is not an integer).

The semantics of `print` is a bit trickier. Recall that `print` takes an arbitrary number of arithmetic expression arguments, and evaluates and outputs

each of them in order, from left to right. For example, `print("Hello", 3/0, "Bye");` outputs "Hello" and then gets stuck on the illegal division by zero operation. In other words, we do not want it to first evaluate all its arguments and then print them, because that would miss outputting potentially valuable information. So the first step is to evaluate the first argument of `print`. In some sense, what we'd like to say is that `print` has the evaluation strategy `strict(1)`. However, strictness attributes only work with individual language constructs, while what we need is an evaluation strategy that involves *two* constructs: `print` and the list (comma) construct of `AExps`. If we naively associate `print` the `strict(1)` evaluation strategy then its first and unique argument, an `AExps` list, will be scheduled for evaluation and the execution will get stuck because we have no rules for evaluating `AExps` terms. If we make the list construct of `AExps` `strict` then we get the wrong semantics for `print` which first evaluates all its arguments and then outputs them. The correct way to tell K that `print` should evaluate only its first argument is by using a context declaration:

```
context print(HOLE:AExp, _);
```

Note the `HOLE` of sort `AExp` above. Contexts allow us to define finer-grain evaluation strategies than the strictness attributes, involving potentially more than one language construct, like above. The `HOLE` indicates the argument which is requested to be evaluated. For example, the `strict` attribute of division corresponds to two contexts:

```
context HOLE / _
context _ / HOLE
```

In their full generality, contexts can be any terms with precisely one occurrence of a `HOLE`, and with arbitrary side conditions on any variables occurring in the context term as well as on the `HOLE`. See Part 6 of the tutorial for more examples.

Once evaluated, the first argument of `print` is expected to become either an integer or a string. Since we want to print both integers and string values, to avoid writing two rules, one for each type of value, we instead add a new syntactic category, `Printable`, which is the union of integers and strings.

Let us `kompile` and `krun` the `io.imp` program discussed in Lesson 1. As expected, it gets stuck with a read construct on top of the computation and with an empty `<in/>` cell. To run it, we need to provide some items in the `<in/>` cell, so that the rule of read can match. Let us add

```
<in> ListItem(3) ListItem(5) ListItem(7) </in>
```

Now, if we `krun io.imp`, we can see that its execution completes normally (the `<k/>` cell is empty), that the first two items have been removed by the two read constructs from the `<in/>` cell, and that the desired strings and numbers have been placed into the `<out/>` cell.

Cells holding semantic lists can be connected to the standard input and standard output buffers, and `krun` knows how to handle these appropriately. Let us connect the `<in/>` cell to the standard input using the cell attribute `stream="stdin"` and the `<out/>` cell to the standard output with the attribute `stream="sdtout"`. A cell connected to the standard input will take its items from the standard input and block the rewriting process when an input is needed until an item is available in the standard input buffer. A cell connected to the standard output buffer will send all its items, in order, to the standard output.

Let us `kompile` and `krun io.imp` again. It prints the message and then waits for your input numbers. Type in two numbers, then press `<Enter>`. A message with their sum is then printed, followed by the final configuration. If you do not want to see the final configuration, and thus obtain a realistic interpreter for our language, then call `krun` with the option `--output none`:

```
krun io.imp --output none
```

Let us now `krun` our interactive sum program, which continuously reads numbers from the console and prints the sum of numbers up to them:

```
krun sum-io.imp
```

Try a few numbers, then `0`. Note that the program terminated, but with junk in the `<k/>` cell, essentially with a `halt` statement on its top. Of course, because `halt` has been reached and it has no semantics yet.

In the next lesson we give the semantics of `halt` and also fix the semantics of blocks with local variable declarations.

Go to Lesson 5, IMP++: Deleting, Saving and Restoring Cell Contents.

MOVIE (out of date) [05'21"]

# Deleting, Saving and Restoring Cell Contents

In this lesson we will see how easily we can delete, save and/or restore contents of cells in order to achieve the desired semantics of language constructs that involve abrupt changes of control or environments. We have seen similar or related K features in the LAMBDA++ language in Part 3 of the tutorial.

Let us start by adding semantics to the `halt` statement. As its name says, what we want is to abruptly terminate the execution of the program. Moreover, we want the program configuration to look as if the program terminated normally, with an empty computation cell. The simplest way to achieve that is to simply empty the computation cell when `halt` is encountered:

```
rule <k> halt; ~> _ => . </k>
```

It is important to mention the entire `<k/>` cell here, with both its membranes closed, to make sure that its entire contents is discarded. Note the anonymous variable, which matches the rest of the computation.

`kompile` and `krun sum-io.imp`. Note that unlike in Lesson 4, the program terminates with an empty computation cell now.

As mentioned earlier, the semantics of blocks that was inherited from IMP is wrong. Program `locals.imp` shows it very clearly: the environments are not correctly restored at block exits. One way to fix the problem is to take a snapshot of the current environment when a block is entered and save it somewhere, and then to restore it when the block is left. There are many ways to do this, which you can explore on your own: for example you can add a new list cell for this task where to push/pop the environment snapshots in a stack style; or you can use the existing environment cell for this purpose, but then you need to change the variable access rules to search through the stacked environments for the variable.

My preferred solution is to follow a style similar to how we saved/restored LAMBDA++ environments in Part 3 of the Tutorial, namely to use the already existing `<k/>` cell for such operations. More specifically, we place a *reminder* item in the computation whenever we need to take a snapshot of some cell contents; the item simply consists of the entire contents of the cell. Then, when the reminder item is reached, we restore the contents of the cell:

```
rule <k> {S} => S ~> Rho ...</k> <env> Rho </env>  [structural]
```

The only thing left now is to give the definition of environment restore:

```
rule <k> Rho => . ...</k> <env> _ => Rho </env>    [structural]
```

Done. `kompile` and `krun locals.imp`. Everything should work correctly now. Note that the rule above is different from the one we had for LAMBDA++ in Part 3 of the tutorial, in that here there is no value preceding the environment restoration item in the computation; that's because IMP++ statements, unlike LAMBDA++'s expressions, evaluate to nothing (`.`).

In the next lesson we will give semantics to the `spawn S` construct, which dynamically creates a concurrent shared-memory thread executing statement `S`.

Go to Lesson 6, IMP++: Adding/Deleting Cells Dynamically; Configuration Abstraction, Part 2.

MOVIE (out of date) [04'30"]

# Adding/Deleting Cells Dynamically; Configuration Abstraction, Part 2

In this lesson we add dynamic thread creation and termination to IMP, and while doing so we learn how to define and use configurations whose structure can evolve dynamically.

Recall that the intended semantics of `spawn S` is to spawn a new concurrent thread that executes `S`. The new thread is being passed at creation time its parent's environment, so it can share with its parent the memory locations that its parent had access to at creation time. No other locations can be shared, and no other memory sharing mechanism is available. The parent and the child threads can evolve unrestricted, in particular they can change their environments by declaring new variables or shadowing existing ones, can create other threads, and so on.

The above suggests that each thread should have its own computation and its own environment. This can be elegantly achieved if we group the `<k/>` and `<env/>` cells in a `<thread/>` cell in the configuration. Since at any given moment during the execution of a program there could be zero, one or more instances of such a `<thread/>` cell in the configuration, it is a good idea to declare the `<thread/>` cell with multiplicity `*` (i.e., zero, one or more):

```
<thread multiplicity="*" color="blue">
  <k color="green"> $PGM:Stmt </k>
  <env color="LightSkyBlue"> .Map </env>
</thread>
```

This multiplicity declaration is not necessary, but it is a good idea to do it for several reasons:

1. it may help the configuration abstraction process, which may in turn significantly increase the compactness and modularity of your subsequent rules;
2. it may help various analysis and execution tools, for example static analyzers to give you error messages when you create cells where you should not, or K compilers to improve performance by starting actual concurrent hardware threads or processes corresponding to each cell instance; and
3. it may help you better understand and control the dynamics of your configuration, and thus your overall semantics.

For good encapsulation, I also prefer to put all thread cells into one cell, `<threads/>`. This is technically unnecessary, though; to convince yourself that this is indeed the case, you can remove this cell once we are done with

the semantics and everything will work without having to make any changes.

Before we continue, let us `kompile` an `krun` some programs that used to work, say `sum-io.imp`. In spite of the relatively radical configuration reorganization, those programs execute just fine! How is that possible? In particular, why do rules like the lookup and assignment still work, unchanged, in spite of the fact that the `<k/>` and `<env/>` cells are not at the same level with the `<store/>` cell in the configuration anymore?

Welcome to configuration abstraction, part 2. Recall that the role of configuration abstraction is to allow you to only write the relevant information in each rule, and have the compiler fill-in the obvious and boring details. According to the configuration that we declared for our new language, there is only one reasonable way to complete rules like the lookup, namely to place the `<k/>` and `</env>` cells inside a `<thread/>` cell, inside a `<threads/>` cell:

```
rule <threads>...
       <thread>...
         <k> X:Id => I ...</k>
         <env>... X |-> N ...</env>
       ...</thread>
     ...<threads/>
     <store>... N |-> I ...</store>  [lookup]
```

This is the most direct, compact and local way to complete the configuration context of the lookup rule. If for some reason you wanted here to match the `<k/>` cell of one thread and the `<env/>` cell of another thread, then you would need to explicitly tell K so, by mentioning the two thread cells, for example:

```
rule <thread>...
         <k> X:Id => I ...</k>
     ...</thread>
     <thread>...
         <env>... X |-> N ...</env>
     ...</thread>
     <store>... N |-> I ...</store>  [lookup]
```

By default, K completes rules in a greedy style. Think this way: what is the minimal number of changes to my rule to make it fit the declared configuration? That's what the K tool will do.

Configuration abstraction is technically unnecessary, but once you start using it and get a feel for how it works, it will become your best friend. It allows you to focus on the essentials of your semantics, and at the same time gives you flexibility in changing the configuration later on without having to touch the rules. For example, it allows you to remove the `<threads/>` cell from the configuration, if you don't like it, without

having to touch any rule.

We are now ready to give the semantics of `spawn`:

```
rule <k> spawn S => . ...</k> <env> Rho </env>
     (. => <thread>... <k> S </k> <env> Rho </env> ...</thread>)
```

Note configuration abstraction at work, again. Taking into account the declared configuration, and in particular the multiplicity information `*` in the `<thread/>` cell, the only reasonable way to complete the rule above is to wrap the `<k/>` and `<env/>` cells on the first line within a `<thread/>` cell, and to fill-in the `...`s in the child thread with the default contents of the other subcells in `<thread/>`. In this case there are no other cells, so we can get rid of those `...`s, but that would decrease the modularity of this rule: indeed, we may later on add other cells within `<thread/>` as the language evolves, for example a function or an exception stack, etc.

In theory, we should be able to write the rule above even more compactly and modularly, namely as

```
rule <k> spawn S => . ...</k> <env> Rho </env>
     (. => <k> S </k> <env> Rho </env>)
```

Unfortunately, this currently does not work in the K tool, due to some known limitations of our current configuration abstraction algorithm. This latter rule would be more modular, because it would not even depend on the cell name `thread`. For example, we may later decide to change `thread` into `agent`, and we would not have to touch this rule. We hope this current limitation will be eliminated soon.

Once a thread terminates, its computation cell becomes empty. When that happens, we can go ahead and remove the useless `thread` cell:

```
rule <thread>... <k> . </k> ...</thread> => .  [structural]
```

Let's see what we've got. `kompile` and `krun spawn.imp`.
Note the following:

- The `<threads/>` cell is empty, so all threads terminated normally;
- The value printed is different from the value in the store; the store value is not even the one obtained if the threads executed sequentially.

Therefore, interesting behaviors may happen; we would like to see them all!

Based on prior experience with `krun`'s search option, we would hope that

```
krun spawn.imp --search
```

shows all the behaviors. However, the above does not work, for two reasons.

First, `spawn.imp` is an interactive program, which reads a number from the standard input. When analyzing programs exhaustively using the search option,

`krun` has to disable the streaming capabilities (just think about it and you will realize why). The best you can do in terms of interactivity with search is to pipe some input to `krun`: `krun` will flush the standard input buffer into the cells connected to it when creating the initial configuration (will do that no matter whether you run it with or without the `--search` option). For example:

```
echo 23 | krun spawn.imp --search
```

puts `23` in the standard input buffer, which is then transferred in the `<in/>` cell as a list item, and then the exhaustive search procedure is invoked.

Second, even after piping some input, the `spawn.imp` program still manifests only one behavior, which does not seem right. There should be many more.

As explained in Lesson 3, by default `kompile` optimizes the generated language model for execution. In particular, it does not insert any backtracking markers where transition attempts should be made, so `krun` lacks the information it needs to exhaustively search the generated language model. Like we did in Lesson 3 with the language constructs, we also have to explicitly tell `kompile` which rules should be considered as actual transitions. A theoretically correct but practically unfeasible approach to search all possible behaviors is to consider all rules as transitions. Even more than with the non-deterministic strictness of language constructs in Lesson 3, such a naive solution would make the number of behaviors, and thus `krun`, explode. Remember that a two-thread program with 150 statements each manifests more behaviors than particles in the known universe! Consequently, unless your multi-threaded programs are very small, you will most likely want to control which rules should be considered transitions and which should not.

A good rule of thumb is to include as transitions only those rules which *compete for behaviors*. That is, those rules which may yield a different behavior if we choose to apply them when other rules match as well. The rule for addition, for example, is a clear example of a rule which should not be a transition: indeed, `3+7` will rewrite to `10` now and also later. On the other hand, the lookup rule should be a transition. Indeed, if we delay the lookup of variable `x`, then other threads may write `x` in the meanwhile (with an increment or an assignment rule) and thus yield a different behavior.

Let us discuss and tag those rules which should be transitions: lookup and increment need to be transitions and we already tagged them in Lesson 3; the read rule needs to also be a transition, because it may complete with other instances of itself in other threads; assignment needs to also be a transition, and so should be the first rule for `print`.

Let us now `kompile` with the transition option set as desired:

```
kompile imp --transition "lookup increment assignment read print"
```

Now `echo 23 | krun spawn.imp --search` gives us all 12 behaviors of the `spawn.imp` program.

Like for non-deterministically strict operations which can be tagged as transitions, it is highly non-trivial to say precisely which rules need to be transitions. So `krun` makes no attempt to automatically detect it. Instead, it provides the functionality to let you decide it.

We currently have no mechanism for thread synchronization. In the next lesson we add a `join` statement, which allows a thread to wait until another completes.

Go to Lesson 7, IMP++: Everything Changes: Syntax, Configuration, Semantics.

MOVIE (out of date) [11'40"]

## Everything Changes: Syntax, Configuration, Semantics

In this lesson we add thread joining, one of the simplest thread synchronization mechanisms. In doing so, we need to add unique ids to threads in the configuration, and to modify the syntax to allow `spawn` to return the id of the newly created thread. This gives us an opportunity to make several other small syntactic and semantics changes to the language, which make it more powerful or more compact at a rather low cost.

Before we start, let us first copy and modify the previous `spawn.imp` program from Lesson 1 to make use of thread joining. Recall from Lesson 6 that in some runs of this program the main thread completed before the child threads, printing a possibly undesired value of `x`. What we want now is to assign unique ids to the two spawned threads, and then to modify the main thread to join the two child threads before printing. To avoid adding a new type to the language, let's assume that thread ids are integer numbers. So we declare two integers, `t1` and `t2`, and assign them the two spawn commands. In order for this to parse, we will have to change the syntax of `spawn` to be an arithmetic expression construct instead of a statement. Once we do that, we have a slight syntactic annoyance: we need to put two consecutive `;` after the spawn assignment, one for the assignment statement inside the spawn, and another for the outer assignment. To avoid the two consecutive semicolons, we can syntactically enforce spawn to take a block as argument, instead of a statement. Now it looks better. The new `spawn.imp` program is still non-deterministic, because the two threads can execute in any order and even continue to have a data-race on the shared variable `x`, but we should see fewer behaviors when we use the `join` statements. If we want to fully synchronize this program, we can have the second thread start with a `join(t1)` statement.

Then we should only see one behavior for this program.

Let us now modify the language semantics. First, we move the `spawn` construct from statements to expressions, and make it take a block. Second, we add one more sub-cell to the thread cell in the configuration, `<id/>`, to hold the unique identifier of the thread. We want the main thread to have id `0`, so we initialize this cell with `0`. Third, we modify the spawn rule to generate a fresh integer identifier, which is put in the `<id/>` cell of the child thread and returned as a result of `spawn` in the parent thread. Fourth, let us add the `join` statement to the language, both syntactically and semantically. So in order for the `join(T)` statement to execute, thread `T` must have its computation empty. However, in order for this to work we have to get rid of the thread termination cleanup rule. Indeed, we need to store somewhere the information that thread `T` terminated; the simplest way to do it is to not remove the terminated threads. Feel free to experiment with other possibilities, too, here. For example, you may add another cell, `<done/>`, in which you can store all the thread ids of the terminated and garbage-collected threads.

Let us now `kompile imp.k` and convince ourselves that the new `spawn.imp` with `join` statements indeed has fewer behaviors than its variant without `join` statements. Also, let us convince ourselves that the fully synchronized variant of it indeed has only one behavior.

Note that now spawn, like variable increment, makes the evaluation of expressions to have side effects. Many programming languages in fact allow expressions to be evaluated only for their side effects, and not for their value. This is typically done by simply adding a `;` after the expression and thus turning it into a statement. For example, `++x;`. Let as also allow arithmetic expressions in our language to be used as statements, by simply adding the production `AExp ";"` to `Stmt`, with evaluation strategy `strict` and with the expected semantics discarding the value of the `AExp`.

Another simple change in syntax and semantics which gives our language more power, is to remove the `;` from the syntax of variable assignments and to make them expression instead of statement constructs. This change, combined with the previous one, will still allow us to parse all the programs that we could parse before, but will also allow us to parse more programs. For example, we can now do sequence assignments like in C: `x = y = z = 0`. The semantics of assignment now has to return the assigned value also to the computation, because we want the assignment expression to evaluate to the assigned value.

Let us also make another change, but this time one which only makes the definition more compact. Instead of defining statement sequential composition as a binary construct for statements, let us define a new syntactic construct, `Stmts`, as whitespace-separated lists of `Stmt`. This allows us to get rid of the empty blocks, because we can change the syntax of blocks to `{Stmts}` and `Stmts` also allows the empty sequence of statements.

However, we do have to make sure that `.Stmts` dissolves.

In general, unless you are defining a well-established programming language, it is quite likely that your definitions will suffer lots of changes like the ones seen in this lecture. You add a new construct, which suggests changes in the existing syntax making in fact your language parse more programs, which then requires corresponding changes in the semantics, and so on. Also, compact definitions are desirable in general, because they are easier to read and easier to change if needed later.

In the next lesson we wrap up and document the definition of IMP++.

Go to Lesson 8, IMP++: Wrapping up Larger Languages.

## Wrapping up Larger Languages

In this lesson we wrap up IMP++'s semantics and also generate its poster. While doing so, we also learn how to display larger configurations in order to make them easier to read and print.

Note that we rearrange a bit the semantics, to group the semantics of old IMP's constructs together, and separate it from the new IMP++'s semantics.

There is a detailed discussion at the end of the document about the `--transition` option of `kompile`, because that is important and we want the poster to include everything we learned in this part of the tutorial.

You can go even further and manually edit the generated Latex document. You typically want to do that when you want to publish your language definition, or parts of it, and you need to finely tune it to fit the editing requirements. For example, you may want to insert some negative spaces, etc.

Part 4 of the tutorial is now complete. At this moment you should know most of K framework's features and how to use the K tool. You can now define or design your own programming languages, and then execute and analyze programs.

MOVIE (out of date) [06'26"]

# Part 5: Defining Type Systems

In this part of the tutorial we will show that defining type systems for languages is essentially no different from defining semantics. The major difference is that programs and fragments of programs now rewrite to their types, instead of to concrete values. In terms of K, we will learn how to use it for a certain particular but important kind of applications.

# Imperative, Environment-Based Type Systems

In this lesson you learn how to define a type system for an imperative language (the IMP++ language defined in Part 4 of the tutorial), using a style based on type environments.

Let us copy the `imp.k` file from Part 4 of the tutorial, Lesson 7, which holds the semantics of IMP++, and modify it into a type system. The resulting type system, when executed, yields a type checker.

We start by defining the new strictness attributes of the IMP++ syntax. While doing so, remember that programs and fragments of programs now reduce to their types. So types will be the new results of our new (type) semantics. We also clean up the semantics by removing the unnecessary tags, and also use `strict` instead of `seqstrict` wherever possible, because `strict` gives implementations more freedom. Interestingly, note that `spawn` is strict now, because the code of the child thread should type in the current parent's type environment. Note that this is not always the case for threads, see for example SIMPLE in the languages tutorial, but it works here for our simpler IMP++.

From a typing perspective, the `&&` construct is strict in both its arguments; its short-circuit (concrete) semantics is irrelevant for its (static) type system. Similarly, both the conditional and the while loop are strict constructs when regarded through the typing lenses.

Finally, the sequential composition is now sequentially strict! Indeed, statements are now going to reduce to their type, `stmt`, and it is critical for sequential composition to type its argument statements left-to-right; for example, imagine that the second argument is a variable declaration (whose type semantics will modify the type environment).

We continue by defining the new results of computations, that is, the actual types. In this simple imperative language, we only have a few constant types: `int`, `bool`, `string`, `block` and `stmt`.

We next define the new configuration, which is actually quite simple. Besides the `<k/>` cell, all we need is a type environment cell, `<tenv/>`, which will hold a map from identifiers to their types. A type environment is therefore like a state in the abstract domain of type values.

Let us next modify the semantic rules, turning them into a type system. In short, the idea is to reduce the basic values to their types, and then have a rule for each language construct reducing it to its result type whenever its arguments have the expected types.

We write the rules in the order given by the syntax declarations, to make sure we do not forget any construct.

Integers reduce to their type, `int`.

So do the strings.

Variables are now looked up in the type environment and reduced to their type there. Since we only declare integer variables in IMP++, their type in `tenv` will always be `int`. Nevertheless, we write the rule generically, so that we would not have to change it later if we add other type declarations to IMP++. Note that we reject programs which lookup undeclared variables. Rejection, in this case, means *rewriting getting stuck*.

Variable increment types to `int`, provided the variable has type `int`.

Read types to `int`, because we only allow integer input.

Division is only allowed on integers, so it rewrites to `int` provided that its arguments rewrite to `int`. Note, however, that in order to write `int / int`, we have to explicitly add `int` to the syntax of arithmetic expressions. Otherwise, the K parser rightfully complains, because `/` was declared on arithmetic expressions, not on types. One simple and generic way to allow types to appear anywhere, is to define `Type` as a syntactic subcategory of all the other syntactic categories. Let's do it on a by-need basis, though.

Addition is overloaded, so we add two typing rules for it: one for integers and another for strings.

As discussed, `spawn` types to `stmt` provided that its argument types to `block`.

The assignment construct was `strict(2)`; its typing policy is that the declared type of `X` should be identical to the type of the assigned value. Like for lookup, we define this rule more generically than needed for IMP++, for any type, not only for `int`.

The typing rules for Boolean expression constructs are in the same spirit. Note that we need only one rule for `&&`.

The typing of blocks is a bit trickier. First, note that we still need to recover the environment after the block is typed, because we do not want the block-local variables to be visible in the outer type environment. We recover the type environment only after the block-enclosed statements type; moreover, we also opportunistically yield a `block` type on the computation when we discard the type environment recovery item. To account for the fact that the block-enclosed statement can itself be a block (e.g., `{{S}}`), we would need an additional rule. Since we do not like repetition, we instead group the types `block` and `stmt` into one syntactic category, `BlockOrStmtType`, and now we can have only one rule. We also include `BlockOrStmtType` in `Type`, as a replacement for the two basic types.

The expression statement types as expected. Recall that we only allow arithmetic expressions, which type to `int`, to be used as statements in IMP++.

The conditional was declared `strict` in all its arguments. Its typing policy is that its first argument types to `bool` and its two branches to `block`. If that is the case, then it yields a `stmt` type.

For `while`, its first argument should type to `bool` and its second to `block`.

Variable declarations add new bindings to the type environment. Recall that we can only declare variables of integer type in IMP++.

The typing policy of `print` is that it can only print integer or string values, and in that case it types to `stmt`. Like for `BlockOrStmtType`, to avoid having two similar rules, one for `int` and another for `string`, we prefer to introduce an additional syntactic category, `PrintableType`, which includes both `int` and `string` types.

`halt` types to `stmt`; so its subsequent code is also typed.

`join` types to `stmt`, provided that its argument types to `int`.

Sequential composition was declared as a whitespace-separated sequentially strict list. Its typing policy is that all the statements in the list should type to `stmt` or `block` in order for the list to type to `stmt`. Since lists are maintained internally as cons-lists, this is probably the simplest way to do it:

```
rule .Stmts => stmt
rule _:BlockOrStmtType Ss => Ss
```

Note that the first rule, which types the empty sequence of statements to `stmt`, is needed anyway, to type empty blocks `{}` (together with the block rule).

`kompile imp.k` and `krun` all the programs in Part 4 of the tutorial. They should all type to `stmt`.

In the next lesson we will define a substitution-based type system for LAMBDA.

Go to Lesson 2, Type Systems: Substitution-Based Higher-Order Type Systems.

MOVIE (out of date) [10'11"]

## Substitution-Based Higher-Order Type Systems

In this lesson you learn how to define a substitution-based type system for a higher-order language, namely the LAMBDA language defined in Part 1 of the
tutorial.

Let us copy the definition of LAMBDA from Part 1 of the tutorial, Lesson 8. We are going to modify it into a type systems for LAMBDA.

Before we start, it is important to clarify an important detail, namely that our type system will yield a type checker when executed, not a type inferencer. In particular, we are going to change the LAMBDA syntax to allow us to associate a type to each declared variable. The

constructs which declare variables are `lambda`, `let`, `letrec` and `mu`.
The syntax of all these will therefore change.

Since here we are not interested in a LAMBDA semantics anymore, we take the freedom to eliminate the `Val` syntactic category, our previous results.
Our new results are going to be the types, because programs will now reduce to their types.

As explained, the syntax of the `lambda` construct needs to change, to also declare the type of the variable that it binds. We add the new syntactic category `Type`, with the following constructs: `int`, `bool`, the function type (which gives it its higher-order status), and parentheses as bracket.
Also, we make types our K results.

We are now ready to define the typing rules.

Let us start with the typing rule for lambda abstraction: `lambda X : T . E` types to the function type `T -> T'`, where `T'` is the type obtained by further typing `E[T/X]`. This can be elegantly achieved by reducing the lambda abstraction to `T -> E[T/X]`, provided that we extend the function type construct
to take expressions, not only types, as arguments, and to be strict.
This can be easily achieved by redeclaring it as a strict expression construct (strictness in the second argument would suffice in this example, but it is more uniform to define it strict overall).

The typing rule for application is as simple as it can get: `(T1->T2) T1 => T2`.

Let us now give the typing rules of arithmetic and Boolean expression constructs. First, let us get rid of `Val`. Second, rewrite each value to its type, similarly to the type system for IMP++ in the previous lesson. Third, replace each semantic rule by its typing rule. Fourth, make sure you do not forget to subsort `Type` to `Exp`, so your rules above will parse.

The typing policy of the conditional statement is that its first argument should type to `bool` and its other two arguments should type to the same type `T`, which will also be the result type of the conditional. So we make the conditional construct `strict` in all its three arguments and we write the obvious rule: `if bool then T:Type else T => T`. We want a runtime check that
the latter arguments are actually typed, so we write `T:Type`.

There is nothing special about `let`, except that we have to make sure we change its syntax to account for the type of the variable that it binds.
This rule is a macro, so the `let` is desugared statically.

Similarly, the syntax of `letrec` and `mu` needs to change to account for the type of the variable that they bind. The typing of `letrec` remains based on its desugaring to `mu`; we have to make sure the types are also included now.

The typing policy of `mu` is that its body should type to the same type `T` of its variable, which is also the type of the entire `mu` expression. This can be elegantly achieved by rewriting it to `(T -> T) E[T/X]`. Recall that application is strict, so `E[T/X]` will be eventually reduced to its type. Then the application types correctly only if that type is also `T`, and in that case the result type will also be `T`.

`kompile` and `krun` some programs. You can, for example, take the LAMBDA programs from the first tutorial, modify them by adding types to their variable declarations, and then type check them using `krun`.

In the next lesson we will discuss an environment-based type system for LAMBDA.

Go to Lesson 3, Type Systems: Environment-Based Higher-Order Type Systems.

MOVIE (out of date) [6'52"]

## Environment-Based Higher-Order Type Systems

In this lesson you learn how to define an environment-based type system for a higher-order language, namely the LAMBDA language defined in Part 1 of the
tutorial.

The simplest and fastest way to proceed is to copy the substitution-based type system of LAMBDA from the previous lesson and modify it into an environment-based one. A large portion of the substitution-based definition will remain unchanged. We only have to modify the rules that use substitution.

We do not need the substitution anymore, so we can remove the require and import statements. The syntax of types and expressions stays unchanged, but we can now remove the `binder` tag of lambda.

Like in the type system of IMP++ in Lesson 1, we need a configuration that contains, besides the `<k/>` cell, a `<tenv/>` cell that will hold the type environment.

In an environment-based definition, unlike in a substitution-based one, we need to lookup variables in the environment. So let us start with the type lookup rule:

```
rule <k> X:Id => T ...</k> <tenv>... X |-> T ...</k>
```

The type environment is populated by the semantic rule of `lambda`:

```
rule <k> lambda X : T . E => (T -> E) ~> Rho ...</k>
     <tenv> Rho => Rho[X <- T] </tenv>
```

So `X` is bound to its type `T` in the type environment, and then `T -> E` is scheduled for processing. Recall that the arrow type construct has been extended into a strict expression construct, so `E` will be eventually reduced to its type. Like in other environment-based definitions, we need to make sure that we recover the type environment after the computation in the scope of the declared variable terminates.

The typing rule of application does not change, so it stays as elegant as it was in the substitution-based definition:

```
rule (T1 -> T2) T1 => T2
```

So do the rules for arithmetic and Boolean constructs, and those for the `if`, and `let`, and `letrec`.

The `mu` rule needs to change, because it was previously defined using substitution. We modify it in the same spirit as we modified the `lambda` rule: bind `X` to its type in the environment, schedule its body for typing in its right context, and then recover the type environment.

Finally, we give the semantics of environment recovery, making sure the environment is recovered only after the preceding computation is reduced to a type:

rule _:Type ~> (Rho => .) ... _ => Rho

The changes that we applied to the substitution-based definition were therefore quite systematic: each substitution invocation was replaced with an appropriate type environment update/recovery.

Go to Lesson 4, Type Systems: A Naive Substitution-Based Type Inferencer.

# A Naive Substitution-Based Type Inferencer

In this lesson you learn how to define a naive substitution-based type inferencer for a higher-order language, namely the LAMBDA language defined in Part 1 of the tutorial.

Unlike in the type checker defined in Lessons 2 and 3, where we had to associate a type with each declared variable, a type inferencer attempts to infer the types of all the variables from the way those variables are used. Let us take a look at this program, say `plus.lambda`:

```
lambda x . lambda y . x + y
```

Since `x` and `y` are used in an integer addition context, we can infer that they must have the type `int` and the result of the addition is also an `int`, so the type of the entire expression is `int -> int -> int`. Similarly, the program `if.lambda`

```
lambda x . lambda y . lambda z .
  if x then y else z
```

can only make sense when `x` has type `bool` and `y` and `z` have the same type, say `t`, in which case the type of the entire expression is `bool -> t -> t -> t`. Since the type `t` can be anything, we say that the type of this expression is *polymorphic*. That means that the code above can be used in different contexts, where `t` can be an `int`, a `bool`, a function type `int -> int`, and so on.

In the `identity.lambda` program

```
let f = lambda x . x
in f 1
```

`f` has such a polymorphic type, which is then applied to an integer, so this program is type-safe and its type is `int`.

A typical polymorphic expression is the composition

```
lambda f . lambda g . lambda x .
  g (f x)
```

which has the type `(t1 -> t2) -> (t2 -> t3) -> (t1 -> t3)`, polymorphic in 3 types.

Let us now define our naive type inferencer and then we discuss more examples. The idea is quite simple: we conceptually do the same operations like we did within the type checker defined in Lesson 2, with two important differences:

1. instead of declaring a type with each declared variable, we assume a fresh type for that variable; and
2. instead of checking that the types of expressions satisfy the type properties of the context in which they are used, we impose those properties as type equality constraints. A general-purpose unification-based constraint solving mechanism is then used to solve the generated type constraints.

Let us start with the syntax, which is essentially identical to that of the type checker in Lesson 2, except that bound variables are not declared a type anymore. Also, to keep things more compact, we put all the `Exp` syntax declarations in one syntax declaration this time.

Before we modify the rules, let us first define our machinery for adding and solving constraints. First, we require and import the unification procedure. We do not discuss unification here, but if you are interested you can consult the `unification.k` files under k-distribution/include/kframework/builtin, which contains our current generic definition of unification, which is written also in K. The generic unification provides a sort, `Mgu`, for *most-general-unifier*, an operation

`updateMgu(Mgu,T1,T2)` which updates `Mgu` with additional constraints generated by forcing the terms `T1` and `T2` to be equal, and an operation `applyMgu(Mgu,T)` which applies `Mgu` to term `T`. For our use of unification here, we do not even need to know how `Mgu` terms are represented internally.

We define a K item construct, `=`, which takes two `Type` terms and enforces them to be equal by means of updating the current `Mgu`. Once the constraints are added to the `Mgu`, the equality dissolves itself. With this semantics of `=` in mind, we can now go ahead and modify the rules of the type checker systematically into rules for a type inferencer. The changes are self-explanatory and mechanical: for example, the rule

```
rule int * int => int
```

changes into rule

```
rule T1:Type  * T2:Type => T1 = int ~> T2 = int ~> int
```

generating the constraints that the two arguments of multiplication have the type `int`, and the result type is `int`. Recall that each type equality on the `<k/>` cell updates the current `Mgu` appropriately and then dissolves itself; thus, the above says that after imposing the constraints `T1=int` and `T2=int`, multiplication yields a type `int`.

As mentioned above, since types of variables are not declared anymore, but inferred, we have to generate a fresh type for each variable at its declaration time, and then generate appropriately constraints for it. For example, the type semantics of `lambda` and `mu` become:

```
rule lambda X . E => T -> E[T/X]  when fresh(T:Type)
rule mu X . E => (T -> T) E[T/X]  when fresh(T:Type)
```

that is, we add a condition stating that the previously declared type is now a fresh one. This type will be further constrained by how the variable `X` is being used within `E`.

Interestingly, the previous typing rule for lambda application is not powerful enough anymore. Indeed, since types are not given anymore, it may very well be the case that the inferred type of the first argument of the application construct is not yet a function type (remember, for example, the program composition.lambda above). What we have to do is to enforce it to be a function type, by means of fresh types and constraints. We can introduce a fresh type for the result of the application, and then write the expected rule as follows:

```
rule T1:Type T2:Type => T1 = (T2 -> T) ~> T  when fresh(T:Type)
```

The conditional requires that its first argument is a `bool` and its second and third arguments have the same type, which is also the result type.

The macros do not change, in particular `let` is desugared into lambda application. We will next see that this is a significant restriction, because it limits the polymorphism of our type system.

We are done. We have a working type inferencer for LAMBDA.

Let's `kompile` it and `krun` the programs above. They all work as expected. Let us also try some additional programs, to push it to its limits.

First, let us test `mu` by means of a `letrec` example:

```
letrec f x = 3
in f
```

We can also try all the programs that we had in our first tutorial, on lambda, for example the `factorial.imp` program:

```
letrec f x = if x <= 1 then 1 else (x * (f (x + -1)))
in (f 10)
```

Those programs are simple enough that they should all work as expected with our naive type inferencer here.

Let us next try to type some tricky programs, which involve more complex and indirect type constraints.

`tricky-1.lambda`:

```
lambda f . lambda x . lambda y . (
  (f x y) + x + (let x = y in x)
)
```

`tricky-2.lambda`:

```
lambda x .
  let f = lambda y . if true then y else x
  in (lambda x . f 0)
```

`tricky-3.lambda`:

```
lambda x . let f = lambda y . if true then x 7 else x y
           in f
```

`tricky-4.lambda`:

```
lambda x . let f = lambda x . x
           in let d = (f x) + 1
              in x
```

`tricky-5.lambda`:

```
lambda x . let f = lambda y . x y
           in let z = x 0 in f
```

It is now time to see the limitations of this naive type inferencer. Consider the program

```
let id = lambda x . x
in if (id true) then (id 1) else (id 2)
```

Our type inferencer fails graciously with a clash in the `<mgu/>` cell between `int` and `bool`. Indeed, the desugaring macro of `let` turns it into a `lambda` and an application, which further enforce `id` to have a type of the form `t -> t` for some fresh type `t`. The first use of `id` in the condition of `if` will then constrain `t` to be `bool`, while the other uses in the two branches will enforce `t` to be `int`. Thus the clash in the `<mgu/>` cell.

Similarly, the program

```
let id = lambda x . x
in id id
```

yields a different kind of conflict: if `id` has type `t -> t`, in order to apply `id` to itself it must be the case that its argument, `t`, equals `t -> t`. These two type terms cannot be unified because there is a circular dependence on `t`, so we get a cycle in the `<mgu/>` cell.

Both limitations above will be solved when we change the semantics of `let` later on, to account for the desired polymorphism.

Before we conclude this lesson, let us see one more interesting example, where the lack of let-polymorphism leads not to a type error, but to a less generic type:

```
let f1 = lambda x . x in
  let f2 = f1 in
    let f3 = f2 in
      let f4 = f3 in
        let f5 = f4 in
          if (f5 true) then f2 else f3
```

Our current type inferencer will infer the type `bool -> bool` for the program above. Nevertheless, since all functions `f1`, `f2`, `f3`, `f4`, `f5` are the identity function, which is polymorphic, we would expect the entire program to type to the same polymorphic identity function type.

This limitation will be also addressed when we define our let-polymorphic type inferencer.

Before that, in the next lesson we will show how easily we can turn the naive substitution-based type inferencer discussed in this lesson into a similarly naive, but environment-based type inferencer.

Go to Lesson 5, Type Systems: A Naive Environment-Based Type Inferencer.

# A Naive Environment-Based Type Inferencer

In this lesson you learn how to define a naive environment-based type inferencer for a higher-order language. Specifically, we take the substitution-based type inferencer for LAMBDA defined in Lesson 4 and turn it into an environment-based one.

Recall from Lesson 3, where we defined an environment-based type checker for LAMBDA based on the substitution-based one in Lesson 2, that the transition from a substitution-based definition to an environment-based one was quite systematic and mechanical: each substitution occurrence `E[T/X]` is replaced by `E`, but at the same time the variable `X` is bound to type `T` in the type environment. One benefit of using type environments instead of substitution is that we replace a linear complexity operation (the substitution) with a constant complexity one (the variable lookup).

There is not much left to say which has not been already said in Lesson 3: we remove the unnecessary binder annotations for the variable binding operations, then add a `<tenv/>` cell to the configuration to hold the type environment, then add a new rule for variable lookup, and finally apply the transformation of substitutions `E[T/X]` into `E` as explained above.

The resulting type inferencer should now work exactly the same way as the substitution-based one, except, of course, that the resulting configurations will contain a `<tenv/>` cell now.

As sanity check, let us consider two more LAMBDA programs that test the static scoping nature of the inferencer. We do that because faulty environment-based definitions often have this problem. The program

```
let x = 1
in let f = lambda a . x
   in let x = true
      in f 3
```

should type to `int`, not to `bool`, and so it does. Similarly, the program

```
let y = 0
in letrec f x = if x <= 0
                then y
                else let y = true
                     in f (x + 1)
```

```
in f 1
```

should also type to `int`, not `bool`, and so it does, too.

The type inferencer defined in this lesson has the same limitations, in terms of polymorphism, as the one in Lesson 4. In the next lesson we will see how it can be parallelized, and in further lessons how to make it polymorphic.

Go to Lesson 6, Type Systems: Parallel Type Checkers/Inferencers.

## Parallel Type Checkers/Inferencers

In this lesson you learn how to define parallel type checkers or inferencers. For the sake of a choice, we will parallelize the one in the previous lesson, but the ideas are general. We are using the same idea to define type checkers for other languages in the K tool distribution, such as SIMPLE and KOOL.

The idea is in fact quite simple. Instead of one monolithic typing task, we generate many smaller tasks, which can be processed in parallel. We use the same approach to define parallel semantics as we used for threads in IMP++ in Part 4 of the tutorial, that is, we add a cell holding all the parallel tasks, making sure we declare the cell holding a task with multiplicity `*`. For the particular type inferencer that we chose here, the one in Lesson 5, each task will hold an expression to type together with a type environment (so it knows where to lookup its free variables). We have the following configuration then:

```
configuration <tasks color="yellow">
                <task color="orange" multiplicity="*">
                  <k color="green"> $PGM:Exp </k>
                  <tenv color="red"> .Map </tenv>
                </task>
              </tasks>
              <mgu color="blue"> .Mgu </mgu>
```

Now we have to take each typing rule we had before and change it to yield parallel typing. For example, our rule for typing multiplication was the following in Lesson 5:

```
rule T1:Type * T2:Type => T1 = int ~> T2 = int ~> int
```

Since `*` was strict, its two arguments eventually type, and once that happens the rule above fires. Unfortunately, the strictness of multiplication makes the typing of the two expressions sequential in our previous definition. To avoid typing the two expressions sequentially and instead generating two parallel tasks, we remove the

`strict` attribute of multiplication and replace the rule above with the following:

```
rule <k> E1 * E2 => int ...</k> <tenv> Rho </tenv>
     (. => <task> <k> E1 = int </k> <tenv> Rho </tenv> </task>
           <task> <k> E2 = int </k> <tenv> Rho </tenv> </task>)
```

Therefore, we generate two tasks for typing `E1` and `E2` in the same type environment as the current task, and let the current task continue by simply optimistically reducing `E1*E2` to its expected result type, `int`. If `E1` or `E2` will not type to `int`, then either their corresponding tasks will get stuck or the `<mgu/>` cell will result into a clash or cycle, so the program will not type overall in spite of the fact that we allowed the task containing the multiplication to continue. This is how we get maximum of parallelism in this case.

Before we continue, note that the new tasks hold equalities in them, where one of its arguments is an expression, while previously the equality construct was declared to take types. What we want now is for the equality construct to possibly take any expressions, and first type them and then generate the type constraint like before. This can be done very easily by just extending the equality construct to expressions and declaring it `strict`:

```
syntax KItem ::= Exp "=" Exp  [strict]
```

Unlike before, where we only passed types to the equality construct, we now need a runtime check that its arguments are indeed types before we can generate the `updateMgu` command:

```
rule <k> T:Type = T':Type => . ...</k>
     <mgu> Theta:Mgu => updateMgu(Theta,T,T') </mgu>
```

Like before, an equality will therefore update the `<mgu/>` cell and then it dissolves itself, letting the `<k/>` cell in the corresponding task empty. Such empty tasks are unnecessary, so they can be erased:

```
rule <task>... <k> . </k> ...</task> => .
```

We can now follow the same style as for multiplication to write the parallel typing rules of the other arithmetic constructs, and even for the conditional.

To parallelize the typing of lambda we generate two fresh types, one for the variable and one for the body, and make sure that we generate the correct type constraint and environment in the body task:

```
rule <k> lambda X . E => Tx -> Te ...</k> <tenv> TEnv </tenv>
     (. => <task> <k> E = Te </k> <tenv> TEnv[Tx/X] </tenv> </task>)
  when fresh(Tx:Type) andBool fresh(Te:Type)
```

Note that the above also allows us to not need to change and then recover the environment of the current cell.

For function application we also need to generate two fresh types:

```
rule <k> E1 E2 => T ...</k> <tenv> Rho </tenv>
     (. => <task> <k> E1 = T2 -> T </k> <tenv> Rho </tenv> </task>
           <task> <k> E2 = T2 </k> <tenv> Rho </tenv> </task>)
  when fresh(T2:Type) andBool fresh(T:Type)
```

The only rule left is that of `mu X . E`. In this case we only need one fresh type, because `X`, `E` and `mu X . E` have all the same type:

```
rule <k> mu X . E => T ...</k>  <tenv> TEnv </tenv>
     (. => <task> <k> E = T </k> <tenv> TEnv[T/X] </tenv> </task>)
  when fresh(T:Type)
```

We do not need the type environment recovery operation, so we delete it.

We can now `kompile` and `krun` all the programs that we typed in Lesson 5. Everything should work.

In this lesson we only aimed at parallelizing the type inferencer in Lesson 5, not to improve its expressiveness; it still has the same limitations in terms of polymorphism. The next lessons are dedicated to polymorphic type inferencers.

Go to Lesson 7, Type Systems: A Naive Substitution-based Polymorphic Type Inferencer.

# A Naive Substitution-based Polymorphic Type Inferencer

In this lesson you learn how little it takes to turn a naive monomorphic type inferencer into a naive polymorphic one, basically only changing a few characters. In terms of the K framework, you will learn that you can have complex combinations of substitutions in K, both over expressions and over types.

Let us start directly with the change. All we have to do is to take the LAMBDA type inferencer in Lesson 4 and only change the macro

```
rule let X = E in E' => (lambda X . E') E  [macro]
```

as follows:

```
rule let X = E in E' => E'[E/X]  [macro]
```

In other words, we are inlining the beta-reduction rule of lambda-calculus within the original rule. In terms of typing, the above forces the type inferencer to type `E` in place for each

occurrence of `X` in `E'`. Unlike in the first rule, where `X` had to get one type only which satisfied the constrains of all `X`'s occurrences in `E'`, we now never associate any type to `X` anymore.

Let us `kompile` and `krun` some examples. Everything that worked with the type inferencer in Lesson 4 should still work here, although the types of some programs can now be more general. For example, reconsider the `nested-lets.lambda` program

```
let f1 = lambda x . x in
  let f2 = f1 in
    let f3 = f2 in
      let f4 = f3 in
        let f5 = f4 in
          if (f5 true) then f2 else f3
```

which was previously typed to `bool -> bool`. With the new rule above, the sequence of lets is iteratively eliminated and we end up with the program

```
if (lambda x . x) true then (lambda x . x) else (lambda x . x)
```

which now types (with both type inferencers) to a type of the form `t -> t`, for some type variable `t`, which is more general than the previous `bool -> bool` type that the program typed to in Lesson 4.

We can also now type programs that were not typable before, such as

```
let id = lambda x . x
in if (id true) then (id 1) else (id 2)
```

and

```
let id = lambda x . x
in id id
```

Let us also test it on some trickier programs, also not typable before, such as

```
let f = lambda x . x
in let g = lambda y . f y
   in g g
```

which gives us a type of the form `t -> t` for some type variable `t`, and as

```
let f = let g = lambda x . x
        in let h = lambda x . lambda x . (g g g g)
           in h
in f
```

which types to `t1 -> t2 -> t3 -> t3` for some type variables `t1`, `t2`, `t3`.

Here is another program which was not typable before, which is trickier than the others above in that a lambda-bound variable appears free in a let-bound expression:

```
lambda x . (
  let y = lambda z . x
  in if (y true) then (y 1) else (y (lambda x . x))
)
```

The above presents no problem now, because once `lambda z . x` gets substituted for `y` we get a well-typed expression which yields that `x` has the type `bool`, so the entire expression types to `bool -> bool`.

The cheap type inferencer that we obtained above therefore works as expected. However, it has two problems which justify a more advanced solution. First, substitution is typically considered an elegant mathematical instrument which is not too practical in implementations, so an implementation of this type inferencer will likely be based on type environments anyway. Additionally, we mix two kinds of substitutions in this definition, one where we substitute types and another where we substitute expressions, which can only make things harder to implement efficiently. Second, our naive substitution of `E` for `X` in `E'` can yield an exponential explosion in size of the original program. Consider, for example, the following classic example which is known to generate a type whose size is exponential in the size of the program (and is thus used as an argument for why let-polymorphic type inference is exponential in the worst-case):

```
let f00 = lambda x . lambda y . x in
  let f01 = lambda x . f00 (f00 x) in
    let f02 = lambda x . f01 (f01 x) in
      let f03 = lambda x . f02 (f02 x) in
        let f04 = lambda x . f03 (f03 x) in
          // ... you can add more nested lets here
          f04
```

The particular instance of the pattern above generates a type which has 17 type variables! The desugaring of each `let` doubles the size of the program and of its resulting type. While such programs are little likely to appear in practice, it is often the case that functions can be quite complex and large while their type can be quite simple in the end, so we should simply avoid retyping each function each time it is used.

This is precisely what we will do next. Before we present the classic let-polymorphic type inferencer in Lesson 9, which is based on environments, we first quickly discuss in Lesson 8 an intermediate step, namely a naive environment-based variant of the inferencer defined here.

Go to Lesson 8, Type Systems: A Naive Environment-based Polymorphic Type Inferencer.

# A Naive Environment-based Polymorphic Type Inferencer

In this short lesson we discuss how to quickly turn a naive environment-based monomorphic type inferencer into a naive let-polymorphic one. Like in the previous lesson, we only need to change a few characters. In terms of the K framework, you will learn how to have both environments and substitution in the same definition.

Like in the previous lesson, all we have to do is to take the LAMBDA type inferencer in Lesson 5 and only change the macro

```
rule let X = E in E' => (lambda X . E') E  [macro]
```

as follows:

```
rule let X = E in E' => E'[E/X]  [macro]
```

The reasons why this works have already been explained in the previous lesson, so we do not repeat them here.

Since our new let macro uses substitution, we have to require the substitution module at the top and also import SUBSTITUTION in the current module, besides the already existing UNIFICATION.

Everything which worked with the type inferencer in Lesson 7 should also work now. Let us only try the exponential type example,

```
let f00 = lambda x . lambda y . x in
  let f01 = lambda x . f00 (f00 x) in
    let f02 = lambda x . f01 (f01 x) in
      let f03 = lambda x . f02 (f02 x) in
        let f04 = lambda x . f03 (f03 x) in
          f04
```

As expected, this gives us precisely the same type as in Lesson 7.

So the only difference between this type inferencer and the one in Lesson 7 is that substitution is only used for LAMBDA-to-LAMBDA transformations, but not for infusing types within LAMBDA programs. Thus, the syntax of LAMBDA programs is preserved intact, which some may prefer. Nevertheless, this type inferencer is still expensive and wasteful, because the let-bound expression is typed over and over again in each place where the let-bound variable occurs.

In the next lesson we will discuss a type inferencer based on the classic Damas-Hindley-Milner type system, which maximizes the reuse of

typing work by means of parametric types.

Go to Lesson 9, Type Systems: Let-Polymorphic Type Inferencer (Damas-Hindley-Milner).

## Let-Polymorphic Type Inferencer (Damas-Hindley-Milner)

In this lesson we discuss a type inferencer based on what we call today *the Damas-Hindley-Milner type system*, which is at the core of many modern functional programming languages. The first variant of it was proposed by Hindley in 1969, then, interestingly, Milner rediscovered it in 1978 in the context of the ML language. Damas formalized it as a type system in his PhD thesis in 1985. More specifically, our type inferencer here, like many others as well as many implementations of it, follows more closely the syntax-driven variant proposed by Clement in 1987.

In terms of K, we will see how easily we can turn one definition which is considered naive (our previous type inferencer in Lesson 8) into a definition which is considered advanced. All we have to do is to change one existing rule (the rule of the let binder) and to add a new one. We will also learn some new predefined features of K, which make the above possible.

The main idea is to replace the rule

```
rule let X = E in E' => E'[E/X]  [macro]
```

which creates potentially many copies of `E` within `E'` with a rule which types `E` once and then reuses that type in each place where `X` occurs free in `E'`. The simplest K way to type `E` is to declare the let construct `strict(2)`. Now we cannot simply bind `X` to the type of `E`, because we would obtain a variant of the naive type inferencer we already discussed, together with its limitations, in Lesson 5 of this tutorial. The trick here is to parameterize the type of `E` in all its unconstrained fresh types, and then create fresh copies of those parameters in each free occurrence of `X` in `E'`.

Let us discuss some examples, before we go into the technical details. Consider the first let-polymorphic example which failed to be typed with our first naive type-inferencer:

```
let id = lambda x . x
in if (id true) then (id 1) else (id 2)
```

When typing `lambda x . x`, we get a type of the form `t -> t`, for some fresh type `t`. Instead of assigning this type to `id` as we did in the

naive type inferencers, we now first parametrize this type in its fresh variable `t`, written

```
(forall t) t -> t
```

and then bind `id` to this parametric type. The intuition for the parameter is that it can be instantiated with any other type, so this parametric type stands, in fact, for infinitely many non-parametric types. This is similar to what happens in formal logic proof systems, where *rule schemas* stand for infinitely many concrete instances of them. For this reason, parametric types are also called *type schemas*.

Now each time `id` is looked up within the let-body, we create a fresh copy of the parameter `t`, which can this way be independently constrained by each local context. Let's suppose that the three `id` lookups yield the types `t1 -> t1`, `t2 -> t2`, and respectively `t3 -> t3`. Then `t1` will be constrained to be `bool`, and `t2` and `t3` to be `int`, so we can now safely type the program above to `int`.

Therefore, a type schema comprises a summary of all the typing work that has been done for typing the corresponding expression, and an instantiation of its parameters with fresh copies represents an elegant way to reuse all that typing work.

There are some subtleties regarding what fresh types can be made parameters. Let us consider another example, discussed as part of Lesson 7 on naive let-polymorphism:

```
lambda x . (
  let y = lambda z . x
  in if (y true) then (y 1) else (y (lambda x . x))
)
```

This program should type to `bool -> bool`, as explained in Lesson 7. The `lambda` construct will bind `x` to some fresh type `tx`. Then the let-bound expression `lambda z . x` types to `tz -> tx` for some additional fresh type `tz`. The question now is what should the parameters of this type be when we generate the type schema? If we naively parameterize in all fresh variables, that is in both `tz` and `tx` obtaining the type schema `(forall tz,tx) tz -> tx`, then there will be no way to infer that the type of `x`, `tx`, must be a `bool`! The inferred type of this expression would then wrongly be `tx -> t` for some fresh types `tx` and `t`. That's because the parameters are replaced with fresh copies in each occurrence of `y`, and thus their relationship to the original `x` is completely lost. This tells us that we cannot parameterize in all fresh types that appear in the type of the let-bound expression. In particular, we cannot parameterize in those which some variables are already bound to in the current type environment (like `x` is bound to `tx` in our example above).

In our example, the correct type schema is `(forall tz) tz -> tx`, which now allows us to correctly infer that `tx` is `bool`.

Let us now discuss another example, which should fail to type:

```
lambda x .
  let f = lambda y . x y
  in if (f true) then (f 1) else (f 2)
```

This should fail to type because `lambda y . x y` is equivalent to `x`, so the conditional imposes the conflicting constraints that `x` should be a function whose argument is either a `bool` or an `int`. Let us try to type it using our currently informal procedure. Like in the previous example, `x` will be bound to a fresh type `tx`. Then the let-bound expression types to `ty -> tz` with `ty` and `tz` fresh types, adding also the constraint `tx = ty -> tz`. What should the parameters of this type be? If we ignore the type constraint and simply make both `ty` and `tz` parameters because no variable is bound to them in the type environment (indeed, the only variable `x` in the type environment is bound to `tx`), then we can wrongly type this program to `tx -> tz` following a reasoning similar to the one in the example above.
In fact, in this example, none of `ty` and `tz` can be parameters, because they are constrained by `tx`.

The examples above tell us two things: first, that we have to take the type constraints into account when deciding the parameters of the schema; second, that after applying the most-general-unifier solution given by the type constraints everywhere, the remaining fresh types appearing anywhere in the type environment are consequently constrained and cannot be turned into parameters. Since the type environment can in fact also hold type schemas, which already bind some types, we only need to ensure that none of the fresh types appearing free anywhere in the type environment are turned into parameters of type schemas.

Thanks to generic support offered by the K tool, we can easily achieve all the above as follows.

First, add syntax for type schemas:

```
syntax TypeSchema ::= "(" "forall" Set ")" Type  [binder]
```

The definition below will be given in such a way that the `Set` argument of a type schema will always be a set of fresh types. We also declare this construct to be a `binder`, so that we can make use of the generic free variable function provided by the K tool.

We now replace the old macro of `let`

```
rule let X = E in E' => E'[E/X]  [macro]
```

with the following rule:

```
rule <k> let X = T:Type in E => E ~> tenv(TEnv) ...</k>
     <mgu> Theta:Mgu </mgu>
     <tenv> TEnv
      => TEnv[(forall freeVariables(applyMgu(Theta, T)) -Set
                      freeVariables(applyMgu(Theta, values TEnv))
              ) applyMgu(Theta, T) / X]
     </tenv>
```

So the type `T` of `E` is being parameterized and then bound to `X` in the type environment. The current mgu `Theta`, which comprises all the type constraints accumulated so far, is applied to both `T` and the types in the type environment. The remaining fresh types in `T` which do not appear free in the type environment are then turned into type parameters. The function `freeVariables` returns, as expected, the free variables of its argument as a `Set`; this is why we declared the type schema to be a binder above.

Now a LAMBDA variable in the type environment can be bound to either a type or a type schema. In the first case, the previous rule we had for variable lookup can be reused, but we have to make sure we check that `T` there is of sort `Type` (adding a sort membership, for example). In the second case, as explained above, we have to create fresh copies of the parameters. This can be easily achieved with another predefined K function, as follows:

```
rule <k> X:Id => freshVariables(Tvs,T) ...</k>
     <tenv>... X |-> (forall Tvs) T ...</tenv>
```

Indeed, `freshVariables` takes a set of variables and a term, and returns the same term but with each of the given variables replaced by a fresh copy.

The operations `freeVariables` and `freshVariables` are useful in many K definitions, so they are predefined in module `substitution.k`.

Our definition of this let-polymorphic type inferencer is now complete. To test it, `kompile` it and then `krun` all the LAMBDA programs discussed since Lesson 4. They should all work as expected.

# K Languages

Here we present several "real-world" language examples. These languages demonstrate many of the features you would expect to find in a full-fledged programming language.

- SIMPLE: Imperative programming language with threads.
- KOOL: SIMPLE extended with object-oriented features.
- FUN: A functional language with algebraic data-types and pattern-matching.

- LOGIK: A logical programming language based on clause unification.

# SIMPLE — Untyped

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped SIMPLE language.
SIMPLE is intended to be a pedagogical and research language that captures the essence of the imperative programming paradigm, extended with several features often encountered in imperative programming languages.
A program consists of a set of global variable declarations and function definitions. Like in C, function definitions cannot be nested and each program must have one function called `main`, which is invoked when the program is executed. To make it more interesting and to highlight some of **K**'s strengths, SIMPLE includes the following features in addition to the conventional imperative expression and statement constructs:

- Multidimensional arrays and array references. An array evaluates to an array reference, which is a special value holding a location (where the elements of the array start) together with the size of the array; the elements of the array can be array references themselves (particularly when the array is multi-dimensional). Array references are ordinary values,
  so they can be assigned to variables and passed/received by functions.
- Functions and function values. Functions can have zero or more parameters and can return abruptly using a `return` statement. SIMPLE follows a call-by-value parameter passing style, with static scoping.
  Function names evaluate to function abstractions, which hereby become ordinary
  values in the language, same like the array references.
- Blocks with locals. SIMPLE variables can be declared anywhere, their scope being from the place where they are declared until the end of the most nested enclosing block.
- Input/Output. The expression `read()` evaluates to the next value in the input buffer, and the statement `write(e)` evaluates `e` and outputs its value to the output buffer. The input and output buffers are lists of values.

- Exceptions. SIMPLE has parametric exceptions (the value thrown as an exception can be caught and bound).
- Concurrency via dynamic thread creation/termination and synchronization. One can spawn a thread to execute any statement. The spawned thread shares with its parent its environment at creation time.
  Threads can be synchronized via a join command which blocks the current thread
  until the joined thread completes, via re-entrant locks which can be acquired
  and released, as well as through rendezvous commands.

Like in many other languages, some of SIMPLE's constructs can be desugared into a smaller set of basic constructs. We do that at the end of the syntax module, and then we only give semantics to the core constructs.

**Note**: This definition is commented slightly more than others, because it is intended to be one of the first non-trivial definitions that the new user of **K** sees. We recommend the beginner user to first check the language definitions discussed in the **K** tutorial.

```
module SIMPLE-UNTYPED-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

We start by defining the SIMPLE syntax. The language constructs discussed above have the expected syntax and evaluation strategies. Recall that in **K** we annotate the syntax with appropriate strictness attributes, thus giving each language construct the desired evaluation strategy.

## Identifiers

Recall from the **K** tutorial that identifiers are builtin and come under the syntactic category `Id`. The special identifier for the function `main` belongs to all programs, and plays a special role in the semantics, so we declare it explicitly. This would not be necessary if the identifiers were all included automatically in semantic definitions, but that is not possible because of parsing reasons (e.g., **K** variables used to match concrete identifiers would then be ambiguously parsed as identifiers). They are only included in the parser generated to parse programs (and used by the `kast` tool). Consequently, we have to explicitly declare all the concrete identifiers that play a special role in the semantics, like `main` below.

```
  syntax Id ::= "main" [token]
```

## Declarations

There are two types of declarations: for variables (including arrays) and for functions. We are going to allow declarations of the form `var x=10, a[10,10], y=23;`, which is why we allow the `var` keyword to take a list of expressions. The non-terminals used in the two productions below are defined shortly.

```
syntax Stmt ::= "var" Exps ";"
              | "function" Id "(" Ids ")" Block
```

## Expressions

The expression constructs below are standard. Increment (`++`) takes an expression rather than a variable because it can also increment an array element. Recall that the syntax we define in **K** is what we call *the syntax of the semantics*: while powerful enough to define non-trivial syntaxes (thanks to the underlying SDF technology that we use), we typically refrain from defining precise syntaxes, that is, ones which accept precisely the well-formed programs (that would not be possible anyway in general). That job is deferred to type systems, which can also be defined in **K**. In other words, we are not making any effort to guarantee syntactically that only variables or array elements are passed to the increment construct, we allow any expression. Nevertheless, we will only give semantics to those, so expressions of the form `++5`, which parse (but which will be rejected by our type system in the typed version of SIMPLE later), will get stuck when executed. Arrays can be multidimensional and can hold other arrays, so their lookup operation takes a list of expressions as argument and applies to an expression (which can in particular be another array lookup), respectively. The construct `sizeOf` gives the size of an array in number of elements of its first dimension. Note that almost all constructs are strict. The only constructs which are not strict are the increment (since its first argument gets updated, so it cannot be evaluated), the input read which takes no arguments so strictness is irrelevant for it, the logical and and or constructs which are short-circuited, the thread spawning construct which creates a new thread executing the argument expression and return its unique identifier to the creating thread (so it cannot just evaluate its argument in place), and the assignment which is only strict in its second argument (for the same reason as the increment).

```
syntax Exp ::= Int | Bool | String | Id
             | "(" Exp ")"              [bracket]
             | "++" Exp
             > Exp "[" Exps "]"         [strict]
             > Exp "(" Exps ")"         [strict]
             | "-" Exp                  [strict]
             | "sizeOf" "(" Exp ")"     [strict]
```

```
| "read" "(" ")"
> left:
  Exp "*" Exp             [strict, left]
| Exp "/" Exp             [strict, left]
| Exp "%" Exp             [strict, left]
> left:
  Exp "+" Exp             [strict, left]
| Exp "-" Exp             [strict, left]
> non-assoc:
  Exp "<" Exp             [strict, non-assoc]
| Exp "<=" Exp            [strict, non-assoc]
| Exp ">" Exp             [strict, non-assoc]
| Exp ">=" Exp            [strict, non-assoc]
| Exp "==" Exp            [strict, non-assoc]
| Exp "!=" Exp            [strict, non-assoc]
> "!" Exp                 [strict]
> left:
  Exp "&&" Exp            [strict(1), left]
| Exp "||" Exp            [strict(1), left]
> "spawn" Block
> Exp "=" Exp             [strict(2), right]
```

We also need comma-separated lists of identifiers and of expressions. Moreover, we want them to be strict, that is, to evaluate to lists of results whenever requested (e.g., when they appear as strict arguments of the constructs above).

```
syntax Ids  ::= List{Id,","}
syntax Exps ::= List{Exp,","}          [strict]  // automatically hybrid now
syntax Exps ::= Ids
syntax Val
syntax Vals ::= List{Val,","}
syntax Bottom
syntax Bottoms ::= List{Bottom,","}
syntax Ids ::= Bottoms
```

## Statements

Most of the statement constructs are standard for imperative languages. We syntactically distinguish between empty and non-empty blocks, because we chose `Stmts` not to be a (`;`-separated) list of `Stmt`. Variables can be declared anywhere inside a block, their scope ending with the block. Expressions are allowed to be used for their side effects only (followed by a semicolon `;`). Functions are allowed to abruptly return. The exceptions are parametric, i.e., one can throw a value which is bound to the variable declared by `catch`. Threads can be dynamically created and terminated, and can synchronize with `join`,

`acquire`, `release` and `rendezvous`. Note that the strictness attributes obey the intended evaluation strategy of the various constructs. In particular, the if-then-else construct is strict only in its first argument (the if-then construct will be desugared into if-then-else), while the loop constructs are not strict in any arguments. The `print` statement construct is variadic, that is, it takes an arbitrary number of arguments.

```
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt ::= Block
                 | Exp ";"                                  [strict]
                 | "if" "(" Exp ")" Block "else" Block      [avoid, strict(1)]
                 | "if" "(" Exp ")" Block
                 | "while" "(" Exp ")" Block
                 | "for" "(" Stmt Exp ";" Exp ")" Block
                 | "return" Exp ";"                         [strict]
                 | "return" ";"
                 | "print" "(" Exps ")" ";"                 [strict]
// NOTE: print strict allows non-deterministic evaluation of its arguments
// Either keep like this but document, or otherwise make Exps seqstrict.
// Of define and use a different expression list here, which is seqstrict.
                 | "try" Block "catch" "(" Id ")" Block
                 | "throw" Exp ";"                          [strict]
                 | "join" Exp ";"                           [strict]
                 | "acquire" Exp ";"                        [strict]
                 | "release" Exp ";"                        [strict]
                 | "rendezvous" Exp ";"                     [strict]
```

The reason we allow `Stmts` as the first argument of `for` instead of `Stmt` is because we want to allow more than one statement to be executed when the loop is initialized. Also, as seens shorly, macros may expand one statement into more statements; for example, an initialized variable declaration statement `var x=0;` desugars into two statements, namely `var x; x=0;`, so if we use `Stmt` instead of `Stmts` in the production of `for` above then we risk that the macro expansion of statement `var x=0;` happens before the macro expansion of `for`, also shown below, in which case the latter would not apply anymore because of syntactic mismatch.

```
  syntax Stmt ::= Stmt Stmt                             [right]

// I wish I were able to write the following instead, but confuses the parser.
//
// syntax Stmts ::= List{Stmt,""}
// syntax Top ::= Stmt | "function" Id "(" Ids ")" Block
```

```
// syntax Pgm ::= List{Top,""}
//
// With that, I could have also eliminated the empty block
```

## Desugared Syntax

This part desugars some of SIMPLE's language constructs into core ones. We only want to give semantics to core constructs, so we get rid of the derived ones before we start the semantics. All desugaring macros below are straightforward.

```
  rule if (E) S => if (E) S else {}                                    [macro]
  rule for(Start Cond; Step) {S} => {Start while (Cond) {S Step;}}     [macro]
  rule for(Start Cond; Step) {} => {Start while (Cond) {Step;}}        [macro]
  rule var E1:Exp, E2:Exp, Es:Exps; => var E1; var E2, Es;             [macro-rec]
  rule var X:Id = E; => var X; X = E;                                  [macro]
```

For the semantics, we can therefore assume from now on that each conditional has both branches, that there are only `while` loops, and that each variable is declared alone and without any initialization as part of the declaration.

```
endmodule


module SIMPLE-UNTYPED
  imports SIMPLE-UNTYPED-SYNTAX
  imports DOMAINS
```

## Basic Semantic Infrastructure

Before one starts adding semantic rules to a **K** definition, one needs to define the basic semantic infrastructure consisting of definitions for `values` and `configuration`. As discussed in the definitions in the **K** tutorial, the values are needed to know when to stop applying the heating rules and when to start applying the cooling rules corresponding to strictness or context declarations. The configuration serves as a backbone for the process of configuration abstraction which allows users to only mention the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically. Although in some cases the configuration could be automatically inferred from the rules, we believe that it is very useful for language designers/semanticists to actually think of and design their configuration explicitly, so the current implementation of **K** requires one to define it.

## Values

We here define the values of the language that the various fragments of programs evaluate to. First, integers and Booleans are values. As discussed, arrays evaluate to special array reference values holding (1) a location from where the array's elements are contiguously allocated in the store, and (2) the size of the array. Functions evaluate to function values as -abstractions (we do not need to evaluate functions to closures because each function is executed in the fixed global environment and function definitions cannot be nested). Like in IMP and other languages, we finally tell the tool that values are **K** results.

```
syntax Val ::= Int | Bool | String
             | array(Int,Int)
             | lambda(Ids,Stmt)
syntax Exp ::= Val
syntax Exps ::= Vals
syntax Vals ::= Bottoms
syntax KResult ::= Val
                 | Vals  // TODO: should not need this
```

The inclusion of values in expressions follows the methodology of syntactic definitions (like, e.g., in SOS): extend the syntax of the language to encompass all values and additional constructs needed to give semantics. In addition to that, it allows us to write the semantic rules using the original syntax of the language, and to parse them with the same (now extended with additional values) parser. If writing the semantics directly on the **K** AST, using the associated labels instead of the syntactic constructs, then one would not need to include values in expressions.

## Configuration

The **K** configuration of SIMPLE consists of a top level cell, `T`, holding a `threads` cell, a global environment map cell `genv` mapping the global variables and function names to their locations, a shared store map cell `store` mapping each location to some value, a set cell `busy` holding the locks which have been acquired but not yet released by threads, a set cell `terminated` holding the unique identifiers of the threads which already terminated (needed for `join`), `input` and `output` list cells, and a `nextLoc` cell holding a natural number indicating the next available location. Unlike in the small languages in the **K** tutorial, where we used the fresh predicate to generate fresh locations, in larger languages, like SIMPLE, we prefer to explicitly manage memory. The location counter in `nextLoc` models an actual physical location in the store; for simplicity, we assume arbitrarily large memory and no garbage collection. The `threads` cell contains one `thread` cell for each existing thread in the program. Note that the thread cell has

multiplicity `*`, which means that at any given moment there could be zero, one or more `thread` cells. Each `thread` cell contains a computation cell `k`, a `control` cell holding the various control structures needed to jump to certain points of interest in the program execution, a local environment map cell `env` mapping the thread local variables to locations in the store, and finally a `holds` map cell indicating what locks have been acquired by the thread and not released so far and how many times (SIMPLE's locks are re-entrant). The `control` cell currently contains only two subcells, a function stack `fstack` which is a list and an exception stack `xstack` which is also a list. One can add more control structures in the `control` cell, such as a stack for break/continue of loops, etc., if the language is extended with more control-changing constructs. Note that all cells except for `k` are also initialized, in that they contain a ground term of their corresponding sort. The `k` cell is initialized with the program that will be passed to the **K** tool, as indicated by the `$PGM` variable, followed by the `execute` task (defined shortly).

```
// the syntax declarations below are required because the sorts are
// referenced directly by a production and, because of the way KIL to KORE
// is implemented, the configuration syntax is not available yet
// should simply work once KIL is removed completely
// check other definitions for this hack as well

syntax ControlCell
syntax ControlCellFragment

configuration <T color="red">
                <threads color="orange">
                  <thread multiplicity="*" color="yellow">
                    <k color="green"> $PGM:Stmt ~> execute </k>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <control color="cyan">
                      <fstack color="blue"> .List </fstack>
                      <xstack color="purple"> .List </xstack>
                    </control>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <env color="violet"> .Map </env>
                    <holds color="black"> .Map </holds>
                    <id color="pink"> 0 </id>
                  </thread>
                </threads>
              //<br/> // TODO(KORE): support latex annotations #1799
                <genv color="pink"> .Map </genv>
                <store color="white"> .Map </store>
                <busy color="cyan"> .Set </busy>
```

```
  <terminated color="red"> .Set </terminated>
//<br/> // TODO(KORE): support latex annotations #1799
  <input color="magenta" stream="stdin"> .List </input>
  <output color="brown" stream="stdout"> .List </output>
  <nextLoc color="gray"> 0 </nextLoc>
</T>
```

## Declarations and Initialization

We start by defining the semantics of declarations (for variables, arrays and functions).

## Variable Declaration

The SIMPLE syntax was desugared above so that each variable is declared alone and its initialization is done as a separate statement. The semantic rule below matches resulting variable declarations of the form `var X;` on top of the `k` cell (indeed, note that the `k` cell is complete, or round, to the left, and is torn, or ruptured, to the right), allocates a fresh location `L` in the store which is initialized with a special value (indeed, the unit `.`, or nothing, is matched anywhere in the map note the tears at both sides and replaced with the mapping `L` ), and binds `X` to `L` in the local environment shadowing previous declarations of `X`, if any. This possible shadowing of `X` requires us to therefore update the entire environment map, which is expensive and can significantly slow down the execution of larger programs. On the other hand, since we know that `L` is not already bound in the store, we simply add the binding `L` to the store, thus avoiding a potentially complete traversal of the the store map in order to update it. We prefer the approach used for updating the store whenever possible, because, in addition to being faster, it offers more true concurrency than the latter; indeed, according to the concurrent semantics of `K`, the store is not frozen while `L` is added to it, while the environment is frozen during the update operation `Env[L/X]`. The variable declaration command is also removed from the top of the computation cell and the fresh location counter is incremented. The undefined symbol added in the store is of sort `KItem`, instead of `Val`, on purpose; this way, the store lookup rules will get stuck when one attempts to lookup an uninitialized location. All the above happen in one transactional step, with the rule below. Note also how configuration abstraction allows us to only mention the needed cells; indeed, as the configuration above states, the `k` and `env` cells are actually located within a `thread` cell within the `threads` cell, but one needs not mention these: the configuration context of the rule is

automatically transformed to match the declared configuration structure.

```
syntax KItem ::= "undefined"  [latex(\bot)]

rule <k> var X:Id; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

## Array Declaration

The **K** semantics of the uni-dimensional array declaration is somehow similar to the above declaration of ordinary variables. First, note the context declaration below, which requests the evaluation of the array dimension. Once evaluated, say to a natural number `N`, then `N +Int 1` locations are allocated in the store for an array of size `N`, the additional location (chosen to be the first one allocated) holding the array reference value. The array reference value `array(L,N)` states that the array has size `N` and its elements are located contiguously in the store starting with location `L`. The operation `L … L'  V`, defined at the end of this file in the auxiliary operation section, initializes each location in the list `L … L'` to `V`. Note that, since the dimensions of array declarations can be arbitrary expressions, this virtually means that we can dynamically allocate memory in SIMPLE by means of array declarations.

```
context var _:Id[HOLE];

rule <k> var X:Id[N:Int]; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> array(L +Int 1, N)
                        (L +Int 1) ... (L +Int N) |-> undefined ...</store>
     <nextLoc> L => L +Int 1 +Int N </nextLoc>
  requires N >=Int 0
```

SIMPLE allows multi-dimensional arrays. For semantic simplicity, we desugar them all into uni-dimensional arrays by code transformation. This way, we only need to give semantics to uni-dimensional arrays. First, note that the context rule above actually evaluates all the array dimensions (that's why we defined the expression lists strict!): Upon evaluating the array dimensions, the code generation rule below desugars multi-dimensional array declaration to uni-dimensional declarations. To this aim, we introduce two special unique variable identifiers, `$1` and `$2`. The first variable, `$1`, iterates through and initializes each element of the first dimension with an array

of the remaining dimensions, declared as variable `$2`:

```
syntax Id ::= "$1" | "$2"
rule var X:Id[N1:Int, N2:Int, Vs:Vals];
  => var X[N1];
     {
       for(var $1 = 0; $1 <= N1 - 1; ++$1) {
         var $2[N2, Vs];
         X[$1] = $2;
       }
     }
   [structural]
```

Ideally, one would like to perform syntactic desugarings like the one above before the actual semantics. Unfortunately, that was not possible in this case because the dimension expressions of the multi-dimensional array need to be evaluated first. Indeed, the desugaring rule above does not work if the dimensions of the declared array are arbitrary expressions, because they can have side effects (e.g., `a[++x,++x]`) and those side effects would be propagated each time the expression is evaluated in the desugaring code (note that both the loop condition and the nested multi-dimensional declaration would need to evaluate the expressions given as array dimensions).

## Function declaration

Functions are evaluated to  -abstractions and stored like any other values in the store. A binding is added into the environment for the function name to the location holding its body. Similarly to the C language, SIMPLE only allows function declarations at the top level of the program. More precisely, the subsequent semantics of SIMPLE only works well when one respects this requirement. Indeed, the simplistic context-free parser generated by the grammar above is more generous than we may want, in that it
allows function declarations anywhere any declaration is allowed, including inside arbitrary blocks. However, as the rule below shows, we are `not` storing the declaration environment with the  -abstraction value as closures do. Instead, as seen shortly, we switch to the global environment whenever functions are invoked, which is consistent with our requirement that functions should only be declared at the top. Thus, if one declares local functions, then one may see unexpected behaviors (e.g., when one shadows a global variable before declaring a local function). The type checker of SIMPLE, also defined in **K** (see `examples/simple/typed/static`), discards programs which do not respect this requirement.

```
rule <k> function F(Xs) S => . ...</k>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L |-> lambda(Xs, S) ...</store>
```

```
      <nextLoc> L => L +Int 1 </nextLoc>
```

When we are done with the first pass (pre-processing), the computation cell `k` contains only the token `execute` (see the configuration declaration above, where the computation item `execute` was placed right after the program in the `k` cell of the initial configuration) and the cell `genv` is empty. In this case, we have to call `main()` and to initialize the global environment by transferring the contents of the local environment into it. We prefer to do it this way, as opposed to processing all the top level declarations directly within the global environment, because we want to avoid duplication of semantics: the syntax of the global declarations is identical to that of their corresponding local declarations, so the semantics of the latter suffices provided that we copy the local environment into the global one once we are done with the pre-processing. We want this separate pre-processing step precisely because we want to create the global environment. All (top-level) functions end up having their names bound in the global environment and, as seen below, they are executed in that same global environment; all these mean, in particular, that the functions "see" each other, allowing for mutual recursion, etc.

```
syntax KItem ::= "execute"
rule <k> execute => main(.Exps); </k>
     <env> Env </env>
     <genv> .Map => Env </genv>  [structural]
```

## Expressions

We next define the **K** semantics of all the expression constructs.

## Variable lookup

When a variable `X` is the first computational task, and `X` is bound to some location `L` in the environment, and `L` is mapped to some value `V` in the store, then we rewrite `X` into `V`:

```
rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]
```

Note that the rule above excludes reading , because  is not a value and `V` is checked at runtime to be a value.

## Variable/Array increment

This is tricky, because we want to allow both `++x` and `++a[5]`. Therefore, we need to extract the lvalue of the expression to increment. To do that, we state that the expression to increment should be wrapped by the auxiliary `lvalue` operation and then evaluated. The semantics

of this auxiliary operation is defined at the end of this file. For now, all we need to know is that it takes an expression and evaluates to a location value. Location values, also defined at the end of the file, are integers wrapped with the operation `loc`, to distinguish them from ordinary integers.

```
context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I => I +Int 1) ...</store>  [increment]
```

## Arithmetic operators

There is nothing special about the following rules. They rewrite the language constructs to their library counterparts when their arguments become values of expected sorts:

```
rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 requires I2 =/=K 0
rule I1 % I2 => I1 %Int I2 requires I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2
```

The equality and inequality constructs reduce to syntactic comparison of the two argument values (which is what the equality on `K` terms does).

```
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
```

The logical negation is clear, but the logical conjunction and disjunction are short-circuited:

```
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E
```

## Array lookup

Untyped SIMPLE does not check array bounds (the dynamically typed version of
it, in `examples/simple/typed/dynamic`, does check for array out of bounds). The first rule below desugars the multi-dimensional array access to

uni-dimensional array access; recall that the array access operation was declared strict, so all sub-expressions involved are already values at this stage. The second rule rewrites the array access to a lookup operation at a precise location; we prefer to do it this way to avoid locking the store. The semantics of the auxiliary `lookup` operation is straightforward, and is defined at the end of the file.

```
// The [anywhere] feature is underused, because it would only be used
// at the top of the computation or inside the lvalue wrapper. So it
// may not be worth, or we may need to come up with a special notation
// allowing us to enumerate contexts for [anywhere] rules.
  rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
    [structural, anywhere]

  rule array(L,_)[N:Int] => lookup(L +Int N)
    [structural, anywhere]
```

## Size of an array

The size of the array is stored in the array reference value, and the `sizeOf` construct was declared strict, so:

```
  rule sizeOf(array(_,N)) => N
```

## Function call

Function application was strict in both its arguments, so we can assume that both the function and its arguments are evaluated to values (the former expected to be a  -abstraction). The first rule below matches a well-formed function application on top of the computation and performs the following steps atomically: it switches to the function body followed by `return;` (for the case in which the function does not use an explicit return statement); it pushes the remaining computation, the current environment, and the current control data onto the function stack (the remaining computation can thus also be discarded from the computation cell, because an unavoidable subsequent `return` statement  see above  will always recover it from the stack); it switches the current environment (which is being pushed on the function stack) to the global environment, which is where the free variables in the function body should be looked up; it binds the formal parameters to fresh locations in the new environment, and stores the actual arguments to those locations in the store (this latter step is easily done by reducing the problem to variable declarations, whose semantics we have already defined; the auxiliary operation `mkDecls` is defined at the end of the file). The second rule pops the computation, the environment and the control data from the function

stack when a `return` statement is encountered as the next computational task, passing the returned value to the popped computation (the popped computation was the context in which the returning function was called). Note that the pushing/popping of the control data is crucial. Without it, one may have a function that contains an exception block with a return statement inside, which would put the `xstack` cell in an inconsistent state (since the exception block modifies it, but that modification should be irrelevant once the function returns). We add an artificial `nothing` value to the language, which is returned by the nulary `return;` statements.

```
syntax KItem ::=  (Map,K,ControlCellFragment)

rule <k> lambda(Xs,S)(Vs:Vals) ~> K => mkDecls(Xs,Vs) S return; </k>
     <control>
       <fstack> .List => ListItem((Env,K,C)) ...</fstack>
       C
     </control>
     <env> Env => GEnv </env>
     <genv> GEnv </genv>

rule <k> return(V:Val); ~> _ => V ~> K </k>
     <control>
       <fstack> ListItem((Env,K,C)) => .List ...</fstack>
       (_ => C)
     </control>
     <env> _ => Env </env>

syntax Val ::= "nothing"
rule return; => return nothing;   [macro]
```

Like for division-by-zero, it is left unspecified what happens when the `nothing` value is used in domain calculations. For example, from the the perspective of the language semantics, `7 +Int nothing` can evaluate to anything, or may not evaluate at all (be undefined). If one wants to make sure that such artificial values are never misused, then one needs to define a static checker (also using **K**, like our the type checker in `examples/simple/typed/static`) and reject programs that do. Note that, unlike the undefined symbol  which had the sort `K` instead of `Val`, we defined `nothing` to be a value. That is because, as explained above, we do not want the program to get stuck when nothing is returned by a function. Instead, we want the behavior to be unspecified; in particular, if one is careful to never use the returned value in domain computation, like it happens when we call a function for its side effects (e.g., with a statement of the

form `f(x);`), then the program does not get stuck.

