FRED Modeling Language Guide

['Epistemix Inc.']

Sep 24, 2022

CONTENTS

Return to docs home

Welcome to FRED, the Framework for Reconstructing Epidemiological Dynamics. This document provides detailed information on using the FRED Modeling Language™ to create FRED programs. This guide is focused on the syntax and structure of a FRED program. For information on compiling and executing FRED programs, please see the documentation for the FRED Modeling Platform™.

The following chapters will provide details on the FRED Modeling Language and the components of a FRED model, including:

CHAPTER

ONE

CHAPTER 1: INTRODUCTION

Here we give a brief introduction to the FRED modeling language. For some background on agent-based modeling in general and FRED, please see the FRED Modeling Language Introduction document.

Note: Throughout this guide, the terms **FRED model** and **FRED program** are used interchangeably. These terms are used when discussing a collection of code that defines a valid model, able to be compiled and executed using the FRED modeling platform.

Before beginning, it is important to note that FRED is not itself a model. Rather, FRED is a language and a platform for building models. The model comes from you, the FRED user, the modeler. It should be recognized that building a model is a challenging activity that often requires a lot of effort on the modeler's part. FRED makes it easier to define and build a model, by efficiently executing simulation runs of the user's model, and by providing numerous ways to collect data and to visualize the results of the model. The ultimate quality of the results depends on the success of the modeler in building an appropriate model for the purpose at hand and in communicating the model to FRED for execution. Having said that, we believe that using FRED can save the modeler significant effort by eliminating the need to develop custom simulation software, and that interacting with the FRED user community may also contribute to the user's successful modeling efforts.

Building a serious model is almost always an iterative process. Working toward building a model with FRED usually includes the following steps:

1. Decide if FRED is suited to your problem
2. Create a conceptual model
3. Create rules for individuals
4. Create and run a FRED Model
5. Analyze model output and test against other known data
6. Revise model and repeat

Hopefully this guide and our other documentation will inspire you to learn FRED and to build great models.

1.1 Hello FRED

Before reviewing the various elements and syntax of FRED models, we begin with a very simple model. The following model prints out "Helo, World!"

```
simulation {
   locations = none
   default_model = none
   start_date = 2020-Jan-01
   end_date = 2020-Jan-01
}

startup {
   print("Hello, World!")
}
```

The first block, the `simulation block`, specifies a location to simulate. In this case, no location is selected. By default, a simulation includes a "default model" that has every individual in the selected location pursue a set of daily activities. In this case, we're skipping the default model. Finally, every FRED model indicates a `start_date` and and `end_date`. In this case, we're simulating a single day.

The *startup block* indicates actions to take at the start of the simulation. In this case the only action is to print a message.

1.2 A First Simulation Model

In addition to the simulation location and time frame, most models include at least one **condition**.

A minimal FRED model is shown below:

```
simulation {
   # simulated location
   locations = Jefferson_County_PA  # a small county in Pennsylvania

   # simulated time frame
   start_date = 2020-Jan-01
   end_date = 2020-Jan-10
}

# a trivial condition
condition ACTIVE {
   start_state = Start

   state Start {
      wait(0)
      default(Excluded)
   }
}
```

When this model is run, the population of Jefferson County, PA, comprised of 45,318 individuals, is loaded into the simulation, and individual agents are assigned to a variety of places that include households, schools, and workplaces. In this model, agents follow their normal activities for a period of 10 days. All individual agents begin in the `Start` **state** of a trivial condition called `ACTIVE` and immediately transition

to the `Excluded` state of this condition, where they remain. Clearly, nothing of interest is added by the `ACTIVE` condition in this model. Later in this guide, more complex conditions will be discussed.

Note: Every condition includes the `Excluded` state which does not need to be explicitly defined.

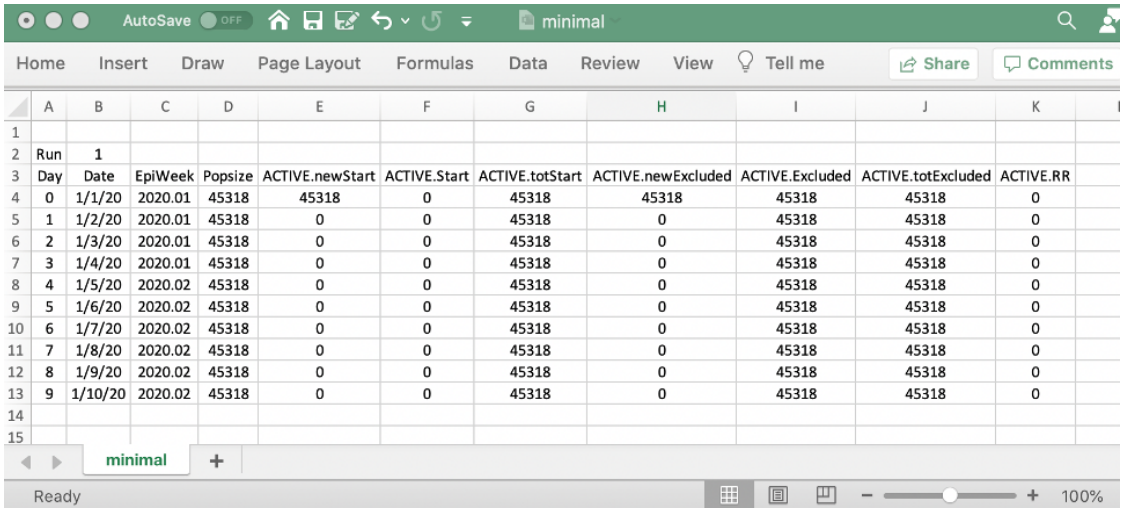
Suppose that the model above is stored in a file called minimal.fred. The model can then be executed using the command line:

```
$ fred_job -k hello_fred -p minimal.fred
```

This command executes the model as a **simulation**. `-k hello_fred` calls this simulation "hello_fred", while `-p minimal.fred` tells the FRED Modeling Platform™ which file contains the model.

Note: It is conventional to use the .fred suffix on FRED model files, but it is not required.

The output of the simulation includes a spreadsheet that shows the counts of how many agents are in each state in the model at the end of each day:



	A	B	C	D	E	F	G	H	I	J	K
1											
2	Run	1									
3	Day	Date	EpiWeek	Popsize	ACTIVE.newStart	ACTIVE.Start	ACTIVE.totStart	ACTIVE.newExcluded	ACTIVE.Excluded	ACTIVE.totExcluded	ACTIVE.RR
4	0	1/1/20	2020.01	45318	45318	0	45318	45318	45318	45318	0
5	1	1/2/20	2020.01	45318	0	0	45318	0	45318	45318	0
6	2	1/3/20	2020.01	45318	0	0	45318	0	45318	45318	0
7	3	1/4/20	2020.01	45318	0	0	45318	0	45318	45318	0
8	4	1/5/20	2020.02	45318	0	0	45318	0	45318	45318	0
9	5	1/6/20	2020.02	45318	0	0	45318	0	45318	45318	0
10	6	1/7/20	2020.02	45318	0	0	45318	0	45318	45318	0
11	7	1/8/20	2020.02	45318	0	0	45318	0	45318	45318	0
12	8	1/9/20	2020.02	45318	0	0	45318	0	45318	45318	0
13	9	1/10/20	2020.02	45318	0	0	45318	0	45318	45318	0
14											
15											

The first three columns give the day counter, the date, and the **epi-week** for each simulation day. This is followed by three columns for each of the states. For example, the `Start` state has columns called `ACTIVE.newStart`, `ACTIVE.Start`, and `ACTIVE.totStart`. The `newStart` column contains the number of agents who enter that state during the day. The `Start` column gives the number of agents in that state and the end of the day. The `totStart` column is a running total of all agents who have ever entered that state. The final column, `ACTIVE.RR`, gives the **reproductive rate**, or transmission rate for this condition, which is not applicable in this particular model.

Note: An epidemiological week, i.e. epi-week, contains seven days, beginning with Sunday and ending with Saturday. By definition, the end of the first week of the year must fall at least four days into the year. Weeks are numbered 1 to 53, although most years consist of only 52 epi-weeks.

1.3 The FRED Simulation Cycle

The FRED language represents a fully functioning agent-based simulation system that can be completely customized as needed. FRED offers a discrete-time model with a time step of one second. The duration of simulations can be from one day to 100 years. To use FRED, the user creates a FRED model in the FRED programming language. The FRED Modeling Platform™ compiles and executes the simulation cycle represented by a FRED program.

The simulation cycle consists of the following:

1. Select a geographic location for the simulation. This is one or more locations represented in the synthetic population.
2. Select the timeframe for the simulation.
    - The simulation begins at midnight on the start date.
    - The simulation ends at 11:59:59pm on the end date.
3. For each step of the simulation, perform the following for each condition in the program.
    - Identify the agents that need to be updated according to the program-defined rules.
    - For each identified agent:
        1. Select the agent's next state.
        2. Perform the actions associated with the agent's next state.
        3. If the condition involves interactions among agents, then:
            - simulate the agent interactions within the defined interaction groups
            - update this or other agent's current state based on these interactions.
4. After each day of simulation, record statistics about the conditions in the population.

A complete simulation cycle for a single program as described above is called a **run**. Since FRED models are stochastic, it is often desirable to perform several runs in order to produce meaningful statistics about the performance of the model. A set of runs of the same model is called a **job**. Upon completion of a FRED job, the user can obtain **reports** of the output, including spreadsheets, plots, and videos that display the location of user-selected events on a **report** of the simulation area.

1.4 Declarative Programming in FRED

FRED is a declarative programming langauge for writing agent based models. Declarative programming is a programming paradigm that expresses the logic of the computation without expressing the control flow. (Lloyd, J.W., Practical Advantages of Declarative Programming). Within the general class of declarative programming languages, FRED can be classified as a domain-specific language whose domain is agent-based modeling (ABM). The FRED language consists of two primary kinds of statements: **property statements** and **rule statements**. Property statements define the static features of the model, including the location, the range of dates for the simulation, all concepts of interest, what information should be tracked and reported, and the initial conditions. Rule statements define the dynamic features of the model, including how agents and their environment change state over time. The FRED platform processes the FRED program, sets up the population, applies the initial conditions, simulates the activities and interactions of the agents, and tracks all user-defined conditions within the population. FRED outputs several reports, charts, and visualizations.

The declarative programming style of FRED provides several advantages for agent-based modeling:

- No traditional computer programming experience is required.
- You can focus on scientific effort (e.g. data collection, conceptual modeling, experimental design).
- FRED provides a simple workflow environment for you and manages all the data produced by the simulation and associated metadata.

CHAPTER

TWO

CHAPTER 2: FRED MODELS

This chapter reviews the fundamental elements of the FRED programming languauge, as well as the basic requirements of any valid FRED model.

2.1 The Structure of a FRED Program

A FRED model consists of a collection of one or more **statements** and **blocks**. Even the most complex models can be broken down into these fundamental elements.

2.1.1 statements

The majority of lines of code in a given model will generally be dedicated to statements. There are three types of statments in the FRED programming langauge:

- `include` statements
- property statements
- rule statements

`include` statements allow models to be split up into multiple FRED files. All FRED files are plain text files. By convention, the file extension .fred is used to indicate a file contains valid FRED model elements. A entire model may be contained in a single .fred file, or it may be split up into mutliple files for convenience.

Include statements have the form:

```
include <filename.fred>
```

This type of statement has the effect of inserting the contents of the file, <filename.fred>, into the FRED model at that point. The `include` statement allows a user to collect mutliple files into a single location. Breaking up models into elements is discussed in more detail in *Chapter 14*.

Property statements have the form:

```
<property> = <value>
```

where <property> is one of the properties defined in later chapters of this guide.

Note: If multiple statements define the same property, the property definition occurring latest in the program will be used to set the value of the property.

Rule statements (i.e rules), are a broad class of statements found within states. These may include **actions**, **wait rules**, and **transition rules**. Each of these rules may contain any number of **expressions**. These rules are discussed in detail in chapter *Chapter 4*.

Note: Statements are terminated by an end-of-line. If the final character of a statement is a \, the statement is continued onto the next line.

Note: Comments are introduced by the # character–all characters after a # are ignored until an end-of-line. Lines containing only comments or white space are ignored.

2.1.2 blocks

Blocks can be thought of as a collection of related property and/or rule statements. There are a variety of blocks which may appear in a FRED model, including some fundamental blocks:

- simulation
- condition
- state
- variables
- place
- network
- startup
- agent_startup
- use
- comment

and a few more advanced-usage blocks:

- configuration
- restart
- agent_restart
- prototype

Blocks have the form:

```
<block_type> [<block_name>] {
    ...
}
```

Some blocks must be named (e.g. condition, state, place, and network blocks) while other blocks do not take a name (e.g. simulation, variables, and startup blocks). For example, in the simple program discussed in the introduction to this guide, a condition block called `ACTIVE` was defined:

```
condition ACTIVE {
   ...
}
```

The *simulation* and *comment* blocks are discussed in this chapter. The remaining block types are discussed at length at various points within this guide.

Other than comment lines beginning with a # and include statements, all other statements must occur within one of these blocks. Some blocks may contain other blocks. For example, a condition block will generally contain several state blocks, and comment blocks can be placed within any other block. As discussed in *Chapter 4*, state blocks may contain any number of **rule statements** that together define the meaning of conditions and the behaviour of agents.

If a program contains multiple instances of any block, the FRED compiler concatenates the instances in the order of occurrence. For example:

```
simulation {
    start_date = 2020-Jan-01
}

simulation {
    end_date = 2020-Dec-31
}
```

is equivalent to

```
simulation {
    start_date = 2020-Jan-01
    end_date = 2020-Dec-31
}
```

The notation <name>.<property> = <value> can also be used to add to or modify property statements to previously defined conditions, places, or networks. This is sometimes referred to as **dot notation**. For example:

```
place School { }
School.contact_rate = 0.35
```

is equivalent to

```
place School {
    contact_rate = 0.35
}
```

2.2 The Simulation Block

The simulation block defines some basic properties of a model, including the simulation location(s), the simulation time frame (start and end dates), and the simulation logging and output levels.

Warning: A simulation block is required for all valid FRED models.

A basic simulation block may look like the following:

```
simulation {
    locations = Pittsburgh_PA
    start_date = 2020-Jan-01
```

(continues on next page)

(continued from previous page)

```
    end_date = 2020-Dec-31
}
```

This block indicates that the simulation will be run with a synthetic population associated with Pittsburgh, PA. The simluation will begin on January 1, 2020 and run through December 31, 2020.

Multiple locations can be specified using a comma-separated list. White space is also permitted around the commas.

```
simulation {
    locations = Allegheny_County_PA, Jefferson_County_PA
    start_date = 2020-Jan-01
    end_date = 2020-Dec-31
}
```

The `locations` property indicates which portion of a synthetic population should be used in a given model. While it may vary for different synthetic populations, valid locations will be associated with an administrative district like a U.S. county or state. Synthetic populations are discussed in more detail in *chapter 3*.

There are two ways to specify the time period, by providing an `end_date` or a `days` property. If `days` is set to a positive number, it overrides the `end_date` property.

A simulation block must contain three statements defining the location(s) and time period. If any of these element is missing, an error will be generated when compiling the model. If no location is desired, set *location = none*.

The complete set of properties and their default values are shown below. Each of theses properties is described in later section of this Guide and in the Reference pages.

```
simulation {

    ##### Simulation Options
    locations = none
    start_date = none
    end_date = none
    days = 0
    seed = 123456
    max_loops = 1000000
    update_progress = 60
    substeps = 0
    default_model = 1
    all_group_agents = 0

    ##### Output Options
    weekly_data = 0
    file_buffer_size = 10000
    dump_files = 0
    test = 0
    snapshot_interval = 0
    snapshot_final = 0
    snapshot_date = none
    snapshots = 0
    condition_timing = 0
```

(continues on next page)

(continued from previous page)

```
    ##### Input Options
    population_directory = none
    country = usa
    population_version = US_2010.v5

    ##### Flags
    use_mean_latitude = 1
    enable_transmission_bias = 1
    enable_population_dynamics = 0
    enable_aging = 1
    use_index_id = 0

}
```

2.3 The Comment Block

A comment block is used to document a model and to provide metadata such as the author's name or the creation date of the model. Comment blocks are optional and are ignored by the FRED compiler. Comment blocks take the form:

```
comment [<optional comment_name>] {
       All contents of this block are treated as comments.
       ...
}
```

Comment blocks may appear anywhere in the program, including within other blocks. All content within a coment block is ignored.

CHAPTER

THREE

CHAPTER 3: AGENTS

FRED is a framework for agent-based modeling. There are three kinds of agents in FRED:

- ordinary individual agents,
- group agents, and
- the Meta agent.

3.1 Individual Agents

Although there is no restriction about what an agent represents in FRED, usually the ordinary agents in FRED represent individual people. We will refer to ordinary agents as either individuals or simply, agents. The synthetic population usually includes a number of demographic characteristics for ordinary agents such as age, race, and sex, but these are all optional, and other characteristcs are easily defined in the model. Agents interact with other agents in mixing groups, such as households, neighborhoods, schools, and workplaces. Several such mixing groups are defined by the synthteic population and others can be defined by the specific model.

3.2 Group Agents

Group agents represent an abstract agency associated with a specific mixing group. For example, the user can define an group agent that represents the principal of a school, and this agent might be responsible for making decisions such as when to close the school in an emergency.

As described in *Chapter 9*, each mixing group has a schedule that defines the days and times that agents can interact within that group. The group agent of a mixing group can override the normal schedule by closing the group. For example, the group agent of a school could decide to close it in the case of a health emergency, or a hospital group agent may decide to open a new community health center. Group agents can also control certain aspects of how agents behave within their mixing group.

Group agents are unlike oridinary agents in that they do not have demogrphics characteristics, and they do not interact directly with ordinary agents. That is, they are not considered to be physically present in mixing groups.

The user enables the use of group agents for a mixing group class by the property statement, `has_group agent = 1`. For example, if a model includes school closures controlled by administrators, the program should include:

```
place School {
    has_group_agent = 1
}
```

The effect is that FRED will generate an group agent agent for each school in the model.

To specify that all groups shoud have a group agent, the following simulation property can be set:

```
simulation {
    all_group_agents = 1
}
```

For each condition, group agents start in a state specified by the property statement `group_start_state = <state-name>` and change states via transition rules just like ordinary agents. If no `group_start_state` is specified for a condition, then group agent agents are not active for that condition. See *Chapter 4* for more details on how to define a condition and states.

There are several special actions that can be taken by group agents, described in detail in *Chapter 10*.

Group agents have the same *id* as their group. For example, suppose the user defines a new place called `Pharmacy`, and generates two pharmacies as follows:

```
place Pharmacy {
    has_group agent = 1
    site = 942003001, 40.451164, -79.999803, 230.1
    site = 942003002, 40.626449, -79.723195, 240.5
}
```

Then two group agents are generated, one with id *942003001* and another with id *942003002*.

3.3 The Meta Agent

There is a single Meta agent for each simulation. This Meta agent can take actions that affect the overall simulation. For example, the Meta agent can start disease outbreaks by infecting selected individuals from a source of infection that is not modeled explicitly in the simulation.

Just like ordinary agents and group agents, the behavior of the Meta agent is controlled by defining conditions, states, and rules.

The Meta agent starts in a state specified by the property statement `meta_start_state = <state-name>` and changes states via transition rules just like other agents. If no `meta_start_state` is specified for a condition, then the Meta agent is not active for that condition.

There are several special actions that can be taken by the Meta agent, described in detail in *Chapter 10*.

3.4 Synthetic Populations

A **Synthetic Population** is a data set that represents each person and household in a given location with geospatial accuracy and contains no personally identifiable information. In addition, a synthetic population may define group divisions (e.g. states and counties in the U.S.) and contain further realistic locations like workplaces and schools, along with the associations between agents and these places.

3.4.1 data sources

FRED uses a synthetic population to define the agents in a given location. Our default synthetic population was developed by RTI, International. In short, RTI used a proportional iterative fitting method developed in Beckman, et al. (1996) to generate an agent population based on the US Census Bureau's Public Use Microdata files (PUMs) and Census aggregated data. See Wheaton, et al. (2009) for a detailed description. The result is that each agent has a set of socio-demographic characteristics and daily behaviors that include age, sex, employment status, occupation, and household location and membership.

As described on the RTI web site:

"Unlike typical sociodemographic data, the RTI U.S. Synthetic Household Population represents households and persons as dots on a Report—matching high-resolution population distributions with the correct mix of households in each census block group."

The RTI synthetic population is formatted into a FRED synthetic population so it can be used as part of FRED modeling efforts.

3.4.2 process

The default synthetic population is based on the U.S. Census Bureau's Public Use Microdata Samples (PUMS) and aggregated data from the 2005-2009 American Community Survey (ACS) 5-year sample–see `Wheaton (2012)` for a details. This open access database comprises a spatially accurate model of all households, schools, workplaces, and group quarters (e.g. prisons, college dorms, military bases and nursing homes) in the United States. Individual agents are defined and assigned to each household, school, and workplace in the database so that the result closely matches the census-based spatial distributions of households and population sizes at the census block group level, as well as commuting patterns across census-tract boundaries. For agent based models (ABMs) that model specific geographic regions in the U.S., this synthetic population provides an excellent source of spatially accurate population information.

3.4.3 other synthetic populations

Epistemix is actively developing additional synthetic populations, both for the United States and other countries/regions. If you are interested in using FRED in a geographic area outside the United States, please contact us via our website at Epistemix.

CHAPTER

FOUR

CHAPTER 4: CONDITIONS AND STATES

Agents in FRED are subject to a set of conditions that represent the aspects of interest to the model. Each condition consists of a set of states that represent the agent's current status within that condition.

4.1 Conditions

To build a model in FRED, you first need to decide what **conditions** you want to track within the population. Conditions might represent communicable diseases like influenza, economic conditions such as poverty, or behaviors such as drug use or vaccine uptake. Models can include as many conditions as needed. Conditions are composed of **properties** and **states**, with each state containing a set of **rules** for what an agent does when they enter the state and how to proceed from state to state. An agent must be in exactly one state in each condition.

4.1.1 condition blocks

Conditions are defined within a **condition block**, such as:

```
condition INFLUENZA { ... }
```

The block of code within the brackets defines the properties and states of the condition. The complete set of properties and their default values are shown below:

```
condition <condition_name> {

   # initial states
   start_state = Excluded
   meta_start_state = Excluded
   group_start_state = Excluded

   # transmission properties
   transmission_mode = none | proximity | network | environmental
   transmissibility = 0
   exposed_state = none
   transmission_network_name = none

   # output properties
   output = 1
   enable_agent_records = 1
}
```

Note: If an `exposed_state` is specified, then it is required that `transmissibility > 0`. Transmission is described in detail in *Chapter 13*.

