Notes and Extensions

- By convention, we call *states* that do not change in value during an execution *parameters*, although Ptolemy treats them the same.
- There are array states, string states, and string array states, as well as all the standard types.
- Calling Error::abortRun does not immediately stop a run. It just alerts the scheduler to stop at a convenient time.
- Other infrastructure classes include:
  - I/O stream classes
  - String classes
  - XGraph
  - BarGraph
  - List classes
  - Hash tables
  - Complex class
  - Fix class
  - Matrix classes
  - Random numbers
  - Histogram

Methods in a Star

Standard methods:

- **constructor: called when the star object is created**
- **setup: called before the scheduler is initialized**
- **begin: called after the scheduler is initialized, before the run**
- **go: called during the run**
- **wrapup: called at the completion of an *error-free* run**
- **destructor: called when the star object is destroyed**

You can also include in a star any other methods of your own design, plus arbitrary code segments in C or C++.

Type Propagation, MultiPortHoles, and Iterators

Fork star

```
defstar {
    name {Fork}
    domain {SDF}
    input {
        name {input}
        type {anytype}
    }
    outmulti {
        name {output}
        type {=input}
    }
    go {
        MPHIter iterator(output);
        PortHole* p;
        while ((p = iterator++) != 0) {
            (*p)%0 = input%0;
        }
    }
}
```

declares the star to be polymorphic

output will be a MultiPortHole object

note type propagation

iterators are a basic mechanism for stepping through lists. They have operator ++ that returns the next object on the list, and returns 0 when the list is finished.

Using Parameters

```
defstar {
    name { MyAverage }
    domain {SDF}
    desc { This star averages a batch of inputs }
    input { name{input} type{float} }
    output { name{output} type{float} }
    state {
        name{howmany}
        type{int}
        default{10}
    }
    setup {
        input.setSDFParams(int(howmany),int(howmany)-1);
    }
    go {
        double sum = 0.0;
        for (int i = 0; i < int(howmany); i++) {
            sum += double(input%i);
        }
        output%0 << sum;
    }
}
```

Consuming Multiple Samples

```
go {
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += double(input%i);
    }
    output%0 << sum;
}
```

the go method is completely unchanged, but:

```
setup {
    input.setSDFParams(10,9);
}
```

the first argument tells the domain how many tokens to discard after the star fires.

Accessing Past Samples

```
go {
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += double(input%i);
    }
    output%0 << sum;
}
```

accessing past samples

need to declare that this will be done:

```
setup {
    input.setSDFParams(1,9);
}
```

second argument tells the domain what the largest argument to the % operator will be. This allows buffers to be allocated statically and old Particles to be reclaimed.

More Sophisticated Go Method

```
go {
    double t = double(input%0);
    if (t <= 0) {
        Error::abortRun (*this, ": log of x, x <= 0");
        output%0 << -100.0;
    }
    else output%0 << log(t);
}
```

For most compilers, this explicit cast is not strictly necessary. Still it is safest to use it.

tells the error handler where the error occurred

One of many infrastructure classes provided to star writers.

Input and Output for Stars

```
go {
    output%0 << log (double(input%0));
}
```

Returns a reference to a Particle

Extracts a double from the Particle

Loads a double into the Particle referenced

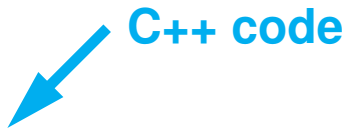
Manipulating Particles Directly (for Anytype)

```
go {
    Particle& current = input%0;
    outputA%0 = current;
    outputB%0 = current;
}
```

object PortHole has operator % defined to return a Particle&

Minimal Star Definition File

```
defstar {
    name { SimpleLog }
    domain {SDF}
    desc { This star computes the Log of the input }
    input {
        name{input}
        type{float}
    }
    output {
        name{output}
        type{float}
    }
    ccinclude { <math.h> }
    go {
        output%0 << log (double(input%0));
    }
}
```

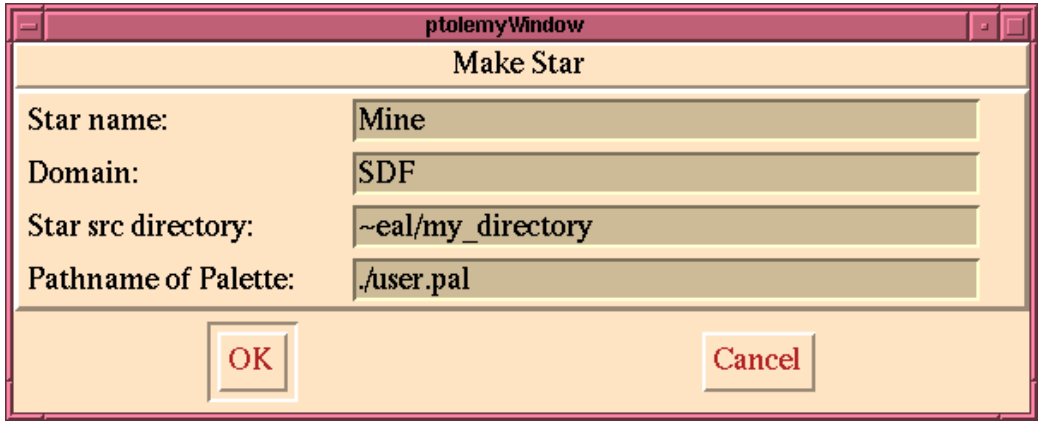


This should go in a file called "SDFSimpleLog.pl".

A preprocessor (ptlang) translates it into a C++ class.

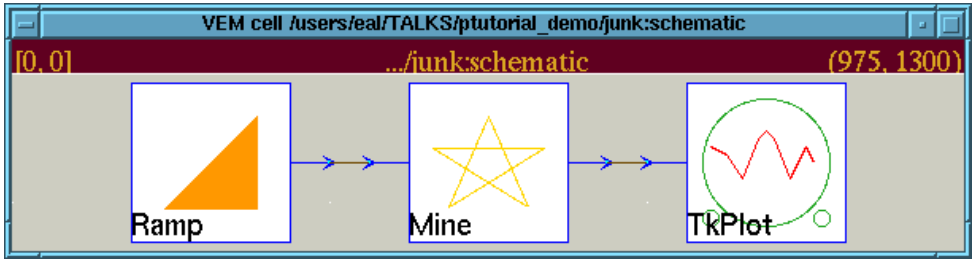
Using Custom Stars

- **Make-star (*)**



  - **If $PTOLEMY != /users/ptolemy, you have to set four (!) environment variables, or the compiler will not work. This is a flaw with the gnu compilers. See the programmer's manual.**

- **Make a test application**



Writing Custom Stars

The quick start:

- Find a star in the source tree.
- Copy it and modify it.

```
cd my_directory
cp $PTOLEMY/src/domains/sdf/stars/SDFSin.pl SDFMine.pl
chmod +w SDFMine.pl
```

- Change the name of the star!
- Replace the appropriate fields with your C++ code.

If your star is a modification of an existing star, consider deriving from the existing star.