## Read

The `read()` expression construct simply evaluates to the next input value, at the same time discarding the input value from the `in` cell.

```
rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]
```

## Assignment

In SIMPLE, like in C, assignments are expression constructs and not statement constructs. To make it a statement all one needs to do is to follow it by a semi-colon `;` (see the semantics for expression statements below).
Like for the increment, we want to allow assignments not only to variables but also to array elements, e.g., `e1[e2] = e3` where `e1` evaluates to an array reference, `e2` to a natural number, and `e3` to any value. Thus, we first compute the lvalue of the left-hand-side expression that appears in an assignment, and then we do the actual assignment to the resulting location:

```
context (HOLE => lvalue(HOLE)) = _

rule <k> loc(L) = V:Val => V ...</k> <store>... L |-> (_ => V) ...</store>
  [assignment]
```

## Statements

We next define the **K** semantics of statements.

## Blocks

Empty blocks are simply discarded, as shown in the first rule below.
For non-empty blocks, we schedule the enclosed statement but we have to make sure the environment is recovered after the enclosed statement executes. Recall that we allow local variable declarations, whose scope is the block enclosing them. That is the reason for which we have to recover the environment after the block. This allows us to have a very simple semantics for variable declarations, as we did above. One can make the two rules below computational if one wants them to count as computational steps.

```
rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]
```

The basic definition of environment recovery is straightforward and given in the section on auxiliary constructs at the end of the file.

There are two common alternatives to the above semantics of blocks. One is to keep track of the variables which are declared in the block and only recover those at the end of the block. This way one does more work for variable declarations but conceptually less work for environment recovery; we say `conceptually` because it is not clear that it is indeed the case that one does less work when AC matching is involved. The other alternative is to work with a stack of environments instead of a flat environment, and push the current environment when entering a block and pop it when exiting it. This way, one does more work when accessing variables (since one has to search the variable in the environment stack in a top-down manner), but on the other hand uses smaller environments and the definition gets closer to an implementation. Based on experience with dozens of language semantics and other **K** definitions, we have found that our approach above is the best trade-off between elegance and efficiency (especially since rewrite engines have built-in techniques to lazily copy terms, by need, thus not creating unnecessary copies), so it is the one that we follow in general.

## Sequential composition

Sequential composition is desugared into **K**'s builtin sequentialization operation (recall that, like in C, the semi-colon `;` is not a statement separator in SIMPLE — it is either a statement terminator or a construct for a statement from an expression). The rule below is structural, so it does not count as a computational step. One can make it computational if one wants it to count as a step. Note that **K** allows to define the semantics of SIMPLE in such a way that statements eventually dissolve from the top of the computation when they are completed; this is in sharp contrast to (artificially) `evaluating` them to a special `skip` statement value and then getting rid of that special value, as it is the case in other semantic approaches (where everything must evaluate to something). This means that once `S` completes in the rule below, `S` becomes automatically the next computation item without any additional (explicit or implicit) rules.

```
rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
```

A subtle aspect of the rule above is that `S` is declared to have sort `Stmts` and not `Stmt`. That is because desugaring macros can indeed produce left associative sequential composition of statements. For example, the code `var x=0; x=1;` is desugared to `(var x; x=0;) x=1;`, so although originally the first term of the sequential composition had sort `Stmt`, after desugaring it became of sort `Stmts`. Note that the attribute `[right]` associated to the sequential compositon production is an attribute of the syntax, and not of the semantics: e.g., it tells the parser to parse `var x; x=0; x=1;` as `var x; (x=0; x=1;)`, but it does not tell the rewrite engine to rewrite `(var x; x=0;) x=1;` to

```
var x; (x=0; x=1;).
```

## Expression statements

Expression statements are only used for their side effects, so their result value is simply discarded. Common examples of expression statements are ones of the form `++x;`, `x=e;`, `e1[e2]=e3;`, etc.

```
rule _:Val; => .
```

## Conditional

Since the conditional was declared with the `strict(1)` attribute, we can assume that its first argument will eventually be evaluated. The rules below cover the only two possibilities in which the conditional is allowed to proceed (otherwise the rewriting process gets stuck).

```
rule if ( true) S else _ => S
rule if (false) _ else S => S
```

## While loop

The simplest way to give the semantics of the while loop is by unrolling. Note, however, that its unrolling is only allowed when the while loop reaches the top of the computation (to avoid non-termination of unrolling). We prefer the rule below to be structural, because we don't want the unrolling of the while loop to count as a computational step; this is unavoidable in conventional semantics, but it is possible in **K** thanks to its distinction between structural and computational rules. The simple while loop semantics below works because our while loops in SIMPLE are indeed very basic. If we allowed break/continue of loops then we would need a completely different semantics, which would also involve the `control` cell.

```
rule while (E) S => if (E) {S while(E)S}  [structural]
```

## Print

The `print` statement was strict, so all its arguments are now evaluated (recall that `print` is variadic). We append each of its evaluated arguments to the output buffer, and discard the residual `print` statement with an empty list of arguments.

```
rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  [print]
rule print(.Vals); => .  [structural]
```

## Exceptions

SIMPLE allows parametric exceptions, in that one can throw and catch a particular value. The statement `try S  catch(X) S` proceeds with the evaluation of `S` . If `S` evaluates normally, i.e., without any exception thrown, then `S` is discarded and the execution continues normally. If `S` throws an exception with a statement of the form `throw E`, then `E` is first evaluated to some value `V` (`throw` was declared to be strict), then `V` is bound to `X`, then `S` is evaluated in the new environment while the reminder of `S` is discarded, then the environment is recovered and the execution continues normally with the statement following the `try S  catch(X) S` statement. Exceptions can be nested and the statements in the `catch` part (`S` in our case) can throw exceptions to the upper level. One should be careful with how one handles the control data structures here, so that the abrupt changes of control due to exception throwing and to function returns interact correctly with each other. For example, we want to allow function calls inside the statement `S` in a `try S  catch(X) S` block which can throw an exception that is not caught by the function but instead is propagated to the `try S  catch(X) S` block that called the function. Therefore, we have to make sure that the function stack as well as other potential control structures are also properly modified when the exception is thrown to correctly recover the execution context. This can be easily achieved by pushing/popping the entire current control context onto the exception stack. The three rules below modularly do precisely the above.

```
syntax KItem ::= (Id,Stmt,K,Map,ControlCellFragment)

syntax KItem ::= "popx"

rule <k> (try S1 catch(X) {S2} => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem((X, S2, K, Env, C)) ...</xstack>
       C
     </control>
     <env> Env </env>

rule <k> popx => . ...</k>
     <xstack> ListItem(_) => .List ...</xstack>

rule <k> throw V:Val; ~> _ => { var X = V; S2 } ~> K </k>
     <control>
       <xstack> ListItem((X, S2, K, Env, C)) => .List ...</xstack>
       (_ => C)
     </control>
```

```
<env> _ => Env </env>
```

The catch statement `S` needs to be executed in the original environment, but where the thrown value `V` is bound to the catch variable `X`. We here chose to rely on two previously defined constructs when giving semantics to the catch part of the statement: (1) the variable declaration with initialization, for binding `X` to `V`; and (2) the block construct for preventing `X` from shadowing variables in the original environment upon the completion of `S` .

## Threads

SIMPLE's threads can be created and terminated dynamically, and can synchronize by acquiring and releasing re-entrant locks and by rendezvous. We discuss the seven rules giving the semantics of these operations below.

## Thread creation

Threads can be created by any other threads using the `spawn S` construct. The spawn expression construct evaluates to the unique identifier of the newly created thread and, at the same time, a new thread cell is added into the configuration, initialized with the `S` statement and sharing the same environment with the parent thread. Note that the newly created `thread` cell is torn. That means that the remaining cells are added and initialized automatically as described in the definition of SIMPLE's configuration. This is part of **K**'s configuration abstraction mechanism.

```
rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
            ...</thread>)
```

## Thread termination

Dually to the above, when a thread terminates its assigned computation (the contents of its `k` cell) is empty, so the thread can be dissolved.
However, since no discipline is imposed on how locks are acquired and released, it can be the case that a terminating thread still holds locks. Those locks must be released, so other threads attempting to acquire them do not deadlock. We achieve that by removing all the locks held by the terminating thread in its `holds` cell from the set of busy locks in the `busy` cell
(`keys(H)` returns the domain of the map `H` as a set, that is, only

the locks themselves ignoring their multiplicity). As seen below, a lock is added to the `busy` cell as soon as it is acquired for the first time by a thread. The unique identifier of the terminated thread is also collected into the `terminated` cell, so the `join` construct knows which threads have terminated.

```
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

## Thread joining

Thread joining is now straightforward: all we need to do is to check whether the identifier of the thread to be joined is in the `terminated` cell. If yes, then the `join` statement dissolves and the joining thread continues normally; if not, then the joining thread gets stuck.

```
rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>
```

## Acquire lock

There are two cases to distinguish when a thread attempts to acquire a lock (in SIMPLE any value can be used as a lock):
(1) The thread does not currently have the lock, in which case it has to take it provided that the lock is not already taken by another thread (see the side condition of the first rule).
(2) The thread already has the lock, in which case it just increments its counter for the lock (the locks are re-entrant). These two cases are captured by the two rules below:

```
rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  requires (notBool(V in Busy))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N => N +Int 1) ...</holds>
```

## Release lock

Similarly, there are two corresponding cases to distinguish when a thread releases a lock:
(1) The thread holds the lock more than once, in which case all it needs to do is to decrement the lock counter.
(2) The thread holds the lock only once, in which case it needs to remove it from its `holds` cell and also from the the shared `busy` cell, so other threads can acquire it if they need to.

```
rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N -Int 1) ...</holds>
  requires N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>
```

## Rendezvous synchronization

In addition to synchronization through acquire and release of locks, SIMPLE also provides a construct for rendezvous synchronization. A thread whose next statement to execute is `rendezvous(V)` gets stuck until another thread reaches an identical statement; when that happens, the two threads drop their rendezvous statements and continue their executions. If three threads happen to have an identical rendezvous statement as their next statement, then precisely two of them will synchronize and the other will remain blocked until another thread reaches a similar rendezvous statement. The rule below is as simple as it can be. Note, however, that, again, it is **K**'s mechanism for configuration abstraction that makes it work as desired: since the only cell which can multiply containing a `k` cell inside is the `thread` cell, the only way to concretize the rule below to the actual configuration of SIMPLE is to include each `k` cell in a `thread` cell.

```
rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

## Auxiliary declarations and operations

In this section we define all the auxiliary constructs used in the above semantics.

## Making declarations

The `mkDecls` auxiliary construct turns a list of identifiers and a list of values in a sequence of corresponding variable declarations.

```
syntax Stmt ::= mkDecls(Ids,Vals)  [function]
rule mkDecls((X:Id, Xs:Ids), (V:Val, Vs:Vals)) => var X=V; mkDecls(Xs,Vs)
rule mkDecls(.Ids,.Vals) => {}
```

## Location lookup

The operation below is straightforward. Note that we tag it with the same `lookup` tag as the variable lookup rule defined above. This way,

both rules will be considered transitions when we include the `lookup` tag in the transition option of `kompile`.

```
syntax Exp ::= lookup(Int)
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]
```

## Environment recovery

We have already discussed the environment recovery auxiliary operation in the IMP++ tutorial:

```
// TODO: eliminate the env wrapper, like we did in IMP++

  syntax KItem ::= setEnv(Map)
  rule <k> setEnv(Env) => . ...</k> <env> _ => Env </env>  [structural]
```

While theoretically sufficient, the basic definition for environment recovery alone is suboptimal. Consider a loop `while (E)S`, whose semantics (see above) was given by unrolling. `S` is a block. Then the semantics of blocks above, together with the unrolling semantics of the while loop, will yield a computation structure in the `k` cell that increasingly grows, adding a new environment recovery task right in front of the already existing sequence of similar environment recovery tasks (this phenomenon is similar to the "tail recursion" problem). Of course, when we have a sequence of environment recovery tasks, we only need to keep the last one. The elegant rule below does precisely that, thus avoiding the unnecessary computation explosion problem:

```
rule (setEnv(_) => .) ~> setEnv(_)  [structural]
```

In fact, the above follows a common convention in **K** for recovery operations of cell contents: the meaning of a computation task of the form `cell(C)` that reaches the top of the computation is that the current contents of cell `cell` is discarded and gets replaced with `C`. We did not add support for these special computation tasks in our current implementation of **K**, so we need to define them as above.

## lvalue and loc

For convenience in giving the semantics of constructs like the increment and the assignment, that we want to operate the same way on variables and on array elements, we used an auxiliary `lvalue(E)` construct which was expected to evaluate to the lvalue of the expression `E`. This is only defined when `E` has an lvalue, that is, when `E` is either a variable or evaluates to an array element. `lvalue(E)` evaluates to a value of the form `loc(L)`, where `L` is the location where the value of `E` can be found; for clarity, we use `loc` to structurally distinguish

natural numbers from location values. In giving semantics to `lvalue` there are two cases to consider. (1) If `E` is a variable, then all we need to do is to grab its location from the environment. (2) If `E` is an array element, then we first evaluate the array and its index in order to identify the exact location of the element of concern, and then return that location; the last rule below works because its preceding context declarations ensure that the array and its index are evaluated, and then the rule for array lookup (defined above) rewrites the evaluated array access construct to its corresponding store lookup operation.

```
// For parsing reasons, we prefer to allow lvalue to take a K

  syntax Exp ::= lvalue(K)
  syntax Val ::= loc(Int)

// Local variable

  rule <k> lvalue(X:Id => loc(L)) ...</k> <env>... X |-> L:Int ...</env>
    [structural]

// Array element: evaluate the array and its index;
// then the array lookup rule above applies.

  context lvalue(_::Exp[HOLE::Exps])
  context lvalue(HOLE::Exp[_::Exps])

// Finally, return the address of the desired object member

  rule lvalue(lookup(L:Int) => loc(L))  [structural]
```

## Initializing multiple locations

The following operation initializes a sequence of locations with the same value:

```
  syntax Map ::= Int "..." Int "|->" K
    [function, latex({#1}\ldots{#2}\mapsto{#3})]
  rule N...M |-> _ => .Map  requires N >Int M
  rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  requires N <=Int M
```

The semantics of SIMPLE is now complete. Make sure you kompile the definition with the right options in order to generate the desired model. No kompile options are needed if you only only want to execute the definition (and thus get an interpreter), but if you want to search for a different program behaviors then you need to kompile with the transition option including rule tags such as lookup, increment, acquire, etc. See the IMP++ tutorial for what the transition option means how to use it.

`endmodule`

Go to Lesson 2, SIMPLE typed static

# SIMPLE — Untyped

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped SIMPLE language.
SIMPLE is intended to be a pedagogical and research language that captures the essence of the imperative programming paradigm, extended with several features often encountered in imperative programming languages.
A program consists of a set of global variable declarations and function definitions. Like in C, function definitions cannot be nested and each program must have one function called `main`, which is invoked when the program is executed. To make it more interesting and to highlight some of **K**'s strengths, SIMPLE includes the following features in addition to the conventional imperative expression and statement constructs:

- Multidimensional arrays and array references. An array evaluates to an array reference, which is a special value holding a location (where the elements of the array start) together with the size of the array; the elements of the array can be array references themselves (particularly when the array is multi-dimensional). Array references are ordinary values,
  so they can be assigned to variables and passed/received by functions.
- Functions and function values. Functions can have zero or more parameters and can return abruptly using a `return` statement. SIMPLE follows a call-by-value parameter passing style, with static scoping.
  Function names evaluate to function abstractions, which hereby become ordinary
  values in the language, same like the array references.
- Blocks with locals. SIMPLE variables can be declared anywhere, their scope being from the place where they are declared until the end of the most nested enclosing block.
- Input/Output. The expression `read()` evaluates to the next value in the input buffer, and the statement `write(e)`

evaluates `e` and outputs its value to the output buffer. The input and output buffers are lists of values.

- Exceptions. SIMPLE has parametric exceptions (the value thrown as an exception can be caught and bound).
- Concurrency via dynamic thread creation/termination and synchronization. One can spawn a thread to execute any statement. The spawned thread shares with its parent its environment at creation time.
  Threads can be synchronized via a join command which blocks the current thread
  until the joined thread completes, via re-entrant locks which can be acquired
  and released, as well as through rendezvous commands.

Like in many other languages, some of SIMPLE's constructs can be desugared into a smaller set of basic constructs. We do that at the end of the syntax module, and then we only give semantics to the core constructs.

**Note**: This definition is commented slightly more than others, because it is intended to be one of the first non-trivial definitions that the new user of **K** sees. We recommend the beginner user to first check the language definitions discussed in the **K** tutorial.

```
module SIMPLE-UNTYPED-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

We start by defining the SIMPLE syntax. The language constructs discussed above have the expected syntax and evaluation strategies. Recall that in **K** we annotate the syntax with appropriate strictness attributes, thus giving each language construct the desired evaluation strategy.

## Identifiers

Recall from the **K** tutorial that identifiers are builtin and come under the syntactic category `Id`. The special identifier for the function `main` belongs to all programs, and plays a special role in the semantics, so we declare it explicitly. This would not be necessary if the identifiers were all included automatically in semantic definitions, but that is not possible because of parsing reasons (e.g., **K** variables used to match concrete identifiers would then be ambiguously parsed as identifiers). They are only included in the parser generated to parse programs (and used by the `kast` tool). Consequently, we have to explicitly declare all the concrete identifiers that play a special role in the semantics, like `main` below.

```
syntax Id ::= "main" [token]
```

## Declarations

There are two types of declarations: for variables (including arrays) and for functions. We are going to allow declarations of the form `var x=10, a[10,10], y=23;`, which is why we allow the `var` keyword to take a list of expressions. The non-terminals used in the two productions below are defined shortly.

```
syntax Stmt ::= "var" Exps ";"
              | "function" Id "(" Ids ")" Block
```

## Expressions

The expression constructs below are standard. Increment (`++`) takes an expression rather than a variable because it can also increment an array element. Recall that the syntax we define in **K** is what we call *the syntax of the semantics*: while powerful enough to define non-trivial syntaxes (thanks to the underlying SDF technology that we use), we typically refrain from defining precise syntaxes, that is, ones which accept precisely the well-formed programs (that would not be possible anyway in general). That job is deferred to type systems, which can also be defined in **K**. In other words, we are not making any effort to guarantee syntactically that only variables or array elements are passed to the increment construct, we allow any expression. Nevertheless, we will only give semantics to those, so expressions of the form `++5`, which parse (but which will be rejected by our type system in the typed version of SIMPLE later), will get stuck when executed. Arrays can be multidimensional and can hold other arrays, so their lookup operation takes a list of expressions as argument and applies to an expression (which can in particular be another array lookup), respectively. The construct `sizeOf` gives the size of an array in number of elements of its first dimension. Note that almost all constructs are strict. The only constructs which are not strict are the increment (since its first argument gets updated, so it cannot be evaluated), the input read which takes no arguments so strictness is irrelevant for it, the logical and and or constructs which are short-circuited, the thread spawning construct which creates a new thread executing the argument expression and return its unique identifier to the creating thread (so it cannot just evaluate its argument in place), and the assignment which is only strict in its second argument (for the same reason as the increment).

```
syntax Exp ::= Int | Bool | String | Id
             | "(" Exp ")"             [bracket]
             | "++" Exp
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict]
```

```
| "-" Exp                  [strict]
| "sizeOf" "(" Exp ")"     [strict]
| "read" "(" ")"
> left:
  Exp "*" Exp              [strict, left]
| Exp "/" Exp              [strict, left]
| Exp "%" Exp              [strict, left]
> left:
  Exp "+" Exp              [strict, left]
| Exp "-" Exp              [strict, left]
> non-assoc:
  Exp "<" Exp              [strict, non-assoc]
| Exp "<=" Exp             [strict, non-assoc]
| Exp ">" Exp              [strict, non-assoc]
| Exp ">=" Exp             [strict, non-assoc]
| Exp "==" Exp             [strict, non-assoc]
| Exp "!=" Exp             [strict, non-assoc]
> "!" Exp                  [strict]
> left:
  Exp "&&" Exp             [strict(1), left]
| Exp "||" Exp             [strict(1), left]
> "spawn" Block
> Exp "=" Exp              [strict(2), right]
```

We also need comma-separated lists of identifiers and of expressions. Moreover, we want them to be strict, that is, to evaluate to lists of results whenever requested (e.g., when they appear as strict arguments of the constructs above).

```
syntax Ids  ::= List{Id,","}
syntax Exps ::= List{Exp,","}          [strict]  // automatically hybrid now
syntax Exps ::= Ids
syntax Val
syntax Vals ::= List{Val,","}
syntax Bottom
syntax Bottoms ::= List{Bottom,","}
syntax Ids ::= Bottoms
```

## Statements

Most of the statement constructs are standard for imperative languages. We syntactically distinguish between empty and non-empty blocks, because we chose `Stmts` not to be a (`;`-separated) list of `Stmt`. Variables can be declared anywhere inside a block, their scope ending with the block. Expressions are allowed to be used for their side effects only (followed by a semicolon `;`). Functions are allowed to abruptly return. The exceptions are parametric, i.e., one can throw a value

which is bound to the variable declared by `catch`. Threads can be dynamically created and terminated, and can synchronize with `join`, `acquire`, `release` and `rendezvous`. Note that the strictness attributes obey the intended evaluation strategy of the various constructs. In particular, the if-then-else construct is strict only in its first argument (the if-then construct will be desugared into if-then-else), while the loop constructs are not strict in any arguments. The `print` statement construct is variadic, that is, it takes an arbitrary number of arguments.

```
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt ::= Block
                 | Exp ";"                                  [strict]
                 | "if" "(" Exp ")" Block "else" Block      [avoid, strict(1)]
                 | "if" "(" Exp ")" Block
                 | "while" "(" Exp ")" Block
                 | "for" "(" Stmt Exp ";" Exp ")" Block
                 | "return" Exp ";"                         [strict]
                 | "return" ";"
                 | "print" "(" Exps ")" ";"                 [strict]
// NOTE: print strict allows non-deterministic evaluation of its arguments
// Either keep like this but document, or otherwise make Exps seqstrict.
// Of define and use a different expression list here, which is seqstrict.
                 | "try" Block "catch" "(" Id ")" Block
                 | "throw" Exp ";"                          [strict]
                 | "join" Exp ";"                           [strict]
                 | "acquire" Exp ";"                        [strict]
                 | "release" Exp ";"                        [strict]
                 | "rendezvous" Exp ";"                     [strict]
```

The reason we allow `Stmts` as the first argument of `for` instead of `Stmt` is because we want to allow more than one statement to be executed when the loop is initialized. Also, as seens shorly, macros may expand one statement into more statements; for example, an initialized variable declaration statement `var x=0;` desugars into two statements, namely `var x; x=0;`, so if we use `Stmt` instead of `Stmts` in the production of `for` above then we risk that the macro expansion of statement `var x=0;` happens before the macro expansion of `for`, also shown below, in which case the latter would not apply anymore because of syntactic mismatch.

```
  syntax Stmt ::= Stmt Stmt                             [right]

// I wish I were able to write the following instead, but confuses the parser.
//
```

```
// syntax Stmts ::= List{Stmt,""}
// syntax Top ::= Stmt | "function" Id "(" Ids ")" Block
// syntax Pgm ::= List{Top,""}
//
// With that, I could have also eliminated the empty block
```

## Desugared Syntax

This part desugars some of SIMPLE's language constructs into core ones. We only want to give semantics to core constructs, so we get rid of the derived ones before we start the semantics. All desugaring macros below are straightforward.

```
  rule if (E) S => if (E) S else {}                                  [macro]
  rule for(Start Cond; Step) {S} => {Start while (Cond) {S Step;}}   [macro]
  rule for(Start Cond; Step) {} => {Start while (Cond) {Step;}}      [macro]
  rule var E1:Exp, E2:Exp, Es:Exps; => var E1; var E2, Es;           [macro-rec]
  rule var X:Id = E; => var X; X = E;                                [macro]
```

For the semantics, we can therefore assume from now on that each conditional has both branches, that there are only `while` loops, and that each variable is declared alone and without any initialization as part of the declaration.

```
endmodule


module SIMPLE-UNTYPED
  imports SIMPLE-UNTYPED-SYNTAX
  imports DOMAINS
```

## Basic Semantic Infrastructure

Before one starts adding semantic rules to a **K** definition, one needs to define the basic semantic infrastructure consisting of definitions for `values` and `configuration`. As discussed in the definitions in the **K** tutorial, the values are needed to know when to stop applying the heating rules and when to start applying the cooling rules corresponding to strictness or context declarations. The configuration serves as a backbone for the process of configuration abstraction which allows users to only mention the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically. Although in some cases the configuration could be automatically inferred from the rules, we believe that it is very useful for language designers/semanticists to actually think of and design their configuration explicitly, so the current implementation of **K** requires one to define it.

## Values

We here define the values of the language that the various fragments of programs evaluate to. First, integers and Booleans are values. As discussed, arrays evaluate to special array reference values holding (1) a location from where the array's elements are contiguously allocated in the store, and (2) the size of the array. Functions evaluate to function values as -abstractions (we do not need to evaluate functions to closures because each function is executed in the fixed global environment and function definitions cannot be nested). Like in IMP and other languages, we finally tell the tool that values are **K** results.

```
syntax Val ::= Int | Bool | String
             | array(Int,Int)
             | lambda(Ids,Stmt)
syntax Exp ::= Val
syntax Exps ::= Vals
syntax Vals ::= Bottoms
syntax KResult ::= Val
                 | Vals  // TODO: should not need this
```

The inclusion of values in expressions follows the methodology of syntactic definitions (like, e.g., in SOS): extend the syntax of the language to encompass all values and additional constructs needed to give semantics. In addition to that, it allows us to write the semantic rules using the original syntax of the language, and to parse them with the same (now extended with additional values) parser. If writing the semantics directly on the **K** AST, using the associated labels instead of the syntactic constructs, then one would not need to include values in expressions.

## Configuration

The **K** configuration of SIMPLE consists of a top level cell, `T`, holding a `threads` cell, a global environment map cell `genv` mapping the global variables and function names to their locations, a shared store map cell `store` mapping each location to some value, a set cell `busy` holding the locks which have been acquired but not yet released by threads, a set cell `terminated` holding the unique identifiers of the threads which already terminated (needed for `join`), `input` and `output` list cells, and a `nextLoc` cell holding a natural number indicating the next available location. Unlike in the small languages in the **K** tutorial, where we used the fresh predicate to generate fresh locations, in larger languages, like SIMPLE, we prefer to explicitly manage memory. The location counter in `nextLoc` models an actual physical location in the store; for simplicity, we assume arbitrarily large memory and no garbage collection. The `threads` cell contains one `thread` cell for each existing thread in the program. Note that the thread cell has

multiplicity `*`, which means that at any given moment there could be zero, one or more `thread` cells. Each `thread` cell contains a computation cell `k`, a `control` cell holding the various control structures needed to jump to certain points of interest in the program execution, a local environment map cell `env` mapping the thread local variables to locations in the store, and finally a `holds` map cell indicating what locks have been acquired by the thread and not released so far and how many times (SIMPLE's locks are re-entrant). The `control` cell currently contains only two subcells, a function stack `fstack` which is a list and an exception stack `xstack` which is also a list. One can add more control structures in the `control` cell, such as a stack for break/continue of loops, etc., if the language is extended with more control-changing constructs. Note that all cells except for `k` are also initialized, in that they contain a ground term of their corresponding sort. The `k` cell is initialized with the program that will be passed to the **K** tool, as indicated by the `$PGM` variable, followed by the `execute` task (defined shortly).

```
// the syntax declarations below are required because the sorts are
// referenced directly by a production and, because of the way KIL to KORE
// is implemented, the configuration syntax is not available yet
// should simply work once KIL is removed completely
// check other definitions for this hack as well

syntax ControlCell
syntax ControlCellFragment

configuration <T color="red">
                <threads color="orange">
                  <thread multiplicity="*" color="yellow">
                    <k color="green"> $PGM:Stmt ~> execute </k>
                    //<br/> // TODO(KORE): support latex annotations #1799
                    <control color="cyan">
                      <fstack color="blue"> .List </fstack>
                      <xstack color="purple"> .List </xstack>
                    </control>
                    //<br/> // TODO(KORE): support latex annotations #1799
                    <env color="violet"> .Map </env>
                    <holds color="black"> .Map </holds>
                    <id color="pink"> 0 </id>
                  </thread>
                </threads>
              //<br/> // TODO(KORE): support latex annotations #1799
                <genv color="pink"> .Map </genv>
                <store color="white"> .Map </store>
                <busy color="cyan"> .Set </busy>
```

```
  <terminated color="red"> .Set </terminated>
//<br/> // TODO(KORE): support latex annotations #1799
  <input color="magenta" stream="stdin"> .List </input>
  <output color="brown" stream="stdout"> .List </output>
  <nextLoc color="gray"> 0 </nextLoc>
</T>
```

## Declarations and Initialization

We start by defining the semantics of declarations (for variables, arrays and functions).

## Variable Declaration

The SIMPLE syntax was desugared above so that each variable is declared alone and its initialization is done as a separate statement. The semantic rule below matches resulting variable declarations of the form `var X;` on top of the `k` cell (indeed, note that the `k` cell is complete, or round, to the left, and is torn, or ruptured, to the right), allocates a fresh location `L` in the store which is initialized with a special value (indeed, the unit `.`, or nothing, is matched anywhere in the map note the tears at both sides and replaced with the mapping `L` ), and binds `X` to `L` in the local environment shadowing previous declarations of `X`, if any. This possible shadowing of `X` requires us to therefore update the entire environment map, which is expensive and can significantly slow down the execution of larger programs. On the other hand, since we know that `L` is not already bound in the store, we simply add the binding `L` to the store, thus avoiding a potentially complete traversal of the the store map in order to update it. We prefer the approach used for updating the store whenever possible, because, in addition to being faster, it offers more true concurrency than the latter; indeed, according to the concurrent semantics of `K`, the store is not frozen while `L` is added to it, while the environment is frozen during the update operation `Env[L/X]`. The variable declaration command is also removed from the top of the computation cell and the fresh location counter is incremented. The undefined symbol added in the store is of sort `KItem`, instead of `Val`, on purpose; this way, the store lookup rules will get stuck when one attempts to lookup an uninitialized location. All the above happen in one transactional step, with the rule below. Note also how configuration abstraction allows us to only mention the needed cells; indeed, as the configuration above states, the `k` and `env` cells are actually located within a `thread` cell within the `threads` cell, but one needs not mention these: the configuration context of the rule is

automatically transformed to match the declared configuration structure.

```
syntax KItem ::= "undefined"  [latex(\bot)]

rule <k> var X:Id; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

## Array Declaration

The **K** semantics of the uni-dimensional array declaration is somehow similar to the above declaration of ordinary variables. First, note the context declaration below, which requests the evaluation of the array dimension. Once evaluated, say to a natural number `N`, then `N +Int 1` locations are allocated in the store for an array of size `N`, the additional location (chosen to be the first one allocated) holding the array reference value. The array reference value `array(L,N)` states that the array has size `N` and its elements are located contiguously in the store starting with location `L`. The operation `L … L'  V`, defined at the end of this file in the auxiliary operation section, initializes each location in the list `L … L'` to `V`. Note that, since the dimensions of array declarations can be arbitrary expressions, this virtually means that we can dynamically allocate memory in SIMPLE by means of array declarations.

```
context var _:Id[HOLE];

rule <k> var X:Id[N:Int]; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> array(L +Int 1, N)
                        (L +Int 1) ... (L +Int N) |-> undefined ...</store>
     <nextLoc> L => L +Int 1 +Int N </nextLoc>
  requires N >=Int 0
```

SIMPLE allows multi-dimensional arrays. For semantic simplicity, we desugar them all into uni-dimensional arrays by code transformation. This way, we only need to give semantics to uni-dimensional arrays. First, note that the context rule above actually evaluates all the array dimensions (that's why we defined the expression lists strict!): Upon evaluating the array dimensions, the code generation rule below desugars multi-dimensional array declaration to uni-dimensional declarations. To this aim, we introduce two special unique variable identifiers, `$1` and `$2`. The first variable, `$1`, iterates through and initializes each element of the first dimension with an array

of the remaining dimensions, declared as variable `$2`:

```
syntax Id ::= "$1" | "$2"
rule var X:Id[N1:Int, N2:Int, Vs:Vals];
  => var X[N1];
     {
       for(var $1 = 0; $1 <= N1 - 1; ++$1) {
         var $2[N2, Vs];
         X[$1] = $2;
       }
     }
   [structural]
```

Ideally, one would like to perform syntactic desugarings like the one above before the actual semantics. Unfortunately, that was not possible in this case because the dimension expressions of the multi-dimensional array need to be evaluated first. Indeed, the desugaring rule above does not work if the dimensions of the declared array are arbitrary expressions, because they can have side effects (e.g., `a[++x,++x]`) and those side effects would be propagated each time the expression is evaluated in the desugaring code (note that both the loop condition and the nested multi-dimensional declaration would need to evaluate the expressions given as array dimensions).

## Function declaration

Functions are evaluated to  -abstractions and stored like any other values in the store. A binding is added into the environment for the function name to the location holding its body. Similarly to the C language, SIMPLE only allows function declarations at the top level of the program. More precisely, the subsequent semantics of SIMPLE only works well when one respects this requirement. Indeed, the simplistic context-free parser generated by the grammar above is more generous than we may want, in that it
allows function declarations anywhere any declaration is allowed, including inside arbitrary blocks. However, as the rule below shows, we are `not` storing the declaration environment with the  -abstraction value as closures do. Instead, as seen shortly, we switch to the global environment whenever functions are invoked, which is consistent with our requirement that functions should only be declared at the top. Thus, if one declares local functions, then one may see unexpected behaviors (e.g., when one shadows a global variable before declaring a local function). The type checker of SIMPLE, also defined in **K** (see `examples/simple/typed/static`), discards programs which do not respect this requirement.

```
rule <k> function F(Xs) S => . ...</k>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L |-> lambda(Xs, S) ...</store>
```

```
      <nextLoc> L => L +Int 1 </nextLoc>
```

When we are done with the first pass (pre-processing), the computation cell `k` contains only the token `execute` (see the configuration declaration above, where the computation item `execute` was placed right after the program in the `k` cell of the initial configuration) and the cell `genv` is empty. In this case, we have to call `main()` and to initialize the global environment by transferring the contents of the local environment into it. We prefer to do it this way, as opposed to processing all the top level declarations directly within the global environment, because we want to avoid duplication of semantics: the syntax of the global declarations is identical to that of their corresponding local declarations, so the semantics of the latter suffices provided that we copy the local environment into the global one once we are done with the pre-processing. We want this separate pre-processing step precisely because we want to create the global environment. All (top-level) functions end up having their names bound in the global environment and, as seen below, they are executed in that same global environment; all these mean, in particular, that the functions "see" each other, allowing for mutual recursion, etc.

```
syntax KItem ::= "execute"
rule <k> execute => main(.Exps); </k>
     <env> Env </env>
     <genv> .Map => Env </genv>  [structural]
```

## Expressions

We next define the **K** semantics of all the expression constructs.

## Variable lookup

When a variable `X` is the first computational task, and `X` is bound to some location `L` in the environment, and `L` is mapped to some value `V` in the store, then we rewrite `X` into `V`:

```
rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]
```

Note that the rule above excludes reading , because  is not a value and `V` is checked at runtime to be a value.

## Variable/Array increment

This is tricky, because we want to allow both `++x` and `++a[5]`. Therefore, we need to extract the lvalue of the expression to increment. To do that, we state that the expression to increment should be wrapped by the auxiliary `lvalue` operation and then evaluated. The semantics

of this auxiliary operation is defined at the end of this file. For now, all we need to know is that it takes an expression and evaluates to a location value. Location values, also defined at the end of the file, are integers wrapped with the operation `loc`, to distinguish them from ordinary integers.

```
context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I => I +Int 1) ...</store>  [increment]
```

## Arithmetic operators

There is nothing special about the following rules. They rewrite the language constructs to their library counterparts when their arguments become values of expected sorts:

```
rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 requires I2 =/=K 0
rule I1 % I2 => I1 %Int I2 requires I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2
```

The equality and inequality constructs reduce to syntactic comparison of the two argument values (which is what the equality on `K` terms does).

```
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
```

The logical negation is clear, but the logical conjunction and disjunction are short-circuited:

```
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E
```

## Array lookup

Untyped SIMPLE does not check array bounds (the dynamically typed version of
it, in `examples/simple/typed/dynamic`, does check for array out of bounds). The first rule below desugars the multi-dimensional array access to

uni-dimensional array access; recall that the array access operation was declared strict, so all sub-expressions involved are already values at this stage. The second rule rewrites the array access to a lookup operation at a precise location; we prefer to do it this way to avoid locking the store. The semantics of the auxiliary `lookup` operation is straightforward, and is defined at the end of the file.

```
// The [anywhere] feature is underused, because it would only be used
// at the top of the computation or inside the lvalue wrapper. So it
// may not be worth, or we may need to come up with a special notation
// allowing us to enumerate contexts for [anywhere] rules.
  rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
    [structural, anywhere]

  rule array(L,_)[N:Int] => lookup(L +Int N)
    [structural, anywhere]
```

## Size of an array

The size of the array is stored in the array reference value, and the `sizeOf` construct was declared strict, so:

```
  rule sizeOf(array(_,N)) => N
```

## Function call

Function application was strict in both its arguments, so we can assume that both the function and its arguments are evaluated to values (the former expected to be a  -abstraction). The first rule below matches a well-formed function application on top of the computation and performs the following steps atomically: it switches to the function body followed by `return;` (for the case in which the function does not use an explicit return statement); it pushes the remaining computation, the current environment, and the current control data onto the function stack (the remaining computation can thus also be discarded from the computation cell, because an unavoidable subsequent `return` statement  see above  will always recover it from the stack); it switches the current environment (which is being pushed on the function stack) to the global environment, which is where the free variables in the function body should be looked up; it binds the formal parameters to fresh locations in the new environment, and stores the actual arguments to those locations in the store (this latter step is easily done by reducing the problem to variable declarations, whose semantics we have already defined; the auxiliary operation `mkDecls` is defined at the end of the file). The second rule pops the computation, the environment and the control data from the function

stack when a `return` statement is encountered as the next computational task, passing the returned value to the popped computation (the popped computation was the context in which the returning function was called). Note that the pushing/popping of the control data is crucial. Without it, one may have a function that contains an exception block with a return statement inside, which would put the `xstack` cell in an inconsistent state (since the exception block modifies it, but that modification should be irrelevant once the function returns). We add an artificial `nothing` value to the language, which is returned by the nulary `return;` statements.

```
syntax KItem ::=  (Map,K,ControlCellFragment)

rule <k> lambda(Xs,S)(Vs:Vals) ~> K => mkDecls(Xs,Vs) S return; </k>
     <control>
       <fstack> .List => ListItem((Env,K,C)) ...</fstack>
       C
     </control>
     <env> Env => GEnv </env>
     <genv> GEnv </genv>

rule <k> return(V:Val); ~> _ => V ~> K </k>
     <control>
       <fstack> ListItem((Env,K,C)) => .List ...</fstack>
       (_ => C)
     </control>
     <env> _ => Env </env>

syntax Val ::= "nothing"
rule return; => return nothing;   [macro]
```

Like for division-by-zero, it is left unspecified what happens when the `nothing` value is used in domain calculations. For example, from the the perspective of the language semantics, `7 +Int nothing` can evaluate to anything, or may not evaluate at all (be undefined). If one wants to make sure that such artificial values are never misused, then one needs to define a static checker (also using **K**, like our the type checker in `examples/simple/typed/static`) and reject programs that do. Note that, unlike the undefined symbol  which had the sort `K` instead of `Val`, we defined `nothing` to be a value. That is because, as explained above, we do not want the program to get stuck when nothing is returned by a function. Instead, we want the behavior to be unspecified; in particular, if one is careful to never use the returned value in domain computation, like it happens when we call a function for its side effects (e.g., with a statement of the

form `f(x);`), then the program does not get stuck.

## Read

The `read()` expression construct simply evaluates to the next input value, at the same time discarding the input value from the `in` cell.

```
rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]
```

## Assignment

In SIMPLE, like in C, assignments are expression constructs and not statement constructs. To make it a statement all one needs to do is to follow it by a semi-colon `;` (see the semantics for expression statements below).
Like for the increment, we want to allow assignments not only to variables but also to array elements, e.g., `e1[e2] = e3` where `e1` evaluates to an array reference, `e2` to a natural number, and `e3` to any value. Thus, we first compute the lvalue of the left-hand-side expression that appears in an assignment, and then we do the actual assignment to the resulting location:

```
context (HOLE => lvalue(HOLE)) = _

rule <k> loc(L) = V:Val => V ...</k> <store>... L |-> (_ => V) ...</store>
  [assignment]
```

## Statements

We next define the **K** semantics of statements.

## Blocks

Empty blocks are simply discarded, as shown in the first rule below.
For non-empty blocks, we schedule the enclosed statement but we have to make sure the environment is recovered after the enclosed statement executes. Recall that we allow local variable declarations, whose scope is the block enclosing them. That is the reason for which we have to recover the environment after the block. This allows us to have a very simple semantics for variable declarations, as we did above. One can make the two rules below computational if one wants them to count as computational steps.

```
rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]
```

The basic definition of environment recovery is straightforward and given in the section on auxiliary constructs at the end of the file.

There are two common alternatives to the above semantics of blocks. One is to keep track of the variables which are declared in the block and only recover those at the end of the block. This way one does more work for variable declarations but conceptually less work for environment recovery; we say `conceptually` because it is not clear that it is indeed the case that one does less work when AC matching is involved. The other alternative is to work with a stack of environments instead of a flat environment, and push the current environment when entering a block and pop it when exiting it. This way, one does more work when accessing variables (since one has to search the variable in the environment stack in a top-down manner), but on the other hand uses smaller environments and the definition gets closer to an implementation. Based on experience with dozens of language semantics and other **K** definitions, we have found that our approach above is the best trade-off between elegance and efficiency (especially since rewrite engines have built-in techniques to lazily copy terms, by need, thus not creating unnecessary copies), so it is the one that we follow in general.

## Sequential composition

Sequential composition is desugared into **K**'s builtin sequentialization operation (recall that, like in C, the semi-colon `;` is not a statement separator in SIMPLE — it is either a statement terminator or a construct for a statement from an expression). The rule below is structural, so it does not count as a computational step. One can make it computational if one wants it to count as a step. Note that **K** allows to define the semantics of SIMPLE in such a way that statements eventually dissolve from the top of the computation when they are completed; this is in sharp contrast to (artificially) `evaluating` them to a special `skip` statement value and then getting rid of that special value, as it is the case in other semantic approaches (where everything must evaluate to something). This means that once `S` completes in the rule below, `S` becomes automatically the next computation item without any additional (explicit or implicit) rules.

```
rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
```

A subtle aspect of the rule above is that `S` is declared to have sort `Stmts` and not `Stmt`. That is because desugaring macros can indeed produce left associative sequential composition of statements. For example, the code `var x=0; x=1;` is desugared to `(var x; x=0;) x=1;`, so although originally the first term of the sequential composition had sort `Stmt`, after desugaring it became of sort `Stmts`. Note that the attribute `[right]` associated to the sequential compositon production is an attribute of the syntax, and not of the semantics: e.g., it tells the parser to parse `var x; x=0; x=1;` as `var x; (x=0; x=1;)`, but it does not tell the rewrite engine to rewrite `(var x; x=0;) x=1;` to

```
var x; (x=0; x=1;).
```

## Expression statements

Expression statements are only used for their side effects, so their result value is simply discarded. Common examples of expression statements are ones of the form `++x;`, `x=e;`, `e1[e2]=e3;`, etc.

```
rule _:Val; => .
```

## Conditional

Since the conditional was declared with the `strict(1)` attribute, we can assume that its first argument will eventually be evaluated. The rules below cover the only two possibilities in which the conditional is allowed to proceed (otherwise the rewriting process gets stuck).

```
rule if ( true) S else _ => S
rule if (false) _ else S => S
```

## While loop

The simplest way to give the semantics of the while loop is by unrolling. Note, however, that its unrolling is only allowed when the while loop reaches the top of the computation (to avoid non-termination of unrolling). We prefer the rule below to be structural, because we don't want the unrolling of the while loop to count as a computational step; this is unavoidable in conventional semantics, but it is possible in **K** thanks to its distinction between structural and computational rules. The simple while loop semantics below works because our while loops in SIMPLE are indeed very basic. If we allowed break/continue of loops then we would need a completely different semantics, which would also involve the `control` cell.

```
rule while (E) S => if (E) {S while(E)S}  [structural]
```

## Print

The `print` statement was strict, so all its arguments are now evaluated (recall that `print` is variadic). We append each of its evaluated arguments to the output buffer, and discard the residual `print` statement with an empty list of arguments.

```
rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  [print]
rule print(.Vals); => .  [structural]
```

## Exceptions

SIMPLE allows parametric exceptions, in that one can throw and catch a particular value. The statement `try S catch(X) S` proceeds with the evaluation of `S` . If `S` evaluates normally, i.e., without any exception thrown, then `S` is discarded and the execution continues normally. If `S` throws an exception with a statement of the form `throw E`, then `E` is first evaluated to some value `V` (`throw` was declared to be strict), then `V` is bound to `X`, then `S` is evaluated in the new environment while the reminder of `S` is discarded, then the environment is recovered and the execution continues normally with the statement following the `try S catch(X) S` statement. Exceptions can be nested and the statements in the `catch` part (`S` in our case) can throw exceptions to the upper level. One should be careful with how one handles the control data structures here, so that the abrupt changes of control due to exception throwing and to function returns interact correctly with each other. For example, we want to allow function calls inside the statement `S` in a `try S catch(X) S` block which can throw an exception that is not caught by the function but instead is propagated to the `try S catch(X) S` block that called the function. Therefore, we have to make sure that the function stack as well as other potential control structures are also properly modified when the exception is thrown to correctly recover the execution context. This can be easily achieved by pushing/popping the entire current control context onto the exception stack. The three rules below modularly do precisely the above.

```
syntax KItem ::= (Id,Stmt,K,Map,ControlCellFragment)

syntax KItem ::= "popx"

rule <k> (try S1 catch(X) {S2} => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem((X, S2, K, Env, C)) ...</xstack>
       C
     </control>
     <env> Env </env>

rule <k> popx => . ...</k>
     <xstack> ListItem(_) => .List ...</xstack>

rule <k> throw V:Val; ~> _ => { var X = V; S2 } ~> K </k>
     <control>
       <xstack> ListItem((X, S2, K, Env, C)) => .List ...</xstack>
       (_ => C)
     </control>
```

```
<env> _ => Env </env>
```

The catch statement `S` needs to be executed in the original environment, but where the thrown value `V` is bound to the catch variable `X`. We here chose to rely on two previously defined constructs when giving semantics to the catch part of the statement: (1) the variable declaration with initialization, for binding `X` to `V`; and (2) the block construct for preventing `X` from shadowing variables in the original environment upon the completion of `S` .

## Threads

SIMPLE's threads can be created and terminated dynamically, and can synchronize by acquiring and releasing re-entrant locks and by rendezvous. We discuss the seven rules giving the semantics of these operations below.

## Thread creation

Threads can be created by any other threads using the `spawn S` construct. The spawn expression construct evaluates to the unique identifier of the newly created thread and, at the same time, a new thread cell is added into the configuration, initialized with the `S` statement and sharing the same environment with the parent thread. Note that the newly created `thread` cell is torn. That means that the remaining cells are added and initialized automatically as described in the definition of SIMPLE's configuration. This is part of **K**'s configuration abstraction mechanism.

```
rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
            ...</thread>)
```

## Thread termination

Dually to the above, when a thread terminates its assigned computation (the contents of its `k` cell) is empty, so the thread can be dissolved.
However, since no discipline is imposed on how locks are acquired and released, it can be the case that a terminating thread still holds locks. Those locks must be released, so other threads attempting to acquire them do not deadlock. We achieve that by removing all the locks held by the terminating thread in its `holds` cell from the set of busy locks in the `busy` cell
(`keys(H)` returns the domain of the map `H` as a set, that is, only

the locks themselves ignoring their multiplicity). As seen below, a lock is added to the `busy` cell as soon as it is acquired for the first time by a thread. The unique identifier of the terminated thread is also collected into the `terminated` cell, so the `join` construct knows which threads have terminated.

```
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

## Thread joining

Thread joining is now straightforward: all we need to do is to check whether the identifier of the thread to be joined is in the `terminated` cell. If yes, then the `join` statement dissolves and the joining thread continues normally; if not, then the joining thread gets stuck.

```
rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>
```

## Acquire lock

There are two cases to distinguish when a thread attempts to acquire a lock (in SIMPLE any value can be used as a lock):
(1) The thread does not currently have the lock, in which case it has to take it provided that the lock is not already taken by another thread (see the side condition of the first rule).
(2) The thread already has the lock, in which case it just increments its counter for the lock (the locks are re-entrant). These two cases are captured by the two rules below:

```
rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  requires (notBool(V in Busy))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N => N +Int 1) ...</holds>
```

## Release lock

Similarly, there are two corresponding cases to distinguish when a thread releases a lock:
(1) The thread holds the lock more than once, in which case all it needs to do is to decrement the lock counter.
(2) The thread holds the lock only once, in which case it needs to remove it from its `holds` cell and also from the the shared `busy` cell, so other threads can acquire it if they need to.

```
rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N -Int 1) ...</holds>
  requires N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>
```

## Rendezvous synchronization

In addition to synchronization through acquire and release of locks, SIMPLE also provides a construct for rendezvous synchronization. A thread whose next statement to execute is `rendezvous(V)` gets stuck until another thread reaches an identical statement; when that happens, the two threads drop their rendezvous statements and continue their executions. If three threads happen to have an identical rendezvous statement as their next statement, then precisely two of them will synchronize and the other will remain blocked until another thread reaches a similar rendezvous statement. The rule below is as simple as it can be. Note, however, that, again, it is **K**'s mechanism for configuration abstraction that makes it work as desired: since the only cell which can multiply containing a `k` cell inside is the `thread` cell, the only way to concretize the rule below to the actual configuration of SIMPLE is to include each `k` cell in a `thread` cell.

```
rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

## Auxiliary declarations and operations

In this section we define all the auxiliary constructs used in the above semantics.

## Making declarations

The `mkDecls` auxiliary construct turns a list of identifiers and a list of values in a sequence of corresponding variable declarations.

```
syntax Stmt ::= mkDecls(Ids,Vals)  [function]
rule mkDecls((X:Id, Xs:Ids), (V:Val, Vs:Vals)) => var X=V; mkDecls(Xs,Vs)
rule mkDecls(.Ids,.Vals) => {}
```

## Location lookup

The operation below is straightforward. Note that we tag it with the same `lookup` tag as the variable lookup rule defined above. This way,

both rules will be considered transitions when we include the `lookup` tag in the transition option of `kompile`.

```
syntax Exp ::= lookup(Int)
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]
```

## Environment recovery

We have already discussed the environment recovery auxiliary operation in the IMP++ tutorial:

```
// TODO: eliminate the env wrapper, like we did in IMP++

  syntax KItem ::= setEnv(Map)
  rule <k> setEnv(Env) => . ...</k> <env> _ => Env </env>  [structural]
```

While theoretically sufficient, the basic definition for environment recovery alone is suboptimal. Consider a loop `while (E)S`, whose semantics (see above) was given by unrolling. `S` is a block. Then the semantics of blocks above, together with the unrolling semantics of the while loop, will yield a computation structure in the `k` cell that increasingly grows, adding a new environment recovery task right in front of the already existing sequence of similar environment recovery tasks (this phenomenon is similar to the "tail recursion" problem). Of course, when we have a sequence of environment recovery tasks, we only need to keep the last one. The elegant rule below does precisely that, thus avoiding the unnecessary computation explosion problem:

```
rule (setEnv(_) => .) ~> setEnv(_)  [structural]
```

In fact, the above follows a common convention in **K** for recovery operations of cell contents: the meaning of a computation task of the form `cell(C)` that reaches the top of the computation is that the current contents of cell `cell` is discarded and gets replaced with `C`. We did not add support for these special computation tasks in our current implementation of **K**, so we need to define them as above.

## lvalue and loc

For convenience in giving the semantics of constructs like the increment and the assignment, that we want to operate the same way on variables and on array elements, we used an auxiliary `lvalue(E)` construct which was expected to evaluate to the lvalue of the expression `E`. This is only defined when `E` has an lvalue, that is, when `E` is either a variable or evaluates to an array element. `lvalue(E)` evaluates to a value of the form `loc(L)`, where `L` is the location where the value of `E` can be found; for clarity, we use `loc` to structurally distinguish

natural numbers from location values. In giving semantics to `lvalue` there are two cases to consider. (1) If `E` is a variable, then all we need to do is to grab its location from the environment. (2) If `E` is an array element, then we first evaluate the array and its index in order to identify the exact location of the element of concern, and then return that location; the last rule below works because its preceding context declarations ensure that the array and its index are evaluated, and then the rule for array lookup (defined above) rewrites the evaluated array access construct to its corresponding store lookup operation.

```
// For parsing reasons, we prefer to allow lvalue to take a K

  syntax Exp ::= lvalue(K)
  syntax Val ::= loc(Int)

// Local variable

  rule <k> lvalue(X:Id => loc(L)) ...</k> <env>... X |-> L:Int ...</env>
    [structural]

// Array element: evaluate the array and its index;
// then the array lookup rule above applies.

  context lvalue(_::Exp[HOLE::Exps])
  context lvalue(HOLE::Exp[_::Exps])

// Finally, return the address of the desired object member

  rule lvalue(lookup(L:Int) => loc(L))  [structural]
```

## Initializing multiple locations

The following operation initializes a sequence of locations with the same value:

```
  syntax Map ::= Int "..." Int "|->" K
    [function, latex({#1}\ldots{#2}\mapsto{#3})]
  rule N...M |-> _ => .Map  requires N >Int M
  rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  requires N <=Int M
```

The semantics of SIMPLE is now complete. Make sure you kompile the definition with the right options in order to generate the desired model. No kompile options are needed if you only only want to execute the definition (and thus get an interpreter), but if you want to search for a different program behaviors then you need to kompile with the transition option including rule tags such as lookup, increment, acquire, etc. See the IMP++ tutorial for what the transition option means how to use it.

`endmodule`

Go to Lesson 2, SIMPLE typed static

# SIMPLE — Typed — Static

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** definition of the static semantics of the typed SIMPLE language, or in other words, a type system for the typed SIMPLE language in **K**. We do not re-discuss the various features of the SIMPLE language here. The reader is referred to the untyped version of the language for such discussions. We here only focus on the new and interesting problems raised by the addition of type declarations, and what it takes to devise a type system/checker for the language.

When designing a type system for a language, no matter in what paradigm, we have to decide upon the intended typing policy. Note that we can have multiple type systems for the same language, one for each typing policy. For example, should we accept programs which don't have a main function? Or should we allow functions that do not return explicitly? Or should we allow functions whose type expects them to return a value (say an `int`) to use a plain `return;` statement, which returns no value, like in C? And so on and so forth. Typically, there are two opposite tensions when designing a type system. On the one hand, you want your type system to be as permissive as possible, that is, to accept as many programs that do not get stuck when executed with the untyped semantics as possible; this will keep the programmers using your language happy. On the other hand, you want your type system to have a reasonable performance when implemented; this will keep both the programmers and the implementers of your language happy. For example, a type system for rejecting programs that could perform division-by-zero is not expected to be feasible in general. A simple guideline when designing typing policies is to imagine how the semantics of the untyped language may get stuck and try to prevent those situations from happening.

Before we give the **K** type system of SIMPLE formally, we discuss, informally, the intended typing policy:

- Each program should contain a `main()` function. Indeed, the untyped SIMPLE semantics will get stuck on any program which does not have a `main` function.
- Each primitive value has its own type, which can be `int` `bool`, or `string`. There is also a type `void` for nonexistent values, for example for the result of a function meant to return no value (but only be used for its side effects, like a procedure).
- The syntax of untyped SIMPLE is extended to allow type declarations for all the variables, including array variables. This is done in a C/Java-style. For example, `int x;` or `int x=7, y=x+3;`, or `int[][][] a[10,20];` (the latter defines a `10 × 20` matrix of arrays of integers). Recall from untyped SIMPLE that, unlike in C/Java, our multi-dimensional arrays use comma-separated arguments, although they have the array-of-array semantics.
- Functions are also typed in a C/Java style. However, since in SIMPLE we allow functions to be passed to and returned by other functions, we also need function types. We will use the conventional higher-order arrow-notation for function types, but will separate the argument types with commas. For example, a function returning an array of `bool` elements and taking as argument an array `x` of two-integer-argument functions returning an integer, is declared using a syntax of the form `bool[] f(((int,int)->int)[] x) { ... }` and has the type `((int,int)->int)[] -> bool[]`.
- We allow any variable declarations at the top level. Functions can only be declared at the top level. Each function can only access the other functions and variables declared at the top level, or its own locally declared variables. SIMPLE has static scoping.
- The various expression and statement constructs take only elements of the expected types.
- Increment and assignment can operate both on variables and on array elements. For example, if `f` has type `int->int[][]` and function `g` has the type `int->int`, then the increment expression `++f(7)[g(2),g(3)]` is valid.
- Functions should only return values of their declared result type. To give the programmers more flexibility, we allow functions to

use `return;` statements to terminate without returning an actual value, or to not explicitly use any return statement, regardless of their declared return type. This flexibility can be handy when writing programs using certain functions only for their side effects. Nevertheless, as the dynamic semantics shows, a return value is automatically generated when an explicit `return` statement is not encountered.

- For simplicity, we here limit exceptions to only throw and catch integer values. We let it as an exercise to the reader to extend the semantics to allow throwing and catching arbitrary-type exceptions. Like in programming languages like Java, one can go even further and define a semantics where thrown exceptions are propagated through try-catch statements until one of the corresponding type is found. We will do this when we define the KOOL language, not here. To keep the definition if SIMPLE simple, here we do not attempt to reject programs which throw uncaught exceptions.

Like in untyped SIMPLE, some constructs can be desugared into a smaller set of basic constructs. In general, it should be clear why a program does not type by looking at the top of the `k` cells in its stuck configuration.

```
module SIMPLE-TYPED-STATIC-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of typed SIMPLE extends that of untyped SIMPLE with support for declaring types to variables and functions.

```
  syntax Id ::= "main" [token]
```

## Types

Primitive, array and function types, as well as lists (or tuples) of types. The lists of types are useful for function arguments.

```
  syntax Type ::= "void" | "int" | "bool" | "string"
                | Type "[" "]"
                | "(" Type ")"              [bracket]
                > Types "->" Type

  syntax Types ::= List{Type,","}
```

## Declarations

Variable and function declarations have the expected syntax. For variables, we basically just replaced the `var` keyword of untyped SIMPLE with a

type. For functions, besides replacing the `function` keyword with a type, we also introduce a new syntactic category for typed variables, `Param`, and lists over it.

```
syntax Param ::= Type Id
syntax Params ::= List{Param,","}

syntax Stmt ::= Type Exps ";"
              | Type Id "(" Params ")" Block
```

## Expressions

The syntax of expressions is identical to that in untyped SIMPLE, except for the logical conjunction and disjunction which have different strictness attributes, because they now have different evaluation strategies.

```
syntax Exp ::= Int | Bool | String | Id
             | "(" Exp ")"             [bracket]
             | "++" Exp
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict]
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
             | Exp "<=" Exp            [strict, non-assoc]
             | Exp ">" Exp             [strict, non-assoc]
             | Exp ">=" Exp            [strict, non-assoc]
             | Exp "==" Exp            [strict, non-assoc]
             | Exp "!=" Exp            [strict, non-assoc]
             > "!" Exp                 [strict]
             > left:
               Exp "&&" Exp            [strict, left]
             | Exp "||" Exp            [strict, left]
             > "spawn" Block
             > Exp "=" Exp             [strict(2), right]
```

Note that `spawn` has not been declared strict. This may

seem unexpected, because the child thread shares the same environment with the parent thread, so from a typing perspective the spawned statement makes the same sense in a child thread as it makes in the parent thread. The reason for not declaring it strict is because we want to disallow programs where the spawned thread calls the `return` statement, because those programs would get stuck in the dynamic semantics. The type semantics of spawn below will reject such programs.

We still need lists of expressions, defined below, but note that we do not need lists of identifiers anymore. They have been replaced by the lists of parameters.

```
syntax Exps ::= List{Exp,","}          [strict]
```

## Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameter. Note that, unlike in untyped SIMPLE, all statement constructs which have arguments and are not desugared are strict, including the conditional and the `while`. Indeed, from a typing perspective, they are all strict: first type their arguments and then type the actual construct.

```
syntax Block ::= "{" "}"
               | "{" Stmt "}"

syntax Stmt ::= Block
              | Exp ";"                                    [strict]
              | "if" "(" Exp ")" Block "else" Block        [avoid, strict]
              | "if" "(" Exp ")" Block
              | "while" "(" Exp ")" Block                  [strict]
              | "for" "(" Stmt Exp ";" Exp ")" Block
              | "return" Exp ";"                           [strict]
              | "return" ";"
              | "print" "(" Exps ")" ";"                   [strict]
              | "try" Block "catch" "(" Param ")" Block    [strict(1)]
              | "throw" Exp ";"                            [strict]
              | "join" Exp ";"                             [strict]
              | "acquire" Exp ";"                          [strict]
              | "release" Exp ";"                          [strict]
              | "rendezvous" Exp ";"                       [strict]
```

Note that the sequential composition is now sequentially strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to the `stmt` type, which is a result.

```
syntax Stmt ::= Stmt Stmt                              [seqstrict, right]
```

## Desugaring macros

We use the same desugaring macros like in untyped SIMPLE, but, of course, including the types of the involved variables.

```
  rule if (E) S => if (E) S else {}                                          [macro]
  rule for(Start Cond; Step) {S:Stmt} => {Start while(Cond){S Step;}}  [macro]
  rule for(Start Cond; Step) {} => {Start while(Cond){Step;}}            [macro]
  rule T:Type E1:Exp, E2:Exp, Es:Exps; => T E1; T E2, Es;                [macro-rec]
  rule T:Type X:Id = E; => T X; X = E;                                       [macro]

endmodule


module SIMPLE-TYPED-STATIC
  imports SIMPLE-TYPED-STATIC-SYNTAX
  imports DOMAINS
```

## Static semantics

Here we define the type system of SIMPLE. Like concrete semantics, type systems defined in **K** are also executable. However, **K** type systems turn into type checkers instead of interpreters when executed.

The typing process is done in two (overlapping) phases. In the first phase the global environment is built, which contains type bindings for all the globally declared variables and functions. For functions, the declared types will be "trusted" during the first phase and simply bound to their corresponding function names and placed in the global type environment. At the same time, type-checking tasks that the function bodies indeed respect their claimed types are generated. All these tasks are (concurrently) verified during the second phase. This way, all the global variable and function declarations are available in the global type environment and can be used in order to type-check each function code. This is consistent with the semantics of untyped SIMPLE, where functions can access all the global variables and can call any other function declared in the same program. The two phases may overlap because of the **K** concurrent semantics. For example, a function task can be started while the first phase is still running; moreover, it may even complete before the first phase does, namely when all the global variables and functions that it needs have already been processed and made available in the global environment by the first phase task.

## Extended syntax and results

The idea is to start with a configuration holding the program to type in one of its cells, then apply rewrite rules on it mixing types and language syntax, and eventually obtain a type instead of the original program. In other words, the program reduces to its type using the **K** rules giving the type system of the language. In doing so, additional typing tasks for function bodies are generated and solved the same way. If this rewriting process gets stuck, then we say that the program is not well-typed. Otherwise the program is well-typed (by definition). We did not need types for statements and for blocks as part of the typed SIMPLE syntax, because programmers are not allowed to use such types explicitly. However, we are going to need them in the type system, because blocks and statements reduce to them.

We start by allowing types to be used inside expressions and statements in our language. This way, types can be used together with language syntax in subsequent **K** rules without any parsing errors. Like in the type system of IMP++ in the **K** tutorial, we prefer to group the block and statement types under one syntactic sub-category of types, because this allows us to more compactly state that certain terms can be either blocks or statements. Also, since programs and fragments of program will reduce to their types, in order for the strictness and context declarations to be executable we state that types are results (same like we did in the IMP++ tutorial).

```
syntax Exp ::= Type
syntax Exps ::= Types
syntax BlockOrStmtType ::= "block" | "stmt"
syntax Type ::= BlockOrStmtType
syntax Block ::= BlockOrStmtType
syntax KResult ::= Type
                 | Types    //TODO: remove this, eventually
```

## Configuration

The configuration of our type system consists of a `tasks` cell holding various typing `task` cells, and a global type environment. Each task includes a `k` cell holding the code to type, a `tenv` cell holding the local type environment, and a `return` cell holding the return type of the currently checked function. The latter is needed in order to check whether return statements return values of the expected type. Initially, the program is placed in a `k` cell inside a `task` cell. Since the cells with multiplicity `?` are not included in the initial configuration, the `task` cell holding the original program in its `k` cell will contain no other subcells.

```
  configuration <T color="yellow">
                  <tasks color="orange">
                    <task multiplicity="*" color="yellow">
                      <k color="green"> $PGM:Stmt </k>
                      <tenv multiplicity="?" color="cyan"> .Map </tenv>
                      <returnType multiplicity="?" color="black"> void </returnType>
                    </task>
                  </tasks>
//                  <br/>
                  <gtenv color="blue"> .Map </gtenv>
                </T>
```

## Variable declarations

Variable declarations type as statements, that is, they reduce to the type `stmt`. There are only two cases that need to be considered: when a simple variable is declared and when an array variable is declared. The macros at the end of the syntax module above take care of reducing other variable declarations, including ones where the declared variables are initialized, to only these two cases. The first case has two subcases: when the variable declaration is global (i.e., the `task` cell contains only the `k` cell), in which case it is added to the global type environment checking at the same time that the variable has not been already declared; and when the variable declaration is local (i.e., a `tenv` cell is available), in which case it is simply added to the local type environment, possibly shadowing previous homonymous variables. The third case reduces to the second, incrementally moving the array dimension into the type until the array becomes a simple variable.

```
  rule <task> <k> T:Type X:Id; => stmt ...</k> </task>
       <gtenv> Rho (.Map => X |-> T) </gtenv>
    requires notBool(X in keys(Rho))
  rule <k> T:Type X:Id; => stmt ...</k> <tenv> Rho => Rho[X <- T] </tenv>

  context _:Type _::Exp[HOLE::Exps];
// The rule below may need to sort E to Exp in the future, if the
// parser gets stricter; without that information, it may not be able
// to complete the LHS into T E[int,Ts],.Exps; (and similarly for the RHS)
  rule T:Type E:Exp[int,Ts:Types]; => T[] E[Ts];  [structural]
// I want to write the rule below as _:Type (E:Exp[.Types] => E),
// but the list completion seems to not work well with that.
  rule T:Type E:Exp[.Types]; => T E;          [structural]
```

## Function declarations

Functions are allowed to be declared only at the top level (the `task` cell holds only its `k` subcell). Each function declaration reduces to a variable declaration (a binding of its name to its declared function type), but also adds a task into the `tasks` cell. The task consists of a typing of the statement declaring all the function parameters followed by the function body, together with the expected return type of the function. The `getTypes` and `mkDecls` functions, defined at the end of the file in the section on auxiliary operations, extracts the list of types and makes a sequence of variable declarations from a list of function parameters, respectively. Note that, although in the dynamic semantics we include a terminating `return` statement at the end of the function body to eliminate from the analysis the case when the function does not provide an explicit return, we do not need to include such a similar `return` statement here. That's because the `return` statements type to `stmt` anyway, and the entire code of the function body needs to type anyway.

```
rule <task> <k> T:Type F:Id(Ps:Params) S => getTypes(Ps)->T F; ...</k> </task>
     (.Bag => <task>
                <k> mkDecls(Ps) S </k> <tenv> .Map </tenv> <returnType> T </returnType>
              </task>)
  [structural]
```

## Checking if `main()` exists}

Once the entire program is processed (generating appropriate tasks to type check its function bodies), we can dissolve the main `task` cell (the one holding only a `k` subcell). Since we want to enforce that programs include a main function, we also generate a function task executing `main()` to ensure that it types (remove this task creation if you do not want your type system to reject programs without a `main` function).

```
rule <task> <k> stmt => main(.Exps); </k> (.Bag => <tenv> .Map </tenv>) </task>
  [structural]
```

## Collecting the terminated tasks

Similarly, once a non-main task (i.e., one which contains a `tenv` subcells) is completed using the subsequent rules (i.e., its `k` cell holds only the `block` or `stmt` type), we can dissolve its corresponding cell. Note that it is important to ensure that we only dissolve tasks containing a `tenv` cell with the rule below, because the main task should

not dissolve this way! It should do what the above rule says. In the end, there should be no task cell left in the configuration when the program correctly type checks.

```
rule <task>... <k> _:BlockOrStmtType </k> <tenv> _ </tenv> ...</task> => .Bag
```

## Basic values

The first three rewrite rules below reduce the primitive values to their types, as we typically do when we define type systems in **K**.

```
rule _:Int => int
rule _:Bool => bool
rule _:String => string
```

## Variable lookup

There are three cases to distinguish for variable lookup: (1) if the variable is bound in the local type environment, then look its type up there; (2) if a local environment exists and the variable is not bound in it, then look its type up in the global environment; (3) finally, if there is no local environment, meaning that we are executing the top-level pass, then look the variable's type up in the global environment, too.

```
rule <k> X:Id => T ...</k> <tenv>... X |-> T ...</tenv>

rule <k> X:Id => T ...</k> <tenv> Rho </tenv> <gtenv>... X |-> T ...</gtenv>
  requires notBool(X in keys(Rho))

rule <task> <k> X:Id => T ...</k> </task> <gtenv>... X |-> T ...</gtenv>
```

## Increment

We want the increment operation to apply to any lvalue, including array elements, not only to variables. For that reason, we define a special context extracting the type of the argument of the increment operation only if that argument is an lvalue. Otherwise the rewriting process gets stuck. The operation `ltype` is defined at the end of this file, in the auxiliary operation section. It essentially acts as a filter, getting stuck if its argument is not an lvalue and letting it reduce otherwise. The type of the lvalue is expected to be an integer in order to be allowed to be incremented, as seen in the rule `++ int => int` below.

```
context ++(HOLE => ltype(HOLE))
rule ++ int => int
```

## Common expression constructs

The rules below are straightforward and self-explanatory:

```
rule int + int => int
rule string + string => string
rule int - int => int
rule int * int => int
rule int / int => int
rule int % int => int
rule - int => int
rule int < int => bool
rule int <= int => bool
rule int > int => bool
rule int >= int => bool
rule T:Type == T => bool
rule T:Type != T => bool
rule bool && bool => bool
rule bool || bool => bool
rule ! bool => bool
```

## Array access and size

Array access requires each index to type to an integer, and the array type to be at least as deep as the number of indexes:

```
// NOTE:
// We used to need parentheses in the RHS, to avoid capturing Ts as rule tag.
// Let's hope that is not a problem anymore.

rule (T[])[int, Ts:Types] => T[Ts]
rule T:Type[.Types] => T
```

`sizeOf` only needs to check that its argument is an array:

```
rule sizeOf(_T[]) => int
```

## Input/Output

The read expression construct types to an integer, while print types to a statement provided that all its arguments type to integers or strings.

```
rule read() => int

rule print(T:Type, Ts => Ts); requires T ==K int orBool T ==K string
rule print(.Types); => stmt
```

## Assignment

The special context and the rule for assignment below are similar to those for increment: the LHS of the assignment must be an lvalue and, in that case, it must have the same type as the RHS, which then becomes the type of the assignment.

```
context (HOLE => ltype(HOLE)) = _
rule T:Type = T => T
```

## Function application and return

Function application requires the type of the function and the types of the passed values to be compatible. Note that a special case is needed to handle the no-argument case:

```
rule (Ts:Types -> T)(Ts) => T requires Ts =/=K .Types
rule (void -> T)(.Types) => T
```

The returned value must have the same type as the declared function return type. If an empty return is encountered, than we should check that we are in a function (and not a thread) context, that is, a `return` cell must be available:

```
rule <k> return T:Type; => stmt ...</k> <returnType> T </returnType>
rule <k> return; => stmt ...</k> <returnType> _ </returnType>
```

## Blocks

To avoid having to recover type environments after blocks, we prefer to start a new task for block body, making sure that the new task is passed the same type environment and return cells. The value returned by `return` statements must have the same type as stated in the `return` cell. The `print` variadic function is allowed to only print integers and strings. The thrown exceptions can only have integer type.

```
rule {} => block

rule <task> <k> {S} => block ...</k> <tenv> Rho </tenv> R </task>
     (.Bag => <task> <k> S </k> <tenv> Rho </tenv> R </task>)
```

## Expression statement

```
rule _:Type; => stmt
```

## Conditional and `while` loop

```
rule if (bool) block else block => stmt
rule while (bool) block => stmt
```

## Exceptions

We currently force the parameters of exceptions to only be integers. Moreover, for simplicity, we assume that integer exceptions can be thrown from anywhere, including from functions which do not define any try-catch block (with the currently unchecked also for simplicity expectation that the caller functions would catch those exceptions).

```
rule try block catch(int X:Id) {S} => {int X; S}  [structural]
rule try block catch(int X:Id) {} => {int X;}  [structural]
rule throw int; => stmt
```

## Concurrency

Nothing special about typing the concurrency constructs, except that we do not want the spawned thread to return, so we do not include any `return` cell in the new task cell for the thread statement.
Same like with the functions above, we do not check for thrown exceptions which are not caught.

```
rule <k> spawn S => int ...</k> <tenv> Rho </tenv>
     (.Bag => <task> <k> S </k> <tenv> Rho </tenv> </task>)
rule join int; => stmt
rule acquire _:Type; => stmt
rule release _:Type; => stmt
rule rendezvous _:Type; => stmt

rule _:BlockOrStmtType _:BlockOrStmtType => stmt
```

## Auxiliary constructs

The function `mkDecls` turns a list of parameters into a list of variable declarations.

```
syntax Stmt ::= mkDecls(Params)  [function]
rule mkDecls(T:Type X:Id, Ps:Params) => T X; mkDecls(Ps)
rule mkDecls(.Params) => {}
```

The `ltype` context allows only expressions which have an lvalue to evaluate.

```
syntax LValue ::= Id
rule isLValue(_:Exp[_:Exps]) => true
```

```
  syntax Exp ::= LValue  // K should be able to infer this
                         // if not added, then it gets stuck with an Id on k cell

// Instead of the second LValue production above you can use a rule:
//  rule isLValue(_:Exp[_:Exps]) => true

  syntax Exp ::= ltype(Exp)
//  context ltype(HOLE:LValue)
// The above context does not work due to some error, so we write instead
  context ltype(HOLE) requires isLValue(HOLE)
```

The function `getTypes` is the same as in SIMPLE typed dynamic.

```
  syntax Types ::= getTypes(Params)  [function]
  rule getTypes(T:Type _:Id) => T, .Types   // I would like to not use .Types
  rule getTypes(T:Type _:Id, P, Ps) => T, getTypes(P,Ps)
  rule getTypes(.Params) => void, .Types

endmodule
```

Go to Lesson 3, SIMPLE typed dynamic

# SIMPLE — Typed — Dynamic

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** dynamic semantics of the typed SIMPLE language. It is very similar to the semantics of the untyped SIMPLE, the difference being that we now dynamically check the typing policy described in the static semantics of typed SIMPLE. Because of the dynamic nature of the semantics, we can also perform some additional checks which were not possible in the static semantics, such as memory leaks due to accessing an array out of its bounds. We will highlight the differences between the dynamically typed and the untyped SIMPLE as we proceed with the semantics. We recommend the reader to consult the typing policy and the syntax of types discussed in the static semantics of the typed SIMPLE language.

```
module SIMPLE-TYPED-DYNAMIC-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of typed SIMPLE extends that of untyped SIMPLE with support for declaring types to variables and functions.