Note: All properties must be set using a *literal*, i.e. propeties cannot be set using an expression containing a variable or function.

4.1.2 naming rules and conventions

Each condition within a model must be given a unique name, but state names may appear in more than one condition. The "full name" of a state includes its condition name and is written `<condition_name>.<state_name>`, for example: `INFLUENZA.Susceptible`. Condition and state names must contain only alphanumeric characters and underscores, with no other punctuation or special characters.

By convention, condition names are written in all caps, while state names are in camel case. For example, the `INFLUENZA` condition might have states called `Susceptible`, `HighlyInfectious`, and `Recovered`. While this convention is not enforced, it is considered good pratice to maintain.

4.2 States

States define the meaning of a condition. Within the states of a condition, agents may perform any number of actions, and after spending a specified amount of time within the state, decide on upon its next state within the condition. Along these lines, there are three major categories of rules within states:

1. Action rules
2. Wait rules, and
3. Transition rules.

4.2.1 state blocks

Each state is defined by a **state block** that includes the rules associated with the state:

```
state <state_name> {
   <action_rules>
   <wait_rules>
   <transition_rules>
}
```

4.2.2 rules

Actions, wait rules, and transition rules are described in detail in Chapters *10*, *11*, and *12* respectively. In this section, we provide a brief overview of the role rules play within states.

Note: Categories of rules are always executed in a fixed order: action rules, followed by wait rules, followed by transition rules.

Action rules answers the question: what happens to the agent or what does the agent do when it enters this state? For example, entering a state might change an agent's susceptibility to a transmissible condition if the state contain the action rule:

```
INF.sus = 0.0
```

Wait rules control how long an agent stays in a given state, expressed in hours. For example, an agent might spend two days in a particular state via the following rule:

```
wait(48)
```

Transition rules control how an agent moves from one state to the next. For example, an agent may transition to a new state:

```
next(Infectious)
```

Together, the rules above may be combined to create a "latent" state in a condition, `INF`, representing an infectious disease:

```
state Latent {
   INF.sus = 0.0
   wait(48)
   next(Infectious)
}
```

All rules can be qualified by one or more tests (i.e. *predicates*):

```
[ if (<test1> & <test2> & ... <testN>) then ] <action/wait/transition_rule>
```

The predicate within the brackets is optional; however, if a predicate is present, then the rest of the rule only applies if all the tests are true for a particular agent.

For example, a transition rule may be predicated on an agent's sex:

```
if (sex == female) then next (Susceptible)
```

For convenience, FRED interprets an empty state block such as

```
state <state_name> { }
```

as

```
state <state_name> {
   wait()
   next()
}
```

This means that the given state has no actions, all agents wait indefinitely, and there are no transitions to other states. This configuration is common among **terminal states**.

4.2.3 rule execution order

FRED executes the rules within a state by first sorting them into the following categories:

1. Action rules
2. Wait rules
3. Transition rules

The rules are then processed in order, starting with action rules.

For example, in the state S below:

```
state S {
    wait(10)
    if (age > 16) then next(Dropout)
    join(Group)
}
```

the rules will be interpreted as:

```
state S {
    join(Group)
    wait(10)
    if (age > 16) then next(Dropout)
    default(Stay_in_School)
}
```

and will be executed in the order:

1. `join(Group)`, an action rule
2. `wait(10)`, a wait rule
3. `if (age > 16) then next(Dropout)`, a transition rule

Rules within the action and wait categories are executed in the order that they appear in the code. Transition rules are evaluated in a special way described in *chapter 12*.

4.2.4 the `start_state`

All ordinary agents begin in the indicated `start_state`. You can set `start_state` to any state in the condition, but if `start_state` is not defined explicitly, the default value for start_state will be `Excluded`.

One common pattern is for the `start_state` to exclude agents for whom the condition does not apply. For example, suppose a model includes a `PREGNANCY` condition. This condition does not apply to males, so the model can include rules in the `Start` state to exclude male agents from the condition, as shown below:

```
condition PREGNANCY {
      start_state = Start

      state Start {
```

(continues on next page)

(continued from previous page)

```
        wait(0)
        if (sex == male) then next(Excluded)
        default(S1)
      }

      state S1 {
        ...
      }
      ...
}
```

Note: When individual agents are born in a simulation, they will begin in the `start_state` of each condition.

4.2.5 the `Excluded` state

The `Excluded` state is automatically included in every condition, and cannot be redefined. `Excluded` represents a "does not apply" state. For example, if there is a condition called `PREGNANCY`, then the `start_state` could include a rule that transitions all males to `Excluded`.

Ordinary agents, group agents, and the Meta agent all begin in the `Excluded` state by default. To override this default, specify a state name in the `meta_start_state` for the Meta agent, or `group_start_state` for admin agents:

```
meta_start_state = <some_state>
group_start_state = <some_other_state>
```

If new group agents are created (by creating a new site that has group agents), the new group agent begins in the `group_start_state` for each condition, or to `Excluded` if the condition has no `group_start_state`.

The `Excluded` state does not have any action rules and always assumes `wait()`, that is, an infinite wait time. The above implies that any transition rules included in `Excluded` will be ignored.

Warning: It is legal to transition from `Excluded` to another state through the `set_state()` action or a `send()` action.

CHAPTER

FIVE

CHAPTER 5: VARIABLES

5.1 Variables Types

There are four types of variables that can be defined in a FRED model. Each type is characterized by the kind of data that it is able to store.

- **scalar variables** represent a single numerical value.
- **list variables** represent a list of numerical values.
- **table variables** store key-value pairs with numerical values.
- **list-table variables** store key-value pairs with list values.

Note: All variables store either real numbers or lists of real numbers.

All variables can be set using the assignment operator, =. For example, a scalar variable, `x`, can be set by any agent with an action rule, e.g.

```
x = 42
```

The **expression** on the right side must evaluate to a single real number.

List variables can store a list-valued expression. For example, a list variable, `primes`, can be set by any agent with an action rule, e.g.

```
primes = list(2, 3, 5, 7, 11, 13)
```

Similarly, table and list-table variables store real numbers and lists of real numbers respectively; however, setting and accessing these variables requires the use of a **key**. The keys to tables are real numbers. Expressions that evaluate to a real number may be used. For example, a table could be used to store the factorial of selected integers, e.g.

```
factorial[1] = 1
factorial[2] = 1 * 2
factorial[3] = 1 * 2 * 3
factorial[4] = 1 * 2 * 3 * 4
```

where the keys are integers, and the values are the factorial of that integer.

Warning: Variables are stored as signed 64-bit doubles in FRED, so there is a limit to largest value that can be stored. In the above example, the largest factorial that could be stored is $18!$.

A list-table, `factors`, could be used to store the positive factors for the associated integer key, e.g.

```
factors[1] = list(1)
factors[2] = list(1, 2)
factors[3] = list(1, 3)
factors[4] = list(1, 2, 4)
factors[42] = list(1, 2, 3, 6, 7, 14, 21, 42)
```

where the keys integers, and the values are the list of positive factors associated with each integer.

All variables must be declared within a *variables block*. For example, the variables used in the examples above could be declared as:

```
variables {
   shared numeric x
   shared list primes
   shared table factorial
   shared list_table factors
}
```

5.2 Shared & Agent Variables

Variables can also be categorized by their scope:

- **shared**: all agents share the same value
- **agent**: individual agents maintain independent values

All agents "see" the same value for shared variables. If any agent changes the value of a shared variable, *all* agents will see the new value. Conversely, each agent may set and maintain a different value for agent variables. If a particular agent changes the value of an agent variable, only that agent will have access to that new value. A model may contain any number of shared and agent variables.

Shared variables include single-valued, list-valued, table, and list-table variables. These must be declared in a variables block as:

```
variables {
   shared numeric <variable-name>
   shared list <list-variable-name>
   shared table <table-name>
   shared list_table <list-table-name>
}
```

Agent variables only include single-valued and list-valued variables. These must also be declared in a variables block as:

```
variables {
   agent numeric <variable-name>
   agent list <list-variable-name>
}
```

Note: A common convention is to prepend agent variable names with `my_` in order to make it easy to distinguish variable types within a FRED program. For example, an agent variable called `my_temperature`

might be used to track the internal body temperature of agents.

5.3 The Variables Block

All variables are declared in a **variables block**.

```
variables {
   ...
}
```

Each new variable defined in a model must be declared by type in a variables block:

```
variables {
   shared x
   <x = 0>
   <x.output = 0>

   shared list x_list
   <x_list = list()>
   <x_list.output = 0>

   shared table t
   <t.output = 0>
   <t.default_value = none>

   shared list_table l_t

   agent numeric y
   <y = 0>

   agent list y_list
   <y_list = list()>
}
```

The statements in brackets, <> are optional with the default value indicated. For scalar-valued and list-valued variables, an constant initial value can be specified using the assignment operator, =.

Table variables may be initialized with a default value by setting `<table-name>.default_value = <value>`. The default value will be returned for a key that has not been set. List-tables are always initialized to empty in a variables block. Only an agent may update this variable type within a state block.

> **Warning:** Accessing a table with no default value or a list-table using a key that has not been set will cause a simulation to abort with an appropriate error message.

Output for shared variables may be "turned on" by setting `<variable-name>.output = 1`. For these variables, a time series of the values will be generated and saved to the results. This type of output is not available for agent variables.

More than one variable of a given type may be declared on a single line within a variables block. For example,

```
variables {
   shared numeric x y z
}
```

Warning: When declaring more than one variable on a single line, separate variable names by spaces only. Using commas will result in a compilation error.

Note: If any variable is declared multiple times, only one instance of the variable is created.

A FRED program may contain any number of variables blocks. Variables blocks are combined according to the order in which they appear in the model. For example,

```
variables {
   shared numeric x
   x = 0
}
variables {
   shared numeric y
   y = 0
}
```

is equivalent to:

```
variables {
   shared numeric x
   x = 0
   shaed numeric y
   y = 0
}
```

List-variables are initialized before single-valued variables. Within these categories, variables are initialized in the order in which they are declared. These rules imply that care should be taken when initializing one shared variable based upon another shared variable, since the second variable may not yet have a value when the first variable is set. If you want to initialize one variable based on the initial value of another variable, it may be better to use a **startup block**, as described next.

Finally, the initial values of all shared variables are stored in the results associated with each simulation run.

5.4 Startup Blocks

Prior to a simulation beginning, shared variables may be updated by the Meta agent in a **startup block**:

```
startup {
   ...
}
```

The assignments in a startup block are evaluated *after* shared variables are initialized in a variables block. Unlike variables blocks, the assignment statements in startup blocks may contain arbitrary expressions and are executed by the Meta agent in the order they appear within the program.

For example, if a model parameter is not well constrained, a modeler may set its value *stochastically* in order to capture the uncertainty in its initial value on the outcome of a model, e.g.

```
variables {
   shared numeric x
}

startup {
   x = normal(0, 1)
}
```

After the Meta agent executes the statements in the **startup block**, all group agents execute the statements in the **group_startup** block. For example, if a place `Hospital` has group agents, we could initialize the number of beds associaterd with hospitals as follows:

```
variables {
   agent numeric beds
}

group_startup {
   if (is_group_agent(Hospital)) then beds = uniform(100, 500)
}
```

After the **group_startup** block is completed, any statements in the **agent_startup** block are evaluated for each agent. For example:

```
variables {
   agent numeric temperature
}

agent_startup {
   temperature = normal(98.6, 1.0)
}
```

will assign to each agent a distinct temperature drawn from a normal distribution with a mean of 98.6 and a standard deviation of 1.0.

5.5 Local Variables

Condition can include one or more variables blocks that can declare any types of variable described apreviously in this chapter, including both `shared` and `agent` variables. Variables that are declared in a `variables` nested within a `condition` block are called local variables. Consider this example:

```
condition COND {

   variables {
      shared numeric x
   }
...
}
```

Declaring a variable called `x` within condition `COND` creates a variable with name `COND.x`. Within the declaring condition, local variables can be referenced use the short name, e.g. `x` will refer to the variable

with the full name `COND.x`.

Local variables provide a condition-based name space, but they are still accessible outside the declaring condition. In the example above. any statement outside the condition `COND` may refer to that variable using the full name `COND.x`.

If there's a variable `x` defined in a non-condition `variables` block, then any statement may refer to that variable using the notation `FRED.x`. This notation can be used to access the non-local variable `x` in a condition that has also declared `x` in its local variables block.

CHAPTER

SIX

CHAPTER 6: READ-ONLY VARIABLES

FRED includes several read-only variables that cannot be directly modified by an agent. Read-only variables can be either shared or agent-specific in scope.

6.1 Shared Read-Only Variables

Shared read-only variables will evaluate to the same quantity for any agent at a given time step. These read-only variables fall into the following categories:

6.1.1 simulation run

- `sim_run` – the simulation run number
- `now` – the current simulation step in hours, starting with 0
- `sim_day` – the simulation day, starting with day 0

6.1.2 date and time

- `hour` – the current hour of the day, in the range $[0, 23]$.
- `day_of_week` – the current day of the week, in the range $[0, 6]$. The days of the week are numbered 0 through 6 starting with Sunday.
- `day_of_month` – the current day of the month, in the range $[1, 31]$
- `day_of_year` – the current day of the year, in the range $[1, 366]$. On January 1st, `day_of_year` is equal to `1`. On non-leap years, December 31st, `day_of_year` is equal to `365`.
- `month` – the current month, in the range $[1, 12]$. The months are numbered 1 through 12 starting with January.
- `year` – the current four digit year.
- `epi_week` – the current epidemiological week of the year, in the range $[1, 53]$.

  Note: An epidemiological week, i.e. epi-week, contains seven days, beginning with Sunday and ending with Saturday. By definition, the end of the first week of the year must fall at least four days into the year. Weeks are numbered 1 to 53, although most years consist of only 52 epi-weeks.

- `epi_year` – the current four digit epidemiological year.

- `today` – the current **datestamp**.

Note: A datestamp is an expression that evaluates to a number of the form `YYYYMMDD`. The formula for a datestamp is:

```
my_datestamp = 10000 * year + 100 * month + day_of_month
```

- `date` – a synonym for `today`.

6.1.3 conditions

- `<condition_name>.<state_name>` – the integer index of the state in the order in which the state is defined within the condition.

6.2 Agent Read-Only Variables

Agent read-only variables may evaluate to different quantities depending on the agent. These read-only variables fall into the following categories:

6.2.1 demographics

- `id` – evaluates to an agent's unique integer ID.
- `age` – evaluates to the integer part of an agent's age.
- `birth_year` – evaluates to the four digit year of the agent's birth.
- `age_in_days` – evaluates to integer number of days since an agent's assigned the agent's assigned birthdate.
- `age_in_years` – evaluates to an agent's age in years computed as `age_in_days / 365.25)`.
- `real_age` – a synonym for `age_in_years`.

6.2.2 groups

- `<group_name>` – evaluates to the agent's mixing group ID for the given group type name or `-1` if no such group exists.

CHAPTER

SEVEN

CHAPTER 7: EXPRESSION AND FUNCTIONS

7.1 Expressions

FRED models include **expressions** that evaluate to either a single numerical value or a lists of values.

Expressions are always evaluated with respect to a single agent, called the evaluating agent. The evaluating agent may be an ordinary agent, a group agent, or the Meta agent.

Expressions may include any read-only variable described in *Chapter 6*. In addition, expressions can include:

- *Numerical Operators*
- *Mathematical Functions*
- *Statistical Distribution Functions*
- *Functions on Lists*
- *Functions on Tables*
- *Date & Time Functions*
- *Functions on Conditions*
- *Functions on Groups*
- *Functions on Other Agents*

7.1.1 single-valued expressions

Single-valued expressions are expressions that evaluate to a numerical value. FRED includes both shared and agent variables, each capable of storing the result of a single-valued expression. Agent variables may have distinct values for eaxch agent. Shared variables are shared by all agents–each agent "sees" the same value. A shared table variable may also store single valued expressions.

An example of a single-valued expression:

```
my_eligibility_to_vote = gte(age, 18)
```

In this example, an agent is setting the `my_eligibility_to_vote` variable to `1` if the agent is at least 18 years old and `0` otherwise. In this expression, the variable `age` is evaluated with respect to the agent evaluating the expression.

The mathematical operators `+`, `-`, `*`, `/` , and `%` will produce single-valued expressions if both arguments are single-valued:

- `<expression> + <expression>` : returns the sum of the expressions.
- `<expression> - <expression>` : returns the difference of the expressions.
- `<expression> * <expression>` : returns the product of the expressions.
- `<expression> / <expression>` : returns the quotient of the expressions.
- `<expression> % <expression>` : returns modulus of expressions.

The following examples may help clarify how the mathematical operators act on single-valued expressions:

```
x = 1 + 2  # 3
x = 2 - 1  # 1
x = 2 * 3  # 6
x = 6 / 3  # 2
x = 6 % 5  # 1
```

Finally, accessing an element of a table will result in a single-valued expression:

- `<table>[key]` : returns a value associated with a key from a table.

7.1.2 list-valued expressions

List-valued expressions are expressions that evaluate to lists of numerical values. FRED includes both shared and agent list-variables, each capable of storing the result of a list-valued expression. Agent list-variables are agent-specific lists of values, while shared list-variables are accessible by all agents.

An example of a list-expression:

```
my_housemates = members(Household)
```

In this example, an agent is setting a list-variable, `my_housemates`, to a list of all agents who are members of the evaluating agent's household. The `members()` function returns a list of agent ID numbers.

A list-valued expression may consist of a simple list, e.g. `list(1, 2, 3, 4)`. A list may be composed of either numerical values or other lists, for example:

```
old_list = list(1, 2, 3)
new_list = list(4, old_list, 5)
```

The list-variable, `new_list`, contains the list-valued expression `list(4, 1, 2, 3, 5)`. `list()` is an example of a function that returns a list:

- `list(<expression>, <expression>, ..., <expression>)` – returns the list of values of the given expressions.

Note: Each argument may be single-valued or list-valued. The result is always *flattened* into single list.

The mathematical operators +, -, * and / can produce list-valued expressions if either argument is a list:

- `<list_expression> + <expression>` returns a list of values consisting of the sum of the single-valued expression and each of the values in the list.
- `<expression> - <list_expression>` - returns a list of values consisting of the difference of the first expression and each of the values in the list.

- <list_expression> - <expression> - returns a list of values consisting of the difference of each of the values in the list with the value of the second expression.
- <list_expression> - <list_expression> - returns a list of values consisting of the difference of the items in the first list with the corresponding items in the second list. The lists must have equal lengths.
- <list_expression> * <expression> - returns a list of values consisting of the product of the single-valued expression and each of the values in the list.
- <list_expression> * <list_expression> - returns a list of values consisting of the product of the items in the first list with the corresponding items in the second list.
- <expression> / <list_expression> - returns a list of values consisting of the quotient of the first expression and each of the values in the list.
- <list_expression> / <expression>) - returns a list of values consisting of the quotient of each of the values in the list with the value of the second expression.
- <list_expression> / <list_expression> - returns a list of values consisting of the quotient of the items in the first list with the corresponding items in the second list. The lists must have equal lengths.

Note: The `add()`, `sub()`, `mult()` and `div()` functional form of the mathematical operators all take list-valued arguments. If either or both arguments is a list, the result is a list of values.

Note: The addition and multiplication operators, + and *, are *commutative*, i.e reversing the order will result in the same expression.

The following examples may help clarify how the mathematical operators act on lists:

```
x_list = list(1, 2, 3)
y_list = list(10, 20, 30)

# addition
x = 10 + x_list  # list(11, 12, 13)

# subtraction
x = 10 - x_list  # list(9, 8, 7)
x = x_list - 10  # list(-9, -8, -7)
z_list = y_list - x_list  # list(9, 18, 27)

# multiplication
x = x_list * 10  # list(10, 20, 30)
x = 10 * x_list  # list(10, 20, 30)
z_list = y_list * y_list  # list(10, 40, 90)

# division
x = x_list / 2  # list(0.5, 1.0, 1.5)
x = 1.5 / x_list  # list(1.5, 0.75, 0.5)
z_list = y_lixt / x_list  # list(10, 10, 10)
```

The following returns a list of integers:

- `range(start,stop,step)` - returns a list of integers from `start` in increments of `step` up (or down if `step < 0`) to but not including `stop`. If two arguments are given, the `step` is assumed to be 1. If only one argument is given, then `start = 0` and `step = 1`. So `range(5)` returns the list `(0, 1, 2,\ 3, 4)`. It is an error if `step = 0`. The list is empty if `step > 0` and `stop <= start`, or if `step < 0` and `start <= stop`.

Accessing an element of a list-table will result in a list-valued expression:

- `<list-table>[key]` : returns a list associated with a key from a list-table.

Finally, accessing a list of elements in a table will result in a list-valued expression:

- `<table>[<list_expression>]` : returns a list of values associated with a list of keys.

7.1.3 table-valued expressions

Table-valued expressions are expressions that evaluate to a table or list-table consisting of *key-value* pairs. Table variables are capable of storing the result of a table-valued expression.

Tables and list-tables are data structures consisting of *key-value* pairs. For tables, the values are real numbers. For list-tables, the values are lists. All tables are shared variables accessible by all agents.

7.2 Numerical Operators

Expressions may contain the usual numerical operators +, -, *, /, and % for addition, subtraction, multiplication, division, and the modulo operator. Each of these can be expressed either in *infix* or *prefix* notation:

infix	prefix
`x + y`	`add(x, y)`
`x - y`	`sub(x, y)`
`x * y`	`mult(x, y)`
`x / y`	`div(x, y)`
`x % y`	`mod(x, y)`

Note: The numerical operators +, -, *, /, and % are expressed in infix notation, where `A + B` is a synonym for `add(A, B)`, etc. FRED parses expressions into prefix notation prior to execution, so you may see the prefix form in error and warning messages.

Note: The numerical operators +, -, *, and / apply to list-valued arguments as well as mixed single-valued and list-valued arguments.

Warning: Division by zero will cause FRED to abort a simulation with an appropriate error message.

7.3 Input Functions

FRED supports input function that read from files. The following function returns a list of values:

- `read(filename, column)` – returns a list of numeric values in the indicated from the named comma-separated value file. Column numbers begin with `0`. Blank lines in the file are skipped, as are any lines with a non-numeric value in the first column.

7.4 Mathematical Functions

FRED supports a number of mathematical functions:

7.4.1 comparison

- `equal(x, y)` – returns 1 if the arguments are equal, `0` otherwise.
- `eq(x, y)` – synonym for `equal(x, y)`.
- `neq(x, y)` - returns 1 if the arguments are not equal, `0` otherwise.
- `lt(x, y)` - returns 1 if `x` is less than `y`, `0` otherwise.
- `lte(x, y)` - returns 1 if `x` is less than or equal to `y`, `0` otherwise.
- `gt(x, y)` - returns 1 if `x` is greater than `y`, `0` otherwise.
- `gte(x, y)` - returns 1 if `x` is greater than or equal to `y`, `0` otherwise.

7.4.2 rounding and remainder

- `floor(x)` – returns the largest integral value less than or equal to `x`.
- `ceil(x)` – returns the largest integral value greater than or equal to `x`.
- `int(x)` – returns the integer part of `x`. the result is equal to `floor(x)` if `x` is non-negative and `ceil(x)` if `x` is negative.
- `round(x)` – returns the integral value nearest to `x`, rounding half-way cases away from zero.

7.4.3 exponential, logarithmic, and power

- `log(x)` - returns the base e logarithm of `x`, $\log_e(x)$
- `pow(x, y)` - returns the base, `x`, raised to the power exponent, `y`, x^y
- `exp(x)` - returns the base e exponential of `x`, which is e raised to the power `x`, e^x.
- `sqrt(x)` - returns the square root of `x`.

7.4.4 trigonometric

- `sin(x)` – returns the sine of `x`, where `x` is interpreted as radians.
- `cos(x)` – returns the cosine of `x`, where `x` is interpreted as radians.
- `tan(x)` – returns the tangent of `x`, where `x` is interpreted as radians.
- `asin(x)` – returns the principal value of the inverse sine of `x`. The result is in the range $[-\pi/2, \pi/2]$.
- `acos(x)` – returns the principle value of the inverse cosine of `x`. The result is in the range $[0, \pi]$.
- `atan(x)` – returns the principal value of the inverse tangent of `x`. The result is in the range $[-\pi/2, \pi/2]$.
- `atan2(y, x)` – returns the principal value of the inverse tangent of `y/x` using the signs of both arguments to determine the quadrant of the returned value.
- `sinh(x)` – returns the hyperbolic sine of `x`.
- `cosh(x)` – returns the hyperbolic cosine of `x`.
- `tanh(x)` – returns the hyperbolic tangent of `x`.
- `asinh(x)` – returns the inverse hyperbolic sine of `x`.
- `acosh(x)` – returns the principle value of the inverse hyperbolic cosine of `x`. The result is in the range $[0, \infty]$.
- `atanh(x)` – returns the inverse hyperbolic tangent of `x`.

7.4.5 miscellaneous

- `min(x, y)` – returns the smallest of the two values.
- `max(x, y)` – returns the largest of the two values.
- `abs(x)` – returns the absolute value of `x`.
- `nprob(p, n)` – returns the probability, math:q, such that, when an agent makes `n` repeated independent choices with probability of success :math`q`, the total probability of success is `p`. This is equivalent to the expression `1- pow(1 - p, 1 / n)`.
- `distance(lat1, lon1, lat2, lon2)` – returns the distance in km between the points, (`lat1`, `lon1`) and (`lat2`, `lon2`), on the surface of the Earth.

Warning: Illegal operations using mathematical functions will cause FRED to abort a simulation with an appropriate error message. Examples of illegal operations include taking the logarithm of a negative number or taking the inverse sine of a number outside the range [-1,1].

7.5 Statistical Distribution Functions

FRED provides the following functions that return draws from statistical distributions:

- `bernoulli(p)` – returns `1` with probability `p`, otherwise `0`.
- `binomial(t, p)` – returns an integer according to a binomial discrete distribution. This distribution produces random integers in the range $[0, t]$, where each value represents the number of successes in a sequence of `t` trials (each with a probability of success equal to `p`).
- `cauchy(a, b)` – returns a floating point number according to a Cauchy distribution with a median `a` and shape parameter, `b`. This is equivalent to a Student-t distribution with one degree of freedom.
- `chi_squared(n)` – returns a floating point number according to a chi-squared distribution. This distribution produces random numbers as if the square of `n` independent standard normal random variables with $\mu = 0$ and $\sigma = 1$, were aggregated, where `n` is the distribution parameter, known as degrees of freedom.
- `exponential()` – returns a floating point number according to an exponential distribution. This distribution produces random numbers where each value represents the interval between two random events that are independent but statistically defined by a constant average rate of occurrence, .
- `extreme_value(a, b)` – returns a floating point number according to a Type I extreme value distribution. This distribution produces random numbers where each value can be interpreted as the extreme (maximum or minimum) of a number of samples of a random variable.
- `fisher_f(m, n)` – returns a floating point number according to a Fisher-F distribution. This distribution produces random numbers as the result of dividing two independent Chi-squared distributions of `m` with `n` degrees of freedom.
- `gamma(, )` – returns a floating point number according to a Gamma distribution. This distribution can be interpreted as the aggregation of exponential distributions, each with as parameter. It is often used to model waiting times.
- `gompertz(shape, scale)` – returns a floating point random number according to a Gompertz distribution.
- `lognormal(median, dispersion)` – returns a floating point random number according to a log-normal distribution with the given `median` and `dispersion`.

Note: The default parameterization is the "multiplicative", also known as the "geometric parameterization", i.e., median = exp() and dispersion = exp().

- `negative_binomial(k, p)` – returns a random integer according to a negative binomial discrete distribution. This distribution produces random integers where each value represents the number of successful trials before `k` unsuccessful trials happen in a sequence of trials, each with a probability of success equal to `p`.
- `normal(, )` – returns a floating point random number from a normal distribution with a given and standard deviation, .
- `poisson()` – returns an integer value drawn from a Poisson distribution with a mean .
- `student_t(n)` – returns a floating point random number according to a Student-T distribution with `n` degrees of freedom. This distribution produces random numbers as the result of normalizing the values of a relatively small sample ($n+1$ values) of independent normally-distributed values. As the sample size increases, the distribution approaches a standard normal distribution.
- `uniform(l, u)` – returns a uniform random number in range [`l`, `u`).

- `weibull(a, b)` – returns a floating point random number according to a 2-parameter Weibull distribution with a probability density function of:

$$p(x|a,b) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} e^{-\left(\frac{x}{b}\right)^a}$$

7.6 Date and Time Functions

FRED operates in steps each representing one hour of simulated time. The functions in this section are often used in connection with *wait rules* to control the amount of time an agent spends in a given state.

The following function returns the simulation step corresponding to a given date and hour:

- `sim_step(year, month, day, hour)` – returns the simulation time step corresponding to the given date and hour. For example, `sim_step(2020, 6, 16, 19)` returns the simulation time step corresponding to June 6, 2020 at 7pm.

Note: `sim_step(year, month, day)` is equivalent to `sim_step(year, month, day, 0)`.

Often this function is used in combination with wait rules. For example, the following `wait` rule would cause an agent to wait until the simulation time step corresponding to Aug 12 at 2 a.m. during the year represented by the variable `Yr`:

```
wait(sim_step(Yr, 8, 12, 2) - now)
```

The `until()` function returns the number of hours until the specified date and time:

- `until(year, month, day_of_month, hour)`

For example, the following `wait` rule would wait until 2020-Dec-31 at 11 p.m.

```
wait(until(2020, 12, 31, 23))
```

The `until()` function understands symbolic terms for months and hours, so the above could also be written as:

```
wait(until(2020, Dec, 31, 11pm))
```

If three arguments are given of the form `until(month, day, hour)`, where the first argument is the symbolic name of the month, the function returns the number of hours until the next occurrence of the given date and time.

For example, the following makes an agent wait until 11 p.m. on the next New Year's Eve:

```
wait(until(Dec, 31, 11pm))
```

The `until()` function also accepts two arguments, as long as the first argument is a symbolic name for a month or a day of the week, for example:

```
wait(until(Sat, 11pm))
wait(until(Dec, 31))
```

In the latter case, the hour is assumed to be `12am`.

If a single argument is given consisting of a number following by `am` or `pm`, `until()` returns the number of hours until the next occurrence of the indicated time, for example:

```
wait(until(1pm))
```

Finally, if `until()` is given a single argument not in the format `<number>am` or `<number>pm`, the argument is interpreted as a **timestamp** or a **datestamp**:

- A **timestamp** is an expression that evaluates to a number of the form `YYYYMMDDHH`, where `YYYY` represents the year, `MM` represents the month (with values `01, ..., 12`), and `HH` represents the hour on a 24-hour clock (with values `00, ..., 23`.

You can construct a timestamp using the the formula:

```
my_timestamp = 1000000 * year + 10000 * month + 100 * day_of_month + hour
```

- A **datestamp** is an expression that evaluates to a number of the form `YYYYMMDD`.

The formula for a datestamp is:

```
my_datestamp = 10000 * year + 100 * month + day_of_month
```

For example, we might have a variable `next_change` whose current value is `20201104` (a valid datestamp). Then the following makes an agent wait until 2020-Nov-04 at 12 a.m.:

```
wait(until(next_change))
```

If the current value of `next_change` is the timestamp `2020110413`, then the following makes an agent wait until 2020-Nov-04 at 1 p.m.:

```
wait(until(next_change))
```

If the expression does not evaluate to a valid timestamp or datestamp, a run-time error will occur and the simulation will be terminated.

The following function is useful for finding the time between two specified timestamps:

- `steps_between(<timestamp1>, <timestamp2>)` returns the number of time steps between the two timestamps.
  - The result is positive if `<timestamp1>` occurs before `<timestamp2>`.
  - If the timestamps are equal, `steps_between()` returns 0.
  - If either timestamp evaluates to a datestamp value of the form `YYYYMMDD`, it is interpreted as `YYYYMMDDHH` where `HH = 00`.
  - An error is thrown if either timestamp is not valid.

The following function extract the components of a timestamp:

- `get_year_from_timestamp(<timestamp>)` returns the value of the `YYYY` portion of the timestamp.
- `get_month_from_timestamp(<timestamp>)` returns the value of the `MM` portion of the timestamp.
- `get_day_from_timestamp(<timestamp>)` returns the value of the `DD` portion of the timestamp.
- `get_hour_from_timestamp(<timestamp>)` returns the value of the `HH` portion of the timestamp.

Timestamps are correctly computed for 100 years before the `start_date` to 100 years after the `end_date`. Outside this range, the results are unpredictable.

7.7 Functions on Conditions

- `current_count(<condition_name>.<state_name>, <group_name>)` - returns the number of agents currently in the given state. The second argument is optional. If a group name is provided, the count is restricted to agents that share the given group with the agent.
- `current_state(<condition_name>)` - returns the current state of the agent in the given condition.
- `daily_count(<condition_name>.<state_name>, <group_name>)` - returns the number of agents that entered the given state on the previous day. The second argument is optional. If a group name is provided, the count is restricted to agents that share the given group with the agent.
- `source(<condition_name>.<state_name>, <group_name>)` - returns the agent ID of the transmission source for the given condition.
- `transmissions(<condition_name>.<state_name>, <group_name>)` - returns the number of other agents that have been exposed to the given condition by the current agent. The result is *0* if the agent has not transmitted the condition.
- `total_count(<condition_name>.<state_name>, <group_name>)` - returns the total number of agents that have ever entered the given state. The second argument is optional. If a group name is provided, the count is restricted to agents that share the given group with the agent.
- `prev_state(<condition_name>)` - returns the agent's previous state before transitioning to the current state.

Note: The previous transition may have been from the current state.

- `sus_list(<condition_name>)` - returns the list of agents that are susceptible to a condition, i.e. agents who have a susceptibility greater than `0.0`.

7.8 Functions on Groups

- `get_group_id(<group_name>)` - returns the group ID of the named group, or `0` if the calling agent is not a member of such a group.

7.8.1 associated agents

- `size(Population)` - returns the number of ordinary agents in the population.
- **`size(<group>)` - returns the number of**
  members in the ordinary agent's given group. Returns `0` if the agent has no such group.
- `get_size()` - returns the number of members in a group agent's group. Error occurs if called by a non-group agent.
- `members(<group>, <group>, ..., <group>)` - returns a list of all agents who share one or more of the groups with the agent.

For example,

```
new_list = members(Household, Classroom)

returns a list of all the agents who are in the agent's household or
```

(continues on next page)

(continued from previous page)

```
classroom, or both. Each agent appears at most once in the resulting list.
The result is sorted by agent ID numbers.
```

- `get_group_agents(<group_name>)` - returns a list of IDs for the group agents associated with the sites of the given group type. If the group type does not have group agents, the list will be empty.
- `get_group_agent(<group_name>)` - returns the group agent ID if the evaluating agent is a member of the named group and the group agent exists. Otherwise, returns `0`.
- `get_group_agent(<group_ID>)` - returns the group agent ID of the group with the given ID, or `0` if the group does not exist or the group has no group agent.

7.8.2 geographical

The following functions return information on the location or relative location of one or more places:

- `latitude(<place_name>)` – latitude of the agent's place. If the argument is omitted, return the latitude of the agent itself.
- `longitude(<place_name>)` – longitude of the agent's place. If the argument is omitted, return the longitude of the agent itself.
- `elevation(<place_name>)` - returns the height in meters above sea level of the agent's place. If the argument is omitted, return the elevation of the agent itself.
- `getx(<place_name>)` – x-coordinate of the agent's place (meters from western edge in simulation area). If the argument is omitted, return the x-coordinate of the agent itself.
- `gety(<place_name>)` – y-coordinate of the agent's place (meters from southern edge of simulation area). If the argument is omitted, return the y-coordinate of the agent itself.
- `dist(<place_name1>, <place_name2>)` – returns the distance in km between the places whose IDs are represented by the arguments.

7.8.3 place containers

- `get_container(<container_type>, <place_name>)` : returns the place ID of the container of the type in the first argument that contains the agent's place of the type in the second argument.

7.8.4 on networks

- `links(Network)` returns a list of agents to which the agent is directly connected by edges in the given undirected network.
- `outlinks(Network)` returns a list of agents to which the agent is directly connected by outward edges in the given directed network.
- `inlinks(Network)` returns a list of agents that have edges to the agent in the given directed network.

7.9 Functions on Other Agents

- `get_population()` – returns a list of IDs of all oridinary agents in the population.
- `ask(<expression1>, <expression2>)` – If the first argument evaluates to the ID of another agent, the function returns the value of the second argument, evaluated with respect to the other agent. If the first argument evaluates to a list of agent IDs, the function returns a list of values of the second expression, evaluated with respect to each of the other agents. If the first argument is the name of a group (such as `School`), then the other agent is the group agent for the primary agent's group of that type. If there is no such group agent, the result of the function is 0.

  The following sets the value of `sibling_age` to the age of the agent whose ID is stored in `my_sibling`:

```
sibling_age = ask(my_sibling, age)
```

  The following sets `my_household_ages_list` to a list of the ages of the members of the agent's household:

```
my_household_ages_list = ask(members(Household), age)
```

  Suppose the evaluating agent is a `member` of a group called `Theater`. Then the following sets `available_tickets` to the value of the `tickets` agent variable for the group agent for the agent's `Theater`:

```
available_tickets = ask(Theater, tickets)
```

7.10 Functions on Lists

7.10.1 selecting an element

An element in a list may be accessed using square brackets, `<list_variable>[<expression>]`. The expressions is truncated to an integer, and used as an index into the list. If the element does not exist, `0.0` is returned. Equivalently, this can be accomplished with the `select()` function.

Note: The parser translates `X[string]` to `select(X, string)`, so the following are equivalent:

- `select(x_list, index_list)` and `x_list[index_list]`
- `select(x_list, _ > 0)` and `x_list[_ > 0]` - returns all positive numbers in `x_list`
- `select(x_list, ask(_, age) > 18)` and `x_list[ask(_, age) > 18]` - returns the list of agents in `x_list` that have `age > 18`.

A single element can be selected from a list by the following functions:

- `select(<list-expression>, <expression>)` – returns the item in the value of the first argument indexed by the second argument.
- `last(<list_expression>)` – returns the last item in the list. Throws an exception if the list is empty.

The `select` function can also be used to select a sub-list:

- `select(<list-expression>, <list_expression>)` - returns the sub-list of the first argument corresponding to the list of indices in the second argument.
- `find_index(<value>, <list_expression>)` – returns the index of the `value` in the `list_expression`, or -1 if the value does not occur in the list.

Elements in a list may also be selected based on a test:

- `select(<list-expression>, <test>)` - returns the sub-list of the first argument consisting of the items that pass the test. The special variable _ in the test refers to the list item being evaluated. For example `select(list(20, 5, 2, 40), (10 < _))` would return the list `(20, 40)`.
- `select_index(<list-expression>, <test>)` - returns the list of the indices for the list in the first argument corresponding to items that pass the test. The special variable _ in the test refers to the list item being evaluated. For example `select_index(list(20, 5, 2, 40), (10 < _))` would return the list `(0, 3)`.

examples

You can access the first or last element in a list `X`:

```
first_item = X[0]
last_item = X[length(X) - 1]
```

or select a member of an agent's household:

```
first_housemate = select(members(Household), 0)
```

- `unique(<list-expression>)` - Returns a sorted list of unique elements that occur in the list parameter.

7.10.2 miscellaneous

- `length(<list_expression>)` – returns the number of items in the value of the list-expression.
- `max(<list_expression>)` – returns the maximum of values in the list. Returns `0` if list is empty.
- `min(<list_expression>)` – returns the minimum of values in the list. Returns `0` if list is empty.
- `percentile(<value>, <list_epression>)` - returns the percentile of a value within a list of sorted list of values.
- `prod(<list_expression>)` – returns the product of values in the list. Returns `0` if list is empty.
- `sum(<list_expression>)` – returns the sum of values in the list. Returns `0` if list is empty.
- `partial_sums(<list_expression>)` - returns the list of partial sums of the values in the list. That is, each value in the resulting list consists of the sum of all items up to an including the current item.
- `unique(<list_expression>)` - returns a list of unique elements from the list.

examples

The following example sets a variable, `mean_X`, equal to the average of a list `X`:

```
if (length(X) > 0) then mean_X = sum(X) / length(X)
```

7.10.3 filtering

Filtering a list entails producing a sub-list from a list that matches some criteria. There are several functions that perform filtering. The following functions act on any list:

- `filter_values(<list_expr>, <op>, <expr>)` - returns a sub-list of the values that satisfy the test of the given operator against the expression, where <op> is <, <=, =, ==, >, >=, or !=.
- `index_values(<list_expr>, <op>, <expr>)` returns the sub-list of values that satisfy the given test.
- `filter_by_index(<list_expr>, <list_of_desired_indexes>)` returns the selected items from a list given the list of desired index values.

Another set of functions specifically act on a list of agent IDs:

- `filter_agents(<list-expression>, <test1>, <test2>, ..., <testN>)` - returns a list of all agents in the list-expression that satisfy all the tests in the remaining arguments.
- `index_agents(<list-expression>, <test1>, ..., <testN>)` - returns a list of indexes of agents on the list that satisfy all the tests.

Note: The <list-expression> argument in the preceding two functions **must** be a list of agent IDs.

---

examples

For example, a list can be filtered based on the value of it's elements:

```
old_list = list(2, 10, 8, 4, 6)
new_list = filter_values(old_list, >, old_list[0] + 3)
```

The resulting list, `new_list`, is equal to `list(10, 8, 6)`.

Similarly, the indices that produce this filtering can be recovered:

```
old_list = list(2, 10, 8, 4, 6)
index_list = index_values(old_list, >, old_list[0] + 3)
new_list = filter_by_index(old_list, index_list)
```

The resulting list, `index_list`, is equal to `list(1, 2, 4)`. That list of indices can be used to generate the filtered list, `new_list`, again equal to `list(10, 8, 6)`.

For example, to generate a list of all the agents who are in an agent's school, older than the agent, and are of the opposite sex:

```
my_older_opposite_sex_friends = filter_agents(members(School),\
   age < ask(_,age), sex != ask(_,sex))
```

Note: The special variable _ is replaced by the current item in the list of agent being processed. For example, in the first test in the example above, the read-only variable `age` refers to the age of the agent evaluating the expression, and the expression `ask(_,age)` returns to the age of the agent being evaluated by the filter, that is the current item in the list `member(Schhol)`.

7.10.4 sampling

The following functions return a random sample from a list. Each call to the function returns a distinct randomized sample.

- `sample_without_replacement(<list-expr>, <expr>)` - returns a random sample of the list of size `k` where `k` is the value of second expression. Sampling is without replacement, meaning that no item is selected more than once.
- `sample_without_replacement(<list-expr>, <expr>, <list-expr2>)` - returns a random sample of the list of size `k` where `k` is the value of second expression. Sampling uses the weights specified in the last argument, which must evaluate to a list of positive values with the same length as the first argument. Sampling is without replacement meaning that no item is selected more than once.
- `sample_with_replacement(<list-expr>, <expr>)` - returns a random sample of the list of size `k` where `k` is the value of second expression. Sampling is performed with replacement, meaning that items may be selected more than once.
- `sample_with_replacement(<list-expr>, <expr>, <list-expr2>)` - returns a random sample of the list of size `k` where `k` is the value of second expression. Sampling uses the weights specified in the last argument, which must evaluate to a list of positive values with the same length as the first argument. Sampling is performed with replacement meaning that items may be selected more than once.

7.10.5 set operations

The following functions implement set union, intersection, and difference operations:

- `union(<list_expr>, <list_expr>)` - returns a list of unique items that occur in either list, sorted in increasing order.
- `intersection(<list_expr>, <list_expr>)` - returns a list of unique items that occur in both lists, sorted in increasing order.
- `set_difference(<list-expr1>, <list-expr2>)` - returns the list of unique items that are in the first list but not in the second list.

7.10.6 sorting

The following functions can be used to reorder a list:

- `sort(<list_expr>)` - returns a sorted list.
- `arg_sort(<list_expr>)` - returns a list of indices that sort the list.
- `shuffle(<list_expr>)` - returns a randomly sorted copy of the list.