The syntax below is identical to that of the static semantics of typed SIMPLE. However, the **K** strictness attributes are like those of the untyped SIMPLE, to capture the desired evaluation strategies of the various language constructs.

```
syntax Id ::= "main" [token]
```

## Types

```
syntax Type ::= "void" | "int" | "bool" | "string"
              | Type "[" "]"
              | "(" Type ")"           [bracket]
              > Types "->" Type
syntax Types ::= List{Type,","}
```

## Declarations

```
syntax Param ::= Type Id
syntax Params ::= List{Param,","}

syntax Stmt ::= Type Exps ";"
              | Type Id "(" Params ")" Block
```

## Expressions

```
syntax Exp ::= Int | Bool | String | Id
             | "(" Exp ")"             [bracket]
             | "++" Exp
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict]
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
```

```
    | Exp "<=" Exp            [strict, non-assoc]
    | Exp ">" Exp             [strict, non-assoc]
    | Exp ">=" Exp            [strict, non-assoc]
    | Exp "==" Exp            [strict, non-assoc]
    | Exp "!=" Exp            [strict, non-assoc]
    > "!" Exp                 [strict]
    > left:
      Exp "&&" Exp            [strict(1), left]
    | Exp "||" Exp            [strict(1), left]
    > "spawn" Block
    > Exp "=" Exp             [strict(2), right]
```

Like in the static semantics, there is no need for lists of identifiers (because we now have lists of parameters).

```
syntax Exps ::= List{Exp,","}          [strict]
syntax Val
syntax Vals ::= List{Val,","}
```

## Statements

```
syntax Block ::= "{" "}"
               | "{" Stmt "}"

syntax Stmt ::= Block
               | Exp ";"                                  [strict]
               | "if" "(" Exp ")" Block "else" Block      [avoid, strict(1)]
               | "if" "(" Exp ")" Block
               | "while" "(" Exp ")" Block
           | "for" "(" Stmt Exp ";" Exp ")" Block
               | "print" "(" Exps ")" ";"                 [strict]
               | "return" Exp ";"                         [strict]
               | "return" ";"
               | "try" Block "catch" "(" Param ")" Block
           | "throw" Exp ";"                             [strict]
               | "join" Exp ";"                           [strict]
               | "acquire" Exp ";"                        [strict]
               | "release" Exp ";"                        [strict]
               | "rendezvous" Exp ";"                     [strict]

syntax Stmt ::= Stmt Stmt                                  [right]
```

The same desugaring macros like in the statically typed SIMPLE.

```
rule if (E) S => if (E) S else {}                                      [macro]
rule for(Start Cond; Step) {S:Stmt} => {Start while(Cond){S Step;}}    [macro]
rule for(Start Cond; Step) {} => {Start while(Cond){Step;}}            [macro]
rule T:Type E1:Exp, E2:Exp, Es:Exps; => T E1; T E2, Es;                [macro-rec]
```

```
  rule T:Type X:Id = E; => T X; X = E;                                    [macro]

endmodule


module SIMPLE-TYPED-DYNAMIC
  imports SIMPLE-TYPED-DYNAMIC-SYNTAX
  imports DOMAINS
```

## Semantics

### Values and results

These are similar to those of untyped SIMPLE, except that the array references and the function abstrations now also hold their types. These types are needed in order to easily compute the type of any value in the language (see the auxiliary `typeOf` operation at the end of this module).

```
  syntax Val ::= Int | Bool | String
               | array(Type,Int,Int)
               | lambda(Type,Params,Stmt)
  syntax Exp ::= Val
  syntax Exps ::= Vals
  syntax KResult ::= Val
                   | Vals  // TODO: should not need this
```

### Configuration

The configuration is almost identical to that of untyped SIMPLE, except for a `return` cell inside the `control` cell. This `return` cell will hold, like in the static semantics of typed SIMPLE, the expected type of the value returned by the function being executed. The contents of this cell will be set whenever a function is invoked and will be checked whenever the evaluation of the function body encounters an explicit `return` statement.

```
  // the syntax declarations below are required because the sorts are
  // referenced directly by a production and, because of the way KIL to KORE
  // is implemented, the configuration syntax is not available yet
  // should simply work once KIL is removed completely
  // check other definitions for this hack as well

  syntax ControlCell
  syntax ControlCellFragment

  configuration <T color="red">
```

```
                  <threads color="orange">
                    <thread multiplicity="*" color="yellow">
                      <k color="green"> ($PGM:Stmt ~> execute) </k>
//                      <br/>
                      <control color="cyan">
                        <fstack color="blue"> .List </fstack>
                        <xstack color="purple"> .List </xstack>
                        <returnType color="LimeGreen"> void </returnType>
                       </control>
//                      <br/>
                      <env color="violet"> .Map </env>
                      <holds color="black"> .Map </holds>
                      <id color="pink"> 0 </id>
                    </thread>
                  </threads>
//                  <br/>
                  <genv color="pink"> .Map </genv>
                  <store color="white"> .Map </store>
                  <busy color="cyan">.Set</busy>
                  <terminated color="red"> .Set </terminated>
                  <input color="magenta" stream="stdin"> .List </input>
                  <output color="brown" stream="stdout"> .List </output>
                  <nextLoc color="gray"> 0 </nextLoc>
                </T>
```

## Declarations and Initialization

### Variable Declaration

The `undefined` construct is now parameterized by a type.
A main difference between untyped SIMPLE and dynamically typed SIMPLE is that the latter assigns a type to each of its locations and that type cannot be changed during the execution of the program. We do not do any memory management in our semantic definitions here, so locations cannot be reclaimed, garbage collected and/or reused. Each location corresponds precisely to an allocated variable or array element, whose type was explicitly or implicitly declared in the program and does not change. It is therefore safe to type each location and then never allow that type to change. The typed undefined values effectively assign both a type and an undefined value to a location.

```
syntax KItem ::= undefined(Type)  [latex(\bot_{#1})]

rule <k> T:Type X:Id; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined(T) ...</store>
```

```
      <nextLoc> L:Int => L +Int 1 </nextLoc>
```

### Array Declaration

The dynamic semantics of typed array declarations is similar to that in untyped SIMPLE, but we have to make sure that we associate the right type to the allocated locations.

```
  rule <k> T:Type X:Id[N:Int]; => . ...</k>
       <env> Env => Env[X <- L] </env>
       <store>... .Map => L |-> array(T, L +Int 1, N)
                          (L +Int 1)...(L +Int N) |-> undefined(T) ...</store>
       <nextLoc> L:Int => L +Int 1 +Int N </nextLoc>
    when N >=Int 0

  context _:Type _::Exp[HOLE::Exps];
```

The desugaring of multi-dimensional arrays into unidimensional ones is also similar to that in untyped SIMPLE, although we have to make sure that all the declared variables have the right types. The auxiliary operation `T<Vs>`, defined at the end of the file, adds the length of `Vs` dimensions to the type `T`.

```
// TODO: Check the desugaring below to be consistent with the one for untyped simple

  syntax Id ::= "$1" | "$2"
  rule T:Type X:Id[N1:Int, N2:Int, Vs:Vals];
    => T[]<Vs> X[N1];
       {
         T[][]<Vs> $1=X;
         for(int $2=0; $2 <= N1 - 1; ++$2) {
           T X[N2,Vs];
           $1[$2] = X;
         }
       }
    [structural]
```

### Function declaration

Store all function parameters, as well as the return type, as part of the lambda abstraction. In the spirit of dynamic typing, we will make sure that parameters are well typed when the function is invoked.

```
  rule <k> T:Type F:Id(Ps:Params) S => . ...</k>
       <env> Env => Env[F <- L] </env>
       <store>... .Map => L |-> lambda(T, Ps, S) ...</store>
       <nextLoc> L => L +Int 1 </nextLoc>
```

### Calling `main()`

When done with the first pass, call `main()`.

```
syntax KItem ::= "execute"
rule <k> execute => main(.Exps); </k>
     <env> Env </env>
     <genv> .Map => Env </genv>  [structural]
```

### Expressions

### Variable lookup

```
rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]
```

### Variable/Array increment

```
context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I:Int => I +Int 1) ...</store>  [increment]
```

### Arithmetic operators

```
rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E
```

### Array lookup

Check array bounds, as part of the dynamic typing policy.

```
// Same comment as for simple untyped regarding [anywhere]
  rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
    [structural, anywhere]

// Same comment as for simple untyped regarding [anywhere]
  rule array(_:Type, L:Int, M:Int)[N:Int] => lookup(L +Int N)
    when N >=Int 0 andBool N <Int M  [structural, anywhere]
```

### Size of an array

```
  rule sizeOf(array(_,_,N)) => N
```

### Function call

Define function call and return together, to see their relationship. Note that the operation `mkDecls` now declares properly typed instantiated variables, and that the semantics of `return` also checks that that type of the returned value is expected one.

```
  syntax KItem ::= (Type,Map,K,ControlCellFragment)

  rule <k> lambda(T,Ps,S)(Vs:Vals) ~> K => mkDecls(Ps,Vs) S return; </k>
       <control>
         <fstack> .List => ListItem((T',Env,K,C)) ...</fstack>
         <returnType> T' => T </returnType>
         C
       </control>
       <env> Env => GEnv </env>
       <genv> GEnv </genv>

  rule <k> return V:Val; ~> _ => V ~> K </k>
       <control>
         <fstack> ListItem((T',Env,K,C)) => .List ...</fstack>
         <returnType> T => T' </returnType>
         (_ => C)
       </control>
       <env> _ => Env </env>
    when typeOf(V) ==K T   // check the type of the returned value
```

Like the `undefined` above, `nothing` also gets tagged with a type now. The empty `return` statement is completed to return the `nothing` value tagged as expected.

```
  syntax Val ::= nothing(Type)
  rule <k> return; => return nothing(T); ...</k> <returnType> T </returnType>
    [structural]
```

### Read

```
rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]
```

### Assignment

The assignment now checks that the type of the assigned location is preserved:

```
context (HOLE => lvalue(HOLE)) = _

rule <k> loc(L) = V:Val => V ...</k> <store>... L |-> (V' => V) ...</store>
  when typeOf(V) ==K typeOf(V')  [assignment]
```

### Statements

### Blocks

```
rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]
```

### Sequential composition

```
rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
```

### Expression statements

```
rule _:Val; => .
```

### Conditional

```
rule if ( true) S else _ => S
rule if (false) _ else S => S
```

### While loop

```
rule while (E) S => if (E) {S while(E)S}  [structural]
```

### Print

We only allow printing integers and strings:

```
rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  when typeOf(V) ==K int orBool typeOf(V) ==K string  [print]
rule print(.Vals); => .  [structural]
```

### Exceptions

Exception parameters are now typed, but note that the semantics below works correctly only when the thrown exception has the same type as the innermost try-catch paramete. To keep things simple, for the time being we can assume that SIMPLE only throws and catches integer values, in which case our semantics below works fine:

```
syntax KItem ::= (Param,Stmt,K,Map,ControlCellFragment)  // Param instead of Id

syntax KItem ::= "popx"

rule <k> (try S1 catch(P) S2 => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem((P, S2, K, Env, C)) ...</xstack>
       C
     </control>
     <env> Env </env>

rule <k> popx => . ...</k>
     <xstack> ListItem(_) => .List ...</xstack>

rule <k> throw V:Val; ~> _ => { T X = V; S2 } ~> K </k>
     <control>
       <xstack> ListItem((T:Type X:Id, S2, K, Env, C)) => .List ...</xstack>
       (_ => C)
     </control>
     <env> _ => Env </env>
```

### Threads

### Thread creation

```
rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
            ...</thread>)
```

### Thread termination

```
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

**Thread joining**

```
rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>
```

**Acquire lock**

```
rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  when (notBool(V in Busy:Set))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N:Int => N +Int 1) ...</holds>
```

**Release lock**

```
rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N:Int -Int 1) ...</holds>
   when N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>
```

**Rendezvous synchronization**

```
rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

**Auxiliary declarations and operations**

Turns a list of parameters and a list of instance values for them
into a list of variable declarations.

```
syntax Stmt ::= mkDecls(Params,Vals)  [function]
rule mkDecls((T:Type X:Id, Ps:Params), (V:Val, Vs:Vals))
  => T X=V; mkDecls(Ps,Vs)
rule mkDecls(.Params,.Vals) => {}
```

Location lookup.

```
syntax Exp ::= lookup(Int)  // see NOTES.md for why Exp instead of KItem
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]
```

Environment recovery.

```
// TODO: same comment regarding setEnv(...) as for simple untyped

syntax KItem ::= setEnv(Map)
```

```
  rule <k> setEnv(Env) => . ...</k>  <env> _ => Env </env>  [structural]
  rule (setEnv(_) => .) ~> setEnv(_)  [structural]
```

lvalue and loc

```
  syntax Exp ::= lvalue(K)
  syntax Val ::= loc(Int)

  rule <k> lvalue(X:Id => loc(L)) ...</k>  <env>... X |-> L:Int ...</env>
    [structural]

  //context lvalue(_[HOLE])
  //context lvalue(HOLE[_])
  context lvalue(_::Exp[HOLE::Exps])
  context lvalue(HOLE::Exp[_::Exps])

  rule lvalue(lookup(L:Int) => loc(L))  [structural]
```

Adds the corresponding depth to an array type

```
  syntax Type ::= Type "<" Vals ">"  [function]
  rule T:Type<_,Vs:Vals> => T[]<Vs>
  rule T:Type<.Vals> => T
```

Sequences of locations.

```
  syntax Map ::= Int "..." Int "|->" K
    [function, latex({#1}\ldots{#2}\mapsto{#3})]
  rule N...M |-> _ => .Map  when N >Int M
  rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  when N <=Int M

// Type of a value.
  syntax Type ::= typeOf(K)  [function]
  rule typeOf(_:Int) => int
  rule typeOf(_:Bool) => bool
  rule typeOf(_:String) => string
  rule typeOf(array(T,_,_)) => (T[])   // () needed! K parses [] as "no tags"
  rule typeOf(lambda(T,Ps,_)) => getTypes(Ps) -> T
  rule typeOf(undefined(T)) => T
  rule typeOf(nothing(T)) => T
```

List of types of a parameter.

```
  syntax Types ::= getTypes(Params)  [function]
  rule getTypes(T:Type _:Id) => T, .Types   // I would like to not use .Types
  rule getTypes(T:Type _:Id, P, Ps) => T, getTypes(P,Ps)
  rule getTypes(.Params) => void, .Types
endmodule
```

# KOOL — Untyped

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped KOOL language. KOOL is aimed at being a pedagogical and research language that captures the essence of the object-oriented programming paradigm. Its untyped variant discussed here is simpler than the typed one, ignoring several intricate aspects of types in the presence of objects. A program consists of a set of class declarations. Each class can extend at most one other class (KOOL is single-inheritance). A class can declare a set of fields and a set of methods, all public and called the class' *members*. Specifically, KOOL includes the following features:

- Class declarations, where a class may or may not explicitly extend another class. In case a class does not explicitly extend another class, then it is assumed that it extends the default top-most and empty (i.e., no members) class called `Object`. Each class is required to declare precisely one homonymous method, called its *constructor*. Each valid program should contain one class named `Main`, whose constructor, `Main()`, takes no arguments. The execution of a program consists of creating an object instance of class `Main` and invoking the constructor `Main()` on it, that is, of executing `new Main();`.
- All features of SIMPLE (see `examples/simple/untyped`), i.e., multidimensional arrays, function (here called "method") abstractions with call-by-value parameter passing style and static scoping, blocks with locals, input/output, parametric exceptions, and concurrency via dynamic thread creation/termination and synchronization.
  The only change in the syntax of SIMPLE when imported in KOOL is the function declaration keyword, `function`, which is changed into `method`. The exact same desugaring macros from SIMPLE are also included in KOOL. We can think of KOOL's classes as embedding SIMPLE programs (extended with OO constructs, as discussed next).
- Object creation using the `new C(e1,...,en)` expression construct. An object instance of class `C` is first created and then the constructor `C(e1,...,en)` is implicitly called on that object. KOOL only allows (and requires) one

constructor per class. The class constructor can be called either implicitly during a new object creation for the class, or explicitly. The superclass constructor is **not** implicitly invoked when a class constructor is invoked; if you want to invoke the superclass constructor from a subclass constructor then you have to do it explicitly.

- An expression construct `this`, which evaluates to the current object.
- An expression construct `super`, which is used (only) in combination with member lookup (see next) to refer to a superclass field or method.
- A member lookup expression construct `e.x`, where `e` is an expression (either an expression expected to evaluate to an object or the `super` construct) and `x` is a class member name, that is, a field or a method name.
- Expression constructs `e instanceOf C` and `(C) e`, where `e` is an expression expected to evaluate to an object and `C` a class name. The former tells whether the class of `e` is a subclass of `C`, that is, whether `e` can be used as an instance of `C`, and the latter changes the class of `e` to `C`. These operations always succeed: the former returns a Boolean value, while the latter changes the current class of `e` to `C` regardless of whether it is safe to do so or not. The typed version of KOOL will check the safety of casting by ensuring that the instance class of the object is a subclass of `C`. In untyped KOOL we do not want to perform this check because we want to allow the programmer maximum of flexibility: if one always accesses only available members, then the program can execute successfully despite the potentially unsafe cast.

There are some specific aspects of KOOL that need to be discussed.

First, KOOL is higher-order, allowing function abstractions to be treated like any other values in the language. For example, if `m` is a method of object `e` then `e.m` evaluates to the corresponding function abstraction. The function abstraction is in fact a closure, because in addition to the method parameters and body it also encapsulates the object value (i.e., the environment of the object together with its current class—see below) that `e` evaluates to. This way, function abstractions can be invoked anywhere and have the capability to change the state of their object. For example, if `m` is a method of object `e` which increments a field `c` of `e` when invoked, and if `getm` is another method of `e` which simply returns

`m` when invoked, then the double application `(e.getm())()` has the same effect as `e.m()`, that is, increments the counter `c` of `e`. Note that the higher-order nature of KOOL was not originally planned; it came as a natural consequence of evaluating methods to closures and we decided to keep it. If you do not like it then do not use it.

Second, since all the fields and methods are public in KOOL and since they can be redeclared in subclasses, it is not immediately clear how to lookup the member `x` when we write `e.x` and `e` is different from `super`. We distinguish two cases, depending on whether `e.x` occurs in a method invocation context (i.e., `e.x(...)`) or in a field context. KOOL has dynamic method dispatch, so if `e.x` is invoked as a method then `x` will be searched for starting with the instance class of the object value to which `e` evaluates. If `e.x` occurs in a non-method-invocation context then `x` will be treated as a field (although it may hold a method closure due to the higher-order nature of KOOL) and thus will be searched starting with the current class of the object value of `e` (which, because of `this` and casting, may be different from its instance class). In order to achieve the above, each object value will consist of a pair holding the current class of the object and an environment stack with one layer for each class in the object's instance class hierarchy.

Third, although KOOL is dynamic method dispatch, its capabilities described above are powerful enough to allow us to mimic static method dispatch. For example, suppose that you want to invoke method `m()` statically. Then all you need to do is to declare a local variable and bind it to `m`, for example `var staticm = m;`, and then call `staticm()`. This works because `staticm` is first bound to the method closure that `m` evaluates to, and then looked up as any local variable when invoked. We only enable the dynamic method dispatch when we have an object member on an application position, e.g., `m()`.

In what follows, we limit our comments to the new, KOOL-specific aspects of the language. We refer the reader to the untyped SIMPLE language for documentation on the the remaining features, because those were all borrowed from SIMPLE.

```
module KOOL-UNTYPED-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of KOOL extends that of SIMPLE with object-oriented constructs. We removed from the **K** annotated syntax of SIMPLE two

constructs, namely the one for function declarations (because we want to call them `methods` now) and the one for function application (because application is not strict in the first argument anymore—needs to initiate dynamic method dispatch). The additional syntax includes:

- First, we need a new dedicated identifier, `Object`, for the default top-most class.
- Second, we rename the `function` keyword of SIMPLE into `method`.
- Third, we add syntax for class declarations together with a macro making classes which extend nothing to extend `Object`.
- Fourth, we change the strictness attribute of application into `strict(2)`.
- Finally, we add syntax and corresponding strictness for the KOOL object-oriented constructs.

```
syntax Id ::= "Object" [token] | "Main" [token]

syntax Stmt ::= "var" Exps ";"
              | "method" Id "(" Ids ")" Block  // called "function" in SIMPLE
              | "class" Id Block               // KOOL
              | "class" Id "extends" Id Block  // KOOL

syntax Exp ::= Int | Bool | String | Id
             | "this"                                  // KOOL
             | "super"                                 // KOOL
             | "(" Exp ")"             [bracket]
             | "++" Exp
             | Exp "instanceOf" Id     [strict(1)]     // KOOL
             | "(" Id ")" Exp          [strict(2)]     // KOOL  cast
             | "new" Id "(" Exps ")"   [strict(2)]     // KOOL
             | Exp "." Id                              // KOOL
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict(2)]     // was strict in SIMPLE
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
             | Exp "<=" Exp            [strict, non-assoc]
```

```
              | Exp ">" Exp             [strict, non-assoc]
              | Exp ">=" Exp            [strict, non-assoc]
              | Exp "==" Exp            [strict, non-assoc]
              | Exp "!=" Exp            [strict, non-assoc]
              > "!" Exp                 [strict]
              > left:
                Exp "&&" Exp            [strict(1), left]
              | Exp "||" Exp            [strict(1), left]
              > "spawn" Block
              > Exp "=" Exp             [strict(2), right]

  syntax Ids  ::= List{Id,","}

  syntax Exps ::= List{Exp,","}         [strict, klabel(exps)]
  syntax Val
  syntax Vals ::= List{Val,","}         [klabel(exps)]

  syntax Block ::= "{" "}"
               | "{" Stmt "}"

  syntax Stmt ::= Block
               | Exp ";"                              [strict]
               | "if" "(" Exp ")" Block "else" Block   [avoid, strict(1)]
               | "if" "(" Exp ")" Block
               | "while" "(" Exp ")" Block
           | "for" "(" Stmt Exp ";" Exp ")" Block
               | "return" Exp ";"                     [strict]
               | "return" ";"
               | "print" "(" Exps ")" ";"             [strict]
               | "try" Block "catch" "(" Id ")" Block
               | "throw" Exp ";"                      [strict]
               | "join" Exp ";"                       [strict]
               | "acquire" Exp ";"                    [strict]
               | "release" Exp ";"                    [strict]
               | "rendezvous" Exp ";"                 [strict]

  syntax Stmt ::= Stmt Stmt                         [right]
```

Old desugaring rules, from SIMPLE

```
  rule if (E) S => if (E) S else {}                                  [macro]
  rule for(Start Cond; Step) {S} => {Start while (Cond) {S Step;}}   [macro]
  rule var E1::Exp, E2::Exp, Es::Exps; => var E1; var E2, Es;        [macro-rec]
  rule var X::Id = E; => var X; X = E;                               [macro]
```

New desugaring rule

```
  rule class C:Id S => class C extends Object S                      // KOOL
```

```
endmodule
```

## Semantics

We first discuss the new configuration of KOOL, which extends that of SIMPLE. Then we include the semantics of the constructs borrowed from SIMPLE unchanged; we refrain from discussing those, because they were already discussed in the **K** definition of SIMPLE. Then we discuss changes to SIMPLE's semantics needed for the more general meaning of the previous SIMPLE constructs (for example for thread spawning, assignment, etc.). Finally, we discuss in detail the semantics of the additional KOOL constructs.

```
module KOOL-UNTYPED
  imports KOOL-UNTYPED-SYNTAX
  imports DOMAINS
```

## Configuration

KOOL removes one cell and adds two nested cells to the configuration of SIMPLE. The cell which is removed is the one holding the global environment, because a KOOL program consists of a set of classes only, with no global declarations. In fact, since informally speaking each KOOL class now includes a SIMPLE program, it is safe to say that the global variables in SIMPLE became class fields in KOOL. Let us now discuss the new cells that are added to the configuration of SIMPLE.

- The cell `crntObj` holds data pertaining to the current object, that is, the object environment in which the code in cell `k` executes: `crntClass` holds the current class (which can change as methods of the current object are invoked); `envStack` holds the stack of environments as a list, each layer corresponding to one class in the objects' instance class hierarchy; `location`, which is optional, holds the location in the store where the current object is or has to be located (this is useful both for method closures and for the semantics of object creation).
- The cell `classes` holds all the declared classes, each class being held in its own `class` cell which contains a name (`className`), a parent (`extends`), and the actual member declarations (`declarations`).

```
  // the syntax declarations below are required because the sorts are
  // referenced directly by a production and, because of the way KIL to KORE
  // is implemented, the configuration syntax is not available yet
  // should simply work once KIL is removed completely
```

```
// check other definitions for this hack as well
syntax EnvCell
syntax ControlCell
syntax EnvStackCell
syntax CrntObjCellFragment

configuration <T color="red">
                <threads color="orange">
                  <thread multiplicity="*" type="Set" color="yellow">
                    <k color="green"> $PGM:Stmt ~> execute </k>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <control color="cyan">
                      <fstack color="blue"> .List </fstack>
                      <xstack color="purple"> .List </xstack>
                    //<br/> // TODO(KORE): support latex annotations #1799
                      <crntObj color="Fuchsia">  // KOOL
                         <crntClass> Object </crntClass>
                         <envStack> .List </envStack>
                         <location multiplicity="?"> .K </location>
                      </crntObj>
                    </control>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <env color="violet"> .Map </env>
                    <holds color="black"> .Map </holds>
                    <id color="pink"> 0 </id>
                  </thread>
                </threads>
              //<br/> // TODO(KORE): support latex annotations #1799
                <store color="white"> .Map </store>
                <busy color="cyan">.Set </busy>
                <terminated color="red"> .Set </terminated>
                <input color="magenta" stream="stdin"> .List </input>
                <output color="brown" stream="stdout"> .List </output>
                <nextLoc color="gray"> 0 </nextLoc>
              //<br/> // TODO(KORE): support latex annotations #1799
                <classes color="Fuchsia">        // KOOL
                   <classData multiplicity="*" type="Map" color="Fuchsia">
                      // the Map has as its key the first child of the cell,
                      // in this case the className cell.
                      <className color="Fuchsia"> Main </className>
                      <baseClass color="Fuchsia"> Object </baseClass>
                      <declarations color="Fuchsia"> .K </declarations>
                   </classData>
                </classes>
              </T>
```

## Unchanged Semantics from untyped SIMPLE

The semantics below is taken over from SIMPLE unchanged.
The semantics of function declaration and invocation, including the use of the special `lambda` abstraction value, needs to change in order to account for the fact that methods are now invoked into their object's environment. The semantics of function return actually stays unchanged. Also, the semantics of program initialization is different: now we have to create an instance of the `Main` class which also calls the constructor `Main()`, while in SIMPLE we only had to invoke the function `Main()`.
Finally, the semantics of thread spawning needs to change, too: the parent thread needs to also share its object environment with the spawned thread (in addition to its local environment, like in SIMPLE). This is needed in order to be able to spawn method invokations under dynamic method dispatch; for example, `spawn { run(); }` will need to look up the method `run()` in the newly created thread, operation which will most likely fail unless the child thread sees the object environment of the parent thread. Note that the `spawn` statement of KOOL is more permissive than the threads of Java. In fact, the latter can be implemented in terms of our `spawn`—see the program `threads.kool` for a sketch.

Below is a subset of the values of SIMPLE, which are also values of KOOL. We will add other values later in the semantics, such as object and method closures.

```
syntax Val ::= Int | Bool | String
             | array(Int,Int)
syntax Exp ::= Val
syntax Exps ::= Vals
syntax KResult ::= Val
syntax KResult ::= Vals
```

The semantics below are taken verbatim from the untyped SIMPLE definition.

```
syntax KItem ::= "undefined"  [latex(\bot)]

rule <k> var X:Id; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined ...</store>
     <nextLoc> L:Int => L +Int 1 </nextLoc>


context var _:Id[HOLE];

rule <k> var X:Id[N:Int]; => . ...</k>
```

```
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> array(L +Int 1, N)
                        (L +Int 1) ... (L +Int N) |-> undefined ...</store>
     <nextLoc> L:Int => L +Int 1 +Int N </nextLoc>
  when N >=Int 0


syntax Id ::= "$1" [token] | "$2" [token]
rule var X:Id[N1:Int, N2:Int, Vs:Vals];
  => var X[N1];
     {
       var $1=X;
       for(var $2=0; $2 <= N1 - 1; ++$2) {
         var X[N2,Vs];
         $1[$2] = X;
       }
     }
  [structural]


rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]


context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I:Int => I +Int 1) ...</store>  [increment]


rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2

rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
```

```
rule false && _ => false
rule true  || _ => true
rule false || E => E


rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
  [structural, anywhere]

rule array(L,_)[N:Int] => lookup(L +Int N)
  [structural, anywhere]


rule sizeOf(array(_,N)) => N
```

The semantics of function application needs to change into dynamic method dispatch invocation, which is defined shortly. However, interestingly, the semantics of return stays unchanged.

```
rule <k> return(V:Val); ~> _ => V ~> K </k>
     <control>
       <fstack> ListItem(fstackFrame(Env,K,XS,<crntObj> CO </crntObj>)) => .List ...</fsta
       <xstack> _ => XS </xstack>
       <crntObj> _ => CO </crntObj>
     </control>
     <env> _ => Env </env>

syntax Val ::= "nothing"
rule return; => return nothing;   [macro]


rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]


context (HOLE => lvalue(HOLE)) = _

rule <k> loc(L) = V:Val => V ...</k> <store>... L |-> (_ => V) ...</store>
  [assignment]


rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]


rule S1::Stmt S2::Stmt => S1 ~> S2  [structural]

rule _:Val; => .
```

```
rule if ( true) S else _ => S
rule if (false) _ else S => S

rule while (E) S => if (E) {S while(E)S}  [structural]

rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  [print]
rule print(.Vals); => .  [structural]


syntax KItem ::= xstackFrame(Id,Stmt,K,Map,K)
// TODO(KORE): drop the additional production once parsing issue #1842 is fixed
               | (Id,Stmt,K,Map,K)

syntax KItem ::= "popx"

rule <k> (try S1 catch(X) {S2} => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem(xstackFrame(X, S2, K, Env, C)) ...</xstack>
       C
     </control>
     <env> Env </env>

rule <k> popx => . ...</k>
     <xstack> ListItem(_) => .List ...</xstack>

rule <k> throw V:Val; ~> _ => { var X = V; S2 } ~> K </k>
     <control>
       <xstack> ListItem(xstackFrame(X, S2, K, Env, C)) => .List ...</xstack>
       (_ => C)
     </control>
     <env> _ => Env </env>
```

Thread spawning needs a new semantics, because we want the child thread to also share the object environment with its parent. The new semantics of thread spawning will be defined shortly. However, interestingly, the other concurrency constructs keep their semantics from SIMPLE unchanged.

```
// TODO(KORE): ..Bag should be . throughout this definition #1772
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
/*
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .)
*/
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

```
rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>

rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  when (notBool(V in Busy:Set))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N:Int => N +Int 1) ...</holds>

rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N:Int -Int 1) ...</holds>
  when N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>

rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

## Unchanged auxiliary operations from untyped SIMPLE

```
syntax Stmt ::= mkDecls(Ids,Vals)  [function]
rule mkDecls((X:Id, Xs:Ids), (V:Val, Vs:Vals)) => var X=V; mkDecls(Xs,Vs)
rule mkDecls(.Ids,.Vals) => {}

// TODO(KORE): clarify sort inferences #1803
syntax Exp ::= lookup(Int)
/*
syntax KItem ::= lookup(Int)
*/
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]

syntax KItem ::= setEnv(Map)
rule <k> setEnv(Env) => . ...</k>  <env> _ => Env </env>  [structural]
rule (setEnv(_) => .) ~> setEnv(_)  [structural]
// TODO: How can we make sure that the second rule above applies before the first one?
//       Probably we'll deal with this using strategies, eventually.

syntax Exp ::= lvalue(K)
syntax Val ::= loc(Int)

rule <k> lvalue(X:Id => loc(L)) ...</k> <env>... X |-> L:Int ...</env>
  [structural]
```

```
context lvalue(_::Exp[HOLE::Exps])
context lvalue(HOLE::Exp[_::Exps])

rule lvalue(lookup(L:Int) => loc(L))  [structural]


syntax Map ::= Int "..." Int "|->" K
  [function, latex({#1}\ldots{#2}\mapsto{#3})]
rule N...M |-> _ => .Map  when N >Int M
rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  when N <=Int M
```

## Changes to the existing untyped SIMPLE semantics

When we extend a language, sometimes we need to do more than just add new language constructs and semantics for them. Sometimes we want to also extend the semantics of existing language constructs, in order to get more from them.

## Program initialization

In SIMPLE, once all the global declarations were processed, the function `main()` was invoked. In KOOL, the global declarations are classes, and their specific semantics is given shortly; essentially, they are pre-processed one by one and added into the `class` cell structure in the configuration.
Once all the classes are processed, the computation item `execute`, which was placed right after the program in the initial configuration, is reached. In SIMPLE, the program was initialized by calling the method `main()`. In KOOL, the program is initialized by creating an object instance of class `Main`. This will also implicitly call the method `Main()` (the `Main` class constructor). The emptiness of the `env` cell below is just a sanity check, to make sure that the user has not declared anything but classes at the top level of the program.

```
syntax KItem ::= "execute"
rule <k> execute => new Main(.Exps); </k> <env> .Map </env>  [structural]
```

The semantics of `new` (defined below) requires the execution of all the class' declarations (and also of its superclasses').

## Object and method closures

Before we can define the semantics of method application (previously called function application in SIMPLE), we need to add two more values

to the language, namely object and method closures:

```
syntax Val ::= objectClosure(Id, List)
             | methodClosure(Id,Int,Ids,Stmt)
```

An object value consists of an `objectClosure`-wrapped bag containing the current class of the object and the environment stack of the object. The current class of an object will always be one of the classes mapped to an environment in the environment stack of the object. A method closure encapsulates the method's parameters and code (last two arguments), as well as the object context in which the method code should execute. This object context includes the current class of the object (the first argument of `methodClosure`) and the object environment stack (located in the object stored at the location specified as the second argument of `methodClosure`).

## Method application

KOOL has a complex mechanism to invoke methods, because it allows both dynamic method dispatch and methods as first-class-citizen values (the latter making it a higher-order language). The invocation mechanism will be defined later. What is sufficient to know for now is that the two arguments of the application construct eventually reduce to values, the first being a method closure and the latter a list of values. The semantics of the method closure application is then as expected: the local environment and control are stacked, then we switch to method closure's class and object environment and execute the method body. The `mkDecls` construct is the one that came with the unchanged semantics of SIMPLE above.

```
syntax KItem ::= fstackFrame(Map,K,List,K)
// TODO(KORE): drop the additional production once parsing issue #1842 is fixed
               | (Map,K,K)

rule <k> methodClosure(Class,OL,Xs,S)(Vs:Vals) ~> K
         => mkDecls(Xs,Vs) S return; </k>
     <env> Env => .Map </env>
     <store>... OL |-> objectClosure(_, EnvStack)...</store>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control>
        <xstack> XS </xstack>
        <fstack> .List => ListItem(fstackFrame(Env, K, XS, <crntObj> Obj' </crntObj>))
        ...</fstack>
        <crntObj> Obj' => <crntClass> Class </crntClass> <envStack> EnvStack </envStack> <
     </control>
```

### Spawn

We want to extend the semantics of `spawn` to also share the current object environment with the child thread, in addition to the current environment. This extension will allow us to also use method invocations in the spawned statements, which will be thus looked up as expected, using dynamic method dispatch. This lookup operation would fail if the child thread did not have access to its parent's object environment.

```
rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
       <crntObj> Obj </crntObj>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
              <crntObj> Obj </crntObj>
            ...</thread>)
```

## Semantics of the new KOOL constructs

### Class declaration

Initially, the classes forming the program are moved into their corresponding cells:

```
rule <k> class Class1 extends Class2 { S } => . ...</k>
     <classes>... (.Bag => <classData>
                             <className> Class1 </className>
                             <baseClass> Class2 </baseClass>
                             <declarations> S </declarations>
                           </classData>)
     ...</classes>  [structural]
```

### Method declaration

Like in SIMPLE, method names are added to the environment and bound to their code. However, unlike in SIMPLE where each function was executed in the same environment, namely the program global environment, a method in KOOL needs to be executed into its object's environment. Thus, methods evaluate to closures, which encapsulate their object's context (i.e., the current class and environment stack of the object) in addition to method's parameters and body. This approach to bind method names to method closures in the environment will also allow objects to pass their methods to other objects, to

dynamically change their methods by assigning them other method closures, and even to allow all these to be done from other objects. This gives the KOOL programmer a lot of power; one should use this power wisely, though, because programs can become easily hard to understand and reason about if one overuses these features.

```
rule <k> method F:Id(Xs:Ids) S => . ...</k>
     <crntClass> Class:Id </crntClass>
     <location> OL:Int </location>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L |-> methodClosure(Class,OL,Xs,S) ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

## New

The semantics of `new` consists of two actions: memory allocation for the new object and execution of the corresponding constructor. Then the created object is returned as the result of the `new` operation; the value returned by the constructor, if any, is discarded. The current environment and object are stored onto the stack and recovered after new (according to the semantics of `return` borrowed from SIMPLE, when the statement `return this;` in the rule below is reached and evaluated), because the object creation part of `new` will destroy them. The rule below also initializes the object creation process by emptying the local environment and the current object, and allocating a location in the store where the created object will be eventually stored (this is what the `storeObj` task after the object creation task in the rule below will do—its rule is defined shortly). The location where the object will be stored is also made available in the `crntObj` cell, so that method closures can refer to it (see rule above).

```
syntax KItem ::= "envStackFrame" "(" Id "," Map ")"

rule <k> new Class:Id(Vs:Vals) ~> K
         => create(Class) ~> storeObj ~> Class(Vs); return this; </k>
     <env> Env => .Map </env>
     <nextLoc> L:Int => L +Int 1 </nextLoc>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control> <xstack> XS </xstack>
       <crntObj> Obj
                 => <crntClass> Object </crntClass>
                    <envStack> ListItem(envStackFrame(Object, .Map)) </envStack>
                    <location> L </location>
       </crntObj>
       <fstack> .List => ListItem(fstackFrame(Env, K, XS, <crntObj> Obj </crntObj>)) ...</
```

```
      </control>
```

The creation of a new object (the memory allocation part only) is a recursive process, requiring to first create an object for the superclass. A memory object representation is a layered structure: for each class on the path from the instance class to the root of the hierarchy there is a layer including the memory allocated for the members (both fields and methods) of that class.

```
  syntax KItem ::= create(Id)

  rule <k> create(Class:Id)
           => create(Class1) ~> setCrntClass(Class) ~> S ~> addEnvLayer ...</k>
       <className> Class </className>
       <baseClass> Class1:Id </baseClass>
       <declarations> S </declarations>  [structural]

  rule <k> create(Object) => . ...</k>  [structural]
```

The next operation sets the current class of the current object. This is necessary to be done at each layer, because the current class of the object is enclosed as part of the method closures (see the semantics of method declarations above).

```
  syntax KItem ::= setCrntClass(Id)

  rule <k> setCrntClass(C) => . ...</k>
       <crntClass> _ => C </crntClass>  [structural]
```

The next operation adds a new tagged environment layer to the current object and gets ready for the next layer by clearing the environment (note that `create` expects the environment to be empty).

```
  syntax KItem ::= "addEnvLayer"

  rule <k> addEnvLayer => . ...</k>
       <env> Env => .Map </env>
       <crntClass> Class:Id </crntClass>
       <envStack> .List => ListItem(envStackFrame(Class, Env)) ...</envStack>
    [structural]
```

The following operation stores the created object at the location reserved by `new`. Note that the location reserved by `new` was temporarily stored in the `crntObj` cell precisely for this purpose. Now that the newly created object is stored at its location and that all method closures are aware of it, the location is unnecessary and thus we delete it from the `crntObj` cell.

```
  syntax KItem ::= "storeObj"

  rule <k> storeObj => . ...</k>
       <crntObj> <crntClass> CC </crntClass> <envStack> ES </envStack> (<location> L:Int </l
       <store>... .Map => L |-> objectClosure(CC, ES) ...</store>
```

### Self reference

The semantics of `this` is straightforward: evaluate to the current object.

```
  rule <k> this => objectClosure(CC, ES) ...</k>
       <crntObj> <crntClass> CC </crntClass> <envStack> ES </envStack> </crntObj>
```

### Object member access

We can access an object member (field or method) either explicitly, using the construct `e.x`, or implicitly, using only the member name `x` directly. The borrowed semantics of SIMPLE will already lookup a sole name in the local environment. The first rule below reduces implicit member access to explicit access when the name cannot be found in the local environment. There are two cases to analyze for explicit object member access, depending upon whether the object is a proper object or it is just a redirection to the parent class via the construct `super`. In the first case, we evaluate the object expression and lookup the member starting with the current class (static scoping). Note the use of the conditional evaluation context. In the second case, we just lookup the member starting with the superclass of the current class. In both cases, the `lookupMember` task eventually yields a `lookup(L)` task for some appropriate location `L`, which will be further solved with the corresponding rule borrowed from SIMPLE. Note that the current object is not altered by `super`, so future method invocations see the entire object, as needed for dynamic method dispatch.

```
  rule <k> X:Id => this . X ...</k> <env> Env:Map </env>
    when notBool(X in keys(Env))  [structural]

  context HOLE._::Id when (HOLE =/=K super)

// TODO: explain how Assoc matching has been replaced with two rules here.
// Maybe also improve it a bit.

/*  rule objectClosure(<crntClass> Class:Id </crntClass>
                       <envStack>... envStackFrame(Class,EnvC) EStack </envStack>)
       . X:Id
    => lookupMember(envStackFrame(Class,EnvC) EStack, X)
```

```
    [structural]*/

  rule objectClosure(Class:Id, ListItem(envStackFrame(Class,Env)) EStack)
       . X:Id
    => lookupMember(ListItem(envStackFrame(Class,Env)) EStack, X)
    [structural]
  rule objectClosure(Class:Id, (ListItem(envStackFrame(Class':Id,_)) => .List) _)
       . _X:Id
    when Class =/=K Class'  [structural]

/*  rule <k> super . X => lookupMember(EStack, X) ...</k>
       <crntClass> Class </crntClass>
       <envStack>... envStackFrame(Class,EnvC) EStack </envStack>
    [structural]*/
  rule <k> super . X => lookupMember(EStack, X) ...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> super . _X ...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class':Id,_)) => .List ...</envStack>
    when Class =/=K Class'  [structural]
```

## Method invocation

Unlike in SIMPLE, in KOOL application was declared strict only in its second argument. That is because we want to ensure dynamic method dispatch when the first argument is a method access. As a consequence, we need to consider all the cases of interest for the first argument and to explicitly say what to do in each case. In all cases except for method access in a proper object (i.e., not `super`), we want the same behavior for the first argument as if it was not in a method invocation position. When it is a member access (the third rule below), we look it up starting with the instance class of the corresponding object. This ensures dynamic dispatch for methods; it actually dynamically dispatches field accesses, too, which is correct in KOOL, because one can assign method closures to fields and the field appeared in a method invocation context. The last context declaration below says that method applications or array accesses are also allowed as first argument to applications; that is because methods are allowed to return methods and arrays are allowed to hold methods in KOOL, since it is higher-order. If that is the case, then we want to evaluate the method call or the array access.

```
  rule <k> (X:Id => V)(_:Exps) ...</k>
       <env>... X |-> L ...</env>
       <store>... L |-> V:Val ...</store>  [lookup]

  rule <k> (X:Id => this . X)(_:Exps) ...</k>
       <env> Env </env>
    when notBool(X in keys(Env))  [structural]

  context HOLE._::Id(_) when HOLE =/=K super

  rule (objectClosure(_, EStack) . X
    => lookupMember(EStack, X:Id))(_:Exps)  [structural]

/*  rule <k> (super . X
             => lookupMember(EStack,X))(_:Exps)...</k>
       <crntClass> Class </crntClass>
       <envStack>... envStackFrame(Class,_) EStack </envStack>
    [structural]*/
  rule <k> (super . X
           => lookupMember(EStack,X))(_:Exps)...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> (super . _X)(_:Exps) ...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class':Id,_)) => .List ...</envStack>
    when Class =/=K Class'  [structural]

  // TODO(KORE): fix getKLabel #1801
  rule (A:Exp(B:Exps))(C:Exps) => A(B) ~> #freezerFunCall(C)
  rule (A:Exp[B:Exps])(C:Exps) => A[B] ~> #freezerFunCall(C)
  rule V:Val ~> #freezerFunCall(C:Exps) => V(C)
  syntax KItem ::= "#freezerFunCall" "(" K ")"
  /*
  context HOLE(_:Exps)
    when getKLabel(HOLE) ==K #klabel(`_(_)`) orBool getKLabel(HOLE) ==K #klabel(`_[_]`)
  */
```

Eventually, each of the rules above produces a `lookup(L)` task as a replacement for the method. When that happens, we just lookup the value at location `L`:

```
  rule <k> (lookup(L) => V)(_:Exps) ...</k>  <store>... L |-> V:Val ...</store>
    [lookup]
```

The value `V` looked up above is expected to be a method closure, in which case the semantics of method application given above will

apply. Otherwise, the execution will get stuck.

### Instance Of

It searches the object environment for a layer corresponding to the desired class. It returns `true` iff it can find the class, otherwise it returns `false`; it only gets stuck when its first argument does not evaluate to an object.

```
  rule objectClosure(_, ListItem(envStackFrame(C,_)) _)
       instanceOf C => true

  rule objectClosure(_, (ListItem(envStackFrame(C,_)) => .List) _)
       instanceOf C'  when C =/=K C'  [structural]
//TODO: remove the sort cast ::Id of C above, when sort inference bug fixed

  rule objectClosure(_, .List) instanceOf _ => false
```

### Cast

In untyped KOOL, we prefer to not check the validity of casting. In other words, any cast is allowed on any object, simply changing the current class of the object to the desired class. The execution will get stuck later if one attempts to access a field which is not available. Moreover, the execution may complete successfully even in the presence of invalid casts, provided that each accessed member during the current execution is, or happens to be, available.

```
  rule (C) objectClosure(_ , EnvStack) => objectClosure(C ,EnvStack)
```

## KOOL-specific auxiliary declarations and operations

Here we define all the auxiliary constructs used in the above KOOL-specific semantics (those used in the SIMPLE fragment have already been defined in a corresponding section above).

### Objects as lvalues

The current machinery borrowed with the semantics of SIMPLE allows us to enrich the set of lvalues, this way allowing new means to assign values to locations. In KOOL, we want object member names to be lvalues, so that we can assign values to them using the already existing machinery. The first rule below ensures that the object is always explicit, the evaluation context enforces the object to be evaluated, and finally the second rule initiates the lookup for the member's location based on the current class of the object.

```
  rule <k> lvalue(X:Id => this . X) ...</k>  <env> Env </env>
    when notBool(X in keys(Env))  [structural]

  context lvalue((HOLE . _)::Exp)

/*  rule lvalue(objectClosure(<crntClass> C </crntClass>
                              <envStack>... envStackFrame(C,EnvC) EStack </envStack>)
              . X
              => lookupMember(<envStack> envStackFrame(C,EnvC) EStack </envStack>,
                              X))  [structural]*/
  rule lvalue(objectClosure(Class, ListItem(envStackFrame(Class,Env)) EStack)
              . X
              => lookupMember(ListItem(envStackFrame(Class,Env)) EStack,
                              X))  [structural]
  rule lvalue(objectClosure(Class, (ListItem(envStackFrame(Class':Id,_)) => .List) _)
              . _X)
    when Class =/=K Class'  [structural]
```

## Lookup member

It searches for the given member in the given environment stack,
starting with the most concrete class and going up in the hierarchy.

```
  // TODO(KORE): clarify sort inferences #1803
  syntax Exp ::= lookupMember(List, Id)  [function]
  /*
  syntax KItem ::= lookupMember(EnvStackCell,Id)  [function]
  */

//  rule lookupMember(<envStack> envStackFrame(_, <env>... X|->L ...</env>) ...</envStack>,
//    => lookup(L)
  rule lookupMember(ListItem(envStackFrame(_, X|->L _)) _, X)
    => lookup(L)

//  rule lookupMember(<envStack> envStackFrame(_, <env> Env </env>) => .List ...</envStack>
//    when notBool(X in keys(Env))
  rule lookupMember(ListItem(envStackFrame(_, Env)) Rest, X) =>
       lookupMember(Rest, X)
    when notBool(X in keys(Env))
//TODO: beautify the above

endmodule
```

Go to Lesson 2, KOOL typed dynamic.

# KOOL — Untyped

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped KOOL language. KOOL is aimed at being a pedagogical and research language that captures the essence of the object-oriented programming paradigm. Its untyped variant discussed here is simpler than the typed one, ignoring several intricate aspects of types in the presence of objects. A program consists of a set of class declarations. Each class can extend at most one other class (KOOL is single-inheritance). A class can declare a set of fields and a set of methods, all public and called the class' *members*. Specifically, KOOL includes the following features:

- Class declarations, where a class may or may not explicitly extend another class. In case a class does not explicitly extend another class, then it is assumed that it extends the default top-most and empty (i.e., no members) class called `Object`. Each class is required to declare precisely one homonymous method, called its *constructor*. Each valid program should contain one class named `Main`, whose constructor, `Main()`, takes no arguments. The execution of a program consists of creating an object instance of class `Main` and invoking the constructor `Main()` on it, that is, of executing `new Main();`.
- All features of SIMPLE (see `examples/simple/untyped`), i.e., multidimensional arrays, function (here called "method") abstractions with call-by-value parameter passing style and static scoping, blocks with locals, input/output, parametric exceptions, and concurrency via dynamic thread creation/termination and synchronization.
  The only change in the syntax of SIMPLE when imported in KOOL is the function declaration keyword, `function`, which is changed into `method`. The exact same desugaring macros from SIMPLE are also included in KOOL. We can think of KOOL's classes as embedding SIMPLE programs (extended with OO constructs, as discussed next).
- Object creation using the `new C(e1,...,en)` expression construct. An object instance of class `C` is first created and then the constructor `C(e1,...,en)` is implicitly called on that object. KOOL only allows (and requires) one

constructor per class. The class constructor can be called either implicitly during a new object creation for the class, or explicitly. The superclass constructor is **not** implicitly invoked when a class constructor is invoked; if you want to invoke the superclass constructor from a subclass constructor then you have to do it explicitly.

- An expression construct `this`, which evaluates to the current object.
- An expression construct `super`, which is used (only) in combination with member lookup (see next) to refer to a superclass field or method.
- A member lookup expression construct `e.x`, where `e` is an expression (either an expression expected to evaluate to an object or the `super` construct) and `x` is a class member name, that is, a field or a method name.
- Expression constructs `e instanceOf C` and `(C) e`, where `e` is an expression expected to evaluate to an object and `C` a class name. The former tells whether the class of `e` is a subclass of `C`, that is, whether `e` can be used as an instance of `C`, and the latter changes the class of `e` to `C`. These operations always succeed: the former returns a Boolean value, while the latter changes the current class of `e` to `C` regardless of whether it is safe to do so or not. The typed version of KOOL will check the safety of casting by ensuring that the instance class of the object is a subclass of `C`. In untyped KOOL we do not want to perform this check because we want to allow the programmer maximum of flexibility: if one always accesses only available members, then the program can execute successfully despite the potentially unsafe cast.

There are some specific aspects of KOOL that need to be discussed.

First, KOOL is higher-order, allowing function abstractions to be treated like any other values in the language. For example, if `m` is a method of object `e` then `e.m` evaluates to the corresponding function abstraction. The function abstraction is in fact a closure, because in addition to the method parameters and body it also encapsulates the object value (i.e., the environment of the object together with its current class—see below) that `e` evaluates to. This way, function abstractions can be invoked anywhere and have the capability to change the state of their object. For example, if `m` is a method of object `e` which increments a field `c` of `e` when invoked, and if `getm` is another method of `e` which simply returns

`m` when invoked, then the double application `(e.getm())()` has the same effect as `e.m()`, that is, increments the counter `c` of `e`. Note that the higher-order nature of KOOL was not originally planned; it came as a natural consequence of evaluating methods to closures and we decided to keep it. If you do not like it then do not use it.

Second, since all the fields and methods are public in KOOL and since they can be redeclared in subclasses, it is not immediately clear how to lookup the member `x` when we write `e.x` and `e` is different from `super`. We distinguish two cases, depending on whether `e.x` occurs in a method invocation context (i.e., `e.x(...)`) or in a field context. KOOL has dynamic method dispatch, so if `e.x` is invoked as a method then `x` will be searched for starting with the instance class of the object value to which `e` evaluates. If `e.x` occurs in a non-method-invocation context then `x` will be treated as a field (although it may hold a method closure due to the higher-order nature of KOOL) and thus will be searched starting with the current class of the object value of `e` (which, because of `this` and casting, may be different from its instance class). In order to achieve the above, each object value will consist of a pair holding the current class of the object and an environment stack with one layer for each class in the object's instance class hierarchy.

Third, although KOOL is dynamic method dispatch, its capabilities described above are powerful enough to allow us to mimic static method dispatch. For example, suppose that you want to invoke method `m()` statically. Then all you need to do is to declare a local variable and bind it to `m`, for example `var staticm = m;`, and then call `staticm()`. This works because `staticm` is first bound to the method closure that `m` evaluates to, and then looked up as any local variable when invoked. We only enable the dynamic method dispatch when we have an object member on an application position, e.g., `m()`.

In what follows, we limit our comments to the new, KOOL-specific aspects of the language. We refer the reader to the untyped SIMPLE language for documentation on the the remaining features, because those were all borrowed from SIMPLE.

```
module KOOL-UNTYPED-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of KOOL extends that of SIMPLE with object-oriented constructs. We removed from the **K** annotated syntax of SIMPLE two

constructs, namely the one for function declarations (because we want to call them `methods` now) and the one for function application (because application is not strict in the first argument anymore—needs to initiate dynamic method dispatch). The additional syntax includes:

- First, we need a new dedicated identifier, `Object`, for the default top-most class.
- Second, we rename the `function` keyword of SIMPLE into `method`.
- Third, we add syntax for class declarations together with a macro making classes which extend nothing to extend `Object`.
- Fourth, we change the strictness attribute of application into `strict(2)`.
- Finally, we add syntax and corresponding strictness for the KOOL object-oriented constructs.

```
syntax Id ::= "Object" [token] | "Main" [token]

syntax Stmt ::= "var" Exps ";"
              | "method" Id "(" Ids ")" Block  // called "function" in SIMPLE
              | "class" Id Block               // KOOL
              | "class" Id "extends" Id Block  // KOOL

syntax Exp ::= Int | Bool | String | Id
             | "this"                                  // KOOL
             | "super"                                 // KOOL
             | "(" Exp ")"             [bracket]
             | "++" Exp
             | Exp "instanceOf" Id     [strict(1)]     // KOOL
             | "(" Id ")" Exp          [strict(2)]     // KOOL  cast
             | "new" Id "(" Exps ")"   [strict(2)]     // KOOL
             | Exp "." Id                              // KOOL
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict(2)]     // was strict in SIMPLE
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
             | Exp "<=" Exp            [strict, non-assoc]
```

```
              | Exp ">" Exp             [strict, non-assoc]
              | Exp ">=" Exp            [strict, non-assoc]
              | Exp "==" Exp            [strict, non-assoc]
              | Exp "!=" Exp            [strict, non-assoc]
              > "!" Exp                 [strict]
              > left:
                Exp "&&" Exp            [strict(1), left]
              | Exp "||" Exp            [strict(1), left]
              > "spawn" Block
              > Exp "=" Exp             [strict(2), right]

  syntax Ids  ::= List{Id,","}

  syntax Exps ::= List{Exp,","}         [strict, klabel(exps)]
  syntax Val
  syntax Vals ::= List{Val,","}         [klabel(exps)]

  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt ::= Block
                 | Exp ";"                              [strict]
                 | "if" "(" Exp ")" Block "else" Block  [avoid, strict(1)]
                 | "if" "(" Exp ")" Block
                 | "while" "(" Exp ")" Block
             | "for" "(" Stmt Exp ";" Exp ")" Block
                 | "return" Exp ";"                     [strict]
                 | "return" ";"
                 | "print" "(" Exps ")" ";"             [strict]
                 | "try" Block "catch" "(" Id ")" Block
                 | "throw" Exp ";"                      [strict]
                 | "join" Exp ";"                       [strict]
                 | "acquire" Exp ";"                    [strict]
                 | "release" Exp ";"                    [strict]
                 | "rendezvous" Exp ";"                 [strict]

  syntax Stmt ::= Stmt Stmt                          [right]
```

Old desugaring rules, from SIMPLE

```
  rule if (E) S => if (E) S else {}                                  [macro]
  rule for(Start Cond; Step) {S} => {Start while (Cond) {S Step;}}   [macro]
  rule var E1::Exp, E2::Exp, Es::Exps; => var E1; var E2, Es;        [macro-rec]
  rule var X::Id = E; => var X; X = E;                               [macro]
```

New desugaring rule

```
  rule class C:Id S => class C extends Object S                      // KOOL
```

```
endmodule
```

## Semantics

We first discuss the new configuration of KOOL, which extends that of SIMPLE. Then we include the semantics of the constructs borrowed from SIMPLE unchanged; we refrain from discussing those, because they were already discussed in the **K** definition of SIMPLE. Then we discuss changes to SIMPLE's semantics needed for the more general meaning of the previous SIMPLE constructs (for example for thread spawning, assignment, etc.). Finally, we discuss in detail the semantics of the additional KOOL constructs.

```
module KOOL-UNTYPED
  imports KOOL-UNTYPED-SYNTAX
  imports DOMAINS
```

## Configuration

KOOL removes one cell and adds two nested cells to the configuration of SIMPLE. The cell which is removed is the one holding the global environment, because a KOOL program consists of a set of classes only, with no global declarations. In fact, since informally speaking each KOOL class now includes a SIMPLE program, it is safe to say that the global variables in SIMPLE became class fields in KOOL. Let us now discuss the new cells that are added to the configuration of SIMPLE.

- The cell `crntObj` holds data pertaining to the current object, that is, the object environment in which the code in cell `k` executes: `crntClass` holds the current class (which can change as methods of the current object are invoked); `envStack` holds the stack of environments as a list, each layer corresponding to one class in the objects' instance class hierarchy; `location`, which is optional, holds the location in the store where the current object is or has to be located (this is useful both for method closures and for the semantics of object creation).
- The cell `classes` holds all the declared classes, each class being held in its own `class` cell which contains a name (`className`), a parent (`extends`), and the actual member declarations (`declarations`).

```
  // the syntax declarations below are required because the sorts are
  // referenced directly by a production and, because of the way KIL to KORE
  // is implemented, the configuration syntax is not available yet
  // should simply work once KIL is removed completely
```

```
// check other definitions for this hack as well
syntax EnvCell
syntax ControlCell
syntax EnvStackCell
syntax CrntObjCellFragment

configuration <T color="red">
                <threads color="orange">
                  <thread multiplicity="*" type="Set" color="yellow">
                    <k color="green"> $PGM:Stmt ~> execute </k>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <control color="cyan">
                      <fstack color="blue"> .List </fstack>
                      <xstack color="purple"> .List </xstack>
                    //<br/> // TODO(KORE): support latex annotations #1799
                      <crntObj color="Fuchsia">  // KOOL
                         <crntClass> Object </crntClass>
                         <envStack> .List </envStack>
                         <location multiplicity="?"> .K </location>
                      </crntObj>
                    </control>
                  //<br/> // TODO(KORE): support latex annotations #1799
                    <env color="violet"> .Map </env>
                    <holds color="black"> .Map </holds>
                    <id color="pink"> 0 </id>
                  </thread>
                </threads>
              //<br/> // TODO(KORE): support latex annotations #1799
                <store color="white"> .Map </store>
                <busy color="cyan">.Set </busy>
                <terminated color="red"> .Set </terminated>
                <input color="magenta" stream="stdin"> .List </input>
                <output color="brown" stream="stdout"> .List </output>
                <nextLoc color="gray"> 0 </nextLoc>
              //<br/> // TODO(KORE): support latex annotations #1799
                <classes color="Fuchsia">        // KOOL
                   <classData multiplicity="*" type="Map" color="Fuchsia">
                      // the Map has as its key the first child of the cell,
                      // in this case the className cell.
                      <className color="Fuchsia"> Main </className>
                      <baseClass color="Fuchsia"> Object </baseClass>
                      <declarations color="Fuchsia"> .K </declarations>
                   </classData>
                </classes>
              </T>
```

## Unchanged Semantics from untyped SIMPLE

The semantics below is taken over from SIMPLE unchanged.
The semantics of function declaration and invocation, including the use of the special `lambda` abstraction value, needs to change in order to account for the fact that methods are now invoked into their object's environment. The semantics of function return actually stays unchanged. Also, the semantics of program initialization is different: now we have to create an instance of the `Main` class which also calls the constructor `Main()`, while in SIMPLE we only had to invoke the function `Main()`.
Finally, the semantics of thread spawning needs to change, too: the parent thread needs to also share its object environment with the spawned thread (in addition to its local environment, like in SIMPLE). This is needed in order to be able to spawn method invokations under dynamic method dispatch; for example, `spawn { run(); }` will need to look up the method `run()` in the newly created thread, operation which will most likely fail unless the child thread sees the object environment of the parent thread. Note that the `spawn` statement of KOOL is more permissive than the threads of Java. In fact, the latter can be implemented in terms of our `spawn`—see the program `threads.kool` for a sketch.

Below is a subset of the values of SIMPLE, which are also values of KOOL. We will add other values later in the semantics, such as object and method closures.

```
syntax Val ::= Int | Bool | String
             | array(Int,Int)
syntax Exp ::= Val
syntax Exps ::= Vals
syntax KResult ::= Val
syntax KResult ::= Vals
```

The semantics below are taken verbatim from the untyped SIMPLE definition.

```
syntax KItem ::= "undefined"  [latex(\bot)]

rule <k> var X:Id; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined ...</store>
     <nextLoc> L:Int => L +Int 1 </nextLoc>


context var _:Id[HOLE];

rule <k> var X:Id[N:Int]; => . ...</k>
```

```
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> array(L +Int 1, N)
                        (L +Int 1) ... (L +Int N) |-> undefined ...</store>
     <nextLoc> L:Int => L +Int 1 +Int N </nextLoc>
  when N >=Int 0


syntax Id ::= "$1" [token] | "$2" [token]
rule var X:Id[N1:Int, N2:Int, Vs:Vals];
  => var X[N1];
     {
       var $1=X;
       for(var $2=0; $2 <= N1 - 1; ++$2) {
         var X[N2,Vs];
         $1[$2] = X;
       }
     }
  [structural]


rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]


context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I:Int => I +Int 1) ...</store>  [increment]


rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2

rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
```

```
rule false && _ => false
rule true  || _ => true
rule false || E => E


rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
  [structural, anywhere]

rule array(L,_)[N:Int] => lookup(L +Int N)
  [structural, anywhere]


rule sizeOf(array(_,N)) => N
```

The semantics of function application needs to change into dynamic method dispatch invocation, which is defined shortly. However, interestingly, the semantics of return stays unchanged.

```
rule <k> return(V:Val); ~> _ => V ~> K </k>
     <control>
       <fstack> ListItem(fstackFrame(Env,K,XS,<crntObj> CO </crntObj>)) => .List ...</fsta
       <xstack> _ => XS </xstack>
       <crntObj> _ => CO </crntObj>
     </control>
     <env> _ => Env </env>

syntax Val ::= "nothing"
rule return; => return nothing;   [macro]


rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]


context (HOLE => lvalue(HOLE)) = _

rule <k> loc(L) = V:Val => V ...</k> <store>... L |-> (_ => V) ...</store>
  [assignment]


rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]


rule S1::Stmt S2::Stmt => S1 ~> S2  [structural]

rule _:Val; => .
```

```
rule if ( true) S else _ => S
rule if (false) _ else S => S

rule while (E) S => if (E) {S while(E)S}  [structural]

rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  [print]
rule print(.Vals); => .  [structural]


syntax KItem ::= xstackFrame(Id,Stmt,K,Map,K)
// TODO(KORE): drop the additional production once parsing issue #1842 is fixed
               | (Id,Stmt,K,Map,K)

syntax KItem ::= "popx"

rule <k> (try S1 catch(X) {S2} => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem(xstackFrame(X, S2, K, Env, C)) ...</xstack>
       C
     </control>
     <env> Env </env>

rule <k> popx => . ...</k>
     <xstack> ListItem(_) => .List ...</xstack>

rule <k> throw V:Val; ~> _ => { var X = V; S2 } ~> K </k>
     <control>
       <xstack> ListItem(xstackFrame(X, S2, K, Env, C)) => .List ...</xstack>
       (_ => C)
     </control>
     <env> _ => Env </env>
```

Thread spawning needs a new semantics, because we want the child thread to also share the object environment with its parent. The new semantics of thread spawning will be defined shortly. However, interestingly, the other concurrency constructs keep their semantics from SIMPLE unchanged.

```
// TODO(KORE): ..Bag should be . throughout this definition #1772
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
/*
rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .)
*/
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

```
rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>

rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  when (notBool(V in Busy:Set))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N:Int => N +Int 1) ...</holds>

rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N:Int -Int 1) ...</holds>
  when N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>

rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

## Unchanged auxiliary operations from untyped SIMPLE

```
syntax Stmt ::= mkDecls(Ids,Vals)  [function]
rule mkDecls((X:Id, Xs:Ids), (V:Val, Vs:Vals)) => var X=V; mkDecls(Xs,Vs)
rule mkDecls(.Ids,.Vals) => {}

// TODO(KORE): clarify sort inferences #1803
syntax Exp ::= lookup(Int)
/*
syntax KItem ::= lookup(Int)
*/
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]

syntax KItem ::= setEnv(Map)
rule <k> setEnv(Env) => . ...</k>  <env> _ => Env </env>  [structural]
rule (setEnv(_) => .) ~> setEnv(_)  [structural]
// TODO: How can we make sure that the second rule above applies before the first one?
//       Probably we'll deal with this using strategies, eventually.

syntax Exp ::= lvalue(K)
syntax Val ::= loc(Int)

rule <k> lvalue(X:Id => loc(L)) ...</k> <env>... X |-> L:Int ...</env>
  [structural]
```

```
context lvalue(_::Exp[HOLE::Exps])
context lvalue(HOLE::Exp[_::Exps])

rule lvalue(lookup(L:Int) => loc(L))  [structural]


syntax Map ::= Int "..." Int "|->" K
  [function, latex({#1}\ldots{#2}\mapsto{#3})]
rule N...M |-> _ => .Map  when N >Int M
rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  when N <=Int M
```

## Changes to the existing untyped SIMPLE semantics

When we extend a language, sometimes we need to do more than just add new language constructs and semantics for them. Sometimes we want to also extend the semantics of existing language constructs, in order to get more from them.

## Program initialization

In SIMPLE, once all the global declarations were processed, the function `main()` was invoked. In KOOL, the global declarations are classes, and their specific semantics is given shortly; essentially, they are pre-processed one by one and added into the `class` cell structure in the configuration.
Once all the classes are processed, the computation item `execute`, which was placed right after the program in the initial configuration, is reached. In SIMPLE, the program was initialized by calling the method `main()`. In KOOL, the program is initialized by creating an object instance of class `Main`. This will also implicitly call the method `Main()` (the `Main` class constructor). The emptiness of the `env` cell below is just a sanity check, to make sure that the user has not declared anything but classes at the top level of the program.

```
syntax KItem ::= "execute"
rule <k> execute => new Main(.Exps); </k> <env> .Map </env>  [structural]
```

The semantics of `new` (defined below) requires the execution of all the class' declarations (and also of its superclasses').

## Object and method closures

Before we can define the semantics of method application (previously called function application in SIMPLE), we need to add two more values

to the language, namely object and method closures:

```
syntax Val ::= objectClosure(Id, List)
             | methodClosure(Id,Int,Ids,Stmt)
```

An object value consists of an `objectClosure`-wrapped bag containing the current class of the object and the environment stack of the object. The current class of an object will always be one of the classes mapped to an environment in the environment stack of the object. A method closure encapsulates the method's parameters and code (last two arguments), as well as the object context in which the method code should execute. This object context includes the current class of the object (the first argument of `methodClosure`) and the object environment stack (located in the object stored at the location specified as the second argument of `methodClosure`).

## Method application

KOOL has a complex mechanism to invoke methods, because it allows both dynamic method dispatch and methods as first-class-citizen values (the latter making it a higher-order language). The invocation mechanism will be defined later. What is sufficient to know for now is that the two arguments of the application construct eventually reduce to values, the first being a method closure and the latter a list of values. The semantics of the method closure application is then as expected: the local environment and control are stacked, then we switch to method closure's class and object environment and execute the method body. The `mkDecls` construct is the one that came with the unchanged semantics of SIMPLE above.

```
syntax KItem ::= fstackFrame(Map,K,List,K)
// TODO(KORE): drop the additional production once parsing issue #1842 is fixed
               | (Map,K,K)

rule <k> methodClosure(Class,OL,Xs,S)(Vs:Vals) ~> K
         => mkDecls(Xs,Vs) S return; </k>
     <env> Env => .Map </env>
     <store>... OL |-> objectClosure(_, EnvStack)...</store>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control>
        <xstack> XS </xstack>
        <fstack> .List => ListItem(fstackFrame(Env, K, XS, <crntObj> Obj' </crntObj>))
        ...</fstack>
        <crntObj> Obj' => <crntClass> Class </crntClass> <envStack> EnvStack </envStack> <
     </control>
```

### Spawn

We want to extend the semantics of `spawn` to also share the current object environment with the child thread, in addition to the current environment. This extension will allow us to also use method invocations in the spawned statements, which will be thus looked up as expected, using dynamic method dispatch. This lookup operation would fail if the child thread did not have access to its parent's object environment.

```
rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
       <crntObj> Obj </crntObj>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
              <crntObj> Obj </crntObj>
            ...</thread>)
```

## Semantics of the new KOOL constructs

### Class declaration

Initially, the classes forming the program are moved into their corresponding cells:

```
rule <k> class Class1 extends Class2 { S } => . ...</k>
     <classes>... (.Bag => <classData>
                             <className> Class1 </className>
                             <baseClass> Class2 </baseClass>
                             <declarations> S </declarations>
                           </classData>)
     ...</classes>  [structural]
```

### Method declaration

Like in SIMPLE, method names are added to the environment and bound to their code. However, unlike in SIMPLE where each function was executed in the same environment, namely the program global environment, a method in KOOL needs to be executed into its object's environment. Thus, methods evaluate to closures, which encapsulate their object's context (i.e., the current class and environment stack of the object) in addition to method's parameters and body. This approach to bind method names to method closures in the environment will also allow objects to pass their methods to other objects, to

dynamically change their methods by assigning them other method closures, and even to allow all these to be done from other objects. This gives the KOOL programmer a lot of power; one should use this power wisely, though, because programs can become easily hard to understand and reason about if one overuses these features.

```
rule <k> method F:Id(Xs:Ids) S => . ...</k>
     <crntClass> Class:Id </crntClass>
     <location> OL:Int </location>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L |-> methodClosure(Class,OL,Xs,S) ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

## New

The semantics of `new` consists of two actions: memory allocation for the new object and execution of the corresponding constructor. Then the created object is returned as the result of the `new` operation; the value returned by the constructor, if any, is discarded. The current environment and object are stored onto the stack and recovered after new (according to the semantics of `return` borrowed from SIMPLE, when the statement `return this;` in the rule below is reached and evaluated), because the object creation part of `new` will destroy them. The rule below also initializes the object creation process by emptying the local environment and the current object, and allocating a location in the store where the created object will be eventually stored (this is what the `storeObj` task after the object creation task in the rule below will do—its rule is defined shortly). The location where the object will be stored is also made available in the `crntObj` cell, so that method closures can refer to it (see rule above).

```
syntax KItem ::= "envStackFrame" "(" Id "," Map ")"

rule <k> new Class:Id(Vs:Vals) ~> K
         => create(Class) ~> storeObj ~> Class(Vs); return this; </k>
     <env> Env => .Map </env>
     <nextLoc> L:Int => L +Int 1 </nextLoc>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control> <xstack> XS </xstack>
       <crntObj> Obj
                 => <crntClass> Object </crntClass>
                    <envStack> ListItem(envStackFrame(Object, .Map)) </envStack>
                    <location> L </location>
       </crntObj>
       <fstack> .List => ListItem(fstackFrame(Env, K, XS, <crntObj> Obj </crntObj>)) ...</
```

```
      </control>
```

The creation of a new object (the memory allocation part only) is a recursive process, requiring to first create an object for the superclass. A memory object representation is a layered structure: for each class on the path from the instance class to the root of the hierarchy there is a layer including the memory allocated for the members (both fields and methods) of that class.

```
  syntax KItem ::= create(Id)

  rule <k> create(Class:Id)
           => create(Class1) ~> setCrntClass(Class) ~> S ~> addEnvLayer ...</k>
       <className> Class </className>
       <baseClass> Class1:Id </baseClass>
       <declarations> S </declarations>  [structural]

  rule <k> create(Object) => . ...</k>  [structural]
```

The next operation sets the current class of the current object. This is necessary to be done at each layer, because the current class of the object is enclosed as part of the method closures (see the semantics of method declarations above).

```
  syntax KItem ::= setCrntClass(Id)

  rule <k> setCrntClass(C) => . ...</k>
       <crntClass> _ => C </crntClass>  [structural]
```

The next operation adds a new tagged environment layer to the current object and gets ready for the next layer by clearing the environment (note that `create` expects the environment to be empty).

```
  syntax KItem ::= "addEnvLayer"

  rule <k> addEnvLayer => . ...</k>
       <env> Env => .Map </env>
       <crntClass> Class:Id </crntClass>
       <envStack> .List => ListItem(envStackFrame(Class, Env)) ...</envStack>
    [structural]
```

The following operation stores the created object at the location reserved by `new`. Note that the location reserved by `new` was temporarily stored in the `crntObj` cell precisely for this purpose. Now that the newly created object is stored at its location and that all method closures are aware of it, the location is unnecessary and thus we delete it from the `crntObj` cell.

```
  syntax KItem ::= "storeObj"

  rule <k> storeObj => . ...</k>
       <crntObj> <crntClass> CC </crntClass> <envStack> ES </envStack> (<location> L:Int </l
       <store>... .Map => L |-> objectClosure(CC, ES) ...</store>
```

### Self reference

The semantics of `this` is straightforward: evaluate to the current object.

```
  rule <k> this => objectClosure(CC, ES) ...</k>
       <crntObj> <crntClass> CC </crntClass> <envStack> ES </envStack> </crntObj>
```

### Object member access

We can access an object member (field or method) either explicitly, using the construct `e.x`, or implicitly, using only the member name `x` directly. The borrowed semantics of SIMPLE will already lookup a sole name in the local environment. The first rule below reduces implicit member access to explicit access when the name cannot be found in the local environment. There are two cases to analyze for explicit object member access, depending upon whether the object is a proper object or it is just a redirection to the parent class via the construct `super`. In the first case, we evaluate the object expression and lookup the member starting with the current class (static scoping). Note the use of the conditional evaluation context. In the second case, we just lookup the member starting with the superclass of the current class. In both cases, the `lookupMember` task eventually yields a `lookup(L)` task for some appropriate location `L`, which will be further solved with the corresponding rule borrowed from SIMPLE. Note that the current object is not altered by `super`, so future method invocations see the entire object, as needed for dynamic method dispatch.

```
  rule <k> X:Id => this . X ...</k> <env> Env:Map </env>
    when notBool(X in keys(Env))  [structural]

  context HOLE._::Id when (HOLE =/=K super)

// TODO: explain how Assoc matching has been replaced with two rules here.
// Maybe also improve it a bit.

/*  rule objectClosure(<crntClass> Class:Id </crntClass>
                       <envStack>... envStackFrame(Class,EnvC) EStack </envStack>)
       . X:Id
    => lookupMember(envStackFrame(Class,EnvC) EStack, X)
```

```
    [structural]*/

  rule objectClosure(Class:Id, ListItem(envStackFrame(Class,Env)) EStack)
       . X:Id
    => lookupMember(ListItem(envStackFrame(Class,Env)) EStack, X)
    [structural]
  rule objectClosure(Class:Id, (ListItem(envStackFrame(Class':Id,_)) => .List) _)
       . _X:Id
    when Class =/=K Class'  [structural]

/*  rule <k> super . X => lookupMember(EStack, X) ...</k>
       <crntClass> Class </crntClass>
       <envStack>... envStackFrame(Class,EnvC) EStack </envStack>
    [structural]*/
  rule <k> super . X => lookupMember(EStack, X) ...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> super . _X ...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class':Id,_)) => .List ...</envStack>
    when Class =/=K Class'  [structural]
```

## Method invocation

Unlike in SIMPLE, in KOOL application was declared strict only in its second argument. That is because we want to ensure dynamic method dispatch when the first argument is a method access. As a consequence, we need to consider all the cases of interest for the first argument and to explicitly say what to do in each case. In all cases except for method access in a proper object (i.e., not `super`), we want the same behavior for the first argument as if it was not in a method invocation position. When it is a member access (the third rule below), we look it up starting with the instance class of the corresponding object. This ensures dynamic dispatch for methods; it actually dynamically dispatches field accesses, too, which is correct in KOOL, because one can assign method closures to fields and the field appeared in a method invocation context. The last context declaration below says that method applications or array accesses are also allowed as first argument to applications; that is because methods are allowed to return methods and arrays are allowed to hold methods in KOOL, since it is higher-order. If that is the case, then we want to evaluate the method call or the array access.

```
  rule <k> (X:Id => V)(_:Exps) ...</k>
       <env>... X |-> L ...</env>
       <store>... L |-> V:Val ...</store>  [lookup]

  rule <k> (X:Id => this . X)(_:Exps) ...</k>
       <env> Env </env>
    when notBool(X in keys(Env))  [structural]

  context HOLE._::Id(_) when HOLE =/=K super

  rule (objectClosure(_, EStack) . X
    => lookupMember(EStack, X:Id))(_:Exps)  [structural]

/*  rule <k> (super . X
             => lookupMember(EStack,X))(_:Exps)...</k>
       <crntClass> Class </crntClass>
       <envStack>... envStackFrame(Class,_) EStack </envStack>
    [structural]*/
  rule <k> (super . X
           => lookupMember(EStack,X))(_:Exps)...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> (super . _X)(_:Exps) ...</k>
       <crntClass> Class </crntClass>
       <envStack> ListItem(envStackFrame(Class':Id,_)) => .List ...</envStack>
    when Class =/=K Class'  [structural]

  // TODO(KORE): fix getKLabel #1801
  rule (A:Exp(B:Exps))(C:Exps) => A(B) ~> #freezerFunCall(C)
  rule (A:Exp[B:Exps])(C:Exps) => A[B] ~> #freezerFunCall(C)
  rule V:Val ~> #freezerFunCall(C:Exps) => V(C)
  syntax KItem ::= "#freezerFunCall" "(" K ")"
  /*
  context HOLE(_:Exps)
    when getKLabel(HOLE) ==K #klabel(`_(_)`) orBool getKLabel(HOLE) ==K #klabel(`_[_]`)
  */
```

Eventually, each of the rules above produces a `lookup(L)` task as a replacement for the method. When that happens, we just lookup the value at location `L`:

```
  rule <k> (lookup(L) => V)(_:Exps) ...</k>  <store>... L |-> V:Val ...</store>
    [lookup]
```

The value `V` looked up above is expected to be a method closure, in which case the semantics of method application given above will

apply. Otherwise, the execution will get stuck.

### Instance Of

It searches the object environment for a layer corresponding to the desired class. It returns `true` iff it can find the class, otherwise it returns `false`; it only gets stuck when its first argument does not evaluate to an object.

```
  rule objectClosure(_, ListItem(envStackFrame(C,_)) _)
       instanceOf C => true

  rule objectClosure(_, (ListItem(envStackFrame(C,_)) => .List) _)
       instanceOf C'  when C =/=K C'  [structural]
//TODO: remove the sort cast ::Id of C above, when sort inference bug fixed

  rule objectClosure(_, .List) instanceOf _ => false
```

### Cast

In untyped KOOL, we prefer to not check the validity of casting. In other words, any cast is allowed on any object, simply changing the current class of the object to the desired class. The execution will get stuck later if one attempts to access a field which is not available. Moreover, the execution may complete successfully even in the presence of invalid casts, provided that each accessed member during the current execution is, or happens to be, available.

```
  rule (C) objectClosure(_ , EnvStack) => objectClosure(C ,EnvStack)
```

## KOOL-specific auxiliary declarations and operations

Here we define all the auxiliary constructs used in the above KOOL-specific semantics (those used in the SIMPLE fragment have already been defined in a corresponding section above).

### Objects as lvalues

The current machinery borrowed with the semantics of SIMPLE allows us to enrich the set of lvalues, this way allowing new means to assign values to locations. In KOOL, we want object member names to be lvalues, so that we can assign values to them using the already existing machinery. The first rule below ensures that the object is always explicit, the evaluation context enforces the object to be evaluated, and finally the second rule initiates the lookup for the member's location based on the current class of the object.

```
  rule <k> lvalue(X:Id => this . X) ...</k>  <env> Env </env>
    when notBool(X in keys(Env))  [structural]

  context lvalue((HOLE . _)::Exp)

/*  rule lvalue(objectClosure(<crntClass> C </crntClass>
                              <envStack>... envStackFrame(C,EnvC) EStack </envStack>)
              . X
              => lookupMember(<envStack> envStackFrame(C,EnvC) EStack </envStack>,
                              X))  [structural]*/
  rule lvalue(objectClosure(Class, ListItem(envStackFrame(Class,Env)) EStack)
              . X
              => lookupMember(ListItem(envStackFrame(Class,Env)) EStack,
                              X))  [structural]
  rule lvalue(objectClosure(Class, (ListItem(envStackFrame(Class':Id,_)) => .List) _)
              . _X)
    when Class =/=K Class'  [structural]
```

## Lookup member

It searches for the given member in the given environment stack,
starting with the most concrete class and going up in the hierarchy.

```
  // TODO(KORE): clarify sort inferences #1803
  syntax Exp ::= lookupMember(List, Id)  [function]
  /*
  syntax KItem ::= lookupMember(EnvStackCell,Id)  [function]
  */

//  rule lookupMember(<envStack> envStackFrame(_, <env>... X|->L ...</env>) ...</envStack>,
//    => lookup(L)
  rule lookupMember(ListItem(envStackFrame(_, X|->L _)) _, X)
    => lookup(L)

//  rule lookupMember(<envStack> envStackFrame(_, <env> Env </env>) => .List ...</envStack>
//    when notBool(X in keys(Env))
  rule lookupMember(ListItem(envStackFrame(_, Env)) Rest, X) =>
       lookupMember(Rest, X)
    when notBool(X in keys(Env))
//TODO: beautify the above

endmodule
```

Go to Lesson 2, KOOL typed dynamic.

# KOOL — Typed — Dynamic

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** dynamic semantics of the typed KOOL language. It is very similar to the semantics of the untyped KOOL, the difference being that we now check the typing policy dynamically. Since we have to now declare the types of variables and methods, we adopt a syntax for those which is close to Java. Like in the semantics of untyped KOOL, where we borrowed almost all the semantics of untyped SIMPLE, we are going to also borrow much of the semantics of dynamically typed SIMPLE here. We will highlight the differences between the dynamically typed and the untyped KOOL as we proceed with the semantics. In general, the type policy of the typed KOOL language is similar to that of Java. You may find it useful to also read the discussion in the preamble of the static semantics of typed KOOL before proceeding.

```
module KOOL-TYPED-DYNAMIC-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

Like for the untyped KOOL language, the syntax of typed KOOL extends that of typed SIMPLE with object-oriented constructs.
The syntax below was produced by copying and modifying/extending the syntax of dynamically typed SIMPLE. In fact, the only change we made to the existing syntax of dynamically typed SIMPLE was to change the strictness of the application construct like in untyped KOOL, from `strict` to `strict(2)` (because application is not strict in the first argument anymore due to dynamic method dispatch). The KOOL-specific syntactic extensions are identical to those in untyped KOOL.

```
  syntax Id ::= "Object" [token] | "Main" [token]
```

## Types

```
  syntax Type ::= "void" | "int" | "bool" | "string"
                | Id                                  // KOOL class
                | Type "[" "]"
                | "(" Type ")"           [bracket]
```

```
              > Types "->" Type
// TODO(KORE): drop klabel once issues #1913 are fixed
syntax Types ::= List{Type,","}   [klabel(_,_::Types)]
/*
syntax Types ::= List{Type,","}
*/
```

## Declarations

```
syntax Param ::= Type Id
syntax Params ::= List{Param,","}

syntax Stmt ::= Type Exps ";" [avoid]
              | Type Id "(" Params ")" Block    // stays like in typed SIMPLE
              | "class" Id Block                // KOOL
              | "class" Id "extends" Id Block   // KOOL
```

## Expressions

```
syntax Exp ::= Int | Bool | String | Id
             | "this"                                  // KOOL
             | "super"                                 // KOOL
             | "(" Exp ")"             [bracket]
             | "++" Exp
             | Exp "instanceOf" Id     [strict(1)]     // KOOL
             | "(" Id ")" Exp          [strict(2)]     // KOOL  cast
             | "new" Id "(" Exps ")"   [strict(2)]     // KOOL
             | Exp "." Id                              // KOOL
             > Exp "[" Exps "]"        [strict]
             > Exp "(" Exps ")"        [strict(2)]     // was strict in SIMPLE
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
             | Exp "<=" Exp            [strict, non-assoc]
             | Exp ">" Exp             [strict, non-assoc]
             | Exp ">=" Exp            [strict, non-assoc]
             | Exp "==" Exp            [strict, non-assoc]
```

```
              | Exp "!=" Exp            [strict, non-assoc]
              > "!" Exp                 [strict]
              > left:
                Exp "&&" Exp            [strict(1), left]
              | Exp "||" Exp            [strict(1), left]
              > "spawn" Block
              > Exp "=" Exp             [strict(2), right]

  syntax Exps ::= List{Exp,","}         [strict, klabel(exps)]
  syntax Val
  syntax Vals ::= List{Val,","}         [klabel(exps)]
```

### Statements

```
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt ::= Block
                | Exp ";"                               [strict]
                | "if" "(" Exp ")" Block "else" Block   [avoid, strict(1)]
                | "if" "(" Exp ")" Block
                | "while" "(" Exp ")" Block
                | "for" "(" Stmt Exp ";" Exp ")" Block
                | "print" "(" Exps ")" ";"              [strict]
                | "return" Exp ";"                      [strict]
                | "return" ";"
                | "try" Block "catch" "(" Param ")" Block
                | "throw" Exp ";"                       [strict]
                | "join" Exp ";"                        [strict]
                | "acquire" Exp ";"                     [strict]
                | "release" Exp ";"                     [strict]
                | "rendezvous" Exp ";"                  [strict]

  syntax Stmt ::= Stmt Stmt                           [right]
```

### Desugaring macros

```
  rule if (E) S => if (E) S else {}                                      [macro]
  rule for(Start Cond; Step) {S::Stmt} => {Start while(Cond){S Step;}} [macro]
  rule T::Type E1::Exp, E2::Exp, Es::Exps; => T E1; T E2, Es;           [macro-rec]
  rule T::Type X::Id = E; => T X; X = E;                                 [macro]

  rule class C:Id S => class C extends Object S                    // KOOL

endmodule
```

## Semantics

We first discuss the new configuration, then we include the semantics of the constructs borrowed from SIMPLE which stay unchanged, then those whose semantics had to change, and finally the semantics of the KOOL-specific constructs.

```
module KOOL-TYPED-DYNAMIC
  imports KOOL-TYPED-DYNAMIC-SYNTAX
  imports DOMAINS
```

## Configuration

The configuration of dynamically typed KOOL is almost identical to that of its untyped variant. The only difference is the cell `return`, inside the `control` cell, whose role is to hold the expected return type of the invoked method. That is because we want to dynamically check that the value that a method returns has the expected type.

```
  // the syntax declarations below are required because the sorts are
  // referenced directly by a production and, because of the way KIL to KORE
  // is implemented, the configuration syntax is not available yet
  // should simply work once KIL is removed completely
  // check other definitions for this hack as well
  syntax EnvCell
  syntax ControlCellFragment
  syntax EnvStackCell
  syntax CrntObjCellFragment

  configuration <T color="red">
                  <threads color="orange">
                    <thread multiplicity="*" type="Set" color="yellow">
                      <k color="green"> ($PGM:Stmt ~> execute) </k>
                    //<br/> // TODO(KORE): support latex annotations #1799
                      <control color="cyan">
                        <fstack color="blue"> .List </fstack>
                        <xstack color="purple"> .List </xstack>
                        <returnType color="LimeGreen"> void </returnType>  // KOOL
                      //<br/> // TODO(KORE): support latex annotations #1799
                        <crntObj color="Fuchsia">  // KOOL
                           <crntClass> Object </crntClass>
                           <envStack> .List </envStack>
                           <location multiplicity="?"> .K </location>
                        </crntObj>
                      </control>
                    //<br/> // TODO(KORE): support latex annotations #1799
```

```
      <env color="violet"> .Map </env>
      <holds color="black"> .Map </holds>
      <id color="pink"> 0 </id>
    </thread>
  </threads>
//<br/> // TODO(KORE): support latex annotations #1799
  <store color="white"> .Map </store>
  <busy color="cyan">.Set </busy>
  <terminated color="red"> .Set </terminated>
  <input color="magenta" stream="stdin"> .List </input>
  <output color="brown" stream="stdout"> .List </output>
  <nextLoc color="gray"> 0 </nextLoc>
//<br/> // TODO(KORE): support latex annotations #1799
  <classes color="Fuchsia">        // KOOL
     <classData multiplicity="*" type="Map" color="Fuchsia">
        <className color="Fuchsia"> Main </className>
        <baseClass color="Fuchsia"> Object </baseClass>
        <declarations color="Fuchsia"> .K </declarations>
     </classData>
  </classes>
</T>
```

## Unchanged semantics from dynamically typed SIMPLE

The semantics below is taken over from dynamically typed SIMPLE unchanged. Like for untyped KOOL, the semantics of function/method declaration and invocation, and of program initialization needs to change. Moreover, due to subtyping, the semantics of several imported SIMPLE constructs can be made more general, such as that of the return statement, that of the assignment, and that of the exceptions. We removed all these from the imported semantics of SIMPLE below and gave their modified semantics right after, together with the extended semantics of thread spawning (which is identical to that of untyped KOOL).

```
syntax Val ::= Int | Bool | String
             | array(Type,Int,Int)
syntax Exp ::= Val
syntax Exps ::= Vals
syntax KResult ::= Val
syntax KResult ::= Vals


syntax KItem ::= undefined(Type)  [latex(\bot_{#1})]

rule <k> T:Type X:Id; => . ...</k>
```

```
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined(T) ...</store>
     <nextLoc> L:Int => L +Int 1 </nextLoc>


rule <k> T:Type X:Id[N:Int]; => . ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> array(T, L +Int 1, N)
                        (L +Int 1)...(L +Int N) |-> undefined(T) ...</store>
     <nextLoc> L:Int => L +Int 1 +Int N </nextLoc>
  requires N >=Int 0

context _:Type _::Exp[HOLE::Exps];


syntax Id ::= "$1" [token] | "$2" [token]
rule T:Type X:Id[N1:Int, N2:Int, Vs:Vals];
  => T[]<Vs> X[N1];
     {
       T[][]<Vs> $1=X;
       for(int $2=0; $2 <= N1 - 1; ++$2) {
         T X[N2,Vs];
         $1[$2] = X;
       }
     }
  [structural]


rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V:Val ...</store>  [lookup]


context ++(HOLE => lvalue(HOLE))
rule <k> ++loc(L) => I +Int 1 ...</k>
     <store>... L |-> (I:Int => I +Int 1) ...</store>  [increment]


rule I1 + I2 => I1 +Int I2
rule Str1 + Str2 => Str1 +String Str2
rule I1 - I2 => I1 -Int I2
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 requires I2 =/=K 0
rule I1 % I2 => I1 %Int I2 requires I2 =/=K 0
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
```

```
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E


rule V:Val[N1:Int, N2:Int, Vs:Vals] => V[N1][N2, Vs]
  [structural, anywhere]

rule array(_:Type, L:Int, M:Int)[N:Int] => lookup(L +Int N)
  requires N >=Int 0 andBool N <Int M  [structural, anywhere]

rule sizeOf(array(_,_,N)) => N


syntax Val ::= nothing(Type)
rule <k> return; => return nothing(T); ...</k> <returnType> T </returnType>
  [structural]


rule <k> read() => I ...</k> <input> ListItem(I:Int) => .List ...</input>  [read]


context (HOLE => lvalue(HOLE)) = _


rule {} => .  [structural]
rule <k> { S } => S ~> setEnv(Env) ...</k>  <env> Env </env>  [structural]


rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]


rule _:Val; => .


rule if ( true) S else _ => S
rule if (false) _ else S => S
```

```
rule while (E) S => if (E) {S while(E)S}  [structural]


rule <k> print(V:Val, Es => Es); ...</k> <output>... .List => ListItem(V) </output>
  requires typeOf(V) ==K int orBool typeOf(V) ==K string  [print]
rule print(.Vals); => .  [structural]


rule (<thread>... <k>.</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>

rule <k> join T:Int; => . ...</k>
     <terminated>... SetItem(T) ...</terminated>

rule <k> acquire V:Val; => . ...</k>
     <holds>... .Map => V |-> 0 ...</holds>
     <busy> Busy (.Set => SetItem(V)) </busy>
  requires (notBool(V in Busy:Set))  [acquire]

rule <k> acquire V; => . ...</k>
     <holds>... V:Val |-> (N:Int => N +Int 1) ...</holds>

rule <k> release V:Val; => . ...</k>
     <holds>... V |-> (N => N:Int -Int 1) ...</holds>
  requires N >Int 0

rule <k> release V; => . ...</k> <holds>... V:Val |-> 0 => .Map ...</holds>
     <busy>... SetItem(V) => .Set ...</busy>

rule <k> rendezvous V:Val; => . ...</k>
     <k> rendezvous V; => . ...</k>  [rendezvous]
```

## Unchanged auxiliary operations from dynamically typed SIMPLE

```
syntax Stmt ::= mkDecls(Params,Vals)  [function]
rule mkDecls((T:Type X:Id, Ps:Params), (V:Val, Vs:Vals))
  => T X=V; mkDecls(Ps,Vs)
rule mkDecls(.Params,.Vals) => {}

syntax Exp ::= lookup(Int)
rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>  [lookup]

syntax KItem ::= setEnv(Map)
rule <k> setEnv(Env) => . ...</k>  <env> _ => Env </env>  [structural]
```

```
rule (setEnv(_) => .) ~> setEnv(_)  [structural]

syntax Exp ::= lvalue(K)
syntax Val ::= loc(Int)
rule <k> lvalue(X:Id => loc(L)) ...</k>  <env>... X |-> L:Int ...</env>
  [structural]

context lvalue(_::Exp[HOLE::Exps])
context lvalue(HOLE::Exp[_::Exps])

rule lvalue(lookup(L:Int) => loc(L))  [structural]

syntax Type ::= Type "<" Vals ">"  [function]
rule T:Type<_,Vs:Vals> => T[]<Vs>
rule T:Type<.Vals> => T

syntax Map ::= Int "..." Int "|->" K
  [function, latex({#1}\ldots{#2}\mapsto{#3})]
rule N...M |-> _ => .Map  requires N >Int M
rule N...M |-> K => N |-> K (N +Int 1)...M |-> K  requires N <=Int M

syntax Type ::= typeOf(K)  [function]
rule typeOf(_:Int) => int
rule typeOf(_:Bool) => bool
rule typeOf(_:String) => string
rule typeOf(array(T,_,_)) => (T[])
rule typeOf(undefined(T)) => T
rule typeOf(nothing(T)) => T

syntax Types ::= getTypes(Params)  [function]
rule getTypes(T:Type _:Id) => T, .Types
rule getTypes(T:Type _:Id, P, Ps) => T, getTypes(P,Ps)
rule getTypes(.Params) => void, .Types
```

## Changes to the existing dynamically typed SIMPLE semantics

We extend/change the semantics of several SIMPLE constructs in order to take advantage of the richer KOOL semantic infrastructure and thus get more from the existing SIMPLE constructs.

## Program initialization

Like in untyped KOOL.

```
syntax KItem ::= "execute"
rule <k> execute => new Main(.Exps); </k> <env> .Map </env>  [structural]
```

### Method application

The only change to untyped KOOL's values is that method closures are now typed (their first argument holds their type):

```
syntax Val ::= objectClosure(Id,List)
             | methodClosure(Type,Id,Int,Params,Stmt)
```

The type held by a method clossure will be the entire type of the method, not only its result type like the lambda-closure of typed SIMPLE. The reason for this change comes from the the need to dynamically upcast values when passed to contexts where values of superclass types are expected; since we want method closures to be first-class-citizen values in our language, we have to be able to dynamically upcast them, and in order to do that elegantly it is convenient to store the entire "current type" of the method closure instead of just its result type. Note that this was unnecessary in the semantics of the dynamically typed SIMPLE language.

Method closure application needs to also set a new return type in the `return` cell, like in dynamically typed SIMPLE, in order for the values returned by its body to be checked against the return type of the method. To do this correctly, we also need to stack the current status of the `return` cell and then pop it when the method returns. We have to do the same with the current object environment, so we group them together in the stack frame.

```
syntax KItem ::= fstackFrame(Map, K, List, Type, K)

rule <k> methodClosure(_->T,Class,OL,Ps,S)(Vs:Vals) ~> K
          => mkDecls(Ps,Vs) S return; </k>
     <env> Env => .Map </env>
     <store>... OL |-> objectClosure(_, EStack)...</store>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control>
        <fstack> .List => ListItem(fstackFrame(Env, K, XS, T', <crntObj> Obj' </crntObj>))
        <xstack> XS </xstack>
        <returnType> T' => T </returnType>
        <crntObj> Obj' => <crntClass> Class </crntClass> <envStack> EStack </envStack> </c
     </control>
```

At method return, we have to check that the type of the returned value is a subtype of the expected return type. Moreover, if that is the case, then we also upcast the returned value to one of the expected type. The computation item `unsafeCast(V,T)` changes

the typeof `V` to `T` without any additional checks; however, it only does it when `V` is an object or a method, otherwise it returns `V` unchanged.

```
rule <k> return V:Val; ~> _
         => subtype(typeOf(V), T) ~> true? ~> unsafeCast(V, T) ~> K
     </k>
     <control>
       <fstack> ListItem(fstackFrame(Env, K, XS, RT, <crntObj> CO </crntObj>)) => .List ..
       <xstack> _ => XS </xstack>
       <returnType> T:Type => RT </returnType>
       <crntObj> _ => CO </crntObj>
     </control>
     <env> _ => Env </env>
```

## Assignment

Typed KOOL allows to assign subtype instance values to supertype lvalues. The semantics of assignment below is similar in spirit to dynamically typed SIMPLE's, but a check is performed that the assigned value's type is a subtype of the location's type. If that is the case, then the assigned value is returned as a result and stored, but it is upcast appropriately first, so the context will continue to see a value of the expected type of the location. Note that the type of a location is implicit in the type of its contents and it never changes during the execution of a program; its type is assigned when the location is allocated and initialized, and then only type-preserving values are allowed to be stored in each location.

```
rule <k> loc(L) = V:Val
         => subtype(typeOf(V),typeOf(V')) ~> true?
            ~> unsafeCast(V, typeOf(V')) ...</k>
     <store>... L |-> (V' => unsafeCast(V, typeOf(V'))) ...</store>
  [assignment]
```

## Typed exceptions

Exceptions are propagated now until a catch that can handle them is encountered.

```
syntax KItem ::= xstackFrame(Param, Stmt, K, Map, K)
syntax KItem ::= "popx"

rule <k> (try S1 catch(P) S2 => S1 ~> popx) ~> K </k>
     <control>
       <xstack> .List => ListItem(xstackFrame(P, S2, K, Env, C)) ...</xstack>
       C
```

```
      </control>
      <env> Env </env>

  rule <k> popx => . ...</k>
      <xstack> ListItem(_) => .List ...</xstack>

  rule <k> throw V:Val; ~> _
       => if (subtype(typeOf(V),T)) { T X = V; S2 } else { throw V; } ~> K
      </k>
      <control>
        <xstack> ListItem(xstackFrame(T:Type X:Id, S2, K, Env, C)) => .List ...</xstack>
        (_ => C)
      </control>
      <env> _ => Env </env>
```

### Spawn

Like in untyped KOOL.

```
  rule <thread>...
         <k> spawn S => !T:Int ...</k>
         <env> Env </env>
         <crntObj> Obj </crntObj>
       ...</thread>
       (.Bag => <thread>...
                <k> S </k>
                <env> Env </env>
                <id> !T </id>
                <crntObj> Obj </crntObj>
              ...</thread>)
```

## Semantics of the new KOOL constructs

### Class declaration

Like in untyped KOOL.

```
  rule <k> class Class1 extends Class2 { S } => . ...</k>
       <classes>... (.Bag => <classData>
                               <className> Class1 </className>
                               <baseClass> Class2 </baseClass>
                               <declarations> S </declarations>
                             </classData>)
       ...</classes>  [structural]
```

## Method declaration

Methods are now typed and we need to store their types in their closures, so that their type contract can be checked at invocation time. The rule below is conceptually similar to that of untyped KOOL; the only difference is the addition of the types.

```
rule <k> T:Type F:Id(Ps:Params) S => . ...</k>
     <crntClass> C </crntClass>
     <location> OL </location>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L|->methodClosure(getTypes(Ps)->T,C,OL,Ps,S) ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

## New

The semantics of `new` in dynamically typed KOOL is also similar to that in untyped KOOL, the main difference being the management of the return types. Indeed, when a new object is created we also have to stack the current type in the `return` cell in order to be recovered after the creation of the new object. Only the first rule below needs to be changed; the others are identical to those in untyped KOOL.

```
syntax KItem ::= envStackFrame(Id, Map)

rule <k> new Class:Id(Vs:Vals) ~> K
         => create(Class) ~> (storeObj ~> ((Class(Vs)); return this;)) </k>
     <env> Env => .Map </env>
     <nextLoc> L:Int => L +Int 1 </nextLoc>
   //<br/> // TODO(KORE): support latex annotations #1799
     <control>
       <xstack> XS </xstack>
       <crntObj> Obj
                 => <crntClass> Object </crntClass>
                    <envStack> ListItem(envStackFrame(Object, .Map)) </envStack>
                    <location> L </location>
       </crntObj>
       <returnType> T => Class </returnType>
       <fstack> .List => ListItem(fstackFrame(Env, K, XS, T, <crntObj>Obj</crntObj>)) ...<
     </control>

syntax KItem ::= create(Id)

rule <k> create(Class:Id)
         => create(Class1) ~> setCrntClass(Class) ~> S ~> addEnvLayer ...</k>
     <className> Class </className>
```

```
      <baseClass> Class1:Id </baseClass>
      <declarations> S </declarations>  [structural]

  rule <k> create(Object) => . ...</k>  [structural]

  syntax KItem ::= setCrntClass(Id)

  rule <k> setCrntClass(C) => . ...</k>
       <crntClass> _ => C </crntClass>  [structural]

  syntax KItem ::= "addEnvLayer"

  rule <k> addEnvLayer => . ...</k>
       <env> Env => .Map </env>
       <crntClass> Class:Id </crntClass>
       <envStack> .List => ListItem(envStackFrame(Class, Env)) ...</envStack>
    [structural]

  syntax KItem ::= "storeObj"

  rule <k> storeObj => . ...</k>
       <crntObj>
         <crntClass> Class </crntClass>
         <envStack> EStack </envStack>
         (<location> L:Int </location> => .Bag)
       </crntObj>
       <store>... .Map => L |-> objectClosure(Class, EStack) ...</store>
```

### Self reference

Like in untyped KOOL.

```
  rule <k> this => objectClosure(Class, EStack) ...</k>
       <crntObj>
         <crntClass> Class </crntClass>
         <envStack> EStack </envStack>
         ...
       </crntObj>
```

### Object member access

Like in untyped KOOL.

```
  rule <k> X:Id => this . X ...</k> <env> Env:Map </env>
    requires notBool(X in keys(Env))  [structural]

  context HOLE . _::Id requires (HOLE =/=K super)
```

```
/*  rule objectClosure(<crntObj> <crntClass> Class:Id </crntClass>
                    <envStack>... ListItem((Class,EnvC:EnvCell)) EStack </envStack> </crntC
       . X:Id
    => lookupMember(<envStack> ListItem((Class,EnvC)) EStack </envStack>, X)
    [structural]*/
  rule objectClosure(Class:Id,
                    ListItem(envStackFrame(Class,Env)) EStack)
       . X:Id
    => lookupMember(ListItem(envStackFrame(Class,Env)) EStack, X)
    [structural]
  rule objectClosure(Class:Id,
                    (ListItem(envStackFrame(Class':Id,_)) => .List) _EStack)
       . _X:Id
    requires Class =/=K Class' [structural]

/*  rule <k> super . X => lookupMember(<envStack>EStack</envStack>, X) ...</k>
       <crntClass> Class </crntClass>
       <envStack>... ListItem((Class,EnvC:EnvCell)) EStack </envStack>
    [structural]*/
  rule <k> super . X => lookupMember(EStack, X) ...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> super . _X ...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> (ListItem(envStackFrame(Class':Id,_)) => .List) _EStack </envStack>
    requires Class =/=K Class' [structural]
```

## Method invocation

The method lookup is the same as in untyped KOOL.

```
  rule <k> (X:Id => V)(_:Exps) ...</k>
       <env>... X |-> L ...</env>
       <store>... L |-> V:Val ...</store>  [lookup]

  rule <k> (X:Id => this . X)(_:Exps) ...</k>
       <env> Env </env>
    requires notBool(X in keys(Env))  [structural]

  context HOLE._::Id(_) requires HOLE =/=K super

  rule (objectClosure(_, EStack) . X
    => lookupMember(EStack, X:Id))(_:Exps)  [structural]
```

```
/*  rule <k> (super . X
            => lookupMember(<envStack>EStack</envStack>,X))(_:Exps)...</k>
       <crntClass> Class </crntClass>
       <envStack>... ListItem((Class,_)) EStack </envStack>
    [structural]*/
  rule <k> (super . X
            => lookupMember(EStack,X))(_:Exps)...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> ListItem(envStackFrame(Class,_)) EStack </envStack>
    [structural]
  rule <k> (super . _X)(_:Exps)...</k>
       <crntClass> Class:Id </crntClass>
       <envStack> (ListItem(envStackFrame(Class':Id,_)) => .List) _EStack </envStack>
    requires Class =/=K Class' [structural]

  // TODO(KORE): fix getKLabel #1801
  rule (A:Exp(B:Exps))(C:Exps) => A(B) ~> #freezerFunCall(C)
  rule (A:Exp[B:Exps])(C:Exps) => A[B] ~> #freezerFunCall(C)
  rule V:Val ~> #freezerFunCall(C:Exps) => V(C)
  syntax KItem ::= "#freezerFunCall" "(" K ")"
  /*
  context HOLE(_:Exps)
    requires getKLabel HOLE ==KLabel '_`(_`) orBool getKLabel HOLE ==KLabel '_`[_`]
  */

  rule <k> (lookup(L) => V)(_:Exps) ...</k>  <store>... L |-> V:Val ...</store>
    [lookup]
```

## Instance of

Like in untyped KOOL.

```
  rule objectClosure(_, ListItem(envStackFrame(C,_)) _)
       instanceOf C => true

  rule objectClosure(_, (ListItem(envStackFrame(C::Id,_)) => .List) _)
       instanceOf C'  requires C =/=K C'  [structural]

  rule objectClosure(_, .List) instanceOf _ => false
```

## Cast

Unlike in untyped KOOL, in typed KOOL we actually check that the object can indeed be cast to the claimed type.

```
  rule (C:Id) objectClosure(Irrelevant, EStack)
    => objectClosure(Irrelevant, EStack) instanceOf C ~> true?
```

```
      ~> objectClosure(C, EStack)
```

## KOOL-specific auxiliary declarations and operations

### Objects as lvalues

Like in untyped KOOL.

```
  rule <k> lvalue(X:Id => this . X) ...</k>  <env> Env </env>
    requires notBool(X in keys(Env))  [structural]

  context lvalue((HOLE . _)::Exp)

/*  rule lvalue(objectClosure(<crntObj> <crntClass> C </crntClass>
                              <envStack>... ListItem((C,EnvC:EnvCell)) EStack </envStack> </cr
              . X
              => lookupMember(<envStack> ListItem((C,EnvC)) EStack </envStack>,
                                X))  [structural]*/
  rule lvalue(objectClosure(C:Id,
                              ListItem(envStackFrame(C,Env)) EStack)
              . X
              => lookupMember(ListItem(envStackFrame(C,Env)) EStack,
                                X))  [structural]
  rule lvalue(objectClosure(C,
                              (ListItem(envStackFrame(C',_)) => .List) _EStack)
              . _X)
    requires C =/=K C' [structural]
```

### Lookup member

Like in untyped KOOL.

```
  syntax Exp ::= lookupMember(List,Id)  [function]

  rule lookupMember(ListItem(envStackFrame(_, X |-> L _)) _, X) => lookup(L)

  // TODO: fix rule below as shown once we support functions with deep rewrites
  // rule lookupMember(<envStack> ListItem((_, <env> Env </env>)) => .List
  //                       ...</envStack>, X)
  //   requires notBool(X in keys(Env))
  rule lookupMember(ListItem(envStackFrame(_, Env)) L, X)
    => lookupMember(L, X)
    requires notBool(X in keys(Env))
```

### `typeOf` for the additional values}

```
  rule typeOf(objectClosure(C,_)) => C
  rule typeOf(methodClosure(T:Type,_,_,_Ps:Params,_)) => T
```

### Subtype checking

The subclass relation induces a subtyping relation.

```
  syntax Exp ::= subtype(Types,Types)

  rule subtype(T:Type, T) => true  [structural]

  rule <k> subtype(C1:Id, C:Id) => subtype(C2, C) ...</k>
       <className> C1 </className>
       <baseClass> C2:Id </baseClass>
    requires C1 =/=K C  [structural]

  rule subtype(Object,Class:Id) => false
    requires Class =/=K Object  [structural]

  rule subtype(Ts1->T2,Ts1'->T2') => subtype(((T2)::Type,Ts1'),((T2')::Type,Ts1))  [structu

// Note that the following rule would be wrong!
//  rule subtype(T[],T'[]) => subtype(T,T')

  rule subtype((T:Type,Ts),(T':Type,Ts')) => subtype(T,T') && subtype(Ts,Ts')
    requires Ts =/=K .Types  [structural]
  rule subtype(.Types,.Types) => true  [structural]
```

### Unsafe Casting

Performs unsafe casting. One should only use it in combination with the subtype relation above.

```
  syntax Val ::= unsafeCast(Val,Type)  [function]

  rule unsafeCast(objectClosure(_,EStack), C:Id)
    => objectClosure(C,EStack)

  rule unsafeCast(methodClosure(_T',C,OL,Ps,S), T) => methodClosure(T,C,OL,Ps,S)

  rule unsafeCast(V:Val, T:Type) => V  requires typeOf(V) ==K T
```

### Generic guard

A generic computational guard: it allows the computation to continue only if a prefix guard evaluates to true.

```
  syntax KItem ::= "true?"
  rule true ~> true? => .  [structural]

endmodule
```

Go to Lesson 3, KOOL typed static.

# KOOL — Typed — Static

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** static semantics of the typed KOOL language.
It extends the static semantics of typed SIMPLE with static semantics for the object-oriented constructs. Also, the static semantics of some of the existing SIMPLE constructs need to change, in order to become more generous with regards to the set of accepted programs, mostly due to subtyping. For example, the assignment construct `x = e` required that both the variable `x` and the expression `e` had the same type in SIMPLE. In KOOL, the type of `e` can be a subtype of the type of `x`.
Specifically, we define the following typing policy for KOOL, everything else not mentioned below borrowing its semantics from SIMPLE:

- Each class `C` yields a homonymous type, which can be explicitly used in programs to type variables and methods, possibly in combination with other types.
- Since now we have user-defined types, we check that each type used in a KOOL program is well-formed, that is, it is constructed only from primitive and class types corresponding to declared classes.
- Class members and their types form a **class type environment**. Each class will have such a type environment. Each member in a class is allowed to be declared only once. Since in KOOL we allow methods to be assigned to fields, we make no distinction between field and method members; in other words, we reject programs declaring both a field and a method with the same name.
- If an identifier is not found in the local type environment, it will be searched for in the current class type environment. If not there, then it will be searched for in its superclass' type

environment. And so on and so forth. If not found until the `Object` class is reached, a typing error is reported.

- The assignment allows variables to be assigned values of more concrete types. The result type of the assignment expression construct will be the (more abstract) type of the assigned variable, and not the (more concrete) type of the expression, like in Java.
- Exceptions are changed (from SIMPLE) to allow throwing and catching only objects, like in Java. Also, unlike in SIMPLE, we do not check whether the type of the thrown exception matches the type of the caught variable, because exceptions can be caught by other `try/catch` blocks, even by ones in other methods. To avoid having to annotate each method with what exceptions it can throw, we prefer to not check the type safety of exceptions (although this is an excellent homework!). We only check that the `try` block type-checks and that the `catch` block type-checks after we bind the caught variable to its claimed type.
- Class declarations are not allowed to have any cycles in their extends relation. Such cycles would lead to non-termination of `new`, as it actually does in the dynamic semantics of KOOL where no such circularity checks are performed.
- Methods overriding other methods should be in the right subtyping relationship with the overridden methods: co-variant in the codomain and contra-variant in the domain.

```
module KOOL-TYPED-STATIC-SYNTAX
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of statically typed KOOL is identical to that of dynamically typed KOOL, they both taking as input the same programs. What differs is the **K** strictness attributes. Like in statically typed SIMPLE, almost all language constructs are strict now, since we want each to type its arguments almost all the time. Like in the other two KOOL definitions, we prefer to copy and then modify/extend the syntax of statically typed SIMPLE.

**Note**: This paragraph is old, now we can do things better. We keep it here only for historical reasons, to see how much we used to suffer

**Annoying K-tool technical problem:**
Currently, the **K** tool treats the "non-terminal" productions (i.e., productions consisting of just one non-terminal), also called "subsorting" production, differently from the other productions. Specifically, it does not insert a node in the AST for them. This may

look desirable at first, but it has a big problem: it does not allow us to treat the subsort differently in different context. For example, since we want *Id* to be both a type (a class name) and a program variable, and since we want expressions to reduce to their types, we are in an impossible situations in which we do not know how to treat an identifier in the semantics: as a type, i.e., a result of computations, or as a program variable, i.e., a non-result. Ideally, we would like to tag the identifiers at parse-time with their local interpretation, but that, unfortunately, is not possible with the current parsing capabilities of the **K** tool, because it requires to insert additional information in the AST for the subsort productions. This will be fixed soon. Until then, unfortunately, we have to do the job of the parser manually. Instead of subsorting *Id* directly to *Type*, we "wrap" it first, say with a wrapper called `class(...)`, exactly how the parser should have done. The major drawback of this is that all the typed KOOL programs in `kool/typed/programs` need to also be modified to always declare class types accordingly. The modified programs can be found in `kool/typed/static/programs`. So make sure you execute the static semantics of KOOL using the modified programs. To avoid seeing the wrapper in the generated documentation, we associate it an "invisibility" latex attribute below.

```
syntax Id ::= "Object" [token] | "Main" [token]
```

## Types

```
syntax Type ::= "void" | "int" | "bool" | "string"
              | Id                      [klabel("class"), avoid]  // see next
              | class(Id)          // explicit KOOL class type
              | Type "[" "]"
              | "(" Type ")"            [bracket]
              > Types "->" Type

syntax Types ::= List{Type,","}
```

## Declarations

```
syntax Param ::= Type Id
syntax Params ::= List{Param,","}

syntax Stmt ::= Type Exps ";" [avoid]
              | Type Id "(" Params ")" Block
              | "class" Id Block
              | "class" Id "extends" Id Block
```

### Expressions

```
syntax FieldReference ::= Exp "." Id          [strict(1)]
syntax ArrayReference ::= Exp "[" Exps "]"    [strict]

syntax Exp ::= Int | Bool | String | Id
             | "this"
             | "super"
             | "(" Exp ")"             [bracket]
             | "++" Exp
             | Exp "instanceOf" Id     [strict(1)]
             | "(" Id ")" Exp          [strict(2)]
             | "new" Id "(" Exps ")"   [strict(2)]
             > Exp "(" Exps ")"        [strict]
             | "-" Exp                 [strict]
             | "sizeOf" "(" Exp ")"    [strict]
             | "read" "(" ")"
             > left:
               Exp "*" Exp             [strict, left]
             | Exp "/" Exp             [strict, left]
             | Exp "%" Exp             [strict, left]
             > left:
               Exp "+" Exp             [strict, left]
             | Exp "-" Exp             [strict, left]
             > non-assoc:
               Exp "<" Exp             [strict, non-assoc]
             | Exp "<=" Exp            [strict, non-assoc]
             | Exp ">" Exp             [strict, non-assoc]
             | Exp ">=" Exp            [strict, non-assoc]
             | Exp "==" Exp            [strict, non-assoc]
             | Exp "!=" Exp            [strict, non-assoc]
             > "!" Exp                 [strict]
             > left:
               Exp "&&" Exp            [strict, left]
             | Exp "||" Exp            [strict, left]
             > "spawn" Block  // not strict: to check return and exceptions
             > Exp "=" Exp             [strict(2), right]

syntax Exp ::= FieldReference | ArrayReference
syntax priority _.__KOOL-TYPED-STATIC-SYNTAX > _[_]_KOOL-TYPED-STATIC-SYNTAX > _(_)_KOOL-T

syntax Exps ::= List{Exp,","}          [strict]
```

### Statements

```
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt ::= Block
                | Exp ";"                                  [strict]
                | "if" "(" Exp ")" Block "else" Block      [avoid, strict]
                | "if" "(" Exp ")" Block
                | "while" "(" Exp ")" Block                [strict]
                | "for" "(" Stmt Exp ";" Exp ")" Block
                | "return" Exp ";"                         [strict]
                | "return" ";"
                | "print" "(" Exps ")" ";"                 [strict]
                | "try" Block "catch" "(" Param ")" Block [strict(1)]
                | "throw" Exp ";"                          [strict]
                | "join" Exp ";"                           [strict]
                | "acquire" Exp ";"                        [strict]
                | "release" Exp ";"                        [strict]
                | "rendezvous" Exp ";"                     [strict]

  syntax Stmt ::= Stmt Stmt                              [seqstrict, right]
```

### Desugaring macros

```
  rule if (E) S => if (E) S else {}                                        [macro]
  rule for(Start Cond; Step) {S:Stmt} => {Start while(Cond){S Step;}}  [macro]
  rule T:Type E1:Exp, E2:Exp, Es:Exps; => T E1; T E2, Es;                  [macro-rec]
  rule T:Type X:Id = E; => T X; X = E;                                     [macro]

  rule class C:Id S => class C extends Object S

endmodule
```

### Static semantics

We first discuss the configuration, then give the static semantics taken over unchanged from SIMPLE, then discuss the static semantics of SIMPLE syntactic constructs that needs to change, and in the end we discuss the static semantics and additional checks specifically related to the KOOL proper syntax.

```
module KOOL-TYPED-STATIC
  imports KOOL-TYPED-STATIC-SYNTAX
  imports DOMAINS
```

## Configuration

The configuration of our type system consists of a `tasks` cell with the same meaning like in statically typed SIMPLE, of an `out` cell streamed to the standard output that will be used to display typing error messages, and of a cell `classes` holding data about each class in a separate `class` cell. The `task` cells now have two additional optional subcells, namely `ctenvT` and `inClass`. The former holds a temporary class type environment; its contents will be transferred into the `ctenv` cell of the corresponding class as soon as all the fields and methods in the task are processed. In fact, there will be three types of tasks in the subsequent semantics, each determined by the subset of cells that it holds:

1. **Main task**, holding only a `k` cell holding the original program as a set of classes. The role of this task is to process each class, generating a class task (see next) for each.
2. **Class task**, holding `k`, `ctenvT`, and `inClass` subcells. The role of this task type is to process a class' contents, generating a class type environment in the `ctenvT` cell and a method task (see next) for each method in the class. To avoid interference with object member lookup rules below, it is important to add the class type environment to a class atomically; this is the reason for which we use `ctenvT` temporary cells within class tasks (instead of adding each member incrementally to the class' type environment).
3. **Method task**, holding `k`, `tenv` and `return` cells. These tasks are similar to SIMPLE's function tasks, so we do not discuss them here any further.

Each `class` cell hods its name (in the `className` cell) and the name of the class it extends (in the `extends` cell), as well as its type environment (in the `ctenv` cell) and the set of all its superclasses (in the `extendsAll` cell). The later is useful for example for checking whether there are cycles in the class extends relation.

```
configuration <T multiplicity="?" color="yellow">
                <tasks color="orange" multiplicity="?">
                  <task multiplicity="*" color="yellow">
                    <k color="green"> $PGM:Stmt </k>
                    <tenv multiplicity="?" color="cyan"> .Map </tenv>
                    <ctenvT multiplicity="?" color="blue"> .Map </ctenvT>
                    <returnType multiplicity="?" color="black"> void </returnType>
                    <inClass multiplicity="?" color="Fuchsia"> .K </inClass>
```

```
                    </task>
                  </tasks>
//                  <br/>
                  <classes color="Fuchsia">
                    <classData multiplicity="*">
                      <className color="Fuchsia"> Object </className>
                      <baseClass color="Fuchsia"> .K </baseClass>
                      <baseClasses color="Fuchsia"> .Set </baseClasses>
                      <ctenv multiplicity="?" color="blue"> .Map </ctenv>
                    </classData>
                  </classes>
                </T>
                <output color="brown" stream="stdout"> .List </output>
```

## Unchanged semantics from statically typed SIMPLE

The syntax and rules below are borrowed unchanged from statically typed SIMPLE, so we do not discuss them much here.

```
  syntax Exp ::= Type
  syntax Exps ::= Types
  syntax BlockOrStmtType ::= "block" | "stmt"
  syntax Type ::= BlockOrStmtType
  syntax Block ::= BlockOrStmtType
  syntax KResult ::= Type
                   | Types  // TODO: should not be needed


  context _:Type _::Exp[HOLE::Exps];

  rule T:Type E:Exp[int,Ts:Types]; => T[] E[Ts];  [structural]
  rule T:Type E:Exp[.Types]; => T E;          [structural]


  rule <task>... <k> _:BlockOrStmtType </k> <tenv> _ </tenv> ...</task> => .Bag


  rule _:Int => int
  rule _:Bool => bool
  rule _:String => string


  rule <k> X:Id => T ...</k> <tenv>... X |-> T ...</tenv>


  context ++(HOLE => ltype(HOLE))
```

```
rule ++ int => int
rule int + int => int
rule string + string => string
rule int - int => int
rule int * int => int
rule int / int => int
rule int % int => int
rule - int => int
rule int < int => bool
rule int <= int => bool
rule int > int => bool
rule int >= int => bool
rule T:Type == T => bool
rule T:Type != T => bool
rule bool && bool => bool
rule bool || bool => bool
rule ! bool => bool


rule (T[])[int, Ts:Types] => T[Ts]
rule T:Type[.Types] => T

rule sizeOf(_T[]) => int


rule read() => int

rule print(T:Type, Ts => Ts); when T ==K int orBool T ==K string
rule print(.Types); => stmt


context (HOLE => ltype(HOLE)) = _


rule <k> return; => stmt ...</k> <returnType> _ </returnType>


rule {} => block

rule <task> <k> {S:Stmt} => block ...</k> <tenv> Rho </tenv> R </task>
     (.Bag => <task> <k> S </k> <tenv> Rho </tenv> R </task>)

rule _:Type; => stmt
rule if (bool) block else block => stmt
rule while (bool) block => stmt
```

```
  rule join int; => stmt
  rule acquire _:Type; => stmt
  rule release _:Type; => stmt
  rule rendezvous _:Type; => stmt

  syntax Stmt ::= BlockOrStmtType
  rule _:BlockOrStmtType _:BlockOrStmtType => stmt
```

## Unchanged auxiliary operations from dynamically typed SIMPLE

```
  syntax Stmt ::= mkDecls(Params)  [function]
  rule mkDecls(T:Type X:Id, Ps:Params) => T X; mkDecls(Ps)
  rule mkDecls(.Params) => {}

  syntax LValue ::= Id
                  | FieldReference
                  | ArrayReference
  syntax Exp ::= LValue

  syntax Exp ::= ltype(Exp)
// We would like to say:
//  context ltype(HOLE:LValue)
// but we currently cannot type the HOLE
  context ltype(HOLE) when isLValue(HOLE)

// OLD approach:
//  syntax Exp ::= ltype(Exp)  [function]
//  rule ltype(X:Id) => X
//  rule ltype(E:Exp [Es:Exps]) => E[Es]

  syntax Types ::= getTypes(Params)  [function]
  rule getTypes(T:Type _:Id) => T, .Types
  rule getTypes(T:Type _:Id, P, Ps) => T, getTypes(P,Ps)
  rule getTypes(.Params) => void, .Types
```

## Changes to the existing statically typed SIMPLE semantics

Below we give the new static semantics for language constructs that come from SIMPLE, but whose SIMPLE static semantics was too restrictive or too permissive and thus had to change.

## Local variable declaration

Since we can define new types in KOOL (corresponding to classes), the variable declaration needs to now check that the claimed types exist.

The operation `checkType`, defined at the end of this module, checks whether the argument type is correct (it actually works with lists of types as well).

```
  rule <k> T:Type X:Id; => checkType(T) ~> stmt ...</k>
       <tenv> Rho => Rho[X <- T] </tenv>
```

## Class member declaration

In class tasks, variable declarations mean class member declarations. Since we reduce method declarations to variable declarations (see below), a variable declaration in a class task can mean either a field or a method declaration. Unlike local variable declarations, which can shadow previous homonymous local or member declarations, member declarations are regarded as a set, so we disallow multiple declarations for the same member (one could improve upon this, like in Java, by treating members with different types or number of arguments as different, etc., but we do not do it here). We also issue an error message if one attempts to redeclare the same class member. The framed variable declaration in the second rule below should be read "stuck". In fact, it is nothing but a unary operation called `stuck`, which takes a **K**-term as argument and does nothing with it; this `stuck` operation is displayed as a frame in this PDF document because of its latex attribute (see the ASCII .k file, at the end of this module).

```
  rule <k> T:Type X:Id; => checkType(T) ~> stmt ...</k>
       <ctenvT> Rho (.Map => X |-> T) </ctenvT>
    when notBool(X in keys(Rho))

  rule <k> T:Type X:Id; => stuck(T X;) ...</k>
       <ctenvT>... X |-> _ ...</ctenvT>
       <inClass> C:Id </inClass>
//       <br/>
       <output>... .List => ListItem("Member \"" +String Id2String(X)
                                +String "\" declared twice in class \""
                                +String Id2String(C) +String "\"!\n") </output>
    [structural]
```

## Method declaration

A method declaration requires two conceptual checks to be performed: first, that the method's type is consistent with the type of the homonymous method that it overrides, if any; and second, that its body types correctly. At the same time, it should also be added to the type environment of its class. The first conceptual task is performed using the `checkMethod` operation defined below, and the second

by generating a corresponding method task. To add it to the class type environment, we take advantage of the fact that KOOL is higher order and reduce the problem to a field declaration problem, which we have already defined. The role of the `ctenvT` cell in the rule below is to structurally ensure that the method declaration takes place in a class task (we do not want to allow methods to be declared, for example, inside other methods).

```
  rule <k> T:Type F:Id(Ps:Params) S
        => checkMethod(F, getTypes(Ps)->T, C')
           ~> getTypes(Ps)->T F; ...</k>
//       <br/>
       <inClass> C </inClass>
       <ctenvT> _ </ctenvT> // to ensure we are in a class pass
       <className> C </className>
       <baseClass> C' </baseClass>
//       <br/>
       (.Bag => <task>
                <k> mkDecls(Ps) S </k>
                <inClass> C </inClass>
                <tenv> .Map </tenv>
                <returnType> T </returnType>
              </task>)
    [structural]
```

## Assignment

A more concrete value is allowed to be assigned to a more abstract variable. The operation `checkSubtype` is defined at the end of the module and it also works with pairs of lists of types.

```
  rule T:Type = T':Type => checkSubtype(T', T) ~> T
```

## Method invocation and return

Methods can be applied on values of more concrete types than their arguments:

```
  rule (Ts:Types -> T:Type) (Ts':Types) => checkSubtype(Ts',Ts) ~> T
```

Similarly, we allow values of more concrete types to be returned by methods:

```
  rule <k> return T:Type; => checkSubtype(T,T') ~> stmt ...</k>
       <returnType> T':Type </returnType>
```

## Exceptions

Exceptions can throw and catch values of any types. Since unlike in Java KOOL's methods do not declare the exception types that they can throw, we cannot test the full type safety of exceptions. Instead, we only check that the `try` and the `catch` statements type correctly.

```
  rule try block catch(T:Type X:Id) S => {T X; S}  [structural]
  rule throw _T:Type ; => stmt
```

## Spawn

The spawned cell needs to also be passed the parent's class.

```
// explain why

  rule <k> spawn S:Block => int ...</k>
       <tenv> Rho </tenv>
       <inClass> C </inClass>
       (.Bag => <task>
                  <k> S </k>
                  <tenv> Rho </tenv>
                  <inClass> C </inClass>
                </task>)
```

## Semantics of the new KOOL constructs

## Class declaration

We process each class in the main task, adding the corresponding data into its `class` cell and also adding a class task for it. We also perform some well-formedness checks on the class hierarchy.

**Initiate class processing**
We create a class cell and a class task for each task. Also, we start the class task with a check that the class it extends is declared (this delays the task until that class is processed using another instance of this rule).

```
// There seems to be some error with the configuration concretization,
// as the rule below does not work when rewriting . to both the task
// and the class cells; I had to include two separate . rewrites

// TODO: the following fails krun; see #2117
  rule <task> <k> class C:Id extends C':Id { S:Stmt } => stmt ...</k> </task>
       (.Bag => <classData>...
                  <className> C </className>
                  <baseClass> C' </baseClass>
```

```
            ...</classData>)
//       <br/>
       (.Bag => <task>
                <k> checkType(class(C')) ~> S </k>
                <inClass> C </inClass>
                <ctenvT> .Map </ctenvT>
              </task>)
    [structural]

// You may want to try the thing below, but that failed, too
/*
syntax Type ::= "stmtStop"

  rule <tasks>...
       <task> <k> class C:Id extends C':Id { S:Stmt } => stmtStop ...</k> </task>
       (.Bag => <task>
                <k> checkType(class(C')) ~> S </k>
                <inClass> C </inClass>
                <ctenvT> .Map </ctenvT>
              </task>)
       ...</tasks>
       <classes>...
       .Bag => <classData>...
               <className> C </className>
               <baseClass> C' </baseClass>
            ...</classData>
       ...</classes>
//       <br/>
  [structural]
*/
```

## Check for unique class names

```
  rule (<T>...
          <className> C </className>
          <className> C </className>
         ...</T> => .Bag)
       <output>... .List => ListItem("Class \"" +String Id2String(C)
                                    +String "\" declared twice!\n") </output>
    [structural]
```

**Check for cycles in class hierarchy**

We check for cycles in the class hierarchy by transitively closing the class extends relation using the `extendsAll` cells, and checking that a class will never appear in its own `extendsAll` cell. The first rule below initiates the transitive closure of the

superclass relation, the second transitively closes it, and the third checks for cycles.

```
rule <baseClass> C </baseClass>
     <baseClasses> .Set => SetItem(C) </baseClasses>  [structural]

rule <classData>...
       <baseClasses> SetItem(C) Cs:Set (.Set => SetItem(C')) </baseClasses>
     ...</classData>
     <classData>... <className>C</className> <baseClass>C'</baseClass> ...</classData>
  when notBool(C' in (SetItem(C) Cs))  [structural]

rule (<T>...
        <className> C </className>
        <baseClasses>... SetItem(C) ...</baseClasses>
       ...</T> => .Bag)
     <output>... .List => ListItem("Class \"" +String Id2String(C)
                                  +String "\" is in a cycle!\n") </output>
  [inheritance-cycle]
```

## New

To type `new` we only need to check that the class constructor can be called with arguments of the given types, so we initiate a call to the constructor method in the corresponding class. If that succeeds, meaning that it types to `stmt`, then we discard the `stmt` type and produce instead the corresponding class type of the new object. The auxiliary `discard` operation is defined also at the end of this module.

```
rule new C:Id(Ts:Types) => class(C) . C (Ts) ~> discard ~> class(C)
```

## Self reference

The typing rule for `this` is straightforward: reduce to the current class type.

```
rule <k> this => class(C) ...</k>
     <inClass> C:Id </inClass>
```

## Super

Similarly, `super` types to the parent class type.
Note that for typing concerns, super can be considered as an object (recall that this was not the case in the dynamic semantics).

```
 rule <k> super => class(C') ...</k>
      <inClass> C:Id </inClass>
```

```
        <className> C </className>
        <baseClass> C':Id </baseClass>
```

### Object member access

There are several cases to consider here. First, if we are in a class task, we should lookup the member into the temporary class type environemnt in cell `ctenvT`. That is because we want to allow initialized field declarations in classes, such as `int x=10;`. This is desugared to a declaration of `x`, which is added to `ctenvT` during the class task processing, followed by an assignment of `x` to 10. In order for the assignment to type check, we need to know that `x` has been declared with type `int`; this information can only be found in the `ctenvT` cell. Second, we should redirect non-local variable lookups in method tasks to corresponding member accesses (the local variables are handled by the rule borrowed from SIMPLE). This is what the second rule below does. Third, we should allow object member accesses as lvalues, which is done by the third rule below. These last two rules therefore ensure that each necessary object member access is explicitly allowed for evaluation. Recall from the annotated syntax module above that the member access operation is strict in the object. That means that the object is expected to evaluate to a class type. The next two rules below define the actual member lookup operation, moving the search to the superclass when the member is not found in the current class. Note that this works because we create the class type environments atomically; thus, a class either has its complete type environment available, in which case these rules can safely apply, or its cell `ctenv` is not yet available, in which case these rules have to wait. Finally, the sixth rule below reports an error when the `Object` class is reached.

```
  rule <k> X:Id => T ...</k>
       <ctenvT>... X |-> T ...</ctenvT>

  rule <k> X:Id => this . X ...</k>
       <tenv> Rho </tenv>
    when notBool(X in keys(Rho))

// OLD approach:
//  rule ltype(E:Exp . X:Id) => E . X  [structural]

  rule <k> class(C:Id) . X:Id => T ...</k>
       <className> C </className>
       <ctenv>... X |-> T:Type ...</ctenv>
```

```
  rule <k> class(C1:Id => C2) . X:Id ...</k>
       <className> C1 </className>
       <baseClass> C2:Id </baseClass>
       <ctenv> Rho </ctenv>
    when notBool(X in keys(Rho))  [structural]

  rule <k> class(Object) . X:Id => stuck(class(Object) . X) ...</k>
       <inClass> C:Id </inClass>
//      <br/>
       <output>... .List => ListItem("Member \"" +String Id2String(X)
                            +String "\" not declared! (see class \""
                            +String Id2String(C) +String "\")\n") </output>
    [structural]
```

## Instance of and casting

As it is hard to check statically whether casting is always safe, the programmer is simply trusted from a typing perspective. We only do some basic upcasting and downcasting checks, to reject casts which will absolutely fail. However, dynamic semantics or implementations of the language need to insert runtime checks for downcasting to be safe.

```
  rule class(_C1:Id) instanceOf _C2:Id => bool
  rule (C:Id) class(C) => class(C)
  rule <k> (C2:Id) class(C1:Id) => class(C2) ...</k>
       <className> C1 </className>
       <baseClasses>...SetItem(C2)...</baseClasses>    // upcast
  rule <k> (C2:Id) class(C1:Id) => class(C2) ...</k>
       <className> C2 </className>
       <baseClasses>...SetItem(C1)...</baseClasses>    // downcast
  rule <k> (C2) class(C1:Id) => stuck((C2) class(C1)) ...</k>
       <classData>...
         <className> C1 </className>
         <baseClasses> S1 </baseClasses>
       ...</classData>
       <classData>...
         <className> C2 </className>
         <baseClasses> S2 </baseClasses>
       ...</classData>
       <output>... .List => ListItem("Classes \"" +String Id2String(C1)
                            +String "\" and \"" +String Id2String(C2)
                            +String "\" are incompatible!\n") </output>
    when notBool(C1 in S2) andBool notBool(C2 in S1)
    [transition]
    // ugly solution to avoid non-confluence (rule may apply before
```

```
    // extendsAll is populated); strategies will solve the problem nicely.
```

### Cleanup tasks

Finally, we need to clean up the terminated tasks. Each of the three types of tasks is handled differently. The main task is replaced by a method task holding `new main();`, which will ensure that a `main` class with a `main()` method actually exists (first rule below). A class task moves its temporary class type environment into its class' cell, and then it dissolves itself (second rule). A method task simply dissolves when terminated (third rule); the presence of the `tenv` cell in that rule ensures that that task is a method task.
Finally, when all the tasks are cleaned up, we can also remove the `tasks` cell, issuing a corresponding message. Note that checking for cycles or duplicate methods can still be performed after the `tasks` cell has been removed.

```
// discard main task when done, issuing a "new main();" command to
// make sure that the class main and the method main() are declared.

  rule <task> <k> stmt => new Main(.Exps); </k>
              (.Bag => <tenv> .Map </tenv>
                    <returnType> void </returnType>
                    <inClass> Main </inClass>)
       </task>
    [structural]

// discard class task when done, adding a ctenv in class

  rule (<task>
          <k> stmt </k>
          <ctenvT> Rho </ctenvT>
          <inClass> C:Id </inClass>
        </task> => .Bag)
        <className> C </className>
        (.Bag => <ctenv> Rho </ctenv>)
    [structural]

// discard method task when done

  rule <task>...
         <k> stmt </k>
         <tenv> _ </tenv>  // only to ensure that this is a method task
       ...</task> => .Bag
    [structural]
```

```
// cleanup tasks and output a success message when done

  rule (<T>... <tasks> .Bag </tasks> ...</T> => .Bag)
       <output>... .List => ListItem("Type checked!\n") </output>
    [structural]
```

## KOOL-specific auxiliary declarations and operations

### Subtype checking

The subclass relation introduces a subtyping relation.

```
  syntax KItem ::= checkSubtype(Types,Types)

  rule checkSubtype(T:Type, T) => .  [structural]

  rule <k> checkSubtype(class(C:Id), class(C':Id)) => . ...</k>
       <className> C </className>
       <baseClasses>... SetItem(C') ...</baseClasses>  [structural]

  rule checkSubtype(Ts1->T2,Ts1'->T2')
    => checkSubtype(((T2)::Type,Ts1'),((T2')::Type,Ts1))  [structural]

// note that the following rule would be wrong!
//  rule checkSubtype(T[],T'[]) => checkSubtype(T,T')

  rule checkSubtype((T:Type,Ts),(T':Type,Ts'))
    => checkSubtype(T,T') ~> checkSubtype(Ts,Ts')
    when Ts =/=K .Types  [structural]

  rule checkSubtype(.Types,.Types) => .  [structural]
  rule checkSubtype(.Types,void) => .  [structural]
```

### Checking well-formedness of types

Since now any *Id* can be used as the type of a class, we need to check that the types used in the program actually exists

```
  syntax KItem ::= checkType(Types)

  rule checkType(T:Type,Ts:Types) => checkType(T) ~> checkType(Ts)
    when Ts =/=K .Types  [structural]
  rule checkType(.Types) => .  [structural]
  rule checkType(int) => .  [structural]
  rule checkType(bool) => .  [structural]
  rule checkType(string) => .  [structural]
```

```
  rule checkType(void) => .  [structural]
  rule <k> checkType(class(C:Id)) => . ...</k> <className> C </className>
    [structural]
  rule checkType(class(Object)) => .  [structural]
  rule checkType(Ts:Types -> T:Type) => checkType(T,Ts)  [structural]
  rule checkType(T:Type[]) => checkType(T)  [structural]
```

### Checking correct overiding of methods

The `checkMethod` operation below searches to see whether the current method overrides some other method in some superclass. If yes, then it issues an additional check that the new method's type is more concrete than the overridden method's. The types `T` and `T'` below can only be function types. See the definition of `checkSubtype` on function types at the end of this module (it is co-variant in the codomain and contra-variant in the domain).

```
  syntax KItem ::= checkMethod(Id,Type,Id)

  rule <k> checkMethod(F:Id, T:Type, C:Id) => checkSubtype(T, T') ...</k>
       <className> C </className>
       <ctenv>... F |-> T':Type ...</ctenv>
    [structural]

  rule <k> checkMethod(F:Id, _T:Type, (C:Id => C')) ...</k>
       <className> C </className>
       <baseClass> C':Id </baseClass>
       <ctenv> Rho </ctenv>
    when notBool(F in keys(Rho))  [structural]

  rule checkMethod(_:Id,_,Object) => .
```

### Generic operations which could be part of the K framework

```
  syntax KItem ::= stuck(K)  [latex(\framebox{${#1}$})]

  syntax KItem ::= "discard"
  rule _:KResult ~> discard => .  [structural]

endmodule
```

## FUN — Untyped — Environment

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped FUN language.
FUN is a pedagogical and research language that captures the essence of the functional programming paradigm, extended with several features often encountered in functional programming languages.
Like many functional languages, FUN is an expression language, that is, everything, including the main program, is an expression.
Functions can be declared anywhere and are first class values in the language.
FUN is call-by-value here, but it has been extended (as student homework assignments) with other parameter-passing styles.
To make it more interesting and to highlight some of **K**'s strengths, FUN includes the following features:

- The basic builtin data-types of integers, booleans and strings.
- Builtin lists, which can hold any elements, including other lists. Lists are enclosed in square brackets and their elements are comma-separated; e.g., `[1,2,3]`.
- User-defined data-types, by means of constructor terms. Constructor names start with a capital letter (while any other identifier in the language starts with a lowercase letter), and they can be followed by an arbitrary number of comma-separated arguments enclosed in parentheses; parentheses are not needed when the constructor takes no arguments.
  For example, `Pair(5,7)` is a constructor term holding two numbers, `Cons(1,Cons(2,Cons(3,Nil)))` is a list-like constructor term holding 3 elements, and `Tree(Tree(Leaf(1), Leaf(2)), Leaf(3))` is a tree-like constructor term holding 3 elements.
  In the untyped version of the FUN language, no type checking or inference is performed to ensure that the data constructors are used correctly.
  The execution will simply get stuck when they are misused.
  Moreover, since no type checking is performed, the data-types are not even declared in the untyped version of FUN.
- Functions and `let`/`letrec` binders can take multiple space-separated arguments, but these are desugared to ones that only take one argument, by currying. For example, the expressions

  ```
  fun x y -> x y
  ```

```
let x y = y in x
```

are desugared, respectively, into the following expressions:

```
fun x -> fun y -> x y
let x = fun y -> y in x
```

- Functions can be defined using pattern matching over the available data-types. For example, the program

```
letrec max = fun [h] -> h
             |   [h|t] -> let x = max t
                          in  if h > x then h else x
in max [1, 3, 5, 2, 4, 0, -1, -5]
```

defines a function `max` that calculates the maximum element of a non-empty list, and the function

```
letrec ack = fun Pair(0,n) -> n + 1
             |   Pair(m,0) -> ack Pair(m - 1, 1)
             |   Pair(m,n) -> ack Pair(m - 1, ack Pair(m, n - 1))
in ack Pair(2,3)
```

calculates the Ackermann function applied to a particular pair of numbers. Patterns can be nested. Patterns can currently only be used in function definitions, and not directly in `let`/`letrec` binders.
For example, this is not allowed:

```
letrec Pai(x,y) = Pair(1,2) in x+y
```

But this is allowed:

```
let f Pair(x,y) = x+y in f Pair(1,2)
```

because it is first reduced to

```
let f = fun Pair(x,y) -> x+y in f Pair(1,2)
```

by uncurrying of the `let` binder, and pattern matching is allowed in function arguments.

- We include a `callcc` construct, for two reasons: first, several functional languages support this construct; second, some semantic frameworks have difficulties defining it. Not **K**.
- Finally, we include mutables by means of referencing an expression, getting the reference of a variable, dereferencing and assignment. We include these for the same reasons as above: there are languages which have them, and they are not easy to define in some semantic frameworks.

Like in many other languages, some of FUN's constructs can be desugared into a smaller set of basic constructs. We do that as usual, using macros, and then we only give semantics to the core constructs.

**Note:**
We recommend the reader to first consult the dynamic semantics of the LAMBDA++ language in the first part of the K Tutorial.
To keep the comments below small and focused, we will not re-explain functional or **K** features that have already been explained in there.

## Syntax

```
//require "modules/pattern-matching.k"

module FUN-UNTYPED-COMMON
  imports DOMAINS-SYNTAX
```

FUN is an expression language. The constructs below fall into several categories: names, arithmetic constructs, conventional functional constructs, patterns and pattern matching, data constructs, lists, references, and call-with-current-continuation (callcc).
The arithmetic constructs are standard; they are present in almost all our **K** language definitions. The meaning of FUN's constructs are discussed in more depth when we define their semantics in the next module.

## The Syntactic Constructs

We start with the syntactic definition of FUN names.
We have several categories of names: ones to be used for functions and variables, others to be used for data constructors, others for types and others for type variables. We will introduce them as needed, starting with the former category. We prefer the names of variables and functions to start with lower case letters. We take the freedom to tacitly introduce syntactic lists/sequences for each nonterminal for which we need them:

```
  syntax Name
  syntax Names ::= List{Name,","}
```

Expression constructs will be defined throughtout the syntax module. Below are the very basic ones, namely the builtins, the names, and the parentheses used as brackets for grouping. Lists of expressions are declared strict, so all expressions in the list get evaluated whenever the list is on a position which can be evaluated:

```
  syntax Exp ::= Int | Bool | String | Name
               | "(" Exp ")"                          [bracket]
  syntax Exps  ::= List{Exp,","}                      [strict]
  syntax Val
  syntax Vals ::= List{Val,","}
  syntax Bottom
  syntax Bottoms ::= List{Bottom,","}
```

We next define the syntax of arithmetic constructs, together with their relative priorities and left-/non-associativities. We also tag all these rules with a new tag, "arith", so we can more easily define global syntax priirities later (at the end of the syntax module).

```
  syntax Exp ::= left:
                   Exp "*" Exp                        [strict, arith]
                 | Exp "/" Exp                        [strict, arith]
                 | Exp "%" Exp                        [strict, arith]
                 > left:
                   Exp "+" Exp                        [strict, left, arith]
                 | Exp "^" Exp                        [strict, left, arith]
// left attribute should not be necessary; currently a parsing bug
                 | Exp "-" Exp                        [strict, prefer, arith]
// the "prefer" attribute above is to not parse x-1 as x(-1)
// Due to some parsing problems, we currently cannot add unary minus:
                 | "-" Exp                            [strict, arith]
                 > non-assoc:
                   Exp "<" Exp                        [strict, arith]
                 | Exp "<=" Exp                       [strict, arith]
                 | Exp ">" Exp                        [strict, arith]
                 | Exp ">=" Exp                       [strict, arith]
                 | Exp "==" Exp                       [strict, arith]
                 | Exp "!=" Exp                       [strict, arith]
                 > "!" Exp                            [strict, arith]
                 > Exp "&&" Exp                       [strict(1), left, arith]
                 > Exp "||" Exp                       [strict(1), left, arith]
```

The conditional construct has the expected evaluation strategy, stating that only the first argument is evaluate:

```
  syntax Exp ::= "if" Exp "then" Exp "else" Exp    [strict(1)]
```

FUN's builtin lists are formed by enclosing comma-separated sequences of expressions (i.e., terms of sort `Exps`) in square brackets. The list constructor `cons` adds a new element to the top of the list, `head` and `tail` get the first element and the tail sublist of a list if they exist, respectively, and get stuck otherwise, and `null??` tests whether a list is empty or not; syntactically, these are just expression constants.
In function patterns, we are also going to allow patterns following the usual head/tail notation; for example, the pattern `[x_1,...,x_n|t]` binds `x_1`, ..., `x_n` to the first elements of the matched list, and `t` to the list formed with the remaining elements. We define list patterns as ordinary expression constructs, although we will make sure that we do not give them semantics if they appear in any other place then in a function case pattern.

```
  syntax Exp ::= "[" Exps "]"                              [strict]
               | "cons" |  "head" | "tail" | "null?"
               | "[" Exps "|" Exp "]"
  syntax Val ::= "[" Vals "]"
```

Data constructors start with capital letters and they may or may not have arguments. We need to use the attribute "prefer" to make sure that, e.g., `Cons(a)` parses as constructor `Cons` with argument `a`, and not as the expression `Cons` (because constructor names are also expressions) regarded as a function applied to the expression `a`. Also, note that the constructor is strict in its second argument, because we want to evaluate its arguments but not the constuctor name itsef.

```
  syntax ConstructorName
  syntax Exp ::= ConstructorName
               | ConstructorName "(" Exps ")"    [prefer, strict(2)]
  syntax Val ::= ConstructorName "(" Vals ")"
```

A function is essentially a `|`-separated ordered sequence of cases, each case of the form `pattern -> expression`, preceded by the language construct `fun`. Patterns will be defined shortly, both for the builtin lists and for user-defined constructors. Recall that the syntax we define in **K** is not meant to serve as a ultimate parser for the defined language, but rather as a convenient notation for **K** abstract syntax trees, which we prefer when we write the semantic rules. It is therefore often the case that we define a more "generous" syntax than we want to allow programs to use. We do it here, too. Specifically, the syntax of `Cases` below allows any expressions to appear as pattern. This syntactic relaxation permits many wrong programs to be parsed, but that is not a problem because we are not going to give semantics to wrong combinations, so those programs will get stuck; moreover, our type inferencer will reject those programs anyway. Function application is just concatenation of expressions, without worrying about type correctness. Again, the type system will reject type-incorrect programs.

```
  syntax Exp ::= "fun" Cases
               | Exp Exp                                [strict, left]
// NOTE: We would like eventually to also have Exp "(" Exps ")
  syntax Case  ::= Exp "->" Exp
  syntax Cases ::= List{Case, "|"}
```

The `let` and `letrec` binders have the usual syntax and functional meaning. We allow multiple `and`-separated bindings. Like for the function cases above, we allow a more generous syntax for the left-hand sides of bindings, noting that the semantics will get stuck on incorrect bindings and that the type system will reject those programs.

```
  syntax Exp ::= "let" Bindings "in" Exp
               | "letrec" Bindings "in" Exp                  [prefer]
// The "prefer" attribute for letrec currently needed due to tool bug,
// to make sure that "letrec" is not parsed as "let rec".
  syntax Binding  ::= Exp "=" Exp
  syntax Bindings ::= List{Binding,"and"}
```

References are first class values in FUN. The construct `ref` takes an expression, evaluates it, and then it stores the resulting value at a fresh location in the store and returns that reference. Syntactically, `ref` is just an expression constant. The construct `&` takes a name as argument and evaluates to a reference, namely the store reference where the variable passed as argument stores its value; this construct is a bit controversial and is further discussed in the environment-based semantics of the FUN language, where we desugar `ref` to it. The construct `@` takes a reference and evaluates to the value stored there. The construct `:=` takes two expressions, the first expected to evaluate to a reference; the value of its second argument will be stored at the location to which the first points (the old value is thus lost). Finally, since expression evaluation now has side effects, it makes sense to also add a sequential composition construct, which is sequentially strict. This evaluates to the value of its second argument; the value of the first argument is lost (which has therefore been evaluated only for its side effects.

```
  syntax Exp ::= "ref"
               | "&" Name
               | "@" Exp                                     [strict]
               | Exp ":=" Exp                                [strict]
               | Exp ";" Exp                        [strict(1), right]
```

Call-with-current-continuation, named `callcc` in FUN, is a powerful control operator that originated in the Scheme programming language, but it now exists in many other functional languages. It works by evaluating its argument, expected to evaluate to a function, and by passing the current continuation, or evaluation context (or computation, in **K** terminology), as a special value to it. When/If this special value is invoked, the current context is discarded and replaced with the one held by the special value and the computation continues from there. It is like taking a snapshot of the execution context at some moment in time and then, when desired, being able to get back in time to that point. If you like games, it is like saving the game now (so you can work on your homework!) and then continuing the game tomorrow or whenever you wish. To issustrate the strength of `callcc`, we also allow exceptions in FUN by means of a conventional `try-catch` construct, which will desugar to `callcc`. We also need to introduce the special expression contant `throw`, but we need to

use it as a function argument name in the desugaring macro, so we define it as a name instead of as an expression constant:

```
  syntax Exp ::= "callcc"
               | "try" Exp "catch" "(" Name ")" Exp
  syntax Name ::= "throw" [token]
```

Finally, FUN also allows polymorphic datatype declarations. These will be useful when we define the type system later on.

```
  syntax Exp ::= "datatype" Type "=" TypeCases Exp
// NOTE: In a future version of K, we want the datatype declaration
// to be a construct by itself, but that is not possible currently
// because K's parser wronly identifies the __ operation allowing
// a declaration to appear in front of an expression with the function
// application construct, giving ambiguous parsing errors.
```

We next need to define the syntax of types and type cases that appear in datatype declarations.

Like in many functional languages, type parameters/variables in user-defined types are quoted identifiers.

```
  syntax TypeVar
  syntax TypeVars ::= List{TypeVar,","}
```

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore all the type declations, so here the syntax of types below is only useful for generating the desired parser. To avoid syntactic ambiguities with the arrow construct for function cases, we use the symbol `-->` as a constructor for function types:

```
  syntax TypeName
  syntax Type ::= "int" | "bool" | "string"
                | Type "-->" Type                              [right]
                | "(" Type ")"                                [bracket]
                | TypeVar
                | TypeName              [klabel(TypeName), avoid]
                | Type TypeName   [klabel(Type-TypeName)]
                | "(" Types ")" TypeName                       [prefer]
  syntax Types ::= List{Type,","}
  syntax Types ::= TypeVars

  syntax TypeCase ::= ConstructorName
                    | ConstructorName "(" Types ")"
  syntax TypeCases ::= List{TypeCase,"|"}     [klabel(_|TypeCase_)]
```

## Additional Priorities

```
  syntax priorities @__FUN-UNTYPED-COMMON
                  > ___FUN-UNTYPED-COMMON
                  > arith
                  > _:=__FUN-UNTYPED-COMMON
                  > let_in__FUN-UNTYPED-COMMON
                    letrec_in__FUN-UNTYPED-COMMON
                    if_then_else__FUN-UNTYPED-COMMON
                  > _;__FUN-UNTYPED-COMMON
                  > fun__FUN-UNTYPED-COMMON
                  > datatype_=___FUN-UNTYPED-COMMON
endmodule

module FUN-UNTYPED-MACROS
  imports FUN-UNTYPED-COMMON
```

## Desugaring macros

We desugar the list non-constructor operations to functions matching over list patterns. In order to do that we need some new variables; for those, we follow the same convention like in the **K** tutorial, where we added them as new identifier constructs starting with the character `$`, so we can easily recognize them when we debug or trace the semantics.

```
  syntax Name ::= "$h" | "$t"
  rule head => fun [$h|$t] -> $h                                  [macro]
  rule tail => fun [$h|$t] -> $t                                  [macro]
  rule null? => fun [.Exps] -> true | [$h|$t] -> false            [macro]
```

Multiple-head list patterns desugar into successive one-head patterns:

```
  rule [E1,E2,Es:Exps|T] => [E1|[E2,Es|T]]                        [macro-rec]
```

Uncurrying of multiple arguments in functions and binders:

```
  rule P1 P2 -> E => P1 -> fun P2 -> E                            [macro-rec]
  rule F P = E => F = fun P -> E                                  [macro-rec]
```

We desugar the `try-catch` construct into callcc:

```
  syntax Name ::= "$k" | "$v"
  rule try E catch(X) E'
    => callcc (fun $k -> (fun throw -> E)(fun X -> $k E'))        [macro]
```

For uniformity, we reduce all types to their general form:

```
//  rule TypeName(Tn:TypeName) => (.TypeVars) Tn                  [macro]
  rule `Type-TypeName`(T:Type, Tn:TypeName) => (T) Tn             [macro]
```

The dynamic semantics ignores all the type declarations:

```
  rule datatype _T = _TCs E => E                                  [macro]

endmodule


module FUN-UNTYPED-SYNTAX
  imports FUN-UNTYPED-COMMON
  imports BUILTIN-ID-TOKENS

  syntax Name ::= r"[a-z][_a-zA-Z0-9]*"           [token, prec(2)]
                | #LowerId                        [token]
  syntax ConstructorName ::= #UpperId             [token]
  syntax TypeVar  ::= r"['][a-z][_a-zA-Z0-9]*"    [token]
  syntax TypeName ::= Name                        [token]
endmodule
```

## Semantics

The semantics below is environment-based. A substitution-based definition of FUN is also available, but that drops the `&` construct as explained above.

```
module FUN-UNTYPED
  imports FUN-UNTYPED-COMMON
  imports FUN-UNTYPED-MACROS
  imports DOMAINS
  //imports PATTERN-MATCHING
```

## Configuration

The `k`, `env`, and `store` cells are standard (see, for example, the definition of LAMBDA++ or IMP++ in the first part of the **K** tutorial).

```
  configuration <T color="yellow">
                  <k color="green"> $PGM:Exp </k>
                  <env color="violet"> .Map </env>
                  <store color="white"> .Map </store>
                </T>
```

## Values and results

We only define integers, Booleans and strings as values here, but will add more values later.

```
  syntax Val ::= Int | Bool | String
  syntax Exp ::= Val
```

```
syntax Exps ::= Vals
syntax Vals ::= Bottoms
syntax KResult ::= Val
```

## Lookup

```
rule <k> X:Name => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V ...</store>
```

## Arithmetic expressions

```
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
rule I1 + I2 => I1 +Int I2
rule S1 ^ S2 => S1 +String S2
rule I1 - I2 => I1 -Int I2
rule - I => 0 -Int I
rule I1 < I2 => I1 <Int I2
rule I1 <= I2 => I1 <=Int I2
rule I1 > I2 => I1 >Int I2
rule I1 >= I2 => I1 >=Int I2
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E
```

## Conditional

```
rule if  true then E else _ => E
rule if false then _ else E => E
```

## Lists

We have already declared the syntactic list of expressions strict, so we can assume that all the elements that appear in a FUN list are evaluated. The only thing left to do is to state that a list of values is a value itself, that is, that the list square-bracket construct is indeed a constructor, and to give the semantics of `cons`. Since `cons` is a builtin function and is expected to take two arguments, we have to also state that `cons` itself is a value (specifically, a function/closure

value, but we do not need that level of detail here), and also that `cons` applied to a value is a value (specifically, it would be a function/closure value that expects the second, list argument):

```
rule isVal(cons) => true
rule isVal(cons _V:Val) => true
rule cons V:Val [Vs:Vals] => [V,Vs]
```

## Data Constructors

Constructors take values as arguments and produce other values:

```
syntax Val ::= ConstructorName
```

## Functions and Closures

Like in the environment-based semantics of LAMBDA++ in the first part of the **K** tutorial, functions evaluate to closures. A closure includes the current environment besides the function contents; the environment will be used at execution time to lookup all the variables that appear free in the function body (we want static scoping in FUN).

```
syntax Val ::= closure(Map,Cases)
rule <k> fun Cases => closure(Rho,Cases) ...</k>  <env> Rho </env>
```

**Note:** The reader may want to get familiar with how the pre-defined pattern matching works before proceeding. The best way to do that is to consult `k/include/modules/pattern-matching.k`.

We distinguish two cases when the closure is applied. If the first pattern matches, then we pick the first case: switch to the closed environment, get the matching map and bind all its variables, and finally evaluate the function body of the first case, making sure that the environment is properly recovered afterwards. If the first pattern does not match, then we drop it and thus move on to the next one.

```
rule (. => getMatching(P, V)) ~> closure(_, P->_ | _) V:Val
rule <k> matchResult(M:Map) ~> closure(Rho, _->E | _) _
         => bindMap(M) ~> E ~> setEnv(Rho') ...</k>
     <env> Rho' => Rho </env>
rule (matchFailure => .) ~> closure(_, (_->_ | Cs:Cases => Cs)) _
//  rule <k> closure(Rho, P->E | _) V:Val
//           => bindMap(getMatching(P,V)) ~> E ~> setEnv(Rho') ...</k>
//       <env> Rho' => Rho </env>  when isMatching(P,V)
//  rule closure(_, (P->_ | Cs:Cases => Cs)) V:Val  when notBool isMatching(P,V)
```

## Let and Letrec

To highlight the similarities and differences between `let` and `letrec`, we prefer to give them direct semantics instead of to desugar them like in LAMBDA. See the formal definitions of `bindTo`, `bind`, and `assignTo` at the end of this module. Informally, `bindTo(Xs, Es)` first evaluates the expressions `Es` in `Exps` in the current environment (i.e., it is strict in its second argument), then it binds the variables in `Xs` in `Names` to new locations and adds those bindings to the environment, and finally writes the values previously obtained after evaluating the expressions `Es` to those new locations; `bind(Xs)` does only the bindings of `Xs` to new locations and adds those bindings to the environment; and `assignTo(Xs,Es)` evaluates the expressions `Es` in the current environment and then it writes the resulting values to the locations to which the variables `Xs` are already bound to in the environment.

Therefore, `let Xs = Es in E` first evaluates `Es` in the current environment, then adds new bindings for `Xs` to fresh locations in the environment, then writes the values of `Es` to those locations, and finally evaluates `E` in the new environment, making sure that the environment is properly recovered after the evaluation of `E`. On the other hand, `letrec` does the same things but in a different order: it first adds new bindings for `Xs` to fresh locations in the environment, then it evaluates `Es` in the new environment, then it writes the resulting values to their corresponding locations, and finally it evaluates `E` and recovers the environment. The crucial difference is that the expressions `Es` now see the locations of the variables `Xs` in the environment, so if they are functions, which is typically the case with `letrec`, their closures will encapsulate in their environments the bindings of all the bound variables, including themselves (thus, we may have a closure value stored at location `L`, whose environment contains a binding of the form `F  L`; this way, the closure can invoke itself).

```
rule <k> let Bs in E
       => bindTo(names(Bs),exps(Bs)) ~> E ~> setEnv(Rho) ...</k>
     <env> Rho </env>

rule <k> letrec Bs in E
       => bind(names(Bs))~>assignTo(names(Bs),exps(Bs))~>E~>setEnv(Rho)...</k>
     <env> Rho </env>
```

Recall that our syntax allows `let` and `letrec` to
take any expression in place of its binding. This allows us to use
the already existing function application construct to bind names to
functions, such as, e.g., `let x y = y in ...`.
The desugaring macro in the syntax module uncurries such declarations,
and then the semantic rules above only work when the remaining
bindings are identifiers, so the semantics will get stuck on programs
that misuse the `let` and `letrec` binders.

## References

The semantics of references is self-explanatory, except maybe for the
desugaring rule of `ref`, which is further discussed. Note
that `&X` grabs the location of `X` from the environment.
Sequential composition, which is needed only to accumulate the
side effects due to assignments, was strict in the first argument.
Once evaluated, its first argument is simply discarded:

```
syntax Name ::= "$x"
rule ref => fun $x -> & $x                                    [macro]
rule <k> & X => L ...</k>  <env>... X |-> L ...</env>
rule <k> @ L:Int => V:Val ...</k>  <store>... L |-> V ...</store>
rule <k> L:Int := V:Val => V ...</k>  <store>... L |-> (_=>V) ...</store>
rule _V:Val; E => E
```

The desugaring rule of `ref` (first rule above) works
because `&` takes a variable and returns its location (like in C).
Note that some "pure" functional programming researchers strongly dislike
the `&` construct, but favor `ref`. We refrain from having
a personal opinion on this issue here, but support `&` in the
environment-based definition of FUN because it is, technically speaking,
more powerful than `ref`. From a language design perspective, it
would be equally easy to drop `&` and instead give a direct
semantics to `ref`. In fact, this is precisely what we do in the
substitution-based definition of FUN, because there appears to be no way
to give a substitution-based definition to the `&` construct.

## Callcc

As we know it from the LAMBDA++ tutorial, call-with-current-continuation
is quite easy to define in **K**. We first need to define a special
value wrapping an execution context, that is, an environment saying
where the variables should be looked up, and a computation structure
saying what is left to execute (in a substitution-based definition,
this special value would be even simpler, as it would only need to
wrap the computation structure---see, for example, the
substitution-based semantics of LAMBDA++ in the the first part of the

**K** tutorial, or the substitution-based definition of FUN). Then `callcc` creates such a value containing the current environment and the current remaining computation, and passes it to its argument function. When/If invoked, the special value replaces the current execution context with its own and continues the execution normally.

```
syntax Val ::= cc(Map,K)
rule isVal(callcc) => true
rule <k> (callcc V:Val => V cc(Rho,K)) ~> K </k>  <env> Rho </env>
rule <k> cc(Rho,K) V:Val ~> _ => V ~> K </k>  <env> _ => Rho </env>
```

## Auxiliary operations

## Environment recovery

The environment recovery operation is the same as for the LAMBDA++ language in the **K** tutorial and many other languages provided with the **K** distribution. The first "anywhere" rule below shows an elegant way to achieve the benefits of tail recursion in **K**.

```
syntax KItem ::= setEnv(Map)  // TODO: get rid of env
//rule (setEnv(_) => .) ~> setEnv(_)  [anywhere]
rule <k> _:Val ~> (setEnv(Rho) => .) ...</k> <env> _ => Rho </env>
  [structural]
```

### `bindTo`, `bind` and `assignTo`

The meaning of these operations has already been explained when we discussed the `let` and `letrec` language constructs above.

```
syntax KItem ::= bindTo(Names,Exps)         [strict(2)]
               | bindMap(Map)
               | bind(Names)

rule (. => getMatchingAux(Xs,Vs)) ~> bindTo(Xs:Names,Vs:Vals)
rule matchResult(M:Map) ~> bindTo(_:Names, _:Vals) => bindMap(M)

rule bindMap(.Map) => .  [structural]
rule <k> bindMap((X:Name |-> V:Val => .Map) _:Map) ...</k>
     <env> Rho => Rho[X <- !L:Int] </env>
     <store>... .Map => !L |-> V ...</store>
  [structural]

rule bind(.Names) => .                  [structural]
rule <k> bind(X:Name,Xs => Xs) ...</k>
     <env> Rho => Rho[X <- !_L:Int] </env>
```

```
  [structural]

syntax KItem ::= assignTo(Names,Exps)  [strict(2)]

rule <k> assignTo(.Names,.Vals) => . ...</k>            [structural]
rule <k> assignTo((X:Name,Xs => Xs),(V:Val,Vs:Vals => Vs)) ...</k>
     <env>... X |-> L ...</env>
     <store>... .Map => L |-> V ...</store>             [structural]
```

## Getters

The following auxiliary operations extract the list of identifiers and of expressions in a binding, respectively.

```
syntax Names ::= names(Bindings)  [function]
rule names(.Bindings) => .Names
rule names(X:Name=_ and Bs) => (X,names(Bs))::Names

syntax Exps ::= exps(Bindings)  [function]
rule exps(.Bindings) => .Exps
rule exps(_:Name=E and Bs) => E,exps(Bs)

/* Extra kore stuff */
syntax KResult ::= Vals
syntax Exps ::= Names
syntax Names ::= Bottoms

/* Matching */
syntax MatchResult ::= getMatching(Exp, Val)                    [function]
                     | getMatchingAux(Exps, Vals)               [function]
                     | mergeMatching(MatchResult, MatchResult)  [function]
                     | matchResult(Map)
                     | "matchFailure"

rule getMatching(C:ConstructorName(Es:Exps), C(Vs:Vals)) => getMatchingAux(Es, Vs)
rule getMatching([Es:Exps], [Vs:Vals])                    => getMatchingAux(Es, Vs)
rule getMatching(C:ConstructorName, C) => matchResult(.Map)
rule getMatching(B:Bool, B)            => matchResult(.Map)
rule getMatching(I:Int, I)             => matchResult(.Map)
rule getMatching(S:String, S)          => matchResult(.Map)
rule getMatching(N:Name, V:Val) => matchResult(N |-> V)
rule getMatching(_, _) => matchFailure        [owise]

rule getMatchingAux((E:Exp, Es:Exps), (V:Val, Vs:Vals)) => mergeMatching(getMatching(E, V)
rule getMatchingAux(.Exps, .Vals)                        => matchResult(.Map)
rule getMatchingAux(_, _) => matchFailure     [owise]
```

```
  rule mergeMatching(matchResult(M1:Map), matchResult(M2:Map)) => matchResult(M1 M2)
    requires intersectSet(keys(M1), keys(M2)) ==K .Set
  //rule mergeMatching(_, _) => matchFailure      [owsie]
  rule mergeMatching(matchResult(_:Map), matchFailure) => matchFailure
  rule mergeMatching(matchFailure, matchResult(_:Map)) => matchFailure
  rule mergeMatching(matchFailure, matchFailure)       => matchFailure
```

Besides the generic decomposition rules for patterns and values, we also want to allow `[head|tail]` matching for lists, so we add the following custom pattern decomposition rule:

```
  rule getMatching([H:Exp | T:Exp], [V:Val, Vs:Vals])
    => getMatchingAux((H, T), (V, [Vs]))
endmodule
```

Go to Lesson 2, FUN untyped, Substitution-Based.

# FUN — Untyped — Environment

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of the untyped FUN language.
FUN is a pedagogical and research language that captures the essence of the functional programming paradigm, extended with several features often encountered in functional programming languages.
Like many functional languages, FUN is an expression language, that is, everything, including the main program, is an expression.
Functions can be declared anywhere and are first class values in the language.
FUN is call-by-value here, but it has been extended (as student homework assignments) with other parameter-passing styles.
To make it more interesting and to highlight some of **K**'s strengths, FUN includes the following features:

- The basic builtin data-types of integers, booleans and strings.
- Builtin lists, which can hold any elements, including other lists. Lists are enclosed in square brackets and their elements are comma-separated; e.g., `[1,2,3]`.
- User-defined data-types, by means of constructor terms. Constructor names start with a capital letter (while any other

identifier in the language starts with a lowercase letter), and they can be followed by an arbitrary number of comma-separated arguments enclosed in parentheses; parentheses are not needed when the constructor takes no arguments.
For example, `Pair(5,7)` is a constructor term holding two numbers, `Cons(1,Cons(2,Cons(3,Nil)))` is a list-like constructor term holding 3 elements, and `Tree(Tree(Leaf(1), Leaf(2)), Leaf(3))` is a tree-like constructor term holding 3 elements.
In the untyped version of the FUN language, no type checking or inference is performed to ensure that the data constructors are used correctly.
The execution will simply get stuck when they are misused.
Moreover, since no type checking is performed, the data-types are not even declared in the untyped version of FUN.

- Functions and `let`/`letrec` binders can take multiple space-separated arguments, but these are desugared to ones that only take one argument, by currying. For example, the expressions

  ```
  fun x y -> x y
  let x y = y in x
  ```

  are desugared, respectively, into the following expressions:

  ```
  fun x -> fun y -> x y
  let x = fun y -> y in x
  ```

- Functions can be defined using pattern matching over the available data-types. For example, the program

  ```
  letrec max = fun [h] -> h
               |   [h|t] -> let x = max t
                            in  if h > x then h else x
  in max [1, 3, 5, 2, 4, 0, -1, -5]
  ```

  defines a function `max` that calculates the maximum element of a non-empty list, and the function

  ```
  letrec ack = fun Pair(0,n) -> n + 1
               |   Pair(m,0) -> ack Pair(m - 1, 1)
               |   Pair(m,n) -> ack Pair(m - 1, ack Pair(m, n - 1))
  in ack Pair(2,3)
  ```

  calculates the Ackermann function applied to a particular pair of numbers.
  Patterns can be nested. Patterns can currently only be used in function definitions, and not directly in `let`/`letrec` binders.
  For example, this is not allowed:

  ```
  letrec Pai(x,y) = Pair(1,2) in x+y
  ```

But this is allowed:

`let f Pair(x,y) = x+y in f Pair(1,2)`

because it is first reduced to

`let f = fun Pair(x,y) -> x+y in f Pair(1,2)`

by uncurrying of the `let` binder, and pattern matching is allowed in function arguments.

- We include a `callcc` construct, for two reasons: first, several functional languages support this construct; second, some semantic frameworks have difficulties defining it. Not **K**.
- Finally, we include mutables by means of referencing an expression, getting the reference of a variable, dereferencing and assignment. We include these for the same reasons as above: there are languages which have them, and they are not easy to define in some semantic frameworks.

Like in many other languages, some of FUN's constructs can be desugared into a smaller set of basic constructs. We do that as usual, using macros, and then we only give semantics to the core constructs.

**Note:**
We recommend the reader to first consult the dynamic semantics of the LAMBDA++ language in the first part of the K Tutorial.
To keep the comments below small and focused, we will not re-explain functional or **K** features that have already been explained in there.

## Syntax

```
//require "modules/pattern-matching.k"

module FUN-UNTYPED-COMMON
  imports DOMAINS-SYNTAX
```

FUN is an expression language. The constructs below fall into several categories: names, arithmetic constructs, conventional functional constructs, patterns and pattern matching, data constructs, lists, references, and call-with-current-continuation (callcc).
The arithmetic constructs are standard; they are present in almost all our **K** language definitions. The meaning of FUN's constructs are discussed in more depth when we define their semantics in the next module.

## The Syntactic Constructs

We start with the syntactic definition of FUN names.
We have several categories of names: ones to be used for functions and variables, others to be used for data constructors, others for types and others for type variables. We will introduce them as needed, starting with the former category. We prefer the names of variables and functions to start with lower case letters. We take the freedom to tacitly introduce syntactic lists/sequences for each nonterminal for which we need them:

```
syntax Name
syntax Names ::= List{Name,","}
```

Expression constructs will be defined throughtout the syntax module. Below are the very basic ones, namely the builtins, the names, and the parentheses used as brackets for grouping. Lists of expressions are declared strict, so all expressions in the list get evaluated whenever the list is on a position which can be evaluated:

```
syntax Exp ::= Int | Bool | String | Name
             | "(" Exp ")"                        [bracket]
syntax Exps  ::= List{Exp,","}                    [strict]
syntax Val
syntax Vals ::= List{Val,","}
syntax Bottom
syntax Bottoms ::= List{Bottom,","}
```

We next define the syntax of arithmetic constructs, together with their relative priorities and left-/non-associativities. We also tag all these rules with a new tag, "arith", so we can more easily define global syntax priirities later (at the end of the syntax module).