7.10.7 evaluating an expression on each item

- `apply(<list_expr>, <expression>)` - returns a list of the values of the expression in the second argument, evaluated with respect to each item in the first argument. Any occurrence of _ in the expression refers to the value of the current item in the list.

examples

For example, a new list can be generated by multiplying every element in a list by `2.0`:

```
x_list = list(1, 2, 3, 4, 5)
y_list = apply(x_list, 2 * _)
# result: y_list = list(2, 4, 6, 8, 10)
```

7.11 Functions on Tables

Tables and list-tables are data structures consisting of key-value pairs. For tables, the values are real numbers. For list-tables, the values are lists. A value in a table or list-table may be selected using square brackets, `<table-name>[<key>]`. Equivalently, this can be accomplished using the `lookup()` function.

A specific position in a list-table can be accessed using two values in square brackets. For example, if `x` is a list-table and the list associated with key `key` has three values in the list `x[key]`, then `x[key][2]` refers to the third value in the list. In the following example, we set the third item in the list to the first item in the list plus 1:

```
x[key][2] = x[key][0] + 1
```

Functions on tables and list-tables include:

- `lookup(<table_name>, <key-expression>)` - returns the value in the table/list-table associated with the given key.

  Note: If the key does not exist, a run-time error occurs unless the table has a defined `default_value` property, in which case the default value is returned. The default value is **not** inserted into the table as the value of the key.

- `length(<table_name>)` – returns the number of `key-value` pairs in the table/list-table.
- `get_keys(<table_name>)` - returns the list of keys stored in the indicated table/list-table.
- `get_values(<table_name>)` - returns the list of values stored in the indicated table.

CHAPTER

EIGHT

CHAPTER 8: PREDICATES

Rule statements can be qualified by one or more tests. Such tests are referred to as **predicates**. Predicates are true/false assertions that are evaluated by an agent within a **state**. Predicates take the form:

```
if (<test1> & <test2> & ... & <testN>) then <rule>
```

In the above example, the predicate is the collection of tests within the parentheses following the `if` keyword. All predicates must be preceded with `if`. Multiple tests are combined using the logical `&`. When multiple tests are present, the if statement will evaluate to true if and only if all the tests evaluate to true.

Note: Predicates may also be treated as expressions that return `1` or `0` if true or false respectively.

Any predicate may be negated by a `not()`:

- `not(<test>)` – true if `<test>` is false, and false if `<test>` is true.

For example,

```
if (not(<test1> & <test2> & ... & <testN>)) then <rule>
```

will result in an agent executing the rule if any of the tests are false.

8.1 Numeric Predicates

Predicates can use any of the *comparison functions*. When used in a predicate, the infix form of the comparison functions may be used instead:

infix	functional
`x == y`	`eq(x, y)`
`x != y`	`neq(x, y)`
`x < y`	`lt(x, y)`
`x <= y`	`lte(x, y)`
`x > y`	`gt(x, y)`
`x >= y`	`gte(x, y)`

Warning: Using the infix form of a comparison outside of a predicate is not allowed. For example,

```
x = x + gt(age, 10)  # adds 1 to x if age > 10
x = x + (age > 10)  # will result in a fred_compiler error message.
```

In addition to the two-way predicates above, there is a three-way numeric predicate:

- `is_in_range(<expression>, <lower_bound>, <upper_bound>)` – evaluates to true if the value of the expression is between the lower and upper bounds, inclusive.

For example, the following is true if an agent's current age is at least 15 and no greater than 20:

```
is_in_range(age, 15, 20)
```

8.2 Predicates on Lists

The following predicate can be used to check for the presence of a value in a list:

- `is_in_list(<list-expression>, <expression>)` – true if the value of the list expression has an item whose value is the second argument.

For example, the following rule would apply to any agent whose ID is in the list, `Agent_List`:

```
if (is_in_list(Agent_List, id)) then ...
```

8.3 Predicates on Agents

The following predicates test which type of *agent* is evaluating the statement:

- `is_group_agent(<group>)` – evaluates to true if the evaluating agent represents a group of the given type.
- `is_meta_agent()` – true for the Meta agent only.

8.4 Predicates on Conditions

A predicate may depend on an agent's state within a condition using the `current_state()` function. For example, the following predicate tests whether an agent is in the `Exposed` state in the `INF` condition:

```
if (current_state(INF) == INF.Exposed) then ...
```

For transmissible conditions, the following predicates depend on the exposure history of an egent:

- `was_exposed_in(<condition>, <group>)` – true if the agent was exposed to the given condition in the given group.
- `was_exposed_externally(<condition>)` – evaluates to true if the agent was exposed to the given condition by an importation event.

8.5 Predicates on Mixing Groups

The following predicates depend on an agents state with respect to a mixing group:

- `is_member(<group>)` – evaluates to true if the agent is a member of the given group type.
- `is_skipping(<group>)` – true if the agent is a member of the specified group type and is temporarily not attending the group as a result of an `skip()` action.
- `is_at(<group>)` – true if the agent is currently attending the specified group type, e.g. `is_at(School)` is true during times when the agent's school is open and agent is actually present at the school.
- `is_open(<group>)` – true if the agent is a member of a group of the given group type and the group is currently open. That is, the current hour is among the usual business hours for the group type and the group has not been temporarily closed by the group agent.
- `is_group_open()` – this is only permiited for group agents and is true if the agent's group is currently open. That is, the current hour is among the usual business hours for the group type and the group has not been temporarily closed by the group agent.
- `is_temporarily_closed(<group>)` – true if the agent is a member of a group of the given type and the group has been temporarily closed by the group agent.
- `was_exposed_in(<condition>, <group>)` – true if the agent was exposed to the given condition in the given group.

The following predicates are specific to networks:

- `is_connected_to(<agent_id>, <network>)` – true if the agent is connected to the agent with agent_id in the given network.
- `is_connected_from(<agent_id>, <network>)` – true if the agent has a connection from the agent with agent_id in the given network.

8.6 Predicates on the Date

There are several ways to test the current simulated date in FRED. One way to test the date is to use the `today` factor in a comparison. Recall that `today` evaluates to a datestamp integer representing the current simulation date in the format `YYYYMMDD`. For example, the following predicate would be satisfied if the simulation date is 2020-Dec-15:

```
if (today == 20201215) then ...
```

The following predicate would be satisfied on any day before 2021-Mar-13:

```
if (today < 20210313) then ...
```

In the following example, either predicate would be true on any day between 2020-Dec-30 and 2021-May-28, inclusive:

```
variables {
    global beginDate endDate
    beginDate = 20201230
    endDate = 20210528
}
```

(continues on next page)

(continued from previous page)

```
...

<COND>.<State> {
   ...
   if (beginDate <= today & today <= endDate) then ...
   if (is_in_range(today, beginDate, endDate)) then ...
   ...
}
```

You can also test the individual values of `year`, `month`, and `day_of_month` in a predicate. The following predicates are equivalent:

```
if (year==2020 & month == 5 & day_of_month <= 15) ...
if (year==2020 & month == May & day_of_month <= 15) ...
```

Note: FRED treats the three-letter abbreviation of a month as an integer between 1 (for `Jan`) and 12 (for `Dec`).

The following predicate tests whether the current simulation dates falls between two dates expressed as month and day-of-month:

- `is_date_in_range(<date1>,<date2>)` – evaluates to true if the current dates is between the dates, taking into account the wrap-around nature of the calendar. Each date is a fixed string of the form `MMM-DD` where `MMM` is a three-letter abbreviation for the month.

For example, the following predicate is true if the current simulation date is between December 10 and January 1, including those dates:

```
if (is_date_in_range(Dec-10,Jan-01)) ...
```

8.7 Predicates on Files

The following predicate can be used to test is a file ahs been opened:

- `is_file_open(<filename>)` – evaluates to true if the indicated filename has been opened using the `open_csv(<filename>)` action.

CHAPTER

NINE

CHAPTER 9: MIXING GROUPS

FRED agents interact with each other in **mixing groups**. Examples of mixing groups include households, schools, workplaces, and friendship networks. Some mixing groups represent specific locations (household, schools) and some do not (friendship networks). This chapter deals with **places**, which are mixing groups that have a specific geo-location, and **networks**, which represent relationships among agents but are not associated with a specific location.

9.1 Declaring a Mixing Group

To declare a mixing group in FRED, use one of the following statement blocks:

```
place <place_name> { ... }
```

```
network <network_name> { ... }
```

The code within the brackets defined the properties of mixing groups of the named type. The convention is to use a capitalized, singular name such as School or Workplace. For example, a model may include a new class of places representing pharmacies by including a place block as follows:

```
place Pharmacy { ... }
```

9.2 Places

Places are mixing groups that have a specific geo-location (latitude, longitude, elevation) as well as other properties. The user can declare any number places in FRED. Each place definition represents a class of individual locations, called **sites**.

The complete set of properties for a place is shown below. Each property has a default value as shown, so when defining a type of Place in a FRED model, you only need to set properties that do not use the default values. The order of the properties does not matter unless explicitly mentioned below.

```
place <place_name> {

   # Administrative Properties
   has_group_agent = 0

   # Transmission Properties
   contact_prob = 0
```

(continues on next page)

(continued from previous page)

```
    contact_rate = 0
    same_age_bias = 0
    contact_prob_for_condition_name> = -1
    contact_rate_for_<condition_name> = -1
    same_age_bias_for_<condition_name> = -1
}
```

The properties fall into two categories:

- The **Administrative property**: `has_group_agent` generates a group agent for each site if set.
- **Transmission properties** concern the spread of conditions from one agent to another within the type of mixing group. Transmission Properties are described in *Chapter 13*.

9.2.1 Places in Synthetic Populations

A synthetic population may define a set of places that are available to any model using that synthetic population. The default synthetic population includes these places:

```
place Household { ... }
place School { ... }
place Grade { ... }
place Workplace { ... }
place Block_Group { ... }
place Census_Tract { ... }
place County { ... }
place State { ... }
place Prison { ... }
place Nursing_Home { ... }
place College_Dorm { ... }
place Barracks { ... }
```

Other places can be added by the user.

By default, all of the initial agents are assigned membership in the appropriate default places according to the input files in the synthetic population. These input files gives the latitude, longitude, and elevation of each site associated with each type of place. Not all agents are members of all places, for example, students are members of a school but are not usually members of a workplace. Membership in user-defined places is controlled by the rules of the model.

9.2.2 User-Defined places

A model can define a new place by adding a place block of the form:

```
place  <place_name> { ... }
```

where <place_name> is the name of the class of places.

There are two ways to set the location of sites for user-defined places:

- Creating new sites by the Meta agent
- Reading sites from an external file

Creating sites

The Meta agent can generate new sites by executing the action `add_site`:

```
add_site(<place_name>, <id_expression>, <latitude_expression>, <longitude_
↪expression>, <elevation_expression>)
```

This action add a new site to the named place with `id`, `latitude`, `longitude`, and `elevation` set to be the values of the last four expressions in the argument list. If the `id` is 0, a unique site id of the new site is generated by FRED.

Reading sites from a file

The Meta agent can read sites from a file using the fnction `read_place_file`:

```
read_place_file(<file_name>,<place_name>)
```

For example, if the model had defined a place type called `Pharmacy` as above, then the Meta agent could call `read_place_file(pharmacy.txt, Pharmacy)` to generate pharmacy sites.

The place input file is a comma-separated text file with a header line. FRED parses the header line to identify fields of interest, which must include one fields labeled `ID`. Other fields can be used to define additional variables for the sites. The column labeled `ID` defines the place ID for each site. If the `ID` field in a given row is, say, `12345`, and if no existing group already has that ID, then a new site will be generated with that ID. If any existing place of the given type already has that ID, then any additional properties read from the file will apply to the existing place. If there is an existing group of another type that has the same ID, then FRED will abort with an error message. Finally, if the `ID` field has the value `0`, then a unique place ID will be generated and a new site created.

If there are other fields in the input file and if the header of a field corresponds to the name of an agent variable in the model, then the value of that agent variable will be set to the value of the corresponding field in the file. For example, suppose the model declares an agent variable as follows:

```
variables {
   agent numeric pharmacists
}
```

Suppose further that the model declares a Pharmacy place type:

```
place Pharmacy {
    has_group_agent = 1
}
```

And suppose the file **pharmacy.txt** contains the lines

```
ID, pharmacists, zipcode
123, 3, 15236
    456, 5, 15243
0, 10, 15243
```

Then the statement

```
read_place_file(pharmacy.txt, Pharmacy)
```

would generate three sites for `Pharmacy` with IDs `123`, `456`, and a unique ID something like `70000001`, assuming the that first two IDs do not clash with any existing group ID. The three pharmacies would be assign values for their `pharmacists` variable of `3`, `5`, and `10` respectively. The `zipcode` field would be ignored, unless `zipcode` were also declared as an `agent numeric` variable.

Note: FRED searches for the indicated file through the list of directories defined in the environmental variable `FRED_PATH`. Environmental variables can be set in your .bashrc file or other shell startup script (.zshrc for Mac users), for example:

```
export FRED_PATH=".:${HOME}/FRED_PROJECTS/project1:${HOME}/FRED_PROJECTS/
↪project2"
```

With the settings above, FRED will search for the filename.txt file first in the current working directory, followed by `${HOME}/FRED_PROJECTS/project1` and then `${HOME}/FRED_PROJECTS/project2`, where `$HOME` is the user's home directory.

If the `FRED_PATH` environmental variable is not set, FRED searches for the file in the current working directory.

9.3 Place Schedules

Agents can only be at one place at a time. In contrast, by default agents may participate in any number of networks at the same time. (Of course, specific models may restrict the participation in network to specific time by including rules that enforce this.)

For an agent to interact with others in a given place at a given time, three requirements must be satisfied:

- The agent must be a member of the place.
- The place must be open for business.
- The agent must be present in the place.

As an example of how to think about this, consider students in schools. In order to attend a school on a given day, the student must be:

1. enrolled in the school,
2. the school must be open that day, and
3. the student must not be skipping that day.

Agents can become a member of a place in a two ways:

- The agent can be assigned to a given place by the synthetic population files.
- **The agent can become a member of a place as the result of a** `join()` action.

Once an agent is a member of a place, the agent is considered to be attending the place during any time that the place is schedule to be open. The times during which a place is open are controoled in three ways:

- The FRED program can read the schedule from a file.
- The Meta agent can alter the schedule for any place.
- The group agent can alter the schedule for its specific site.
- The group agent can temporariliy close its place via an action.

9.3.1 changing a place's schedule

The following actions are used to define place schedules within FRED models:

Actions by the Meta agent:

- **clear_schedule(expression)**
  [If the expression is a place-type] name, this clears the schedule for the named place type. This means that the place type has no scheduled hours of operation. If the expresssion evaluates to an individual place ID, the schedule is cleared just for that place.
- add_to_schedule(expression, day-of-week, start-hour, start-minute, end-hour, end-min) : if the expression is a a place-type name, the place type schedule is changed to be open on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute. If the expression evaluates to an individual place ID, the schedule is changed just for the specific place. The hours must evaluate to 0 .. 23 and the minutes must evaluate to 0..59. Symbolic aliases are recognized, such as Mon for a day_of_week value of 1 and 2pm for an hour value of 14. In both add_to_schedule() and remove_from_schedule(), the day-of-week argument can be Weekdays meaning Monday through Friday, or Weekends meaning Sat and Sun, or Everyday meaning every day of the week.
- remove_from_schedule(expression, day-of-week, start-hour, start-minute, end-hour, end-min) : if the expression is a a place-type name, the place type schedule is changed by removing from the the open time the period on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute. If the expression evaluates to an individual place ID, the schedule is changed just for the specific place. The hours must evaluate to 0 .. 23 and the minutes must evaluate to 0..59. Symbolic aliases are recognized, such as Mon for a day_of_week value of 1 and 2pm for an hour value of 14.
- reset_schedule(place-ID) : removes the individualized schedule from the given place, reverting its schedule to that of its place type.

The group agent for a place may call the same actions as above, except that the schedule changes only apply to the group agent's specific place site:

- **clear_schedule()**
  [the group agent's place is changed to have no] scheduled hours of operation.
- add_to_schedule(day-of-week, start-hour, start-minute, end-hour, end-min) : the schedule for the group agent's place is changed to be open on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute.
- remove_from_schedule(day-of-week, start-hour, start-minute, end-hour, end-min) : the schedule for the group agent's place is changed by removing from the the open time the period on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute.
- reset_schedule() : removes the individualized schedule from the group agent's specific place, reverting its schedule to that of its place type.

The default_model in the synthetic population includes scheduling actions that define a default daily schedule for agents. For example, an agent that is a student has the following schedule on weekdays:

- 6am-8am: interact with members of the household.
- 9am-11am: interact with members of the same grade.
- 1pm-3pm: interact with members of the same school.
- 4pm-5pm: interact with members of the same census block group.
- 5pm-6pm: interact with members of the same census tract.

- 6pm-7pm: interact with members of the same county.

The default weekend schedule is the same except that school activities do not occur and the community activities are extended to a three hour period (50% more than on weekdays) by adding a one hour period of interaction with members of the same census block group.

9.3.2 reading schedule changes from a file

The Meta agent can call an action to read schedule changes from a file:

- `read_schedule_file(filename)` : read in a file which each line has the format: `ID,DAYS,OPEN,CLOSE` where `ID` is the ID of a specific place site, `DAYS` has a format like `Mon` for Monday, or `Mon-Wed` for Monday and Wednesday, or `Weekdays`, `Weekends` or ``Everyday` for thse groups of days, and `OPEN` and `CLOSE` have the format `HH:MM` of hour and minute using a 24-hour clock. The specific place schedule is changed to add the indicate times and days to the schedule.

9.3.3 skipping or attending a place

By default, an agent is present in any of its membership places whenever that place is open. However, agents may decide to temporaily skip attending a place by taking the action `skip(place)`. This action suspends the agent from participating in the place until the agent takes an action `attend(place)` within the same condition as the `skip()` action. More details are discussed in *chapter 10*.

9.3.4 temporary closures

To handle dynamic place closures (e.g. temporary school closures), you may assign a group agent to each group.

```
has_group_agent = 1  # default = 0
```

The group agent can close a place by executing a `close()` action which results in the place being temporarily closed, until the group agent performs a `reopen()` action in the same condition as the `close()` action. More details are discussed in *chapter 10*).

9.4 Containers

Places may be contained within other places. For example, a classroom may be contained within a school.

Containers are defined by the synthetic population `container_metadata.txt` file, which in turns lists all the place-container files included in the synthetic population.

For example, in the default synthetic population, schools are located within a specific zip code. The synthetic population includes a file called `school-zipcode.txt` that contains the following lines for one county:

```
PLACE,CONTAINER
450111632,15851
450110845,15825
450057407,15746
450132458,15767
450066817,15701
```

(continues on next page)

(continued from previous page)

```
450151186,15747
...
```

Each place ID in the first column represents a specific school, and each value in the second column represents a zip code. Each place site may have at most one container of any given place type. For example, a school may be contained in only one zip code.

Within a model, agents can access container information by the following function:

- `get_container(container, place)` : returns the place ID of the container of the type in the first argument that contains the agent's place of the type in the second argument.

That is, `get_container(County, Household)` will return the ID of the County that contains the agent's Household. The function follows the container relationships, so if Households are contained in a Census_Tracts and a Census_Tracts are contained in a County, then the function call above will return the specific County that contains the agent's Household.

The function returns `-1` if there is no such container or if the agent has no such place.

Places and their containers generally have distinct schedules, that is, times they are "open for business". However, if a group agent executes a temporary closure on its place, then any places contained within the temporarily closed place are also temporarily closed. For example, closing a school causes all classrooms within the school to be temporarily closed.

9.5 Networks

Networks are mixing groups that do not have a specific geo-location. Networks differ fundamentally from places in that all the people who share a given place at a given time are assumed to interact with each other uniformly (except for a possible bias to interact with people of a similar age), whereas people interact in a given network only with other people to whom they are explicitly linked. Networks in FRED can be used to represent network relationships such as friendships, sexual partners, buyers/sellers, health care providers/patients, and other relationships.

The user can declare any number of networks in FRED, and each network has a set of properties that determine how agents interact within that network. Networks are declared by a code block of the form:

```
network <network_name> {

   # STRUCTURAL PROPERTY
   is_directed = 1

   # ADMINISTRATIVE PROPERTY
   has_group_agent = 0

   # OUTPUT PROPERTY
   output_interval = 0
}
```

Network output is described in the SIMS Guide.

Note: Unlike a `<place_name>` that is generally associated with many sites, each `<network_name>` is associated with exactly one network. Networks in FRED may contain several connected components; that is, there may not be a path between every pair of individuals in a network. In general, individuals are linked to a subset of the other members of the network. Networks may be undirected or directed. There

are no built-in networks, but transmissible conditions can generate transmissible conditions by setting the `transmission_network` property.

If an edge is added, both agents identified in each edge become members of the network if they are not already members.

CHAPTER

TEN

CHAPTER 10: ACTION RULES

Actions are one of the three types of rules found within states (see *Chapter 4*). Actions may be categorized by what they act upon:

- *Actions on Variables*
- *Actions on Conditions*
- *Actions on Groups*
- *Actions on Other Agents*
- *Actions on Self*

by the type of agent(s) performing the action:

- *Actions by Ordinary Agents*
- *Actions by Group Agents*
- *Actions by the Meta Agent*

as well as a special set of *actions* which can generate or modify output from a simulation.

Note: In this section, the notation <value> denotes a single-valued expression while <values> denotes a list-valued expression.

10.1 Actions by Ordinary Agents

Most actions are available to oridinary agents with the exception of those as noted below.

10.2 Actions on Variables

Agents can modify variables that keep track of values of interest. Recall that scaler and list variables may be either either *shared or agent* in scope. The effect of an action on these types of variables depends on this scope. An action on a shared variable will update the value(s) stored in that variable for all agents. Conversely, an action on an agent variable will only effect the value(s) stored in the variable relative to a particular agent, usually the agent performing the action.

10.2.1 assignment actions

A common action is assiging a value (or values) to a variable. This action can be carried out using the assignment operator, =, or equivalently using the `set()` action.

- `<variable> = <value>` - set the agent or shared variable to a scaler value.
- `<list_variable> = <values>` - set the agent or shared list variable to a list of values.
- `set(<variable>, <value>)` - set the agent or shared variable to a scaler value.
- `set(<list_variable>, <values>)` - set the agent or shared list variable to a list of values.

An element in a list variable may be assigned:

- `<list_variable>[<index>] = <value>` - set the value of a particular element in a list variable.

Entries in a table may be assigned:

- `<table_variable>[<key>] = <value>` - set the value of a key in a table variable.
- `<table_variable>[<keys>] = <values>` - set the value for each key in a list of keys in a table variable.