```
syntax Exp ::= left:
                 Exp "*" Exp                      [strict, arith]
             | Exp "/" Exp                        [strict, arith]
             | Exp "%" Exp                        [strict, arith]
             > left:
                 Exp "+" Exp                      [strict, left, arith]
             | Exp "^" Exp                        [strict, left, arith]
// left attribute should not be necessary; currently a parsing bug
             | Exp "-" Exp                        [strict, prefer, arith]
// the "prefer" attribute above is to not parse x-1 as x(-1)
// Due to some parsing problems, we currently cannot add unary minus:
             | "-" Exp                            [strict, arith]
             > non-assoc:
                 Exp "<" Exp                      [strict, arith]
             | Exp "<=" Exp                       [strict, arith]
             | Exp ">" Exp                        [strict, arith]
```

```
  | Exp ">=" Exp                        [strict, arith]
  | Exp "==" Exp                        [strict, arith]
  | Exp "!=" Exp                        [strict, arith]
  > "!" Exp                             [strict, arith]
  > Exp "&&" Exp                        [strict(1), left, arith]
  > Exp "||" Exp                        [strict(1), left, arith]
```

The conditional construct has the expected evaluation strategy, stating that only the first argument is evaluate:

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp    [strict(1)]
```

FUN's builtin lists are formed by enclosing comma-separated sequences of expressions (i.e., terms of sort `Exps`) in square brackets. The list constructor `cons` adds a new element to the top of the list, `head` and `tail` get the first element and the tail sublist of a list if they exist, respectively, and get stuck otherwise, and `null??` tests whether a list is empty or not; syntactically, these are just expression constants. In function patterns, we are also going to allow patterns following the usual head/tail notation; for example, the pattern `[x_1,...,x_n|t]` binds `x_1`, ..., `x_n` to the first elements of the matched list, and `t` to the list formed with the remaining elements. We define list patterns as ordinary expression constructs, although we will make sure that we do not give them semantics if they appear in any other place then in a function case pattern.

```
syntax Exp ::= "[" Exps "]"                              [strict]
             | "cons" |  "head" | "tail" | "null?"
             | "[" Exps "|" Exp "]"
syntax Val ::= "[" Vals "]"
```

Data constructors start with capital letters and they may or may not have arguments. We need to use the attribute "prefer" to make sure that, e.g., `Cons(a)` parses as constructor `Cons` with argument `a`, and not as the expression `Cons` (because constructor names are also expressions) regarded as a function applied to the expression `a`. Also, note that the constructor is strict in its second argument, because we want to evaluate its arguments but not the constuctor name itsef.

```
syntax ConstructorName
syntax Exp ::= ConstructorName
             | ConstructorName "(" Exps ")"    [prefer, strict(2)]
syntax Val ::= ConstructorName "(" Vals ")"
```

A function is essentially a `|`-separated ordered sequence of cases, each case of the form `pattern -> expression`, preceded by the language construct `fun`. Patterns will be defined

shortly, both for the builtin lists and for user-defined constructors. Recall that the syntax we define in **K** is not meant to serve as a ultimate parser for the defined language, but rather as a convenient notation for **K** abstract syntax trees, which we prefer when we write the semantic rules. It is therefore often the case that we define a more "generous" syntax than we want to allow programs to use. We do it here, too. Specifically, the syntax of `Cases` below allows any expressions to appear as pattern. This syntactic relaxation permits many wrong programs to be parsed, but that is not a problem because we are not going to give semantics to wrong combinations, so those programs will get stuck; moreover, our type inferencer will reject those programs anyway. Function application is just concatenation of expressions, without worrying about type correctness. Again, the type system will reject type-incorrect programs.

```
  syntax Exp ::= "fun" Cases
               | Exp Exp                                [strict, left]
// NOTE: We would like eventually to also have Exp "(" Exps ")"
  syntax Case  ::= Exp "->" Exp
  syntax Cases ::= List{Case, "|"}
```

The `let` and `letrec` binders have the usual syntax and functional meaning. We allow multiple `and`-separated bindings. Like for the function cases above, we allow a more generous syntax for the left-hand sides of bindings, noting that the semantics will get stuck on incorrect bindings and that the type system will reject those programs.

```
  syntax Exp ::= "let" Bindings "in" Exp
               | "letrec" Bindings "in" Exp                [prefer]
// The "prefer" attribute for letrec currently needed due to tool bug,
// to make sure that "letrec" is not parsed as "let rec".
  syntax Binding  ::= Exp "=" Exp
  syntax Bindings ::= List{Binding,"and"}
```

References are first class values in FUN. The construct `ref` takes an expression, evaluates it, and then it stores the resulting value at a fresh location in the store and returns that reference. Syntactically, `ref` is just an expression constant. The construct `&` takes a name as argument and evaluates to a reference, namely the store reference where the variable passed as argument stores its value; this construct is a bit controversial and is further discussed in the environment-based semantics of the FUN language, where we desugar `ref` to it. The construct `@` takes a reference and evaluates to the value stored there. The construct `:=` takes two expressions, the first expected to evaluate to a reference; the value of its second argument will be stored at the location to which the first points (the old value is thus lost). Finally, since expression evaluation now has side effects, it makes sense to also add a sequential composition

construct, which is sequentially strict. This evaluates to the value of its second argument; the value of the first argument is lost (which has therefore been evaluated only for its side effects.

```
  syntax Exp ::= "ref"
               | "&" Name
               | "@" Exp                                        [strict]
               | Exp ":=" Exp                                   [strict]
               | Exp ";" Exp                          [strict(1), right]
```

Call-with-current-continuation, named `callcc` in FUN, is a powerful control operator that originated in the Scheme programming language, but it now exists in many other functional languages. It works by evaluating its argument, expected to evaluate to a function, and by passing the current continuation, or evaluation context (or computation, in **K** terminology), as a special value to it. When/If this special value is invoked, the current context is discarded and replaced with the one held by the special value and the computation continues from there. It is like taking a snapshot of the execution context at some moment in time and then, when desired, being able to get back in time to that point. If you like games, it is like saving the game now (so you can work on your homework!) and then continuing the game tomorrow or whenever you wish. To issustrate the strength of `callcc`, we also allow exceptions in FUN by means of a conventional `try-catch` construct, which will desugar to `callcc`. We also need to introduce the special expression contant `throw`, but we need to use it as a function argument name in the desugaring macro, so we define it as a name instead of as an expression constant:

```
  syntax Exp ::= "callcc"
               | "try" Exp "catch" "(" Name ")" Exp
  syntax Name ::= "throw" [token]
```

Finally, FUN also allows polymorphic datatype declarations. These will be useful when we define the type system later on.

```
  syntax Exp ::= "datatype" Type "=" TypeCases Exp
// NOTE: In a future version of K, we want the datatype declaration
// to be a construct by itself, but that is not possible currently
// because K's parser wronly identifies the __ operation allowing
// a declaration to appear in front of an expression with the function
// application construct, giving ambiguous parsing errors.
```

We next need to define the syntax of types and type cases that appear in datatype declarations.

Like in many functional languages, type parameters/variables in user-defined types are quoted identifiers.

```
  syntax TypeVar
  syntax TypeVars ::= List{TypeVar,","}
```

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore all the type declations, so here the syntax of types below is only useful for generating the desired parser. To avoid syntactic ambiguities with the arrow construct for function cases, we use the symbol `-->` as a constructor for function types:

```
  syntax TypeName
  syntax Type ::= "int" | "bool" | "string"
                | Type "-->" Type                              [right]
                | "(" Type ")"                                [bracket]
                | TypeVar
                | TypeName              [klabel(TypeName), avoid]
                | Type TypeName   [klabel(Type-TypeName)]
                | "(" Types ")" TypeName                       [prefer]
  syntax Types ::= List{Type,","}
  syntax Types ::= TypeVars

  syntax TypeCase ::= ConstructorName
                    | ConstructorName "(" Types ")"
  syntax TypeCases ::= List{TypeCase,"|"}     [klabel(_|TypeCase_)]
```

## Additional Priorities

```
  syntax priorities @__FUN-UNTYPED-COMMON
                  > ___FUN-UNTYPED-COMMON
                  > arith
                  > _:=__FUN-UNTYPED-COMMON
                  > let_in__FUN-UNTYPED-COMMON
                    letrec_in__FUN-UNTYPED-COMMON
                    if_then_else__FUN-UNTYPED-COMMON
                  > _;__FUN-UNTYPED-COMMON
                  > fun__FUN-UNTYPED-COMMON
                  > datatype_=___FUN-UNTYPED-COMMON
endmodule

module FUN-UNTYPED-MACROS
  imports FUN-UNTYPED-COMMON
```

## Desugaring macros

We desugar the list non-constructor operations to functions matching over list patterns. In order to do that we need some new variables; for those, we follow the same convention like in the **K** tutorial, where we

added them as new identifier constructs starting with the character `$`, so we can easily recognize them when we debug or trace the semantics.

```
  syntax Name ::= "$h" | "$t"
  rule head => fun [$h|$t] -> $h                                  [macro]
  rule tail => fun [$h|$t] -> $t                                  [macro]
  rule null? => fun [.Exps] -> true | [$h|$t] -> false            [macro]
```

Multiple-head list patterns desugar into successive one-head patterns:

```
  rule [E1,E2,Es:Exps|T] => [E1|[E2,Es|T]]                        [macro-rec]
```

Uncurrying of multiple arguments in functions and binders:

```
  rule P1 P2 -> E => P1 -> fun P2 -> E                            [macro-rec]
  rule F P = E => F = fun P -> E                                  [macro-rec]
```

We desugar the `try-catch` construct into callcc:

```
  syntax Name ::= "$k" | "$v"
  rule try E catch(X) E'
    => callcc (fun $k -> (fun throw -> E)(fun X -> $k E'))        [macro]
```

For uniformity, we reduce all types to their general form:

```
//  rule TypeName(Tn:TypeName) => (.TypeVars) Tn                  [macro]
  rule `Type-TypeName`(T:Type, Tn:TypeName) => (T) Tn            [macro]
```

The dynamic semantics ignores all the type declarations:

```
  rule datatype _T = _TCs E => E                                  [macro]

endmodule


module FUN-UNTYPED-SYNTAX
  imports FUN-UNTYPED-COMMON
  imports BUILTIN-ID-TOKENS

  syntax Name ::= r"[a-z][_a-zA-Z0-9]*"           [token, prec(2)]
                | #LowerId                        [token]
  syntax ConstructorName ::= #UpperId             [token]
  syntax TypeVar  ::= r"['][a-z][_a-zA-Z0-9]*"    [token]
  syntax TypeName ::= Name                        [token]
endmodule
```

## Semantics

The semantics below is environment-based. A substitution-based definition of FUN is also available, but that drops the `&` construct as explained above.

```
module FUN-UNTYPED
  imports FUN-UNTYPED-COMMON
  imports FUN-UNTYPED-MACROS
  imports DOMAINS
  //imports PATTERN-MATCHING
```

## Configuration

The `k`, `env`, and `store` cells are standard
(see, for example, the definition of LAMBDA++ or IMP++ in the first part of the **K** tutorial).

```
  configuration <T color="yellow">
                  <k color="green"> $PGM:Exp </k>
                  <env color="violet"> .Map </env>
                  <store color="white"> .Map </store>
                </T>
```

## Values and results

We only define integers, Booleans and strings as values here, but will add more values later.

```
  syntax Val ::= Int | Bool | String
  syntax Exp ::= Val
  syntax Exps ::= Vals
  syntax Vals ::= Bottoms
  syntax KResult ::= Val
```

## Lookup

```
  rule <k> X:Name => V ...</k>
       <env>... X |-> L ...</env>
       <store>... L |-> V ...</store>
```

## Arithmetic expressions

```
  rule I1 * I2 => I1 *Int I2
  rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
  rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
  rule I1 + I2 => I1 +Int I2
  rule S1 ^ S2 => S1 +String S2
  rule I1 - I2 => I1 -Int I2
  rule - I => 0 -Int I
  rule I1 < I2 => I1 <Int I2
  rule I1 <= I2 => I1 <=Int I2
  rule I1 > I2 => I1 >Int I2
```

```
rule I1 >= I2 => I1 >=Int I2
rule V1:Val == V2:Val => V1 ==K V2
rule V1:Val != V2:Val => V1 =/=K V2
rule ! T => notBool(T)
rule true  && E => E
rule false && _ => false
rule true  || _ => true
rule false || E => E
```

## Conditional

```
rule if  true then E else _ => E
rule if false then _ else E => E
```

## Lists

We have already declared the syntactic list of expressions strict, so we can assume that all the elements that appear in a FUN list are evaluated. The only thing left to do is to state that a list of values is a value itself, that is, that the list square-bracket construct is indeed a constructor, and to give the semantics of `cons`. Since `cons` is a builtin function and is expected to take two arguments, we have to also state that `cons` itself is a value (specifically, a function/closure value, but we do not need that level of detail here), and also that `cons` applied to a value is a value (specifically, it would be a function/closure value that expects the second, list argument):

```
rule isVal(cons) => true
rule isVal(cons _V:Val) => true
rule cons V:Val [Vs:Vals] => [V,Vs]
```

## Data Constructors

Constructors take values as arguments and produce other values:

```
syntax Val ::= ConstructorName
```

## Functions and Closures

Like in the environment-based semantics of LAMBDA++ in the first part of the **K** tutorial, functions evaluate to closures. A closure includes the current environment besides the function contents; the environment will be used at execution time to lookup all the variables that appear free in the function body (we want static scoping in FUN).

```
syntax Val ::= closure(Map,Cases)
rule <k> fun Cases => closure(Rho,Cases) ...</k>  <env> Rho </env>
```

**Note:** The reader may want to get familiar with
how the pre-defined pattern matching works before proceeding.
The best way to do that is to consult
`k/include/modules/pattern-matching.k`.

We distinguish two cases when the closure is applied.
If the first pattern matches, then we pick the first case: switch to
the closed environment, get the matching map and bind all its
variables, and finally evaluate the function body of the first case,
making sure that the environment is properly recovered afterwards.
If the first pattern does not match, then we drop it and thus move on
to the next one.

```
  rule (. => getMatching(P, V)) ~> closure(_, P->_ | _) V:Val
  rule <k> matchResult(M:Map) ~> closure(Rho, _->E | _) _
           => bindMap(M) ~> E ~> setEnv(Rho') ...</k>
       <env> Rho' => Rho </env>
  rule (matchFailure => .) ~> closure(_, (_->_ | Cs:Cases => Cs)) _
//  rule <k> closure(Rho, P->E | _) V:Val
//           => bindMap(getMatching(P,V)) ~> E ~> setEnv(Rho') ...</k>
//       <env> Rho' => Rho </env>  when isMatching(P,V)
//  rule closure(_, (P->_ | Cs:Cases => Cs)) V:Val  when notBool isMatching(P,V)
```

## Let and Letrec

To highlight the similarities and differences between `let` and
`letrec`, we prefer to give them direct semantics instead of
to desugar them like in LAMBDA. See the formal definitions of
`bindTo`, `bind`, and `assignTo` at the end of
this module. Informally, `bindTo(Xs, Es)` first
evaluates the expressions `Es` in `Exps` in the current
environment (i.e., it is strict in its second argument), then it binds
the variables in `Xs` in `Names` to new locations and adds
those bindings to the environment, and finally writes the values
previously obtained after evaluating the expressions `Es` to those
new locations; `bind(Xs)` does only the bindings of
`Xs` to new locations and adds those bindings to the environment;
and `assignTo(Xs,Es)` evaluates the expressions
`Es` in the current environment and then it writes the resulting
values to the locations to which the variables `Xs` are already
bound to in the environment.

Therefore, `let Xs = Es in E` first
evaluates `Es` in the current environment, then adds new
bindings for `Xs` to fresh locations in the environment, then
writes the values of `Es` to those locations, and finally
evaluates `E` in the new environment, making sure that the

environment is properly recovered after the evaluation of `E`.
On the other hand, `letrec` does the same things but in a
different order: it first adds new bindings for `Xs` to fresh
locations in the environment, then it evaluates `Es` in the new
environment, then it writes the resulting values to their
corresponding locations, and finally it evaluates `E` and
recovers the environment. The crucial difference is that the
expressions `Es` now see the locations of the variables `Xs`
in the environment, so if they are functions, which is typically the
case with `letrec`, their closures will encapsulate in their
environments the bindings of all the bound variables, including
themselves (thus, we may have a closure value stored at location
`L`, whose environment contains a binding of the form
`F  L`; this way, the closure can invoke
itself).

```
rule <k> let Bs in E
      => bindTo(names(Bs),exps(Bs)) ~> E ~> setEnv(Rho) ...</k>
     <env> Rho </env>

rule <k> letrec Bs in E
      => bind(names(Bs))~>assignTo(names(Bs),exps(Bs))~>E~>setEnv(Rho)...</k>
     <env> Rho </env>
```

Recall that our syntax allows `let` and `letrec` to
take any expression in place of its binding. This allows us to use
the already existing function application construct to bind names to
functions, such as, e.g., `let x y = y in ...`.
The desugaring macro in the syntax module uncurries such declarations,
and then the semantic rules above only work when the remaining
bindings are identifiers, so the semantics will get stuck on programs
that misuse the `let` and `letrec` binders.

## References

The semantics of references is self-explanatory, except maybe for the
desugaring rule of `ref`, which is further discussed. Note
that `&X` grabs the location of `X` from the environment.
Sequential composition, which is needed only to accumulate the
side effects due to assignments, was strict in the first argument.
Once evaluated, its first argument is simply discarded:

```
syntax Name ::= "$x"
rule ref => fun $x -> & $x                                    [macro]
rule <k> & X => L ...</k>  <env>... X |-> L ...</env>
rule <k> @ L:Int => V:Val ...</k>  <store>... L |-> V ...</store>
rule <k> L:Int := V:Val => V ...</k>  <store>... L |-> (_=>V) ...</store>
```

```
rule _V:Val; E => E
```

The desugaring rule of `ref` (first rule above) works
because `&` takes a variable and returns its location (like in C).
Note that some "pure" functional programming researchers strongly dislike
the `&` construct, but favor `ref`. We refrain from having
a personal opinion on this issue here, but support `&` in the
environment-based definition of FUN because it is, technically speaking,
more powerful than `ref`. From a language design perspective, it
would be equally easy to drop `&` and instead give a direct
semantics to `ref`. In fact, this is precisely what we do in the
substitution-based definition of FUN, because there appears to be no way
to give a substitution-based definition to the `&` construct.

## Callcc

As we know it from the LAMBDA++ tutorial, call-with-current-continuation
is quite easy to define in **K**. We first need to define a special
value wrapping an execution context, that is, an environment saying
where the variables should be looked up, and a computation structure
saying what is left to execute (in a substitution-based definition,
this special value would be even simpler, as it would only need to
wrap the computation structure---see, for example, the
substitution-based semantics of LAMBDA++ in the the first part of the
**K** tutorial, or the substitution-based definition of FUN). Then
`callcc` creates such a value containing the current
environment and the current remaining computation, and passes it to
its argument function. When/If invoked, the special value replaces
the current execution context with its own and continues the execution
normally.

```
syntax Val ::= cc(Map,K)
rule isVal(callcc) => true
rule <k> (callcc V:Val => V cc(Rho,K)) ~> K </k>  <env> Rho </env>
rule <k> cc(Rho,K) V:Val ~> _ => V ~> K </k>  <env> _ => Rho </env>
```

## Auxiliary operations

## Environment recovery

The environment recovery operation is the same as for the LAMBDA++
language in the **K** tutorial and many other languages provided with the
**K** distribution. The first "anywhere" rule below shows an elegant
way to achieve the benefits of tail recursion in **K**.

```
syntax KItem ::= setEnv(Map)  // TODO: get rid of env
//rule (setEnv(_) => .) ~> setEnv(_)  [anywhere]
```

```
rule <k> _:Val ~> (setEnv(Rho) => .) ...</k> <env> _ => Rho </env>
  [structural]
```

### bindTo, bind and assignTo

The meaning of these operations has already been explained when we discussed the `let` and `letrec` language constructs above.

```
syntax KItem ::= bindTo(Names,Exps)         [strict(2)]
               | bindMap(Map)
               | bind(Names)

rule (. => getMatchingAux(Xs,Vs)) ~> bindTo(Xs:Names,Vs:Vals)
rule matchResult(M:Map) ~> bindTo(_:Names, _:Vals) => bindMap(M)

rule bindMap(.Map) => .  [structural]
rule <k> bindMap((X:Name |-> V:Val => .Map) _:Map) ...</k>
     <env> Rho => Rho[X <- !L:Int] </env>
     <store>... .Map => !L |-> V ...</store>
  [structural]

rule bind(.Names) => .                    [structural]
rule <k> bind(X:Name,Xs => Xs) ...</k>
     <env> Rho => Rho[X <- !_L:Int] </env>
  [structural]

syntax KItem ::= assignTo(Names,Exps)  [strict(2)]

rule <k> assignTo(.Names,.Vals) => . ...</k>            [structural]
rule <k> assignTo((X:Name,Xs => Xs),(V:Val,Vs:Vals => Vs)) ...</k>
     <env>... X |-> L ...</env>
     <store>... .Map => L |-> V ...</store>             [structural]
```

## Getters

The following auxiliary operations extract the list of identifiers and of expressions in a binding, respectively.

```
syntax Names ::= names(Bindings)  [function]
rule names(.Bindings) => .Names
rule names(X:Name=_ and Bs) => (X,names(Bs))::Names

syntax Exps ::= exps(Bindings)  [function]
rule exps(.Bindings) => .Exps
rule exps(_:Name=E and Bs) => E,exps(Bs)
```

```
  /* Extra kore stuff */
  syntax KResult ::= Vals
  syntax Exps ::= Names
  syntax Names ::= Bottoms

  /* Matching */
  syntax MatchResult ::= getMatching(Exp, Val)                      [function]
                       | getMatchingAux(Exps, Vals)                 [function]
                       | mergeMatching(MatchResult, MatchResult)    [function]
                       | matchResult(Map)
                       | "matchFailure"

  rule getMatching(C:ConstructorName(Es:Exps), C(Vs:Vals)) => getMatchingAux(Es, Vs)
  rule getMatching([Es:Exps], [Vs:Vals])                    => getMatchingAux(Es, Vs)
  rule getMatching(C:ConstructorName, C) => matchResult(.Map)
  rule getMatching(B:Bool, B)            => matchResult(.Map)
  rule getMatching(I:Int, I)             => matchResult(.Map)
  rule getMatching(S:String, S)          => matchResult(.Map)
  rule getMatching(N:Name, V:Val) => matchResult(N |-> V)
  rule getMatching(_, _) => matchFailure        [owise]

  rule getMatchingAux((E:Exp, Es:Exps), (V:Val, Vs:Vals)) => mergeMatching(getMatching(E, V)
  rule getMatchingAux(.Exps, .Vals)                        => matchResult(.Map)
  rule getMatchingAux(_, _) => matchFailure     [owise]

  rule mergeMatching(matchResult(M1:Map), matchResult(M2:Map)) => matchResult(M1 M2)
    requires intersectSet(keys(M1), keys(M2)) ==K .Set
  //rule mergeMatching(_, _) => matchFailure      [owsie]
  rule mergeMatching(matchResult(_:Map), matchFailure) => matchFailure
  rule mergeMatching(matchFailure, matchResult(_:Map)) => matchFailure
  rule mergeMatching(matchFailure, matchFailure)       => matchFailure
```

Besides the generic decomposition rules for patterns and values, we also want to allow `[head|tail]` matching for lists, so we add the following custom pattern decomposition rule:

```
  rule getMatching([H:Exp | T:Exp], [V:Val, Vs:Vals])
    => getMatchingAux((H, T), (V, [Vs]))
endmodule
```

Go to Lesson 2, FUN untyped, Substitution-Based.

# FUN — Untyped — Substitution

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the substitution-based definition of FUN. For additional explanations regarding the semantics of the various FUN constructs, the reader should consult the emvironment-based definition of FUN.

## Syntax

```
require "substitution.md"
//require "modules/pattern-matching.k"

module FUN-UNTYPED-COMMON
  imports DOMAINS-SYNTAX
```

## The Syntactic Constructs

```
  syntax Name
  syntax Names ::= List{Name,","}

  syntax Exp ::= Int | Bool | String | Name
               | "(" Exp ")"                        [bracket]
  syntax Exps  ::= List{Exp,","}                    [strict]
  syntax Val
  syntax Vals ::= List{Val,","}

  syntax Exp ::= left:
                   Exp "*" Exp                      [strict, arith]
               | Exp "/" Exp                        [strict, arith]
               | Exp "%" Exp                        [strict, arith]
               > left:
                   Exp "+" Exp                      [strict, left, arith]
               | Exp "^" Exp                        [strict, left, arith]
               | Exp "-" Exp                        [strict, prefer, arith]
               | "-" Exp                            [strict, arith]
               > non-assoc:
                   Exp "<" Exp                      [strict, arith]
               | Exp "<=" Exp                       [strict, arith]
               | Exp ">" Exp                        [strict, arith]
               | Exp ">=" Exp                       [strict, arith]
               | Exp "==" Exp                       [strict, arith]
               | Exp "!=" Exp                       [strict, arith]
               > "!" Exp                            [strict, arith]
               > Exp "&&" Exp                       [strict(1), left, arith]
```

```
              > Exp "||" Exp                      [strict(1), left, arith]

  syntax Exp ::= "if" Exp "then" Exp "else" Exp    [strict(1)]

  syntax Exp ::= "[" Exps "]"                      [strict]
              | "cons" |  "head" | "tail" | "null?"
              | "[" Exps "|" Exp "]"
  syntax Val ::= "[" Vals "]"

  syntax ConstructorName
  syntax Exp ::= ConstructorName
              | ConstructorName "(" Exps ")"       [prefer, strict(2)]
  syntax Val ::= ConstructorName "(" Vals ")"

  syntax Exp ::= "fun" Cases
              | Exp Exp                            [strict, left]
  syntax Case  ::= Exp "->" Exp                    [binder]
// NOTE: The binder attribute above is the only difference between this
// module and the syntax module of environment-based FUN.  We need
// to fix a bug in order to import modules and override the attributes
// of operations.
  syntax Cases ::= List{Case, "|"}

  syntax Exp ::= "let" Bindings "in" Exp
              | "letrec" Bindings "in" Exp                 [prefer]
  syntax Binding  ::= Exp "=" Exp
  syntax Bindings ::= List{Binding,"and"}

  syntax Exp ::= "ref"
              | "&" Name
              | "@" Exp                            [strict]
              | Exp ":=" Exp                       [strict]
              | Exp ";" Exp                        [strict(1), right]

  syntax Exp ::= "callcc"
              | "try" Exp "catch" "(" Name ")" Exp
  syntax Name ::= "throw" [token]

  syntax Exp ::= "datatype" Type "=" TypeCases Exp

  syntax TypeVar
  syntax TypeVars ::= List{TypeVar,","}

  syntax TypeName
  syntax Type ::= "int" | "bool" | "string"
               | Type "-->" Type                           [right]
```

```
               | "(" Type ")"                                [bracket]
               | TypeVar
               | TypeName              [klabel(TypeName), avoid]
               | Type TypeName   [klabel(Type-TypeName), onlyLabel]
               | "(" Types ")" TypeName                     [prefer]
  syntax Types ::= List{Type,","}
  syntax Types ::= TypeVars

  syntax TypeCase ::= ConstructorName
                    | ConstructorName "(" Types ")"
  syntax TypeCases ::= List{TypeCase,"|"}     [klabel(_|TypeCase_)]
```

## Additional Priorities

```
  syntax priorities @__FUN-UNTYPED-COMMON
                  > ___FUN-UNTYPED-COMMON
                  > arith
                  > _:=__FUN-UNTYPED-COMMON
                  > let_in__FUN-UNTYPED-COMMON
                    letrec_in__FUN-UNTYPED-COMMON
                    if_then_else__FUN-UNTYPED-COMMON
                  > _;__FUN-UNTYPED-COMMON
                  > fun__FUN-UNTYPED-COMMON
                  > datatype_=___FUN-UNTYPED-COMMON
endmodule

module FUN-UNTYPED-MACROS
  imports FUN-UNTYPED-COMMON
```

## Desugaring macros

```
  rule P1 P2 -> E => P1 -> fun P2 -> E                        [macro-rec]
  rule F P = E => F = fun P -> E                              [macro-rec]

  rule [E1,E2,Es:Exps|T] => [E1|[E2,Es|T]]                   [macro-rec]

//  rule 'TypeName(Tn:TypeName) => (.TypeVars) Tn              [macro]
  rule `Type-TypeName`(T:Type, Tn:TypeName) => (T) Tn         [macro]

  syntax Name ::= "$h" | "$t"
  rule head => fun [$h|$t] -> $h                              [macro]
  rule tail => fun [$h|$t] -> $t                              [macro]
  rule null? => fun [.Exps] -> true | [$h|$t] -> false        [macro]

  syntax Name ::= "$k" | "$v"
  rule try E catch(X) E'
```

```
    => callcc (fun $k -> (fun throw -> E)(fun X -> $k E'))   [macro]

  rule datatype _T = _TCs E => E                                [macro]
```

mu needed for letrec, but we put it here so we can also write
programs with mu in them, which is particularly useful for testing.

```
  syntax Exp ::= "mu" Case

endmodule


module FUN-UNTYPED-SYNTAX
  imports FUN-UNTYPED-COMMON
  imports BUILTIN-ID-TOKENS

  syntax Name ::= r"[a-z][_a-zA-Z0-9]*"            [token, prec(2)]
                | #LowerId                         [token]
  syntax ConstructorName ::= #UpperId              [token]
  syntax TypeVar  ::= r"['][a-z][_a-zA-Z0-9]*"     [token]
  syntax TypeName ::= Name                         [token]
endmodule
```

## Semantics

```
module FUN-UNTYPED
  imports FUN-UNTYPED-COMMON
  imports FUN-UNTYPED-MACROS
  imports DOMAINS
  imports SUBSTITUTION
  //imports PATTERN-MATCHING

  configuration <T color="yellow">
                  <k color="green"> $PGM:Exp </k>
                  <store color="white"> .Map </store>
                </T>
```

Both Name and functions are values now:

```
  syntax Val ::= Int | Bool | String | Name
  syntax Exp ::= Val
  syntax Exps ::= Vals
  syntax KResult ::= Val
  syntax Exps ::= Names
  syntax Vals ::= Names

  rule I1 * I2 => I1 *Int I2
  rule I1 / I2 => I1 /Int I2 when I2 =/=K 0
```

```
  rule I1 % I2 => I1 %Int I2 when I2 =/=K 0
  rule I1 + I2 => I1 +Int I2
  rule S1 ^ S2 => S1 +String S2
  rule I1 - I2 => I1 -Int I2
  rule - I => 0 -Int I
  rule I1 < I2 => I1 <Int I2
  rule I1 <= I2 => I1 <=Int I2
  rule I1 > I2 => I1 >Int I2
  rule I1 >= I2 => I1 >=Int I2
  rule V1:Val == V2:Val => V1 ==K V2
  rule V1:Val != V2:Val => V1 =/=K V2
  rule ! T => notBool(T)
  rule true  && E => E
  rule false && _ => false
  rule true  || _ => true
  rule false || E => E

  rule if  true then E else _ => E
  rule if false then _ else E => E

  rule isVal(cons) => true
  rule isVal(cons _V:Val) => true
  rule cons V:Val [Vs:Vals] => [V,Vs]

  syntax Val ::= ConstructorName

  rule isVal(fun _) => true
  syntax KVar ::= Name
  syntax Name ::= freshName(Int)    [freshGenerator, function]
  rule freshName(I:Int) => {#parseToken("Name", "#" +String Int2String(I))}:>Name

  rule (. => getMatching(P, V)) ~> (fun P->_ | _) V:Val
  rule matchResult(M:Map) ~> (fun _->E | _) _ => E[M]
  rule (matchFailure => .) ~> (fun (_->_ | Cs:Cases => Cs)) _
//  rule (fun P->E | _) V:Val => E[getMatching(P,V)]  when isMatching(P,V)
//  rule (fun (P->_ | Cs:Cases => Cs)) V:Val  when notBool isMatching(P,V)
```

We can reduce multiple bindings to one list binding, and then
apply the usual desugaring of let into function application.
It is important that the rule below is a macro, so let is eliminated
immediately, otherwise it may interfere in ugly ways with substitution.

```
  rule let Bs in E => ((fun [names(Bs)] -> E) [exps(Bs)])    [macro]
```

We only give the semantics of one-binding letrec.
Multipe bindings are left as an exercise.

```
  // changed because of parsing error
```

```
  //rule mu X:Name -> E => E[(mu X -> E) / X]
  rule mu X:Name -> E => E[X |-> (mu X -> E)]
  rule letrec F:Name = E in E' => let F = (mu F -> E) in E'  [macro]
```

We cannot have `&` anymore, but we can give direct semantics to `ref`. We also have to declare `ref` to be a value, so that we will never heat on it.

```
//  rule <k> & X => L ...</k>  <env>... X |-> L </env>
  rule isVal(ref) => true
  rule <k> ref V:Val => !L:Int ...</k> <store>... .Map => !L |-> V ...</store>
  rule <k> @ L:Int => V:Val ...</k>  <store>... L |-> V ...</store>
  rule <k> L:Int := V:Val => V ...</k>  <store>... L |-> (_=>V) ...</store>
  rule _V:Val; E => E

  syntax Val ::= cc(K)
  rule isVal(callcc) => true
  rule <k> (callcc V:Val => V cc(K)) ~> K </k>
  rule <k> cc(K) V:Val ~> _ => V ~> K </k>
```

Auxiliary getters

```
  syntax Names ::= names(Bindings)  [function]
  rule names(.Bindings) => .Names
  rule names(X:Name=_ and Bs) => X,names(Bs)

  syntax Exps ::= exps(Bindings)  [function]
  rule exps(.Bindings) => .Exps
  rule exps(_:Name=E and Bs) => E,exps(Bs)

  /* Extra kore stuff */
  syntax KResult ::= Vals
  syntax Exps ::= Names

  /* Matching */
  syntax MatchResult ::= getMatching(Exp, Val)                      [function]
                       | getMatchingAux(Exps, Vals)                 [function]
                       | mergeMatching(MatchResult, MatchResult)    [function]
                       | matchResult(Map)
                       | "matchFailure"

  rule getMatching(C:ConstructorName(Es:Exps), C(Vs:Vals)) => getMatchingAux(Es, Vs)
  rule getMatching([Es:Exps], [Vs:Vals])                   => getMatchingAux(Es, Vs)
  rule getMatching(C:ConstructorName, C) => matchResult(.Map)
  rule getMatching(B:Bool, B)            => matchResult(.Map)
  rule getMatching(I:Int, I)             => matchResult(.Map)
  rule getMatching(S:String, S)          => matchResult(.Map)
  rule getMatching(N:Name, V:Val) => matchResult(N |-> V)
```

```
  rule getMatching(_, _) => matchFailure        [owise]

  rule getMatchingAux((E:Exp, Es:Exps), (V:Val, Vs:Vals)) => mergeMatching(getMatching(E, V)
  rule getMatchingAux(.Exps, .Vals)                        => matchResult(.Map)
  rule getMatchingAux(_, _) => matchFailure     [owise]

  rule mergeMatching(matchResult(M1:Map), matchResult(M2:Map)) => matchResult(M1 M2)
    requires intersectSet(keys(M1), keys(M2)) ==K .Set
  //rule mergeMatching(_, _) => matchFailure      [owsie]
  rule mergeMatching(matchResult(_:Map), matchFailure) => matchFailure
  rule mergeMatching(matchFailure, matchResult(_:Map)) => matchFailure
  rule mergeMatching(matchFailure, matchFailure)       => matchFailure
```

Besides the generic decomposition rules for patterns and values, we also want to allow `[head|tail]` matching for lists, so we add the following custom pattern decomposition rule:

```
  rule getMatching([H:Exp | T:Exp], [V:Val, Vs:Vals])
    => getMatchingAux((H, T), (V, [Vs]))
endmodule
```

# LOGIK

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of LOGIK, a trivial language capturing the essence of the logic programming paradigm. In this definition, we explicitly focus on simplicity and mathematical clarity, not on advanced logic programming features or performance. Those are covered in the LOGIK++ extension under `examples/logik++`.

Specifically, a LOGIK program consists of a sequence of **Horn clauses** of the form

```
P :- P1, P2, ..., Pn .
```

followed by a **query** of the form

```
?- Q1, Q2, ..., Qm .
```

where `P`, `P1`, `P2`, ..., `Pn`, `Q1`, `Q2`, ..., `Qm` are **literals**. The symbol `:-` is read "if". A literal has the form

`p(T1,T2,...,Tk)`, where `p` is a **predicate symbol** and where `T1,T2,...,Tk` are **terms**. Terms are built as usual, with **operation symbols** and **variables**. A common convention in logic programming languages, also adopted here, is that variables are capitalized and operation symbols are not. Operations with zero arguments are called **constants** and are written without parentheses, that is, `c` instead of `c()`. Horn clauses without conditions, called **facts**, are written without `:-`, that is, `P.` instead of `P :- .`.

For example, the LOGIK program below gives a few facts about a `parent` predicate, then several clauses defining some useful predicates including an `ancestor` predicate, and finally a query asking for those who both have ancestors and are ancestors themselves in the `parent` relation:

```
parent(david,john).
parent(jim,david).
parent(steve,jim).
parent(nathan,steve).

grandparent(A,B):-
  parent(A,X),
  parent(X,B).

ancestor(A,B):-
  parent(A,X),
  parents(X,B).

parents(X,X).
parents(A,B):-
  ancestor(A,B).

both(X) :- ancestor(A,X), ancestor(X,B).

?- both(X).
```

Above, we only have constant operation symbols, so these and variables are the only terms that can be used in predicates. As expected, the LOGIK program above will give us three solutions for `X`: `david`, `steve`, and `jim`. If we inline the `both(X)` predicate in the query, that is, if we replace the query with `?- ancestor(A,X), ancestor(X,B).` then we get 10 solutions, one for for each triple `A`, `X`, and `B` satisfying both predicates `ancestor(A,X)` and `ancestor(X,B)`.

As another example, the program below defines an `append`

predicate followed by a simple goal:

```
append(nil,L,L).
append(cons(H,T),L,cons(H,Z)) :- append(T,L,Z).

?- append(cons(a,nil), cons(b,nil), V).
```

Besides the predicate symbol `append`, the program above also includes a constant symbol `nil` and a binary operation symbol `cons`. Additionally, the query also includes two more constants, `a` and `b`. The capitalized identifiers are all variables. As expected, the LOGIK program above yields only one solution, namely `V = cons(a,cons(b,nil))`. On the other hand, if we change the query to:

```
?- append(L1, cons(a,L2), cons(a,cons(b,cons(a,nil)))).
```

then LOGIK yields two solutions: one where `L1` is `cons(a,cons(b,nil))` and `L2` is `nil`, and another where `L1` is `nil` and `L2` is `cons(a,cons(b,nil))`.

The programs above all generated **ground solutions**, that is, solutions where the query variables are mapped to ground terms (i.e., terms without variables). Let us now consider the following query:

```
?- append(cons(a,nil), Y, Z).
```

There are obviously infinitely many ground solutions for the query above, e.g.,
`Y = nil` and `Z = cons(a,nil)`,
`Y = cons(a,nil)` and `Z = cons(a,cons(a,nil))`,
`Y = cons(b,nil)` and `Z = cons(a,cons(b,nil))`,
`Y = cons(c,cons(b,nil))` and `Z = cons(a,cons(c,cons(b,nil)))`,
etc. However, all the ground solutions for the query above can be elegantly characterized by the property that `Z` is bound to a list starting with `a` and followed by the list that `Y` is bound to. This property can in fact be described as a **symbolic solution** to the query: `Z = cons(a,Y)` or, equivalently, `Y = Symb` and `Z = cons(a,Symb)`. It is possible to define a "more general than" relation on such symbolic solutions, in the sense that the more particular solution can be obtained as a specialization/substitution of the more general one, and then it can be shown that the above is the **most general solution** to the stated query. Logic programming languages, including our LOGIK, attempt to always compute such most general solutions.

Logic programming languages are highly non-deterministic, in that several Horn clauses may be used at the same time, each possibly resulting in a different solution. Implementations of logic

programming languages consist of complex, optimized search and indexing algorithms, which we are not concerned with here. Instead, we here take advantage of **K**'s builtin support for search. Specifically, to find all the solutions of a LOGIK program, we have to use `krun` with the option `--search`. However, note that some programs have infinitely many solutions which cannot relate to each other by the "more general" relation. For example, the query

```
?- append(L1, cons(a,L2), L3) .
```

To address such cases and terminate, logic programming languages allow the user to choose how many solutions to be computed and displayed. In LOGIK, we can use the `--bound` option of `krun` for this purpose.

Finally, note that some queries have no solution. In some cases that is easy to detect by exhaustive analysis, such as for the following query:

```
?- append(cons(a,L1), L2, cons(b,L3)).
```

Logic programming languages, including LOGIK, terminate in such cases and report a no solution answer. However, there are cases where exhaustive analysis is not sufficient, such as for the query:

```
?- append(cons(a,L), nil, L).
```

In such cases, logic programming languages do not terminate. While one may devise techniques to detect non-termination in some cases, one cannot do it in general (same like for all Turing-complete languages).

```
require "unification.k"

module LOGIK-COMMON
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of LOGIK is straightforward: a program is a sequence of Horn clauses followed by a query:

```
  syntax Literal
  syntax Term ::= Literal | Literal "(" Terms ")"
  syntax Terms ::= List{Term,","}
  syntax Clause ::= Term ":-" Terms "." | Term "."
  syntax Query ::= "?-" Terms "."
  syntax Pgm ::= Query | Clause Pgm
endmodule
```

```
module LOGIK-SYNTAX
  imports LOGIK-COMMON
  imports BUILTIN-ID-TOKENS
```

Variables and literals are defined as tokens following the conventions used in Prolog (variables start with _ or capital letter, while literals start with lower case letters):

```
  syntax #KVariable ::= r"[A-Z_][A-Za-z0-9_]*"   [token, prec(2)]
                      | #UpperId                 [token]
  syntax Term ::= #KVariable [klabel(#SemanticCastToTerm)]
  syntax Literal ::= r"[a-z][a-zA-Z0-9_]*"                 [token]
                   | #LowerId                              [token]
endmodule

module LOGIK
  imports LOGIK-COMMON
  imports DOMAINS
  imports UNIFICATION
```

Unification is at the core of logic programming. Here we are going to use the predefined unification procedure (the same one we used in the type inferencers in Tutorial 5).

## Configuration

The configuration stores each clause in its own cell for easy access, and the most general unifier in a cell named `mgu`, same like the type inferencers. The `k` cell holds the query and the `fresh` cell holds a fresh clause instance to be attempted on the next query item. To more easily read the solutions, we add a second top-level cell, `solution`. Both top cells are optional. Indeed, we start with the main top cell and, when a solution is found, we move it into the `solution` cell and discard the main cell.

```
  configuration <T color="yellow" multiplicity="?">
                  <k color="green"> $PGM:Pgm </k>
                  <fresh color="orange"> .K </fresh>
                  <clauses color="red">
                    <clause color="pink" multiplicity="*"> .K </clause>
                  </clauses>
                  <mgu> .K </mgu>
                </T>
                <solution multiplicity="?"> .K </solution>
```

## Pre- and post-processing

Before we launch the semantics, we first scan the given program and place each clause in its own cell, and then place the query in the `k` cell and initialize the mgu with the variables from the query.

Note that we put a fresh instance of the clause to avoid interference with the query variables. By a "fresh instance" of a clause we mean one whose variables are renamed with fresh names; we need that in order to avoid undesired unification conflicts due to particular names chosen for variables in the original program, as well as conflicts due to subsequent uses of the same clause. It is safe to rename the variables in a clause, because clauses are universally quantified in their variables. This process of creating a fresh instance of a clause is similar to how we created fresh instances of type schemas in the higher-order type inferencer discussed in Tutorial 5. Indeed, we can safely regard clauses as "clause schemas" comprising infinitely many instances, one for each context.

```
  rule <k> C:Clause Pgm => Pgm </k>
       (.Bag => <clause> #renameVariables(C) </clause>)

  rule <k> ?- Ls:Terms. => Ls ...</k>
       <mgu> _ => #variablesMap(#variables(Ls)) </mgu>
```

We also sequentialize the goals for easier processing:

```
  rule L:Term, Ls:Terms => L ~> Ls
  rule .Terms => .
```

When all the goals are solved, indicated by the empty `k` cell, the calculated most general unifier (mgu) is in the `mgu` cell. In that case, to ease reading of the final solution we move the mgu in the `solution` cell and delete the rest of the configuration:

```
rule <T>... <k> . </k> <mgu> Theta </mgu> ...</T>
  => <solution> Theta </solution>
```

Since we are not interested in seeing the failed attempts to solve the query, we collapse all the error configurations into an empty configuration (recall that both top-level cells in the configuration were declared optional). This way, if we see an empty configuration when we search for all solutions, we know that some attempts failed (but we do not know which ones).

```
// this would be nice, but we need feedback from the external unifier
// for this.
//  rule <T>... <mgu> _:MguError </mgu> ...</T> => .
```

## Semantics

Once all the infrastructure is in place, the actual semantics of LOGIK is quite simple. All we have to do is to pick some (fresh instance of a) clause, then unify its conclusion with the first query literal, and then replace that literal with condition of the clause. The intuition here is the following: to satisfy the first literal in the query, we need to find some instance of some clause that matches it, and then to similarly show that we can satisfy the conditions of that clause. Mathematically, this is an instance of the proof principle called **resolution**: if `p  q` and `¬ p  r` hold, then so does `q  r`. We let it as an exercise to the reader to see how the two relate (hint: assume the negation of the goal together with all the clauses, and then derive **false**).

The following two rules are tightly connected and they together perform the following core task: pick a fresh instance of a clause which unifies with the first goal item, then add its conditions as new goals.

Pick a clause and generate a fresh instance of it when the `fresh` cell is empty:

```
rule <fresh> . => #renameVariables(C) </fresh> <clause> C </clause>
     <k> T:Term ...</k>
requires #unifiable(T,head(C))
  [transition]

syntax Term ::= head(Clause) [function]
rule head(L.) => L
rule head(L:-_.) => L
```

If the goal is unifiable with the fresh clause's head, replace the goal with the clause body, and empty the `fresh` cell (so that another clause can be chosen using the rule above):

```
rule <k> L:Term => . ...</k>
     <fresh>  L:Term . => . </fresh>

rule <k> L:Term :KItem => Ls ...</k>
     <fresh>  L:Term :- Ls:Terms. => . </fresh>
```

Note that there is no problem if a clause is chosen whose conclusion literal does not unify with the first goal literal. The search
option of `krun` will systematically try all clauses, so no solution is missed. Of course, the above is not the most efficient way to implement a logic programming language, but recall that our objective here was to present a simple and mathematically clean

solution. We encourage the interested reader to consult the LOGIK++ language definition for a more efficient definition of a richer logic programming language.

`endmodule`

# LOGIK

Author: Grigore Roșu (grosu@illinois.edu)
Organization: University of Illinois at Urbana-Champaign

Author: Traian Florin Șerbănuță (traian.serbanuta@unibuc.ro)
Organization: University of Bucharest

## Abstract

This is the **K** semantic definition of LOGIK, a trivial language capturing the essence of the logic programming paradigm. In this definition, we explicitly focus on simplicity and mathematical clarity, not on advanced logic programming features or performance. Those are covered in the LOGIK++ extension under `examples/logik++`.

Specifically, a LOGIK program consists of a sequence of **Horn clauses** of the form

```
P :- P1, P2, ..., Pn .
```

followed by a **query** of the form

```
?- Q1, Q2, ..., Qm .
```

where `P`, `P1`, `P2`, ..., `Pn`, `Q1`, `Q2`, ..., `Qm` are **literals**. The symbol `:-` is read "if". A literal has the form `p(T1,T2,...,Tk)`, where `p` is a **predicate symbol** and where `T1,T2,...,Tk` are **terms**. Terms are built as usual, with **operation symbols** and **variables**. A common convention in logic programming languages, also adopted here, is that variables are capitalized and operation symbols are not. Operations with zero arguments are called **constants** and are written without parentheses, that is, `c` instead of `c()`. Horn clauses without conditions, called **facts**, are written without `:-`, that is, `P.` instead of `P :- .`.

For example, the LOGIK program below gives a few facts about a `parent` predicate, then several clauses defining some useful predicates including an `ancestor` predicate, and finally a query asking for those who both have ancestors and are ancestors themselves in the `parent` relation:

```
parent(david,john).
parent(jim,david).
parent(steve,jim).
parent(nathan,steve).

grandparent(A,B):-
  parent(A,X),
  parent(X,B).

ancestor(A,B):-
  parent(A,X),
  parents(X,B).

parents(X,X).
parents(A,B):-
  ancestor(A,B).

both(X) :- ancestor(A,X), ancestor(X,B).

?- both(X).
```

Above, we only have constant operation symbols, so these and variables are the only terms that can be used in predicates. As expected, the LOGIK program above will give us three solutions for `X`: `david`, `steve`, and `jim`. If we inline the `both(X)` predicate in the query, that is, if we replace the query with `?- ancestor(A,X), ancestor(X,B).` then we get 10 solutions, one for for each triple `A`, `X`, and `B` satisfying both predicates `ancestor(A,X)` and `ancestor(X,B)`.

As another example, the program below defines an `append` predicate followed by a simple goal:

```
append(nil,L,L).
append(cons(H,T),L,cons(H,Z)) :- append(T,L,Z).

?- append(cons(a,nil), cons(b,nil), V).
```

Besides the predicate symbol `append`, the program above also includes a constant symbol `nil` and a binary operation symbol `cons`. Additionally, the query also includes two more constants, `a` and `b`. The capitalized identifiers are all variables. As expected, the LOGIK program above yields only one solution, namely `V = cons(a,cons(b,nil))`. On the other hand, if we change the query to:

```
?- append(L1, cons(a,L2), cons(a,cons(b,cons(a,nil)))).
```

then LOGIK yields two solutions: one where `L1` is
`cons(a,cons(b,nil))` and `L2` is `nil`,
and another where `L1` is `nil` and `L2` is
`cons(a,cons(b,nil))`.

The programs above all generated **ground solutions**, that is, solutions where the query variables are mapped to ground terms (i.e., terms without variables). Let us now consider the following query:

```
?- append(cons(a,nil), Y, Z).
```

There are obviously infinitely many ground solutions for the query above, e.g.,
`Y = nil` and `Z = cons(a,nil)`,
`Y = cons(a,nil)` and `Z = cons(a,cons(a,nil))`,
`Y = cons(b,nil)` and `Z = cons(a,cons(b,nil))`,
`Y = cons(c,cons(b,nil))` and `Z = cons(a,cons(c,cons(b,nil)))`,
etc. However, all the ground solutions for the query above can be elegantly characterized by the property that `Z` is bound to a list starting with `a` and followed by the list that `Y` is bound to. This property can in fact be described as a **symbolic solution** to the query: `Z = cons(a,Y)` or, equivalently,
`Y = Symb` and `Z = cons(a,Symb)`. It is possible to define a "more general than" relation on such symbolic solutions, in the sense that the more particular solution can be obtained as a specialization/substitution of the more general one, and then it can be shown that the above is the **most general solution** to the stated query. Logic programming languages, including our LOGIK, attempt to always compute such most general solutions.

Logic programming languages are highly non-deterministic, in that several Horn clauses may be used at the same time, each possibly resulting in a different solution. Implementations of logic programming languages consist of complex, optimized search and indexing algorithms, which we are not concerned with here. Instead, we here take advantage of **K**'s builtin support for search. Specifically, to find all the solutions of a LOGIK program, we have to use `krun` with the option `--search`. However, note that some programs have infinitely many solutions which cannot relate to each other by the "more general" relation. For example, the query

```
?- append(L1, cons(a,L2), L3) .
```

To address such cases and terminate, logic programming languages allow the user to choose how many solutions to be computed and displayed. In LOGIK, we can use the `--bound` option of `krun` for this purpose.

Finally, note that some queries have no solution. In some cases that

is easy to detect by exhaustive analysis, such as for the following query:

```
?- append(cons(a,L1), L2, cons(b,L3)).
```

Logic programming languages, including LOGIK, terminate in such cases and report a no solution answer. However, there are cases where exhaustive analysis is not sufficient, such as for the query:

```
?- append(cons(a,L), nil, L).
```

In such cases, logic programming languages do not terminate. While one may devise techniques to detect non-termination in some cases, one cannot do it in general (same like for all Turing-complete languages).

```
require "unification.k"

module LOGIK-COMMON
  imports DOMAINS-SYNTAX
```

## Syntax

The syntax of LOGIK is straightforward: a program is a sequence of Horn clauses followed by a query:

```
  syntax Literal
  syntax Term ::= Literal | Literal "(" Terms ")"
  syntax Terms ::= List{Term,","}
  syntax Clause ::= Term ":-" Terms "." | Term "."
  syntax Query ::= "?-" Terms "."
  syntax Pgm ::= Query | Clause Pgm
endmodule

module LOGIK-SYNTAX
  imports LOGIK-COMMON
  imports BUILTIN-ID-TOKENS
```

Variables and literals are defined as tokens following the conventions used in Prolog (variables start with _ or capital letter, while literals start with lower case letters):

```
  syntax #KVariable ::= r"[A-Z_][A-Za-z0-9_]*"   [token, prec(2)]
                      | #UpperId                 [token]
  syntax Term ::= #KVariable [klabel(#SemanticCastToTerm)]
  syntax Literal ::= r"[a-z][a-zA-Z0-9_]*"                [token]
                   | #LowerId                             [token]
endmodule

module LOGIK
```

```
imports LOGIK-COMMON
imports DOMAINS
imports UNIFICATION
```

Unification is at the core of logic programming. Here we are going to use the predefined unification procedure (the same one we used in the type inferencers in Tutorial 5).

## Configuration

The configuration stores each clause in its own cell for easy access, and the most general unifier in a cell named `mgu`, same like the type inferencers. The `k` cell holds the query and the `fresh` cell holds a fresh clause instance to be attempted on the next query item. To more easily read the solutions, we add a second top-level cell, `solution`. Both top cells are optional. Indeed, we start with the main top cell and, when a solution is found, we move it into the `solution` cell and discard the main cell.

```
configuration <T color="yellow" multiplicity="?">
                <k color="green"> $PGM:Pgm </k>
                <fresh color="orange"> .K </fresh>
                <clauses color="red">
                  <clause color="pink" multiplicity="*"> .K </clause>
                </clauses>
                <mgu> .K </mgu>
              </T>
              <solution multiplicity="?"> .K </solution>
```

## Pre- and post-processing

Before we launch the semantics, we first scan the given program and place each clause in its own cell, and then place the query in the `k` cell and initialize the mgu with the variables from the query.

Note that we put a fresh instance of the clause to avoid interference with the query variables. By a "fresh instance" of a clause we mean one whose variables are renamed with fresh names; we need that in order to avoid undesired unification conflicts due to particular names chosen for variables in the original program, as well as conflicts due to subsequent uses of the same clause. It is safe to rename the variables in a clause, because clauses are universally quantified in their variables. This process of creating a fresh instance of a clause is similar to how we created fresh instances of type schemas in the higher-order type inferencer discussed in Tutorial 5. Indeed, we can safely regard clauses as "clause schemas" comprising infinitely

many instances, one for each context.

```
rule <k> C:Clause Pgm => Pgm </k>
     (.Bag => <clause> #renameVariables(C) </clause>)

rule <k> ?- Ls:Terms. => Ls ...</k>
     <mgu> _ => #variablesMap(#variables(Ls)) </mgu>
```

We also sequentialize the goals for easier processing:

```
rule L:Term, Ls:Terms => L ~> Ls
rule .Terms => .
```

When all the goals are solved, indicated by the empty `k` cell, the calculated most general unifier (mgu) is in the `mgu` cell. In that case, to ease reading of the final solution we move the mgu in the `solution` cell and delete the rest of the configuration:

```
rule <T>... <k> . </k> <mgu> Theta </mgu> ...</T>
  => <solution> Theta </solution>
```

Since we are not interested in seeing the failed attempts to solve the query, we collapse all the error configurations into an empty configuration (recall that both top-level cells in the configuration were declared optional). This way, if we see an empty configuration when we search for all solutions, we know that some attempts failed (but we do not know which ones).

```
// this would be nice, but we need feedback from the external unifier
// for this.
//  rule <T>... <mgu> _:MguError </mgu> ...</T> => .
```

## Semantics

Once all the infrastructure is in place, the actual semantics of LOGIK is quite simple. All we have to do is to pick some (fresh instance of a) clause, then unify its conclusion with the first query literal, and then replace that literal with condition of the clause. The intuition here is the following: to satisfy the first literal in the query, we need to find some instance of some clause that matches it, and then to similarly show that we can satisfy the conditions of that clause. Mathematically, this is an instance of the proof principle called **resolution**: if `p  q` and `¬ p  r` hold, then so does `q  r`. We let it as an exercise to the reader to see how the two relate (hint: assume the negation of the goal together with all the clauses, and then derive **false**).

The following two rules are tightly connected and they together perform the following core task: pick a fresh instance of a clause

which unifies with the first goal item, then add its conditions as new goals.

Pick a clause and generate a fresh instance of it when the `fresh` cell is empty:

```
rule <fresh> . => #renameVariables(C) </fresh> <clause> C </clause>
     <k> T:Term ...</k>
requires #unifiable(T,head(C))
  [transition]

syntax Term ::= head(Clause) [function]
rule head(L.) => L
rule head(L:-_.) => L
```

If the goal is unifiable with the fresh clause's head, replace the goal with the clause body, and empty the `fresh` cell (so that another clause can be chosen using the rule above):

```
rule <k> L:Term => . ...</k>
     <fresh>  L:Term . => . </fresh>

rule <k> L:Term :KItem => Ls ...</k>
     <fresh>  L:Term :- Ls:Terms. => . </fresh>
```

Note that there is no problem if a clause is chosen whose conclusion literal does not unify with the first goal literal. The search option of `krun` will systematically try all clauses, so no solution is missed. Of course, the above is not the most efficient way to implement a logic programming language, but recall that our objective here was to present a simple and mathematically clean solution. We encourage the interested reader to consult the LOGIK++ language definition for a more efficient definition of a richer logic programming language.

```
endmodule
```

# K User Manual

**NOTE:** The K User Manual is still **under construction**; some features of K may have partial or missing documentation.

## Introduction

### Why K?

The K Framework is a programming language and system design toolkit made for

practioners and researchers alike.

**K For Practioners:**
*K is a framework for deriving programming languages tools from their semantic specifications.*

Typically, programming language tool development follows a similar pattern. After a new programming language is designed, separate teams will develop separate language tools (e.g. a compiler, interpreter, parser, symbolic execution engine, etc). Code reuse is uncommon. The end result is that for each
new language, the same basic tools and patterns are re-implemented again and again.

K approaches the problem differently -- it generates each of these tools from a single language specification.
The work of programming language design and tool implementation are made separate concerns.
The end result is that the exercise of
designing new languages and their associated tooling is now reduced to developing a single language specification from which we derive our tooling *for free*.

**K For Researchers:**
*K is a configuration- and rewrite-based executable semantic framework.*

In more detail, K specifications are:

1. **Executable:** compile into runnable and testable programs;
2. **Semantic:** correspond to a logical theory with a sound and relatively complete proof system;
3. **Configuration-based:** organize system states into compositional, hierarchical, labelled units called cells;
4. **Rewrite-based:** define system transitions using rewrite rules.

K specifications are compiled into particular *matching logic* theories, giving them a simple and expressive semantics. K semantic rules are implicitly defined over the entire configuration structure, but omit unused cells, enabling a highly modular definitional style. Furthermore, K has been used to develop programming languages, type systems, and formal analysis tools.

### Manual Objectives

As mentioned in the *Why K?* section above, the K Framework is designed as a collection of language-generic command-line interface (CLI) tools which revolve around K specifications. These tools cover a broad range of uses, but they typically fall into one of the following categories:

1. Transforming K Specs (e.g. compilation)
2. Running K Specs (e.g. concrete and symbolic execution)

3. Analyzing K Specs (e.g. theorem proving)

The main *user-facing* K tools include:

- `kompile` - the K compiler driver
- `kparse` - the stanadlone K parser and abstract syntax tree (AST) transformation tool
- `krun` - the K interpreter and symbolic execution engine driver
- `kprove` - the K theorem prover

This user manual is designed to be a tool reference.
In particular, it is not desgined to be a tutorial on how to write K specifications or to teach the logical foundations of K. New K users should consult our dedicated
K tutorial,
or the more language-design oriented
PL tutorial.
Researchers seeking to learn more about the logic underlying K are encouraged to peruse the
growing literature on K and matching logic.
We will consider the manual complete when it provides a complete description of
all user-facing K tools and features.

## Introduction to K

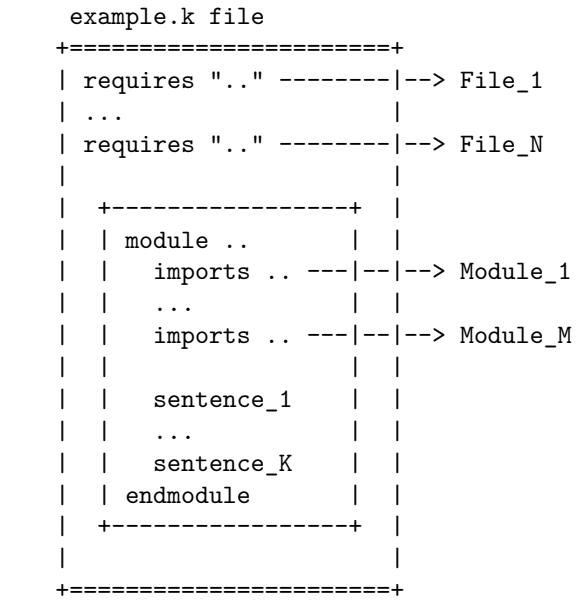
Since K specifications are the primary input into the entire system, let us take a moment to describe them. At the highest level, K specifications describe a programming language or system using three different pieces:

1. the *system primitives*, the base datatypes used during system operation, e.g., numbers, lists, maps, etc;
2. the *system state*, a tuple or record over system primitives which gives a complete snapshot of the system at any given moment;
3. the *system behavior*, a set of rules which defines possible system evolutions.

K specifications are then defined by a collection of *sentences* which correspond to the three concepts above:

1. `syntax` declarations encode the *system primitives*;
2. `configuration` declarations encode the *system state*;
3. `context` and `rule` declarations encode the *system behavior*.

K sentences are then organized into one or *modules* which are stored in one or more *files*. In this scheme, files may *require* other files and modules may *import* other modules, giving rise to a hierarchy of files and modules. We give an intuitive sketch of the two levels of grouping in the diagram below:




where:

- files and modules are denoted by double-bordered and single-borded boxes respectively;
- file or module identifiers are denoted by double dots (`..`);
- potential repititions are denoted by triple dots (`...`).

In the end, we require that the file and module hierarchies both form a directed acyclic graph (DAG). This is, no file may recursively require itself, and likewise, no module may recursively import itself.

We now zoom in further to discuss the various kinds of sentences contained in K
specifications:

1. sentences that define our *system's primitives*, including:
    - **sort declarations:** define new categories of primitive datatypes
    - **Backus-Naur Form (BNF) grammar declarations:** define the operators that inhabit our primitive datatypes
    - **lexical syntax declarations:** define lexemes/tokens for the lexer/tokenizer
    - **syntax associativity declarations:** specify the associativity/grouping of our declared operators
    - **syntax priority declarations:** specify the priority of potential ambiguous operators
2. sentences that define our *system's state*, including:

- **configuration declarations:** define labelled, hierarchical records using an nested XML-like syntax

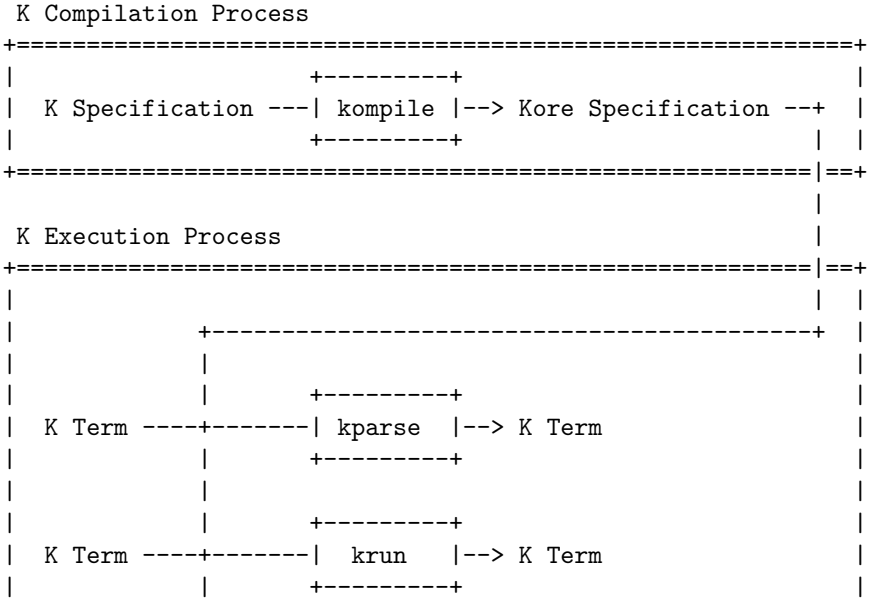
3. sentences that define our *system's behavior*, including:
   - **context declarations:** describe how primitives and configurations can simplify
   - **context alias declarations:** define templates that can generate new contexts
   - **rule declarations:** define how the system transitions from one state to the next

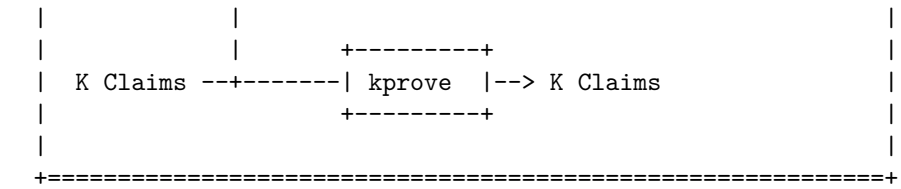
### K Process Overview

We now examine how the K tools are generally used. The main input to all of the K tools is a K specification. For effieciency reasons, this specification is first compiled into an intermediate representation called Kore. Once we have obtained this intermediate representation, we can use it to do:

1. parsing/pretty-printing, i.e., converting a K term, whose syntax is defined by a K specification, into a alternate representation
2. concrete and abstract execution of a K specification
3. theorem proving, i.e., verifying whether a set of claims about a K specification hold

We represent the overall process using the graphic below:






where:

- process outlines are denoted by boxes with double-lined borders
- executables are denoted by boxes with single-lined borders
- inputs and outputs are denoted by words attached to lines
- K terms typically correspond to *programs* defined in a particular language's syntax (which are either parsed using `kparse` or executed using `krun`)
- K claims are a notation for describing *how* certain K programs *should* execute (which are checked by our theorem prover `kprove`)

**K Compilation Process:**
Let us start with a description of the compilation process. According to the above diagram, the compiler driver is called `kompile`. For our purposes, it is enough to view the K compilation process as a black box that transforms a K specification into a lower-level Kore specification that encodes the same information, but that is easier to work with programmatically.

**K Execution Process:**
We now turn our attention to the K execution process. Abstractly, we can divide the K execution process into the following stages:

1. the kore specification is loaded (which defines a lexer, parser, and unparser among other things)
2. the input string is lexed into a token stream
3. the token stream is parsed into K terms/claims
4. the K term/claims are transformed according the K tool being used (e.g. `kparse`, `krun`, or `kprove`)
5. the K term/claims are unparsed into a string form and printed

Note that all of the above steps performed in K execution process are fully prescribed by the input K specification. Of course, there are entire languages devoted to encoding these various stages proces individually, e.g., `flex` for lexers, `bison` for parsers, etc. What K offers is a *consistent* language to package the above concepts in a way that we believe is convenient and practical for a wide range of uses.

## Module Declaration

K modules are declared at the top level of a K file. They begin with the `module` keyword and are followed by a **module ID** and an optional set of

attributes. They continue with zero or more imports and zero or more sentences until the `endmodule` keyword is reached.

A module ID consists of an optional `#` at the beginning, followed by one or more components separated by hyphens. Each component can contain letters, numbers, or underscores.

After the module ID, attributes can be specified in square brackets. See below for an (incomplete) list of allowed module attributes.

Following the attributes, a module can contain zero or more **imports**. An import consists of the `import` or `imports` keywords followed by a module ID. An import tells the compiler that this module should contain all the sentences (recursively) contained by the module being imported.

Imports can be **public** or **private**. By default, they are public, which means that all the imported syntax can be used by any module that imports the
module doing the import. However, you can explicitly override the visibility of the import with the `public` or `private` keyword immediately prior to the module name. A module imported privately does not export its syntax to modules
that import the module doing the import.

Following imports, a module can contain zero or more sentences. A sentence can
be a syntax declaration, a rule, a configuration declaration, a context, a claim, or a context alias. Details on each of these can be found in subsequent sections.

#### `private` attribute

If the module is given the `private` attribute, all of its imports and syntax are private by default. Individual pieces of syntax can be made public with the `public` attribute, and individual imports can be made public with the `public` keyword. See relevant sections on syntax and modules for more details on what it means for syntax and imports to be public or private.

#### `symbolic` and `concrete` attribute

These attributes may be placed on modules to indicate that they should only be used by the Haskell and LLVM backends respectively. If the definition is compiled on the opposite backend, they are implicitly removed from the definition prior to parsing anywhere they are imported. This can be useful when used in limited capacity in order to provide alternate semantics for certain features on different backends. It should be used sparingly as it makes it more difficult to trust the correctness of your semantics, even in the presence of testing.

## Syntax Declaration

### Named Non-Terminals

We have added a syntax to Productions which allows non-terminals to be given a
name in productions. This significantly improves the ability to document K, by providing a way to explicitly explain what a field in a production corresponds to instead of having to infer it from a comment or from the rule body.

The syntax is:

`name: Sort`

This syntax can be used anywhere in a K definition that expects a non-terminal.

### `klabel(_)` and `symbol` attributes

By default K generates for each syntax definition a long and obfuscated klabel string, which serves as a unique internal identifier and also is used in kast format of that syntax. If we need to reference a certain syntax production externally, we have to manually define the klabels using the `klabel` attribute. One example of where you would want to do this is to be able to refer to a given symbol via the `syntax priorities` attribute, or to enable overloading of a given symbol.

If you only provide the `klabel` attribute, you can use the provided `klabel` to refer to that symbol anywhere in the frontend K code. However, the internal identifier seen by the backend for that symbol will still be the long obfuscated generated string. Sometimes you want control over the internal identfier used as
well, in which case you use the `symbol` attribute. This tells the frontend to use whatever the declared `klabel` is directly as the internal identfier.

For example:

```
module MYMODULE
    syntax FooBarBaz ::= #Foo( Int, Int ) [klabel(#Foo), symbol] // symbol1
                       | #Bar( Int, Int ) [klabel(#Bar)]         // symbol2
                       | #Baz( Int, Int )                        // symbol3
endmodule
```

Here, we have that:

- In frontend K, you can refer to "symbol1" as `#Foo` (from `klabel(#Foo)`), and the backend will see `'Hash'Foo` as the symbol name.
- In frontend K, you can refer to "symbol2" as `Bar` (from `klabel(#Bar)`), and the backend will see `'Hash'Bar'LParUndsCommUndsRParUnds'MYMODULE'Unds'FooBarBaz'Unds'Int'Unds'Int` as the symbol name.

- In frontend K, you can refer to "symbol3" as
  `#Baz(_,_)_MYMODULE_FooBarBaz_Int_Int` (from auto-generated klabel),
  and
  the backend will see
  `'Hash'Baz'LParUndsCommUndsRParUnds'MYMODULE'Unds'FooBarBaz'Unds'Int'Unds'Int`
  as the symbol name.

The `symbol` provided *must* be unique to this definition. This is enforced by K. In general, it's recommended to use `symbol` attribute whenever you use `klabel` unless you explicitely have a reason not to (eg. you want to *overload* symbols, or you're using a deprecated backend). It can be very helpful use the `symbol` attribute for debugging, as many debugging messages are printed in Kast format which will be more readable with the `symbol` names you explicitely declare. In addition, if you are programatically manipulating definitions via the JSON Kast format, building terms using the user-provided pretty `symbol, klabel(...)` is easier and less error-prone when the auto-generation process for klabels changes.

#### Parametric productions and `bracket` attributes

Some syntax productions, like the rewrite operator, the bracket operator, and the #if #then #else #fi operator, cannot have their precise type system expressed using only concrete sorts.

Prior versions of K solved this issue by using the K sort in this case, but this introduces inexactness in which poorly typed terms can be created even without having a cast operator present in the syntax, which is a design consideration we would prefer to avoid.

It also introduces cases where terms cannot be placed in positions where they ought to be well sorted unless their return sort is made to be KBott, which in turn vastly complicates the grammar and makes parsing much slower.

In order to introduce this, we provide a new syntax for parametric productions in K. This allows you to express syntax that has a sort signature based on parametric polymorphism. We do this by means of an optional curly-brace-enclosed list of parameters prior to the return sort of a production.

Some examples:

```
syntax {Sort} Sort ::= "(" Sort ")" [bracket]
syntax {Sort} KItem ::= Sort
syntax {Sort} Sort ::= KBott
syntax {Sort} Sort ::= Sort "=>" Sort
syntax {Sort} Sort ::= "#if" Bool "#then" Sort "#else" Sort "#fi"
syntax {Sort1, Sort2} Sort1 ::= "#fun" "(" Sort2 "=>" Sort1 ")" "(" Sort2 ")"
```

Here we have:

1. Brackets, which can enclose any sort but should be of the same sort that was
   enclosed
2. Every sort is a KItem.
3. A KBott term can appear inside any sort
4. Rewrites, which can rewrite a value of any sort to a value of the same sort, or to a different sort which is allowed in that context
5. If then else, which can return any sort but which must contain that sort on
   both the true and false branches.
6. lambda applications, in which the argument and parameter must be the same
   sort, and the return value of the application must be the same sort as the return value of the function.

Note the last case, in which two different parameters are specified separated by a comma. This indicates that we have multiple independent parameters which
must be the same each place they occur, but not the same as the other parameters.

In practice, because every sort is a subsort of K, the `Sort2`
parameter in #6 above does nothing during parsing. It cannot
actually reject any parse, because it can always infer that the sort of the argument and parameter are K, and it has no effect on the resulting sort of the term. However, it will nevertheless affect the kore generated from the term by introducing an additional parameter to the symbol generated for the term.

#### `function` and `functional` attributes

Many times it becomes easier to write a semantics if you have "helper" functions written which can be used in the RHS of rules. The `function` attribute tells K that a given symbol should be simplified immediately when it appears anywhere in the configuration. Semantically, it means that evaluation of that symbol will result in at most one return value (that is, the symbol is a *partial function*).

The `functional` attribute indicates to the symbolic reasoning engine that a given symbol is a *total function*, that is it has *exactly* one return value
for every possible input.

For example, here we define the `_+Word_` total function and the `_/Word_` partial function, which can be used to do addition/division modulo
`2 ^Int 256`. These functions can be used anywhere in the semantics where integers should not grow larger than `2 ^Int 256`. Notice how `_/Word_` is *not* defined when the denominator is `0`.

```
syntax Int ::= Int "+Word" Int [function, functional]
```

```
              | Int "/Word" Int [function]

rule I1 +Word I2 => (I1 +Int I2) modInt (2 ^Int 256)
rule I1 /Word I2 => (I1 /Int I2) modInt (2 ^Int 256) requires I2 =/=Int 0
```

#### `freshGenerator` attribute

In K, you can access "fresh" values in a given domain using the syntax `!VARNAME:VarSort` (with the `!`-prefixed variable name). This is supported for builtin sorts `Int` and `Id` already. For example, you can generate fresh memory locations for declared identifiers as such:

```
rule <k> new var x ; => . ... </k>
     <env> ENV => ENV [ x <- !I:Int ] </env>
     <mem> MEM => MEM [ !I <- 0     ] </mem>
```

Each time a `!`-prefixed variable is encountered, a new integer will be used, so each variable declared with `new var _ ;` will get a unique position in the `<mem>`.

Sometimes you want to have generation of fresh constants in a user-defined sort. For this, K will still generate a fresh `Int`, but can use a converter function you supply to turn it into the correct sort. For example, here we can generate fresh `Foo`s using the `freshFoo(_)` function annotated with `freshGenerator`.

```
syntax Foo ::= "a" | "b" | "c" | d ( Int )

syntax Foo ::= freshFoo ( Int ) [freshGenerator, function, functional]

rule freshFoo(0) => a
rule freshFoo(1) => b
rule freshFoo(2) => c
rule freshFoo(I) => d(I) [owise]

rule <k> new var x ; => . ... </k>
     <env> ENV => ENV [ x <- !I:Int  ] </env>
     <mem> MEM => MEM [ !I <- !F:Foo ] </mem>
```

Now each newly allocated memory slot will have a fresh `Foo` placed in it.

#### `token` attribute

The `token` attribute signals to the Kore generator that the associated sort will be inhabited by domain values. Sorts inhabited by domain values must not have any constructors declared.

```
syntax Bytes [hook(BYTES.Bytes), token]
```

### Converting between [token] sorts

You can convert between tokens of one sort via Strings by defining functions implemented by builtin hooks.
The hook STRING.token2string allows conversion of any token to a string:

```
syntax String ::= FooToString(Foo)  [function, functional, hook(STRING.token2string)]
```

Similarly, the hook STRING.string2Token allows the inverse:

```
syntax Bar ::= StringToBar(String) [function, functional, hook(STRING.string2token)]
```

WARNING: This sort of conversion does *NOT* do any sort of parsing or validation.
Thus, we can create arbitary tokens of any sort:

```
StringToBar("The sun rises in the west.")
```

Composing these two functions lets us convert from Foo to Bar

```
syntax Bar ::= FooToBar(Foo) [function]
rule FooToBar(F) => StringToBar(FooToString(F))
```

### Parsing comments, and the #Layout sort

Productions for the #Layout sort are used to describe tokens that are considered "whitespace". The scanner removes tokens matching these productions
so they are not even seen by the parser. Below, we use it to define lines begining with ; (semicolon) as comments.

```
syntax #Layout ::= r"(;[^\\n\\r]*)"    // Semi-colon comments
                 | r"([\\ \\n\\r\\t])" // Whitespace
```

### prec attribute

Consider the following naive attempt at creating a language what syntax that allows two types of variables: names that contain underbars, and names that contain sharps/hashes/pound-signs:

```
syntax NameWithUnderbar ::= r"[a-zA-Z][A-Za-z0-9_]*"  [token]
syntax NameWithSharp    ::= r"[a-zA-Z][A-Za-z0-9_#]*" [token]
syntax Pgm ::= underbar(NameWithUnderbar)
             | sharp(NameWithSharp)
```

Although, it seems that K has enough information to parse the programs underbar(foo) and sharp(foo) with, the lexer does not take into account whether a token is being parsed for the sharp or for the underbar production. It chooses an arbitary sort for the token foo (perhaps NameWithUnderbar). Thus, during paring it is unable to construct a valid term for one of those programs (sharp(foo)) and produces the error message: Inner Parser: Parse error: unexpected token 'foo'.

Since calculating inclusions and intersections between regular expressions is tricky, we must provide this information to K. We do this via the `prec(N)` attribute. The lexer will always prefer longer tokens to shorter tokens. However, when it has to choose between two different tokens of equal length, token productions with higher precedence are tried first. Note that the default precedence value is zero when the `prec` attribute is not specified.

We also need to make sorts with more specific tokens subsorts of ones with more general tokens. We add the token attribute to this production so that all tokens of a particular sort are marked with the sort it is parsed as, and not a subsort thereof. e.g. we get `underbar(#token("foo", "NameWithUnderbar"))` instead of `underbar(#token("foo", "#LowerId"))`

The `BUILTIN-ID-TOKENS` module defines `#UpperId` and `#LowerId` with attributes `prec(2)`.

```
imports BUILTIN-ID-TOKENS
syntax NameWithUnderbar ::= r"[a-zA-Z][A-Za-z0-9_]*" [prec(1), token]
                          | #UpperId                  [token]
                          | #LowerId                  [token]
syntax NameWithSharp ::= r"[a-zA-Z][A-Za-z0-9_#]*" [prec(1), token]
                       | #UpperId                   [token]
                       | #LowerId                   [token]
syntax Pgm ::= underbar(NameWithUnderbar)
             | sharp(NameWithSharp)
```

#### `unused` attribute

K will warn you if you declare a symbol that is not used in any of the rules of your definition. Sometimes this is intentional, however; in this case, you can suppress the warning by adding the `unused` attribute to the production or cell.

```
syntax Foo ::= foo() [unused]

configuration <foo unused=""> .K </foo>
```

### Symbol priority and associativity

Unlike most other parser generators, K combines the task of parsing with AST generation. A production declared with the `syntax` keyword in K is both a piece of syntax used when parsing, and a symbol that is used when rewriting. As a result, it is generally convenient to describe expression grammars using priority and associativity declarations rather than explicitly transforming your grammar into a series of nonterminals, one for each level of operator precedence. Thus, for example, a simple grammar for addition and multiplication
will look like this:

```
syntax Exp ::= Exp "*" Exp
             | Exp "+" Exp
```

However, this grammar is ambiguous. The term `x+y*z` might refer to `x+(y*z)` or to `(x+y)*z`. In order to differentiate this, we introduce a partial ordering between productions known as priority. A symbol "has tighter priority" than another symbol if the first symbol can appear under the second, but the second cannot appear under the first without a bracket. For example, in traditional arithmetic, multiplication has tighter priority than addition, which means that `x+y*z` cannot parse as `(x+y)*z` because the addition operator would appear directly beneath the multiplication, which is forbidden by the priority filter.

Priority is applied individually to each possible ambiguous parse of a term. It then either accepts or rejects that parse. If there is only a single remaining parse (after all the other disambiguation steps have happened), this is the parse that is chosen. If all the parses were rejected, it is a parse error. If multiple parses remain, they might be resolved by further disambiguation such as via the `prefer` and `avoid` attributes, but if multiple parses remain after disambiguation finishes, this is an ambiguous parse error, indicating there is not a unique parse for that term. In the vast majority of cases, this is an error and indicates that you ought to either change your grammar or add brackets to the term in question.

Priority is specified in K grammars by means of one of two different mechanisms. The first, and simplest, simply replaces the `|` operator in a sequence of K productions with the `>` operator. This operator indicates that everything prior to the `>` operator (including transitively) binds tighter than what comes after. For example, a more complete grammar for simple arithmetic might be:

```
syntax Exp ::= Exp "*" Exp
             | Exp "/" Exp
             > Exp "+" Exp
             | Exp "-" Exp
```

This indicates that multiplication and division bind tigher than addition and subtraction, but that there is no relationship in priority between multiplication and division.

As you may have noticed, this grammar is also ambiguous. `x*y/z` might refer to
`x*(y/z)` or to `(x*y)/z`. Indeed, if we removed division and subtraction entirely, the grammar would still be ambiguous: `x*y*z` might parse as `x*(y*z)`, or as `(x*y)*z`. To resolve this, we introduce another feature: associativity. Roughly, asssociativity tells us how symbols are allowed to nest within other symbols with the same priority. If a set of symbols is left associative, then symbols in that set cannot appear as the rightmost child of other symbols in that set. If a set of symbols is right associative, then

symbols in that set cannot appear as the leftmost child of other symbols in that set. Finally, if a set of symbols is non-associative, then symbols in that set cannot appear as the rightmost or leftmost child of other symbols in that set. For example, in the above example, if addition and subtraction are left associative, then `x+y+z` will parse as `(x+y)+z` and `x+y-z` will parse as `(x+y)-z` (because the other parse will have been rejected).

You might notice that this seems to apply only to binary infix operators. In fact, the real behavior is slightly more complicated. Priority and associativity (for technical reasons that go beyond the scope of this document) really only apply when the rightmost or leftmost item in a production is a nonterminal. If the rightmost nonterminal is followed by a terminal (or respectively the leftmost preceded), priority and associativity do not apply. Thus we can generalize these concepts to arbitrary context-free grammars.

Note that in some cases, this is not the behavior you want. You may actually want to reject parses even though the leftmost and rightmost item in a production are terminals. You can accomplish this by means of the `applyPriority` attribute. When placed on a production, it tells the parser which nonterminals of a production the priority filter ought to reject children under, overriding the default behavior. For example, I might have a production like `syntax Exp ::= foo(Exp, Exp) [applyPriority(1)]`. This tells the parser
to reject terms with looser priority binding under the first `Exp`, but not the second. By default, with this production, neither position would apply to the priority filter, because the first and last items of the production are both terminals.

Associativity is specified in K grammars by means of one of two different mechanisms. The first, and simplest, adds the associativity of a priority block of symbols prior to that block. For example, we can remove the remaining ambiguities in the above grammar like so:

```
syntax Exp ::= left:
                 Exp "*" Exp
               | Exp "/" Exp
               > right:
                 Exp "+" Exp
               | Exp "-" Exp
```

This indicates that multiplication and division are left-associative, ie, after symbols with higher priority are parsed as innermost, symbols are nested with the rightmost on top. Addition and subtraction are right associative, which is the opposite and indicates that symbols are nested with the leftmost on top. Note that this is similar but different from evaluation order, which also concerns itself with the ordering of symbols, which is described in the next section.

You may note we have not yet introduced the second syntax for priority

and associativity. In some cases, syntax for a grammar might be spread across multiple modules, sometimes for very good reasons with respect to code modularity. As a result, it becomes infeasible to declare priority and associativity inline within a set of productions, because the productions are not contiguous within a single file.

For this purpose, we introduce the equivalent `syntax priorities`, `syntax left`, `syntax right`, and `syntax non-assoc` declarations. For example, the above grammar can be written equivalently as:

```
syntax Exp ::= Exp "*" Exp [mult]
             | Exp "/" Exp [div]
             | Exp "+" Exp [add]
             | Exp "-" Exp [sub]

syntax priorities mult div > add sub
syntax left mult div
syntax right add sub
```

Here we use user-defined attributes to refer to a group of sentences collectively. The sets are flattened together. We could equivalently have written:

```
syntax Exp ::= Exp "*" Exp [mult]
             | Exp "/" Exp [mult]
             | Exp "+" Exp [add]
             | Exp "-" Exp [add]

syntax priorities mult > add
syntax left mult
syntax right add
```

Note that `syntax [left|right|non-assoc]` should not be used to group together
productions with different priorities. For example, this code would be invalid:

```
syntax priorities mult > add
syntax left mult add
```

Note that there is one other way to describe associativity, but it is prone to a very common mistake. You can apply the attribute `left`, `right`, or `non-assoc` directly to a production to indicate that it is, by itself, left-, right-, or non-associative.

However, this often does not mean what users think it means. In particular:

```
syntax Exp ::= Exp "+" Exp [left]
             | Exp "-" Exp [left]
```

is not equivalent to:

```
syntax Exp ::= left:
                 Exp "+" Exp
               | Exp "-" Exp
```

Under the first, each production is associative with itself, but not each other. Thus, `x+y+z` will parse unambiguously as `(x+y)+z`, but `x+y-z` will be ambiguous. However, in the second, `x+y-z` will parse unambiguously as `(x+y)-z`.

Think carefully about how you want your grammar to parse. In general, if you're
not sure, it's probably best to group associativity together into the same blocks you use for priority, rather than using `left`, `right`, or `non-assoc` attributes on the productions.

### Lexical identifiers

Sometimes it is convenient to be able to give a certain regular expression a name and then refer to it in one or more regular expression terminals. This can be done with a `syntax lexical` sentence in K:

```
syntax lexical Alphanum = r"[0-9a-zA-Z]"
```

This defines a lexical identifier `Alphanum` which can be expanded in any regular expression terminal to the above regular expression. For example, I might choose to then implement the syntax of identifiers as follows:

```
syntax Id ::= r"[a-zA-Z]{Alphanum}*" [token]
```

Here `{Alphanum}` expands to the above regular expression, making the sentence equivalent to the following:

```
syntax Id ::= r"[a-zA-Z]([0-9a-zA-Z])*" [token]
```

This feature can be used to more modularly construct the lexical syntax of your language. Note that K does not currently check that lexical identifiers used in regular expressions have been defined; this will generate an error when creating the scanner, however, and the user ought to be able to debug what happened.

### `assoc`, `comm`, `idem`, and `unit` attributes

These attributes are used to indicate whether a collection or a production is associative, commutative, idempotent, and/or has a unit.
In general, you should not need to apply these attributes to productions yourself, however, they do have certain special meaning to K. K will generate axioms related to each of these concepts into your definition for you automatically. It will also automatically sort associative-commutative collections, and flatten the indentation of associative collections, when unparsing.

#### `public` and `private` attribute

K allows users to declare certain pieces of syntax as either public or private. All syntax is public by default. Public syntax can be used from any module that
imports that piece of syntax. A piece of syntax can be declared private with the `private` attribute. This means that that syntax can only be used in the module in which it is declared; it is not visible from modules that import that module.

You can also change the default visibility of a module with the `private` attribute, when it is placed directly on a module. A module with the `private` attribute has all syntax `private` by default; this can be overridden on specific sentences with the `public` attribute.

Note that the `private` module attribute also changes the default visiblity of imports; please refer to the appropriate section elsewhere in the manual for more details.

Here is an example usage:

```
module WIDGET-SYNTAX

  syntax Widget ::= foo()
  syntax WidgetHelper ::= bar() [private] // this production is not visible
                                          // outside this module
endmodule

module WIDGET [private]
  imports WIDGET-SYNTAX

  syntax Widget ::= fooImpl() // this production is not visible outside this
                              // module

  // this production is visible outside this module
  syntax KItem ::= adjustWidget(Widget) [function, public]
endmodule
```

### Configuration Declaration

#### `exit` attribute

A single configuration cell containing an integer may have the ”exit” attribute. This integer will then be used as the return value on the console when executing the program.

For example:

```
configuration <k> $PGM:Pgm </k>
              <status-code exit=""> 1 </status-code>
```

declares that the cell `status-code` should be used as the exit-code for invocations of `krun`. Additionally, we state that the default exit-code is `1` (an error state). One use of this is for writing testing harnesses which assume that the test fails until proven otherwise and only set the `<status-code>` cell to `0` if the test succeeds.

### Collection Cells: `multiplicity` and `type` attributes

Sometimes a semantics needs to allow multiple copies of the same cell, for example if you are making a concurrent multi-threading programming language. For this purpose, K supports the `multiplicity` and `type` attributes on cells declared in the configuration.

`multiplicity` can take on values `*` and `?`. Declaring `multiplicity="*"` indicates that the cell may appear any number of times in a runtime configuration. Setting `multiplicity="?"` indicates that the cell may only appear exactly 0 or 1 times in a runtime configuration. If there are no configuration variables present in the cell collection, the initial configuration will start with exactly 0 instances of the cell collection. If there are configuration variables present in the cell collection, the initial configuration will start with exactly 1 instance of the cell collection.

`type` can take on values `Set`, `List`, and `Map`. For example, here we declare several collecion cells:

```
configuration <k> $PGM:Pgm </k>
              <sets>  <set  multiplicity="?" type="Set">  0:Int </set>  </sets>
              <lists> <list multiplicity="*" type="List"> 0:Int </list> </lists>
              <maps>
                <map multiplicity="*" type="Map">
                  <map-key> 0:Int </map-key>
                  <map-value-1> "":String </map-value-1>
                  <map-value-2> 0:Int     </map-value-2>
                </map>
              </maps>
```

Declaring `type="Set"` indicates that duplicate occurrences of the cell should be de-duplicated, and accesses to instances of the cell will be nondeterministic choices (constrained by any other parts of the match and side-conditions). Similarly, declaring `type="List"` means that new instances of the cell can be added at the front or back, and elements can be accessed from the front or back, and the order of the cells will be maintained. The following are examples of introduction and elimination rules for these collections:

```
rule <k> introduce-set(I:Int) => . ... </k>
     <sets> .Bag => <set> I </set> </sets>

rule <k> eliminate-set => I ... </k>
```

```
     <sets> <set> I </set> => .Bag </sets>

rule <k> introduce-list-start(I:Int) => . ... </k>
     <lists> (.Bag => <list> I </list>) ... </lists>

rule <k> introduce-list-end(I:Int) => . ... </k>
     <lists> ... (.Bag => <list> I </list>) </lists>

rule <k> eliminate-list-start => I ... </k>
     <lists> (<list> I </list> => .Bag) ... </lists>

rule <k> eliminate-list-end => I ... </k>
     <lists> ... (<list> I </list> => .Bag) </lists>
```

Notice that for `multiplicity="?"`, we only admit a single `<set>` instance at a time. For the `type=List` cell, we can add/eliminate cells from the from or back of the `<lists>` cell. Also note that we use `.Bag` to indicate the empty cell collection in all cases.

Declaring `type="Map"` indicates that the first sub-cell will be used as a cell-key. This means that matching on those cells will be done as a map-lookup operation if the cell-key is mentioned in the rule (for performance). If the cell-key is not mentioned, it will fallback to normal nondeterministic constrained by other parts of the match and any side-conditions. Note that there
is no special meaning to the name of the cells (in this case `<map>`, `<map-key>`, `<map-value-1>`, and `<map-value-2>`). Additionally, any number of
sub-cells are allowed, and the *entire* instance of the cell collection is considered part of the cell-value, including the cell-key (`<map-key>` in this case) and the surrounding collection cell (`<map>` in this case).

For example, the following rules introduce, set, retrieve from, and eliminate `type="Map"` cells:

```
rule <k> introduce-map(I:Int) => . ... </k>
     <maps> ... (.Bag => <map> <map-key> I </map-key> ... </map>) ... </maps>

rule <k> set-map-value-1(I:Int, S:String) => . ... </k>
     <map> <map-key> I </map-key> <map-value-1> _ => S </map-value-1> ... </map>

rule <k> set-map-value-2(I:Int, V:Int) => . ... </k>
     <map> <map-key> I </map-key> <map-value-2> _ => V </map-value-2> ... </map>

rule <k> retrieve-map-value-1(I:Int) => S ... </k>
     <map> <map-key> I </map-key> <map-value-1> S </map-value-1> ... </map>

rule <k> retrieve-map-value-2(I:Int) => V ... </k>
```

```
     <map> <map-key> I </map-key> <map-value-2> V </map-value-2> ... </map>

rule <k> eliminate-map(I:Int) => . ... </k>
     <maps> ... (<map> <map-key> I </map-key> ... </map> => .Bag) ... </maps>
```

Note how each rule makes sure that `<map-key>` cell is mentioned, and we continue to use `.Bag` to indicate the empty collection. Also note that when introducing new map elements, you may omit any of the sub-cells which are
not the cell-key. In case you do omit sub-cells, you must use structural framing `...` to indicate the missing cells, they will receive the default value given in the `configuration ...` declaration.

## Rule Declaration

### Pattern Matching operator

Sometimes when you want to express a side condition, you want to say that a rule matches if a particular term matches a particular pattern, or if it instead does /not/ match a particular pattern.

The syntax in K for this is :=K and :/=K. It has similar meaning to ==K and =/=K, except that where ==K and =/=K express equality, :=K and =/=K express
model membership. That is to say, whether or not the rhs is a member of the set
of terms expressed by the lhs pattern. Because the lhs of these operators is a pattern, the user can use variables in the lhs of the operator. However, due to current limitations, these variables are *NOT* bound in the rest of the term. The user is thus encouraged to use anonymous variables only, although this is not required.

This is compiled by the K frontend down to an efficient pattern matching on a fresh function symbol.

### Anonymous function applications

There are a number of cases in K where you would prefer to be able to take some
term on the RHS, bind it to a variable, and refer to it in multiple different places in a rule.

You might also prefer to take a variable for which you know some of its structure, and modify some of its internal structure without requiring you to match on every single field contained inside that structure.

In order to do this, we introduce syntax to K that allows you to construct anonymous functions in the RHS of a rule and apply them to a term.

The syntax for this is:

`#fun(RuleBody)(Argument)`

Note the limitations currently imposed by the implementation. These functions are not first-order: you cannot bind them to a variable and inject them like you can with a regular klabel for a function. You also cannot express multiple rules or multiple parameters, or side conditions. All of these are extensions we would like to support in the future, however.

In the following, we use three examples to illustrate the behavior of `#fun`. We point out that the support for `#fun` is provided by the frontend, not the backends.

The three examples are real examples borrowed or modified from existing language
semantics.

*Example 1 (A Simple Self-Explained Example).*

```
#fun(V:Val => isFoo(V) andBool isBar(V))(someFunctionReturningVal())
```

*Example 2 (Nested #fun).*

```
   #fun(C
=> #fun(R
=> #fun(E
=> foo1(E, R, C)
  )(foo2(C))
  )(foo3(0))
  )(foo4(1))
```

This example is from the `beacon`
semantics:https://github.com/runtimeverification/beacon-chain-spec/blob/master/b
eacon-chain.k at line 302, with some modification for simplicity. Note how variables `C, R, E` are bound in the nested `#fun`.

*Example 3 (Matching a structure).*

```
rule foo(K, RECORD) =>
  #fun(record(... field: _ => K))(RECORD)
```

Unlike previous examples, the LHS of `#fun` in this example is no longer a variable, but a structure. It has the same spirit as the first two examples, but we match the `RECORD` with a structure `record( DotVar, field: X)`, instead
of a standalone variable. We also use K's local rewrite syntax (i.e., the rewriting symbol `=>` does not occur at the top-level) to prevent writing duplicate expressions on the LHS and RHS of the rewriting.

### Macros and Aliases

A rule can be tagged with the `macro`, `alias`, `macro-rec`, or `alias-rec` attributes. In all cases, what this signifies is that this is a macro rule. Macro rules are applied statically during compilation on all terms that they match, and statically before program execution on the initial configuration. Currently, macros are required to not have side conditions, although they can contain sort checks.

When a rule is tagged with the `alias` attribute, it is also applied statically in reverse prior to unparsing on the final configuration. Note that a macro can have unbound variables in the right hand side. When such a macro exists, it should be used only on the left hand side of rules, unless the user is performing symbolic execution and expects to introduce symbolic terms into the
subject being rewritten.

However, when used on the left hand side of a rule, it functions similarly to a pattern alias, and allows the user to concisely express a reusable pattern that they wish to match on in multiple places.

For example, consider the following semantics:

```
syntax KItem ::= "foo" | "foobar"
syntax KItem ::= bar(KItem) | baz(Int, KItem)
rule foo => foobar [alias]
rule bar(I) => baz(?_, I) [macro]
rule bar(I) => I
```

This will rewrite `baz(0, foo)` to `foo`. First `baz(0, foo)` will be rewritten statically to `baz(0, foobar)`. Then the non-`macro` rule will apply (because the rule will have been rewritten to `rule baz(_, I) => I`). Then `foobar` will be rewritten statically after rewriting finishes to `foo` via the reverse form of the alias.

Note that macros do not apply recursively within their own expansion. This is done so as to ensure that macro expansion will always terminate. If the user genuinely desires a recursive macro, the `macro-rec` and `alias-rec` attributes can be used to provide this behavior.

For example, consider the following semantics:

```
syntax Exp ::= "int" Exps ";" | Exp Exp | Id
syntax Exps ::= List{Exp,","}

rule int X:Id, X':Id, Xs:Exps ; => int X ; int X', Xs ; [macro]
```

This will expand `int x, y, z;` to `int x; int y, z;` because the macro does not apply the second time after applying the substitution of the first application. However, if the `macro` attribute were changed to the `macro-rec`

attribute, it would instead expand (as the user likely intended) to `int x; int y; int z;`.

The `alias-rec` attribute behaves with respect to the `alias` attribute the same way the `macro-rec` attribute behaves with respect to `macro`.

#### `anywhere` rules

Some rules are not functional, but you want them to apply anywhere in the configuration (similar to functional rules). You can use the `anywhere` attribute on a rule to instruct the backends to make sure they apply anywhere they match in the entire configuration.

For example, if you want to make sure that some associative operator is always right-associated anywhere in the configuration, you can do:

```
syntax Stmt ::= Stmt ";" Stmt

rule (S1 ; S2) ; S3 => S1 ; (S2 ; S3) [anywhere]
```

Then after every step, all occurances of `_;_` will be re-associated. Note that this allows the symbol `_;_` to still be a constructor, even though it is simplified similarly to a `function`.

#### `smt-lemma`, `lemma`, and `trusted` attributes

These attributes guide the prover when it tries to apply rules to discharge a proof obligation.

- `smt-lemma` can be applied to a rule to encode it as an equality when sending queries to Z3.
- `lemma` distinguishes normal rules from lemma rules in the semantics, but has no affect.
- `trusted` instructs the prover that it should not attempt proving a given proof obligation, instead trusting that it is true.

### Projection and Predicate functions

K automatically generates certain predicate and projection functions from the syntax you declare. For example, if you write:

```
syntax Foo ::= foo(bar: Bar)
```

It will automatically generate the following K code:

```
syntax Bool ::= isFoo(K) [function]
syntax Foo ::= "{" K "}" ":>Foo" [function]
syntax Bar ::= bar(Foo) [function]

rule isFoo(F:Foo) => true
```

```
rule isFoo(_) => false [owise]

rule { F:Foo }:>Foo => F
rule bar(foo(B:Bar)) => B
```

The first two types of functions are generated automatically for every sort in your K definition, and the third type of function is generated automatically for each named nonterminal in your definition. Essentially, `isFoo` for some sort `Foo` will tell you whether a particular term of sort `K` is a `Foo`, `{F}:>Foo` will cast `F` to sort `Foo` if `F` is of sort `Foo` and will be undefined (i.e., theoretically defined as `#Bottom`, the bottom symbol in matching logic) otherwise. Finally, `bar` will project out the child of a `foo` named `bar` in its production declaration.

Note that if another term of equal or smaller sort to `Foo` exists and has a child named `bar` of equal or smaller sort to `Bar`, this will generate an ambiguity during parsing, so care should be taken to ensure that named nonterminals are sufficiently unique from one another to prevent such ambiguities. Of course, the compiler will generate a warning in this case.

#### `simplification` attribute (Haskell backend)

The simplification attribute identifies rules outside the main semantics that are used to simplify function patterns.

**Conditions**: A simplification rule is applied by *matching* the function arguments, instead of unification as when applying function definition rules. This allows function symbols to appear nested as arguments to other functions on the left-hand side of a simplification rule, which is forbidden in function definition rules. For example, this rule would not be accepted as a function definition rule:

```
rule (X +Int Y) +Int Z => X +Int (Y +Int Z) [simplification]
```

A simplification rule is only applied when the current side condition *implies* the `requires` clause of the rule, like function definition rules.

**Order**: Simplification rules are applied after definition rules, if no definition rule did apply. The `simplification` attribute accepts an optional integer argument which is the rule's *priority*; if the optional argument is not specified, it is equivalent to a priority of 50. Simplification rules are applied in order of their priority. `simplification` rules may not have the `priority` attribute.

For example, for the following definition:

```
    syntax WordStack ::= Int ":" WordStack | ".WordStack"
    syntax Int ::= sizeWordStack    ( WordStack       ) [function]
                 | sizeWordStackAux ( WordStack , Int ) [function]
 // ----------------------------------------------------------------
```

```
rule sizeWordStack(WS) => sizeWordStackAux(WS, 0)

rule sizeWordStackAux(.WordStack, N) => N
rule sizeWordStackAux(W : WS    , N) => sizeWordStackAux(WS, N +Int 1)
```

We might add the following simplification lemma:

```
rule sizeWordStackAux(WS, N) => N +Int sizeWordStackAux(WS, 0)
  requires N =/=Int 0
  [simplification]
```

Then this simplification rule will only apply if the Haskell backend can prove that `notBool N =/=Int 0` is unsatisfiable. This avoids an infinite cycle of applying this simplification lemma.

**NOTE**: The frontend and Haskell backend **do not check** that supplied simplification rules are sound, this is the developer's responsibility. In particular, rules with the simplification attribute must preserve definedness; that is, if the left-hand side refers to any partial function then:

- the right-hand side must be `#Bottom` when the left-hand side is `#Bottom`, or
- the rule must have an `ensures` clause that is `false` when the left-hand side is `#Bottom`, or
- the rule must have a `requires` clause that is `false` when the left-hand side is `#Bottom`.

These conditions are in order of decreasing preference: the best option is to preserve `#Bottom` on the right-hand side, the next best option is to have an `ensures` clause, and the least-preferred option is to have a `requires` clause. The most preferred option is to write total functions and avoid the entire issue.

**NOTE**: The Haskell backend does not attempt to prove claims which right-hand
side is `#Bottom`. The reason for this is that the general case is undecidable, and the backend might enter an infinite loop. Therefore, the backend emits a warning if it encounters such a claim.

#### `concrete` attribute, `#isConcrete` and `#isVariable` function (Java backend)

**NOTE**: The Haskell backend *does not* and *will not* support the meta-functions `#isConcrete` and `#isVariable`. See below for information about
the `concrete` and `symbolic` attributes in the Haskell backend.

Sometimes you only want a given function to simplify if all (or some) of the arguments are concrete (non-symbolic). To do so, you can use either the `concrete` attribute (if you want it to only apply when all arguments are concrete), or the `#isConcrete(_)` side-condition (when you only want it to

apply if some arguments are concrete). Conversely, the function `#isVariable(_)` will only return true when the argument is a variable.

For example, the following will only re-associate terms when all arguments are concrete:

```
rule X +Int (Y +Int Z) => (X +Int Y) +Int Z [concrete]
```

And the following rules will only re-associate terms when it will end up grouping concrete sub-terms:

```
rule X +Int (Y +Int Z) => (X +Int Y) +Int Z
  requires #isConcrete(X)
   andBool #isConcrete(Y)
   andBool #isVariable(Z)

rule X +Int (Y +Int Z) => (X +Int Z) +Int Y
  requires #isConcrete(X)
   andBool #isConcrete(Z)
   andBool #isVariable(Y)
```

#### `concrete` and `symbolic` attributes (Haskell backend)

Sometimes you only want a rule to apply if some or all arguments are concrete (not symbolic). This is done with the `concrete` attribute. Conversely, the `symbolic` attribute will allow a rule to apply only when some arguments are not
concrete. These attributes should only be given with the `simplification` attribute.

For example, the following will only re-associate terms when all arguments are concrete:

```
rule X +Int (Y +Int Z) => (X +Int Y) +Int Z [simplification, concrete]
```

These rules will re-associate and commute terms to combine concrete arguments:

```
rule (A +Int Y) +Int Z => A +Int (Y +Int Z)
  [concrete(Y, Z), symbolic(A), simplification]

rule X +Int (B +Int Z) => B +Int (X +Int Z)
  [concrete(X, Z), symbolic(B), simplification]
```

#### The `unboundVariables` attribute

Normally, K rules are not allowed to contain regular (i.e., not fresh, not existential) variables in the RHS / `requires` / `ensures` clauses which are not bound in the LHS.

However, in certain cases this behavior might be desired, like, for example, when specifying a macro rule which is to be used in the LHS of other rules.

To allow for such cases, but still be useful and perform the unboundness checks in regular cases, the `unboundVariables` attributes allows the user to specify a comma-separated list of names of variables which can be unbound in the rule.

For example, in the macro declaration

```
rule cppEnumType => bar(_, scopedEnum() #Or unscopedEnum() ) [macro, unboundVariables(_)]
```

the declaration `unboundVariables(_)` allows the rule to pass the unbound variable checks, and this in turn allows for `cppEnumType` to be used in the LHS of a rule to mean the pattern above:

```
rule inverseConvertType(cppEnumType, foo((cppEnumType #as T::CPPType => underlyingType(T))
```

### The `memo` attribute

The `memo` attribute is a hint from the user to the backend to memoize a function. Not all backends support memoization, but when the attribute is used and the definition is compiled for a `memo`-supporting backend, then calls to the function may be cached. At the time of writing, only the Haskell backend supports memoization.

**Limitations of memoization with the Haskell backend** The Haskell backend will only cache a function call if all arguments are concrete.

It is recommended not to memoize recursive functions, as each recursive call will be stored in the cache, but only the first iteration will be retrieved from the cache; that is, the cache will be filled with many unreachable entries. Instead, we recommend to perform a worker-wrapper transformation on
recursive functions, and apply the `memo` attribute to the wrapper.

**Warning:** A function declared with the `memo` attribute must not use uninterpreted functions in the side-condition of any rule. Memoizing such an impure function is unsound. To see why, consider the following rules:

```
syntax Bool ::= impure( Int ) [function]

syntax Int ::= unsound( Int ) [function, memo]
rule unsound(X:Int) => X +Int 1 requires impure(X)
rule unsound(X:Int) => X        requires notBool impure(X)
```

Because the function `impure` is not given rules to cover all inputs, `unsound` can be memoized incoherently. For example,

```
{unsound(0) #And {impure(0) #Equals true}} #Equals 1
```

but

```
{unsound(0) #And {impure(0) #Equals false}} #Equals 0
```

The memoized value of `unsound(0)` would be incoherently determined by which pattern the backend encounters first.

### Variable Sort Inference

In K, it is not required that users declare the sorts of variables in rules or in the initial configuration. If the user does not explicitly declare the sort of a variable somewhere via a cast (see below), the sort of the variable is inferred from context based on the sort signature of every place the variable appears in the rule.

As an example, consider the rule for addition in IMP:

```
syntax Exp ::= Exp "+" Exp | Int

rule I1 + I2 => I1 +Int I2
```

Here `+Int` is defined in the INT module with the following signature:

```
syntax Int ::= Int "+Int" Int [function]
```

In the rule above, the sort of both `I1` and `I2` is inferred as `Int`. This is because a variable must have the same sort every place it appears within the same rule. While a variable appearing only on the left-hand-side of the rule could have sort `Exp` instead, the same variable appears as a child of `+Int`, which constriants the sorts of `I1` and `I2` more tightly. Since the sort must be a subsort of `Int` or equal to `Int`, and `Int` has no subsorts, we infer `Int` as the sorts of `I1` and `I2`. This means that the above rule will not match until `I1` and `I2` become integers (i.e., have already been evaluated).

More complex examples are possible, however:

```
syntax Exp ::= Exp "+" Int | Int
rule _ + _ => 0
```

Here we have two anonymous variables. They do not refer to the same variable as one another, so they can have different sorts. The right side is constrained by `+` to be of sort `Int`, but the left side could be either `Exp` or `Int`. When this occurs, we have multiple solutions to the sorts of the variables in the rule. K will only choose solutions which are **maximal**, however. To be precise, if two different solutions exist, but the sorts of one solution are all greater than or equal to the sorts of the other solution, K will discard the smaller solution. Thus, in the case above, the variable on the left side of the `+` is inferred of sort `Exp`, because the solution (`Exp`, `Int`) is strictly greater than the solution (`Int`, `Int`).

It is possible, however, for terms to have multiple maximal solutions:

```
syntax Exp ::= Exp "+" Int | Int "+" Exp | Int
rule I1 + I2 => 0
```

In this example, there is an ambiguous parse. This could parse as either the first `+` or the second. In the first case, the maximal solution chosen is (`Exp`, `Int`). In the second, it is (`Int`, `Exp`). Neither of these solutions is greater than the other, so both are allowed by K. As a result, this program will emit an error because the parse is ambiguous. To pick one solution over the other, a cast or a `prefer` or `avoid` attribute can be used.

### Casting

There are three main types of casts in K: the semantic cast, the strict cast, and the projection cast.

### Semantic casts

For every sort `S` declared in your grammar, K will define the following production for you for use in rules:

```
syntax S ::= S ":S"
```

The meaning of this cast is that the term inside the cast must be less than or equal to `Sort`. This can be used to resolve ambiguities, but its principle purpose is to guide execution by telling K what sort variables must match in order for the rule to apply. When compiled, it will generate a pattern that matches on an injection into `Sort`.

### Strict casts

K also introduces the strict cast:

```
syntax S ::= S "::S"
```

The meaning at runtime is exactly the same as the semantic cast; however, it restricts the sort of the term inside the cast to **exactly** `Sort`. That is to say, if you use it on something that is a strictly smaller sort, it will generate a type error. This is useful in certain circumstances to help disambiguate terms, when a semantic cast would not have resolved the ambiguity.
As such, it is primarily used to solve ambiguities rather than to guide execution.

### Projection casts

K also introduces the projection cast:

```
syntax {S2} S ::= "{" S2 "}" ":>S"
```

The meaning of this cast at runtime is that if the term inside is of sort `Sort`, it should have it injection stripped away and the value inside is returned as a term of static sort `Sort`. However, if the term is of a different sort, it is an error and execution will get stuck. Thus the primary

usefulness of this cast is to cast the return value of a function with a greater sort down to a strictly smaller sort that you expect the return value of the function to have. For example:

```
syntax Exp ::= foo(Exp) [function] | bar(Int) | Int
rule foo(I:Int) => I
rule bar(I) => bar({foo(I +Int 1)}:>Int)
```

Here we know that `foo(I +Int 1)` will return an Int, but the return sort of `foo` is `Exp`. So we project the result into the `Int` sort so that it can be placed as the child of a `bar`.

#### `owise` and `priority` attributes.

Sometimes, it is simply not convenient to explicitly describe every single negative case under which a rule should **not** apply. Instead, we simply wish to say that a rule should only apply after some other set of rules have been tried. K introduces two different attributes that can be added to rules which will automatically generate the necessary matching conditions in a manner which is performant for concrete execution (indeed, it generally outperforms during concrete execution code where the conditions are written explicitly).

The first is the `owise` attribute. Very roughly, rules without an attribute indicating their priority apply first, followed by rules with the `owise` attribute only if all the other rules have been tried and failed. For example, consider the following function:

```
syntax Int ::= foo(Int) [function]
rule foo(0) => 0
rule foo(_) => 1 [owise]
```

Here `foo(0)` is defined explicitly as `0`. Any other integer yields the integer `1`. In particular, the second rule above will only be tried after the first rule has been shown not to apply.

This is because the first rule has a lower number assigned for its priority than the second rule. In practice, each rule in your semantics is implicitly or explicitly assigned a numerical priority. Rules are tried in increasing order of priority, starting at zero and trying each increasing numerical value successively.

You can specify the priority of a rule with the `priority` attribute. For example, I could equivalently write the second rule above as:

```
rule foo(_) => 1 [priority(200)]
```

The number `200` is not chosen at random. In fact, when you use the `owise` attribute, what you are doing is implicitly setting the priority of the rule to `200`. This has a couple of implications:

1. Multiple rules with the owise attribute all have the same priority and thus can apply in any order.
2. Rules with priority higher than `200` apply **after** all rules with the `owise` attribute have been tried.

There is one more rule by which priorities are assigned: a rule with no attributes indicating its priority is assigned the priority 50. Thus, with each priority explicitly declared, the above example looks like:

```
syntax Int ::= foo(Int) [function]
rule foo(0) => 0 [priority(50)]
rule foo(_) => 1 [owise]
```

One final note: the llvm backend reserves priorities between 50 and 150 inclusive for certain specific purposes. Because of this, explicit priorities which are given within this region may not behave precisely as described above. This is primarily in order that it be possible where necessary to provide guidance to the pattern matching algorithm when it would otherwise make bad choices about which rules to try first. You generally should not give any rule a priority within this region unless you know exactly what the implications are with respect to how the llvm backend orders matches.

## Evaluation Strategy

### `strict` and `seqstrict` attributes

The strictness attributes allow defining evaluation strategies without having to explicitely make rules which implement them. This is done by injecting *heating* and *cooling* rules for the subterms. For this to work, you need to define what a *result* is for K, by extending the `KResult` sort.

For example:

```
syntax AExp ::= Int
              | AExp "+" AExp [strict]
```

This generates two heating rules (where the hole syntaxes `"[]" "+" AExp` and `AExp "+" "[]"` is automatically added to create an evaluation context):

```
rule <k> HOLE:AExp +  AE2:AExp => HOLE ~>  [] + AE2 ... </k> [heat]
rule <k>  AE1:AExp + HOLE:AExp => HOLE ~> AE1 +  [] ... </k> [heat]
```

And two corresponding cooling rules:

```
rule <k> HOLE:AExp ~>  [] + AE2 => HOLE +  AE2 ... </k> [cool]
rule <k> HOLE:AExp ~> AE1 +  [] =>  AE1 + HOLE ... </k> [cool]
```

You will note that these rules can apply one after another infinitely. In practice, the `KResult` sort is used to break this cycle by ensuring that only terms that are not part of the `KResult` sort will be heated. The `heat` and `cool` attributes are used to tell the compiler that these are heating and

cooling rules and should be handled in the manner just described. Nothing stops
the user from writing such heating and cooling rules directly if they wish, although we describe other more convenient syntax for most of the advanced cases below.

One other thing to note is that in the above sentences, `HOLE` is just a variable, but it has special meaning in the context of sentences with the `heat` or `cool` attribute. In heating or cooling rules, the variable named `HOLE` is considered to be the term being heated or cooled and the compiler will generate `isKResult(HOLE)` and `notBool isKResult(HOLE)` side conditions appropriately to ensure that the backend does not loop infinitely.

In order for this functionality to work, you need to define the `KResult` sort. For instance, we tell K that a term is fully evaluated once it becomes an `Int` here:

```
syntax KResult ::= Int
```

Note that you can also say that a given expression is only strict only in specific argument positions. Here we use this to define "short-circuiting" boolean operators.

```
syntax KResult ::= Bool

syntax BExp ::= Bool
              | BExp "||" BExp [strict(1)]
              | BExp "&&" BExp [strict(1)]

rule <k> true  || _    => true ... </k>
rule <k> false || REST => REST ... </k>

rule <k> true  && REST => REST  ... </k>
rule <k> false && _    => false ... </k>
```

If you want to force a specific evaluation order of the arguments, you can use the variant `seqstrict` to do so. For example, this would make the boolean operators short-circuit in their *second* argument first:

```
syntax KResult ::= Bool

syntax BExp ::= Bool
              | BExp "||" BExp [seqstrict(2,1)]
              | BExp "&&" BExp [seqstrict(2,1)]

rule <k> _    || true  => true ... </k>
rule <k> REST || false => REST ... </k>

rule <k> REST && true  => REST  ... </k>
```

```
rule <k> _    && false => false ... </k>
```

This will generate rules like this in the case of `_||_` (note that `BE1` will not be heated unless `isKResult(BE2)` is true, meaning that `BE2` must be evaluated first):

```
rule <k>  BE1:BExp || HOLE:BExp => HOLE ~> BE1 ||  [] ... </k> [heat]
rule <k> HOLE:BExp ||  BE2:BExp => HOLE ~>  [] || BE2 ... </k> requires isKResult(BE2) [heat

rule <k> HOLE:BExp ~>  [] || BE2 => HOLE ||  BE2 ... </k> [cool]
rule <k> HOLE:BExp ~> BE1 ||  [] =>  BE1 || HOLE ... </k> [cool]
```

### Context Declaration

Sometimes more advanced evaluation strategies are needed. By default, the `strict` and `seqstrict` attributes are limited in that they cannot describe the *context* in which heating or cooling should occur. When this type of control over the evaluation strategy is required, `context` sentences can be used to simplify the process of declaring heating and cooling when it would be unnecessarily verbose to write heating and cooling rules directly.

For example, if the user wants to heat a term if it exists under a `foo` constructor if the term to be heated is of sort `bar`, one might write the following context:

```
context foo(HOLE:Bar)
```

Once again, note that `HOLE` is just a variable, but one that has special meaning to the compiler indicating the position in the context that should be heated or cooled.

This will automatically generate the following sentences:

```
rule <k> foo(HOLE:Bar) => HOLE ~> foo([]) ... </k> [heat]
rule <k> HOLE:Bar ~> foo([]) => foo(HOLE) ... </k> [cool]
```

The user may also write the K cell explicitly in the context declaration if they want to match on another cell as well, for example:

```
context <k> foo(HOLE:Bar) ... </k> <state> .Map </state>
```

This context will now only heat or cool if the `state` cell is empty.

### Side conditions in context declarations

The user is allowed to write a side condition in a context declaration, like so:

```
context foo(HOLE:Bar) requires baz(HOLE)
```

This side condition will be appended verbatim to the heating rule that is generated, however, it will not affect the cooling rule that is generated:

```
rule <k> foo(HOLE:Bar) => HOLE ~> foo([]) ... </k> requires baz(HOLE) [heat]
rule <k> HOLE:Bar ~> foo([]) => foo(HOLE) ... </k> [cool]
```

### Rewrites in context declarations

The user can also include exactly one rewrite operation in a context declaration if that rule rewrites the variable `HOLE` on the left hand side to a term containing `HOLE` on the right hand side. For exampl;e:

```
context foo(HOLE:Bar => bar(HOLE))
```

In this case, the code generated will be as follows:

```
rule <k> foo(HOLE:Bar) => bar(HOLE) ~> foo([]) ... </k> [heat]
rule <k> bar(HOLE:Bar) ~> foo([]) => foo(HOLE) ... </k> [cool]
```

This can be useful if the user wishes to evaluate a term using a different set of rules than normal.

### `result` attribute

Sometimes it is necessary to be able to evaluate a term to a different sort than `KResult`. This is done by means of adding the `result` attribute to a strict production, a context, or an explicit heating or cooling rule:

```
syntax BExp ::= Bool
              | BExp "||" BExp [seqstrict(2,1), result(Bool)]
```

In this case, the sort check used by `seqstrict` and by the `heat` and `cool` attributes will be `isBool` instead of `isKResult`. This particular example does not really require use of the `result` attribute, but if the user wishes to evaluate a term of sort KResult further, the result attribute would be required.

### `hybrid` attribute

In certain situations, it is desirable to treat a particular production which has the `strict` attribute as a result if the term has had its arguments fully evaluated. This can be accomplished by means of the `hybrid` attribute:

```
syntax KResult ::= Bool

syntax BExp ::= Bool
              | BExp "||" BExp [strict(1), hybrid]
```

This attribute is equivalent in this case to the following additional axiom being added to the definition of `isKResult`:

```
rule isKResult(BE1:BExp || BE2:BExp) => true requires isKResult(BE1)
```

Sometimes you wish to declare a production hybrid with respect to a predicate other than `isKResult`. You can do this by specifying a sort as the body of the `hybrid` attribute, e.g.:

```
syntax BExp ::= BExp "||" BExp [strict(1), hybrid(Foo)]
```

generates the rule:

```
rule isFoo(BE1:BExp || BE2:BExp) => true requires isFoo(BE1)
```

Properly speaking, `hybrid` takes an optional comma-separated list of sort names. If the list is empty, the attribute is equivalent to `hybrid(KResult)`. Otherwise, it generates hybrid predicates for exactly the sorts named.

### Context aliases

Sometimes it is necessary to define a fairly complicated evaluation strategy for a lot of different operators. In this case, the user *could* simply write a number of complex `context` declarations, however, this quickly becomes tedious. For this purpose, K has a concept called a *context alias*. A context alias is a bit like a template for describing contexts. The template can then be instantiated against particular productions using the `strict` and `seqstrict` attributes.

Here is a (simplified) example taken from the K semantics of C++:

```
context alias [c]: <k> HERE:K ... </k> <evaluate> false </evaluate>
context alias [c]: <k> HERE:K ... </k> <evaluate> true </evaluate> [result(ExecResult)]

syntax Expr ::= Expr "=" Init [strict(c; 1)]
```

This defines the evaluation strategy during the translation phase of a C++ program for the assignment operator. It is equivalent to writing the following context declarations:

```
context <k> HOLE:Expr = I:Init ... </k> <evaluate> false </evaluate>
context <k> HOLE:Expr = I:Init ... </k> <evaluate> true </evaluate> [result(ExecResult)]
```

What this is saying is, if the `evaluate` cell is false, evaluate the term like normal to a `KResult`. But if the `evaluate` cell is true, instead evaluate it to the `ExecResult` sort.

Essentially, we have given a name to this evaluation strategy in the form of the rule label on the context alias sentences (in this case, `c`). We can then say that we want to use this evaluation strategy to evaluate particular arguments of particular productions by referring to it by name in a `strict` attribute. For example, `strict(c)` will instantiate these contexts once for each argument of the production, whereas `strict(c; 1)` will instantiate it only for the first argument. The special variable `HERE` is used to tell the compiler where you want to place the production that is to be heated or cooled.

You can also specify multiple context aliases for different parts of a production, for example:

```
syntax Exp ::= foo(Exp, Exp) [strict(left; 1; right; 2)]
```

This says that we can evaluate the left and right arguments in either order, but to evaluate
the left using the `left` context alias and the right using the `right` context alias.

We can also say `seqstrict(left; 1; right; 2)`, in which case we additionally must evaluate
the left argument before the right argument. Note, all strict positions are considered collectively
when determining the evaluation order of `seqstrict` or the `hybrid` predicates.

A `strict` attribute with no rule label associated with it is equivalent to
a `strict` attribute given with the following context alias:

```
context alias [default]: <k> HERE:K ... </k>
```

One syntactic convenience that is provided is that if you wish to declare the following context:

```
context foo(HOLE => bar(HOLE))
```

you can simply write the following:

```
syntax Foo ::= foo(Bar) [strict(alias)]

context alias [alias]: HERE [context(bar)]
```

## Pattern Matching

### As Patterns

New syntax has been added to K for matching a pattern and binding the resulting
match in its entirety to a variable.

The syntax is:

```
Pattern #as V::Var
```

In this case, Pattern, including any variables, is matched and the resulting variables are added to the substitution if matching succeeds. Furthermore, the term matched by Pattern is added to the substitution as V.

This code can also be used outside of any rewrite, in which case matching occurs as if it appeared on the left hand side, and the right hand side becomes a variable corresponding to the alias.

It is an error to use an as pattern on the right hand side of a rule.

### Record-like KApply Patterns

We have added a syntax for matching on KApply terms which mimics the record syntax in functional languages. This allows us to more easily express patterns involving a KApply term in which we don't care about some or most of the children, without introducing a dependency into the code on the number of arguments which could be changed by a future refactoring.

The syntax is:

```
record(... field1: Pattern1, field2: Pattern2)
```

Note that this only applies to productions that are prefix productions.
A prefix production is considered by the implementation to be any production whose production items match the following regular expression:

```
(Terminal(_)*) Terminal("(")
(NonTerminal (Terminal(",") NonTerminal)* )?
Terminal(")")
```

In other words, any sequence of terminals followed by an open parenthesis, an optional comma separated list of non-terminals, and a close parenthesis.

If a prefix production has no named nonterminals, a `record(...)` syntax is allowed, but in order to reference specific fields, it is necessary to give one or more of the non-terminals in the production names.

Note: because the implementation currently creates one production per possible set of fields to match on, and because all possible permutations of all possible subsets of a list of n elements is a number that scales factorially and reaches over 100 thousand productions at n=8, we currently do not allow fields to be matched in any order like a true record, but only in the same order as appears in the production itself.

Given that this only reduces the number of productions to the size of the power set, this will still explode the parsing time if we create large productions of 10 or more fields that all have names. This is something that should probably be improved, however, productions with that large of an arity are rare, and thus it has not been viewed as a priority.

### Or Patterns

Sometimes you wish to express that a rule should match if one out of multiple patterns should match the same subterm. We can now express this in K by means
of using the `#Or` ML connective on the left hand side of a rule.

For example:

```
rule foo #Or bar #Or baz => qux
```

Here any of foo, bar, or baz will match this rule. Note that the behavior is ill-defined if it is not the case that all the clauses of the or have the same bound variables.

### Matching global context in function rules

On occasion it is highly desirable to be able to look up information from the global configuration and match against it when evaluating a function. For this purpose, we introduce a new syntax for function rules.

This syntax allows the user to match on *function context* from within a function rule:

```
syntax Int ::= foo(Int) [function]

rule [[ foo(0) => I ]]
     <bar> I </bar>

rule something => foo(0)
```

This is completely desugared by the K frontend and does not require any special support in the backend. It is an error to have a rewrite inside function context, as we do not currently support propagating such changes back into the global configuration. It is also an error if the context is not at the top level of a rule body.

Desugared code:

```
syntax Int ::= foo(Int, GeneratedTopCell) [function]

rule foo(0, <generatedTop>
              <bar> I </bar>
              ...
            </generatedTop> #as Configuration) => I
rule <generatedTop>
       <k> something ... </k>
       ...
     </generatedTop> #as Configuration
  => <generatedTop>
       <k> foo(0, Configuration> ... </k>
       ...
     </generatedTop>
```

### Collection patterns

It is allowed to write patterns on the left hand side of rules which refer to complex terms of sort Map, List, and Set, despite these patterns ostensibly breaking the rule that terms which are functions should not appear on the left

hand side of rules. Such terms are destructured into pattern matching operations.

The following forms are allowed:

```
// 0 or more elements followed by 0 or 1 variables of sort List followed by
// 0 or more elements
ListItem(E1) ListItem(E2) L:List ListItem(E3) ListItem(E4)

// the empty list
.List

// 0 or more elements in any order plus 0 or 1 variables of sort Set
// in any order
SetItem(K1) SetItem(K2) S::Set SetItem(K3) SetItem(K4)

// the empty set
.Set

// 0 or more elements in any order plus by 0 or 1 variables of sort Map
// in any order
K1 |-> E1 K2 |-> E2 M::Map K3 |-> E3 K4 |-> E4

// the empty map
.Map
```

Here K1, K2, K3, K4 etc can be any pattern except a pattern containing both function symbols and unbound variables. An unbound variable is a variable whose binding cannot be determined by means of decomposing non-set-or-map patterns or map elements whose keys contain no unbound variables.

This is determined recursively, ie, the term `K1 |-> E2 E2 |-> E3 E3 |-> E4` is considered to contain no unbound variables.

Note that in the pattern `K1 |-> E2 K3 |-> E4 E4 |-> E5`, K1 and K3 are unbound, but E4 is bound because it is bound by deconstructing the key E3, even though E3 is itself unbound.

In the above examples, E1, E2, E3, and E4 can be any pattern that is normally allowed on the lhs of a rule.

When a map or set key contains function symbols, we know that the variables in that key are bound (because of the above restriction), so it is possible to evaluate the function to a concrete term prior to performing the lookup.

Indeed, this is the precise semantics which occurs; the function is evaluated and the result is looked up in the collection.

For example:

```
syntax Int ::= f(Int) [function]
rule f(I:Int) => I +Int 1
rule <k> I:Int => . ... </k> <state> ... SetItem(f(I)) ... </state>
```

This will rewrite `I` to `.` if and only if the state cell contains `I +Int 1`.

Note that in the case of Set and Map, one guarantee is that K1, K2, K3, and K4
represent /distinct/ elements. Pattern matching fails if the correct number of distinct elements cannot be found.

### Matching on cell fragments

K allows matching fragments of the configuration and using them to construct terms and use as function parameters.

```
configuration <t>
                <k> #init ~> #collectOdd ~> $PGM </k>
                <fs>
                  <f multiplicity="*" type="Set"> 1 </f>
                </fs>
              </t>
```

The `#collectOdd` construct grabs the entire content of the `<fs>` cell.
We may also match on only a portion of its content. Note that the fragment must be wrapped in a `<f>` cell at the call site.

```
syntax KItem ::= "#collectOdd"
rule <k> #collectOdd => collectOdd(<fs> Fs </fs>) ... </k>
     <fs> Fs </fs>
```

The `collectOdd` function collects the items it needs

```
syntax Set ::= collectOdd(FsCell) [function]
rule collectOdd(<fs> <f> I </f> REST </fs>) => SetItem(I) collectOdd(<fs> REST </fs>) requi
rule collectOdd(<fs> <f> I </f> REST </fs>) =>            collectOdd(<fs> REST </fs>) requi
rule collectOdd(<fs> .Bag </fs>) => .Set
```

### `all-path` and `one-path` attributes to distinguish reachability claims

As the Haskell backend can handle both one-path and all-path reachability claims, but both these are encoded as rewrite rules in K, these attributes can be used to clarify what kind of claim a rule is.

In addition of being able to annotate a rule with one of them
(if annotating with more at the same time, only one of them would be chosen),
one can also annotate whole modules, to give a default claim type for all rules
in that module.

Additionally, the Haskell backend introduces an extra command line option
for the K frontend, `--default-claim-type`, with possible values
`all-path` and `one-path` to allow choosing a default type for all
claims.

### Set Variables

**Motivation** Set variables were introduced as part of Matching Mu Logic, the mathematical
foundations for K. In Matching Mu Logic, terms evaluate to sets of values.
This is useful for both capturing partiality (as in `3/0`) and capturing
non-determinism (as in `3 #Or 5`). Consequently, symbol interpretation is
extended to have a collective interpretation over sets of input values.

Usually, K rules are given using regular variables, which expect that the term
they match is both defined and has a unique interpretation.

However, it is sometimes useful to have simplification rules which work over
any kind of pattern, be it undefined or non-deterministic. This behavior can be
achieved by using set variables to stand for any kind of pattern.

**Syntax** Any variable prefixed by `@` will be considered a set variable.

**Example** Below is a simplification rule which motivated this extension:

```
rule #Ceil(@I1:Int /Int @I2:Int) =>
  {(@I2 =/=Int 0) #Equals true} #And #Ceil(@I1) #And #Ceil(@I2)
  [anywhere]
```

This rule basically says that `@I1:Int /Int @I2:Int` is defined if `@I1` and
`@I2` are defined and `@I2` is not 0. Using sets variables here is important as
it allows the simplification rule to apply *any* symbolic patterns, without
caring whether they are defined or not.

This allows simplifying the expression `#Ceil((A:Int /Int B:Int) / C:Int)`
to:

```
{(C =/=Int 0) #Equals true} #And #Ceil(C) #And ({(B =/=Int 0) #Equals true}
#And #Ceil(B) #And #Ceil(A)`
```

See kframework/kore#729 for
more details.

**SMT Translation** K makes queries to an SMT solver (Z3) to discharge proof obligations when doing
symbolic execution. You can control how these queries are made using the attributes `smtlib` and `smt-hook` on declared productions.

- `smt-hook(...)` allows you to specify a term in SMTLIB2 format which should
  be used to encode that production, and assumes that all symbols appearing in
  the term are already declared by the SMT solver.
- `smtlib(...)` allows you to declare a new SMT symbol to be used when that
  production is sent to Z3, and gives it *uninterpreted function* semantics.

```
syntax Int ::= "~Int" Int          [function, klabel(~Int_), symbol,
                                    smtlib(notInt)]
             | Int "^%Int" Int Int [function, klabel(_^%Int__), symbol,
                                    smt-hook((mod (^ #1 #2) #3))]
```

In the example above, we declare two productions `~Int_` and `_^%Int__`, and tell the SMT solver to:

- use uninterpreted function semantics for `~Int_` via SMTLIB2 symbol `notInt`, and
- use the SMTLIB2 term `(mod (^ #1 #2) #3)` (where `#N` marks the `N`th production non-terminal argument positions) for `_^%Int__`, where `mod` and
  `^` already are declared by the SMT solver.

**Caution** Set variables are currently only supported by the Haskell backend. The use of rules with set variables should be sound for all other backends which just execute by rewriting, however it might not be safe for backends which want to guarantee coverage.

### Variables occurring only in the RHS of a rule

This section presents possible scenarios requiring variables to only appear in the RHS of a rule.

**Summary** Except for `?` variables and `!` (fresh) variables, which are **required** to only appear in the RHS of a rule, all other variables **must** also appear in the LHS of a rule. This restriction also applies to anonymous variables; in particular, for claims, `?_` (not `_`) should be used in the RHS to indicate that something changes but we don't care to what value.

To support specifying random-like behavior, the above restriction can be relaxed by annotating a rule with the `unboundVariables` attribute whenever the rule intentionally contains regular variables only occurring in the RHS.

**Introduction** K uses question mark variables of the form `?X` to refer to existential variables, and uses `ensures` to specify logical constraints on those variables.
These variables are only allowed to appear in the RHS of a K rule.

If the rules represent rewrite (semantic) steps or verification claims, then the `?` variables are existentially quantified at the top of the RHS; otherwise, if they represent equations, the `?` variables are quantified at the top of the entire rule.

Note that when both `?`-variables and regular variables are present, regular variables are (implicitly) universally quantified on top of the rule (already containing the existential quantifications).
This essentially makes all `?` variables depend on all regular variables.

All examples below are intended more for program verification / symbolic execution, and thus concrete implementations might choose to ignore them altogether or to provide ad-hoc implementations for them.

**Example: Verification claims** Consider the following definition of a (transition) system:

```
module A
  rule foo => true
  rule bar => true
  rule bar => false
endmodule
```

Consider also, the following specification of claims about the definition above:

```
module A-SPEC
  rule [s1]: foo => ?X:Bool
  rule [s2]: foo =>  X:Bool  [unboundVariables(X)]
  rule [s3]: bar => ?X:Bool
  rule [s4]: bar =>  X:Bool  [unboundVariables(X)]
endmodule
```

**One-path interpretation**

- (s1) says that there exists a path from `foo` to some boolean, which is satisfied easily using the `foo => true` rule
- (s3) says the same thing about `bar` and can be satisfied by either of `bar => true` and `bar => false` rules
- (s2) and (s4) can be better understood by replacing them with instances for
  **each** element of type `Bool`, which can be interpreted that both `true` and `false` are reachable from `foo` for (s2), or `bar` for (s4), respectively.
  - (s2) cannot be verified as we cannot find a path from `foo` to `false`.

  - (s4) can be verified by using `bar => true` to show `true` is reachable
    and
    `bar => false` to achieve the same thing for `false`

**All-path interpretation**

- (s1) says that all paths from `foo` will reach some boolean, which is satisfied by the `foo => true` rule and the lack of other rules for `foo`
- (s3) says the same thing about `bar` and can be satisfied by checking that both `bar => true` and `bar => false` end in a boolean, and there are no other rules for `bar`
- (s2) and (s4) can be better understood by replacing them with instances for
  **each** element of type `Bool`, which can be interpreted that
  both `true` and `false` are reachable **in all paths** originating in
  `foo` for (s2), or `bar` for (s4), respectively.
  This is a very strong claim, requiring that all paths originating in
  `foo` (`bar`) pass through **both** `true` and `false`,
  so neither (s2) nor (s4) can be verified.

  Interestingly enough, adding a rule like `false => true` would make both (s2) and (s4) hold.

**Example: Random Number Construct `rand()`** The random number construct `rand()` is a language construct which could be
easily conceived to be part of the syntax of a programming language:

```
Exp ::= "rand" "(" ")"
```

The intended semantics of `rand()` is that it can rewrite to any integer in a single step. This could be expressed as the following following infinitely many rules.

```
rule  rand() => 0
rule  rand() => 1
rule  rand() => 2
  ...    ...
rule rand() => (-1)
rule rand() => (-2)
  ...    ...
```

Since we need an instance of the rule for **every** integer, one could summarize the above infinitely many rules with the rule

```
rule rand() => I:Int [unboundVariables(I)]
```

Note that `I` occurs only in the RHS in the rule above, and thus the rule needs the `unboundVariables(I)` attribute to signal that this is intentionally.

One can define variants of `rand()` by further constraining the output variable as a precondition to the rule.

**Rand-like examples**

1. `randBounded(M,N)` can rewrite to any integer between `M` and `N`

   ```
   syntax Exp ::= randBounded(Int, Int)
   rule randBounded(M, N) => I
     requires M <=Int I andBool I <=Int N
     [unboundVariables(I)]
   ```

2. `randInList(Is)` takes a list `Is` of items
   and can rewrite in one step to any item in `Is`.

   ```
   syntax Exp ::= randInList (List)
   rule randInList(Is) => I
     requires I inList Is
     [unboundVariables(I)]
   ```

3. `randNotInList(Is)` takes a list `Is` of items
   and can rewrite in one step to any item *not* in `Is`.

   ```
   syntax Exp ::= randNotInList (List)
   rule randNotInList(Is) => I
     requires notBool(I inList Is)
     [unboundVariables(I)]
   ```

4. `randPrime()`, can rewrite to any *prime number*.

   ```
   syntax Exp ::= randPrime ()
   rule randPrime() => X:Int
     requires isPrime(X)
     [unboundVariables(X)]
   ```

   where `isPrime(_)` is a predicate that can be defined in the usual way.

Note 1: all above are not function symbols, but language constructs.

Note 2: Currently the frontend does not allow rules with universally quantified variables in the RHS which are not bound in the LHS.

Note 3. Allowing these rules in a concrete execution engine would require an algorithm for generating concrete instances for such variables, satisfying the given constraints; thus the `unboundVariables` attribute serves two purposes:

- to allow such rules to pass the variable checks, and
- to signal (concrete execution) backends that specialized algorithm would be
  needed to instantiate these variables.

**Example: Fresh Integer Construct `fresh(Is)`** The fresh integer construct `fresh(Is)` is a language construct.

```
Exp ::= ... | "fresh" "(" List{Int} ")"
```

The intended semantics of `fresh(Is)` is that it can always rewrite to an integer that in not in `Is`.

Note that `fresh(Is)` and `randNotInList(Is)` are different; the former does not *need* to be able to rewrite to every integers not in `Is`, while the latter requires so.

For example, it is *correct* to implement `fresh(Is)` so it always returns the smallest positive integer that is not in `Is`, but same implementation for `randNotInList(Is)` might be considered *inadequate*.
In other words, there exist multiple correct implementations of `fresh(Is)`, some of which may be deterministic, but there only exists a unique implementation of `randNotInList(Is)`.
Finally, note that `randNotInList(Is)` is a *correct* implementation for `fresh(Is)`; Hence, concrete execution engines can choose to handle such rules accordingly.

We use the following K syntax to define `fresh(Is)`

```
syntax Exp ::= fresh (List{Int})
rule fresh(Is:List{Int}) => ?I:Int
  ensures notBool (?I inList{Int} Is)
```

A variant of this would be a `choiceInList(Is)` language construct which would choose some number from a list:

```
syntax Exp ::= choiceInList (List{Int})
rule choiceInList(Is:List{Int}) => ?I:Int
  ensures ?I inList{Int} Is
```

Note: This definition is different from one using a `!` variable to indicate freshness because using `!` is just syntactic sugar for generating globally unique instances and relies on a special configuration cell, and cannot be constrained, while the `fresh` described here is local and can be constrained. While the first is more appropriate for concrete execution, this might be better for symbolic execution / program verification.

**Example: Arbitrary Number (Unspecific Function) `arb()`** The function `arb()` is not a PL construct, but a mathematical function.
Therefore, its definition should not be interpreted as an execution step, but rather as an equality.

The intended semantics of `arb()` is that it is an unspecified nullary function. The exact return value of `arb()` is unspecified in the semantics but up to the implementations.

However, being a mathematical function, `arb()` must return the same value in any one implementation.

We do not need special frontend syntax to define `arb()`.
We only need to define it in the usual way as a function
(instead of a language construct), and provide no axioms for it.
The `functional` attribute ensures that the function is total, i.e.,
that it evaluates to precisely one value for each input.

**Variants** There are many variants of `arb()`. For example, `arbInList(Is)` is an unspecified function whose return value must be an element from `Is`.

Note that `arbInList(Is)` is different from `choiceInList(Is)`, because `choiceInList(Is)` *transitions* to an integer in `Is` (could be a different one each time it is used), while `arbInList(Is)` *is equal to* a (fixed) integer not in `Is`.

W.r.t. the `arb` variants, we can use `?` variables and the `function` annotation to signal that we're defining a function and the value of the function is fixed, but non-determinate.

```
syntax Int ::= arbInList(List{Int}) [function]
rule arbInList(Is:List{Int}) => ?I:Int
  ensures ?I inList{Int} Is
```

If elimination of existentials in equational rules is needed, one possible approach would be through Skolemization,
i.e., replacing the `?` variable with a new uninterpreted function depending on the regular variables present in the function.

**Example: Interval (Non-function Symbols) `interval()`** The symbol `interval(M,N)` is not a PL construct, nor a function in the first-order sense, but a proper matching-logic symbol, whose interpretation is in the powerset of its domain.
Its axioms will not use rewrites but equalities.

The intended semantics of `interval(M,N)` is that it equals the *set* of integers that are larger than or equal to `M` and smaller than or equal to `N`.

Since expressing the axiom for `interval` requires an an existential quantification on the right-hand-side, thus making it a non-functional symbol defined through an equation, using `?` variables might be confusing since their usage would be different from that presented in the previous sections.

Hence, the proposal to support this would be to write this as a proper ML rule. A possible syntax for this purpose would be:

```
eq  interval(M,N)
    ==
```

```
#Exists X:Int .
    (X:Int #And { X >=Int M #Equals true } #And { X <=Int N #Equals true })
```

Additionally, the symbol declaration would require a special attribute to signal the fact that it is not a constructor but a *defined* symbol.

Since this feature is not clearly needed by K users at the moment, it is only presented here as an example; its implementation will be postponed for such time
when its usefulness becomes apparent.

## Parser Generation

In addition to on-the-fly parser generation using `kast`, K is capable of ahead-of-time parser generation of LR(1) or GLR parsers using Flex and Bison. This can be done one of two different ways.

1. You can explicitly request for a particular parser to be generated by invoking `kast --gen-parser <outputFile>` or
   `kast --gen-glr-parser <outputFile>` respectively. `kast` will then create a
   parser based on the same command line flags that govern on-the-fly parsing,
   like `-s` to specify the starting sort, and `-m` to specify the module to parse under. By default, this generates a parser for the sort of the `$PGM` configuration variable in the main syntax module of the definition.
2. You can request that a specific set of parsers be generated for all the configuration variables of your definition by passing the `--gen-bison-parser` or `--gen-glr-bison-parser` flags to `kompile`. `kompile` will decide the sorts to use as start symbols based on the sorts in the configuration declaration for the configuration variables. The `$PGM` configuration variable will be generated based on the main syntax module of the definition. The user must explicitly annotate the configuration declaration with the other modules to use to parse the other configuration variables as attributes. For example, if I have the following cell in the configuration declaration: `<cell> foo($FOO:Foo, $BAR:Bar) </cell>`, One might annotate it with the attribute pair `parser="FOO, TEST; BAR, TEST2"`
   to indicate that configuration variable `$FOO` should be parsed in the `TEST` module, and configuration variable `$BAR` should be parsed in the `TEST2` module. If the user forgets to annotate the declaration with the parser attribute, only the `$PGM` parser will be generated.

Bison-generated parsers are extremely fast compared to `kast`, but they have some important limitations:

- Bison parsers will always output Kore. You can then pass the resulting AST

directly to `llvm-krun` or `kore-exec` and bypass the `krun` frontend, making them very fast, but lower-level.

- Bison parsers do not yet support macros. This may change in a future release.
  Note that you can use anywhere rules instead of macros in most cases to get
  around this limitation, although they will not benefit from unparsing via the
  `alias` attribute.
- Obligation falls on the user to ensure that the grammar they write is LR(1) if they choose to use LR(1) parsing. If this does not happen, the parser generated will have shift/reduce or reduce/reduce conflicts and the parser may behave differently than `kast` would (`kast` is a GLL parser, ie, it is based on LL parsers and parses all unambiguous context-free grammars). K provides an attribute, `not-lr1`, which can be applied to modules known to
  not be LR(1), and will trigger a warning if the user attempts to generate an
  LR(1) parser which recursively imports that module.
- If you are using LR(1) based parsing, the `prefer` and `avoid` attributes are
  ignored. It is only possible to implement these attributes by means of generalized LL or LR parsing and a postprocessing on the AST to remove the
  undesirable ambiguity.
- Obligation falls on the user to ensure that the grammar they write has as few conflicts as possible if they are using GLR parsing. Bison's GLR support
  is quite primitive, and in the worst case it can use exponential space and time to parse a program, which generally leads the generated parser to report
  "memory exhausted", indicating that the parse could not be completed within
  the stack space allocated by Bison. It's best to ensure that the grammar is
  as close to LR(1) as possible and only utilizes conflicts where absolutely necessary. One tool that can be used to facilitate this is to pass `--bison-lists` to kompile. This will disable support for the `List{Sort}` syntax production, and it will make `NeList{Sort}` left associative, but the
  resulting productions generated for `NeList{Sort}` will be LR(1) and use bounded
  stack space.
- If the grammar you are parsing is context-sensitive (for example, because it requires a symbol table to parse), one thing you can do to make this language parse in K is to implement the language as an ambiguous gram-

mar.
Bison's GLR parser will generate an `amb` production that is parametric in the sort of the ambiguity. You can then import the `K-AMBIGUITIES` module
and use rewriting to resolve the ambiguities using whatever preprocessing mechanisms you prefer.

## Location Information

K is able to insert file, line, and column metadata into the parse tree on a per-sort basis when parsing using a bison-generated parser. To enable this, mark the sort with the `locations` attribute.

```
syntax Exp [locations]
syntax Exp ::= Exp "/" Exp | Int
```

K implicitly wraps productions of these sorts in a `#location` term (see the `K-LOCATIONS` module in `kast.md`). The metadata can thus be accessed with ordinary rewrite rules:

```
rule #location(_ / 0, File, StartLine, _StartColumn, _EndLine, _EndColumn) =>
"Error: Division by zero at " +String File +String ":" Int2String(StartLine)
```

Sometimes it is desirable to allow code to be written in a file which overwrites the current location information provided by the parser. This can be done via a combination of the `#LineMarker` sort and the `--bison-file` flag to the parser generator. If you declare a production of sort `#LineMarker` which contains a regular expression terminal, this will be treated as a
**line marker** by the bison parser. The user will then be expected to provide an implementation of the parser for the line marker in C. The function expected by the parser has the signature `void line_marker(char *, yyscan_t)`, where `yyscan_t` is a
reentrant flex scanner.
The string value of the line marker token as specified by your regular expression can be found in the first parameter of the function, and you can set the line number used by the scanner using `yyset_lineno(int, yyscan_t)`. If
you declare the variable `extern char *filename`, you can also set the current file name by writing a malloc'd, zero-terminated string to that variable.

## Unparsing

A number of factors go into how terms are unparsed in K. Here we describe some
of the features the user can use to control how unparsing happens.

### Brackets

One of the phases that the unparser goes through is to insert productions tagged with the `bracket` attribute where it believes this is necessary in order to create a correct string that will be parsed back into the original AST. The most common case of this is in expression grammars. For example, consider the following grammar:

```
syntax Exp ::= Int
             | Exp "*" Exp
             > Exp "+" Exp
```

Here we have declared that expressions can contain integer addition and multiplication, and that multiplication binds tighter than addition. As a result, when writing a program, if we want to write an expression that first applies addition, then multiplication, we must use brackets: `(1 + 2) * 3`. Similarly, if we have such an AST, we must **insert** brackets into the AST in order to faithfully unparse the term in a manner that will be parsed back into the same ast, because if we do not, we end up unparsing the term as `1 + 2 * 3`, which will be parsed back as `1 + (2 * 3)` because of the priority declaration in the grammar.

You can control how the unparser will insert such brackets by adding a production with the `bracket` attribute and the correct sort. For example, if, instead of parentheses, you want to use curly braces, you could write:

```
syntax Exp ::= "{" Exp "}" [bracket]
```

This would signal to the unparser how brackets should look for terms of sort `Exp`, and it will use this syntax when unparsing terms of sort `Exp`.

### Commutative collections

One thing that K will do (unless you pass the `--no-sort-collections` flag to krun) is to sort associative, commutative collections (such as `Set` and `Map`) alphanumerically. For example, if I have a collection whose keys are sort `Id` and they have the values a, b, c, and d, then unparsing will always print first the key a, then b, then c, then d, because this is the alphabetic order of these keys when unparsed.

Furthermore, K will sort numeric keys numerically. For example, if I have a collection whose keys are `1, 2, 5, 10, 30`, it will first display 1, then 2, then 5, then 10, then 30, because it will sort these keys numerically. Note that this is different than an alphabetic sort, which would sort them as `1, 10, 2, 30, 5`. We believe the former is more intuitive to users.

### Substitution filtering

K will remove substitution terms corresponding to anonymous variables when using the `--pattern` flag if those anonymous variables provide no information

about the named variables in your serach pattern. You can disable this behavior by passing `--no-substitution-filtering` to krun. When this flag is not passed,
and you are using the Haskell backend, any equality in a substitution (ie, an `#Equals` under an `#And` under an `#Or`), will be hidden from the user if the left hand side is a variable that was anonymous in the `--pattern` passed by the user, unless that variable appears elsewhere in the substitution. If you want to see that variable in the substitution, you can either disable this filtering, or give that variable a name in the original search pattern.

### Variable alpha renaming

K will automatically rename variables that appear in the output configuration. Similar to commutative collections, this is done to **normalize** the resulting configuration so that equivalent configurations will be printed identically regardless of how they happen to be reached. This pass can be disabled by passing `--no-alpha-renaming` to krun.

### Macro expansion

K will apply macros in reverse on the output configuration if the macro was created with the `alias` or `alias-rec` attribute. See the section on macro expansion for more details.

### Formatting

**`format` attribute** K allows you to control how terms are unparsed using the `format` attribute.
By default, a domain value is unparsed by printing its string value verbatim, and an application pattern is unparsed by printing its terminals and children in the sequence implied by its concrete syntax, separated by spaces. However, K gives you complete control over how you want to unparse the symbol.

A format attribute is a string containing zero or more escape sequences that tell K how to unparse the symbol. Escape sequences begin with a '%' and are followed by either an integer, or a single non-digit character. Below is a list of escape sequences recognized by the formatter:

| Escape Sequence | Meaning |
|---|---|
| n | Insert '\n' followed by the current indentation level |
| i | Increase the current indentation level by 1 |
| d | Decrease the current indentation level by 1 |
| c | Move to the next color in the list of colors for this production |
| r | Reset color to the default foreground color for the terminal (See below for more informatio |
| an integer | Print a terminal or nonterminal from the production (See below for more information) |
| any other char | Print that character verbatim |

**Using the integer escape sequence** In the integer escape sequence `%a`, the integer `a` is treated as a 1-based
index into the terminals and nonterminals of the production.

- If the offset refers to a terminal, move to the next color in the list of colors for this production, print the value of that terminal, then reset the color to the default foreground color for the terminal.
- If the offset refers to a regular expression terminal, it is an error.
- If the offset refers to a nonterminal, print the unparsed representation of the corresponding child of the current term.

**`color` and `colors` attributes** K allows you to take advantage of ANSI terminal codes for foreground color
in order to colorize output pretty-printed by the unparser. This is controlled via the `color` and `colors` attributes of productions. These attributes combine with the `format` attribute to control how a term is colorized.

The first thing to understand about how colorization works is that the `color` and `colors` attributes are used to construct a **list** of colors associated with each production, and the format attribute then uses that list to choose the color for each part of the production. For more information on how the format attribute chooses a color from the list, see above, but essentially, each terminal or `%c` in the format attribute advances the pointer in the list by one element, and terminals and `%r` reset the current color to the default foreground color of the terminal afterwards.

There are two ways you can construct a list of colors associated with a production:

- The `color` attribute creates the entire list all with the same color, as specified by the value of the attribute. When combined with the default format
  attribute, this will color all the terminals in that production that color, but
  more advanced techniques can be used as well.
- The `colors` attribute creates the list from a manual, comma-separated list
  of colors. The attribute is invalid if the length of the list is not equal to the number of terminals in the production plus the number of `%c` substrings in
  the `format` attribute.

## Debugging

The LLVM Backend has support for integration with GDB. You can run the debugger

on a particular program by passing the `--debugger` flag to krun, or by invoking the llvm backend interpreter directly. Below we provide a simple tutorial to explain some of the basic commands supported by the LLVM backend.

### The K Definition

Here is a sample K definition we will use to demonstrate debugging capabilities:

```
module TEST
  imports INT

  configuration <k> foo(5) </k>
  rule [test]: I:Int => I +Int 1 requires I <Int 10

  syntax Int ::= foo(Int) [function]
  rule foo(I) => 0 -Int I

endmodule
```

You should compile this definition with `--backend llvm -ccopt -g` and without
`-ccopt -O2` in order to use the debugger most effectively.

### Stepping

**Important:** When you first run `krun` with option `--debugger`, GDB will instruct you on how to modify ~/.gdbinit to enable printing abstract syntax of K terms in the debugger. If you do not perform this step, you can still use all the other features, but K terms will be printed as their raw address in memory. GDB will need the kompiled interpreter in its safe path in order to access the pretty printing python script within it. A good way to do this would be to pick a minimum top-level path that covers all of your kompiled semantics (ie. `set auto-load safe-path ~/k-semantics`)

You can break before every step of execution is taken by setting a breakpoint on the `step` function:

```
(gdb) break definition.kore:step
Breakpoint 1 at 0x25e340
(gdb) run
Breakpoint 1, 0x000000000025e340 in step (subject=`<generatedTop>{}`(`<k>{}`(`kseq{}`(`inj{I
(gdb) continue
Continuing.

Breakpoint 1, 0x000000000025e340 in step (subject=`<generatedTop>{}`(`<k>{}`(`kseq{}`(`inj{I
(gdb) continue 2
```

```
Will ignore next crossing of breakpoint 1.  Continuing.

Breakpoint 1, 0x000000000025e340 in step (subject=`<generatedTop>{}`(`<k>{}`(`kseq{}`(`inj{I
(gdb)
```

### Breaking on a specific rule

You can break when a rule is applied by giving the rule a rule label. If the module name is TEST and the rule label is test, you can break when the rule applies by setting a breakpoint on the `TEST.test.rhs` function:

```
(gdb) break TEST.test.rhs
Breakpoint 1 at 0x25e250: file /home/dwightguth/test/./test.k, line 4.
(gdb) run
Breakpoint 1, TEST.test.rhs (VarDotVar0=`<generatedCounter>{}`(#token("0", "Int")), VarDotVa
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb)
```

Note that the substitution associated with that rule is visible in the description of the frame.

You can also break when a side condition is applied using the `TEST.test.sc` function:

```
(gdb) break TEST.test.sc
Breakpoint 1 at 0x25e230: file /home/dwightguth/test/./test.k, line 4.
(gdb) run
Breakpoint 1, TEST.test.sc (VarI=#token("0", "Int")) at /home/dwightguth/test/./test.k:4
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb)
```

Note that every variable used in the side condition can have its value inspected when stopped at this breakpoint, but other variables are not visible.

You can also break on a rule by its location:

```
(gdb) break test.k:4
Breakpoint 1 at 0x25e230: test.k:4. (2 locations)
(gdb) run
Breakpoint 1, TEST.test.sc (VarI=#token("0", "Int")) at /home/dwightguth/test/./test.k:4
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb) continue
Continuing.

Breakpoint 1, TEST.test.rhs (VarDotVar0=`<generatedCounter>{}`(#token("0", "Int")), VarDotVa
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb) continue
Continuing.
```

```
Breakpoint 1, TEST.test.sc (VarI=#token("1", "Int")) at /home/dwightguth/test/./test.k:4
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb)
```

Note that this sets a breakpoint at two locations: one on the side condition and one on the right hand side. If the rule had no side condition, the first would not be set. You can also view the locations of the breakpoints and disable them individually:

```
(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   <MULTIPLE>
        breakpoint already hit 3 times
1.1                         y     0x000000000025e230 in TEST.test.sc at /home/dwightguth/tes
1.2                         y     0x000000000025e250 in TEST.test.rhs at /home/dwightguth/te
(gdb) disable 1.1
(gdb) continue
Continuing.

Breakpoint 1, TEST.test.rhs (VarDotVar0=`<generatedCounter>{}`(#token("0", "Int")), VarDotVa
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb) continue
Continuing.