Finally, a key in a list-table may be assigned to a list of values:

- `<list_table_variable>[<key>] = <values>` - set the value of a particular entry in a list-table variable.

10.2.2 on single-valued variables

- `tell(<other_agent_id>, <variable>, <expression>)` - set the variable of the agent whose ID is the value of the first argument to the value of the expression in the third argument.

10.2.3 on list-valued variables

- `push(<list_variable>, <expression>)` - appends the value of the expression to the list. If the expression is a list-valued expression, the elements in the expression are all appended to the end of the original list.
- `pop(<list_variable>)` - removes the last item from the list. Throws an exception if the list is empty.

10.2.4 on table variables

The following actions can be performed on both table and list-table variables:

- `clear(<table>)` - erase all key-value pairs in a table.
- `erase(<table>, <key_expression>)` - erase a particular key-value pair in a table.

10.2.5 on list-table variables

- **push(<list_table_variable>[<key>], <expression>) - appends the** value of the expression to the list associated with the key in the list_table. If the expression is a list-valued expression, the elements in the expression are all appended to the end of the original list.
- **pop(<list__table_variable>[<key>]) - removes the last item from** the list associated with the key in the named list-table. Throws an exception if the list is empty.
- An assignment list-table[key][pos] = value creates the list list-table[key] if it does not already exist, and assigns value to the 0-indexed position pos in the list. Any newly created items with index less than pos are assigned 0.

10.2.6 reading variables from a file

The following actions support reading in lists and tables from files:

- read_table(<table_name>, <filename>, <key_column>, <value_column>) - populates a table with key-value pairs from a specified CSV file. The <table_name>> and <filename> arguments are static. The third argument is evaluated to obtain a column index (starting with 0) from which to read the keys, and the fourth argument is evaluated to obtain a column index from which to read the values. The file must be a comma-separated-value file that contains the indicated columns. The assignments of key-value pairs is sequential, so that if a key occurs more than once in the file, the final value will be associated with that key in the table.

  Note: This function does not change any key-value pair in the table if the key does not occur in the file. That is, this function can only increase the number of key-value pairs in the table.

- read_list_table(<list_table_name>, <filename>, <key_column>, <value_column>) - populates a list_table with key-list_value pairs from a specified CSV file. The <list_table_name>> and <filename> arguments are static. The third argument is evaluated to obtain a column index (starting with 0) from which to read the keys, and the fourth argument is evaluated to obtain a column index from which to read the values. The file must be a comma-separated-value file that contains the indicated columns. For each key-value pairs read from the file, the value is appended to the list associated with the key in list_table.

  Note: This function overwrites any previous contents of the list_table.

Note: Lists can also be read from a file using the read() function, e.g.:

```
some_list = read(data_file.csv, column_index)


This function reads the indicated column from the file and stores all
the values in that column in the list variable.
```

10.3 Actions on Conditions

Entering a state may cause an agent to change its susceptibility or transmissibility for a condition through the following actions:

- <condition_name>.sus = <value> - set the agent's susceptibility to the condition to the value of the expression. Typically, a value of 1.0 is used to indicate the agent is "fully susceptible".

  Note: Setting <condition_name>.sus to a value less than 0.0 has the same effect as setting it to 0.0.

- <condition_name>.trans = <value> - set the agent's relative transmissibility for the condition to the value of the expression. The agent's transmissibility is multiplied by the condition's transmissibility to determine how infectious the agent is for the condition. Typically, a value of 1.0 is used to indicate "fully transmissible."

  Note: Setting <condition_name>.trans to a value less than 0.0 has the same effect as setting it to 0.0.

  For example, if an agent enters one of two states within a transmissible condition, COVID19, with the following actons:

```
state Asymptomatic {
    COVID19.trans = 0.5
}

state Symptomatic {
   COVID19.trans = 1.0
}
```

  The agent in the Asymptomatic state will be half as infectious as an agent that enters the Symptomatic state.

Warning: <condition_name>.sus and <condition_name>.trans may be set to values larger than 1.0. These values come into play when agent *A* attempts to transmit a condition to agent *B*. At that time a uniform random number in the range [0,1) is compared against the product the transmissibility of agent *A* and susceptibility of agent *B*. If the random number is less than the product then the attempted transmission succeeds. So the transmissibility of the the first agent can compensate for the susceptibility of the second agent, and vice versa. However, if the product is greater than or equal to 1.0, the transmission will always be successful–further increases in either quantity will have no effect.

Entering a state may cause the agent to change from one state to another state in another condition. As an example, an agent that enters a state representing receiving immunity from a vaccine may have the effect of changing from a susceptible state to a non-susceptible state for one or more disease conditions.

- set_state(<condition_name>, <state_name1>, <state_name2>) - if the agent is currently in <condition_name>.<state_name1>, then the agent's current state in <condition_name> becomes <state_name2>.
- set_state(<condition_name>, <state_name>) - the agent's current state in <condition_name> becomes <state_name>, regardless of the previous state of the agent in <condition_name>.

Examples,

```
state A {
    set_state(MOOD, Excited)
    set_state(VOTE, Waiting, Ready)
}
```

In this example, the agent entering state A changes its state in condition MOOD to the Excited state, regardless of the current state of the agent's MOOD. The second rule changes the agent to the Ready state in the VOTE condition only if the agent is already in the state Waiting.

10.4 Actions on Groups

Individual agents interact with others in mixing groups, including places and networks. The following actions affect how the agent interacts with mixing groups:

- join(<group_name>) - The agent will select and join a random group of the given type. If the agent already belongs to group of the given type, the action has no effect.
- join(<group_name>, <expression>) - The agent will select and join a specific group of the given type. The expression evaluates to the ID of a specific group. If the agent already belongs to group of the given type, the action has no effect.
- quit(<group_name>) - If the agent belongs to a group of the given type, the agent will leave that group; otherwise, the action has no effect. If the agent quits a Place, the agent also quits any partition of that Place.
- skip(<group_name>, ..., <group_name>) - The agent temporarily stops attending any of the listed groups. The agent continues to skip until the agent performs an attend() action in this same condition. If the list is empty, the agent stops attending any group.
- attend(<group_name>, ..., <group_name>) - This action cancels any previous skip() action for the named groups that was excuted in the current condition. If the list is empty, the agent resumes its normal attendance at group activities.

10.4.1 on places

Places can change locations in FRED. Movement occurs when an agent performs the move() action, with the first argument being a place name to which the agent belongs and the other arguments giving the change in x and y (in meters).

- move(<place_name>, <x>, <y>) - move the agent's place <x> meters East and <y> meters North.

For example:

```
move(Car, dx, dy)
```

This example assumes that model defines a place called Car and that the agent is a member of the specific car being moved.

Another action moves an agent's place to the location of another place:

- move_to(<place_name>, <place_name>) – move the agent's first argument place to the location of the agent's second argument place. For example,

```
move_to(Car, Household)
```

This moves the agent's car to the agent's household location.

A third kind of action moves an agent's place to a specific location specified as latitude and longitude:

- `move_to_location(<place_name>, <latitude>, <longitude>)` - move the agent's place to the indicate latitude and longitude. For example,

```
move_to_location(Car, lat, lon)
```

In all the above cases, a run-time error occurs if the agent does not belong to the indicated place.

10.4.2 on networks

- `add_edge_to(<network_name>, <other_agent_id>)` - An edge in the given network is added from the agent to the agent whose ID is the value of the second argument, an expression that is interpreted as the ID of another agent. Both agents join the network if they are not already members. If the expression does not evaluate to valid agent ID, the action has no effect. If the expression is a list-valued expression (that is, a list of agent IDs), then an edge is added to each agent in the list. For example, the followng action would create an edge in the network `HH` to all members of the agent household:

```
add_edge_to(HH, members(Household))
```

- `add_edge_from(<network_name>, <other_agent_id>)` - An edge in the given network is added to the agent from the agent whose ID is the value of the second argument. Both agents join the network if they are not already members. If the second argument does not evaluate to valid agent ID, the action has no effect. If the second argument is a list-valued expression (that is, a list of agent IDs), then an edge is added from each agent in the list.
- `delete_edge_to(<network_name>, <other_agent_id>)` - If an edge exists in the given network from the agent to the agent whose ID is the value of the second argument, then the edge is deleted. Otherwise, the action has no effect. If the second argument is a list-valued expression (that is, a list of agent IDs), then an edge to any agent in the list is deleted.
- `delete_edge_from(<network_name>, <other_agent_id>)` - If an edge exists in the given network to the agent from the agent whose ID is the value of the second argument, then the edge is deleted. Otherwise, the action has no effect. If the second argument is a list-valued expression (that is, a list of agent IDs), then an edge from any agent in the list is deleted.
- `set_weight_to(<network_name>, <other_agent_id>, <pair_expression>)` - If an edge exists in the given network from the agent to the agent whose ID is the value of the second argument, then the weight of the edge is set to the value of the third argument, which may include factors defined over either or both agents (see the discussion of pair context in *chapter 7*). Otherwise, the action has no effect. If the second argument is a list-valued expression (that is, a list of agent IDs), then any edge to any agent in the list is affected.
- `set_weight_from(<network_name>, <other_agent_id>, <pair_expression>)` - If an edge exists in the given network to the agent from the agent whose ID is the value of the second argument, then the weight of the edge is set to the value of the third argument, which may include factors defined over either or both agents (see the discussion of pair context in *chapter 7*). Otherwise, the action has no effect. If the second argument is a list-valued expression (that is, a list of agent IDs), then any edge to any agent in the list is affected.

10.5 Actions on Other Agents

Any agent may change the state of any other agent or list of agents with the action:

- `send(<expr>, <condition_name>, <state_name>)`

The expression may evaluate to either an agent ID or a list of agent IDs. Each agent in the list is immediately transitioned to the given state in the given condition.

If the condition is transmissible and the destination state is the exposed_state, then this is treated as an exposure from the executing agent. The sending agent need not be transmissible for the destination condition.

Warning: It is legal to send an agent from the `Excluded` state to another state.

10.6 Actions on Self

Some actions directly impact the agent preforming the action.

10.6.1 on location

Each agent has a specific location specified by its latitude and longitude. The agent's location is initialized to the agent's household and **does not change** unless explicitly modified by an action.

- `move(dx dy)` - move the agent itself dx meters to the east and dy meters north.
- `move_to(<place>)` - move the agent to a specific place to which the agent belongs.
- `move_to_location(<lat>, <lon>)` - move to specific location specieifed by the lattitude and longitude coordinates.

10.6.2 the birth and death actions

The following actions effect the life and death of agents:

- `give_birth()` - generates a new agent as offspring to the agent performing the action.
- `die()` - causes an agent to die.

10.7 Actions by Group Agents

Group agents are assigned to individual mixing groups, include places and networks. The group agent can control certain aspects about their groups using the following actions.

The following actions are only permitted by group agents. A run-time error if any other agent attempts to perform these actions.

- `close()` - The place associated with this group agent is temporarily closed. The schedule for the place is not changed. The temporary closure remains in effect until the group agent performs a `reopen()` action within the same condition as the `close()` action. A run-time error occurs if no such place exists for this agent.

- `reopen()` - Any temporary closure for the place associated with this group agent is lifted for the current condition. If the place still has a temporary closure due to another condition, the place remains closed. A run-time error occurs if no such place exists for this agent.
- `adjust_contacts(<multiplier>)` - The contact_rate for the group associated with this group agent is multiplied by the value of the given expression. The effect is to change the number of contacts among the agents attending the group. This can change the rate of transmissions that occur within the group. This change remains in effect until the group agent adjusts the contacts again. To return to the initial contact rates, the agent should do `adjust_contacts(1.0)`.

A group agent associated with a specific place may alter the times at which that place is open for business using the following actions. If any other agent attempts to perform these action, a run-time error occurs.

- `add_to_schedule(<day-of-week>, <start-hour>, <start-minute>, <end-hour>, <end-minute>)` - the group agent's place is scheduled to be open on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute. The `<day-of_week>` argument may be an expression that evaluates to a value in the range `0..6` where `0` represents Sunday. The `<day-of_week>` argument may also be one of the following symbolic terms: `Sun`, `Mon`, ..., `Sat` (meaning the corresponding of the days of the week), `Weekdays` (meaning Monday through Friday), `Weekends` (meaning Saturday and Sunday), or `Everyday`. The hours must evaluate to a value in the range `0..23`. Symbolic aliases for hours are recognized, such as `2pm` for `14`. The minutes must evaluate to a value in the range `0..59`.
- `remove_from_schedule(<day-of-week>, <start-hour>, <start-minute>, <end-hour>, <end-minute>)` - the indicated period is removed from the schedule for the group agent's place. See `add_to_schedule()` for a description of the arguments.
- `clear_schedule()` - The place associated with the group agent has no scheduled hours of operation.
- `reset_schedule()` - removes any site-specific schedule from the group agent's place, reverting its schedule to that of its place type.

10.8 Actions by the Meta Agent

Some actions can only be performed by the Meta agent.

10.8.1 reading agent file

The Meta agent can add new ordinary agents to the population, or add new features to existing agents with the following action:

- `read_agent_file(<filename>)` - reads a user data file using the same methods used to read an agent_file from the synthetic population. The fields are specified by a header line in the file. The field `ID` if it exists gives the agent ID. A missing ID field or a value of 0 in a specific row results in a new agent generated with a unique ID. The other fields in the file are interpreted as agent variable values if the column header corresponds to the name of a declared agent variable. `read_agent_file(filename)` can also be used as a list-expression in which case it returns the list of IDs of the agents read from the file.

10.8.2 reading place file

The Meta agent can add new place locations to the population, or add new features to existing places with the following action:

- `read_place_file(<filename>,<place_type>)` - reads a user data file using the same methods used to read a place_file from the synthetic population. Both arguments are static. The fields in the file are specified by a header line in the file. The field `ID` if it exists gives the place ID. A missing ID field or a value of 0 in a specific row results in a new place being generated with a unique ID. The other fields in the file are interpreted as agent variable values for the group agent associated with the place, if the column header corresponds to the name of a declared agent variable. `read_place_file(<filename>,<place_type>)` can also be used as a list-expression in which case it returns the list of IDs of the places read from the file.

10.8.3 on transmissibility

- `<condition_name>.trans = <value>` - Set the transmissibility of the condition to the value of the expression. This change remains in effect until further modification by the Meta agent.

10.8.4 on places

The Meta agent may create a new site for any place type. If any other agent attempts to perform this action, a run-time error occurs.

- `add_site(<place_name>, <sp_id>, <latitude>, <longitude>, <elevation>)` - Adds a new site to the named place with `sp_id`, `latitude`, `longitude`, and `elevation` set to be the values of the last four expressions in the argument list.

The Meta agent may alter the times at which places are open for business using the following actions. If any other agent attempts to perform these actions, a run-time error occurs.

- `add_to_schedule(<place>, <day-of-week>, <start-hour>, <start-minute>, <end-hour>, <end-minute>)` - the place is scheduled to be open on the given day of the week, starting at the start-hour and start-minute, and ending just prior to the end-hour and end-minute. If the `<place>>` argument is the name of a place type, the changes apply to all sites if that place type that do not have site-specific schedules. If `<place>` is an expression that evaluates to a place ID, then the change applies only to that specific site. The `<day-of_week>` argument may be an expression that evaluates to a value in the range `0..6` where `0` represents Sunday. The `<day-of_week>` argument may also be one of the following symbolic terms: `Sun`, `Mon`, ..., `Sat` (meaning the corresponding of the days of the week), `Weekdays` (meaning Monday through Friday), `Weekends` (meaning Saturday and Sunday), or `Everyday`. The hours must evaluate to a value in the range `0..23`. Symbolic aliases for hours are recognized, such as `2pm` for `14`. The minutes must evaluate to a value in the range `0..59`.
- `remove_from_schedule(<place-type-name>, <day-of-week>, <start-hour>, <start-minute>, <end-hour>, <end-minute>)` - the indicated period is removed from the schedule for the indicated place. See `add_to_schedule()` for a description of the arguments.
- `clear_schedule(<place>)` - If the expression is a place-type name, this action clears the schedule for the named place type. This means that the place type has no scheduled hours of operation. If the expression evaluates to an individual place ID, the schedule is cleared just for that site.
- `reset_schedule(<place-ID>)` - removes any site-specific schedule from the given site, reverting its schedule to that of its place type.

- `read_schedule_file(<filename>)` - reads in a file which each line has the format: `ID,DAYS,OPEN,CLOSE` where `ID` is the ID of a specific place site; `DAYS` has a format like `Mon` for Monday or `Mon-Wed` for Monday and Wednesday or `Weekdays` (meaning Monday through Friday), `Weekends` (meaning Saturday and Sunday), or `Everyday`; and `OPEN` and `CLOSE` have the format `HH:MM` for hour and minute using a 24-hour clock. The specific place schedule is altered to add the indicate times and days to the schedule.

10.8.5 on networks

The Meta agent can alter the edges in a network to create networks with specific degree distributions:

- `randomize_network(<network_name>, <mean_degree>, <max_degree>)` - The network is converted to a random network. The new mean degree of the network is determined by the value of the second argument, and the maximum degree allowed is determined by the value of the third argument.
- `preferential_attachment_network(<network_name>, <trials>, <samples_per_trial>)` - The network is converted to a preferential attachment network. The process is to repeatedly select a *source node* in the network and one outward edge from the source node (one source node per trial), and then to pick a number of *sample nodes* as possible destinations for the original edge. The original edge is replaced by an edge from the source node to the sample node with the largest in-degree, if the in-degree of the sample if larger than the in-degree of the original destination node.

Note: This "rewiring" procedure has been shown by Barabasi et al. (1999) to result in scale-free networks. The number of trials is determined by the value of the second argument, and the number of samples per trials is determined by the value of the third argument. The larger the number of trials, the more the degree distribution tends to a power-law distribution. this action leaves the mean degree of the network unchanged; that is, the total number of edges in the network does not change.

10.9 Actions Generating Output

Various actions can be used to alter the kinds of output generated by agents.

10.9.1 printing to file

Any agent may perform the following action to produce output:

- `open_csv(<filename>,...)` - opens a csv (comma-seperated vales) file with each additional argument being a quoted string serving as a column header. The first argument `<filename>` is static. All further output to this file must have the same number of fields as the column header.

It is a run-time error to open the same file more than once. There is a predicate `is_file_open(<filename>)` which returns `true` if the file has been opened, and `false` otherwise. It is common practice to have the Meta agent open all output files in the `startup` block:

```
startup {
   open_csv(file1.csv, "ID", "VALUE")
   open_csv(file2.csv, "DAY", "HOUR", "REPORT")
}
```

- `print_csv(<filename>,...)` - appends one line to the previously opened csv file `<filename>`. The first argument `<filename>` is static. There must be one additional argument for each column in the file. Each argument can be either a single value or a quoted string. The output is comma-separated.
- `print_file(<filename>,...)` - appends one line to the named file. The first argument `<filename>` is static. Each additional argument can be either a single value, a list expression, or a quoted string. In a list argument occurs, the values in the list are printed separated by spaces. There is no separation between fields in the output, so use " " as additional arguments where needed.
- `print(...)` - behaves like `print_file()` except the output is directed to the terminal at the end of the run. Each argument can be either a single value, a list expression, or a quoted string. In a list argument occurs, the values in the list are printed separated by spaces. There is no separation between fields in the output, so use " " as additional arguments where needed.

Note: All output files are written to the directory `<path_to_results>/JOB/<id>/OUT/RUN<run_number>/CSV/<filename>`.

Note: An output file can be retrieved by the command `fred_get_file -k <key> -f <csv_file> -r <run>`. If `-r` is omitted, then the files for all runs are retrieved. File are printed to standard output, with a header of the form:

```
RUN<run> ======================
```

Consider the following sample test program called test_print.fred:

```
simulation {
    locations = Jefferson_County_PA
    start_date = 2020-Jan-01
    end_date = 2020-Jan-04
}

startup {
    open_csv(out.csv, "ID", "SIM_DAY", "HOUR", "REAL_AGE", "SIM_RUN")
}

condition FOO {
    start_state = Start

    state Start {
        wait(0)
        if (age > 99) then next(A)
        default(Excluded)
    }

    state A {
        print_csv(out.csv, id, sim_day, hour, real_age, sim_run)
        wait(1)
        next(A)
    }
}
```

This population happens to have only two agents over the age of 99, so we should see output from each of these two agents during each time step they enter state `A`.

Suppose we run the program in a job with 2 runs:

```
$ fred_job -k test_print -p test_print.fred -n 2 -m 2
fred_job: starting job test_print at Tue Feb  2 15:36:50 EST 2021

fred_compile -p test_print.fred
No errors found.
No warnings.

fred_job: running job test_print id 154 run 1 ...
fred_job: running job test_print id 154 run 2 ...
run_set 0 completed at Tue Feb  2 15:36:51 EST 2021

fred_job: finished job test_print 154 at Tue Feb  2 15:36:51 EST 2021
```

Now let's retrieve the output file from run 2:

```
$ fred_get_file -k test_print -f out.csv --run 2

ID,SIM_DAY,HOUR,REAL_AGE,SIM_RUN
185882382,0,0,102.932,2
354736951,0,0,106.464,2
185882382,0,1,102.932,2
354736951,0,1,106.464,2
185882382,0,2,102.932,2
354736951,0,2,106.464,2
185882382,0,3,102.932,2
354736951,0,3,106.464,2
185882382,0,4,102.932,2
354736951,0,4,106.464,2
185882382,0,5,102.932,2
354736951,0,5,106.464,2
185882382,0,6,102.932,2
354736951,0,6,106.464,2
185882382,0,7,102.932,2
354736951,0,7,106.464,2
...
```

10.9.2 printing an event message

It is often useful to print messages associated with an agent's current state. This is mainly useful when developing or debugging a model, but it might also be used whenever it is necessary to trace the activity of agents. The following actions address this need:

- `print_event_file(<filename>, <arg_list>)` - print one line of text to the named file. The line begins with the simulation date, the simulation time, the simulation day, the agent id and the current condition and state. This is then followed by the information in the arg list in the same way as *print_file()*.
- `print_event(<arg_list>)` - behaves the same as `print_event_file()` except that the output is directed to stdout.