Breakpoint 1, TEST.test.rhs (VarDotVar0=`<generatedCounter>{}`(#token("0", "Int")), VarDotVa
4         rule [test]: I:Int => I +Int 1 requires I <Int 10
(gdb)
```

Now only the breakpoint when the rule applies is enabled.

### Breaking on a function

You can also break when a particular function in your semantics is invoked:

```
(gdb) info functions foo
All functions matching regular expression "foo":

File /home/dwightguth/test/./test.k:
struct __mpz_struct *Lblfoo'LParUndsRParUnds'TEST'UndsUnds'Int(struct __mpz_struct *);
(gdb) break Lblfoo'LParUndsRParUnds'TEST'UndsUnds'Int
Breakpoint 1 at 0x25e640: file /home/dwightguth/test/./test.k, line 6.
(gdb) run
Breakpoint 1, Lblfoo'LParUndsRParUnds'TEST'UndsUnds'Int (_1=#token("1", "Int")) at /home/dwi
6         syntax Int ::= foo(Int) [function]
(gdb)
```

In this case, the variables have numbers instead of names because the names of arguments in functions in K come from rules, and we are stopped before any

specific rule has applied. For example, `_1` is the first argument to the function.

You can also set a breakpoint in this location by setting it on the line associated with its production:

```
(gdb) break test.k:6
Breakpoint 1 at 0x25e640: file /home/dwightguth/test/./test.k, line 6.
(gdb) run
Breakpoint 1, Lblfoo'LParUndsRParUnds'TEST'UndsUnds'Int (_1=#token("1", "Int")) at /home/dwi
6         syntax Int ::= foo(Int) [function]
```

These two syntaxes are equivalent; use whichever is easier for you.

You can also view the stack of function applications:

```
(gdb) bt
#0  Lblfoo'LParUndsRParUnds'TEST'UndsUnds'Int (_1=#token("1", "Int")) at /home/dwightguth/te
#1  0x000000000025e5f8 in apply_rule_111 (VarDotVar0=`<generatedCounter>{}`(#token("0", "Int
#2  0x0000000000268a52 in take_steps ()
#3  0x000000000026b7b4 in main ()
(gdb)
```

Here we see that `foo` was invoked while applying the rule on line 9 of test.k, and we also can see the substitution of that rule. If foo was evaluated while evaluating another function, we would also be able to see the arguments of that function as well, unless the function was tail recursive, in which case no stack frame would exist once the tail call was performed.

### Breaking on a set of rules or functions

Using `rbreak <regex>` you can set breakpoints on multiple functions.

- `rbreak Lbl` - sets a breakpoint on all non hooked `function`s
- `rbreak Lbl.*TEST` - sets a breakpoint on all `function`s from module `TEST`
- `rbreak hook_INT` - sets a breakpoint on all hooks from module `INT`

### Other debugger issues

- `<optimized out>` try kompiling without `-O1`, `-O2`, or `-O3`.
- `(gdb) break definition.kore:break -> No source file named definition.kore.`
  send `-ccopt -g` to kompile in order to generate debug info symbols.

## Attributes Reference

### Attribute Syntax Overview

In K, many different syntactic categories accept *attributes*, an optional trailing list of keywords or user-defined identifiers. Attribute lists have two different syntaxes, depending on where they occur. Each attribute also has a type which describes where it may occur.

The first syntax is a square-bracketed (`[]`) list of words. This syntax is available for following attribute types:

1. `module` attributes - may appear immediately after the `module` keyword
2. `sort` attributes - may appear immediately after a sort declaration
3. `production` attributes - may appear immediately after a BNF production alternative
4. `rule` attributes - may appear immediately after a rule
5. `context` attributes - may appear immediately after a context or context alias
6. `context alias` attributes - may appear immediately after a context alias
7. `claim` attributes - may appear immediately after a claim

The second syntax is the XML attribute syntax, i.e., a space delemited list of key-and-quoted-value pairs appearing inside the start tag of an XML element: `<element key1="value" key2="value2" ... > </element>`. This syntax is available for the following attribute types:

1. `cell` attributes - may appear inside of the cell start tag in configuration declarations

Note that, currently, *unknown* attributes are *ignored*. Essentially, this means that there is no such thing as an *invalid* attribute. When we talk about the *type* of an attribute, we mean a syntactic category to which an attribute can be attached where the attribute has some semantic effect.

### Attribute Index

We now provide an index of available attributes organized alphabetically with a
brief description of each. Note that the same attribute may appear in the index multiple times to indicate its effect in different contexts or with/without arguments. A legend describing how to interpret the index follows.

| Name | Type | Backend | Reference |
|---|---|---|---|
| `alias-rec` | rule | all | Macros and Aliases |
| `alias` | rule | all | Macros and Aliases |
| `all-path` | claim | haskell | `all-path` and `one-path` attributes to distinguish reachability cla |
| `anywhere` | rule | all | `anywhere` rules |
| `applyPriority(_)` | prod | all | Symbol priority and associativity |

| Name | Type | Backend | Reference |
|---|---|---|---|
| `avoid` | prod | all | Symbol priority and associativity |
| `binder` | prod | all | No reference yet. |
| `bracket` | prod | all | Parametric productions and `bracket` attributes |
| `color(_)` | prod | all | `color` and `colors` attributes |
| `colors(_)` | prod | all | `color` and `colors` attributes |
| `concrete` | mod | llvm | `symbolic` and `concrete` attribute |
| `concrete(_)` | rule | haskell | `concrete` and `symbolic` attributes (Haskell backend) |
| `concrete` | rule | haskell | `concrete` and `symbolic` attributes (Haskell backend) |
| `context(_)` | alias | all | Context aliases |
| `cool` | rule | all | `strict` and `seqstrict` attributes |
| `exit = ""` | cell | all | `exit` attribute |
| `format` | prod | all | `format` attribute |
| `freshGenerator` | prod | all | `freshGenerator` attribute |
| `functional` | rule | all | `function` and `functional` attributes |
| `function` | rule | all | `function` and `functional` attributes |
| `heat` | rule | all | `strict` and `seqstrict` attributes |
| `hook(_)` | prod | all | No reference yet |
| `hybrid(_)` | prod | all | `hybrid` attribute |
| `hybrid` | prod | all | `hybrid` attribute |
| `klabel(_)` | all | all | `klabel(_)` and `symbol` attributes |
| `latex(_)` | prod | all | No reference yet |
| `left` | prod | all | Symbol priority and associativity |
| `lemma` | rule | all | `smt-lemma`, `lemma`, and `trusted` attributes |
| `locations` | sort | all | Location Information |
| `macro-rec` | rule | all | Macros and Aliases |
| `macro` | rule | all | Macros and Aliases |
| `memo` | rule | haskell | The `memo` attribute |
| `multiplicity = "_"` | cell | all | Collection Cells: `multiplicity` and `type` attributes |
| `non-assoc` | prod | all | Symbol priority and associativity |
| `one-path` | claim | all | `all-path` and `one-path` attributes to distinguish reachability cla |
| `owise` | rule | all | `owise` and `priority` attributes |
| `prec(_)` | token | all | `prec` attribute |
| `prefer` | prod | all | Symbol priority and associativity |
| `priority(_)` | rule | all | `owise` and `priority` attributes |
| `private` | mod | all | `private` attribute |
| `private` | prod | all | `public` and `private` attribute |
| `public` | mod | all | No reference yet. |
| `public` | prod | all | `public` and `private` attribute |
| `result(_)` | ctxt | all | `result` attribute |
| `result(_)` | rule | all | `result` attribute |
| `right` | prod | all | Symbol priority and associativity |
| `seqstrict(_)` | prod | all | `strict` and `seqstrict` attributes |
| `seqstrict` | prod | all | `strict` and `seqstrict` attributes |
| `simplification` | rule | haskell | `simplification` attribute (Haskell backend) |

| Name | Type | Backend | Reference |
|---|---|---|---|
| `simplification(_)` | rule | haskell | `simplification` attribute (Haskell backend) |
| `smt-hook(_)` | prod | haskell | SMT Translation |
| `smt-lemma` | rule | all | `smt-lemma`, `lemma`, and `trusted` attributes |
| `smtlib(_)` | prod | haskell | SMT Translation |
| `strict` | prod | all | `strict` and `seqstrict` attributes |
| `strict(_)` | prod | all | `strict` and `seqstrict` attributes |
| `symbolic` | mod | haskell | `symbolic` and `concrete` attribute |
| `symbolic` | rule | haskell | `concrete` and `symbolic` attributes (Haskell backend) |
| `symbolic(_)` | rule | haskell | `concrete` and `symbolic` attributes (Haskell backend) |
| `symbol` | prod | all | `klabel(_)` and `symbol` attributes |
| `token` | prod | all | `token` attribute |
| `token` | sort | all | `token` attribute |
| `trusted` | claim | haskell | `smt-lemma`, `lemma`, and `trusted` attributes |
| `type = "_"` | cell | all | Collection Cells: `multiplicity` and `type` attributes |
| `unboundVariables(_)` | rule | all | The `unboundVariables` attribute |
| `unused` | prod | all | `unused` attribute |

### Internal Attribute Index

Some attributes should not generally appear in user code, except in some unusual or complex examples. Such attributes are typically generated by the compiler and used internally. We list these attributes below as a reference for interested readers:

| Name | Type | Backend | Reference |
|---|---|---|---|
| `assoc` | prod | all | `assoc`, `comm`, `idem` and `unit` attributes |
| `comm` | prod | all | `assoc`, `comm`, `idem` and `unit` attributes |
| `idem` | prod | all | `assoc`, `comm`, `idem` and `unit` attributes |
| `unit` | prod | all | `assoc`, `comm`, `idem` and `unit` attributes |
| `userList` | prod | all | Identifies the desugared form of `Lst ::= List{Elm,"delim"}` |
| `predicate` | prod | all | Specifies the sort of a predicate label |
| `element` | prod | all | Specifies the label of the elements in a list |
| `bracketLabel` | prod | all | Keep track of the label of a bracket production since it can't have a klabe |

### Index Legend

- `Name` - the attribute's name (optionally followed by an underscore `_` to indicate the attribute takes arguments)
- `Type` - the syntactic categories where this attribute is *not* ignored; the possible values are the types mentioned above or shorthands:
  1. `all` - short for any type except `cell`
  2. `mod` - short for `module`

3. `sort`
4. `prod` - short for `production`
5. `rule`
6. `ctxt` - short for `context` or `context alias`
7. `claim`
8. `cell`

- `Backend` - the backends that do *not* ignore this attribute; possible values:

  1. `all` - all backends
  2. `llvm` - the LLVM backend
  3. `haskell` - the Haskell backend

- `Effect` - the attribute's effect (when it applies)

## Pending Documentation

Backend features not yet given documentation:

- Parser of KORE terms and definitions
- Term representation of K terms
- Hooked sorts and symbols
- Substituting a substitution into the RHS of a rule
  - domain values
  - functions
  - variables
  - symbols
  - polymorphism
  - hooks
  - injection compaction
  - overload compaction
- Pattern Matching / Unification of subject and LHS of rule
  - domain values
  - symbols
  - side conditions
  - and/or patterns
  - list patterns
  - nonlinear variables
  - map/set patterns
    - deterministic
    - nondeterministic
  - modulo injections
  - modulo overloads
- Stepping
  - initialization
  - termination
- Print kore terms
- Equality/comparison of terms

- Owise rules
- Strategy #STUCK axiom
- User substitution
  - binders
  - kvar

To get a complete list of hooks supported by K, you can run:

```
grep -P -R "(?<=[^-])hook\([^)]*\)" k-distribution/include/kframework/builtin/ \
     --include "*.k" -ho | \
sed 's/hook(//' | sed 's/)//' | sort | uniq | grep -v org.kframework
```

All of these hooks will also eventually need documentation.

# K Builtins

The K Builtins (also referred to as the K Prelude or the K Standard Library) consists of several files which contain definitions that make working with K simpler. These files can be found under `include/kframework/builtin` in your K
installation directory, and can be imported with `requires "FILENAME"` (without
the path prefix).

- domains: Basic datatypes which are universally useful.
- kast: Representation of K internal data-structures (not to be included in normal definitions).
- prelude: Automatically included into every K definition.
- ffi: FFI interface for calling out to native C code from K.
- json: JSON datatype and parsers/unparsers for JSON strings.
- rat: Rational number representation.
- substitution: Hooked implementation of capture-aware sustitution for K definitions.
- unification: Hooked implementation of unification exposed directly to K definitions.

# Basic Builtin Types in K

A major piece of the K prelude consists of a series of modules that contain implementations of basic data types and language features in K. You do not need
to require this file yourself; it is required automatically in every K definition unless `--no-prelude` is passed to kompile. K may not work correctly if some of these modules do not exist or do not declare certain functions.

Note that some functions in the K prelude functions are not total, that is, they are not defined on all possible input values. When you invoke such a

function on an undefined input, the behavior is undefined. In particular, when this happens, interpreters generated by the K LLVM backend may crash.

```
require "kast.md"
```

## Default Modules

K declares certain modules that contain most of the builtins you usually want when defining a language in K. In particular, this includes integers, booleans, strings, identifiers, I/O, lists, maps, and sets. The `DOMAINS-SYNTAX` module is designed to be imported by the syntax module of the language and contains only the program-level syntax of identifiers, integers, booleans, and strings. The `DOMAINS` module contains the rest of the syntax, including builtin functions over those and the remaining types.

Note that not all modules are included in DOMAINS. A few less-common modules
are not, including `ARRAY`, `COLLECTIONS`, `FLOAT`, `STRING-BUFFER`, `BYTES`, `K-REFLECTION`, `MINT`, and `STRATEGY`.

```
module DOMAINS-SYNTAX
  imports SORT-K
  imports ID-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  imports BOOL-SYNTAX
  imports STRING-SYNTAX
endmodule

module DOMAINS
  imports DOMAINS-SYNTAX
  imports INT
  imports BOOL
  imports STRING
  imports BASIC-K
  imports LIST
  imports K-IO
  imports MAP
  imports SET
  imports ID
  imports K-EQUAL
endmodule
```

## Arrays

Provided here is an implementation for fixed-sized, contiguous maps from `Int` to `KItem`. In some previous versions of K, the `Array` type was a builtin type backed by mutable arrays of objects. However, in modern K, the `Array` type is

implemented by means of the `List` type; users should not access this interface directly and should instead make only of the functions listed below. Users of this module should import only the `ARRAY` module.

```
module ARRAY-SYNTAX
  imports private LIST

  syntax Array [hook(ARRAY.Array), unit(arrayCtor), element(_[_<-_])]
```

### Array lookup

You can look up an element in an `Array` by its index in O(log(N)) time. Note that the base of the logarithm is a relatively high number and thus the time is effectively constant.

```
  syntax KItem ::= Array "[" Int "]" [function, hook(ARRAY.lookup)]
```

### Array update

You can create a new `Array` with a new value for a key in O(log(N)) time, or effectively constant.

```
  syntax Array ::= Array "[" key: Int "<-" value: KItem "]" [function, hook(ARRAY.update), k
```

### Array reset

You can create a new `Array` where a particular key is reset to its default value in O(log(N)) time, or effectively constant.

```
  syntax Array ::= Array "[" Int "<-" "undef" "]" [function, hook(ARRAY.remove)]
```

### Multiple array update

You can create a new `Array` from a `List` `L` of size `N` where the `N` elements starting at `index` are replaced with the contents of `L`, in O(N*log(K)) time (where K is the size of the array), or effectively linear. Having `index + N > K` yields an exception.

```
  syntax Array ::= updateArray(Array, index: Int, List) [function, hook(ARRAY.updateAll)]
```

### Array fill

You can create a new `Array` where the `length` elements starting at `index` are replaced with `value`, in O(length*log(N)) time, or effectively linear.

```
  syntax Array ::= fillArray(Array, index: Int, length: Int, value: KItem) [function, hook(A
```

### Array range check

You can test whether an integer is within the bounds of an array in O(1) time.

```
  syntax Bool ::= Int "in_keys" "(" Array ")" [function, functional, hook(ARRAY.in_keys)]

endmodule

module ARRAY-IN-K [private]
  imports public ARRAY-SYNTAX
  imports private LIST
  imports private K-EQUAL
  imports private INT
  imports private BOOL
```

### Array creation

You can create an array with `length` elements where each element is initialized to `value` in O(1) time. Note that the array is stored in a manner where only the highest element that is actually modified is given a value in its internal representation, which means that subsequent array operations may incur a one-time O(N) resizing cost, possibly amortized across multiple operations.

```
  syntax Array ::= makeArray(length: Int, value: KItem) [function, hook(ARRAY.make), public]
```

### Implementation of Arrays

The remainder of this section consists of an implementation in K of the operations listed above. Users of the `ARRAY` module should not make use of any of the syntax defined in any of these modules.

```
  syntax Array ::= arr(List, Int, KItem)

  rule makeArray(I::Int, D::KItem) => arr(.List, I, D)

  rule arr(L::List, _, _       ) [ IDX::Int ] => L[IDX] requires 0 <=Int IDX andBool IDX  <I
  rule arr(_      , _, D::KItem) [ _        ] => D      [owise]

  syntax List ::= ensureOffsetList(List, Int, KItem) [function]
  rule ensureOffsetList(L::List, IDX::Int, D::KItem) => L makeList(IDX +Int 1 -Int size(L),
  rule ensureOffsetList(L::List, IDX::Int, _::KItem) => L

  rule arr(L::List, I::Int, D::KItem) [ IDX::Int <- VAL::KItem ] => arr(ensureOffsetList(L,

  rule arr(L::List, I::Int, D::KItem) [ IDX::Int <- undef ] => arr(L, I, D) [ IDX <- D ]

  rule updateArray(arr(L::List, I::Int, D::KItem), IDX::Int, L2::List) => arr(updateList(ens
```

```
  rule fillArray(arr(L::List, I::Int, D::KItem), IDX::Int, LEN::Int, VAL::KItem) => arr(fill

  rule IDX::Int in_keys(arr(_, I::Int, _)) => IDX >=Int 0 andBool IDX <Int I
endmodule

module ARRAY-SYMBOLIC [symbolic]
  imports ARRAY-IN-K
endmodule

module ARRAY-KORE [kore]
  imports ARRAY-IN-K
endmodule

module ARRAY
  imports ARRAY-SYMBOLIC
  imports ARRAY-KORE
endmodule
```

## Maps

Provided here is the syntax of an implementation of immutable, associative, commutative maps from `KItem` to `KItem`. This type is hooked to an implementation of maps provided by the backend. For more information on matching on maps and allowable patterns for doing so, refer to K's user documentation.

```
module MAP
  imports private BOOL-SYNTAX
  imports private INT-SYNTAX
  imports private LIST
  imports private SET

  syntax Map [hook(MAP.Map)]
```

### Map concatenation

The `Map` sort represents a generalized associative array. Each key can be paired with an arbitrary value, and can be used to reference its associated value. Multiple bindings for the same key are not allowed.

You can construct a new Map consisting of key/value pairs of two Maps. The result is `#False` if the maps have keys in common (in particular, this will yield an exception during concrete execution). This operation is O(N*log(M)) *where N is the size of the smaller map, when it appears on the right hand side. When it appears on the left hand side and all variables are bound, it is* O(*N*log(M)) where M is the size of the map it is matching and N is the number of elements being matched. When it appears on the left hand side containing

variables not bound elsewhere in the term, it is O(N^K) where N is the size of the map it is matching and K is the number of unbound keys being matched. In
other words, one unbound variable is linear, two is quadratic, three is cubic, etc.

```
syntax Map ::= Map Map                          [left, function, hook(MAP.concat), klabel(_M
```

### Map unit

The map with zero elements is represented by `.Map`.

```
syntax Map ::= ".Map"                           [function, functional, hook(MAP.unit), klabe
```

### Map elements

An element of a `Map` is constructed via the `|->` operator. The key is on the left and the value is on the right.

```
syntax Map ::= KItem "|->" KItem                      [function, functional, hook(MAP.elem

syntax priorities _|->_ > _Map_ .Map
syntax non-assoc _|->_
```

### Map lookup

You can look up the value associated with the key of a map in O(log(N)) time. Note that the base of the logarithm is a relatively high number and thus the time is effectively constant. The value is `#False` if the key is not in the map (in particular, this will yield an exception during concrete execution).

```
syntax KItem ::= Map "[" KItem "]"                    [function, hook(MAP.lookup), klabel(
```

### Map lookup with default

You can also look up the value associated with the key of a map using a total function that assigns a specific default value if the key is not present in the map. This operation is also O(log(N)), or effectively constant.

```
syntax KItem ::= Map "[" KItem "]" "orDefault" KItem      [function, functional, hook(MAP.
```

### Map update

You can insert a key/value pair into a map in O(log(N)) time, or effectively constant.

```
syntax Map ::= Map "[" key: KItem "<-" value: KItem "]"           [function, functional, k
```

### Map delete

You can remove a key/value pair from a map via its key in O(log(N)) time, or effectively constant.

```
syntax Map ::= Map "[" KItem "<-" "undef" "]"     [function, functional, hook(MAP.remove)
```

### Map difference

You can remove the key/value pairs in a map that are present in another map in
O(N*log(M)) time (where M is the size of the first map and N is the size of the second), or effectively linear. Note that only keys whose value is the same in both maps are removed. To remove all the keys in one map from another map,
you can say `removeAll(M1, keys(M2))`.

```
syntax Map ::= Map "-Map" Map                  [function, functional, hook(MAP.difference)
```

### Multiple map update

You can update a map by adding all the key/value pairs in the second map in O(N*log(M)) time (where M is the size of the first map and N is the size of the second map), or effectively linear. If any keys are present in both maps, the value from the second map overwrites the value in the first. This function is total, which is distinct from map concatenation, a partial function only defined on maps with disjoint keys.

```
syntax Map ::= updateMap(Map, Map)             [function, functional, hook(MAP.updateAll)]
```

### Multiple map removal

You can remove a `Set` of keys from a map in O(N*log(M)) time (where M is the size of the `Map` and `N` is the size of the `Set`), or effectively linear.

```
syntax Map ::= removeAll(Map, Set)             [function, functional, hook(MAP.removeAll)]
```

### Map keys (as `Set`)

You can get a `Set` of all the keys in a Map in O(N) time.

```
syntax Set ::= keys(Map)                       [function, functional, hook(MAP.keys)]
```

### Map keys (as `List`)

You can get a `List` of all the keys in a Map in O(N) time.

```
syntax List ::= "keys_list" "(" Map ")"        [function, hook(MAP.keys_list)]
```

### Map key membership

You can check whether a key is present in a map in O(1) time.

```
  syntax Bool ::= KItem "in_keys" "(" Map ")"       [function, functional, hook(MAP.in_keys)
```

### Map values (as `List`)

You can get a `List` of all the values in a map in O(N) time.

```
  syntax List ::= values(Map)                       [function, hook(MAP.values)]
```

### Map size

You can get the number of key/value pairs in a map in O(1) time.

```
  syntax Int ::= size(Map)                          [function, functional, hook(MAP.size), klabe
```

### Map inclusion

You can determine whether a `Map` is a strict subset of another `Map` in O(N) time (where N is the size of the first map). Only keys that are bound to the same value are considered equal.

```
  syntax Bool ::= Map "<=Map" Map                   [function, functional, hook(MAP.inclusion)]
```

### Map choice

You can get an arbitrarily chosen key of a `Map` in O(1) time. The same key will always be returned for the same map, but no guarantee is given that two different maps will return the same element, even if they are similar.

```
  syntax KItem ::= choice(Map)                      [function, hook(MAP.choice), klabel(Map
```

### Implementation of Maps

The remainder of this section contains lemmas used by the Java and Haskell backend to simplify expressions of sort `Map`. They do not affect the semantics of maps, merely describing additional rules that the backend can use to simplify terms.

```
endmodule

module MAP-KORE-SYMBOLIC [kore,symbolic]
  imports MAP
  imports private K-EQUAL
  imports private BOOL

  rule #Ceil(@M:Map [@K:KItem]) => {(@K in_keys(@M)) #Equals true} #And #Ceil(@M) #And #Ceil

  // Symbolic update
```

```
  // Adding the definedness condition `notBool (K in_keys(M))` in the ensures clause of the
  // because K also appears in the rhs, preserving the case when it's #Bottom.
  rule (K |-> _ M:Map) [ K <- V ] => (K |-> V M) [simplification]
  rule M:Map [ K <- V ] => (K |-> V M) requires notBool (K in_keys(M)) [simplification]
  rule M:Map [ K <- _ ] [ K <- V ] => M [ K <- V ] [simplification]
  // Adding the definedness condition `notBool (K1 in_keys(M))` in the ensures clause of the
  // because K1 also appears in the rhs, preserving the case when it's #Bottom.
  rule (K1 |-> V1 M:Map) [ K2 <- V2 ] => (K1 |-> V1 (M [ K2 <- V2 ])) requires K1 =/=K K2 [s

  // Symbolic remove
  rule (K |-> _ M:Map) [ K <- undef ] => M ensures notBool (K in_keys(M)) [simplification]
  rule M:Map [ K <- undef ] => M requires notBool (K in_keys(M)) [simplification]
  // Adding the definedness condition `notBool (K1 in_keys(M))` in the ensures clause of the
  // because K1 also appears in the rhs, preserving the case when it's #Bottom.
  rule (K1 |-> V1 M:Map) [ K2 <- undef ] => (K1 |-> V1 (M [ K2 <- undef ])) requires K1 =/=K

  // Symbolic lookup
  rule (K  |->  V M:Map) [ K ]  => V ensures notBool (K in_keys(M)) [simplification]
  rule (K1 |-> _V M:Map) [ K2 ] => M [K2] requires K1 =/=K K2 ensures notBool (K1 in_keys(M)
  rule (_MAP:Map [ K  <-  V1 ]) [ K ]  => V1 [simplification]
  rule ( MAP:Map [ K1 <- _V1 ]) [ K2 ] => MAP [ K2 ] requires K1 =/=K K2 [simplification]

  // Symbolic in_keys
  rule K in_keys(_M [ K <- undef ]) => false [simplification]
  rule K in_keys(_M [ K <- _ ]) => true [simplification]
  rule K1 in_keys(M [ K2 <- _ ]) => true requires K1 ==K K2 orBool K1 in_keys(M) [simplifica
  rule K1 in_keys(M [ K2 <- _ ]) => K1 in_keys(M) requires K1 =/=K K2 [simplification]

  rule {false #Equals @Key in_keys(.Map)} => #Ceil(@Key) [simplification]
  rule {@Key in_keys(.Map) #Equals false} => #Ceil(@Key) [simplification]
  rule {false #Equals @Key in_keys(Key' |-> Val @M)} => #Ceil(@Key) #And #Ceil(Key' |-> Val
  rule {@Key in_keys(Key' |-> Val @M) #Equals false} => #Ceil(@Key) #And #Ceil(Key' |-> Val

/*
// The rule below is automatically generated by the frontend for every sort
// hooked to MAP.Map. It is left here to serve as documentation.

  rule #Ceil(@M:Map (@K:KItem |-> @V:KItem)) => {(@K in_keys(@M)) #Equals false} #And #Ceil(
    [simplification]
*/
endmodule

module MAP-JAVA-SYMBOLIC [kast, symbolic]
  imports MAP
  imports private K-EQUAL
```

```
  imports private BOOL
  rule .Map [ K1 <- V1 ] => K1 |-> V1 [simplification]

  rule ((K1 |->  V1) _MAP) [ K2 ] => V1         requires K1  ==K K2 [simplification]
  rule ((K1 |-> _V1)  MAP) [ K2 ] => MAP [ K2 ] requires K1 =/=K K2 [simplification]

  rule (_MAP:Map [ K1 <-  V1 ]) [ K2 ] => V1         requires K1  ==K K2 [simplification]
  rule ( MAP:Map [ K1 <- _V1 ]) [ K2 ] => MAP [ K2 ] requires K1 =/=K K2 [simplification]

  rule ((K1 |-> _V1) MAP) [ K2 <- V2 ] => (K1 |-> V2) MAP                requires K1  ==K K2
  rule ((K1 |->  V1) MAP) [ K2 <- V2 ] => (K1 |-> V1) (MAP [ K2 <- V2 ]) requires K1 =/=K K2

  rule (MAP:Map [ K1 <- _V1 ]) [ K2 <- V2 ] => MAP              [ K1 <- V2 ] requires K1  ==

  // potential infinite loop
  // rule (MAP:Map [ K1 <- V1 ]) [ K2 <- V2 ] => MAP [ K2 <- V2 ] [ K1 <- V1 ] requires K1 =

  rule K1 in_keys(M K2 |-> _) => true          requires K1  ==K K2 orBool K1 in_keys(M) [sim
  rule K1 in_keys(M K2 |-> _) => K1 in_keys(M) requires K1 =/=K K2 [simplification]

  rule K1 in_keys(M [ K2 <- _ ]) => true          requires K1  ==K K2 orBool K1 in_keys(M)
  rule K1 in_keys(M [ K2 <- _ ]) => K1 in_keys(M) requires K1 =/=K K2 [simplification]
endmodule

module MAP-SYMBOLIC
  imports MAP-JAVA-SYMBOLIC
  imports MAP-KORE-SYMBOLIC
endmodule
```

## Sets

Provided here is the syntax of an implementation of immutable, associative, commutative sets of `KItem`. This type is hooked to an implementation of sets provided by the backend. For more information on matching on sets and allowable
patterns for doing so, refer to K's
user documentation.

```
module SET
  imports private INT-SYNTAX
  imports private BASIC-K

  syntax Set [hook(SET.Set)]
```

### Set concatenation

The `Set` sort represents a mathematical set (A collection of unique items). The sets are nilpotent, i.e., the concatenation of two sets containing elements in common is `#False` (note however, this may be silently allowed during concrete execution). If you intend to add an element to a set that might already be present in the set, use the `|Set` operator instead.

The concatenation operator is O(N*log(M)) where N is the size of the smaller set, when it appears on the right hand side. When it appears on the left hand side and all variables are bound, it is O(N log(M)) where M is the size of the set it is matching and N is the number of elements being matched. When it appears on the left hand side containing variables not bound elsewhere in the term, it is O(N^K) where N is the size of the set it is matching and K is the number of unbound keys being mached. In other words, one unbound variable is
linear, two is quadratic, three is cubic, etc.

```
syntax Set ::= Set Set                  [left, function, functional, hook(SET.concat), kla
```

### Set unit

The set with zero elements is represented by `.Set`.

```
syntax Set ::= ".Set"                   [function, functional, hook(SET.unit), klabel(.Set
```

### Set elements

An element of a `Set` is constructed via the `SetItem` operator.

```
syntax Set ::= SetItem(KItem)               [function, functional, hook(SET.element), klab
```

### Set union

You can compute the union of two sets in O(N*log(M)) time (Where N is the size
of the smaller set). Note that the base of the logarithm is a relatively high number and thus the time is effectively linear. The union consists of all the elements present in either set.

```
syntax Set ::= Set "|Set" Set            [left, function, functional, hook(SET.union)]
rule S1:Set |Set S2:Set => S1 (S2 -Set S1)
```

### Set intersection

You can compute the intersection of two sets in O(N*log(M)) time (where N is the size of the smaller set), or effectively linear. The intersection consists of all the elements present in both sets.

```
syntax Set ::= intersectSet(Set, Set)   [function, functional, hook(SET.intersection)]
```

#### Set complement

You can compute the relative complement of two sets in O(N*log(M)) time (where
N is the size of the second set), or effectively linear. This is the set of elements in the first set that are not present in the second set.

```
  syntax Set ::= Set "-Set" Set           [function, functional, hook(SET.difference), latex
```

#### Set membership

You can compute whether an element is a member of a set in O(1) time.

```
  syntax Bool ::= KItem "in" Set              [function, functional, hook(SET.in), klabel(Se
```

#### Set inclusion

You can determine whether a `Set` is a strict subset of another `Set` in O(N) time (where N is the size of the first set).

```
  syntax Bool ::= Set "<=Set" Set         [function, functional, hook(SET.inclusion)]
```

#### Set size

You can get the number of elements (the cardinality) of a set in O(1) time.

```
  syntax Int ::= size(Set)                [function, functional, hook(SET.size)]
```

#### Set choice

You can get an arbitrarily chosen element of a `Set` in O(1) time. The same element will always be returned for the same set, but no guarantee is given that two different sets will return the same element, even if they are similar.

```
  syntax KItem ::= choice(Set)                [function, hook(SET.choice), klabel(Set:choice
```

```
endmodule
```

## Lists

Provided here is the syntax of an implementation of immutable, associative lists of `KItem`. This type is hooked to an implementation of lists provided by the backend. For more information on matching on lists and allowable patterns for doing so, refer to K's
user documentation.

```
module LIST
  imports private INT-SYNTAX
  imports private BASIC-K

  syntax List [hook(LIST.List)]
```

### List concatenation

The `List` sort is an ordered collection that may contain duplicate elements. They are backed by relaxed radix balanced trees, which means that they support efficiently adding elements to both sides of the list, concatenating two lists, indexing, and updating elements.

The concatenation operator is O(log(N)) (where N is the size of the longer list) when it appears on the right hand side. When it appears on the left hand side, it is O(N), where N is the number of elements matched on the front and back of the list.

```
syntax List ::= List List               [left, function, functional, hook(LIST.concat), kl
```

### List unit

The list with zero elements is represented by `.List`.

```
syntax List ::= ".List"                 [function, functional, hook(LIST.unit), klabel(.Li
```

### List elements

An element of a `List` is constucted via the `ListItem` operator.

```
syntax List ::= ListItem(KItem)             [function, functional, hook(LIST.element), kla
```

### List indexing

You can get an element of a list by its integer offset in O(log(N)) time, or effectively constant. Positive indices are 0-indexed from the beginning of the list, and negative indices are -1-indexed from the end of the list. In other words, 0 is the first element and -1 is the last element.

```
syntax KItem ::= List "[" Int "]"           [function, hook(LIST.get), klabel(List:get), s
```

### List update

You can create a new `List` with a new value at a particular index in O(log(N)) time, or effectively constant.

```
syntax List ::= List "[" index: Int "<-" value: KItem "]" [function, hook(LIST.update), kl
```

### List of identical elements

You can create a list with `length` elements, each containing `value`, in O(N) time.

```
syntax List ::= makeList(length: Int, value: KItem) [function, hook(LIST.make)]
```

### Multiple list update

You can create a new `List` which is equal to `dest` except the `N` elements starting at `index` are replaced with the contents of `src` in O(N*log(K)) time (where `K` is the size of `dest`and `N` is the size of `src`), or effectively linear. Having `index + N > K` yields an exception.

```
  syntax List ::= updateList(dest: List, index: Int, src: List) [function, hook(LIST.updateA
```

### List fill

You can create a new `List` where the `length` elements starting at `index` are replaced with `value`, in O(length*log(N)) time, or effectively linear.

```
  syntax List ::= fillList(List, index: Int, length: Int, value: KItem) [function, hook(LIST
```

### List slicing

You can compute a new `List` by removing `fromFront` elements from the front of the list and `fromBack` elements from the back of the list in O((fromFront+fromBack)*log(N)) time, or effectively linear.

```
  syntax List ::= range(List, fromFront: Int, fromBack: Int)   [function, hook(LIST.range),
```

### List membership

You can compute whether an element is in a list in O(N) time. For repeated comparisons, it is much better to first convert to a set using `List2Set`.

```
  syntax Bool ::= KItem "in" List             [function, functional, hook(LIST.in), klabel(_
```

### List size

You can get the number of elements of a list in O(1) time.

```
  syntax Int ::= size(List)               [function, functional, hook(LIST.size), klabel (si
```

```
endmodule
```

## Collection Conversions

It is possible to convert from a `List` to a `Set` or from a `Set` to a list. Converting from a `List` to a `Set` and back will not provide the same list; duplicates will have been removed and the list may be reordered. Converting from a `Set` to a `List` and back will generate the same set.

Note that because sets are unordered and lists are ordered, converting from a Set to a List will generate some arbitrary ordering of elements, which may be different from the natural ordering you might assume, or may not. Two equal sets are guaranteed to generate the same ordering, but no guarantee is otherwise provided about what the ordering will be. In particular, adding an

element to a set may completely reorder the elements already in the set, when it is converted to a list.

```
module COLLECTIONS
  imports LIST
  imports SET
  imports MAP

  syntax List ::= Set2List(Set) [function, functional, hook(SET.set2list)]
  syntax Set ::= List2Set(List) [function, functional, hook(SET.list2set)]

endmodule
```

## Booleans

Provided here is the syntax of an implementation of boolean algebra in K. This type is hooked to an implementation of booleans provided by the backend. Note that this algebra is different from the builtin truth in matching logic. You can, however, convert from the truth of the `Bool` sort to the truth in matching logic via the expression `{B #Equals true}`.

The boolean values are `true` and `false`.

```
module SORT-BOOL
  syntax Bool [hook(BOOL.Bool)]
endmodule

module BOOL-SYNTAX
  imports SORT-BOOL
  syntax Bool ::= "true"  [token]
  syntax Bool ::= "false" [token]
endmodule

module BOOL-COMMON
  imports private BASIC-K
  imports BOOL-SYNTAX
```

### Basic boolean arithmetic

You can:

- Negate a boolean value.
- AND two boolean values.
- XOR two boolean values.
- OR two boolean values.
- IMPLIES two boolean values (i.e., `P impliesBool Q` is the same as `notBool P orBool Q`)
- Check equality of two boolean values.

- Check inequality of two boolean values.

Note that only `andThenBool` and `orElseBool` are short-circuiting. `andBool` and `orBool` may be short-circuited in concrete backends, but in symbolic ackends, both arguments will be evaluated.

```
syntax Bool ::= "notBool" Bool          [function, functional, klabel(notBool_), symbol, s
              > Bool "andBool" Bool     [function, functional, klabel(_andBool_), symbol,
              | Bool "andThenBool" Bool [function, functional, klabel(_andThenBool_), symb
              | Bool "xorBool" Bool     [function, functional, klabel(_xorBool_), symbol,
              | Bool "orBool" Bool      [function, functional, klabel(_orBool_), symbol, l
              | Bool "orElseBool" Bool  [function, functional, klabel(_orElseBool_), symbo
              | Bool "impliesBool" Bool [function, functional, klabel(_impliesBool_), symb
              > left:
                Bool "==Bool" Bool      [function, functional, klabel(_==Bool_), symbol, l
              | Bool "=/=Bool" Bool     [function, functional, klabel(_=/=Bool_), symbol,
```

### Implementation of Booleans

The remainder of this section consists of an implementation in K of the operations listed above.

```
rule notBool true => false
rule notBool false => true

rule true andBool B:Bool => B:Bool
rule B:Bool andBool true => B:Bool
rule false andBool _:Bool => false
rule _:Bool andBool false => false

rule true andThenBool K::Bool => K
rule K::Bool andThenBool true => K
rule false andThenBool _ => false
rule _ andThenBool false => false

rule false xorBool B:Bool => B:Bool
rule B:Bool xorBool false => B:Bool
rule B:Bool xorBool B:Bool => false

rule true orBool _:Bool => true
rule _:Bool orBool true => true
rule false orBool B:Bool => B
rule B:Bool orBool false => B

rule true orElseBool _ => true
rule _ orElseBool true => true
rule false orElseBool K::Bool => K
```

```
  rule K::Bool orElseBool false => K

  rule true impliesBool B:Bool => B
  rule false impliesBool _:Bool => true
  rule _:Bool impliesBool true => true
  rule B:Bool impliesBool false => notBool B

  rule B1:Bool =/=Bool B2:Bool => notBool (B1 ==Bool B2)
endmodule

module BOOL-KORE [kore, symbolic]
  imports BOOL-COMMON

  rule {true #Equals notBool @B} => {false #Equals @B} [simplification]
  rule {notBool @B #Equals true} => {@B #Equals false} [simplification]
  rule {false #Equals notBool @B} => {true #Equals @B} [simplification]
  rule {notBool @B #Equals false} => {@B #Equals true} [simplification]

  rule {true #Equals @B1 andBool @B2} => {true #Equals @B1} #And {true #Equals @B2} [simplif
  rule {@B1 andBool @B2 #Equals true} => {@B1 #Equals true} #And {@B2 #Equals true} [simplif
  rule {false #Equals @B1 orBool @B2} => {false #Equals @B1} #And {false #Equals @B2} [simpl
  rule {@B1 orBool @B2 #Equals false} => {@B1 #Equals false} #And {@B2 #Equals false} [simpl
endmodule

module BOOL
  imports BOOL-COMMON
  imports BOOL-KORE
endmodule
```

## Integers

Provided here is the syntax of an implementation of arbitrary-precision integer arithmetic in K. This type is hooked to an implementation of integers provided by the backend. For a fixed-width integer type, see the `MINT` module below.

The `UNSIGNED-INT-SYNTAX` module provides a syntax of whole numbers in K. This is useful because often programming languages implement the sign of an integer as a unary operator rather than part of the lexical syntax of integers. However, you can also directly reference integers with a sign using the `INT-SYNTAX` module.

```
module UNSIGNED-INT-SYNTAX
  syntax Int [hook(INT.Int)]
  syntax Int ::= r"[0-9]+" [prefer, token, prec(2)]
endmodule
```

```
module INT-SYNTAX
  imports UNSIGNED-INT-SYNTAX
  syntax Int ::= r"[\\+-]?[0-9]+" [prefer, token, prec(2)]
endmodule

module INT-COMMON
  imports INT-SYNTAX
  imports private BOOL
```

### Integer arithmetic

You can:

- Compute the bitwise complement `~Int` of an integer value in twos-complement.
- Compute the exponentiation `^Int` of two integers.
- Compute the exponentiation of two integers modulo another integer (`^%Int`).
  `A ^%Int B C` is equal in value to `(A ^Int B) %Int C`, but has a better asymptotic complexity.
- Compute the product `*Int` of two integers.
- Compute the quotient `/Int` or modulus `%Int` of two integers using t-division, which rounds towards zero. Division by zero is `#False`.
- Compute the quotient `divInt` or modulus `modInt` of two integers using Euclidean division, in which the remainder is always non-negative. Division
  by zero is `#False`.
- Compute the sum `+Int` or difference `-Int` of two integers.
- Compute the arithmetic right shift `>>Int` of two integers. Shifting by a negative quantity is `#False`.
- Compute the left shift of two integers. Shifting by a negative quantity is `#False`.
- Compute the bitwise and of two integers in twos-complement.
- Compute the bitwise xor of two integers in twos-complement.
- Compute the bitwise inclusive-or of two integers in twos-complement.

```
  syntax Int ::= "~Int" Int                    [function, klabel(~Int_), symbol, functional
               > left:
                 Int "^Int" Int                [function, klabel(_^Int_), symbol, left, smt
               | Int "^%Int" Int Int           [function, klabel(_^%Int__), symbol, left, s
               > left:
                 Int "*Int" Int                [function, functional, klabel(_*Int_), symbo
               /* FIXME: translate /Int and %Int into smtlib */
               /* /Int and %Int implement t-division, which rounds towards 0 */
               | Int "/Int" Int                [function, klabel(_/Int_), symbol, left, smt
               | Int "%Int" Int                [function, klabel(_%Int_), symbol, left, smt
               /* divInt and modInt implement e-division according to the Euclidean divisior
```

```
  | Int "divInt" Int              [function, klabel(_divInt_), symbol, left, s
  | Int "modInt" Int              [function, klabel(_modInt_), symbol, left, s
  > left:
    Int "+Int" Int                [function, functional, klabel(_+Int_), symbo
  | Int "-Int" Int                [function, functional, klabel(_-Int_), symbo
  > left:
    Int ">>Int" Int               [function, klabel(_>>Int_), symbol, left, la
  | Int "<<Int" Int               [function, klabel(_<<Int_), symbol, left, la
  > left:
    Int "&Int" Int                [function, functional, klabel(_&Int_), symbo
  > left:
    Int "xorInt" Int              [function, functional, klabel(_xorInt_), sym
  > left:
    Int "|Int" Int                [function, functional, klabel(_|Int_), symbo
```

### Integer minimum and maximum

You can compute the minimum and maximum `minInt` and `maxInt` of two integers.

```
syntax Int ::= "minInt" "(" Int "," Int ")"   [function, functional, smt-hook((ite (< #1 #
             | "maxInt" "(" Int "," Int ")"   [function, functional, smt-hook((ite (< #1 #
```

### Absolute value

You can compute the absolute value `absInt` of an integer.

```
syntax Int ::= absInt ( Int )                 [function, functional, smt-hook((ite (< #1 (
```

### Log base 2

You can compute the log base 2, rounded towards zero, of an integer. The log base 2 of an integer is equal to the index of the highest bit set in the representation of a positive integer. Log base 2 of zero or a negative number is `#False`.

```
syntax Int ::= log2Int ( Int )                [function, hook(INT.log2)]
```

### Bit slicing

You can compute the value of a range of bits in the twos-complement representation of an integer, as interpeted either unsigned or signed, of an integer. `index` is offset from 0 and `length` is the number of bits, starting with `index`, that should be read. The number is assumed to be represented in little endian notation with each byte going from least significant to most significant. In other words, 0 is the least-significant bit, and each successive bit is more significant than the last.

```
  syntax Int ::= bitRangeInt           ( Int, index: Int, length: Int ) [function, hook(INT.
               | signExtendBitRangeInt ( Int, index: Int, length: Int ) [function, hook(INT.
```

### Integer comparisons

You can compute whether two integers are less than or equal to, less than, greater than or equal to, greater than, equal, or unequal to another integer.

```
  syntax Bool ::= Int "<=Int" Int         [function, functional, klabel(_<=Int_), symbol, le
                | Int "<Int" Int          [function, functional, klabel(_<Int_), symbol, lef
                | Int ">=Int" Int         [function, functional, klabel(_>=Int_), symbol, le
                | Int ">Int" Int          [function, functional, klabel(_>Int_), symbol, lef
                | Int "==Int" Int         [function, functional, klabel(_==Int_), symbol, le
                | Int "=/=Int" Int        [function, functional, klabel(_=/=Int_), symbol, I
```

### Divides

You can compute whether one integer evenly divides another. This is the case when the second integer modulo the first integer is equal to zero.

```
  syntax Bool ::= Int "dividesInt" Int    [function]
```

### Random integers

You can, on concrete backends, compute a pseudorandom integer, or seed the pseudorandom number generator. These operations are represented as uninterpreted functions on symbolic backends.

```
  syntax Int ::= randInt(Int) [function, hook(INT.rand), impure]
  syntax K ::= srandInt(Int) [function, hook(INT.srand), impure]
```

### Implementation of Integers

The remainder of this section consists of an implementation in K of some of the operators above, as well as lemmas used by the Java and Haskell backend to simplify expressions of sort `Int`. They do not affect the semantics of integers, merely describing additional rules that the backend can use to simplify terms.

```
endmodule

module INT-SYMBOLIC [symbolic]
  imports INT-COMMON
  imports INT-SYMBOLIC-KORE
  imports private BOOL

  // Arithmetic Normalization
  rule I +Int 0 => I [simplification]
  rule I -Int 0 => I [simplification]
```

```
  rule X modInt N => X requires 0 <=Int X andBool X <Int N [simplification]
  rule X   %Int N => X requires 0 <=Int X andBool X <Int N [simplification]

  // Bit-shifts
  rule X <<Int 0 => X [simplification]
  rule 0 <<Int _ => 0 [simplification]
  rule X >>Int 0 => X [simplification]
  rule 0 >>Int _ => 0 [simplification]
endmodule

module INT-SYMBOLIC-KORE [symbolic, kore]
  imports INT-COMMON
  imports ML-SYNTAX
  imports private BOOL

  // Definability Conditions
  rule #Ceil(@I1:Int /Int   @I2:Int) => {(@I2 =/=Int 0) #Equals true} #And #Ceil(@I1) #And #
  rule #Ceil(@I1:Int %Int   @I2:Int) => {(@I2 =/=Int 0) #Equals true} #And #Ceil(@I1) #And #
  rule #Ceil(@I1:Int modInt @I2:Int) => {(@I2 =/=Int 0) #Equals true} #And #Ceil(@I1) #And #
  rule #Ceil(@I1:Int >>Int  @I2:Int) => {(@I2 >=Int 0)  #Equals true} #And #Ceil(@I1) #And #
  rule #Ceil(@I1:Int <<Int  @I2:Int) => {(@I2 >=Int 0)  #Equals true} #And #Ceil(@I1) #And #

  // Arithmetic Normalization
  rule I +Int B => B +Int I          [concrete(I), symbolic(B), simplification(51)]
  rule A -Int I => A +Int (0 -Int I) [concrete(I), symbolic(A), simplification(51)]

  rule (A +Int I2) +Int I3 => A +Int (I2 +Int I3) [concrete(I2, I3), symbolic(A), simplifica
  rule I1 +Int (B +Int I3) => B +Int (I1 +Int I3) [concrete(I1, I3), symbolic(B), simplifica
  rule I1 -Int (B +Int I3) => (I1 -Int I3) -Int B [concrete(I1, I3), symbolic(B), simplifica
  rule I1 +Int (I2 +Int C) => (I1 +Int I2) +Int C [concrete(I1, I2), symbolic(C), simplifica
  rule I1 +Int (I2 -Int C) => (I1 +Int I2) -Int C [concrete(I1, I2), symbolic(C), simplifica
  rule (I1 -Int B) +Int I3 => (I1 +Int I3) -Int B [concrete(I1, I3), symbolic(B), simplifica
  rule I1 -Int (I2 +Int C) => (I1 -Int I2) -Int C [concrete(I1, I2), symbolic(C), simplifica
  rule I1 -Int (I2 -Int C) => (I1 -Int I2) +Int C [concrete(I1, I2), symbolic(C), simplifica
  rule (C -Int I2) -Int I3 => C -Int (I2 +Int I3) [concrete(I2, I3), symbolic(C), simplifica

  rule I1 &Int (I2 &Int C) => (I1 &Int I2) &Int C [concrete(I1, I2), symbolic(C), simplifica

endmodule

module INT-KAST [kast]
  imports private K-EQUAL
  imports INT-COMMON

  rule I1:Int ==Int I2:Int => I1 ==K I2
```

```
endmodule

module INT-KORE [kore, symbolic]
  imports private K-EQUAL
  imports private BOOL
  imports INT-COMMON

  rule I1:Int ==K I2:Int => I1 ==Int I2 [simplification]
  rule {K1 ==Int K2 #Equals true} => {K1 #Equals K2} [simplification]
  rule {true #Equals K1 ==Int K2} => {K1 #Equals K2} [simplification]
  rule {K1 ==Int K2 #Equals false} => #Not({K1 #Equals K2}) [simplification]
  rule {false #Equals K1 ==Int K2} => #Not({K1 #Equals K2}) [simplification]
  rule {K1 =/=Int K2 #Equals true} => #Not({K1 #Equals K2}) [simplification]
  rule {true #Equals K1 =/=Int K2} => #Not({K1 #Equals K2}) [simplification]
  rule {K1 =/=Int K2 #Equals false} => {K1 #Equals K2} [simplification]
  rule {false #Equals K1 =/=Int K2} => {K1 #Equals K2} [simplification]

endmodule

module INT
  imports INT-COMMON
  imports INT-SYMBOLIC
  imports INT-KAST
  imports INT-KORE
  imports private K-EQUAL
  imports private BOOL

  rule bitRangeInt(I::Int, IDX::Int, LEN::Int) => (I >>Int IDX) modInt (1 <<Int LEN)

  rule signExtendBitRangeInt(I::Int, IDX::Int, LEN::Int) => (bitRangeInt(I, IDX, LEN) +Int (

  rule I1:Int divInt I2:Int => (I1 -Int (I1 modInt I2)) /Int I2
  requires I2 =/=Int 0
  rule
    I1:Int modInt I2:Int
  =>
    ((I1 %Int absInt(I2)) +Int absInt(I2)) %Int absInt(I2)
  requires I2 =/=Int 0    [concrete, simplification]

  rule minInt(I1:Int, I2:Int) => I1 requires I1 <=Int I2
  rule minInt(I1:Int, I2:Int) => I2 requires I1 >=Int I2

  rule I1:Int =/=Int I2:Int => notBool (I1 ==Int I2)
  rule (I1:Int dividesInt I2:Int) => (I2 %Int I1) ==Int 0
```

```
  syntax Int ::= freshInt(Int)    [freshGenerator, function, functional, private]
  rule freshInt(I:Int) => I
endmodule
```

## IEEE 754 Floating-point Numbers

Provided here is the syntax of an implementation of arbitrary-precision floating-point arithmetic in K based on a generalization of the IEEE 754 standard. This type is hooked to an implementation of floats provided by the backend.

The syntax of ordinary floating-point values in K consists of an optional sign (+ or -) followed by an optional integer part, followed by a decimal point, followed by an optional fractional part. Either the integer part or the fractional part must be specified. The mantissa is followed by an optional exponent part, which consists of an `e` or `E`, an optional sign (+ or -), and an integer. The expoennt is followed by an optional suffix, which can be either `f`, `F`, `d`, `D`, or `pNxM` where `N` and `M` are positive integers. `p` and `x` can be either upper or lowercase.

The value of a floating-point literal is computed as follows: First the mantissa is read as a rational number. Then it is multiplied by 10 to the power of the exponent, which is interpreted as an integer, and defaults to zero if it is not present. Finally, it is rounded to the nearest possible value in a floating-point type represented like an IEEE754 floating-point type, with the number of bits of precision and exponent specified by the suffix. A suffix of `f` or `f` represents the IEEE `binary32` format. A suffix of `d` or `D`, or no suffix, represents the IEEE `binary64` format. A suffix of `pNxM` (either upper or lowercase) specifies exactly `N` bits of precision and `M` bits of exponent. The number of bits of precision is assumed to include any optional `1` that precedes the IEEE 754 mantissa. In other words, `p24x8` is equal to the IEEE `binary32` format, and `p53x11` is equal to the IEEE `binary64` format.

```
module FLOAT-SYNTAX
  syntax Float [hook(FLOAT.Float)]
  syntax Float ::= r"([\\+-]?[0-9]+(\\.[0-9]*)?|\\.[0-9]+)([eE][\\+-]?[0-9]+)?([fFdD]|([pP]
  syntax Float ::= r"[\\+-]?Infinity([fFdD]|([pP][0-9]+[xX][0-9]+))?" [token, prec(3)]
  syntax Float ::= r"NaN([fFdD]|([pP][0-9]+[xX][0-9]+))?" [token, prec(3)]
endmodule

module FLOAT
  imports FLOAT-SYNTAX
  imports private BOOL
  imports private INT-SYNTAX
```

### Float precision

You can retrieve the number of bits of precision in a `Float`.

```
syntax Int ::= precisionFloat(Float) [function, functional, hook(FLOAT.precision)]
```

### Float exponent bits

You can retrieve the number of bits of exponent range in a `Float`.

```
syntax Int ::= exponentBitsFloat(Float) [function, functional, hook(FLOAT.exponentBits)]
```

### Float exponent

You can retrieve the value of the exponent bits of a `Float` as an integer.

```
syntax Int ::= exponentFloat(Float) [function, functional, hook(FLOAT.exponent)]
```

### Float sign

You can retrieve the value of the sign bit of a `Float` as a boolean. True means the sign bit is set.

```
syntax Bool ::= signFloat(Float)      [function, functional, hook(FLOAT.sign)]
```

### Float special values

You can check whether a `Float` value is infinite or Not-a-Number.

```
syntax Bool ::= isNaN(Float)          [function, functional, smt-hook(fp.isNaN), hook(FLOA
              | isInfinite(Float)     [function, functional]
```

### Float arithmetic

You can:

- Compute the unary negation `--Float` of a float. `--Float X` is distinct from `0.0 -Float X`. For example, `0.0 -Float 0.0` is positive zero. `--Float 0.0` is negative zero.
- Compute the exponentation `^Float` of two floats.
- Compute the product `*Float`, quotient `/Float`, or remainder `%Float` of two floats. The remainder is computed based on rounding the quotient of the two floats to the nearest integer.
- Compute the sum `+Float` or difference `-Float` of two floats.

```
syntax Float ::= "--Float" Float             [function, functional, smt-hook(fp.neg), hook
               > Float "^Float" Float        [function, left, latex({#1}^{#2}), hook(FLOAT
               > left:
                 Float "*Float" Float        [function, left, smt-hook((fp.mul roundNeares
```

```
                | Float "/Float" Float        [function, left, smt-hook((fp.div roundNeares
                | Float "%Float" Float        [function, left, smt-hook((fp.rem roundNeares
                > left:
                  Float "+Float" Float        [function, left, smt-hook((fp.add roundNeares
                | Float "-Float" Float        [function, left, smt-hook((fp.sub roundNeares
```

**Floating-point mathematics**

You can:

- Compute the Nth integer root `rootFloat` of a float.
- Compute the absolute value `absFloat` of a float.
- Round a floating-point number to a specified precision and exponent range (`roundFloat`). The resulting `Float` will yield the specified values when calling `precisionFloat` and `exponentBitsFloat` and when performing
  further computation.
- Round a float to the next lowest floating-point value which is an integer (`floorFloat`).
- Round a float to the next highest floating-point value which is an integer (`ceilFloat`).
- Round a float to the next closest floating-point value which is an integer, in
  the direction of zero (`truncFloat`).
- Compute the natural exponential `expFloat` of a float (i.e. e^x).
- Compute the natural logarithm `logFloat` of a float.
- Compute the sine `sinFloat` of a float.
- Compute the cosine `cosFloat` of a float.
- Compute the tangent `tanFlooat` of a float.
- Compute the arcsine `asinFloat` of a float.
- Compute the arccosine `acosFloat` of a float.
- Compute the arctangent `atanFloat` of a float.
- Compute the arctangent `atan2Float` of two floats.
- Compute the maximum `maxFloat` of two floats.
- Compute the minimum `minFloat` of two floats.
- Compute the square root `sqrtFloat` of a float.
- Compute the largest finite value expressible in a specified precision and exponent range (`maxValueFloat`).
- Compute the smallest positive finite value expressible in a specified precision and exponent range (`minValueFloat`).

```
syntax Float ::= rootFloat(Float, Int)        [function, hook(FLOAT.root)]
               | absFloat(Float)              [function, functional, smt-hook(fp.abs), hoo
               | roundFloat(Float, precision: Int, exponentBits: Int)  [function, hook(FLO
               | floorFloat(Float)            [function, functional, hook(FLOAT.floor)]
               | ceilFloat(Float)             [function, functional, hook(FLOAT.ceil)]
               | truncFloat(Float)            [function, functional, hook(FLOAT.trunc)]
```

```
  | expFloat(Float)              [function, functional, hook(FLOAT.exp)]
  | logFloat(Float)              [function, hook(FLOAT.log)]
  | sinFloat(Float)              [function, functional, hook(FLOAT.sin)]
  | cosFloat(Float)              [function, functional, hook(FLOAT.cos)]
  | tanFloat(Float)              [function, hook(FLOAT.tan)]
  | asinFloat(Float)             [function, hook(FLOAT.asin)]
  | acosFloat(Float)             [function, hook(FLOAT.acos)]
  | atanFloat(Float)             [function, functional, hook(FLOAT.atan)]
  | atan2Float(Float, Float)     [function, hook(FLOAT.atan2)]
  | maxFloat(Float, Float)       [function, smt-hook(fp.max), hook(FLOAT.max)
  | minFloat(Float, Float)       [function, smt-hook(fp.min), hook(FLOAT.min)
  | sqrtFloat(Float)             [function]
  | maxValueFloat(precision: Int, exponentBits: Int)      [function, hook(FLO
  | minValueFloat(precision: Int, exponentBits: Int)      [function, hook(FLO
```

### Floating-point comparisons

Compute whether a float is less than or equasl to, less than, greater than or equal to, greater than, equal, or unequal to another float. Note that `X ==Float Y` and `X ==K Y` might yield different values. The latter should be used in cases where you want to compare whether two values of sort `Float` contain the same term. The former should be used when you want to implement the `==` operator of a programming language. In particular, `NaN =/=Float NaN` is true, because `NaN` compares unequal to all values, including itself, in IEEE 754 arithmetic. `0.0 ==Float -0.0` is also true.

```
syntax Bool ::= Float "<=Float" Float       [function, left, smt-hook(fp.leq), latex({#1}\
              | Float "<Float" Float        [function, left, smt-hook(fp.lt), latex({#1}\m
              | Float ">=Float" Float       [function, left, smt-hook(fp.geq), latex({#1}\
              | Float ">Float" Float        [function, left, smt-hook(fg.gt), latex({#1}\m
              | Float "==Float" Float       [function, left, smt-hook(fp.eq), latex({#1}\m
              | Float "=/=Float" Float      [function, left, smt-hook((not (fp.eq #1 #2)))

rule F1:Float =/=Float F2:Float => notBool (F1 ==Float F2)
```

### Conversion between integer and float

You can convert an integer to a floating-point number with the specified precision and exponent range. You can also convert a floating-point number to the nearest integer. This operation rounds to the nearest integer, but it also avoids the double-rounding that is present in `ceilFloat` and `floorFloat` if the nearest integer is not representable in the specified floating-point type.

```
syntax Float ::= Int2Float(Int, precision: Int, exponentBits: Int)    [function, latex({\\
syntax Int ::= Float2Int(Float)    [function, functional, latex({\\it{}Float2Int}), hook(F
```

### Implementation of Floats

The remainder of this section consists of an implementation in K of some of the operators above.

```
  rule sqrtFloat(F:Float) => rootFloat(F, 2)

  rule isInfinite(F:Float) => F >Float maxValueFloat(precisionFloat(F), exponentBitsFloat(F)

endmodule
```

## Strings

Provided here is the syntax of an implementation of Unicode strings in K. This type is hooked to an implementation of strings provided by the backend. The implementation is currently incomplete and does not fully support encodings and code points beyond the initial 256 code points of the Basic Latin and Latin-1 Supplement blocks. In the future, there may be breaking changes to the semantics of this module in order to support this functionality.

The syntax of strings in K is delineated by double quotes. Inside the double quotes, any character can appear verbatim except double quotes, backslash, newline, and carriage return. K also supports the following escape sequences:

- " - the " character
- \ - the \ character
- \n - newline character
- \r - carriage return character
- \t - tab character
- \f - form feed character
- \xFF - \x followed by two hexadecimal characters indicates a code point between 0x00 and 0xff
- \uFFFF - \u followed by four hexadecimal characters indicates a code point
  between 0x0000 and 0xffff
- \UFFFFFFFF - \U followed by eight hexadecimal characters indicates a code
  point between 0x000000 and 0x10ffff

```
module STRING-SYNTAX
  syntax String [hook(STRING.String)]
  syntax String ::= r"[\\"](([^\\"\\n\\r\\\\])|([\\\\][nrtf\\"\\\\])|([\\\\][x][0-9a-fA-F
endmodule

module STRING-COMMON
  imports STRING-SYNTAX
  imports private INT
  imports private FLOAT-SYNTAX
```

```
imports private K-EQUAL
imports private BOOL
```

### String concatenation

You can concatenate two strings in O(N) time. For successive concatenation operations, it may be better to use the `STRING-BUFFER` module.

```
syntax String ::= String "+String" String    [function, functional, left, latex({#1}+_{\sc
```

### String length

You can get the length of a string in O(1) time.

```
syntax Int ::= lengthString ( String ) [function, functional, hook(STRING.length)]
```

### Character and integer conversion

You can convert between a character (as represented by a string containing a single code point) and an integer in O(1) time.

```
syntax String ::= chrChar ( Int )      [function, hook(STRING.chr)]
syntax Int ::= ordChar ( String )      [function, hook(STRING.ord)]
```

### String substring

You can compute a substring of a string in O(N) time (where N is the length of the substring). There are two important facts to note:

1. the range generated includes the character at `startIndex` but excludes the
   character at `endIndex`, i.e., the range is `[startIndex..endIndex)`.
2. this function is only defined on valid indices (i.e., it is defined when `startIndex < endIndex` and `endIndex` is less than or equal to the string length).

```
syntax String ::= substrString ( String , startIndex: Int , endIndex: Int ) [function, fu
```

### String search

You can find the first (respectively, last) occurrence of a substring, starting at a certain `index`, in another string in O(N*M) time.
Returns `-1` if the substring is not found.

```
syntax Int ::= findString ( haystack: String , needle: String , index: Int )   [function,
syntax Int ::= rfindString ( haystack: String , needle: String , index: Int )  [function,
```

### String character search

You can find the first (respectively, last) occurrence of one of the characters of the search string, starting at a certain `index`, in another string in O(N*M) time.

```
syntax Int ::= findChar ( haystack: String , needles: String , index: Int )     [function,
syntax Int ::= rfindChar ( haystack: String , needles: String , index: Int )    [function,
```

### String and Bool conversion

```
syntax String ::= Bool2String(Bool) [function, functional]
rule Bool2String(true)  => "true"
rule Bool2String(false) => "false"

syntax Bool ::= String2Bool(String) [function]
rule String2Bool("true")  => true
rule String2Bool("false") => false
```

### String and float conversion

You can convert between a `String` and a `Float`. The String will be represented in the syntax of the `Float` sort (see the section on the `FLOAT` module above for details of that syntax). Which particular string is returned by `Float2String` is determined by the backend, but the same `Float` is guaranteed to return the same `String`, and converting that `String` back to a `Float` is guaranteed to return the original `Float`.

You can also convert a `Float` to a string in a particular syntax using the variant of `Float2String` with a `format`. In this case, the resulting string is one which results directly from passing that `format` to `mpfr_printf`. This functionality may not be supported on backends that do not use Gnu MPFR to implement floating-point numbers.

```
syntax String ::= Float2String ( Float )              [function, functional, hook(STRING.f
syntax String ::= Float2String ( Float , format: String )     [function, klabel(FloatForma
syntax Float  ::= String2Float ( String )             [function, hook(STRING.string2float)
```

### String and integer conversions

You can convert between a `String` and an `Int`. The String will be represented in the syntax of the `INT` module (i.e., a nonempty sequence of digits optionally prefixed by a sign). When converting from an `Int` to a `String`, the sign will not be present unless the integer is negative.

You can also convert between a `String` and an `Int` in a particular radix. This radix can be anywhere between 2 and 36. For a radix 2 <= N <= 10, the digits 0 to N-1 will be used. For a radix 11 <= N <= 36, the digits 0 to 9 and the first N-10 letters of the Latin alphabet will be used. Both uppercase

and lowercase letters are supported by `String2Base`. Whether the letters returned by `Base2String` are upper or lowercase is determined by the backend, but the backend will consistently choose one or the other.

```
syntax Int    ::= String2Int   ( String )             [function, hook(STRING.string2int)]
syntax String ::= Int2String   ( Int )                [function, functional, hook(STRING.i
syntax String ::= Base2String  ( Int , base: Int )          [function, hook(STRING.base2st
syntax Int    ::= String2Base  ( String , base: Int )       [function, hook(STRING.string2
```

### String count and replace

You can replace one, some, or all occurrences of a string within another string in O(N*M) time. The `replaceAll`, `replace`, and `replaceFirst` methods are identical, except `replaceFirst` replaces exactly one ocurrence of the string, the first occurrence. `replace` replaces the first `times` occurrences. And `replaceAll` replaces every occurrence.

You can also count the number of times a string occurs within another string using `countAllOccurrences`.

```
syntax String ::= "replaceAll" "(" haystack: String "," needle: String "," replacement: St
syntax String ::= "replace" "(" haystack: String "," needle: String "," replacement: Strin
syntax String ::= "replaceFirst" "(" haystack: String "," needle: String "," replacement:
syntax Int ::= "countAllOccurrences" "(" haystack: String "," needle: String ")"
```

### String equality and lexicographic comparison

You can compare whether two strings are equal or unequal, or whether one string
is less than, less than or equal to, greater than, or greater than or equal to another according to the natural lexicographic ordering of strings.

```
syntax Bool ::= String "==String" String [function, functional, left, hook(STRING.eq)]
              | String "=/=String" String      [function, functional, left, hook(STRING.ne
rule S1:String =/=String S2:String => notBool (S1 ==String S2)

syntax Bool ::= String  "<String" String [function, functional, hook(STRING.lt)]
              | String "<=String" String [function, functional, hook(STRING.le)]
              | String  ">String" String [function, functional, hook(STRING.gt)]
              | String ">=String" String [function, functional, hook(STRING.ge)]
```

### Implementation of Strings

What follows is a few String hooks which are deprecated and only are supported on certain outdated backends of K, as well as an implementation of several of the above operations in K.

```
syntax String ::= categoryChar(String)       [function, hook(STRING.category)]
                | directionalityChar(String) [function, hook(STRING.directionality)]
```

```
  syntax String ::= "newUUID" [function, hook(STRING.uuid), impure]

  rule S1:String <=String S2:String => notBool (S2 <String S1)
  rule S1:String >String S2:String => S2 <String S1
  rule S1:String >=String S2:String => notBool (S1 <String S2)

  rule findChar(S1:String, S2:String, I:Int) => #if findString(S1, substrString(S2, 0, 1), I
  rule findChar(_, "", _) => -1
  rule rfindChar(S1:String, S2:String, I:Int) => maxInt(rfindString(S1, substrString(S2, 0,
  rule rfindChar(_, "", _) => -1

  rule countAllOccurrences(Source:String, ToCount:String) => 0
           requires findString(Source, ToCount, 0) <Int 0
  rule countAllOccurrences(Source:String, ToCount:String) => 1 +Int countAllOccurrences(subs
           requires findString(Source, ToCount, 0) >=Int 0

  rule replaceFirst(Source:String, ToReplace:String, Replacement:String) => substrString(Sou
               +String Replacement +String substrString(Source, findString(Source, ToReplac
               requires findString(Source, ToReplace, 0) >=Int 0
  rule replaceFirst(Source:String, ToReplace:String, _:String) => Source
       requires findString(Source, ToReplace, 0) <Int 0


  rule replace(Source:String, ToReplace:String, Replacement:String, Count:Int) =>
       substrString(Source, 0, findString(Source, ToReplace, 0)) +String Replacement +String
       replace(substrString(Source, findString(Source, ToReplace, 0) +Int lengthString(ToRep
        requires Count >Int 0
  rule replace(Source:String, _, _, 0) => Source
  rule replaceAll(Source:String, ToReplace:String, Replacement:String) => replace(Source, To

endmodule

module STRING-KAST [kast]
  imports private K-EQUAL
  imports STRING-COMMON

  rule S1:String ==String S2:String => S1 ==K S2

endmodule

module STRING-KORE [kore, symbolic]
  imports private K-EQUAL
  imports STRING-COMMON

  rule S1:String ==K S2:String => S1 ==String S2 [simplification]
```

```
endmodule

module STRING
  imports STRING-COMMON
  imports STRING-KAST
  imports STRING-KORE
endmodule
```

### String Buffers

It is a well known fact that repeated string concatenations are quadratic in performance whereas use of an efficient mutable representation of arrays can yield linear performance. We thus provide such a sort, the `StringBuffer` sort. Axiomatically, it is implemented below on symbolic backends using the `String` module. However, on concrete backends it provides an efficient implementation of string concatenation. There are three operations:

- `.StringBuffer` creates a new `StringBuffer` with current content equal to the empty string.
- `+String` takes a `StringBuffer` and a `String` and appends the `String` to the end of the `StringBuffer`
- `StringBuffer2String` converts a `StringBuffer` to a `String`. This operation
  copies the string so that subsequent modifications to the `StringBuffer` will not change the value of the `String` returned by this function.

```
module STRING-BUFFER-IN-K [symbolic]
  imports private BASIC-K
  imports STRING

  syntax StringBuffer ::= ".StringBuffer" [function, functional]
  syntax StringBuffer ::= StringBuffer "+String" String [function, functional, left, avoid]
  syntax StringBuffer ::= String
  syntax String ::= StringBuffer2String ( StringBuffer ) [function, functional]

  rule {SB:String +String S:String}<:StringBuffer => (SB +String S)::String
  rule .StringBuffer => ""
  rule StringBuffer2String(S:String) => S
endmodule

module STRING-BUFFER-HOOKED [concrete]
  imports private BASIC-K
  imports STRING

  syntax StringBuffer [hook(BUFFER.StringBuffer)]
  syntax StringBuffer ::= ".StringBuffer" [function, functional, hook(BUFFER.empty), impure]
```

```
  syntax StringBuffer ::= StringBuffer "+String" String [function, functional, left, hook(BU
  syntax String ::= StringBuffer2String ( StringBuffer ) [function, functional, hook(BUFFER.
endmodule

module STRING-BUFFER
  imports STRING-BUFFER-HOOKED
  imports STRING-BUFFER-IN-K
endmodule
```

## Byte Arrays

Provided here is the syntax of an implementation of fixed-width arrays of Bytes in K. This type is hooked to an implementation of bytes provided by the backend. In concrete backends, this representation is mutable and thus multiple references can occur to the same `Bytes` object and when one is modified, the others are also modified. Care should be taken not to rely on this fact however as this is not the case in symbolic backends and thus you will experience divergent behavior unless the `Bytes` type is used in a manner that preserves consistency.

```
module BYTES-SYNTAX
  imports private STRING-SYNTAX

  syntax Bytes [hook(BYTES.Bytes)]
  syntax Bytes ::= r"b[\\"](([^\\"\\n\\r\\\\])|([\\\\][nrtf\\"\\\\])|([\\\\][x][0-9a-fA-F
endmodule

module BYTES-STRING-ENCODE [kore, symbolic]
  imports BYTES-SYNTAX
```

### Encoding/decoding between Bytes and String

You can encode/decode between Bytes and String using `UTF-8`, `UTF-16LE`, `UTF-16BE`, `UTF-32LE`, and `UTF-32BE`

```
    syntax String ::= decodeBytes ( encoding: String , contents: Bytes ) [function, hook(BYT
    syntax Bytes ::= encodeBytes ( encoding: String , contents: String ) [function, hook(BYT
endmodule

module BYTES-HOOKED
  imports STRING-SYNTAX
  imports BYTES-SYNTAX
  imports BYTES-STRING-ENCODE
```

### Empty byte array

The byte array of length zero is represented by `.Bytes`.

```
  syntax Bytes ::= ".Bytes" [function, functional, hook(BYTES.empty)]
```

### Endianness

When converting to/from an integer, byte arrays can be treated as either little endian (ie, least significant byte first) or big endian (ie, most significant byte first).

```
syntax Endianness ::= "LE" [klabel(littleEndianBytes), symbol]
                    | "BE" [klabel(bigEndianBytes), symbol]
```

### Signedness

When converting to/from an integer, byte arrays can be treated as either signed or unsigned.

```
syntax Signedness ::= "Signed" [klabel(signedBytes), symbol]
                    | "Unsigned" [klabel(unsignedBytes), symbol]
```

### Integer and Bytes conversion

You can convert from a `Bytes` to an `Int`. In order to do this, the endianness and signedness of the `Bytes` must be provided. The resulting integer is created by means of interpreting the `Bytes` as either a twos-complement representation, or an unsigned representation, of an integer, in the specified byte order.

You can also convert from an `Int` to a `Bytes`. This comes in two variants. In the first, the `length` of the resulting `Bytes` in bytes is explicitly specified. If the `length` is greater than the highest set bit in the magnitude of the integer, the result is padded with 0 bits if the number is positive and 1 bits if the number is negative. If the `length` is less than the highest bit set in the magnitude of the integer, the most-significant bits of the integer will be truncated. The endianness of the resulting `Bytes` object is as specified.

In the second variant, both endianness and signedness are specified, and the resulting `Bytes` object will be the smallest number of bytes necessary for the resulting `Bytes` object to be convertible back to the original integer via `Bytes2Int`. In other words, if the highest bit set in the magnitude of the integer is N, then the byte array will be at least N+1 bits long, rounded up to the nearest byte.

```
syntax Int ::= Bytes2Int(Bytes, Endianness, Signedness) [function, functional, hook(BYTES.
syntax Bytes ::= Int2Bytes(length: Int, Int, Endianness) [function, functional, hook(BYTES
               | Int2Bytes(Int, Endianness, Signedness) [function, functional, klabel(Int2
```

### String and Bytes conversion

You can convert between a `Bytes` and a `String` in O(N) time. The resulting value is a copy of the original and will not be affected by subsequent mutations of the input or output value.

```
syntax String ::= Bytes2String(Bytes) [function, functional, hook(BYTES.bytes2string)]
syntax Bytes ::= String2Bytes(String) [function, functional, hook(BYTES.string2bytes)]
```

### Bytes update

You can set the value of a particular byte in a `Bytes` object in O(1) time. The result is `#False` if `value` is not in the range [0..255] or if `index` is not a valid index (ie, less than zero or greater than or equal to the length of the `Bytes` term).

```
syntax Bytes ::= Bytes "[" index: Int "<-" value: Int "]" [function, hook(BYTES.update)]
```

### Bytes lookup

You can get the value of a particular byte in a `Bytes` object in O(1) time. The result is `#False` if `index` is not a valid index (see above).

```
syntax Int ::= Bytes "[" Int "]" [function, hook(BYTES.get)]
```

### Bytes substring

You can get a new `Bytes` object containing a range of bytes from the input `Bytes` in O(N) time (where N is the length of the substring). The range of bytes included is `[startIndex..endIndex)`. The resulting `Bytes` is a copy and mutations to it do not affect mutations to the original `Bytes`. The result is `#False` if `startIndex` or `endIndex` are not valid.

```
syntax Bytes ::= substrBytes(Bytes, startIndex: Int, endIndex: Int) [function, hook(BYTES.
```

### Multiple bytes update

You can modify a `Bytes` to return a `Bytes` which is equal to `dest` except the `N` elements starting at `index` are replaced with the contents of `src` in O(N) time. This does not create a new `Bytes` object and will instead modify the original on concrete backends. The result is `#False` if `index` + `N` is not a valid index.

```
syntax Bytes ::= replaceAtBytes(dest: Bytes, index: Int, src: Bytes) [function, hook(BYTES
```

### Bytes padding

You can create a new `Bytes` object which is at least `length` bytes long by taking the input sequence and padding it on the right (respectively, on the left) with the specified `value`. This does not create a new `Bytes` object if the input is already at least `length` bytes long, and will instead return the input unchanged. The result is `#False` if `value` is not in the range `[0..255]`, or if the length is negative.

```
syntax Bytes ::= padRightBytes(Bytes, length: Int, value: Int) [function, hook(BYTES.padRi
               | padLeftBytes(Bytes, length: Int, value: Int) [function, hook(BYTES.padLef
```

### Bytes reverse

You can reverse a `Bytes` object in O(N) time. This does not create a new `Bytes` object and will instead modify the original on concrete backends.

```
  syntax Bytes ::= reverseBytes(Bytes) [function, functional, hook(BYTES.reverse)]
```

### Bytes length

You can get the length of a `Bytes` term in O(1) time.

```
  syntax Int ::= lengthBytes(Bytes) [function, functional, hook(BYTES.length), smtlib(lengt
```

### Bytes concatenation

You can create a new `Bytes` object by concatenating two `Bytes` objects together in O(N) time.

```
  syntax Bytes ::= Bytes "+Bytes" Bytes [function, functional, hook(BYTES.concat), right]

endmodule
```

### Implementation of Bytes

The remainder of this module consists of an implementation of some of the operators listed above in K. This implementation is primarily used by outdated backends and should not be viewed as authoritative, nor should the user use the `nilBytes` or `:` operators in their definition.

```
module BYTES-IN-K [symbolic, kast]
  imports INT
  imports K-EQUAL
  imports STRING
  imports STRING-BUFFER
  imports BOOL

  syntax Bytes ::= "nilBytes"
                 | Int ":" Bytes
  syntax Endianness ::= "LE" [klabel(littleEndianBytes), symbol]
                      | "BE" [klabel(bigEndianBytes), symbol]
  syntax Signedness ::= "Signed" [klabel(signedBytes), symbol]
                      | "Unsigned" [klabel(unsignedBytes), symbol]

  syntax Bytes ::= ".Bytes" [function, functional]
  rule .Bytes => nilBytes

  syntax Int ::= Bytes2Int(Bytes, Endianness, Signedness) [function, functional]
  rule Bytes2Int(nilBytes, _, _) => 0
  rule Bytes2Int(B : nilBytes, BE, Unsigned) => B
```

```
rule Bytes2Int(B0 : B1 : BS, BE, Unsigned) => Bytes2Int(((B0 <<Int 8) |Int B1) : BS, BE, U
rule Bytes2Int(B0 : BS, BE, Signed) => signExtendBitRangeInt(Bytes2Int(B0 : BS, BE, Unsign
rule Bytes2Int(B0 : BS, LE, S) => Bytes2Int(reverseBytes(B0 : BS), BE, S)

syntax Bytes ::= Int2Bytes(Int, Bytes) [function, klabel(Int2BytesAux)]
syntax Bytes ::= Int2Bytes(Int, Int, Endianness) [function, functional]
               | Int2Bytes(Int, Endianness, Signedness) [function, functional, klabel(Int2
rule Int2Bytes(LEN, I, BE) => padLeftBytes(Int2Bytes(bitRangeInt(I, 0, LEN <<Int 3), nilBy
rule Int2Bytes(LEN, I, LE) => reverseBytes(Int2Bytes(LEN, I, BE))
rule Int2Bytes(0, BS) => BS
rule Int2Bytes(I, BS) => Int2Bytes(I >>Int 8, I &Int 255 : BS) requires I =/=Int 0

syntax String ::= Bytes2String(Bytes, StringBuffer) [function, klabel(Bytes2StringAux)]
syntax String ::= Bytes2String(Bytes) [function, functional]
rule Bytes2String(BS) => Bytes2String(BS, .StringBuffer)
rule Bytes2String(nilBytes, BUFFER) => StringBuffer2String(BUFFER)
rule Bytes2String(B : BS, BUFFER) => Bytes2String(BS, BUFFER +String chrChar(B))

syntax Bytes ::= String2Bytes(String) [function, functional]
rule String2Bytes(S) => ordChar(substrString(S, 0, 1)) : String2Bytes(substrString(S, 1, I
rule String2Bytes("") => nilBytes

syntax Bytes ::= Bytes "[" Int "<-" Int "]" [function]
rule BS [ N <- M ] => substrBytes(BS, 0, N) +Bytes M : substrBytes(BS, N +Int 1, lengthByt

syntax Int ::= Bytes "[" Int "]" [function]
rule (B : _) [ 0 ] => B
rule (_ : BS) [ I ] => BS [ I -Int 1] requires I >Int 0

syntax Bytes ::= substrBytes(Bytes, Int, Int) [function]
rule substrBytes(_, 0, 0) => nilBytes
rule substrBytes(_ : BS, N, M) => substrBytes(BS, N -Int 1, M -Int 1) requires N >Int 0
rule substrBytes(B : BS, 0, M) => B : substrBytes(BS, 0, M -Int 1) requires M >Int 0

syntax Bytes ::= replaceAtBytes(Bytes, Int, Bytes) [function]
rule replaceAtBytes(BS, _, nilBytes) => BS
rule replaceAtBytes(B : BS, N, BS') => B : replaceAtBytes(BS, N -Int 1, BS') requires N >I
rule replaceAtBytes(_ : BS, 0, B : BS') => B : replaceAtBytes(BS, 0, BS')

syntax Bytes ::= padRightBytes(Bytes, Int, Int) [function]
               | padLeftBytes(Bytes, Int, Int) [function]
rule padRightBytes(BS, LEN, VAL) => reverseBytes(padLeftBytes(reverseBytes(BS), LEN, VAL))
rule padLeftBytes(BS, LEN, _) => BS requires lengthBytes(BS) >=Int LEN andBool 0 <=Int LEN
rule padLeftBytes(BS, LEN, VAL) => padLeftBytes(VAL : BS, LEN, VAL) requires lengthBytes(B

syntax Bytes ::= reverseBytes(Bytes) [function, functional]
```

```
  syntax Bytes ::= reverseBytes(Bytes, Bytes) [function, klabel(reverseBytesAux)]
  rule reverseBytes(BS) => reverseBytes(BS, nilBytes)
  rule reverseBytes(nilBytes, BS) => BS
  rule reverseBytes(B : BS, BS') => reverseBytes(BS, B : BS')

  syntax Int ::= lengthBytes(Bytes) [function, functional, smtlib(lengthBytes)]
  syntax Int ::= lengthBytes(Bytes, Int) [function, klabel(lengthBytesAux), smtlib(lengthByt
  rule lengthBytes(BS) => lengthBytes(BS, 0)
  rule lengthBytes(nilBytes, SIZE) => SIZE
  rule lengthBytes(_ : BS, SIZE) => lengthBytes(BS, SIZE +Int 1)

  syntax Bytes ::= Bytes "+Bytes" Bytes [function, functional, right]
  rule nilBytes +Bytes B2 => B2
  rule (B : BS) +Bytes B2 => B : (BS +Bytes B2)
endmodule

module BYTES-CONCRETE [concrete]
  imports BYTES-HOOKED
endmodule

module BYTES-KORE [kore]
  imports BYTES-HOOKED
  imports BYTES-SYMBOLIC-CEIL
endmodule

module BYTES-SYMBOLIC-CEIL [symbolic, kore]
  imports BYTES-HOOKED
  imports private INT
  imports private BOOL

  rule #Ceil(padRightBytes(_, LEN, VAL)) => {(0 <=Int LEN andBool 0 <=Int VAL andBool VAL <I
  rule #Ceil(padLeftBytes(_, LEN, VAL))  => {(0 <=Int LEN andBool 0 <=Int VAL andBool VAL <I
endmodule

module BYTES
  imports BYTES-CONCRETE
  imports BYTES-KORE
  imports BYTES-IN-K
  imports private INT

  rule Int2Bytes(I::Int, E::Endianness, Unsigned) => Int2Bytes((log2Int(I) +Int 8) /Int 8, I
    requires I >Int 0
  rule Int2Bytes(0, _::Endianness, _) => .Bytes
  rule Int2Bytes(I::Int, E::Endianness, Signed) => Int2Bytes((log2Int(I) +Int 9) /Int 8, I,
    requires I >Int 0
  rule Int2Bytes(I::Int, E::Endianness, Signed) => Int2Bytes((log2Int(~Int I) +Int 9) /Int 8
```

```
    requires I <Int -1
  rule Int2Bytes(-1, E::Endianness, Signed) => Int2Bytes(1, -1, E)
endmodule
```

## Program identifiers

Provided here is an implementation for program identifiers in K. Developers of semantics for a particular language may wish to use their own implementation instead of the one provided here if their syntax differs from the syntax defined below. However, this is provided for convenience for developers who do not care about the lexical syntax of identifiers.

Provided are the following pieces of functionality:

- `Id2String` - Convert an `Id` to a `String` containing its name
- `String2Id` - Convert a `String` to an `Id` with the specified name
- !X:Id - You can get a fresh identifier distinct from any previous identifier generated by this syntax.