For example:

```
simulation {
   locations = Jefferson_County_PA
       start_date = 2020-Jan-01
       end_date = 2020-Jan-01
}

condition TEST {
   start_state = Start
       state Start {
           if (age > 96) then print_event(age)
               wait()
               default()
}
```

produces the output:

```
====================================
output from RUN1:
2020-01-01 00:00:00 day 0 agent 221220472 in TEST.Start 98
2020-01-01 00:00:00 day 0 agent 335367245 in TEST.Start 99
2020-01-01 00:00:00 day 0 agent 365122652 in TEST.Start 98
2020-01-01 00:00:00 day 0 agent 376295212 in TEST.Start 98
2020-01-01 00:00:00 day 0 agent 402760673 in TEST.Start 98
2020-01-01 00:00:00 day 0 agent 133928247 in TEST.Start 99
2020-01-01 00:00:00 day 0 agent 190108957 in TEST.Start 97
2020-01-01 00:00:00 day 0 agent 236048890 in TEST.Start 98
2020-01-01 00:00:00 day 0 agent 343232988 in TEST.Start 97
2020-01-01 00:00:00 day 0 agent 939614941 in TEST.Start 104
2020-01-01 00:00:00 day 0 agent 939614910 in TEST.Start 102
====================================
```

10.9.3 printing an abort message

The following action will terminate a FRED run under program control:

- `abort()` - causes FRED to terminate with an error message:

```
FRED ERROR: Agent <id> aborts in condition <cond_name> state <state_name> on␣
↪sim day <N> sim date <YYYY-MM-DD>:
<rule_containing_abort>
```

Example:

```
condition FOO {
      ...
    state B {
        if (age < 18) then abort()
        ...
    }
}
```

If an agent under 18 enters state `B` above, FRED exits with an error message like:

```
FRED terminated with errors:
====================================
RUN1: FRED ERROR: Agent 164581176 aborts in condition FOO state B on sim day␣
↪10 sim date 2020-01-11:
if(age<18) then abort()
```

Note:

The `abort()` function provides functionality normally associated with an `assert()` operator since FRED prints the rule that causes an abort.

10.10 Control Structures for Actions

FRED includes several control structures that determine which actions are performed by an agent.

10.10.1 if-then-else

Any action may be preceded by an `if-then` test. The action is performed only if the test is `true` for the agent executing the statement.

For example,

```
if (age < 10) then print("agent ", id, " is younger than 10")
```

In this example, the agent executing the statement prints a message if the agent's age is less than 10.

The test may contain any number of `clauses` separated by &, each of which must be true for the action to be performed.

For example,

```
if (age < 10 & size(Household) < 5) then <some_action>
```

Here, the action is performed if the agent is less than 10 and belongs to a Household with fewer than 5 members.

To perform more than one action if the test is true, put the actions into a code block:

```
if (<test_1> & <test_2> & ... <test_I>) then {
   <action_1>
   <action_2>
   ...
   <action_N>
}
```

To perform some actions if the test is `true` and some other actions if the test is `false`, use an *if-then-else* statement:

```
if (<test_1> & <test_2> & ... <test_I>) then {
   <action_if_true_1>
   <action_if_true_2>
```

(continues on next page)

(continued from previous page)

```
   ...
   <action_if_true_N>
}
else {
   <action_if_false_1>
   <action_if_false_2>
   ...
   <action_if_false_M>
}
```

Example:

```
if (age < 10) then {
   count_under_10 = count_under_10 + 1
   print("agent ", id, " is younger than 10")
}
else {
   count_10_or_older = count_10_or_older + 1
   print("agent ", id, " is at least 10")
}
```

Note: If a statement includes an `else` part, then brackets are required for both the `then` block and the `else` block, even if these block contain only one action.

The spacing in an `if-then-else` is ignored, so it is acceptable to have an entire statement on a single line, for example:

```
if (age < 10) then {x = 0} else {x = 1}
```

Note that brackets are required here because the statement includes an `else` part.

10.10.2 for-loops

A `for-loop` is used for iterating over a list in a manner similar to the `for-loop` in `Python`. A set of actions is performed by the agent, once for each item in the list.

```
for (<shared_var>, <list-expression>) do {
   <action_1>
   <action_2>
   ...
   <action_N>
}
```

The `for` statement has two arguments. The first argument must be the name of a `shared numeric` variable that serves as a `loop variable`. The second argument is an expression that evaluates to a list of values.

The loop variable is assigned to each value in the list and the block of actions is performed for each value of the loop variable.

Examples:

```
my_list = list(2,6,5,8)
for (x, my_list) do {
   if (x > 5) then print(x)
}
```

The above example would print out 6, and 8, one value per line.

To loop through a set of actions a specified number of times, use the `range()` function. The `range()` function returns a list of numbers, starting with `0` by default, and increments by 1 (by default), and ends at a specified number.

With a single argument, `range(N)` returns the list `0..(N-1)`. For example,

```
for (x, range(6)) do {
    print(x)
}
```

The above code would print of the values `0` through 5, one value per line.

10.10.3 while-loops

A `while-loop` is used for iterating over a set of actions as long as a given test is true.

```
while (<test>) do {
   <action_1>
   <action_2>
   ...
   <action_N>
}
```

For example:

```
i = 0
while (i < 6) do {
   print(i)
   i = i + 1
}
```

The above code would print of the values `0` through 5, one value per line.

Note: Remember to add 1 to `i` in the above example, or the statement would run forever.

10.10.4 nested statements

It is acceptable to nest control statements in FRED. For example, you can use nested `for-loops` or use `if-then-else` statements inside of `while-loops`.

CHAPTER

ELEVEN

CHAPTER 11: WAIT RULES

For each state in a given condition, the modeler should answer three questions:

1. What happens when an agent enters this state? That is, what does this state mean in terms of the agent's own status or how the agent interacts with others?
2. How long does the agent stay in this state?
3. What state does the agent go to next?

The answers to these questions are expressed in rules that control how agents change during a FRED simulation. There are three major categories of rules in FRED that corresponds to the questions above:

1. Action rules
2. Wait rules, and
3. Transition rules.

This Chapter focuses on wait rules and how event timing occurs.

Whenever an agent enters a state, FRED determines how long the agent should wait in that state before the transition rules are applied to select the next state. The wait time for each state is defined through wait rules that have the form:

```
[ if (<test1> & <test2> & ... <testN>) then ] wait(<expression>)
```

The portion in brackets is optional. The argument to `wait()` specifies a duration (in hours) for how long to wait before the transition decisions are made. Some examples:

- `wait(1)` – Wait 1 hour
- `wait(0)` – Apply transition rules immediately. Transient states with zero wait times are often useful, since they can cause effects that change the agent's other conditions, or they can serve as decision points that separate the state trajectories of agents based on their properties.
- `wait()` – Wait indefinitely.
- `wait(-1)` – Wait indefinitely with any negative value.
- `wait(1.6)` – Wait 2 hours. The value of the expression is rounded to the nearest integer.
- `wait(0.4)` – Same as `wait(0)`. The value of the expression is rounded to the nearest integer.

Any expression is permitted, so if the wait time should be drawn from a normal distribution with median 5.5 days and standard deviation of 2.0 days, you can say:

```
wait(24 * normal(5.5, 2.0) )
```

Note: State durations are often defined using statistical distributions. See *chapter 7* for a list of distributions.

11.1 The until() Function

The `until()` function is especially useful in wait rules. This function returns the number of hours until the specified date and time:

```
until(year, month, day_of_month, hour)
```

For example: the following wait rule would wait until 2020-Dec-31 at 11pm.

```
wait(until(2020,12,31,23))
```

Warning: If an agent encounters a wait-until statement at a moment greater than or equal to the date and time indicated within the `until()` function, the agent will wait forever in the state. This situation is treated as equivalent to encountering an empty wait statement, i.e. `wait()`. A common way to account for this situation is to add a conditonal wait prior to the wait-until statement. For example,

```
if (now >= sim_step(2020,12,31,23)) then wait(0)
wait(until(2020,12,31,23))
```

In the above example, if an agent enters the state on or after 2020-Dec-31 at 11pm, they will transition to the next state immediately.

The until function understands symbolic terms for months and hours, so the above could also be written as:

```
wait(until(2020, Dec, 31, 11pm))
```

Any argument can be an expression, for example:

```
wait(until( year+1, Jan, 1, 12am))  # wait until next New Year's day
```

Note: An illegal argument to `until()` will cause the simulation to abort with an error message, for example `wait(yr, mon, day, hr)` where `hr < 0` or `hr > 23`.

If only three arguments are given, the first argument must be the abbreviation for a month, such as:

```
until(Dec,12,3pm)
```

If only two arguments are given, the first argument must be the symbolic name of a month or a day of the week:

```
wait(until(Sat,11pm))
wait(until(Dec,31))
```

The latter case assumes a time of `12am`.

If a single argument is supplied of the form <number>am or <number>pm, it is interpreted as a time of day on a 12-hour clock:

```
wait(until(2pm))  # wait until 2pm in the afternoon
```

If until is given a single argument not in the format <number>am or <number>pm, the argument is interpreted as a timestamp or a datestamp:

- A **timestamp** is an expression that evaluates to a number of the form YYYYMMDDHH, where YYYY represents the year, MM represents the month (with values 01,...,12), and HH represents the hour on a 24-hour clock (with values 00,...,23.

You can construct a timestamp using the the formula:

```
my_timestamp = 1000000*year + 10000*month + 100*day_of_month + hour
```

- A **datestamp** is an expression that evaluates to a number of the form YYYYMMDD.

The formula for a datestamp is:

```
my_datestamp = 10000*year + 100*month + day_of_month
```

For example, we might have a variable next_change whose current value is 20201104 (a valid datestamp). Then the following makes an agent wait until 2020-Nov-04 at 12am:

```
wait(until(next_change))
```

If the current value of next_change is the timestamp 2020110413, then the following makes an agent wait until 2020-Nov-04 at 1pm:

```
wait(until(next_change))
```

If the expression does not evaluate to a valid timestamp or datestamp, a run-time error will occur and the simulation will be terminated.

The until() function always returns a positive number. That is, it always refers to a future time step.

Example:

```
state A {
    wait(until(12am))
    next(B)
}
```

If an agent enters state A at 12am, the agent will wait 24 hours.

To have the agent go to state B at the earliest possible midnight, use this pattern:

```
state A {
    if (hour == 12am) then wait(0)
    wait(until(12am))
    next(B)
}
```

Warning: the wait_until_ factors in FRED 6 are no longer supported.

11.2 Multiple Wait Rules

If a state contains multiple wait rules, the first rule that applies to the agent is used

```
state Symptoms {
    if (age <= 10) then wait(3*24)
    if (age <= 20) then wait(4*24)
    wait(5*24)
    next(Recovered)
}
```

In the example above, we assign a duration of `Symptoms` that depends on the age of the agent: 3 days for agents with age 10 or less, 4 days for agents between 11 and 20 years old, and 5 days for agents over 20.

Note: Each state must have at least one unconditional wait rule, or a compiler error will result and the program will not execute. Given how wait rules are evaluated, any wait rule after the first unconditional wait rule will be ignored.

11.3 Zero-duration Wait Rules

If an agent executes a wait rule with zero duration, e.g. `wait(0)`, the agent immediately evaluates the transition rules for the current state (as described in *chapter 13*) and proceeds to the selected next state. If the agent also executes a zero-duration in the next state, the agent again evaluates that state's transitions rules and proceeds to the next state. This process continues until the agent reaches a state in which it executes a non-zero duration wait rule.

This pattern is often useful when the the agent needs to make a series of decisions during a given time steps, and each zero-duration state represents a separate decision branch opportunity. A series of zero-duration states is also useful when an agent needs to execute actions in a loop, for example, to process items in a list.

However, it is also possible to create infinite loops through a series of zero-duration state transitions, for example:

```
state A {
      ...
      wait(0)
      next(B)
}

state B {
      ...
      wait(0)
      next(A)
}
```

It is not possible for the FRED compiler to detect all such situations. To protect against infinite loops, the FRED platform counts the number of zero-duration transitions performed by an agent during each step, and terminates with an error message if an agent performs more than a set maximum number of zero-duration transitions:

```
FRED ERROR: LOOP DETECTED AFTER 1000000 LOOPS condition <condition_name> day
↪<sim_day> hour <hour>  person <id> old_state <state_name> new_state <state_
↪name>
```

The maximum number is controlled by the `max_loops` configuration parameter, with default value `max_loops = 1000000`. If you need to have an agent perform more than this number of zero-duration transition, for example, to loop over very large lists, you can set the property in the simulation block, for example:

```
simulation {
       max_loops = 999999999
}
```

CHAPTER

TWELVE

CHAPTER 12: STATE TRANSITIONS

This chapter describes how **Transition Rules** are used to decide what state an agent goes to next.

12.1 Transition Rules

Transition rules control the next state that the agent will assume. There are two kinds of transition rules:

- Probabilistic rules
- Default rules

12.1.1 Probabilistic Transition Rules

```
[ if (<test1> & <test2> & ... <testN>) then ] next(<state_name>) [ with prob(
↪<expression>) ]
```

This rule means that if the agent satisfies all the tests specified, then the probability of the agent entering the named state is the value of `<expression>`. If `with prob(<expression>)` is omitted, it is equivalent to `with prob(1)`.

An example of a transition rule in a FRED program might be:

```
if (age > 16) then next(Dropout) with prob(0.2)
```

The rule says, "if an agent is more than 16 years old, then the probability of the agent's becoming a Dropout is 20%."

12.1.2 Default Next State rules

Within each state block, there may be transitions rule of the form:

```
default(<state_name>)
```

Note that this rule does not include any tests. This rule says that the default next state is `<state_name>` if none of the other next states are selected by the other transition rules. Specifically, if the next state probabilities for all other next states sum to a value $p < 1.0$, then the transition probability from the current state to the default State is set to $(1 - p)$.

If more than one default transition rule is present for a given state block, the last one in the program file will apply.

For each state, the built-in default transition rule is:

```
state S {
    default(S)
}
```

That is, an agent will stay in the same state if there is no other transition rule. If no explicit transition rules or default rules are included in the FRED model file, the compiler will issue a warning to remind the user that the state will transition back to itself by default.

12.2 State Transitions

Many systems are described by a state-transition matrix that defines the probability for moving from the state that labels a row in the matrix to any of the other states, with the individual probabilities represented by the columns in the matrix. For example, in the matrix below, there is a probability of 0.5 of moving from state `A` to either state `B` or state `C`.

	A	B	C
A	0.0	0.5	0.5
B	0.1	0.8	0.1
C	0.0	0.0	1.0

In FRED, only the portion of the state-transition matrix required for each agent is computed dynamically, so that the transition probabilities can vary from agent to agent and can also vary over time for an individual agent. When an agent reaches the state transition duration for its current state, the agent's next state is computed as follows:

1. Compute the transition probabilities for this agent in the row of current state.
2. Select next state using the probability distribution for the row of the current state.

For example, given the transition matrix shown above for a given agent in state `B`, the agent would transition to state `A` with a probability of 0.1, transition to state `C` with a probability of 0.1, and remain in state `B` with a probability of 0.8.

12.3 Multiple Probabilistic Rules

Suppose we have an agent in state `C.i` for some condition `C` that has states `n` states. For each possible next state `j`, FRED computes the probability `p(i,j)` of the transition from `i` to `j` as follows:

1. Find all qualifying rules for the given agent from state `i` to state `j`. These are the rules that specify the current state `C.i` and the next state `j` and for which the agent meets all the predicates in the rule.
2. For each qualifying rule, compute the probability term for the given agent. If the probability term is omitted, the probability for this rule is 1.0.
3. Use the maximum probability computed for any qualifying rule as the tentative probability `p(i,j)` for the agent to transition from state `i` to state `j`.

12.3.1 Default Next State Rule and Normalization

After repeating this process for each possible next state, we have a probability vector of the form:

```
p(i,1), p(i,2), ..., p(i,n)
```

Let the total probability be the sum of the values in the vector. The total may not equal 1.0, so the values need to be normalized to create a valid probability distribution. But first, FRED applies the default next state rule for state `i`. Let state `k` be the default next state for state `i`. The effect of the default next state is:

Note: If the total probability is < `1.0`, then add `(1.0 - total)` to `p(i,k)`. Otherwise, divide all the values in the vector by the total probability.

After applying the rule above, the values in the modified vector <`p(i,1), p(i,2), ... p(i,n)`> sum to 1.0. The next state is selected by choosing a state randomly using this vector as the probability weight for the states.

12.4 Cautions

It is important to understand the use of transition rules. Incorrect use of transition rules can lead to unexpected results. This section highlights a few areas that require attention.

12.4.1 The difference between using `next()` and `default()`

Suppose we have three states `A`,`B`, and `C`, and we are writing transition rules for state `A`. Suppose we want to have agents less than 10 years old transition from `A` to `B` with probability 0.25 and to state `C` with probability 0.75; all other agents should always transition to state `C`.

The preferred code for this situation would be:

```
state A {
    ...
    if (age < 10) then next(B) with prob(0.25)
    default(C)
}
```

If an agent less than 10 years old is in state `A` then the `if ()` statement is the only qualifying transition rule, so the tentative probability of going to state `B` is set to 0.25. Because the total probability is less than 1.0, the `default()` next state rule applies, and the probability of going to `C` is set to 0.75 = 1.0 - 0.25.

If an agent 10 or older is in state `A` then there are no qualifying transition rules, so the total probability equals 0.0; In this case the `default()` transition rule applies and sets the probability of going to state `C` to 1.0 = 1.0 - 0.0.

The following code would lead to *incorrect results*:

```
state A {
    ...
    if (age < 10) then next(B) with prob(0.25)
    next(C)  # WRONG!
}
```

Things work fine for agents 10 or older, but if an agent less than 10 years old is in state `A` then *both* transition rules apply. The `if ()` statement sets the tentative probability of going to state `B` to 0.25. The `next()` statement sets the tentative probability of going to `C` to 1.0. The Total is 1.25 = 0.25 + 1.0, so the probabilities are normalized to

```
p(A,B) = 0.25/1.25 = 0.20
```

and

```
p(A,C) = 1.0/1.25 = 0.80
```

which are not the desired results.

12.4.2 Missing default Rule

If no `default()` rule is included in a state, the default next state is the state itself, so these two snippets are equivalent:

```
state A {
    ...
    if (age < 10) then next(B)
}
```

```
state A {
    ...
    if (age < 10) then next(B)
    default(A)
}
```

This might lead to unexpected results if a missing default rules is applied. For example, consider this snippet:

```
state A {
    wait(0)
    if (age <= 10) then next(B)  # WRONG!
}
```

The above code would lead to an infinite loop back to state `A` for all agents over age 10.

12.5 Example: Using Rules to Represent Health Risks

This section shows how to assign to each individual in the population a risk of having various conditions. For many health conditions, we can find tables that show the prevalence of the health condition based on demographic factors such as age, race, and sex. For example, the following table shows the prevalence of asthma in the United States based on data from the CDC:

Asthma prevalence

Age	White		Nonwhite	
	Male	Female	Male	Female
0-4	3.7	3.7	11.5	9.0
5-14	9.5	8.4	20.1	17.7
15-19	9.4	11.2	12.8	17.9
20-24	6.5	11.5	14.3	13.2
25-34	6.5	10.5	8.8	12.5
35-64	5.9	10.7	6.3	12.7
65+	5.5	7.5	7.8	7.9

Source: National Health Interview Survey, National Center for Health Statistics, CDC

Suppose we are creating a FRED model of asthma in which all individuals begin the the `Start` state. At the start of the simulation, we want each individual to move to either the `AtRisk` state (meaning that they are subject to asthma attacks) or to the `Negative` state (meaning they do not suffer from asthma attacks), according to data in the Table above.

This can be accomplished by having one transition rule per cell in the Table, as shown in the following condition:

```
condition ASTHMA {
   transmission_mode = environmental
   start_state = Start
   ...


   state Start {
      wait(0)
      if (sex==male & race==white & range(age, 0, 4)) then next(AtRisk) with␣
↪prob(0.037)
      if (sex==male & race==white & range(age, 5, 14)) then next(AtRisk) with␣
↪prob(0.095)
      if (sex==male & race==white & range(age, 15, 19)) then next(AtRisk) with␣
↪prob(0.094)
      if (sex==male & race==white & range(age, 20, 24)) then next(AtRisk) with␣
↪prob(0.065)
      if (sex==male & race==white & range(age, 25, 34)) then next(AtRisk) with␣
↪prob(0.065)
      if (sex==male & race==white & range(age, 35, 64)) then next(AtRisk) with␣
↪prob(0.059)
      if (sex==male & race==white & 65<=age) then next(AtRisk) with prob(0.055)
      if (sex==female & race==white & range(age, 0, 4)) then next(AtRisk) with␣
↪prob(0.037)
      if (sex==female & race==white & range(age, 5, 14)) then next(AtRisk)␣
↪with prob(0.084)
      if (sex==female & race==white & range(age, 15, 19)) then next(AtRisk)␣
↪with prob(0.112)
      if (sex==female & race==white & range(age, 20, 24)) then next(AtRisk)␣
↪with prob(0.115)
      if (sex==female & race==white & range(age, 25, 34)) then next(AtRisk)␣
↪with prob(0.105)
```

(continues on next page)

(continued from previous page)

```
      if (sex==female & race==white & range(age, 35, 64)) then next(AtRisk)␣
↪with prob(0.107)
      if (sex==female & race==white & 65<=age) then next(AtRisk) with prob(0.
↪075)
      if (sex==male & race!=white & range(age, 0, 4)) then next(AtRisk) with␣
↪prob(0.115)
      if (sex==male & race!=white & range(age, 5, 14)) then next(AtRisk) with␣
↪prob(0.201)
      if (sex==male & race!=white & range(age, 15, 19)) then next(AtRisk) with␣
↪prob(0.128)
      if (sex==male & race!=white & range(age, 20, 24)) then next(AtRisk) with␣
↪prob(0.143)
      if (sex==male & race!=white & range(age, 25, 34)) then next(AtRisk) with␣
↪prob(0.088)
      if (sex==male & race!=white & range(age, 35, 64)) then next(AtRisk) with␣
↪prob(0.063)
      if (sex==male & race!=white & 65<=age) then next(AtRisk) with prob(0.078)
      if (sex==female & race!=white & range(age, 0, 4)) then next(AtRisk) with␣
↪prob(0.09)
      if (sex==female & race!=white & range(age, 5, 14)) then next(AtRisk)␣
↪with prob(0.177)
      if (sex==female & race!=white & range(age, 15, 19)) then next(AtRisk)␣
↪with prob(0.179)
      if (sex==female & race!=white & range(age, 20, 24)) then next(AtRisk)␣
↪with prob(0.132)
      if (sex==female & race!=white & range(age, 25, 34)) then next(AtRisk)␣
↪with prob(0.125)
      if (sex==female & race!=white & range(age, 35, 64)) then next(AtRisk)␣
↪with prob(0.127)
      if (sex==female & race!=white & 65<=age) then next(AtRisk) with prob(0.
↪079)
      default(Negative)
   }

   state AtRisk {
      ASTHMA.sus = 1.0
      wait()
      next()
   }

   state Negative {
      ASTHMA.sus = 0.0
      wait()
      next()
   }

}
```

Each rule contains a set of tests that matches the demographic factors of each cell in the table and then assigns the same probability of being at risk as the corresponding cell in the table. It is worth noting that any individual in the population matches only one of the probability rules, because the predicates are mutually exclusive. For example, a 5-year old white male only matches the second rule, so his transition probability from `Start` to `AtRisk` is 0.095. The effect of the default rule is to set the transition probability

from `Start` to `Negative` of this same individual to `1.0 - 0.095 = 0.905`. So, we expect that about 9.5% of 5-year old white males in the population will be in the AtRisk state after the transition. Of course, the results will vary from run to run, since each individual makes a probabilistic transition.

12.6 Concurrent Events

The FRED language's approach to agent-based modeling is to provide a set of rules that apply to individual agents. Usually, the interesting results of the model arise from the combined effects of many agents interacting with each other and with their environment.

Although it is natural to think about all the agents in a simulation performing their rules in parallel, in practice, the FRED platform interprets the rules one agent at a time. As a result, there may be cases where the modeler has to understand how **concurrent events**, that is, events that are scheduled to occur on the same simulation step, are handled in practice.

12.6.1 The Order of Conditions

In FRED, conditions are updated in the order in which they are defined in the FRED program. That is, for a given time step, agents that are scheduled for a state transition in condition `A` experience state transition before agents that are scheduled for a state transition in condition `B` if condition `A` is defined before condition `B` in the FRED program.

For example, suppose state `A1` is the first state in condition `A` and state `B1` is the first state in condition `B`, and we have the following state definitions:

```
condition A {
   start_state = A1

   state A1 {
      wait(24)
      next(A2)
   }

   state A2 {
      <action A2>
      ...
   }
}

condition B {
   start_state = B1

   state B1 {
      wait(24)
      next(B2)
   }

   state B2 {
      <action B2>
      ...
   }
}
```

Then every agent is scheduled to enter both state `A2` and `B2` at time step 24. Because condition `A` is defined before condition `B`, all the agents will perform `<action A2>` before any agent performs `<action B2>`. If the model assumes any other order of events, it will likely produce unexpected results.

Note: It is considered a good modeling practice to avoid assumptions about the order in which state transition are processed across condition, if possible.

12.6.2 Concurrent State Transitions

FRED is a discrete-time model and the time, with a step of one hour. For each hour of the simulation, FRED performs the following for each user-defined Condition:

- Identify all agents that need to be updated according to the user-defined rules.
- For each identified agent:
  - Select the agent's next state according to the user-defined rules.
  - Perform any actions that are associated with the agent's next state.
  - If the condition involves interactions among agents, simulate the agent interactions within the defined interaction groups. Interactions may result in some agents changing their states.

Let's examine these steps in in more detail.

FRED maintains an **event queue** for each condition consisting of the list of agents that are scheduled to perform state transitions during the next simulation step. For each agent on the event queue, FRED performs the following sub-steps:

1. Evaluate the transition rules in the agent's current state and select the agent's new state (which may be the same as the agent's old state).
2. Execute the action rules in the agent's new state.
3. Evaluate the wait rules in the agent's new state, and schedule the next event for the agent at the end of the duration of the wait step.

If multiple agents are scheduled for state transitions at the same simulation time step, the FRED language does not define the order in which agents are processed from the event queue. Different implementations may use different orderings, for example, first-in-first-out (FIFO) or perhaps random order.

For example, consider the program fragment below:

```
condition FOO {
   start_state = A

   state A {
      wait(24)
      next(B)
   }

   state B {
      agent_id = id
      wait(24)
      next(C)
   }
}
```

In the example above, all individual agents start in state `A` where they wait for 24 hours. At timestep 24, the agents are processed in some order and proceed to state `B`, where the value of `agent_id` will be the id of the last agent to enter state `B`. In some implementations this will be the last agent in the population, but it may be different in other implementations, or even in the next run of the model.

> **Warning:** It is considered a modeling error to assume a given order of agents who perform concurrent state transitions within a given state.

In some cases, it may be desired to impose a random order on concurrent events to avoid systematic biases toward certain agents. This can be achieved by setting the condition property `shuffle`, as follows:

```
condition <condition_name> {
    shuffle = 1
    ...
}
```

With this setting, agents who are scheduled for a state transition at the same simulation step will be selected from the event queue in random order. For example:

```
condition FOO {
   start_state = A
   shuffle = 1

   state A {
      wait(24)
      next(B)
   }

   state B {
      agent_id = id
      wait(24)
      next(C)
   }

   state C {
      agent_id = id
      wait()
      next()
   }
}
```

In the example above, all individual agents start in state `A` where they wait for 24 hours. At timestep 24, the agents are processed in random order and proceed to state `B`, where the value of `agent_id` will be the id of the last agent to enter state `B`, that is, the id of a random agent. After waiting 24 hours in state `B`, the agents are again processed in random order, so the value of `agent_id` is set to another random agent id in state `C` at timestep 48.

12.6.3 Effects of `set_state()`

Concurrent event can also arise as the result of the `set_state()` action, which immediately changes the state of an agent in a different condition. The affected agent also immediately performs any actions associated with its new state. Consider the following example:

```
condition COND1 {

   ...

   state A {
      set_state(COND2,B,C)
      x = 1
      wait()
      next()
   }

   ...

}

condition COND2 {

   ...

   state B {
      wait()
      next()
   }

   state C {
      x = 2
      wait()
      next()
   }

   ...

}
```

If an agent enters state `COND1.A`, the final value of `x` is 1. The order of event is:

- Agent enters `COND1.A`
- Agent runs the `set_state()` action, which:
- Causes the agent to enter `COND2.C`
- Agent sets `x = 2`
- Agent waits in state `COND2.C`
- Agent continues to execute actions in state `COND1.A` and sets `x = 1`

That is, the `set_state()` action is not queued. It runs immediately, and the actions in the new state are performed before returning to the calling state.

Note: It is considered a good modeling practice to avoid assumptions about the order of agents who perform concurrent state transitions.

CHAPTER

THIRTEEN

CHAPTER 13: TRANSMISSION

FRED was originally designed to model the transmission of infectious diseases from one person to another within a population. The concept of transmission can also be applied to many aspects of social behavior.

One way that agents interact in FRED is through a process called **transmission**. Transmission means that one agent causes another agent to enter a certain state within a condition. The first agent is said to be **transmissible** for the condition, and the second agent is said to be **susceptible** to the condition. We also say that the first agent **exposes** the second agent to the condition. Transmission can also be caused by the action of the Meta agent.

13.1 Transmissibility

There are three forms of **transmissibility** that each serve a different function in FRED models:

- the transmissibility of a condition
- the transmissibility of each agent
- the transmissible actions of the Meta agent.

Each of these is described below.

13.1.1 The Transmissibility of a Condition

To make a condition transmissible, the FRED program must declare a transmissibility property for the condition:

```
transmissibility = <value>
```

The value (0 by default) is the level of contagiousness for the condition and is used to determine the probability that an encounter between a transmissible agent and susceptible agent results in a transmission event. The higher the value, the greater the likelihood of contagion from one agent to another. A value of 1.0 corresponds to the transmissibility of a "standard" influenza model (R0 = 1.4). The Section on calibration discusses how to set transmissibility for other conditions if the R0 is known.

Currently, the only mode of transmission built into FRED is "proximity transmission", meaning that two agents must be in the same place at the same time for transmission to occur. Other modes of transmission, such as transmission via network contacts or transmission through environmental contact, require an explicit user-designed model.

Each transmissible condition must have a distinguished state called the **exposed state**. When an agent is exposed to a transmissible condition, the agent immediately goes into the state specified by the condition's `exposed_state` property:

```
exposed_state = <state_name>
```

For example:

```
condition INFLUENZA {
        exposed_state = Exposed
}
```

tells FRED that if an agent who is infectious with `INFLUENZA` transmits the disease to another agent, the second agent will enter the state called `Exposed`. This name is not required; any state may be designated as the `exposed_state`.

13.1.2 The Transmissibility of an Agent

Each agent has an individual level of transmissibility for each transmissible condition. An agent's transmissibility may change over time through an assignment statement:

```
<condition_name>.trans = <expression>
```

An agent's transmissibility usually changes, for example, when the agent moves from a non-infectious state of a transmissible disease to an infectious state.

Each agent also has an individual level of susceptibility to each transmissible condition, and this may change depending on the state of the agent, controlled by the statement:

```
<condition_name>.sus =  <expression>
```

When a transmissible agent with individual transmissibility `t` interacts with a susceptible agent with individual susceptibility `s`, the probability of an exposure is proportional to `T*t*s`, where `T` is the transmissibility of the condition. The probability also depends on other factors like the contact likelihood in the place where they meet and other factors described below.

13.1.3 Transmission Actions by the Meta Agent

The Meta agent can change the transmissibility of a condition by executing the statement:

```
<condition_name>.trans = <expression>
```

The result is that the transmissibility of the named condition is changed to the value of the expression. The change stays in effect until the Meta agent changes the condition's transmissibility again.

The Meta agent can cause a number of agents to be transmiited a condition by executing the statement:

```
import_exposures(<condition_name>, <expression>)
```

The first argument must be a name of a declared condition. The second argument is single-valued expression that is evaluated to determined the number of agent to be selected for exposure. The agents are selected **without replacement** from the population. The selected agents are exposed to the named condition with probability equal to their susceptibility.

- An error occurs if this action executed by non-meta agent.
- An error occurs if the number requested is greater than the population size.
- An error occurs if the number requested is negative.

For example, the Meta agent selects ten agents for possible exposure to INFLUENZA with the statement:

```
import_exposures(INFLUENZA, 10)
```

13.2 Transmission Models

There are two ways to model transmission in FRED:

- use the built-in transmission model
- design a custom transmission model

The built-in transmission model is a simple model that applies to proximity transmission in places. That is, it applies when a transmissible agent interacts with susceptible agents in a given place at a given time. The built-in transmission model is designed to model respiratory transmission of disease such as influenza. While it is modifiable through a number of parameters described below, it is not intended to cover every option for modeling transmission. For example, it does not include transmission via contacts with surfaces (that is, fomite transmission), or the affects of exact distances between agents within a place.

In order to activate the built-in transmission model for a given condition, the condition definition must include the property:

```
condition <condition_name> {
    transmission_model = proximity
}
```

The default value for this property is `transmission_model = none`. With this default setting, the FRED program may include a fully customized transmission model that takes advantage of any features of FRED including, for example, the exact distance between two agents or user-defined agent attributes such as an individual agent's viral load.

Custom models are required to model transmission modes such as transmission on networks (for example, sexually transmitted diseases or the spread of rumors). Custom models can also be designed to model vector transmission (for example, mosquito-borne diseases) or environmentally mediated conditions.

Examples of both the built-in transmission model and customized models are provided in the FRED Library.

13.2.1 Built-in Transmission in Places

This Section describes the built-in transmission model in FRED.

When transmissible and susceptible agents are present in the same place, a transmissible agent may transmit a condition to susceptible agent. Transmission is a stochastic event and depends on several factors including:

- the transmissibility of the condition
- the transmission mode for the place (by rate or by probability)
- the hourly contact rate or contact probability for the place
- the duration of the interactions in the place
- random selections of which agents interacts during a given meeting
- the transmissibility of the transmissible agent

- the susceptibility of the susceptible agent
- the similarity of ages if age biases are specific for the given place.

Each place type has a set of contact parameters that control proximity transmission between agents within the sites associated with the place type:

```
place <place_name> {
    contact_prob_for_condition_name> = 0
    contact_prob = 0
    contact_rate_for_<condition_name> = 0
    contact_rate = 0
    same_age_bias_for_<condition_name> = 0
    same_age_bias = 0
}
```

Potentially Transmissible Contacts

For each hourly step of the simulation and for each place that includes at least one transmissible agent, FRED determines the number of *potentially transmissible contacts*. That is, the number of contacts between a transmissible agent and a potentially susceptible agent.

For each type of place, the model can specify one of two options for computing potentially transmissible contacts: by rate or by probability.

Transmission by Contact Rate

Transmission by rate is the default contact mode for places in FRED. In this mode, each person is assumed to contact a specified number of other individuals during each hour that the place operates. In this mode, the total number of potential transmission contacts for a given condition in a place during its open hours on a given day is given by the formula:

```
contacts = (hourly contact rate of place) \* (number of transmissible
agents) \* (transmissibility of the condition) \* (duration of the
meeting in the place)
```

The hourly contact rate is, by default, the same for all places of a given place type and is set by the property:

```
contact_rate = <number>
contact_rate_for_<condition_name> = <number>
```

However, the group agents for individual places can change the contact rate using the `adjust_contacts()` function.

The first property specifies the contact rate per hour for all conditions. If a property of the second form occurs, it overrides the first property for the named condition. The default number of hourly contacts is 0.0. The following hourly contact properties are included in the `default_model` included in the default synthetic population:

```
place Household {
    contact_rate = 0.18
}

place School {
```

(continues on next page)

(continued from previous page)

```
    contact_rate = 0.20
}

place Grade {
    contact_rate = 0.40
}

place Workplace {
    contact_rate = 0.12
}

place Block_Group {
    contact_rate = 0.27
}

place Census_Tract {
    contact_rate = 0.27
}

place County {
    contact_rate = 0.27
}

place Prison {
    contact_rate = 0.12
}

place College_Dorm {
    contact_rate = 0.12
}

place Nursing_Home {
    contact_rate = 0.12
}

place Barracks {
    contact_rate = 0.12
}
```

Note: The above values were obtained through a calibration process to reflect the age bias observed in published contact matrices.

Transmission by Contact Probability

The second contact mode in places is **transmission by probability**. In this mode, each transmissible agent has a fixed probability of contacts with any other agent within any given hour that the place is in operation. In this mode, the total number of potential transmission contacts for a given condition in a place during its open hours on a given day is given by the formula:

```
contacts = (hourly contact probability of place) * (number of members
of the place -1) * (number of transmissible agents) *
(transmissibility of the condition) * (duration of the meeting in
the place)
```

To select the transmission by probability mode, the definition of the place type includes the following properties:

```
place PlaceType {
    contact_prob = <number>
    contact_prob_for_<condition_name> = <number>
}
```

The first property specifies the contact probability per hour for all conditions. If a property of the second form occurs, it overrides the first property for the named Condition.

For example,

```
place School {
    contact_prob_for_INF = 0.0001
}
```

The effect is to use transmission by probability for the `INF` condition with a probability = 0.0001 for contact occurring between any two individuals in a given hour of time in the school setting.

Selecting a Transmission Mode

The choice of appropriate transmission mode may vary according to the purpose of the model and by each condition. Transmission by contact rate is **independent of density**, and transmission by contact probability is **dependent on density**. For example, if a condition is transmitted through close contacts between individuals, then a density independent mode may be appropriate because an individual can only have a limited number of close contacts within a given time period. On the other hand, if a condition is freely transmitted to everyone an individual comes into slight contact with (for example, a highly contagious respiratory disease like measles), then a density-dependent transmission mode may be more appropriate because an infectious individual may transmit the condition to some fraction of the other individuals in a given crowded place.

Transmission from a Source to a Target

For each potential transmission contact event, FRED selects a transmissible agent at random, the **source**, and one other agent at random, the **target**, from the set of members of the place. The source and the target must be distinct agents.

The probability that the source successfully exposes the target is:

```
probability of transmission = (transmissibility of source) *
(susceptibility of target) * (same_age_bias)
```

Same-age Bias

The final term in the equation above is a factor the reflects the tendency of agents of similar age to interact with a given place and is controlled by the property:

```
place <place_id> {
    same_age_bias = <number>
}
```

There is also a simulation property that enables transmission in places to take age-bias into account, and it is enabled by default:

```
simulation {
    enable_transmission_bias = 1
}
```

If the place has a positive age-bias, the bias is computed by the formula:

```
same-age-bias(source, target) = exp(-(same_age_bias of place)*(|age_of_source -
↪ age_of_target|))
```

That is, the probability of transmission decreases with a larger difference in age between the interacting agents.

The following age-bias properties are included in the `default_model` included in the default synthetic population:

```
place Household {
    same_age_bias = 0.05
}

place Block_Group {
    same_age_bias = 0.1
}

place Census_Tract {
    same_age_bias = 0.1
}

place County {
    same_age_bias = 0.1
}
```

Note: The above values were selected to reflect the age bias observed in published contact matrices. The bias factors in community settings such as `Block_Group`, `Census_Tract`, and `County` model the tendency of people to interact with others closers to their own age.

Notes on Transmission in Places

There several important details in this process:

- Agents do not contact themselves in this process. Each contact is assumed to be between two distinct agents.
- The source and target agents may be selected more than once. That is, if an agent in a given place is assumed to have n contacts per day, it is not assumed that these contacts are necessarily with n distinct other agents.
- The target is selected among all current members of the place, not just those currently attending the place. In other words, it is possible that the target agent is absent when selected. The assumption here is that the number of contacts per agent per place is the nominal number of contacts if everyone is present. If many agents are absent (due to staying home sick, for example), then the number of contacts is effectively reduced proportionally.
- The selection of the target is independent of the susceptibility, that is, the target is not necessarily susceptible.

13.2.2 Custom Transmission in Places

FRED program may include a custom model of transmission in places for any condition.

```
condition <condition_name> {
   transmissibility = <value>
   transmission_model = none
   exposed_state = <state_name>
}
```

The value for `transmissibility` must be greater than `0`. The `transmission_model` is `none` by default so this line is optional. The `exposed_state` must be the name of a defined state within the condition. When agents are exposed to the condition, they will immediately transition to the exposed state.

In a custom place-based transmission model, the transmission between agents is controlled by the group agent associated with a place site. The group agent may transmit a condition to from a source agent to a destination agent using the action:

```
transmit(<condition_name>, <source_agent_id>, <destination_agent_id>)
```

The rules that the group agent uses to select the source agent and the destination agent are defined by the user model. For example, the group agent may or may not take into account the transmissibility of the source agent, the susceptibility of the destination agent, the ages of the agents, the position of the agents, or any other factors such as the agents' individual viral load, etc.

13.3 Transmission by the Meta Agent

The Meta agent may cause transmissions in either the built-in transmission model or custom transmission models, using the same action as described in the last section:

```
transmit(<condition_name>, <source_agent_id>, <destination_agent_id>)
```

To model the importation of an exposure (that is, the introduction of a case from an unknown source), the Meta agent can use itself as the source agent:

```
transmit(<condition_name>, id, <destination_agent_id>)
```

When the Meta agent transmits to a destination agent, the location of the exposure has the value `UNKNOWN`.

13.4 Other Ways to Model Transmission

Both the built-in transmission model and the `transmit()` action are provided for convenience, but it is not necessary to use either to model the transmission of a condition from one agent to another. In some cases, it may be more convenient for a model to use the more general action `send()`:

```
send(<agent_id>, <condition_name>, <state_name>)
```

This action may be executed by any agent and sends the indicated agent to the indicated state. In a fully customized transmission model, this action can be used to `transmit` the given state to any agent using whatever rules are appropriate to the given model.

CHAPTER

FOURTEEN

CHAPTER 14: MODULARITY

Many FRED models need similar components, so FRED supports several ways to re-use FRED code. This chapter address the use of `include` statements that includes other files of FRED code, and `use` statements that invoke FRED modules.

14.1 Include Files and FRED_PATH

One way to re-use FRED code is through the use of `include` statements. If a FRED program has a statement of the form:

```
include <filename.fred>
```

then the indicated file is inserted into the FRED program at that point.

FRED searches for the indicated file through the list of directories defined in the environmental variable `FRED_PATH`. Environmental variables are set in your .bashrc file or other shell startup script (.zshrc for new Mac users), for example:

```
export FRED_PATH=".:${HOME}/FRED_PROJECTS/project1:${HOME}/FRED_PROJECTS/
↪project2"
```

With the settings above, the search for an included file would first look in the current directory, followed by `${HOME}/FRED_PROJECTS/project1` and then `${HOME}/FRED_PROJECTS/project2`, where `$HOME` is the user's home directory. Continuing with the example settings above, if the FRED program contains the statement:

```
include sub_project_z/submodel.fred
```

then the file would be found if it is located at `${HOME}/FRED_PROJECTS/project2/sub_project_z/submodel.fred` and does not appear in any earlier directory in `FRED_PATH`.

The included filename may also include the relative path element `../`.

If the `FRED_PATH` environmental variable is not set, FRED searches for the file in the current working directory.

14.2 Modules

The FRED module construct takes file inclusion further by supporting parameters that replace parts of the included code. In programming terms, a module as a kind of macro whose expansion is inserted into the calling program.

A FRED module is included in a FRED program by a code block of the form:

```
use <module_id> {
    <parameter> = <value>
    <parameter> = <value>
    ...
    <parameter> = <value>
}
```

where each parameter is defined within the module and each value may be any non-empty string. The FRED code in the module is inserted into the FRED program at this point, but each instance of a parameter is replaced by its associated value. Parameters may appear anywhere with the module's code, even within other strings, so modules can be used in a wide range of situations, including partially rewriting the module's code.

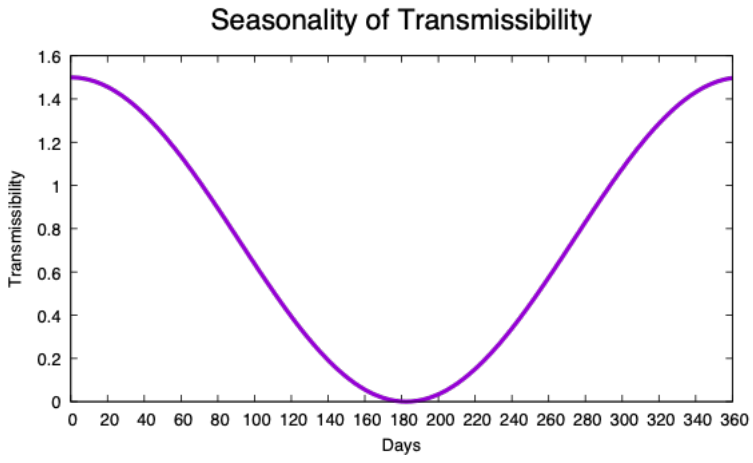
The `<module_id>` may include a path and may refer to the FRED environmental variables, just like included files. As special case is that if the `<module_id>` has the form `FRED::<module_name>` then the module is called from the FRED Library, as explained in the Section below.