```
module ID-SYNTAX-PROGRAM-PARSING
  imports BUILTIN-ID-TOKENS
  syntax Id ::= r"(?<![A-Za-z0-9\\_])[A-Za-z\\_][A-Za-z0-9\\_]*"     [prec(1), token]
              | #LowerId                                              [token]
              | #UpperId                                              [token]
endmodule

module ID-SYNTAX
  syntax Id [token]
endmodule

module ID-COMMON
  imports ID-SYNTAX
  imports private STRING

  syntax String ::= Id2String ( Id )    [function, functional, hook(STRING.token2string)]
  syntax Id ::= String2Id (String) [function, functional, hook(STRING.string2token)]
  syntax Id ::= freshId(Int)    [freshGenerator, function, functional, private]

  rule freshId(I:Int) => String2Id("_" +String Int2String(I))
endmodule

module ID
  imports ID-COMMON
  imports ID-SYMBOLIC
endmodule

module ID-SYMBOLIC [symbolic, kast, private]
```

```
  imports public ID-COMMON
  imports private STRING

  syntax KItem  ::= "#parseIdToken"  "(" String "," String ")"  [function, hook(STRING.parse
  rule String2Id(S:String) => {#parseIdToken("Id", S)}:>Id
endmodule
```

## Equality and conditionals

Provided here are implementations of two important primitives in K:

- `==K` - the equality between two terms. Returns `true` if they are equal and `false` if they are not equal.
- `#if #then #else #fi` - polymorphic conditional function. If the first argument evaluates to `true`, the second argument is returned. Otherwise, the third argument is returned. Note that this does not short-circuit on symbolic backends.

```
module K-EQUAL-SYNTAX
  imports private BOOL
  imports private BASIC-K

  syntax Bool ::= left:
                  K "==K" K           [function, functional, smt-hook(=), hook(KEQUAL.eq), k
                | K "=/=K" K          [function, functional, smt-hook(distinct), hook(KEQUAL

  syntax priorities equalEqualK notEqualEqualK > boolOperation mlOp

  syntax {Sort} Sort ::= "#if" Bool "#then" Sort "#else" Sort "#fi"   [function, functional,

endmodule

module K-EQUAL-KORE [kore, symbolic]
  import private BOOL
  import K-EQUAL-SYNTAX

  rule K1:Bool ==K K2:Bool => K1 ==Bool K2 [simplification]
  rule {K1 ==K K2 #Equals true} => {K1 #Equals K2} [simplification]
  rule {true #Equals K1 ==K K2} => {K1 #Equals K2} [simplification]
  rule {K1 ==K K2 #Equals false} => #Not({K1 #Equals K2}) [simplification]
  rule {false #Equals K1 ==K K2} => #Not({K1 #Equals K2}) [simplification]
  rule {K1 =/=K K2 #Equals true} => #Not({K1 #Equals K2}) [simplification]
  rule {true #Equals K1 =/=K K2} => #Not({K1 #Equals K2}) [simplification]
  rule {K1 =/=K K2 #Equals false} => {K1 #Equals K2} [simplification]
  rule {false #Equals K1 =/=K K2} => {K1 #Equals K2} [simplification]

endmodule
```

```
module K-EQUAL-KAST [kast]
  import private BOOL
  import K-EQUAL-SYNTAX

  rule K1:Bool ==Bool K2:Bool => K1 ==K K2

endmodule

module K-EQUAL
  import private BOOL
  import K-EQUAL-SYNTAX
  import K-EQUAL-KAST
  import K-EQUAL-KORE

  rule K1:K =/=K K2:K => notBool (K1 ==K K2)

  rule #if C:Bool #then B1::K #else _ #fi => B1 requires C
  rule #if C:Bool #then _ #else B2::K #fi => B2 requires notBool C

endmodule
```

### Meta operations

Provided below are a few miscellaneous, mostly deprecated functions in K.
It is not recommended to use any of them directly as they are largely unsupported in modern K. There are a few exceptions:

- `#getenv` - Returns the value of an environment variable
- `#parseKORE` - Takes a String containing a K intermediate representation of
  a term such as is returned by `kast -o kore` and converts it to a term. This is NOT type-safe. The responsibility is on the user to ensure that the
  string they provide is a valid representation of a term of the sort *exactly* equal to the sort where the function appears.
- `#kompiledDirectory` - Returns the path to the current compiled K definition
  directory.
- `#unparseKORE` = Takes a K term and converts it to a string.

```
module K-REFLECTION
  imports BASIC-K
  imports STRING
  imports K-REFLECTION-SYMBOLIC

  syntax K ::= "#configuration" [function, impure, hook(KREFLECTION.configuration)]
```

```
  syntax String ::= #sort(K) [function, hook(KREFLECTION.sort)]
  syntax KItem ::= #fresh(String)   [function, hook(KREFLECTION.fresh), impure]
  syntax KItem ::= getKLabel(K)  [function, hook(KREFLECTION.getKLabel)]

  syntax K ::= #getenv(String) [function, impure, hook(KREFLECTION.getenv)]

  syntax String ::= #kompiledDirectory() [function, hook(KREFLECTION.kompiledDir)]

  // meaningful only for the purposes of compilation to a binary, otherwise
  // undefined
  syntax List ::= #argv() [function, hook(KREFLECTION.argv)]

  // Takes as input a string and returns a K term
  syntax {Sort} Sort ::= #parseKORE(String) [function, hook(KREFLECTION.parseKORE)]
  syntax {Sort} String ::= #unparseKORE(Sort) [function, hook(KREFLECTION.printKORE)]
  syntax {Sort} Sort ::= #parseKAST(String) [function, hook(KREFLECTION.parseKAST)]
  syntax IOError ::= "#noParse" "(" String ")" [klabel(#noParse), symbol]

endmodule

module K-REFLECTION-SYMBOLIC [symbolic, kast]
  imports BASIC-K
  imports STRING

  // return empty string if the term has no klabel
  syntax String ::= #getKLabelString(K) [function, hook(KREFLECTION.getKLabelString)]

  // return true if no variable nor unresolved function appears in any subterm
  syntax Bool ::= #isConcrete(K) [function, hook(KREFLECTION.isConcrete)]
  syntax Bool ::= #isVariable(K) [function, hook(KREFLECTION.isVariable)]
endmodule
```

## I/O in K

Concrete execution in K supports I/O operations. This functionality is not supported during symbolic execution, because symbolic execution must exist completely free of side-effects, and I/O is an irreducible type of side effect. However, it is useful in many cases when defining concrete execution to be able to make reference to I/O operations.

The design of these I/O operations is based on the POSIX standard, for the most
part. For example, the `#read` K function maps to the `read` POSIX function. We
do not at this time have a higher-level API for I/O, but this may be implemented at some point in the future.

I/O operations generally return either their result, or an `IOError` term corresponding to the `errno` returned by the underlying system call.

```
module K-IO
  imports private LIST
  imports private STRING
  imports private INT
```

### I/O errors

Aside from EOF, which is returned by `#getc` if the file is at end-of-file, all of the below I/O errors correspond to possible values for `errno` after calling a library function. If the `errno` returned is not one of the below errnos known to K, `#unknownIOError` is returned along with the integer errno value.

```
  syntax IOError ::= "#EOF" [klabel(#EOF), symbol] | #unknownIOError(errno: Int) [symbol]
                   | "#E2BIG" [klabel(#E2BIG), symbol]
                   | "#EACCES" [klabel(#EACCES), symbol]
                   | "#EAGAIN" [klabel(#EAGAIN), symbol]
                   | "#EBADF" [klabel(#EBADF), symbol]
                   | "#EBUSY" [klabel(#EBUSY), symbol]
                   | "#ECHILD" [klabel(#ECHILD), symbol]
                   | "#EDEADLK" [klabel(#EDEADLK), symbol]
                   | "#EDOM" [klabel(#EDOM), symbol]
                   | "#EEXIST" [klabel(#EEXIST), symbol]
                   | "#EFAULT" [klabel(#EFAULT), symbol]
                   | "#EFBIG" [klabel(#EFBIG), symbol]
                   | "#EINTR" [klabel(#EINTR), symbol]
                   | "#EINVAL" [klabel(#EINVAL), symbol]
                   | "#EIO" [klabel(#EIO), symbol]
                   | "#EISDIR" [klabel(#EISDIR), symbol]
                   | "#EMFILE" [klabel(#EMFILE), symbol]
                   | "#EMLINK" [klabel(#EMLINK), symbol]
                   | "#ENAMETOOLONG" [klabel(#ENAMETOOLONG), symbol]
                   | "#ENFILE" [klabel(#ENFILE), symbol]
                   | "#ENODEV" [klabel(#ENODEV), symbol]
                   | "#ENOENT" [klabel(#ENOENT), symbol]
                   | "#ENOEXEC" [klabel(#ENOEXEC), symbol]
                   | "#ENOLCK" [klabel(#ENOLCK), symbol]
                   | "#ENOMEM" [klabel(#ENOMEM), symbol]
                   | "#ENOSPC" [klabel(#ENOSPC), symbol]
                   | "#ENOSYS" [klabel(#ENOSYS), symbol]
                   | "#ENOTDIR" [klabel(#ENOTDIR), symbol]
                   | "#ENOTEMPTY" [klabel(#ENOTEMPTY), symbol]
                   | "#ENOTTY" [klabel(#ENOTTY), symbol]
                   | "#ENXIO" [klabel(#ENXIO), symbol]
                   | "#EPERM" [klabel(#EPERM), symbol]
```

```
| "#EPIPE" [klabel(#EPIPE), symbol]
| "#ERANGE" [klabel(#ERANGE), symbol]
| "#EROFS" [klabel(#EROFS), symbol]
| "#ESPIPE" [klabel(#ESPIPE), symbol]
| "#ESRCH" [klabel(#ESRCH), symbol]
| "#EXDEV" [klabel(#EXDEV), symbol]
| "#EWOULDBLOCK" [klabel(#EWOULDBLOCK), symbol]
| "#EINPROGRESS" [klabel(#EINPROGRESS), symbol]
| "#EALREADY" [klabel(#EALREADY), symbol]
| "#ENOTSOCK" [klabel(#ENOTSOCK), symbol]
| "#EDESTADDRREQ" [klabel(#EDESTADDRREQ), symbol]
| "#EMSGSIZE" [klabel(#EMSGSIZE), symbol]
| "#EPROTOTYPE" [klabel(#EPROTOTYPE), symbol]
| "#ENOPROTOOPT" [klabel(#ENOPROTOOPT), symbol]
| "#EPROTONOSUPPORT" [klabel(#EPROTONOSUPPORT), symbol]
| "#ESOCKTNOSUPPORT" [klabel(#ESOCKTNOSUPPORT), symbol]
| "#EOPNOTSUPP" [klabel(#EOPNOTSUPP), symbol]
| "#EPFNOSUPPORT" [klabel(#EPFNOSUPPORT), symbol]
| "#EAFNOSUPPORT" [klabel(#EAFNOSUPPORT), symbol]
| "#EADDRINUSE" [klabel(#EADDRINUSE), symbol]
| "#EADDRNOTAVAIL" [klabel(#EADDRNOTAVAIL), symbol]
| "#ENETDOWN" [klabel(#ENETDOWN), symbol]
| "#ENETUNREACH" [klabel(#ENETUNREACH), symbol]
| "#ENETRESET" [klabel(#ENETRESET), symbol]
| "#ECONNABORTED" [klabel(#ECONNABORTED), symbol]
| "#ECONNRESET" [klabel(#ECONNRESET), symbol]
| "#ENOBUFS" [klabel(#ENOBUFS), symbol]
| "#EISCONN" [klabel(#EISCONN), symbol]
| "#ENOTCONN" [klabel(#ENOTCONN), symbol]
| "#ESHUTDOWN" [klabel(#ESHUTDOWN), symbol]
| "#ETOOMANYREFS" [klabel(#ETOOMANYREFS), symbol]
| "#ETIMEDOUT" [klabel(#ETIMEDOUT), symbol]
| "#ECONNREFUSED" [klabel(#ECONNREFUSED), symbol]
| "#EHOSTDOWN" [klabel(#EHOSTDOWN), symbol]
| "#EHOSTUNREACH" [klabel(#EHOSTUNREACH), symbol]
| "#ELOOP" [klabel(#ELOOP), symbol]
| "#EOVERFLOW" [klabel(#EOVERFLOW), symbol]
```

### I/O result sorts

Here we see sorts defined to contain either an `Int` or an `IOError`, or either a `String` or an `IOError`. These sorts are used to implement the return sort of functions that may succeed, in which case they return a value, or may fail, in which case their return value indicates an error and the error indicated is returned via `errno`.

```
syntax IOInt ::= Int | IOError
syntax IOString ::= String | IOError
```

### Opening a file

You can open a file in K using `#open`. An optional mode indicates the file open mode, which can have any value allowed by the `fopen` function in C. The returned value is the file descriptor that was opened, or an error.

```
syntax IOInt ::= "#open" "(" path: String ")" [function]
               | "#open" "(" path: String "," mode: String ")" [function, hook(IO.open), imp

rule #open(S:String) => #open(S:String, "r+")
```

### Get/set position in file

You can get the current offset in a file using `#tell`. You can also seek to a particular offset using `#seek` or `#seekEnd`. `#seek` is implemented via a call to `lseek` with the `SEEK_SET` whence. `#seekEnd` is implemented via a call to `lseek` with the `SEEK_END` whence. You can emulate the `SEEK_CUR` whence by means of `#seek(FD, #tell(FD) +Int Offset)`.

```
syntax IOInt ::= "#tell" "(" fd: Int ")" [function, hook(IO.tell), impure]
syntax K ::= "#seek" "(" fd: Int "," index: Int ")" [function, hook(IO.seek), impure]
           | "#seekEnd" "(" fd: Int "," fromEnd: Int ")" [function, hook(IO.seekEnd), impu
```

### Read from file

You can read a single character from a file using `#getc`. `#EOF` is returned if you are at end-of-fie.

You can also read up to `length` characters in a file using `#read`. The resulting read characters are returned, which may be fewer characters than requested. A string of zero length being returned indicates end-of-file.

```
syntax IOInt ::= "#getc" "(" fd: Int ")"             [function, hook(IO.getc), impure]
syntax IOString ::= "#read" "(" fd: Int "," length: Int ")"    [function, hook(IO.read), i
```

### Write to file

You can write a single character to a file using `#putc`. You can also write a string to a file using `#write`. The returned value on success is `.K`.

```
syntax K ::= "#putc" "(" fd: Int "," value: Int ")"      [function, hook(IO.putc), impure]
           | "#write" "(" fd: Int "," value: String ")" [function, hook(IO.write), impure]
```

### Closing a file

You can close a file using `#close`. The returned value on success is `.K`.

```
syntax K ::= "#close" "(" fd: Int ")" [function, hook(IO.close), impure]
```

### Locking/unlocking a file

You can lock or unlock parts of a file using the `#lock` and `#unlock` functions. The lock starts at the beginning of the file and continues for `endIndex` bytes. Note that Unix systems do not actually prevent locked files from being read and modified; you will have to lock both sides of a concurrent access to guarantee exclusivity.

```
syntax K ::= "#lock" "(" fd: Int "," endIndex: Int ")" [function, hook(IO.lock), impure]
           | "#unlock" "(" fd: Int "," endIndex: Int ")" [function, hook(IO.unlock), impu
```

### Networking

You can accept a connection on a socket using `#accept`, or shut down the write end of a socket with `#shutdownWrite`. Note that facility is not provided for opening, binding, and listening on sockets. These functions are implemented in order to support creating stateful request/response servers where the request loop is implemented using rewriting in K, but the connection initialization is written in native code and linked into the LLVM backend.

```
syntax IOInt ::= "#accept" "(" fd: Int ")" [function, hook(IO.accept), impure]
syntax K ::= "#shutdownWrite" "(" fd: Int ")" [function, hook(IO.shutdownWrite), impure]
```

### Time

You can get the current time in seconds since midnight UTC on January 1, 1970 using `#time`.

```
syntax Int ::= "#time" "(" ")" [function, hook(IO.time), impure]
```

### Builtin file descriptors

Provided here are functions that return the file descriptor for standard input, standard output, and standard error.

```
syntax Int ::= "#stdin"   [function, functional]
             | "#stdout" [function, functional]
             | "#stderr" [function, functional]

rule #stdin => 0
rule #stdout => 1
rule #stderr => 2
```

### Shell access

You can execute a command using the shell using the `#system` operator. Care must be taken to sanitize inputs to this function or security issues may

result. Note that K has no facility for reasoning about logic that happens outside its process, so any functionality that you wish to be able to formally reason about in K should not be implemented via the `#system` operator.

```
syntax KItem ::= #system ( String ) [function, hook(IO.system), impure]
               | "#systemResult" "(" Int /* exit code */ "," String /* stdout */ "," Strin
```

### Temporary files

You can get a temporary file and open it atomically using the `#mkstemp` operator. The resulting file will be closed and deleted when K rewriting ends. For more info on the argument to `#mkstemp`, see `man mkstemp`.

```
syntax IOFile ::= #mkstemp(template: String) [function, hook(IO.mkstemp), impure]
syntax IOFile ::= IOError
                | "#tempFile" "(" path: String "," fd: Int ")" [klabel(#tempFile), symbol]
```

### Deleting a file

You can delete a file using its absolute or relative path using the `#remove` operator. It returns `.K` on success or an `IOError` on failure.

```
syntax K ::= #remove(path: String) [function, functional, hook(IO.remove), impure]
```

### Logging

You can log information to disk using the `#logToFile` operator. Semantically, this operator returns `.K`. However, it has a side effect that is not reasoned about which is that `value` will be written to a uniquely-identified file containing `name` in its name. The file is only flushed to disk when rewriting finishes.

```
syntax K ::= #logToFile(name: String, value: String) [function, functional, hook(IO.log),
```

Terms can also be logged to standard error in *surface syntax*, rather than as KORE using `#trace`. This operator has similar semantics to `#logToFile` (i.e. it returns `.K`, but prints as an impure side effect). Note that calling `#trace` is equivalent to invoking the `kprint` tool for the first term that is logged, which requires re-parsing the underlying K definition. Subsequent calls do not incur this overhead again; the definition is cached.

```
syntax K ::= #trace(value: KItem) [function, functional, hook(IO.traceTerm), impure, retu
           | #traceK(value: K)    [function, functional, hook(IO.traceTerm), impure, retu
```

### Implementation of high-level I/O streams in K

Below is an implementation of the `stream="stdin"` and `stream="stdout"` cell attributes in K. You should not refer to these symbols or modules directly in your definition. It is provided only so that the K compiler can make use of

it. For more information on how to use this feature, refer to IMP++ in the K tutorial.

```
  syntax Stream ::= #buffer(K)
                  | #istream(Int)
                  | #parseInput(String, String)
                  | #ostream(Int)

endmodule

// NOTE: DO NOT DIRECTLY IMPORT *-STREAM MODULES
// These stream modules will be automatically instantiated and implicitly imported
// into the main module when `stream` attributes appear in configuration cells.
// Only `Stream` productions and `[stream]` rules will be imported.
// The cell name will be replaced with the one of the main configuration.

module STDIN-STREAM
  imports K-IO
  imports K-REFLECTION
  imports LIST
  imports INT
  imports BOOL

  configuration <stdin> ListItem(#buffer($STDIN:String)) ListItem($IO:String) ListItem(#ist

  // read one character at a time until we read whitespace
  rule [stdinGetc]:
       <stdin>
       ListItem(#parseInput(_:String, Delimiters:String))
       ListItem(#buffer(S:String => S +String chrChar({#getc(N)}:>Int)))
       ListItem("on")
       ListItem(#istream(N:Int))
       </stdin>
    requires findChar(S, Delimiters, 0) ==Int -1 // [stdin]
       [stream, priority(200)]

  // when we reach whitespace, if it parses create a ListItem
  rule [stdinParseString]:
       <stdin>
       (ListItem(#parseInput("String", Delimiters:String)) => ListItem(S))
       ListItem(#buffer(S:String => ""))
       _:List
       </stdin>
    requires findChar(S, Delimiters, 0) =/=Int -1 // [stdin]
       [stream]
```

```
// a hack: handle the case when we read integers without the help of the IO server
rule [stdinParseInt]:
     <stdin>
     (ListItem(#parseInput("Int", Delimiters:String))
     => ListItem(String2Int(substrString(S, 0, findChar(S, Delimiters, 0)))))
     ListItem(#buffer(S:String => substrString(S,findChar(S, Delimiters, 0) +Int 1, lengt
     _:List
     </stdin>
  requires findChar(S, Delimiters, 0) =/=Int -1
     andBool lengthString(S) >Int 1 // [stdin]
     [stream]

rule [stdinParseArbitrarySort]:
     <stdin>
     (ListItem(#parseInput(Sort:String, Delimiters:String))
     => ListItem(#parseKAST(substrString(S, 0, findChar(S, Delimiters, 0)))))
     ListItem(#buffer(S:String => substrString(S,findChar(S, Delimiters, 0) +Int 1, lengt
     _:List
     </stdin>
  requires findChar(S, Delimiters, 0) =/=Int -1
     andBool Sort ==String "K"
     andBool lengthString(S) >Int 1 // [stdin]
     [stream]

rule [stdinTrim]:
     <stdin>
     ListItem(#parseInput(Sort:String, Delimiters:String))
     ListItem(#buffer(S:String => substrString(S, 1, lengthString(S))))
     _:List
     </stdin>
  requires findChar(S, Delimiters, 0) =/=Int -1
     andBool Sort =/=String "String"
     andBool lengthString(S) <=Int 1 // [stdin]
     [stream]

// NOTE: This unblocking rule will be instantiated and inserted carefully
// when necessary according to user-defined rules, since otherwise it will
// lead to a diverging (i.e., non-terminating) transition system definition.
// Currently, it supports only a simple pattern matching on the top of the
// input stream cell, e.g.,
//   rule <k> read() => V ... </k> <in> ListItem(V:Int) => .List ...  </in>
// Non-supported rules that refer to the input stream cell in a sophisticated
// way will get stuck in concrete execution mode with real IO enabled (i.e.,
// under `--io on` option), while they will still work in symbolic execution
// mode or concrete execution mode with real IO disabled (i.e., under `--io
// off`, `--search`, or `--debug` options).
```

```
  //
  // TODO: More patterns need to be supported as well. In that case, we need to
  // have a way to specify such patterns.
  rule [stdinUnblock]:
       <stdin>
         (.List => ListItem(#parseInput(?Sort:String, ?Delimiters:String)))
         ListItem(#buffer(_:String))
         ...
       </stdin>
       [unblock]

  /*
  syntax Stream ::= "#noIO"

  rule ListItem(#buffer(_))
       (ListItem(#noIO) ListItem(#istream(_:Int)) => .List) [stdin]
  */

endmodule

module STDOUT-STREAM
  imports K-IO
  imports LIST
  imports STRING

  configuration <stdout> ListItem(#ostream(#stdout)) ListItem($IO:String) ListItem(#buffer('
//configuration <stderr> ListItem(#ostream(#stderr)) ListItem($IO:String) ListItem(#buffer('

  rule [stdoutBufferFloat]:
       <stdout>
       ListItem(#ostream(_))
       ListItem(_)
       ListItem(#buffer(Buffer:String => Buffer +String Float2String(F)))
       (ListItem(F:Float) => .List)
       _:List
       </stdout>
       // [stdout, stderr]
       [stream, priority(25)]
  rule [stdoutBufferInt]:
       <stdout>
       ListItem(#ostream(_))
       ListItem(_)
       ListItem(#buffer(Buffer:String => Buffer +String Int2String(I)))
       (ListItem(I:Int) => .List)
       _:List
       </stdout>
```

```
      // [stdout, stderr]
      [stream, priority(25)]
  rule [stdoutBufferString]:
      <stdout>
      ListItem(#ostream(_))
      ListItem(_)
      ListItem(#buffer(Buffer:String => Buffer +String S))
      (ListItem(S:String) => .List)
      _:List
      </stdout>
      // [stdout, stderr]
      [stream, priority(25)]

  // Send first char from the buffer to the server
  rule [stdoutWrite]:
      <stdout>
      ListItem(#ostream(N:Int => {#write(N, S) ~> N:Int}:>Int))
      ListItem("on")
      ListItem(#buffer(S:String => ""))
      _:List
      </stdout>
    requires S =/=String "" // [stdout, stderr]
      [stream, priority(30)]

  /*
  syntax Stream ::= "#noIO"

  rule ListItem(#buffer(Buffer:String => Buffer +String Float2String(F)))
      (ListItem(F:Float) => .List)
      _:List [stdout, stderr]
  rule ListItem(#buffer(Buffer:String => Buffer +String Int2String(I)))
      (ListItem(I:Int) => .List)
      _:List [stdout, stderr]
  rule ListItem(#buffer(Buffer:String => Buffer +String S))
      (ListItem(S:String) => .List)
      _:List [stdout, stderr]

  rule (ListItem(#ostream(_:Int)) ListItem(#noIO) => .List)
      ListItem(#buffer(_))
      _:List [stdout, stderr]
  */

endmodule
```

## Machine Integers

Provided here is an implementation of arbitrarily large fixed-precision binary integers in K. This type is hooked to an implementation of integers provided by the backend, and in particular makes use of native machine integers for certain sizes of integer. For arbitrary-precision integers, see the `INT` module above.

The syntax of machine integers in K is the same as arbitrary-precision integers (i.e., an optional sign followed by a sequence of digits) except that machine integers always end in a suffix `pN` where `N` is an integer indicating the width in bits of the integer. The `MInt` sort is parametric, and this is reflected in the literals. For example, the sort of `0p8` is `MInt{8}`.

```
module MINT-SYNTAX
  /*@\section{Description} The MInt implements machine integers of arbitrary
   * bit width represented in 2's complement. */
  syntax {Width} MInt{Width} [hook(MINT.MInt)]

  /*@ Machine integer of bit width and value. */
  syntax {Width} MInt{Width} ::= r"[\\+-]?[0-9]+[pP][0-9]+" [token, prec(2), hook(MINT.liter
endmodule

module MINT
  imports MINT-SYNTAX
  imports private INT
  imports private BOOL
```

### Bitwidth of MInt

You can get the number of bits of width in an MInt using `bitwidthMInt`.

```
  syntax {Width} Int ::= bitwidthMInt(MInt{Width})   [function, functional, hook(MINT.bitwid
```

### Int and MInt conversions

You can convert from an `MInt` to an `Int` using the `MInt2Signed` and `MInt2Unsigned` functions. an `MInt` does not have a sign; its sign is instead reflected in how operators interpret its value either as a signed integer or as an unsigned integer. Thus, you can interpret a `MInt` as a signed integer witth `MInt2Signed`, or as an unsigned integer respectively using `MInt2Unsigned`.

You can also convert from an `Int` to an `MInt` using `Int2MInt`. Care must be given to ensure that the sort context where the `Int2MInt` operator appears has the correct bitwidth, as this will influence the width of the resulting `MInt`.

```
  syntax {Width} Int ::= MInt2Signed(MInt{Width})     [function, functional, hook(MINT.svalu
                       | MInt2Unsigned(MInt{Width})     [function, functional, hook(MINT.uva
```

```
syntax {Width} MInt{Width} ::= Int2MInt(Int) [function, functional, hook(MINT.integer), sm
```

### MInt min and max values

You can get the minimum and maximum values of a signed or unsigned `MInt` with az specified bit width using `sminMInt`, `smaxMInt`, `uminMInt`, and `umaxMInt`.

```
syntax Int ::= sminMInt(Int)    [function]
             | smaxMInt(Int)    [function]
             | uminMInt(Int)    [function]
             | umaxMInt(Int)    [function]
rule sminMInt(N:Int) => 0 -Int (1 <<Int (N -Int 1))
rule smaxMInt(N:Int) => (1 <<Int (N -Int 1)) -Int 1
rule uminMInt(_:Int) => 0
rule umaxMInt(N:Int) => (1 <<Int N) -Int 1
```

### MInt bounds checking

You can check whether a specified `Int` will be represented in an `MInt` with a specified `width` without any loss of precision when interpreted as a signed or unsigned integer using `soverflowMInt` and `uoverflowMInt`.

```
syntax Bool ::= soverflowMInt(width: Int, Int)   [function]
              | uoverflowMInt(width: Int, Int)   [function]
rule
  soverflowMInt(N:Int, I:Int)
=>
  I <Int sminMInt(N) orBool I >Int smaxMInt(N)
rule
  uoverflowMInt(N:Int, I:Int)
=>
  I <Int uminMInt(N) orBool I >Int umaxMInt(N)
```

### MInt arithmetic

You can:

- Compute the bitwise complement `~MInt` of an `MInt`.
- Compute the unary negation `--MInt` of an `MInt`.
- Compute the product `*MInt` of two `MInt`s.
- Compute the quotient `/sMInt` of two `MInt`s interpreted as signed integers.
- Compute the modulus `%sMInt` of two `MInt`s interpreted as signed integers.
- Compute the quotient `/uMInt` of two `MInt`s interpreted as unsigned integers.
- Compute the modulus `%uMInt` of two `MInt`s interpreted as unsigned integers.

- Compute the sum `+MInt` of two `MInt`s.
- Compute the difference `-MInt` of two `MInt`s.
- Compute the left shift `<<MInt` of two `MInt`s. The second `MInt` is always interpreted as positive.
- Compute the arithmetic right shift `>>aMInt` of two `MInt`s. The second `MInt` is always interpreted as positve.
- Compute the logical right shift `>>lMInt` of two `MInt`s. The second `MInt` is always interpreted as positive.
- Compute the bitwise and `&MInt` of two `MInt`s.
- Compute the bitwise xor `xorMInt` of two `MInt`s.
- Compute the bitwise inclusive or `|MInt` of two `MInt`s.

```
syntax {Width} MInt{Width} ::= "~MInt" MInt{Width} [function, functional, hook(MINT.not),
                             | "--MInt" MInt{Width} [function, functional, hook(MINT.neg),
                             > left:
                               MInt{Width} "*MInt" MInt{Width} [function, functional, hook
                             | MInt{Width} "/sMInt" MInt{Width} [function, hook(MINT.sdiv)
                             | MInt{Width} "%sMInt" MInt{Width} [function, hook(MINT.srem)
                             | MInt{Width} "/uMInt" MInt{Width} [function, hook(MINT.udiv)
                             | MInt{Width} "%uMInt" MInt{Width} [function, hook(MINT.urem)
                             > left:
                               MInt{Width} "+MInt" MInt{Width} [function, functional, hook
                             | MInt{Width} "-MInt" MInt{Width} [function, functional, hook
                             > left:
                               MInt{Width} "<<MInt" MInt{Width} [function, hook(MINT.shl),
                             | MInt{Width} ">>aMInt" MInt{Width} [function, hook(MINT.ashr
                             | MInt{Width} ">>lMInt" MInt{Width} [function, hook(MINT.lshr
                             > left:
                               MInt{Width} "&MInt" MInt{Width} [function, functional, hook
                             > left:
                               MInt{Width} "xorMInt" MInt{Width} [function, functional, ho
                             > left:
                               MInt{Width} "|MInt" MInt{Width} [function, functional, hook
```

### MInt comparison

You can compute whether one `MInt` is less than, less than or equal to, greater than, or greater than or equal to another `MInt` when interpreted as signed or unsigned integers. You can also compute whether one `MInt` is equal to or unequal to another `MInt`.

```
syntax {Width} Bool ::= MInt{Width} "<sMInt" MInt{Width} [function, functional, hook(MINT.
                      | MInt{Width} "<uMInt" MInt{Width} [function, functional, hook(MINT.
                      | MInt{Width} "<=sMInt" MInt{Width} [function, functional, hook(MINT
                      | MInt{Width} "<=uMInt" MInt{Width} [function, functional, hook(MINT
                      | MInt{Width} ">sMInt" MInt{Width} [function, functional, hook(MINT.
                      | MInt{Width} ">uMInt" MInt{Width} [function, functional, hook(MINT.
```

```
                              | MInt{Width} ">=sMInt" MInt{Width} [function, functional, hook(MINT
                              | MInt{Width} ">=uMInt" MInt{Width} [function, functional, hook(MINT
                              | MInt{Width} "==MInt" MInt{Width} [function, functional, hook(MINT.
                              | MInt{Width} "=/=MInt" MInt{Width} [function, functional, hook(MINT
```

#### MInt min/max

You can compute the signed minimum `sMinMInt`, the signed maximum `sMaxMInt`,
the unsigned minimum `uMinMInt`, and the unsigned maximum `uMaxMInt` of two `MInt`s.

```
  syntax {Width} MInt{Width} ::= sMaxMInt(MInt{Width}, MInt{Width}) [function, functional, h
                                 | sMinMInt(MInt{Width}, MInt{Width}) [function, functional, h
                                 | uMaxMInt(MInt{Width}, MInt{Width}) [function, functional, h
                                 | uMinMInt(MInt{Width}, MInt{Width}) [function, functional, h
```

#### MInt to MInt conversion

You can convert an `MInt` of one width to another width with `roundMInt`.
The resulting `MInt` will be truncated starting from the most significant bit if the resulting width is smaller than the input. The resulting `MInt` will be zero-extended with the same low-order bits if the resulting width is larger than the input.

```
  syntax {Width1, Width2} MInt{Width1} ::= roundMInt(MInt{Width2}) [function, functional, ho
```

```
endmodule
```

## Strategies

K supports a built-in strategy language that allows you to control how rules apply. In order to enable it, simply import the `STRATEGY` module in your definition. This includes the following basic strategy constructs:

- `^ Category:#RuleTag` - This is a strategy that indicates that you should apply a rule with the specified category exactly once. By default, all rules get the `regular` tag. This can be changed on individual rules with the `tag`
  attribute.
- `~ Category:#RuleTag` - This is the state the strategy cell will be in after a rule has applied. In other words, all rules that do not mention the strategy cell are automatically instrumented so that they rewrite the current
  top of the `<s>` cell from `^` to `~`
- `<s>` - This is the strategy cell and contains the current strategy.
- `#STUCK()` - By default, a rule is automatically inserted into the definition which adds `#STUCK()` to the top of the `<s>` cell if no other rules apply

and if it is not already at the top of the <s> cell.

```
module STRATEGY
    imports ML-SYNTAX
    imports KVARIABLE-SYNTAX
    imports private K-EQUAL

    syntax #RuleTag ::= #KVariable

    syntax Strategy ::= #STUCK()    [symbol]
                      | "^" #RuleTag [symbol, klabel(#applyRule)]
                      | "~" #RuleTag [symbol, klabel(#appliedRule)]

    configuration <s> $STRATEGY:K </s>
endmodule

module RULE-TAG-SYNTAX
    imports BUILTIN-ID-TOKENS

    syntax #RuleTag ::= #LowerId [token]
endmodule
```

This is not a complete strategy language. However, it provides several basics. The user can extend this strategy language into a complete strategy language in one of three ways. First, they can import the `DEFAULT-STRATEGY` module, which provides a very basic strategy that is essentially equivalent to execution without a strategy. This can be useful if you wish to wholly manipulate the strategy cell yourself within other rules.

```
module DEFAULT-STRATEGY-CONCRETE [concrete]
    imports STRATEGY
    imports RULE-TAG-SYNTAX
    rule ~ regular => ^ regular [anywhere]
endmodule

module DEFAULT-STRATEGY-SYMBOLIC [symbolic]
    imports STRATEGY
    imports RULE-TAG-SYNTAX
    rule <s> ~ regular => ^ regular ... </s>
endmodule

module DEFAULT-STRATEGY
    imports STRATEGY
    imports DEFAULT-STRATEGY-CONCRETE
    imports DEFAULT-STRATEGY-SYMBOLIC

    rule initSCell(_) => <s> ^ regular </s> [priority(25)]
```

```
endmodule
```

The second way you can extend the strategy language is with the `STRATEGY-ABSTRACT` module. This provides a slightly more advanced set of strategies which you can use to compose the basic strategies in the `STRATEGY` module into more complex strategies. Note however that the functionality provided is still relatively basic.

```
module STRATEGY-ABSTRACT
    imports STRATEGY

    syntax #RuleTag ::= "(" #RuleTag ")" [bracket]
 // ---------------------------------------------

    syntax Strategy ::= ".Strategy"
                      | "(" Strategy ")" [bracket]
 // ---------------------------------------------
    rule <s> .Strategy => . ... </s>

    syntax KItem ::= #catchSTUCK ( Strategy )
 // ----------------------------------------
    rule <s> #catchSTUCK(_) => . ... </s>
    rule <s> #STUCK() ~> (_S:Strategy => .) ... </s>
    rule <s> #STUCK() ~> #catchSTUCK(S) => S ... </s>

    syntax Strategy ::= Strategy ";" Strategy [left]
 // -----------------------------------------------
    rule <s> S:Strategy ; S':Strategy => S ~> S' ... </s>

    syntax #RuleTag ::= #RuleTag "|" #RuleTag [left, klabel(#alternateRule)]
 // ------------------------------------------------------------------------
    rule <s> ^ RT:#RuleTag | RT':#RuleTag => ^ RT ~> #catchSTUCK(^ RT') ... </s>

    syntax #RuleTag ::= #RuleTag "*" [klabel(#repeatRule)]
 // -----------------------------------------------------
    rule <s> ^ RT:#RuleTag * => ^ RT ~> ^ RT * ~> #catchSTUCK(.Strategy) ... </s>
endmodule
```

The third mechanism for extending the strategy language is to define your own strategy language. No special facility is required in K in order to compose strategies together, so you are free to write whatever rules that manipulate the strategy cell you want, and thus whatever more complex strategies you can compose from the base builtin strategies.

# K Language Features

Defined below is a series of modules used to parse inner syntax in K (ie, the contents of rules, configuration declarations, and contexts).

Much of this file exists in tight correspondence with the K implementation, and K will not work correctly if it is altered without corresponding changes to the source code of the K tools.

Users should only import a few modules from this file. In particular, this includes `SORT-K`, `BASIC-K`, `ML-SYNTAX`, `DEFAULT-LAYOUT`, `DEFAULT-CONFIGURATION`, and `K-AMBIGUITIES`. The remaining modules should not
be imported by the user; they are used implicitly by the implementation of K.

## Basic K Sorts

The `SORT-K` module declares the `K` sort, and nothing else.

```
module SORT-K
  syntax K [hook(K.K)]
endmodule
```

The `BASIC-K` module declares the `K`, `KItem`, and `KConfigVar` sorts, and imports the syntax of matching logic.

```
module BASIC-K
  imports ML-SYNTAX
  imports SORT-BOOL
  syntax KItem [hook(K.KItem)]
  syntax K     ::= KItem
  syntax KConfigVar [token]
endmodule
```

## KAST Abstract Syntax

Below is defined the abstract syntax of concrete terms in K, the `KAST` syntax. Users should rarely if ever have to refer to this syntax; in general, it suffices to use concrete syntax in rules, configuration declarations, contexts, etc.

This syntax is used directly by the K implementation, and exists here as a reference for the syntax of `KAST`, but it should not be imported directly by the user.

```
module KSTRING
  syntax KString ::= r"[\\\"](([^\\\"\\n\\r\\\\])|([\\\\][nrtf\\\"\\\\])|([\\\\][x][0-9a-fA-
  // optionally qualified strings, like in Scala "abc", i"abc", r"a*bc", etc.
endmodule
```

```
module BUILTIN-ID-TOKENS
  syntax #LowerId ::= r"[a-z][a-zA-Z0-9]*"                    [prec(2), token]
  syntax #UpperId ::= r"[A-Z][a-zA-Z0-9]*"                    [prec(2), token]
endmodule

module SORT-KBOTT
  imports SORT-K
  syntax KBott
endmodule

module KAST
  imports BASIC-K
  imports SORT-KBOTT
  imports KSTRING
  imports BUILTIN-ID-TOKENS

  syntax KBott ::= "#token" "(" KString "," KString ")"  [klabel(#KToken), symbol]
                 | "#klabel" "(" KLabel ")"              [klabel(#WrappedKLabel), symbol]
                 | KLabel "(" KList ")"                  [klabel(#KApply), symbol]
  syntax KItem ::= KBott

  syntax KLabel ::= r"`(\\\\`|\\\\\\\\\|[^`\\\\\\n\\r])+`" [token]
                  | #LowerId                                    [token]
                  | r"(?<![a-zA-Z0-9])[#a-z][a-zA-Z0-9]*"               [token, prec(1)]
                       // something that doesn't collide with meta-variables

  syntax KList ::= K
                 | ".KList"          [klabel(#EmptyKList), symbol]
                 | ".::KList"        [klabel(#EmptyKList), symbol]
                 | KList "," KList   [klabel(#KList), left, assoc, unit(#EmptyKList), symbol
endmodule


// To be used when parsing/pretty-printing ground configurations
module KSEQ
  imports KAST
  imports K-TOP-SORT
  syntax K ::= ".K"      [klabel(#EmptyK), symbol, unparseAvoid]
             | "."       [klabel(#EmptyK), symbol]
             | ".::K"    [klabel(#EmptyK), symbol, unparseAvoid]
  syntax K ::= K "~>" K  [klabel(#KSequence), left, assoc, unit(#EmptyK), symbol]
  syntax left #KSequence
  syntax {Sort} Sort     ::= "(" Sort ")"    [bracket, defaultBracket, applyPriority(1)]
endmodule
```

## Syntax of Matching Logic

K provides direct access to the symbols of Matching Logic, while giving them their own concrete syntax distinct from the syntax of the `KORE` intermediate representation. These symbols are primarily used during symbolic execution. The LLVM Backend has relatively little understanding of Matching Logic directly
and use of these symbols directly in rules is likely to cause it to crash. However, these symbols are necessary when providing lemmas and other types of
logical assistance to proofs and symbolic execution in the Haskell Backend.

The correspondance between K symbols and KORE symbols is as follows:

- `#Top` - `\top`
- `#Bottom` - `\bottom`
- `#Not` - `\not`
- `#Ceil` - `\ceil`
- `#Floor` - `\floor`
- `#Equals` - `\equals`
- `#And` - `\and`
- `#Or` - `\or`
- `#Implies` - `\implies`
- `#Exists` - `\exists`
- `#Forall` - `\forall`
- `#AG` - `allPathGlobally`
- `#wEF` - `weakExistsFinally`
- `#wAF` - `weakAlwaysFinally`

```
module ML-SYNTAX [not-lr1]
  imports SORT-K

  syntax {Sort} Sort ::= "#Top" [klabel(#Top), symbol, mlUnary]
                       | "#Bottom" [klabel(#Bottom), symbol, mlUnary]
                       | "#True" [klabel(#Top), symbol, mlUnary, unparseAvoid]
                       | "#False" [klabel(#Bottom), symbol, mlUnary, unparseAvoid]
                       | "#Not" "(" Sort ")" [klabel(#Not), symbol, mlOp, mlUnary]

  syntax {Sort1, Sort2} Sort2 ::= "#Ceil" "(" Sort1 ")" [klabel(#Ceil), symbol, mlOp, mlUnar
                                | "#Floor" "(" Sort1 ")" [klabel(#Floor), symbol, mlOp, mlU
                                | "{" Sort1 "#Equals" Sort1 "}" [klabel(#Equals), symbol, ml

  syntax priorities mlUnary > mlEquals > mlAnd

  syntax {Sort} Sort ::= Sort "#And" Sort [klabel(#And), symbol, assoc, left, comm, unit(#To
                       > Sort "#Or" Sort [klabel(#Or), symbol, assoc, left, comm, unit(#Bott
                       > Sort "#Implies" Sort [klabel(#Implies), symbol, mlOp, mlImplies, fo
```

```
  syntax priorities mlImplies > mlQuantifier

  syntax {Sort1, Sort2} Sort2 ::= "#Exists" Sort1 "." Sort2 [klabel(#Exists), symbol, mlOp,
                                | "#Forall" Sort1 "." Sort2 [klabel(#Forall), symbol, mlOp,

  syntax {Sort} Sort ::= "#AG" "(" Sort ")" [klabel(#AG), symbol, mlOp]
                       | "#wEF" "(" Sort ")" [klabel(weakExistsFinally), symbol, mlOp]
                       | "#wAF" "(" Sort ")" [klabel(weakAlwaysFinally), symbol, mlOp]
endmodule
```

## Variables in K

Provided below is the syntax of variables in K. There are four types of variables in K:

1. Regular variables. These are denoted by variables that begin with an underscore or a capital letter. These variables match exactly one value and can be used to refer to it on the right-hand-side.
2. Fresh constants. These are denoted by variables that begin with an `!`. This
   is a convenience syntax which can be used on the right-hand-side only, and
   refer to a unique value of the specified sort which is distinct from any other value that has been generated or will be generated by the `!X` syntax. Note that this may not be distinct from values produced via other means.
3. Existential variables. This refers to variables that are existentially quantified and begin with a `?`. They are not required to appear on the left-hand-side prior to appearing on the right-hand-side, and generally refer to symbolic quantities that are introduced during rewriting. Refer to
   K's documentation for more details.
4. Set variables. These are denoted by variables that begin with a `@`. These variables refer to a set of values and are generally used when writing simplification rules in the Haskell Backend. For more information, refer to
   K's documentation.

There is also a fifth type of "variable", although it is not technically a variable. This refers to configuration variables, which are used to insert values into the initial configuration that come from outside the semantics. The most common of these is the `$PGM` variable, which conventionally contains the program being executed and is placed in the `<k>` cell in the configuration declaration. These "variables" begin with a `$` and their values are populated by the frontend prior to symbolic or concrete execution of a program.

```
module KVARIABLE-SYNTAX
```

```
  syntax #KVariable
endmodule

// To be used when parsing/pretty-printing symbolic configurations
module KSEQ-SYMBOLIC
  imports KSEQ
  imports ML-SYNTAX
  imports KVARIABLE-SYNTAX

  syntax #KVariable ::= r"(?<![A-Za-z0-9_\\$!\\?@])(\\!|\\?|@)?([A-Z][A-Za-z0-9'_]*|_|_[A-Z]
                      | #UpperId                                                              [t
  syntax KConfigVar ::= r"(?<![A-Za-z0-9_\\$!\\?@])(\\$)([A-Z][A-Za-z0-9'_]*)"               [t
  syntax KBott      ::= #KVariable
  syntax KBott      ::= KConfigVar
  syntax KLabel     ::= #KVariable
endmodule
```

## Syntax of Cells

While the backend treats cells as regular productions like any other, the frontend provides a significant amount of convenience notation for dealing with groups of cells, in order to make writing modular definitions easier. As a result, we need a syntax for groups of cells and for referring to cells within rules, configuration declarations, and functions.

For historical reasons, the `Bag` sort is used to refer to groups of cells. This may change in a future release. Users can combine cells in any order by concatenating them together, and can refer to the absence of any cells with the `.Bag` symbol. You can also refer to cells within a function by placing the cell context symbol, `[[ K ]]` at the top of a rule, placing a function symbol inside, and referring to cells afterwards. This implicitly inserts a reference to the configuration at the time prior to the currently-applied rule being applied which can be matched on within the function. Functions with such context cannot be referred to in the initial configuration, because the prior configuration does not yet exist.

```
module KCELLS
  imports KAST

  syntax Cell
  syntax Bag ::= Bag Bag  [left, assoc, klabel(#cells), symbol, unit(#cells)]
               | ".Bag"   [klabel(#cells), symbol]
               | ".::Bag" [klabel(#cells), symbol]
               | Cell
  syntax Bag ::= "(" Bag ")" [bracket]
  syntax KItem ::= Bag
  syntax #RuleBody ::= "[" "[" K "]" "]" Bag    [klabel(#withConfig), symbol, avoid]
```

```
  syntax non-assoc #withConfig
  syntax Bag ::= KBott
endmodule
```

Users can also refer to cells in rules. When doing so, an optional `...` can be placed immediately after the start of the cell or immediately before the end. In a cell whose contents are commutative, these are equivalent to one another and are also equivalent to placing `...` in both places. This means that what is placed in the cell will be combined with the cell contents' concatenation operator with an unnamed variable. In other words, you match on
some number of elements in the collection and do not care about the rest of the collection.

In a cell whose contents are not commutative, the `...` operators correspond to a variable on the respective side of the contents of the cell that the `...` appears. For example, `<foo>... L </foo>`, if `L` is a list, means some number of elements followed by L. Note that not all combinations are supported. Cells whose contents are sort `K` can only have `...` appear at the tail of the cell, and cells whose contents are sort `List` can only have `...` appear on at most one side in a single rule.

```
module RULE-CELLS
  imports KCELLS
  imports RULE-LISTS
  // if this module is imported, the parser automatically
  // generates, for all productions that have the attribute 'cell' or 'maincell',
  // a production like below:
  //syntax Cell ::= "<top>" #OptionalDots K #OptionalDots "</top>" [klabel(<top>)]

  syntax #OptionalDots ::= "..." [klabel(#dots), symbol]
                         | ""    [klabel(#noDots), symbol]
endmodule
```

Users can also declare cells in a configuration declaration. This generates a specific set of productions that is used internally to implement the cell. The most important of these is the cell itself, and attributes on this production can be specified in an xml-attribute-like syntax.

You can also use an xml-short-tag-like syntax to compose configuration cells together which were defined in different modules. However, it is a requirement that any K definition have at most one fully-composed configuration; thus, all other configuration declarations must appear composed within another configuration declaration.

```
module CONFIG-CELLS
  imports KCELLS
  imports RULE-LISTS
  syntax #CellName ::= r"[a-zA-Z][a-zA-Z0-9\\-]*"  [token, prec(1)]
```

```
                    | #LowerId            [token]
                    | #UpperId            [token]

  syntax Cell ::= "<" #CellName #CellProperties ">" K "</" #CellName ">" [klabel(#configCell
  syntax Cell ::= "<" #CellName "/>" [klabel(#externalCell), symbol]

  syntax #CellProperties ::= #CellProperty #CellProperties [klabel(#cellPropertyList), symbo
                           | ""                            [klabel(#cellPropertyListTerminat
  syntax #CellProperty ::= #CellName "=" KString           [klabel(#cellProperty), symbol]
endmodule
```

## Syntax of Rules

Rules can have an optional requires clause or an ensures clause. For backwards-compatibility, you can refer to the requires clause with both the `requires` and `when` keywords; The latter, however, is deprecated and may be removed in a future release.

The requires clause specifies the preconditions that must be true in order for the rule to apply. The ensures clause specifies the information which becomes true after the rule has applied. It is a requirement that information present in the `ensures` clause refer to existential variables only.

When doing concrete execution, you can think of the `requires` clause as a side-condition. In other words, even if the rule matches, it will not apply unless the `requires` clause, which must be of sort `Bool`, evaluates to `true`.

```
module REQUIRES-ENSURES
  imports BASIC-K

  syntax #RuleBody ::= K

  syntax #RuleContent ::= #RuleBody                                [klabel("#ruleNoConditio
                        | #RuleBody "requires" Bool                [klabel("#ruleRequires")
                        | #RuleBody "when" Bool                    [klabel("#ruleRequires")
                        | #RuleBody "ensures"  Bool                [klabel("#ruleEnsures"),
                        | #RuleBody "requires" Bool "ensures" Bool  [klabel("#ruleRequiresEn
                        | #RuleBody "when" Bool "ensures" Bool      [klabel("#ruleRequiresEn
endmodule
```

## Miscellaneous modules

The below modules are used in various ways as indicators to the implementation that certain automatically generated syntax should be created by the parser. These modules should not be imported directly by the user.

```
module K-TOP-SORT
  imports SORT-KBOTT
  syntax KItem ::= KBott
  syntax {Sort} KItem ::= Sort
endmodule

module K-BOTTOM-SORT
  imports SORT-KBOTT
  syntax KItem ::= KBott
  syntax {Sort} Sort ::= KBott
endmodule

module K-SORT-LATTICE
  imports K-TOP-SORT
  imports K-BOTTOM-SORT
endmodule

module AUTO-CASTS
  // if this module is imported, the parser automatically
  // generates, for all sorts, productions of the form:
  // Sort  ::= Sort ":Sort"  // semantic cast - force the inner term to be `Sort` or a subso
  // Sort  ::= Sort "::Sort" // strict cast - force the inner term to be exactly `Sort`. Use
  // Sort ::= "{" Sort "}" "<:Sort" // synonym for strict cast
  // Sort  ::= "{" K "}"    ":>Sort" // projection cast. Allows any term to be placed in a c
  // this is part of the mechanism that allows concrete user syntax in K
endmodule

module AUTO-FOLLOW
  // if this module is imported, the parser automatically
  // generates a follow restriction for every terminal which is a prefix
  // of another terminal. This is useful to prevent ambiguities such as:
  // syntax K ::= "a"
  // syntax K ::= "b"
  // syntax K ::= "ab"
  // syntax K ::= K K
  // #parse("ab", "K")
  // In the above example, the terminal "a" is not allowed to be followed by a "b"
  // because it would turn the terminal into the terminal "ab".
endmodule

module PROGRAM-LISTS
  imports SORT-K
  // if this module is imported, the parser automatically
  // replaces the default productions for lists:
  // Es ::= E "," Es [userList("*"), klabel('_,_)]
  //      | ".Es"    [userList("*"), klabel('.Es)]
```

```
  // into a series of productions more suitable for programs:
  // Es#Terminator ::= ""      [klabel('.Es)]
  // Ne#Es ::= E "," Ne#Es     [klabel('_,_)]
  //         | E Es#Terminator [klabel('_,_)]
  // Es ::= Ne#Es
  //      | Es#Terminator      // if the list is *
endmodule

module RULE-LISTS
  // if this module is imported, the parser automatically
  // adds the subsort production to the parsing module only:
  // Es ::= E        [userList("*")]

endmodule

module RECORD-PRODUCTIONS
  // if this module is imported, prefix productions of the form
  // syntax Sort ::= name(Args)
  // will be able to be parsed with don't-care variables according
  // to their nonterminal's names
endmodule

module SORT-PREDICATES
  // if this module is imported, the Bool sort will be annotated with
  // syntax Bool ::= isSort(K) [function]
  // and all sorts will be annotated with
  // syntax Sort ::= project:Sort(K) [function]
endmodule
```

## Additional Syntax for K Terms in Rules

Certain additional features are available when parsing the contents of rules and contexts. For more information on each of these, refer to K's documentation.

```
module KREWRITE
  syntax {Sort} Sort ::= Sort "=>" Sort [klabel(#KRewrite), symbol]
  syntax non-assoc #KRewrite
  syntax priority #KRewrite > #withConfig
endmodule

// To be used to parse semantic rules
module K
  imports KSEQ-SYMBOLIC
  imports REQUIRES-ENSURES
  imports RECORD-PRODUCTIONS
```

```
  imports SORT-PREDICATES
  imports K-SORT-LATTICE
  imports AUTO-CASTS
  imports AUTO-FOLLOW
  imports KREWRITE

  syntax {Sort} Sort ::= Sort "#as" Sort [klabel(#KAs), symbol]
  // functions that preserve sorts and can therefore have inner rewrites
  syntax {Sort} Sort ::= "#fun" "(" Sort ")" "(" Sort ")" [klabel(#fun2), symbol, prefer]
  // functions that do not preserve sort and therefore cannot have inner rewrites
  syntax {Sort1, Sort2} Sort1 ::= "#fun" "(" Sort2 "=>" Sort1 ")" "(" Sort2 ")" [klabel(#fun

  syntax {Sort1, Sort2} Sort1 ::= "#let" Sort2 "=" Sort2 "#in" Sort1 [klabel(#let), symbol]

  /*@ Set membership over terms. In addition to equality over
      concrete patterns, K also supports computing equality
      between a concrete pattern and a symbolic pattern.
      This is compiled efficiently down to pattern matching,
      and can be used by putting a term with unbound variables
      in the left child of :=K or =/=K. Note that this does not
      bind variables used on the lhs however (although this may
      change in the future).*/

  syntax Bool ::= left:
                  K ":=K" K           [function, functional, klabel(_:=K_), symbol, equalEqu
                | K ":/=K" K          [function, functional, klabel(_:/=K_), symbol, notEqua
endmodule

// To be used to parse terms in full K
module K-TERM
  imports KSEQ-SYMBOLIC
  imports RECORD-PRODUCTIONS
  imports SORT-PREDICATES
  imports K-SORT-LATTICE
  imports AUTO-CASTS
  imports AUTO-FOLLOW
  imports KREWRITE
endmodule
```

## Layout Information

When constructing a scanner for use during parsing, often you wish to ignore certain types of text, such as whitespace and comments. However, the specific syntax which each language must ignore is a little different from language to language, and thus you wish to specify it manually. You can do this by defining productions of the `#Layout` sort. For more information, refer to

K's documentation. However, this module will be implicitly imported if no productions are declared of sort `#Layout`. This module will also be used for the purposes of parsing K rules. If you wish to declare a language with no layout productions, simply create a sort declaration for the `#Layout` sort in your code (e.g. `syntax #Layout`).

```
module DEFAULT-LAYOUT
    syntax #Layout ::= r"(\\/\\*([^\\*]|(\\*+([^\\*\\/])))*\\*+\\/)" // C-style multi-line c
                     | r"(\\/\\/[^\\n\\r]*)"                         // C-style single-line
                     | r"([\\ \\n\\r\\t])"                           // Whitespace
endmodule
```

## Default Configuration

If the user has no configuration declaration in their seamantics, the below configuration declaration will be implicitly imported.

```
module DEFAULT-CONFIGURATION
  imports BASIC-K

  configuration <k> $PGM:K </k>
endmodule
```

## Parsing Ambiguous Languages

On occasion, it may be desirable to parse a language with an ambiguous grammar
when parsing a program, and perform additional semantic analysis at a later time in order to resolve the ambiguities. A good example of this is as a substitute for the lexer hack in parsers of the `C` programming language.

The following module contains a declaration for ambiguities in K. Usually, an ambiguous parse is an error. However, when you use the `--gen-glr-parser` flag to `kast`, or the `--gen-glr-bison-parser` flag to `kompile`, ambiguities instead become instances of the below parametric production, which you can use
regular K rules to disambiguate as necessary.

```
module K-AMBIGUITIES

  syntax {Sort} Sort ::= amb(Sort, Sort) [symbol]

endmodule
```

## Annotating Parses with Locations

Another feature of K's Bison parser is the ability to annotate terms parsed with location information about the file and line where they occurred. For

more information about how to use this, refer to K's documentation. However, the below module exists to provide a user syntax for the annotations that are generated by the parser.

```
module K-LOCATIONS
  imports STRING-SYNTAX
  imports INT-SYNTAX

  // filename, startLine, startCol, endLine, endCol
  syntax {Sort} Sort ::= #location(Sort, String, Int, Int, Int, Int) [symbol, format(%3)]

endmodule
```

# K Prelude

The following files, integral to defining semantics in K, are automatically required by every definition via this file. This behavior can be disabled via `kompile --no-prelude`, however, semantics will likely break unless they provide their own versions of these files, which are assumed to exist by the compiler. There are, however, circumstances where passing this flag is appropriate, such as if you are manually requiring these files in your definition, if your definition was automatically condensed into a single file with `kompile -E`, or if you wish to modify the inner syntax of K by providing your own version of these files with different syntax.

```
require "kast.md"
require "domains.md"
```

# K Foreign Function Interface

The K Foreign Function Interface (FFI) module provides a way to call native functions directly from a K semantics using the C ABI. It also provides utilities for allocating and deallocating byte buffers with static addresses that are suitable for being passed to native code.

It is built off of the underlying libffi library (https://sourceware.org/libffi/) and is subject to some of the same limitations as that library. Bear in mind, because this library exposes a number of unsafe C APIs directly, misuse of the library is likely to lead to memory corruption in your interpreter and can cause segmentation faults or corrupted term representations that lead to undefined behavior at runtime.

```
require "domains.md"

module FFI-SYNTAX
  imports private LIST
```

The FFIType sort is used to declare the native C ABI types of operands passed to the `#ffiCall` function. These types roughly correspond to the types declared in `ffi.h` by libffi.

```
  syntax FFIType ::= "#void" [klabel(#ffi_void), symbol]
                   | "#uint8" [klabel(#ffi_uint8), symbol]
                   | "#sint8" [klabel(#ffi_sint8), symbol]
                   | "#uint16" [klabel(#ffi_uint16), symbol]
                   | "#sint16" [klabel(#ffi_sint16), symbol]
                   | "#uint32" [klabel(#ffi_uint32), symbol]
                   | "#sint32" [klabel(#ffi_sint32), symbol]
                   | "#uint64" [klabel(#ffi_uint64), symbol]
                   | "#sint64" [klabel(#ffi_sint64), symbol]
                   | "#float" [klabel(#ffi_float), symbol]
                   | "#double" [klabel(#ffi_double), symbol]
                   | "#uchar" [klabel(#ffi_uchar), symbol]
                   | "#schar" [klabel(#ffi_schar), symbol]
                   | "#ushort" [klabel(#ffi_ushort), symbol]
                   | "#sshort" [klabel(#ffi_sshort), symbol]
                   | "#uint" [klabel(#ffi_uint), symbol]
                   | "#sint" [klabel(#ffi_sint), symbol]
                   | "#ulong" [klabel(#ffi_ulong), symbol]
                   | "#slong" [klabel(#ffi_slong), symbol]
                   | "#longdouble" [klabel(#ffi_longdouble), symbol]
                   | "#pointer" [klabel(#ffi_pointer), symbol]
                   | "#complexfloat" [klabel(#ffi_complexfloat), symbol]
                   | "#complexdouble" [klabel(#ffi_complexdouble), symbol]
                   | "#complexlongdouble" [klabel(#ffi_complexlongdouble), symbol]
                   | "#struct" "(" List ")" [klabel(#ffi_struct), symbol]
endmodule

module FFI
  imports FFI-SYNTAX
  imports private BYTES
  imports private STRING
  imports private BOOL
  imports private LIST
  imports private INT
```

## FFI Calls

The `#ffiCall` functions are designed to call a native C ABI function and return a native result. They come in three variants:

### Non-variadic

In the first variant, `#ffiCall(Address, Args, ArgTypes, ReturnType)` takes an integer address of a function (which can be obtained from `#functionAddress`), a `List` of `Bytes` containing the arguments of the function, a `List` of `FFITypes` containing the types of the parameters of the function, and an `FFIType` containing the return type of the function, and returns the return value of the function as a `Bytes`.

```
syntax Bytes ::= "#ffiCall" "(" Int "," List "," List "," FFIType ")" [function, hook(FFI
```

### Variadic

In the second variant,
`#ffiCall(Address, Args, FixedTypes, VariadicTypes, ReturnType` takes an
integer address of a function, a `List` of `Bytes` containing the arguments of the call, a `List` of `FFITypes` containing the types of the fixed parameters of the function, a `List` of `FFITypes` containing the types of the variadic parameters of the function, and an `FFIType` containing the return type of the function, and returns the return value of the function as a `Bytes`.

```
syntax Bytes ::= "#ffiCall" "(" Int "," List "," List "," List "," FFIType ")" [function,
```

### Generic

In the third variant,
`#ffiCall(IsVariadic, Address, Args, ArgTypes, NFixed, ReturnType` takes
a boolean indicating whether the function is variadic or not, an integer address of a function, a `List` of `Bytes` containing the arguments of the call, a `List` of `FFITypes` containing the parameter typess of the call followed by the types of the variadic arguments of the call, if any, an `Int` containing how many of the arguments of the call are fixed or not, and an `FFIType` containing the return type of the function, and returns the return value of the function as a `Bytes`.

```
syntax Bytes ::= "#ffiCall" "(" Bool "," Int "," List "," List "," Int "," FFIType ")" [fu

rule #ffiCall(false, Addr::Int, Args::List, Types::List, _, Ret::FFIType) => #ffiCall(Addr
rule #ffiCall(true, Addr::Int, Args::List, Types::List, NFixed::Int, Ret::FFIType) => #ffi
```

## Symbol Lookup

The FFI module provides a mechanism to look up any function symbol and return
that function's address.

```
syntax Int ::= "#functionAddress" "(" String ")" [function, hook(FFI.address)]
```

## Direct Memory Management

Most memory used by the LLVM backend to represent terms is managed automatically via garbage collection. However, a consequence of this is that a particular term does not have a fixed address across its entire lifetime in most cases. Sometimes this is undesirable, especially if you intend for the address of the memory to be taken by the semantics or if you intend to pass this memory directly to native code. As a result, the FFI module exposes the following unsafe APIs for memory management. Note that use of these APIs leaves the burden of memory management completely on the user, and thus misuse of these functions can lead to things like use-after-free and other memory corruption bugs.

### Allocation

`#alloc(Key, Size, Align)` will allocate `Size` bytes with an alignment requirement of `Align` (which must be a power of two), and return it as a `Bytes` term. The memory is uniquely identified by its key and that key will be used later to free the memory. The memory is not implicitly freed by garbage collection; failure to call `#free` on the memory at a later date can lead to memory leaks.

```
syntax Bytes ::= "#alloc" "(" KItem "," Int "," Int ")" [function, hook(FFI.alloc)]
```

### Addressing

`#addess(B)` will return an `Int` representing the address of the first byte of B, which must be a `Bytes`. Unless the `Bytes` term was allocated by `#alloc`, the return value is unspecified and may not be the same across multipl invocations on the same byte buffer. However, it is guaranteed that memory allocated by `#alloc` will have the same address throughout its lifetime.

```
syntax Int ::= "#address" "(" Bytes ")" [function, hook(FFI.bytes_address)]
```

### Deallocation

`#free(Key)` will free the memory of the `Bytes` object that was allocated by a previous call to `#alloc`. If `Key` was not used in a previous call to `#alloc`, or the memory was already freed, no action is taken. It will generate undefined behavior if the `Bytes` term returned by the previous call to `#alloc` is still referenced by any other term in the configuration or a currently evaluating rule. The function returns `.K`.

```
syntax K ::= "#free" "(" KItem ")" [function, hook(FFI.free)]
```

### Reading

`#nativeRead(Addr, Mem)` will read native memory at address `Addr` into `Mem`, reading exactly `lengthBytes(Mem)` bytes. This will generate undefined behavior if `Addr` does not point to a readable segment of memory at least `lengthBytes(Mem)` bytes long.

```
  syntax K ::= "#nativeRead" "(" Int "," Bytes ")" [function, hook(FFI.read)]
```

### Writing

`#nativeWrite(Addr, Mem)` will write the contents of `Mem` to native memory at address `Addr`. The memory will be read prior to being written, and a write will only happen if the memory has a different value than the current value of `Mem`. This will generate undefined behavior if `Addr` does not point to a readable segment of memory at least `lengthBytes(Mem)` bytes long, or if the memory at address `Addr` has a different value than currently contained in `Mem`, and the memory in question is not writeable.

```
  syntax K ::= "#nativeWrite" "(" Int "," Bytes ")" [function, hook(FFI.write)]
endmodule
```

## Syntax of JSON

K provides builtin support for reading/writing to JSON. While the `JSON-SYNTAX` module is not precisely the syntax of JSON (utilizing K's syntax for strings, integers, and floating point numbers rather than the syntax used by JSON), you can still convert directly to/from the actual syntax of JSON using the `JSON2String` and `String2JSON` hooks.

```
module JSON-SYNTAX
    imports INT-SYNTAX
    imports STRING-SYNTAX
    imports BOOL-SYNTAX
    imports FLOAT-SYNTAX

    syntax JSONs   ::= List{JSON,","}      [klabel(JSONs)      , symbol]
    syntax JSONKey ::= String
    syntax JSON    ::= "null"              [klabel(JSONnull)   , symbol]
                     | String | Int | Float | Bool
                     | JSONKey ":" JSON    [klabel(JSONEntry)  , symbol]
                     | "{" JSONs "}"       [klabel(JSONObject) , symbol]
                     | "[" JSONs "]"       [klabel(JSONList)   , symbol]
endmodule
```

### Conversion between `JSON` and `String`

Given a string written in valid JSON, you can convert it to the `JSON` sort with the `String2JSON` function. Assuming the user has not extended the syntax of the `JSON` sort with their own constructors, any term of sort `JSON` can also be converted to a `String` using the `JSON2String` function.

```
module JSON
    imports JSON-SYNTAX

    syntax String ::= JSON2String(JSON) [function, hook(JSON.json2string)]

    syntax JSON ::= String2JSON(String) [function, hook(JSON.string2json)]
endmodule
```

## Rational Numbers in K

K provides support for arbitrary-precision rational numbers represented as a quotient between two integers. The sort representing these values is `Rat`. `Int` is a subsort of `Rat`, and it is guaranteed that any integer will be represented as an `Int` and can be matched as such on the left hand side of rules. K also supports the usual arithmetic operators over rational numbers.

```
module RAT-SYNTAX
  imports INT-SYNTAX
  imports private BOOL

  syntax Rat

  syntax Rat ::= Int
```

### Arithmetic

You can:

- Raise a rational number to any negative or nonnegative integer.
- Multiply or divide two rational numbers to obtain a product or quotient.
- Add or subtract two rational numbers to obtain a sum or difference.

```
  syntax Rat ::= left:
                   Rat "^Rat" Int [function, functional, klabel(_^Rat_), symbol, left, smtlib(
                 > left:
                   Rat "*Rat" Rat [function, functional, klabel(_*Rat_), symbol, left, smtlib(
                 | Rat "/Rat" Rat [function,             klabel(_/Rat_), symbol, left, smtlib(
                 > left:
                   Rat "+Rat" Rat [function, functional, klabel(_+Rat_), symbol, left, smtlib(
                 | Rat "-Rat" Rat [function, functional, klabel(_-Rat_), symbol, left, smtlib(
```

## Comparison

You can determine whether two rational numbers are equal, unequal, or compare one of less than, less than or equalto, greater than, or greater than or equal to the other:

```
  syntax Bool ::= Rat  "==Rat" Rat [function, functional, klabel(_==Rat_),  symbol, smtlib(r
                | Rat "=/=Rat" Rat [function, functional, klabel(_=/=Rat_), symbol, smtlib(r
                | Rat   ">Rat" Rat [function, functional, klabel(_>Rat_),   symbol, smtlib(r
                | Rat  ">=Rat" Rat [function, functional, klabel(_>=Rat_),  symbol, smtlib(r
                | Rat   "<Rat" Rat [function, functional, klabel(_<Rat_),   symbol, smtlib(r
                | Rat  "<=Rat" Rat [function, functional, klabel(_<=Rat_),  symbol, smtlib(r
```

## Min/Max

You can compute the minimum and maximum of two rational numbers:

```
  syntax Rat ::= minRat(Rat, Rat) [function, functional, klabel(minRat), symbol, smtlib(ratm
               | maxRat(Rat, Rat) [function, functional, klabel(maxRat), symbol, smtlib(ratm
```

## Conversion to Floating Point

You can convert a rational number to the nearest floating point number that is representable in a `Float` of a specified number of precision and exponent bits:

```
  syntax Float ::= Rat2Float(Rat, precision: Int, exponentBits: Int) [function]
endmodule
```

## Implementation of Rational Numbers

The remainder of this file consists of an implementation in K of the operations listed above. Users of the RAT module should not use any of the syntax defined in any of these modules.

As a point of reference for users, it is worth noting that rational numbers are normalized to a canonical form by this module,. with the canonical form bearing the property that it is either an `Int`, or a pair of integers
`I /Rat J` such that
`I =/=Int 0 andBool J >=Int 2 andBool gcdInt(I, J) ==Int 1` is always true.

```
module RAT-COMMON
  imports RAT-SYNTAX

  // invariant of < I , J >Rat : I =/= 0, J >= 2, and I and J are coprime
  syntax Rat ::= "<" Int "," Int ">Rat" [format(%2 /Rat %4)]
endmodule
```

```
module RAT-SYMBOLIC [symbolic, kore]
  imports private RAT-COMMON
  imports ML-SYNTAX
  imports private BOOL

  rule
    #Ceil(@R1:Rat /Rat @R2:Rat)
  =>
    {(@R2 =/=Rat 0) #Equals true} #And #Ceil(@R1) #And #Ceil(@R2)
  [simplification]
endmodule

module RAT-KORE [kore]
  imports private RAT-COMMON
  imports private K-EQUAL

  /*
   * equalities
   */

  // NOTE: the two rules below may not work correctly in non-kore backends

  rule R ==Rat S => R ==K S

  rule R =/=Rat S => R =/=K S
endmodule

module RAT-KAST [kast]
  imports private RAT-COMMON
  imports private INT
  imports private BOOL

  /*
   * equalities for non-kore backends such as the java backend
   */

  rule < I , I' >Rat ==Rat < J , J' >Rat => I ==Int J andBool I' ==Int J'
  rule _:Int         ==Rat < _ , _  >Rat => false
  rule < _ , _  >Rat ==Rat _:Int         => false
  rule I:Int         ==Rat J:Int         => I ==Int J

  rule R =/=Rat S => notBool (R ==Rat S)
endmodule

module RAT [private]
  imports private RAT-COMMON
```

```
imports public RAT-SYMBOLIC
imports public RAT-KORE
imports public RAT-KAST
imports public RAT-SYNTAX
imports private INT
imports private BOOL

/*
 * arithmetic
 */

rule < I , I' >Rat +Rat < J , J' >Rat => ((I *Int J') +Int (I' *Int J)) /Rat (I' *Int J')
rule I:Int         +Rat < J , J' >Rat => ((I *Int J') +Int J) /Rat J'
rule < J , J' >Rat +Rat I:Int         => I +Rat < J , J' >Rat
rule I:Int         +Rat J:Int         => I +Int J

rule < I , I' >Rat *Rat < J , J' >Rat => (I *Int J) /Rat (I' *Int J')
rule I:Int         *Rat < J , J' >Rat => (I *Int J) /Rat J'
rule < J , J' >Rat *Rat I:Int         => I *Rat < J , J' >Rat
rule I:Int         *Rat J:Int         => I *Int J

rule < I , I' >Rat /Rat < J , J' >Rat => (I *Int J') /Rat (I' *Int J)
rule I:Int         /Rat < J , J' >Rat => (I *Int J') /Rat J
rule < I , I' >Rat /Rat J:Int         => I /Rat (I' *Int J) requires J =/=Int 0
rule I:Int         /Rat J:Int         => makeRat(I, J)      requires J =/=Int 0

// derived

rule R -Rat S => R +Rat (-1 *Rat S)

// normalize

syntax Rat ::= makeRat(Int, Int)      [function]
             | makeRat(Int, Int, Int) [function]

rule makeRat(0, J) => 0 requires J =/=Int 0

rule makeRat(I, J) => makeRat(I, J, gcdInt(I,J)) requires I =/=Int 0 andBool J =/=Int 0

// makeRat(I, J, D) is defined when I =/= 0, J =/= 0, D > 0, and D = gcd(I,J)
rule makeRat(I, J, D) => I /Int D                       requires J ==Int D // implies J >
rule makeRat(I, J, D) => < I /Int D , J /Int D >Rat     requires J >Int 0 andBool J =/=Int
rule makeRat(I, J, D) => makeRat(0 -Int I, 0 -Int J, D) requires J <Int 0

// gcdInt(a,b) computes the gcd of |a| and |b|, which is positive.
syntax Int ::= gcdInt(Int, Int) [function, public]
```

```
rule gcdInt(A, 0) => A        requires A >Int 0
rule gcdInt(A, 0) => 0 -Int A requires A <Int 0
rule gcdInt(A, B) => gcdInt(B, A %Int B) requires B =/=Int 0 // since |A %Int B| = |A| %In

/*
 * exponentiation
 */

rule _ ^Rat 0 => 1
rule 0 ^Rat N => 0 requires N =/=Int 0

rule < I , J >Rat ^Rat N => powRat(< I , J >Rat, N) requires N >Int 0
rule X:Int        ^Rat N => X ^Int N                requires N >Int 0

rule X ^Rat N => (1 /Rat X) ^Rat (0 -Int N) requires X =/=Rat 0 andBool N <Int 0

// exponentiation by squaring

syntax Rat ::= powRat(Rat, Int) [function]

// powRat(X, N) is defined when X =/= 0 and N > 0
rule powRat(X, 1) => X
rule powRat(X, N) => powRat(X *Rat X, N /Int 2) requires N >Int 1 andBool N %Int 2  ==Int
rule powRat(X, N) => powRat(X, N -Int 1) *Rat X requires N >Int 1 andBool N %Int 2 =/=Int

/*
 * inequalities
 */

rule R >Rat S => R -Rat S >Rat 0 requires S =/=Rat 0

rule < I , _ >Rat >Rat 0 => I >Int 0
rule I:Int        >Rat 0 => I >Int 0

// derived

rule R >=Rat S => notBool R <Rat S

rule R <Rat S => S >Rat R

rule R <=Rat S => S >=Rat R

rule minRat(R, S) => R requires R <=Rat S
rule minRat(R, S) => S requires S <=Rat R
```

```
  rule maxRat(R, S) => R requires R >=Rat S
  rule maxRat(R, S) => S requires S >=Rat R

  syntax Float ::= #Rat2Float(Int, Int, Int, Int) [function, hook(FLOAT.rat2float)]
  rule Rat2Float(Num:Int, Prec:Int, Exp:Int) => #Rat2Float(Num, 1, Prec, Exp)
  rule Rat2Float(< Num, Dem >Rat, Prec, Exp) => #Rat2Float(Num, Dem, Prec, Exp)

endmodule
```

## Capture-Aware Substitution in K

One of the traditional ways in which functional languages are given operational semantics is via substitution. In particular, you can view a function as declaring a particular bound variable, the parameter of the function, as well as the body of the function, within which both bound and free variables can occur, and implement the process of beta-reduction (one of the axioms of the lambda calculus) by means of a substitution operator which is aware of the difference between free variables and bound variables and prevents variable capture.

In K this is implemented using two mechanisms: The `KVar` sort, and the `binder` attribute.

### The `KVar` Sort

K introduces a new hooked sort, `KVar`, which the substitution operator (defined below) understands in a particular way. The syntax of `KVar` is the same as for sort `Id` in `DOMAINS`, but with a different sort name. Similarly, some of the same operators are defined over `KVar` which are defined for `Id`, such as conversion from `String` to `KVar` and support for the `!Var:KVar` syntax.

A `KVar` is simply an identifier with special meaning during substitution. `KVar`s must begin with a letter or underscore,
and can be followed by zero or more letters, numbers, or underscores.

```
module KVAR-SYNTAX-PROGRAM-PARSING
  imports BUILTIN-ID-TOKENS

  syntax KVar ::= r"(?<![A-Za-z0-9\\_])[A-Za-z\\_][A-Za-z0-9\\_]*"     [prec(1), token]
                | #LowerId                                             [token]
                | #UpperId                                             [token]
endmodule

module KVAR-SYNTAX
  syntax KVar [token, hook(KVAR.KVar)]
endmodule
```

```
module KVAR-COMMON
  imports KVAR-SYNTAX
  imports private STRING

  syntax KVar ::= String2KVar (String) [function, functional, hook(STRING.string2token)]
  syntax KVar ::= freshKVar(Int)    [freshGenerator, function, functional, private]

  rule freshKVar(I:Int) => String2KVar("_" +String Int2String(I))
endmodule

module KVAR-SYMBOLIC [symbolic, kast]
  imports KVAR-COMMON
  imports private STRING

  syntax KItem  ::= "#parseToken"  "(" String "," String ")"  [function, klabel(#parseKVar),
  rule String2KVar(S:String) => {#parseToken("KVar", S)}:>KVar
endmodule

module KVAR
  imports KVAR-COMMON
  imports KVAR-SYMBOLIC
endmodule
```

### The `binder` Attribute

A production can be given the attribute `binder`. Such a production must have at least two nonterminals. The first nonterminal from left to right must be of sort `KVar`, and contains the bound variable. The last nonterminal from left to right contains the term that is bound. For example, I could describe lambdas in the lambda calculus with the production
`syntax Val ::= "lambda" KVar "." Exp [binder]`.

### Substitution

K provides a hooked implementation of substitution, currently only implemented
on the Java and LLVM backends. Two variants exist: the first substitutes a single `KVar` for a single `KItem`. The second takes a `Map` with `KVar` keys and `KItem` values, and substitutes each element in the map atomically.

Internally, this is implemented in the LLVM backend by a combination of `de Bruijn` indices for bound variables and names for free variables. Free variables are also sometimes given a unique numeric identifier in order to prevent capture, and the rewriter will automatically assign unique names to such identifiers when rewriting finishes. The names assigned will always begin

with the original name of the variable and be followed by a unique integer suffix. However, the names assigned after rewriting finishes might be different from the names that would be assigned if rewriting were to halt prematurely, for example due to `krun --depth`.

```
module SUBSTITUTION
  imports private MAP
  imports KVAR

  syntax {Sort} Sort ::= Sort "[" KItem "/" KItem "]"  [function, hook(SUBSTITUTION.substOne
  syntax {Sort} Sort ::= Sort "[" Map "]"      [function, hook(SUBSTITUTION.substMany), impu
endmodule
```

# Projects using K

## Featured

- **KWasm (Aug 2015 - Present)**

  KWasm is the K semantics of WebAssembly.
  WebAssembly is a low-level (but simple and streamlined) assembly language that was originally developed to provide a fast execution engine for browser-based tools.
  More recently, it has been used in several blockchain smart-contract platforms as the underlying language for executing financial agreements.
  KWasm has been used for measuring coverage of test-suites over Wasm code and verifying programs which are compiled to Wasm.

- **KEVM (Sep 2017 - Present)**

  KEVM is the K semantics of the Ethereum Virtual Machine.
  It passes all of the Ethereum Test Suite, and is used for verifying EVM programs.

- **IELE (Oct 2016 - Present)**

  IELE is the underlying VM integrated into the Cardano blockchain.
  IELE is a register-based VM (inspired by LLVM), which attempts to avoid many of the missteps in design present in EVM.

- **K-Michelson (Oct 2019 - Present)**

  K-Michelson is the K semantics of Michelson blockchain programming language, which powers the Tezos blockchain.

KMichelson provides additional testing tools for developers, including a unit-testing framework which is extendable to symbolic property testing.

- **C (Jul 2010 - Present)**

  The K semantics of the C programming language specifies the translation, linking, and execution semantics of the C language according to the official C standard.
  It has been used to build tools like RV-Match, which detects undefined behaviors in users programs by running their test-suites through the C semantics.

## Archived

- llvm (2011-2018)
- python (2012-2013)
- java (2012-2016)
- ocaml (2012-2013)
- javascript (2013-2015)
- aadl (2013-2013)
- alk (2013-2014)
- cink (2013-2015)
- jvm (2013-2014)
- modelink (2013-2013)
- javacard (2014-2014)
- orc (2016-2017)
- haskell core (2016-2017)
- plutus core (2016-2019)
- X86-64 (2017-2020)
- vyper (2017-2018)
- erc20 (2017-2018)
- solidity (2018-2019)
- eei (2018-2019)
- erc777 (2018-2018)
- k (2018-2020)
- yul (2019-2019)
- KEwasm (2019-2020)
- hybrid programs (2020-2020)
- Boogie (2020)
- Elrond (2020-2021)
- P4 (2016-2021)