For example, if a module with id `Influenza` is located in the file `$FRED_PROJECT/Influenza.fredmod` then it may be called with `use $FRED_PROJECT/Influenza`.

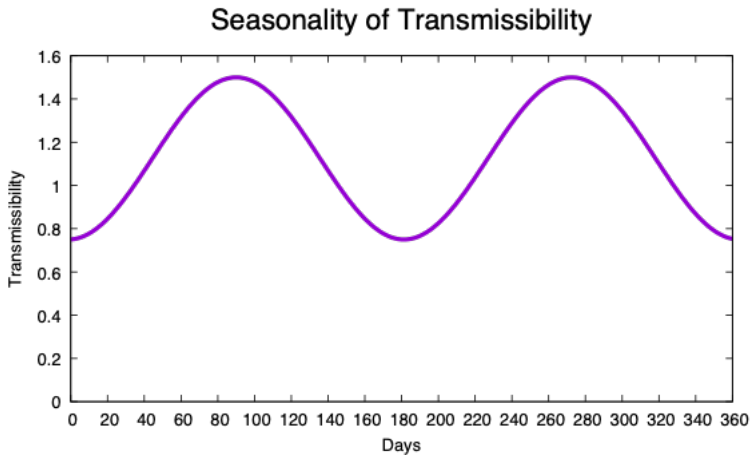
14.2.1 Example Module: Seasonality of Transmission

Before discussing how to create modules, let's consider a specific example. Many diseases have seasonality, meaning that the transmissibility varies according to the time of the year. Seasonality may be due to changes in temperature, humidity, human contacts patterns, or other causes. One way to model seasonality is to change the transmissibility of a condition by multiplying the default transmissibility by the current value of a function that cycles over the course of a year and whose apex corresponds to the time of year with peak transmissibility and whose nadir corresponds to the time of minimal transmissibility.

The following figure shows a seasonally adjusted transmissibility curve with a maximum transmissibility of 1.5 on January 1 of each year with a reduction of 100% occurring six months later.



For other conditions, different seasonal adjustments may be appropriate. The following figure shows a seasonally adjusted transmissibility curve with a maximum transmissibility of 1.5 in the Spring and Fall of each year with a reduction of 50% occurring in the Summer and Winter.



Rather than having to copy and edit the code for seasonality for each condition, we can create a single module called `Seasonal_Transmission` and then invoke that module for as many different conditions as we need, passing appropriate parameters to give different characteristics of seasonality for each condition.

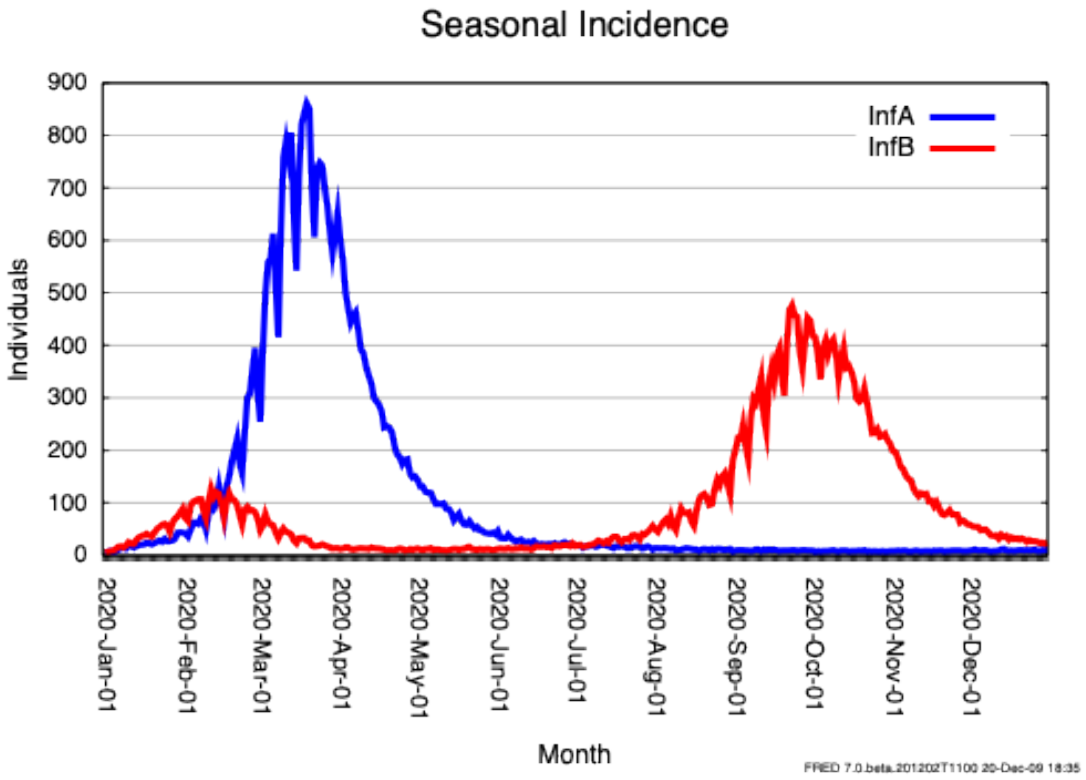
Let's see how this would work in a FRED program that contains two conditions, say for influenza A (`InfA`) and influenza B (`InfB`). Suppose we know that `InfA` has its peak about April 1 and has a 90% seasonal decline 6 months later, and that `InfB` has its peak about November 7 and has n 80% seasonal decline 6 months later.

The main program might contain the following fragments of code:

```
simulation {
    locations = Jefferson_County_PA
    start_date = 2020-Jan-01
    end_date = 2020-Dec-31
}

include InfA.fred
include InfB.fred

...

use FRED::Seasonal_Transmission {
    condition = InfA
    peak_day_of_year = 120
    seasonal_reduction = 0.9
    seasonal_period = 365
}


use FRED::Seasonal_Transmission {
    condition = InfB
    peak_day_of_year = 300
    seasonal_reduction = 0.8
    seasonal_period = 365
}
```

This program uses the `Seasonal_Transmission` module from the FRED library for each disease, passing in the appropriate control parameters.

Assuming that 5 cases of each disease are imported on each day of the simulation (using another module of the FRED Library discussed below), the resulting epidemic curves look like this:



The simulation starts on January 1 when the season for `InfB` is already past its peak, so only a small outbreak occurs before it fades out. The epidemic for `InfB` takes off again in the Fall, but its peak is lower the the curve for `InfA`, since there is already some immunity within the population from the Spring InfB outbreak.

14.3 Defining a FRED Module

A module is similar to an ordinary FRED component model containing one or more condition, with some specific differences:

- A module must be a separate file with the extension .fredmod
- If the module contains parameters, it must contain a prototype of the module call of the form:

```
prototype <module_id> {
    <parameter> = <value>
    <parameter> = <value>
    ...
    <parameter> = <value>
}
```

- Within the code of the module, each instance of a parameter in the prototype that is to be substituted for must be delimited (before and after) by the dollar sign symbol $.

When the module is called, the values supplied in the `use` block in the calling program are substituted for the delimited parameters within the module code.

If the calling program omits any parameter, then the value on the right-hand-side is used as a default value.

The calling program may include the parameters in any order.

After substitutions for the parameters, the result is included at that point in the FRED program.

14.3.1 Module Tags

Modules often construct one or more new conditions each time they are called. Modules usually use one or more of the parameters as a **module tags** to distinguish different invocations of the modules from one another.

For example, in the `Seasonal_Transmission` example discussed here, the prototype contains:

```
prototype Seasonal_Transmission {
    condition = CONDITION_TAG
    ...
}
```

where the `condition` parameter is a module tag that names the condition whose transmission should be affected by the invocation of the module. In the main program above, there are two calls to the modules, one with

```
condition = InfA
```

and one with

```
condition = InfB
```

Since the modules tags are different, a different set of FRED statements will be included in the main program for each call to the module.

14.3.2 Example of a Module

Here is an example of the `Seasonal_Transmission` module from the FRED Library. The module is contained in a file named Seasonal_Transmission.fredmod:

```
########################################
# Seasonal_Transmission Module
# Author: John Grefenstette, john.grefenstette@epistemix.com
# Date: 10 Jun 2020
########################################

prototype Seasonal_Transmission {
    condition = CONDITION_TAG
    peak_day_of_year = 1
    seasonal_reduction = 0
    seasonal_period = 365
}

variables {
```

(continues on next page)

(continued from previous page)

```
    # variable used to hold original transmissibility
    global Seasonal_Transmission_$condition$_max_trans

    global Seasonal_Transmission_$condition$_peak_day_of_year
    Seasonal_Transmission_$condition$_peak_day_of_year = $peak_day_of_year$

    global Seasonal_Transmission_$condition$_seasonal_reduction
    Seasonal_Transmission_$condition$_seasonal_reduction = $seasonal_reduction$

    global Seasonal_Transmission_$condition$_seasonal_period
    Seasonal_Transmission_$condition$_seasonal_period = $seasonal_period$
}

condition SEASONAL_TRANSMISSION_OF_$condition$ {
    start_state = Start
    meta_start_state = Adjust

    state Start {
          wait(0)
        next(Excluded)
    }

    state Adjust {

        if (sim_day == 0) then \
           Seasonal_Transmission_$condition$_max_trans = transmissibility_of_
↪$condition$

        $condition$.trans = Seasonal_Transmission_$condition$_max_trans * \
           (1.0 - Seasonal_Transmission_$condition$_seasonal_reduction * \
            0.5 * (1 + sin(2*3.14159  * \
           (abs(day_of_year - Seasonal_Transmission_$condition$_peak_day_of_
↪year) /
            Seasonal_Transmission_$condition$_seasonal_period + 0.75))))

        wait(24)
        next(Adjust)
    }
}
```

The module contains three components.

- The `prototype` block defines the parameters and their default values.
- The `variables` block defines a set of variables used in the module.
- The `condition` block defines a condition that will be included in the calling program.

Notice that the variable names and the condition name contain the module tag supplied in the prototype, so each time the module is called with a distinct tag, distinct variables and conditions will be created.

The details of the formula used to adjust the transmissibility need not concern us here, other than to note that the formula alters `$condition$.trans`, which will refer to the transmissibility of the condition named in the tag parameter.

When the module is called with `condition = InfA` in the example above, the parameters are replaced by the values specified, producing the code segment:

```
condition SEASONAL_TRANSMISSION_OF_InfA {
    start_state = Start
    meta_start_state = Adjust

    state Start {
          wait(0)
        next(Excluded)
    }

    state Adjust {

        if (sim_day==0) then \
       Seasonal_Transmission_InfA_max_trans = transmissibility_of_InfA

        InfA.trans = Seasonal_Transmission_InfA_max_trans * \
           (1.0 - Seasonal_Transmission_InfA_seasonal_reduction * \
            0.5 * (1 + sin(2*3.14159  * \
           (abs(day_of_year - Seasonal_Transmission_InfA_peak_day_of_year) /
            Seasonal_Transmission_InfA_seasonal_period + 0.75))))

        wait(24)
        next(Adjust)
    }
}
```

Notice that the name of the condition added by the module is `SEASONAL_TRANSMISSION_OF_InfA` as a result of the substitution of `InfA` for the string `$condition$` in the module file. The state `SEASONAL_TRANSMISSION_OF_InfA.Adjust` changes the transmissibility of the `InfA` condition. Likewise, the second invocation of the module will produce code for a condition called `SEASONAL_TRANSMISSION_OF_InfB` and adjust the transmissibility of the `InfB` condition.

CHAPTER

FIFTEEN

CHAPTER 15: SNAPSHOTS AND RESTARTS

Fred models can be run for a certain period of simulated time, and then be saved so that they can be continued later. The information required to do this is called a `snapshot`. The continuation of a simulation from a snapshot is called a `restart`.

15.1 Creating Snapshots

Snapshots are files that contains all the information needed to restart a FRED job from a given simulation date. Snapshots are only created at the end of a simulation day.

Snapshot files are large and they take significant time to make, so the user should carefully consider how frequently to take snapshots based on the specific needs of their workflow.

Snapshot files are compressed are not meant to be human-readable.

The following properties determines the number of snapshots for the job:

```
simulation {
   snapshots = <n>
   snapshot_final = <n>
   snapshot_date = <date_string>
   snapshot_interval = <n>
}
```

The number of snapshots for the job is determined by the following rules:

If the property `snapshot_final` is set, then a final snapshot is always generated.

If the property `snapshot_date` is set and the simulation reaches the specified simulation date, then the snapshot of that date is generated. It is not set by default. It is a string of the form `YYYY-MM-DD`, e.g., `2021-03-25`.

If the property `snapshot_intervals` has a value `n` greater than `0`, then snapshots will be generated after every `n` days. The final periodic snapshot is guaranteed to be kept.

If the `snapshots` property exceeds the number of snapshots required be the preceding rules, then additional periodic snapshots (if any) will be kept, until the total number of snapshots reaches the value of `snapshots`, at which point the oldest periodic snapshot will be deleted.

For example, the following property settings will keep the final two weekly snapshots and the final snapshot:

```
simulation {
    snapshot_final = 1
            snapshot_interval = 7
    snapshots = 3
    }
```

The following property settings will keep the final snapshot, the snapshot on the specifed date, and the final weekly snapshot:

```
simulation {
    snapshot_final = 1
    snapshot_date = 2022-05-22
            snapshot_interval = 7
    }
```

The following settings will produce a snapshot every 30 days. The last two will be kept.

```
simulation {
            snapshot_interval = 30
    snapshots = 2
    }
```

15.2 Fetching Snapshots

Each run in a FRED job produces its own snapshot, and all run-specific snapshots are rolled into a single snapshot for the job.

The FRED command:

```
$ fred_get_snapshot -k <key>
```

fetches the snapshot file for the job with the given key and copies it to `snapshot.tgz` in the current directory by default. To give the snapshot file another name using the `-o` option:

```
$ fred_get_snapshot -k <key> -o <output_file_name>
```

15.3 Restarts

To restart a job from a snapshot, using the `-R` option to `fred_job`:

```
$ fred_job -k <key> ... -R <snapshot_file_name>
```

The resulting job will be initialized using the given snapshot, and will set the simulation day to the day after the snapshot day. The `start_date` of the new job must be the same as the `start_date` of the snapshot job, and the `end_date` of the new job must be after the snapshot day.

15.3.1 Restart blocks

A model may contain a `restart_block` to reset variables after a restart.

```
restart {
    # actions to be performed by the Mate agent
}
```

The `restart` block is used to override shared variables after a restart. This block is executed after reading the snapshot files, so the actions in the block will override the settings of any shared variables defined in the snapshot.

It is an error to set a variable in a `restart` block that is not declared elsewhere in the program in a `variables` block.

For example, the following `restart` block overrides the `end_date` variable, and this extends the simuation to the new end date.

```
restart {
    end_date = 2022-Dec-31
}
```

15.3.2 Rules specific to a restart

A FRED model can contain rules that apply only during a restart by using the following predicate:

- `is_restart()` : returns 1 (true) if and only if it is executed during a FRED restart.

For example,

```
if (is_restart()) then intervention_policy = new_policy
```

CHAPTER

SIXTEEN

CHAPTER 16: TRANSITION GUIDE FOR FRED USERS

FRED 8 represents a major revision of FRED intended to make the FRED language easier to learn and to make the process of developing models in FRED more convenient for a wider range of tasks.

This Chapter highlights some of the changes that will be especially important to current users of FRED 7 as they convert existing models to FRED 8. The details for all these changes appear in the previous Chapters of this Guide as well as in the Reference pages.

16.1 FRED_LIBRARY environmental variable

The FRED Library has been moved from the core FRED distribution to a separate GitHub repository. Epistemix will use this public repository to make available a set of free FRED modules.

A new environmental variable `FRED_LIBRARY` is used to point to the user's local copy of the Library. If not defined, it defaults to a small Library within the FRED distribution. Users may add their own modules to their local Library without sharing it with other organizations.

16.2 Random Number Stream

FRED 8 uses the `Boost` random library to generate random number streams and random variates. This means that FRED gives consistent numerical results across all computer platforms and operating systems.

16.3 Admin Agents

In addition to ordinary agents and the Meta agent, agents who control specific mixing group are now called `group_agents` (instead of "admin agents"). See Chapter 2.

16.4 Declaring Variables

The FRED variable types have been renamed for consistency and clarity.

- `global` is now `shared numeric`
- `global_list` is now `shared list`
- `personal` is now `agent numeric`
- `personal_list` is now `agent list`

- `table` is now `shared table`
- `list_table` is now `shared list_table`

See Chapter 3.

16.5 Initialization

The `parameters` and `restart_parameters` blocks have been removed.

Any statements in the `startup` block are executed by the Meta agent prior to the start of the run. Any Meta agent action is permitted in the startup block, including initializing shared variables and opening output files.

Any statements in the `group_startup` block are executed by the group agents immediately after the `startup` block, if any.

Any statements in the `agent_startup` block are executed by the ordinary agents immediately after the `group_startup` block, if any.

Upon a restart of a snapshot, the `startup` block is skipped, and the statements in the `restart` block are executed by the Meta agent.

See Chapter 5.

16.6 Start State

The `Start` state has been eliminated as a default for each condition. The default for ordinary agents is `start_state = Excluded`, just like group agents and the Meta agent.

User are encouraged to explicitly include the start state property for each type of agent, for example:

```
condition COND {
    start_state = Start
    group_start_state = Excluded
    meta_start_state = Meta_Start
}
```

16.7 Factors

Factors have been replaced by read-only variables and several new function. See Chapter 6 for the current set of read-only variables.

16.8 Order of Agent Transition Events

For each time step during which any state transition occurs, the order in which agents are updated is:

- the Meta agent updates in all conditions
- the group agents update in all conditions
- the ordinary agents update in all conditions

This is a change from FRED 7 in which each condition was executed in order of (a) Meta agent, (b) group agents, (c) ordinary agents. in FRED 8, the Meta agent updates all of its conditions before any group or ordinary agent.

For example, if you want the Meta agent to expose some agents to a transmissible condition at time `0`, then the agents' susceptibility to that condition needs to be set in the `agent_startup` block, because the Meta agent will execute its actions at time `0` before any ordinary executes its start state.

16.9 Synthetic Population

The default synthetic population for FRED 8 is `US_2010.v5`. Some differences with FRED 7:

- the only built-in agent variable that is set for each ordinary agent is its `age`. All other variables in the synthetic population are optional and must be declared as agent variables to be included in the model.

For example, including the following declarations will read the value `sex` and `race` from the files in the synthetic population and add then as agent attributes,

```
variables {
    agent numeric sex
    agent numeric race
}
```

- the only built-in place variables are latitude, longitude, and elevation. All other variables in the synthetic population are optional and must be declared as group agent variables to be included in the model.

For example, including the following declarations will read the value `income` from the files in the synthetic population and add then as group agent attributes,

```
place Household {
    has_group_agent = 1
}

variables {
    agent numeric income
}
```

This says that each household has an associated group agent, and that agents may have an `income` variable. The synthetic population has an `income` field for each household, so this variable will be set for each household group agent. An ordinary agent can then access its household income via `ask(Household, income)`.

The household variable `income` is the only additional place attribute in the default synthetic population.

16.9.1 Bypassing the Synthetic Population

FRED program can bypass the default synthetic population entirely by setting the `locations = none` property in the `simulation` block. In this case, no agents or places are defined for the model. Agents and places can be added using the methods in the next section.

16.10 Adding Agents and Places

FRED programs may read file containing additional agents and additional places, using the function `read_agent_file(filename)` and `read_place_file(filename)`. These files contain comma-separated rows corresponding to one agent or one place per row. The files contain a header row that defines each field. Certain fields are predefined, as explained in Chapter 10. If these file contain an identifier in the header row that corresponds to a declared agent variable, then this field will be used to set the variables for each agent corresponding to each row.

With the availability of the functions, the `alternative population` feature is discontinued.

16.11 Default Model

Each synthetic population contains a file called `default_model.fred` that is automatically included in each FRED program by default. The default model model can be used to:

- define a set of places available in the synthetic population,
- define schedules for those places,
- cause agents to join places,
- or any other actions that are needed for the synthetic population to operate as intended.

Users can substitute their own default models by setting the property:

```
simulation {
    default_model = <filename>
}
```

The following causes the default model to be skipped:

```
simulation {
    default_model = none
}
```

If `default_model = none` but `locations` is not `none` then the FRED program will read in the agents in the synthetic population for the indicated location and will use only those places explicitly defined in the FRED program. The program will need to define schedules for all defined places as discussed in Chapter 9.

16.12 Printing

The FRED printing functions have been revised and expanding, including the option of printing message to the terminal. See Chapter 10.

16.13 Renamed Predicates

- `has_been_closed(place)` is now `is_temporarily_closed(place)`
- `date_range(date1, date2)` is now `is_date_in_range(date1, date2)`
- `is_admin(group)` is now `is_group_agent(group)`
- `is_absent(group)` is now `is_skipping(group)`

16.14 New Functions

The following functions have been added (Chapter 7)

- `get_contact_rate(group)`
- `get_container(container_type, place_type)`
- `get_group_agents(group)`
- `get_number_of_transmissibles(group)`
- `get_population()`
- `get_role()`
- `get_same_age_bias(group)`
- `get_size()`
- `get_transmissibility()`
- `read_agent_file(filename)`
- `read_group_file(filename)`
- `read_place_file(filename)`
- `unique(list-expression)`

16.15 Renamed Functions

The following functions have been renamed

- `admin_id(group)` is now `get_group_id(group)`
- `get_admin_agents(group)` is now `get_group_agents(group)`
- `neg_binomial()` is now `negative_binomial()`
- `state(condition)` is now `current_state(condition)`

16.16 Removed Functions

The following functions have been removed:

- `adi_state()`
- `adi_national()`
- `block_group()`
- `census_tract()`
- `county()`
- `filter()`
- `get_other()`
- `income()`
- `sample_cdf()`
- `value()`
- `zipcode()`

  The functions related to geographical regions have been replaced with the more general concept of a `container`. The synthetic population defines container relationships of places with their surrounding areas.

  For example to get the block group if an agent's household, the agent could do:

```
my_block_group = get_container(Block_Group, Household)
```

For information on compiling and executing FRED programs, please see the FRED Modeling Platform™ documentation. If you have questions about the company or are interested in speaking with our sales, recruiting, or other teams, please visit the Epistemix web site.